JLP:
A LINEAR PROGRAMMING PACKAGE FOR MANAGEMENT PLANNING

Juha Lappi

Metsäntutkimuslaitoksen tiedonantoja 414

The Finnish Forest Research Institute. Research Papers 414

JLP: A LINEAR PROGRAMMING PACKAGE FOR MANAGEMENT PLANNING

Juha Lappi
The Finnish Forest Research Institute
Department of Forest Resources
Suonenjoki Research Station

Metsäntutkimuslaitoksen tiedonantoja 414
The Finnish Forest Research Institute. Research Papers 414

Suonenjoki 1992

Lappi, J. 1992. JLP: A linear programming package for management planning. Metsäntutkimuslaitoksen tiedonantoja 414. The Finnish Forest Research Institute. Research Papers 414. 134 p. ISBN 951-40-1218-6, ISSN 0358-4283.

JLP is a linear programming software package designed for management planning systems that generate several treatment schedules for management units and select optimal schedule combinations with linear programming. Constraints can be specified to domains, i.e., subsets of management units. The optimization algorithm is based on the generalized upper bound technique. The optimization algorithm can be accessed through a flexible command interface that includes data transformations. The program is written in portable FORTRAN 77, and it is currently running in Macintosh, VMS, UNIX, and OS/2 environments. User defined data input, transformations, report writer and interface routines can be linked to the program. The report includes i) a user's guide for the built-in interface, ii) a guide for installing the program into user's system, and iii) description of the mathematical basis of the optimization algorithm.

Keywords: linear programming, management planning

Author's address: Finnish Forest Research Institute, Suonenjoki Research Station, SF-77600 Suonenjoki, Finland.

Publisher: The Finnish Forest Research Institute; Project 3002-5. Accepted for publication by Prof. Erkki Tomppo in June 5, 1992.

Distributor: Suonenjoki Research Station, SF-77600 Suonenjoki, Finland.

ISBN 951-40-1218-6
ISSN 0358-4283

Suonenjoen kirjapaino Ky
Suonenjoki 1992

Contents

Preface

Ten years ago I applied linear programming for optimization problems where the MELA simulator generated treatment schedules for a number of treatment units. When doing some computing experiments with heuristic methods for solving iteratively small problems, I worked out how the computations could be done without heuristics so that the basis matrix is small and schedules are checked one after another. I began to implement the ideas, but then other problems intervened. In 1989, when Tuula Nuutinen needed a linear programming algorithm in her short term planning system, I returned to these ideas and promised to make her a nice little subroutine. It was soon found that I had reinvented the well known 'generalized upper bound technique'.

Initial goals for the software were very modest, but the project got out of the control as I followed the possibilities appearing during development process, or tried to meet the wishes of Tuula Nuutinen, or of Markku Siitonen who began to use the program in his MELA system. The first phase was an optimization subroutine for simple problems without interface properties. The second phase was to build an interface including transformations and possibilities to state constraints for domains. Third major phase was to include capabilities to handle general linear programming problems.

Installing the software to the GAYA-LP system of Hans Fredrik Hoen helped to make the code more portable.

Tuula Nuutinen and Markku Siitonen read this manuscript, corrected several errors and suggested improvements.

Suonenjoki June 3, 1992

Juha Lappi

1. INTRODUCTION

1.1 General

JLP is a linear programming software package designed to solve efficiently (fast and in a small computer memory) planning problems of the following type. The plan is made simultaneously for a number of treatment units (e.g. forest stands). A number of treatment schedules is derived for each treatment unit. Treatment units can also be called calculation units to indicate that they may result from grouping similar treatment units together. It is hereafter expressed that schedules are *simulated*, but JLP does not care how the treatment alternatives are generated. Each schedule is associated with a vector of input and output variables over time. For simplicity these variables will be called output variables. The decision maker is interested in the aggregated output variables, i.e., in the sums of variables over the treatment schedules. Treatment schedules can also be aggregated within some domains, i.e., in subsets of calculation units.

It is assumed that the goals of the decision maker can be described as a linear programming optimization problem. For instance, we may want to maximize net present value of future incomes, subject to constraints that the income level is nondecreasing in each subregion and the total volume after planning period is above a minimum level. For the general background for using linear programming in management planning see, e.g., Kilkki (1987) and Dykstra (1984). In this report, it is assumed that the reader is familiar with the basic properties of linear programming.

In addition to the aggregated output variables, the problem formulation may contain other variables whose values are determined in the optimization process. For instance, a goal programming problem (see, e.g., Steuer 1986) includes variables describing how much aggregated output variables deviate from target values, and the utility model of Lappi and Siitonen (1985) includes variables for consumption, savings and loans.

JLP is designed to be portable among different computers and planning systems. The package is planned to be distributed as FORTRAN 77 source code. It includes a general purpose precompiler JMAKE by which the user can easily tailor vectors and working areas according to the available memory and size of the problems (the user in charge of building an executable version of the program will be hereafter called 'system manager'). JMAKE can also be used to add some system dependent features to the programs. The package includes subroutine templates that the system manager can

modify for the special input and report generating tasks. The two main parts of the package are the interface part (subroutine *jlpin*) and the optimization part (subroutine *jlpopt*). To guarantee portability, the interface is built using simple command language that is interpreted with standard FORTRAN I/O functions. It is also possible to communicate with the interface using simple buffers. Thus a more sophisticated (and computer dependent) interface with menus and windows, etc. can be built on the provided interface. The system manager can easily access the solution also in binary form.

The program can read data using different formats, and the system manager can provide special input subroutines. These input routines can also simulate the schedules instead of reading them from files. The program can save the data in fast working files. If there is enough memory available, the whole data are stored in the memory. If there is not enough memory, a part of the data is stored in a working file. Thus also large problems can be solved with small memory. New variables can be created using transformations. It is possible that data contain only the physical variables, and, e.g., the cost and price variables are created during the optimization. The same transformation compiler is used for defining domains where different constraints should be fulfilled. The domains can overlap in any manner (i.e. there can be simultaneously domains for North and South, and poor and good sites).

1.2 Optimization Problem

Mathematically the optimization problems considered can be described as follows (more complete mathematical treatment is in Part 6). Let us first define a linear programming problem without assuming domains for constraints. An optimization problem can be presented as:

$$\text{Max or Min } z_0 = \sum_{k=1}^{p} a_{0k} x_k + \sum_{k=1}^{q} b_{0k} z_k \tag{1.1}$$

subject to the following constraints:

$$c_t \le \sum_{k=1}^{p} a_{tk} x_k + \sum_{k=1}^{q} b_{tk} z_k \le C_t, \quad t = 1, \ldots, r \tag{1.2}$$

$$x_k - \sum_{i=1}^{m} \sum_{j=1}^{n_i} x_k^{ij} w_{ij} = 0, \quad k = 1, \ldots, p \tag{1.3}$$

$$\sum_{j=1}^{n_i} w_{ij} = 1, \quad i = 1, \ldots, m \tag{1.4}$$

$$w_{ij} \geq 0 \quad \text{for all } i \text{ and } j \tag{1.5}$$

$$z_k \geq 0 \quad \text{for } k = 1,\ldots,q\,, \tag{1.6}$$

where

m = number of treatment units

n_i = number of management schedules for unit i

w_{ij} = the weight (proportion) of the treatment unit i managed according to management schedule j

x_k^{ij} = amount of item k produced or consumed by unit i if schedule j is applied

x_k = obtained amount of output variable k, k=1,...,p

z_k = an additional decision variable, k=1,...,q

a_{tk} = fixed real constants for t=1,...,r, k=1,...,p

b_{tk} = fixed real constants for t=1,...,r, k=1,...,q

r = number of utility constraints

The problem is solved by finding proper values for the unknown variables w_{ij}, x_k and z_k.

The constraints of form (1.2) are for the aggregated variables and other decision variables of which the decision maker is interested. These constraints will be called utility constraints. Term 'constraint' without qualifications refers later to the utility constraints. Constraints (1.3) define the aggregated output variables x_k as the sums over the calculation units. Coefficients x_k^{ij} are known constants produced by the simulation system. If the simulation system computes output quantities per area unit, then coefficients x_k^{ij} are obtained from these relative figures by multiplying with the area of the unit. The constraint (1.3) can be equivalently written as:

$$x_k = \sum_{i=1}^{m}\sum_{j=1}^{n_i} x_k^{ij} w_{ij}, \quad k = 1,\ldots,p \tag{1.7}$$

The less intuitive form is used in (1.3) in order to follow the linear programming convention that the right hand side is always a constant. Depending on the context, term *x-variable* refers either to an aggregated x_k -variable defined in (1.3) or in (1.7), or to constants x_k^{ij}.

Constraints (1.4) are so called area constraints saying that proportions of treatment schedules in a treatment unit need to sum up to one. A variable w_{ij} is called a *w-variable* or a *weight*.. A variable z_k is called a *z-variable*. *W*-variables and *z*-variables are *decision variables* by which we can fix a possible solution. Even if aggregated x_k variables are formally unknown variables of the optimization problem, their values can be trivially computed from Eq. (1.7) if the values of *w*-variables are known. *Z*-variables and (aggregated) *x*-variables are *utility variables* that determine how good the solution is. As described, e.g., by Kilkki (1987), all variables in a linear programming problem can be interpreted as variables in an implicit utility model. It is assumed in the above problem formulation that the identity of management units is preserved throughout the planning horizon. Thus the planning model can be classified as type *Model I* in the *Model I / Model II* terminology (see, e.g., Dykstra 1984)

The problem is a standard linear programming problem (some simple technical tricks may be needed depending on what is meant by 'standard'), and thus any linear programming software can be used to solve it.

A domain specific objective function or constraint can be defined in the above formulation by defining x_k^{ij} to be zero if unit *i* does not belong to the intended domain. The domain specifications are made explicit in the following formulation. Let D_t denote a subset of units (i.e. a subset of the set {1,...,*m*}) that are used on row *t*. Domains for different rows can be equal. Then a linear programming problem with domain specifications is:

$$\text{Max or Min} \quad z_0 = \sum_{k=1}^{p} a_{0k} x_{kD_0} + \sum_{k=1}^{q} b_{0k} z_k, \tag{1.8}$$

subject to:

$$c_t \le \sum_{k=1}^{p} a_{tk} x_{k D_t} + \sum_{k=1}^{q} b_{tk} z_k \le C_t, \quad t = 1,\ldots,r \tag{1.9}$$

$$x_{kD_t} - \sum_{i \in D_t} \sum_{j=1}^{n_i} x_k^{ij} w_{ij} = 0 \quad , \quad k = 1,\ldots,p, \qquad t = 1,\ldots,r \tag{1.10}$$

$$\sum_{j=1}^{n_i} w_{ij} = 1, \quad i = 1,\ldots,m \tag{1.11}$$

$$w_{ij} \ge 0 \quad \text{for all } i \text{ and } j \tag{1.12}$$

$$z_k \ge 0 \quad \text{for } k = 1,\ldots,q \tag{1.13}$$

It is thus assumed that aggregated output variables appearing in the same constraint are all for the same domain. *X*-variables from different domains can be included in the same constraint using additional *z*-variables, as will be described later. *Z*-variables are always assumed to be global. Variables x_{kD_t} will be called *domain variables* if it is emphasized that the summation is over a given domain.

A user of JLP needs only to define objective function (1.1) or (1.8) and the utility constraints (1.2) or (1.10), and JLP takes care of the other constraints utilizing the special structure of the problem.

1.3 Purpose of the Report

The purpose of this report is:

1) To be user's guide and reference manual for the JLP software when used '*as is*'.

2) To help to install the software into a larger planning system.

3) To help to understand how linear programming can be used in (forest) management planning problems, and how the results can be interpreted.

4) To give insight to mathematical structure of planning problems considered.

5) To make a break point in the development process of the software.

The main chapters of the report are called 'parts' in order to indicate that they can be largely read independently of each others. The main parts of the 'Parts' are called 'Chapters'. Because the report is intended to serve as a manual, a certain amount of repetition is intentional.

2. USER'S GUIDE

2.1 Overview

This part describes the standard interface of the JLP package. A system manager can add extra features or develop an own interface (see Part 4).

Some of the basic features of the standard JLP are:

Commands can be entered from the terminal or read from files using **`include`** command. A specific section of a file can be included. Included files can be nested. Included files can be listed without executing the commands.

Output can be written to output files. The amount of output can be controlled.

On-line help is available. The user or system manager can modify the help file.

With **`time`** command the user can measure the time of any section of the session.

Data can be read in from several files, using different formats or subroutines provided by the system manager.

Data can be stored in a special JLP format.

Variables are referred with variable names. New variables can be created with transformations. Transformed data can be written on the disk.

New schedules can be created by duplicating old schedules and modifying duplicates with transformations.

A treatment unit can be split so that different parts inherit different schedules from the original unit.

In addition to the *x*-variables that describe the simulated alternatives, JLP can utilize variables that describe data files (*d-variables*) and treatment units (*c-variables*). These variables can be used as parameters in transformations of *x*-variables, or they can be used to define domains for constraints.

Several RHS's can be defined in the same problem definition, and the alternative problems implied can be solved in a loop.

JLP can solve ordinary linear programming problems (i.e. without simulated data for treatment units).

Values of aggregated *x*-variables not included in the problem can be computed using weights provided by the solution.

JLP can compute the shadow prices of treatment units, shadow prices of *x*-variables, reduced costs of the nonbasic *z*-variables, and reduced costs for forcing a nonoptimal schedule into the solution. JLP can compute also the cost of forcing an *x*-variable to get a smaller or greater value than it had in the solution.

In this chapter only the basic features of different commands are described. For each topic, more details are given in the reference manual (Part 3).

2.2 Command Syntax

All commands need to be in lower case. A command line can contain spaces and tabs. If the last character of a line is '>', then the logical command line continues to the next physical line (record). Commands can be read from terminal (or input stream in batch mode) or from files using **`include`** command. Command lines starting with '*', '!', or ';' are comments, and the rest of line following '!' is also a comment. Continuation character '>' is significant also after '!', the next line is regarded as the continuation of the comment.

pIf '>' is the last character of a command line, then the next line is **not** interpreted as a continuation line if '!' is put after the command. This is important to remember when using **`path`** command in systems where directory can be given as: '<*directory*>'. For instance, the following command works as intended:

```
path disk2:<mela.data> !
```

Names of commands are checked as long as the name is uniquely determined (usually four characters are significant). The rest of the command name is ignored. In this manual, a longer form of a command name may be used when the command is introduced. A shorter form is used thereafter.

Commands can have options starting with '/'. Options are appended to the command name without space. In options (e.g. '`/all`') only the first character is usually significant, except in negation options (e.g. '`/nocost`') three characters are significant. If more characters are significant, it is always indicated. If a command has several options, the order of options is free.

Examples:

The following input lines are equivalent (in **`schedules`** command four characters are significant):

```
schedules/all     20   ! comment
     sche/a     20
schepeteus/argum#entr  >
   20
```

A group of commands belonging together is called a *paragraph*. Paragraphs end always with '/'. For instance the following paragraph defines a linear programming problem:

```
problem
x1=0
x2 max
/
```

2.3 Examples

JLP solved the following examples in Macintosh Quadra 700. JLP was compiled with Language Systems FORTRAN 3.0 compiler. Data were simulated with MELA system (see Siitonen 1983). MELA specific features are not used in examples. Management schedules were simulated for five ten-year periods. Management operations are assumed to take place at midpoints of periods. The following variables were taken from MELA files into standard sequential files:

`vol.0,-vol.5`	=	total volumes, initial and after each period
`npv.0,-npv.5`	=	net present values
`cutvol.1,-cutvol.5`	=	annual volume harvested in each period
`clearcut.1,-clearcut.5`	=	annual clear cut areas
`income.1,-income.5`	=	annual net incomes

When JLP is started, it prompts

```
jlp>
```

and waits for commands from terminal (or input stream). The commands for the following examples are stored in file *ex.in*. File *ex.in* contains a section:

```
*ex1
...commands
*end ex1
```

This section can be submitted using **`include`** command.

2.3.1 A problem with *x*-variables: nondecreasing incomes

```
jlp>incl ex.in/*ex1:*
> *ex1
> xdat savo.xdb    ! file containing x-data (simulated schedules)
> xform b          ! x-variables are in binary file
> xvar vol.0,-vol.5,npv.0,-npv.5,cutvol.1,-cutvol.5,clearcut.1,-clearcut.5,>
>  income.1,-income.5  !variables in x-data
> cdat savo.cda    ! text file containing c-data
> cvar ns       ! c-data must always contain number_of_schedules_variable 'ns'
> cform *       ! c-data can be read using FORTRAN free format '*'
> time          ! start timing
starting timing..
> prob
 . . . . . . . . . . . . . . .needs to read data ...
reading xdat-file: savo.xdb
reading cdat-file: savo.cda
```

```
number of calculation units, schedules: . . . . .     433  12100
number of variables in xmat-matrix, max of ns  . .     27    181
memory used by xmat, units written to disk . . . .     37%     0
x-variables: vol.0,vol.1,vol.2,vol.3,vol.4,vol.5,npv.0,npv.1,npv.2,npv.3,
npv.4,npv.5,cutvol.1,cutvol.2,cutvol.3,cutvol.4,cutvol.5,clearcut.1,
clearcut.2,clearcut.3,clearcut.4,clearcut.5,income.1,income.2,income.3,
income.4,income.5
number of variables in cmat-matrix: . . . . . .       1
memory used by cmat  . . . . . .     44%
c-variables:ns
number of rejected schedules:  . . . . . .      0
 . . . . . data ready.
> income.2-income.1>0
> income.3-income.2>0
> income.4-income.3>0
> income.5-income.4>0
> npv.0 max
> /
number of domains, domain combinations: . . . . .       1      1
number of z-variables, temporary x-variables . . .      0      4

domain:                                          # of units
row    tolerance       min             max

all:                                                  433
   1  0.4911442E-01  -1018118.       2126654.
   2  0.7530341E-01  -1961494.       3260638.
   3  0.1083217      -3036788.       4690328.
   4  0.1455694      -4459802.       6303157.
   5  0.7732226      0.2381406E+08  0.3348054E+08
> time    ! print time since last time command
elapsed: 32.71655 total: 32.71655
> solve   ! solve the problem defined in problem paragraph
starting optimization...
ok(0)  constr.   2:  80632.089    w+z basics:   0   0
ok(0)  constr.   3:  98079.493    w+z basics:   0   0
ok(0)  constr.   4:  684918.26    w+z basics:   0   0
ok(2)  constr.   1:  664.13396    w+z basics:   0   0
**FEASIBLE
**OBJECT VARIABLE:   30759195.    w+z basics:   0   0
unit=    1, OBJ VAR= 32697588.    w+z basics:   1   0
unit=    1, OBJ VAR= 33225259.    w+z basics:   3   0
...
unit=    1, OBJ VAR= 33455594.    w+z basics:   4   0
unit=    1, OBJ VAR= 33458323.    w+z basics:   4   0
unit=    1, OBJ VAR= 33458704.    w+z basics:   4   0
unit=    1, OBJ VAR= 33459040.    w+z basics:   4   0
unit=    1, OBJ VAR= 33459073.    w+z basics:   4   0
**SOLUTION, OBJ VAR= 33459073.    w+z basics:   4   0 unit=  332
*s* solution,      optimization time ...11.14990
time for computing x-variables:   20.35009

DOMAIN all:                                                  433 units

row                                    value       shadow     lower     upper
                                                   price      bound     bound

  1) income.2-income.1  . . . . .  0.00000000 -0.2052712   0.000000         L
  2) income.3-income.2  . . . . .  0.00000000 -0.2147269   0.000000         L
  3) income.4-income.3  . . . . .  0.00000000 -0.0895401   0.000000         L
  4) income.5-income.4  . . . . .  0.00000000 -0.0410396   0.000000         L
  5) npv.0 . . . . . 33459072.9 1.00000000                        max

     x-variable                        value       shadow    cost of   cost of
                                                   price     decrease  increase

     vol.0  . . . . . .  146895.466              INF       INF
     vol.1  . . . . . .  160940.368          0.00575244 0.09252250
     vol.2  . . . . . .  168948.219          0.00345704 0.00000000
     vol.3  . . . . . .  176632.726          0.00433317 0.00000000
     vol.4  . . . . . .  201095.434          0.00000000 0.02377209
     vol.5  . . . . . .  247916.318          0.00391562 0.00000000
```

```
     npv.0  . . . . . .  33459072.9 1.00000000 1.00000000    INF
     npv.1  . . . . . .  35829849.0            0.00017427 0.00000000
     npv.2  . . . . . .  39015973.1            0.00007436 0.00000000
     npv.3  . . . . . .  43297856.3            0.00004199 0.00000000
     npv.4  . . . . . .  49052348.2            0.00002650 0.00000000
     npv.5  . . . . . .  56785902.2            0.00001771 0.00000000
     cutvol.1  . . . . . . 5514.61876          1.45063281 0.09019085
     cutvol.2  . . . . . . 5894.09835          0.80362908 0.31202058
     cutvol.3  . . . . . . 6087.74131          0.08545117 0.61943881
     cutvol.4  . . . . . . 5926.31498          0.41053064 0.02250821
     cutvol.5  . . . . . . 5340.53036          0.00000000 1.26948442
     clearcut.1  . . . . . . 18.3550669        1131.97075 191.506863
     clearcut.2  . . . . . . 8.74457034           INF     10068.0952
     clearcut.3  . . . . . . 3.80288005        228.412333 1507.36195
     clearcut.4  . . . . . . 7.36293925        88.5195472 10242.1571
     clearcut.5  . . . . . . 7.71705472        212.978217 1084.61137
     income.1  . . . . . . 788108.446 -0.2052712 0.00000000 0.00202079
     income.2  . . . . . . 788108.443 -0.0094556 0.00000000 0.00202079
     income.3  . . . . . . 788108.441 0.12518680 0.00000000 0.00202079
     income.4  . . . . . . 788108.439 0.04850047 0.00000000 0.00202079
     income.5  . . . . . . 788108.439 0.04103964 0.00000000 0.00202079
-------------------------------------------------------------------------
> *end ex1
jlp>
```

A '>' character at the beginning of a line indicates that the line is read from the included file. Commands **`xdata`**, **`xform`** and **`xvar`** define the treatment schedule information for JLP. Commands **`cdata`**, **`cform`** and **`cvar`** define information about the calculation units. JLP needs to know at least how many schedules there are in each treatment unit (variable *`ns`*). When the **`problem`** paragraph starts, JLP reads data into the memory. Thus the data definitions need to be before **`problem`** command but the order of definitions does not matter. After reading the problem definition, JLP computes the smallest and largest possible values of the aggregated output variables (x-variables). These bounds will rule out some problems as infeasible immediately. These bounds are also used for determining tolerance values of round-off errors.

In the example, all calculation units belonged to the same domain (*`all:`*). Command **`solve`** asks JLP to solve the problem. The x-variable section of the output is computed after the solution is obtained. These computations took more time than the optimization as such (times are in seconds). A part or all of these after-solution computations can be avoided. The *shadow prices* of *x*-variables `income.1,-income.5` are the shadow prices of constraints (1.3) defining the *x*-variables. They tell how the objective function would change if the problem remains the same and we get an extra unit of the *x*-variable from another source. The *cost of decrease* and *cost of increase* are the marginal changes in the objective function if we would add a new constraint that would require the corresponding *x*-variable to decrease or increase by one unit from the value implied by the solution. The output and interpretation of shadow prices are later described in detail.

2.3.2 A problem with *x*- and *z*-variables: goal programming

The next example includes also *z*-variables, i.e., technical variables needed in some linear programming problems. Suppose that we would like to have such a management plan that variables `income.1,-income.5` would have values 800, 850, 900, 1000 and 1100 thousands, and variable `npv.5` would have value 50 mill. However, a problem with these constraints is infeasible. We might then search for a plan that minimizes the sum of differences between income variables and the target values. As all *z*-variables are nonnegative, we need to define deviations from target values using slack and surplus variables. Such a goal programming problem definition is stored in file *ex.in* starting with an label '`*ex2`'. Thus our session might continue as follows (part of the output is deleted).

```
jlp>incl ex.in/*ex2:*
> *ex2
> prob
> income.1 -sp.1 + sl.1  =  800000 / = 850000 ! The values after '/' define
> ;                                             alternative RHS's
> income.2 -sp.2 + sl.2  =  850000 / = 900000
> income.3 -sp.3 + sl.3  =  900000 / = 1000000
> income.4 -sp.4 + sl.4  = 1000000 / = 1000000
> income.5 -sp.5 + sl.5  = 1100000 / = 1100000
> npv.5 > 50000000
> sp.1 + sl.1 + sp.2 + sl.2 + sp.3 + sl.3 + sp.4 + sl.4 + sp.5 + sl.5  min
> /
number of domains, domain combinations: . . . . .       1      1
number of z-variables, temporary x-variables . . .     10      0
_____________________________________________________________________________
domain:                                             # of units
row    tolerance       min             max
_____________________________________________________________________________
all:                                                      433
   1  0.2798653E-01  -2704.495       1211817.
   2  0.4927205E-01   4919.161       2133480.
   3  0.7595614E-01  -26151.70       3288901.
   4  0.1088024      -35820.53       4711143.
   5  0.1466871      -42605.84       6351553.
   6   2.487337      0.2907604E+08  0.1077017E+09
   7  0.2798653E-01
> *end ex2
jlp>solve
starting optimization...
ok(0)  constr.   6:  57299366.    w+z basics:   0   0
ok(3)  constr.   1:  800000.00    w+z basics:   0   1
ok(3)  constr.   2:  850000.00    w+z basics:   0   2
ok(3)  constr.   3:  900000.00    w+z basics:   0   3
ok(3)  constr.   4: 1000000.00    w+z basics:   0   4
ok(3)  constr.   5:  1100000.0    w+z basics:   0   5
**FEASIBLE
**OBJECT VARIABLE:   744657.25    w+z basics:   0   5
unit=    1, OBJ VAR= 741184.00    w+z basics:   0   5
unit=    1, OBJ VAR= 213000.08    w+z basics:   4   2
...
unit=    1, OBJ VAR= 137757.85    w+z basics:   5   1
unit=    1, OBJ VAR= 137750.84    w+z basics:   5   1
unit=    1, OBJ VAR= 137750.25    w+z basics:   5   1
**SOLUTION, OBJ VAR= 137750.25    w+z basics:   5   1 unit=  146
*s* solution,      optimization time ...24.41650
time for computing x-variables:   27.23339
_____________________________________________________________________________
DOMAIN all:                                                       433 units
_____________________________________________________________________________
row                                  value       shadow     lower     upper
```

```
                                                    price      bound    bound
-------------------------------------------------------------------------------
  1) income.1-sp.1+sl.1  . . . . . 800000.000 -1.0000000        800000.0     U
  2) income.2-sp.2+sl.2  . . . . . 850000.000 -0.7715559        850000.0     U
  3) income.3-sp.3+sl.3  . . . . . 900000.000 -0.5931115        900000.0     U
  4) income.4-sp.4+sl.4  . . . . . 1000000.00 -0.4444153        1000000.     U
  5) income.5-sp.5+sl.5  . . . . . 1100000.00 -0.3276779        1100000.     U
  6) npv.5 . . . . . . . . . . . . 50000000.0 -0.0260183   50000000          U
  7) sp.1+sl.1+sp.2+sl.2+
     sp.3+sl.3+sp.4+sl.4+sp.5+sl.5 137750.254 1.00000000          min
-------------------------------------------------------------------------------
     x-variable                      value       shadow    cost of   cost of
                                             price     decrease  increase
-------------------------------------------------------------------------------
     vol.0  . . . . . .  146895.466              INF       INF
     vol.1  . . . . . .  172770.584          0.00000000 0.00000000
     vol.2  . . . . . .  181429.917          0.00000000 0.00000000
     vol.3  . . . . . .  182321.570          0.00000000 0.00000000
     vol.4  . . . . . .  186760.039          0.00000000 0.00000000
     vol.5  . . . . . .  196640.038          0.00000000 0.00000000
     npv.0  . . . . . .  33334940.3          0.00000000 0.96435642
     npv.1  . . . . . .  37122072.3          0.00000000 0.00000000
     npv.2  . . . . . .  40035120.1          0.00000000 0.00000000
     npv.3  . . . . . .  43370374.6          0.00000000 0.00000000
     npv.4  . . . . . .  46693402.9          0.00000000 0.00000000
     npv.5  . . . . . .  50000000.0 0.02601837    INF       INF
     cutvol.1  . . . . . .  4547.23514        0.00000000 0.00000000
     cutvol.2  . . . . . .  6167.46797        0.00000000 0.00000000
     cutvol.3  . . . . . .  6858.98612        0.00000000 0.00000000
     cutvol.4  . . . . . .  7328.18340        0.00000000 0.00000000
     cutvol.5  . . . . . .  7693.73133        0.00000000 0.00000000
     clearcut.1  . . . . . .  13.8998666      0.00000000 0.00000000
     clearcut.2  . . . . . .  10.0533873      0.00000000 0.00000000
     clearcut.3  . . . . . .  6.55008829      0.00000000 0.00000000
     clearcut.4  . . . . . .  8.95997921      0.00000000 0.00000000
     clearcut.5  . . . . . .  14.4669117      0.00000000 0.00000000
     income.1  . . . . . .  662249.745 1.00000000 0.00000000 0.29608234
     income.2  . . . . . .  849999.999 0.77155591 0.22844408 1.77155588
     income.3  . . . . . .  900000.000 0.59311153 0.40688847 1.59311154
     income.4  . . . . . .  1000000.00 0.44441532 0.55558468 1.44441535
     income.5  . . . . . .  1099999.99 0.32767794 0.67232208 1.32767801
-------------------------------------------------------------------------------
z-variable              value      reduced cost
-------------------------------------------------------------------------------
sp.1  . . . . . 0.00000000   0.00000000
sl.1  . . . . . 137750.254   0.00000000
sp.2  . . . . . 0.00000000   0.22844408
sl.2  . . . . . 0.00000000   1.77155591
sp.3  . . . . . 0.00000000   0.40688846
sl.3  . . . . . 0.00000000   1.59311153
sp.4  . . . . . 0.00000000   0.55558467
sl.4  . . . . . 0.00000000   1.44441532
sp.5  . . . . . 0.00000000   0.67232205
sl.5  . . . . . 0.00000000   1.32767794
jlp>
```

We see that when deviations have the same weight for each period, only income during the first period deviates from the target. Output '`w+z basics:  5  1`' tells that there are 5 basic weight variables and one basic *z*-variable in the solution. JLP computes for each *z*-variable the reduced cost that tells the marginal price of forcing the variable to the solution. For a basic (nonzero) *z*-variable, the reduced cost is zero.

The problem with the second set of RHS's defined in the **`problem`** paragraph could be then solved with:

```
jlp>solve 2
```

Both *x*- and *z*-variables are also needed to solve, e.g., the utility model of Lappi and Siitonen (1985) which provides an alternative linear programming problem formulation for studying smooth income patterns.

2.3.3 A problem with *z*-variables: an ordinary LP problem

The third example shows that JLP can solve also ordinary linear programming problems (i.e. without simulated treatment schedules):

```
jlp>incl ex.in/*lu52:*
> *lu52 This example is from Luenberger (1973) p. 52
> prob
> 2*x1 +x2 +3*x3-2*x4+10*x5 min ! x1, -x5 are here z-variables, because they
;                                were not defined in xvar or xtran
> x1+x3-x4+2*x5=5
> x2+2*x3+2*x4+x5=9
> x1<7
> x2<10
> x3<1
> x4<5
> x5<3
> /
no x-variables, number of z-variables  . . . . . .      5
tolerance for all rows: 0.00010000
> *end lu52
jlp>solve
starting optimization...
ok(0)  constr.   4: 0.00000000    w+z basics:   0   0
ok(0)  constr.   5: 0.00000000    w+z basics:   0   0
ok(0)  constr.   6: 0.00000000    w+z basics:   0   0
ok(0)  constr.   7: 0.00000000    w+z basics:   0   0
ok(0)  constr.   8: 0.00000000    w+z basics:   0   0
ok(3)  constr.   2:  5.0000000    w+z basics:   0   1
ok(3)  constr.   3:  9.0000000    w+z basics:   0   3
**FEASIBLE
**OBJECT VARIABLE:   29.000000    w+z basics:   0   3
unit=    1, OBJ VAR= 12.000000    w+z basics:   0   4
unit=    1, OBJ VAR= 12.000000    w+z basics:   0   4
**SOLUTION, OBJ VAR= 12.000000    w+z basics:   0   4 unit=    1
*s* solution,      optimization time ...0.250000
```

row	value	shadow price	lower bound	upper bound	
1) 2*x1+x2+3*x3-2*x4+10*x5 . .	12.0000000	1.00000000		min	
2) x1+x3-x4+2*x5	5.00000000	4.00000000	5.000000		L
3) x2+2*x3+2*x4+x5	9.00000000	1.00000000	9.000000		L
4) x1	7.00000000	-2.0000000		7.000000	U
5) x2	1.00000000	0.00000000		10.00000	
6) x3	1.00000000	-3.0000000		1.000000	U
7) x4	3.00000000	0.00000000		5.000000	
8) x5	0.00000000	0.00000000		3.000000	

z-variable	value	reduced cost
x1	7.00000000	0.00000000
x2	1.00000000	0.00000000
x3	1.00000000	0.00000000
x4	3.00000000	0.00000000
x5	0.00000000	1.00000000

```
jlp>
```

Note that the term `unit` appearing in the printing of the optimization algorithm does not mean anything. JLP interprets the variables `x1,-x5` as *z*-variables because they

were not defined in a previous **xvar** command or in **xtran** transformations (transformations creating new *x*-variables). JLP is not efficient in solving ordinary linear programming problems, but in small problems that may be of less interest.

2.3.4 A problem with several data files and domains

The following problem uses following properties of JLP: data can be read from several files, there can be transformations of variables, symbolic names for constants can be defined, constraints can be defined for different domains (subsets of units), results can be printed for additional printing domains (most part of printing is deleted):

```
jlp>incl ex.in/*exd:*
> *exd   ! example including domains
> xdat savo.xdb,vaasa.xdb  ! two data sets
> xform b
> xvar vol.0,-vol.5,npv.0,-npv.5,cutvol.1,-cutvol.5,clearcut.1,-clearcut.5,>
>  income.1,-income.5
> cdat savo.cda,vaasa.cda   !  c-data
> cvar ns
> cform *
> ctran ! data do not contain c-variables, let us make an artificial 'owner'
> owner=unit-2*int(unit/2) ! owner = 0,1
> /
> const private,public=1,0 ! make symbolic names for owner groups
> prob
 . . . . . .needs to read data ...
reading xdat-file: savo.xdb
reading cdat-file: savo.cda
number of calculation units, schedules: . . . . .     433  12100
reading xdat-file: vaasa.xdb
reading cdat-file: vaasa.cda
number of calculation units, schedules: . . . . .     406  12872
total number of calculation units, schedules: . .     839  24972
number of variables in xmat-matrix, max of ns  . .     27    200
memory used by xmat, units written to disk . . . .     75%     0
x-variables: vol.0,vol.1,vol.2,vol.3,vol.4,vol.5,npv.0,npv.1,npv.2,npv.3,
npv.4,npv.5,cutvol.1,cutvol.2,cutvol.3,cutvol.4,cutvol.5,clearcut.1,
clearcut.2,clearcut.3,clearcut.4,clearcut.5,income.1,income.2,income.3,
income.4,income.5
number of variables in cmat-matrix: . . . . . .       2
memory used by cmat  . . . . . .     84%
c-variables:ns,owner
number of rejected schedules:  . . . . . .      0
 . . . . . data ready.
> data=savo.and.owner=private: data=vaasa.and.owner=private: owner=public:
> income.2-income.1>0
> income.3-income.2>0
> income.4-income.3>0
> income.5-income.4>0
> all:
> npv.0 max
> /
number of domains, domain combinations: . . . . .       4      3
number of z-variables, temporary x-variables . . .      0      4
_____________________________________________________________________________
domain:                                          # of units
row    tolerance       min             max
_____________________________________________________________________________
data=savo.and.owner=private:                              217
   1  0.4705846E-01  -525772.3       1021169.
...
data=vaasa.and.owner=private:                             203
   5  0.1653200E-01  -173746.1       335599.5
...
owner=public:                                             419
```

```
   9  0.3611564E-01  -723164.0       1513245.
...
all:                                                    839
  13  0.5274830      0.3129301E+08  0.4425582E+08
> ! print results for some x-variables only (default is all variables)
> ! variables npv.1,-npv.4 are not printed
> ! we may get results for domains not used in problem
> show/domain  vol.0,-vol.5,npv.5,cutvol.1,-cutvol.5,clearcut.1,-clearcut.5
> data=savo:
> data=vaasa:
> /
> solve   ! solve the problem defined in problem paragraph
starting optimization...
ok(0)  constr.   2:  46468.188    w+z basics:   0   0
...
ok(2)  constr.   5:  2228.2402    w+z basics:   1   0
**FEASIBLE
**OBJECT VARIABLE:   40315249.    w+z basics:   1   0
unit=    1, OBJ VAR= 41193914.    w+z basics:   4   0
...
unit=    1, OBJ VAR= 44209549.    w+z basics:  12   0
**SOLUTION, OBJ VAR= 44209549.    w+z basics:  12   0 unit=   69
*s* solution,      optimization time ...30.09960
time for computing x-variables:   195.7001
```

DOMAIN data=savo.and.owner=private: 217 units

row	value	shadow price	lower bound	upper bound	
1) income.2-income.1	0.00000000	-0.1056677	0.000000		L
2) income.3-income.2	0.00000000	-0.1513036	0.000000		L
3) income.4-income.3	0.00000000	-0.0753490	0.000000		L
4) income.5-income.4	0.00000000	-0.0418552	0.000000		L

x-variable	value	shadow price	cost of decrease	cost of increase
npv.0	16503894.6		1.00000000	INF
vol.0	72450.4998		INF	INF

...

DOMAIN data=vaasa.and.owner=private: 203 units

row	value	shadow price	lower bound	upper bound	
5) income.2-income.1	0.00000000	-0.3187801	0.000000		L
6) income.3-income.2	0.00000000	-0.3297849	0.000000		L
7) income.4-income.3	0.00000000	-0.1762447	0.000000		L
8) income.5-income.4	0.00000000	-0.0971891	0.000000		L

x-variable	value	shadow price	cost of decrease	cost of increase
npv.0	5152362.07		1.00000000	INF

...

DOMAIN owner=public: 419 units

row	value	shadow price	lower bound	upper bound	
9) income.2-income.1	0.00000000	-0.2996129	0.000000		L
10) income.3-income.2	0.00000000	-0.2838323	0.000000		L
11) income.4-income.3	0.00000000	-0.1511955	0.000000		L
12) income.5-income.4	0.00000000	-0.0655068	0.000000		L

x-variable	value	shadow price	cost of decrease	cost of increase
npv.0	22553292.3		1.00000000	INF

```
...
_____________________________________________________________________________
DOMAIN all:                                                          839 units
_____________________________________________________________________________
row                                value        shadow      lower     upper
                                                price       bound     bound
_____________________________________________________________________________
 13) npv.0 . . . . . . . . . . . . 44209549.0 1.00000000           max
_____________________________________________________________________________
     x-variable                    value        shadow    cost of   cost of
                                                price     decrease  increase
_____________________________________________________________________________
     npv.0  . . . . . . . . . . .  44209549.0 1.00000000 0.00000000 0.00750263
...
_____________________________________________________________________________
show/domain data=savo:                                               433 units
_____________________________________________________________________________
     x-variable                    value        shadow    cost of   cost of
                                                price     decrease  increase
_____________________________________________________________________________
     npv.0  . . .
...
_____________________________________________________________________________
show/domain data=vaasa:                                              406 units
_____________________________________________________________________________
     npv.0  . . .
...
```

For each domain, JLP prints first the problem rows, and thereafter the *x*-variables implied by the options of the **`show`** command. For domains given only with '**`show/dom`**', there are no problem rows to be printed.

2.4 General Operating Commands

In this section the following general operating commands are described:

`batch`	– JLP is running in batch mode
`include`	– include commands from a file
`list`	– list a section of a file
`help`	– get on-line help
`outfile`	– write output into a file
`outlevel`	– define the amount of the output
`printlevel`	– define the amount of the terminal output
`time`	– timing
`pause`	- halt execution

2.4.1 Batch mode

The default is that the program is running in interactive mode. If the program is running in batch mode, then the first command should be **`batch`**. In batch mode, the program stops (or control returns to the main program provided by the system manager) if an error occurs while in interactive mode only error messages are printed,

open include files are closed and the control is given to the terminal input. In batch mode, the program does not print the prompts (e.g. 'jlp>') when reading from the input stream.

2.4.2 Include

Commands can be read in from files using **`include`** command. For instance:

```
incl data.def
```

Part of the file can be read in giving initial and final address:

```
incl ex.in/*ex2:*
```

In this case the program reads the file until it reaches a line starting with '`*ex2`' (initial spaces are ignored). Input from file stops as a line starting with '`*`' is met. Both the first and last line can be ordinary commands that are executed. It may be a good practice to use comment lines (starting with '`*`','`!`' or '`;`') as addresses in the files. If the final address is not given, then the rest of file is included:

```
incl ex.in/*ex2:
```

If the command is:

```
incl ex.in/const
```

then only one line is read in. The default is that included files can be nested up to 6 levels.

2.4.3 List

A file or a part of it can be printed using **`list`** command. The syntax of **`list`** is as of **`include`** command, the difference is that all lines read are interpreted as comments. This command may be useful if you want to check what a file or a part of it contains before executing it using **`include`** command. The headers of all subroutines in file *jlpsub.src* are printed for Part 4 using:

```
list/all jlpsub.src/*=:**
```

Option '`/all`' means that all segments between lines starting with '`*=`' and '`**`' are printed. This option is available also for **`include`**.

2.4.4 Help

On-line help is based on the **`list`** command and on a help file that the user or system manager can edit. Command **`help`** alone is translated internally as:

`list/all jlp.hlp/*`. This prints all header lines in file *jlp.hlp* starting with '*'. The contents of a cell of the help file starting e.g. with '`*list`' can be seen using command

```
help list
```

You can also change the help file:

```
helpfile own.hlp
```

Thereafter the help information is read from file *own.hlp*. Each cell in the help file ends at a line starting with '`;`'. The current help file *jlp.hlp* is printed in Part 3.

2.4.5 Output

The user can control both the amount and channels of output (terminal and/or file and/or an internal buffer). Output to the different channels is controlled independently, so that the user can direct output to any combination of the output channels (e.g. so that output goes to all three channels, or nowhere). Output to the internal buffer and the access to the solution vectors are described in Part 4.

Output file (**`outfile`**)

Printed output can be written at any time to a file (in Macintosh with LS-FORTRAN this is seldom useful as all output goes to a window that can be edited, printed and saved after the session). An output file is opened as:

```
outfile out.txt
```

The output file can be changed by giving a new **`outfile`** command. If no file name is given, then the old output file is just closed. Writing to the output file does not affect the terminal output, which is controlled independently. See Chapter 4.2 for how new files are opened in a non-VMS environment.

Level of output (**`printlevel, outlevel`**)The amount of output to the default output unit (the screen in interactive mode) is controlled using **`printleve`**l command. Command

```
printlevel 0
```

prevents all printing. Using '`printlevel 1`' only the solutions of the linear programming problems are printed (in addition to the commands read from include files). Printlevels 2 and 3 give information about the structure of the data and the about the progress of the optimization algorithm. Higher printlevels than 3 should be used only in trouble-shooting.

The amount of output to the output file is controlled using **`outlevel`** command. It works exactly as the **`printlevel`** command. Note that one can use different `printlevel` and `outlevel` in different parts of the job. Selected solutions of the linear programming problems can be printed to the output file as follows:

```
outfile sol.out
outlevel 0        ! 'outfile' sets automatically the outlevel to 1
problem
... !
solve
```

... the solution shows that the problem needs to be modified

```
solve +1  ! solve with next RHS or define and solve a new problem
```

...solution is OK

```
outlevel 1 ! start printing to the file
recall     ! the last solution can be reprinted with recall
schedules  ! basic schedules can be printed with this command
outlevel 0 ! start searching new interesting solutions
```

2.4.6 Time

If the system manger has provided a subroutine for measuring time, then the elapsed time between two points in the flow of program can be measured with command **`time`**. The total time from the first **`time`** command is also printed. If the system manager has supplied a subroutine for measuring CPU time, it is also printed. Time used to solve a problem is automatically printed.

2.4.7 Pause

If the commands are read from an include file, JLP may print results too fast. With **`pause`** command the execution of the program halts if the program is not running in batch mode. Typing <return>, the program continues. The system manager may provide a subroutine with better scrolling properties for the terminal output (see Chapter 4.11).

2.5 Data Management

JLP can solve linear programming problems including *x*-variables defined as (see Eqs.1.3, 1.7 and 1.10):

$$xk = \sum_{i=1}^{m} \sum_{j=1}^{n_i} x_k^{ij} w_{ij}, \quad k = 1,\ldots, p$$

The simulation system generates coefficients x_k^{ij} for different treatment units *i* schedules *j* and variables *k*. Generally the number of coefficients x_k^{ij} can be very large. Thus the main effort in the data access is to handle efficiently these coefficients (*x*-variables).

JLP can solve problems with different constraints for different domains, i.e., groups of treatment units. Thus JLP needs also the capabilities for handling variables that describe treatment units. These variables are called *c*-variables. A common description (identification) to all units read from the same **`xdat`** file can be given with *d*-variables. The user can also define symbolic names for numeric constants. Constants, *d*- and *c*-variables can be used as parameters of transformations or for defining domains. This chapter describes briefly how JLP manages constants, *d*-variables, *c*-variables and *x*-variables (i.e. coefficients x_k^{ij}). These variables are called *data variable* (other variables getting values in the optimization process, e.g., *z*- and *w*-variables are *linear programming variables*).

2.5.1 Summary of data manipulation

Initilization commands

(necessary commands are in bold)

`path`	directory of the data
`dtran`	transformations made when data files change
`xdat`	the names of *x*- files
`xvar`	the names of *x*-variables in the xdat files
`xform`	format for reading **`xdat`** files
`xtran`	transformations of *x*-variables, '`then reject`' transformation interpreted as '`then reject = -1`'
`keepx`	the *x*-variables stored, default: **`xvar`**- variables and output variables of **`xtran`** transformations
`cdat`	the names of *c*- files, necessary unless '`xform m`' is in effect
`cvar`	the names of *c*-variables in the *c*- files, must include *ns*
`ctran`	transformations of *c*-variables
`keepc`	the *c*-variables stored, default: **`cvar`**- variables and output variables of **`ctran`** transformations
`cform`	format for reading **`cdat`** files, default: same as **`xform`**
`save`	the data are saved in JLP format on disk
`const`	constants used in transformations

JLP reads the data using following logic:

Reading data

```
If keepx command was not given put all xvar variables into keepx list.
If keepc command was not given put all cvar variables into keepc list.
do ifi=1, (number of xdat files)
     Open ifi^th cdat file.
     Open ifi^th xdat file.
     V(ivdata) = ifi    ! ivdata is the number of variable 'data'
     Make dtran transformations.
     iunit=0
     do until end-of-file in cdat file
          iunit=iunit+1
          Read cvar variables of the unit iunit from the cdat file.
          V(ivunit)= iunit
          Make ctran transformations.
          Store keepc variables in cmat matrix.
          If there is not enough space in xmat matrix, write first
          units in xmat in the save or scratch file.
          do is = 1, V(ivns)! V(ivns) = number of schedules
               Read xvar variables from the xdat file.
               V(ivs)=is   ! ivs is # of the variable 's'
               V(ivrej)=0  ! ivrej is # of the variable 'reject'
               Make xtran transformations.
               Store keepx variables in xmat matrix.
          end of loop over schedules
     end of loop over units
end of loop over files
If save command was given, save data in JLP format.
```

When the data need to be modified later, then basically the same loops are executed but instead of reading data from **`cdat`** and **`xdat`** files, data are obtained from `cmat-` and `xmat-` matrices (and possibly from a working file, if data are too large for `xmat`).

Modification commands

`dtran`	transformations made when data files change
`xtran`	transformations of *x*-variables
`ctran`	transformations of *c*-variables

`const`	constants used in transformations
`dupl`	transformations defining how to duplicate schedules
`split`	what variables are split if splitting of units is indicated in xtran transformations
`save/later`	the data are saved in JLP format after modifications

If **`xtran`**, **`ctran`**, or **`dupl`** transformations are given, then JLP executes the following modification loops when **`make`** or **`problem`** command is met:

Transforming data

```
do ifi=1, (number of xdat files)
     V(ivdata) = ifi    ! ivdata is the number of variable 'data'
     Make dtran transformations.
     do iunit=1, (number of units in the xdat file)
          Get old keepc variables of unit from cmat matrix into V-vector
          V(ivunit)= iunit ! variable 'unit' is the within file unit #
          Make ctran transformations.
          Store old keepc variables and new output variables in cmat
          If unit is stored in working file, read one record (which
            contains one or more units) into xmat matrix
          Make space for new x-variables
          do is = 1, (number of schedules)
               Get keepx variables from the xmat matrix.
               V(ivs)=is   ! ivs is # of the variable 's'
               V(ivdupl)=0 ! ivdupl is # of variable 'duplicate'
               If 'reject' was not in keepxl then V(ivrej)=0
               Make dupl transformations
               ndupl=V(ivdupl) ! get number of duplicates
               do i=0, ndupl
                    V(ivdupl)=i
                    Make xtran transformations.
                    Store keepx variables
                    If V(ivsplit)>0 store nesessary information
                    !  ivsplit is # of variable 'split'
               end of loop over duplicates
          end of loop over schedules
          If unit was split, sort schedules
          If schedules were duplicated, update number of schedules
```

```
      end of loop over units
  end of loop over files
  If any unit was split, generate new units by duplicating rows of cmat and
   by updating unit related lists
  If save/later command was given, save data in JLP format.
```

2.5.2 Data variables

This section describes handling of data variables, i.e., constants, *d*-, *c*-, and *x*-variables. Constants, *d*-variables and *c*-variables can be used to define domains for constraints. Constants and *d*-variables can be used as parameters in **`ctran`**-, and **`xtran`** transformations, and *c*-variables can be used as parameters in **`xtran`** transformations.

All data variables are referred using symbolic names. All variable names are stored in the same vector. When JLP reads and transforms data, all current values of different variable levels are put to the same vector. This makes it possible that transformations of a given level of variables can use variables at the higher levels.

Variable names must start with a letter `A-Z` or `a-z` (not with 'ÄÅÖäöå') and cannot contain characters `!"=*,/:-%`. Variable names are case sensitive. For instance, name 'a#$.1' is a valid variable name. The system manager can decide the maximum length of variable names (see Chapter 4.1, 32 characters is the default).

Variables are referred using variable lists. Variable lists are formed by separating variable names with commas. A variable list with consecutive variable names can be formed, e.g., as follows:

```
xvar income.1,-income.3,volume != income.1,income.2,income.3,volume
const a,-d=2*1,2,3             ! a,b,c,d =1,1,2,3
```

Note that ',–' construction is interpreted using variable names and not the order of variables (compare to transformation loops described in section 2.5.3). For instance, if **`xvar`** command is:

```
xvar income.1,volume.1,income.2
```

then row

```
income.1,-income.2
```

in a **`problem`** paragraph is equivalent to:

```
income.1,income.2
```

The following variables are automatically created:

`data` = the number of data file to be read in (d-variable)
`unit` = the number of calculation unit (c-variable)

`s` = the number of the treatment schedule (x-variable)

Variables `duplicate, split, and reject` have also predefined meanings, as will be explained later in this chapter and in the *variables* section of the reference manual.

Constants

Constants are created and given values using **`constant`** command, or are created by **xdat** command. For instance

```
constant harvestcost,logprice,private = 100,230, 2
```

creates three new variables (if they do not already exist) and gives values to these constants. These constants can then be used as parameters in transformations or domain specifications when defining the linear programming problem. For instance:

```
xtran
income.1=volume.1*(logprice-harvestcost) ! income during first period
/
problem
owner=private:
income.1 > 1000
...
```

We can use standard transformations defined in an include file and load current parameters from a second include file.

Command **`xdat`** creates automatically constants from the file names. For instance, command

```
xdat south.xda,north.xda
```

creates constants '`south`' and '`north`' with values 1 and 2. If data file names start with a digit then letter '`d`' is prepended to the name (e.g. constant '`d21`' is created from file name *21.dat*). If the data file name is not a valid variable name, (e.g. it contains characters '`%/:`'), then no error occurs but the file names cannot be used as constants in transformations or domain specifications. Note that the directory specification for *x*-data or *c*-data files can be given using **`path`** command.

D-variables

D-variables (variables describing data sets) get new values when the data file changes. A *d*-variable '`data`' gets automatically the number of the data file, and other *d*-variables are defined by **`dtran`** transformations. In **`dtran`** transformations all constants can be used as input variables. With *d*-variables one can create parameters for ctran- or xtran-transformations or define domains. For instance, assume that **`xdat`** command is

```
xdat south.xda,north.xda,east.xda
```

Then we can define transformations and domains as:

```
dtran
if data=south then logprice=200
if data=north then logprice=150
if data=east then logprice=180
/
problem
data=south & owner=private:
income.1,-income.4 > 2000
...
```

If data are stored in the JLP-format then **dtran** transformations are written automatically to the '*.sav*' file. When the saved data are used later, the original division into different data files remains (from user's point of view).

C-variables

C-variables (class variables) get new values when the treatment unit changes. *C*-variables are read from **cdat** files or made by **ctran** transformations. Command **cvar** tells what are the *c*-variables in the **cdat** file, and format is given by **cform** command. Constants and *d*-variables can be used in **ctran** transformations. A *c*-variable 'unit' gets automatically the number of the treatment unit within the **cdat** file (i.e. for the first unit of a new file, variable 'unit' gets again value one). With *c*-variables one can create parameters for **xtran** transformations or define domains. *C*-variables need to include variable with name 'ns' which tells the number of treatment schedules for each unit. For instance:

```
cform *
cdat south.cda,north.cda,east.cda
cvar ns,owner,distance
ctran
logging_cost=a*distance + b ! a and b are defined with dtran or const
/
problem
owner=private:
logging_cost min
...
```

The format of *c*-variables data needs to be one of the following types:

```
cform *          ! FORTRAN free format
cform b          ! the data are in binary sequential file(s)
cform (10f5.2)   ! FORTRAN format, all data are read in as real variables
```

If the format of *c*-data is the same as the format of *x*-data, then **cform** command is not necessary. Use of input subroutines written by the system manager is indicated by '**xform** m' command and no **cform** command is needed in that case either.

The directory for the files can be given by **path** command. Commands **cdat**, **cvar**, **ctran** and **cform** can be given at any order before the data are read in by **read** command or at first **problem** command.

By default all **cvar** variables and output variables of the **ctran** transformations are stored in memory and in '*.cdj*' file if data are saved in the JLP-format. If some of these variables are not used in later problem definitions and there is a shortage of memory, then you can define what variables are stored using **keepc** command. For instance:

```
cvar ns,c1,-c100
keepc ns,c6,-c10
```

X-variables

X-variables get new values when the treatment schedule changes. *X*-variables are read from **xdat** files or made by **xtran** transformations. Command **xvar** tells what are the *x*-variables in the xdat file. Constants, *d*-variables and *c*-variables can be used in **xtran** transformations. An *x*-variable 's' gets automatically the number of the treatment schedule within the treatment unit. *X*-variables are used to define constraints and the objective function for the linear programming problem. Treatment schedules can be rejected using special 'reject' variable. If you have 'reject' among the **xvar** variables then all schedules with negative value of 'reject' are rejected. Schedules can also be rejected using **xtran** transformations:

```
xtran
if unit.eq.1.and.s.eq.3 then reject
if herbicides>0 then reject
/
```

In the above transformations reject is in fact interpreted as 'reject=-1'. If variable reject is read from the data, then its values can be changed in **xtran** transformations. Rejected schedules remain in the data, and they can be accepted again with new **xtran** transformations by giving value 0 to variable reject . For instance, no schedules will be rejected after the following **xtran** paragraph:

```
xtran
reject=0
/
```

The format of *x*-variables data needs to be one of the following types:

```
xform *          ! FORTRAN free format
xform b          ! the data are in binary sequantial file
xform (10f5.2)   ! FORTRAN format, all data are read in as real variables
xform m          ! use input functions defined by the system manager
```

If **xform** is 'm', then there need not be **cdat** or **cform** commands.

The directory for the files can be given by **path** command. Commands **xdat**, **xvar**, **xtran** and **xform** can be given in any order before the data are read in by **read** command or at first **problem** command.

By default all **xvar** variables and output variables of the **xtran** transformations are stored in memory and in '*.xdj*' file if data are saved in the JLP-format. If some of these variables are not used in later problem definitions and there is shortage of memory, then you can define what variables are stored using **keepx** command before the data are read. For instance:

```
xvar x1,-x100
keepx x1,x6,-x20
```

Selecting the type of a data variable

Functionally equivalent results can be often obtained by defining a variable as a constant, *d*-variable, *c*-variable or *x*-variable. For instance, assume that **xtran** transformations include:

```
income=price*volume
```

If the same price applies everywhere, then the variable 'income' will get the same values if variable 'price' is given a value within **const** command or within **dtran**, **ctran** or **xtran** transformations. But with **const** the value of 'price' is given only once while using **xtran**, for instance, JLP creates a vector having as many elements as there are treatment schedules, and each element has the same value. In large problems it is useful to keep variables at the highest possible level, or at least above the *x*-variable level.

It is possible that a variable with a given name belongs to two or more variable levels simultaneously (e.g. the variable is both among **cvar** and **xvar** variables). However, this will probably cause trouble if it is not carefully taken into account how different variables get their values (see section 2.5.1).

2.5.3 Transformations

New variables can be created with transformations. The same transformation compiler is used for **dtran**-, **ctran**- and **xtran** transformations and to interpret domain specifications in problem definitions. Compiled transformations are fast to compute.

Dtran transformations (transformations defined after **dtran** command) are computed when the data file changes. **Ctran** transformations are computed when the treatment unit changes. **Xtran** transformations are computed as the treatment schedule changes. Transformations should (if their use can be anticipated) be defined before the data are read in at the first **problem** command or using **read** command. When data are read in, then **ctran**- and **xtran** transformation definitions are cleared (not **dtran** transformations). One can later define new transformations, and these new

transformations will be computed automatically when the next **problem** command is encountered or when computation is explicitly required with **make** command (both **read** and **make** are optional commands). The syntax of transformations is similar to the FORTRAN syntax except that all functions must be written in lower case. For instance:

```
x5=sin(x2**2+sqrt(log(x4-2)))
if (x3+x2=4 .or. sin(x3)>0.5) then    ! outer parentheses are not necessary
x7=x5-7
else
x4=x3**2.2+tan(x5)
end if
```

Transformations are defined in **dtran**, **ctran**, and **xtran** paragraphs. If there are old transformations (i.e. an **xtran** paragraph, for instance, is given for second time before the transformations are actually computed), then new transformations are appended to the old ones. One can clear all previous compiled transformations at any time by entering 'transformation' *clear*. **Ctran** and **xtran** transformations already computed and stored in the memory cannot be withdrawn. All previously defined **dtran** transformations can be cleared. When defining new transformations, existing variables can be used as output variables (the old values will be replaced). Examples:

```
xtran
x1=0
/
prob   ! after this we cannot recover what x1 was earlier
```

If we had noticed before the **prob** command that we were accidentally zeroing `'x1'` we could prevented this:

```
xtran
clear
/
prob   ! x1 is what it used to be
```

Arithmetic operations

Standard **,*,/,+, and - operations are available. In addition there is an additional "– operation for raising a variable to an integer power (internally all data variables are REAL*4). For instance, (-1)"2=1 but (-1)**2 is undefined. Integer powers, when applicable are faster to compute and are defined for negative arguments. The hierarchy of operators is: ",**,*,/,-,+.

Supported FORTRAN intrinsic functions

```
abs(x)            = absolute value  of x
atan(x)           = arctangent, result is in radians
cos(x)            = cosine, x in radians
```

```
cosd(x)           = cosine, x in degrees
exp(x)            = exponential
int(x)            = truncation to integer
log(x)            = natural logarithm
log10             = log base 10
max(x1,...,xn)    = largest value of x1,...,xn
min(x1,...,xn)    = smallest value of x1,...,xn
mod(x1,x2)        = remainder of x1/x2
sin(x)            = sine, x  in radians
sind(x)           = sine, x in degrees
sqrt(x)           = square root
tan(x)            = tangent, x in radians
tanh(x)           = hyperbolic tangent
```

Additional functions

```
ran(x)       = uniform random number between 0 and 1, with seed x
               based on RAN1-algorithm of Press et al. (1986,p. 196)
x1=swap(x2)  = change values of x1 and x2
```

Own functions

As described in Chapter 4.7, the system manager can create transformations that can be used in the same way as the predefined functions. The following function is included as an example of a 'user defined function':

```
npv(interest_percent,income1,time1,...,incomen,timen) = net present value
```

Logical operators

Following logical operators are implemented (below equivalent operators):

```
.gt.    .lt.      .ge.     .le.     .eq.    .ne.    .and.   .or.     .not.
 >       <         >=       <=       =               &
```

Constant π

In transformations one can use π with name '.pi':

```
atd=90*atan(x)/.pi    ! arctangent in degrees
```

If ... then structures

Examples:

```
if unit=1 then cost=cost+1      ! one-line if, this is equivalent to:
if (unit.eq.1) then cost=cost+1

if data=south.or..not.(sitetype<3) then
cost=2
price=4
end if

if data=south then
cost=2
else
cost=2.7
end if

if data=south then cost=3
np=npv(3,100,0,200,3)    ! this is computed always
then price=2             ! the previous test remains valid
diff=income.2-income.1   ! there can be transformation between
else price=3
```

Note that unlike in FORTRAN, '`then`' is necessary also in one-line if statement. No nested if...then structures are allowed. After one if...then statement, the test remains in effect and can be used in one-line '`then`' or '`else`' statements. No error will occur if 'if...then...(else)' structure is not closed with '`end if`', all statements after '`then`' belong also in that case to the range of 'if...then'.

Loops

Transformations can contain simple loops. Examples:

```
out=0   ! initialize out
%5:out=out+%x1   !out= x1+x2+x3+x4+x5
%4:%z1=%x1 +  %y1  !z1=x1+y2; z2=x2+y2
out2=0
%3:out2=out2+x1"%   !out2=x1+x1"2+x1"3
out3=0
%3,2:out3=out3+%x1 +%  !out3=x1+1+x3+2+x5+3
%3:      ! loop can contain several lines
tmp=%x1/%   ! tmp=x1/1 ; tmp=x2/2 ; tmp=x3/3
%z1=tmp*%y1 ! z1= tmp+y1 ;z2= tmp
%end
```

Loops begin with '%*n*' where n tell how many times the loop is done. Character '%' in front of a variable tells that the variable number is incremented at each iteration. The default increment is one. If the loop begins '%*n,i*' then the increment is *i*. Within transformations variable '%' gets values 1,2,..,*n* (even if the increment *i* is greater than 1, %-variable is incremented by steps of one). The variable numbers are incremented without a reference to names of variables. Consecutive new variables are created by **`const`**, **`xvar`** and **`cvar`** commands, and by **`dtran`**, **`ctran`** and **`xtran`** transformations. For instance, let **`xvar`** command and **`xtran`** transformations be:

```
xvar x1,y1,x2,y2
xtran
out=0
%3:out=%x1
/
```

Then variable out gets value `x1+y1+x2`. One cannot create new variables within a loop. New consecutive variables can be created using **`const`** command:

```
const out1,-out4
xtran
%4:%out1=%x1+%z1
/
```

If variables `out1,-out4` did not exist and were not created with **`const`**, then only variable `out1` will be created properly.

If a %-loop of several statements is initialized and not closed properly, then everything after the beginning of the loop is computed once (i.e. the loop is ignored), and no error occurs.

2.5.4 Saving data in JLP format

Data can be stored in a special JLP-format. If the *x*-variables data exceed the memory reserved, the initial part of the *x*-data is stored automatically in this format. The data are saved if one uses the following command:

```
save filename
```

The exact effect of the **`save`** command depends where the command is given. If the data are not yet read in, data are saved later at the time when the data are read at **`problem`** or **`read`** commands. If the data have been already read in, then the data are saved immediately. This makes a difference in case *x*-data exceed the memory. If **`save`** command is not given before reading data, then a part of *x*-data (treatment schedules) are written twice, first to a scratch file when reading data, and then to a named file when saving data. New variables created during the session can also be saved with **`save`** command. If new variables have been already created, then saving is done immediately, in other case at the same time as new variables are created with **`problem`** or **`make`** commands. Even if there are unsaved variables, the saving can be postponed to the next time new variables are created by giving command in form:

```
save/later filename
```

When the data are stored, three files are created. A binary file *filename.xdj* contains the *x*-data. Another binary file *filename.cdj* contains the *c*-data (variables describing treatment units). A text (ASCII) file *filename.sa*v contains **`const`**-, **`xdat`**-, **`keepx`**-, **`keepc`**-, **`dtran`**- and **`unsave`** commands that are needed to read in data stored in JLP-format. The file contains also the history of the file as comments. The saved data can be read with command:

```
include filename.sav
```

The following example with the data used in section 2.3.1 shows that significant savings in computer time can be obtained with saving data in JLP -format.

```
jlp>incl ex.in/*save:*
> *save
> xdat savo.xdb
...
> time
starting timing..
> read  ! data can be read in with read command, this is not necessary
reading xdat-file: savo.xdb
reading cdat-file: savo.cda
number of calculation units, schedules: . . . . .     433  12100
...
> time
elapsed: 27.00000 total: 27.00000
> save savo !saves data with JLP -format
**definitions saved in file: savo.sav
       c-data saved in file: savo.cdj
       x-data saved in file: savo.xdj
> init  !get a fresh start
> time
elapsed: 6.617187 total: 33.61718
> incl savo.sav  ! this will read in saved data
>  ** saved data:*
>  xform b
...
>  ;# of units in files   433
>  ;total number of schedules:       12100
>  ;number of rejected schedules:            0
>  unsave savo.cdj    savo.xdj
> time
elapsed: 5.349609 total: 38.96679
jlp>
```

With **save** command the data are written to binary files with a special record structure (see section *saveform* in the reference manual). Data can be written to disk with a simple record structure similar to the structure of **xdat** and **cdat** input files using **write** command. This is needed, e.g., when transferring data in ASCII files to a different computer system.

2.6 Problem Definition

LP-problems are defined in **problem** paragraph. One **problem** paragraph may specify several RHS's, and which problem is actually solved depends on the **solve** command. A **problem** paragraph consists of sections:

```
domain1: ... domainn:
constraint (or objective)
...
constraint  (or objective)
```

Each domain in a domain specification line applies to all *x*-variables in the following constraints.

2.6.1 Domains

If the domain consists of all treatment units, then the domain specification is given as:

```
all:
```

If this is the first domain in the **problem** paragraph, the domain specification can be omitted.

An ordinary domain specification is given by a logical statement determining when the domain applies. The transformation compiler interprets a domain specification in form:

```
if (domain specification) then (domain applies)
```

In the domain specification one can use constants, *d*-variables and *c*-variables as well as arithmetic operations, for instance:

```
data=south.and.(owner=private.or.sin(elevation)+altitude.gt.10) :
(unit>2.and.unit.le.237).or.sitetype=wasteland :
```

Note that the colon ':' is used to indicate the end of a domain definition. A domain can consist even of a single treatment unit only. See section 2.7.2 for printing domains, i.e., domains that are used to classify units only in the printing of an LP solution.

2.6.2 Constraints

The form of a (utility) constraint line is either:

```
x-variable_list range_1 / range_2 / ....
 or
coef1*var1+coef2*var2 + ..coefn*varn.  range_1 / range_2 / ....
```

Examples:

```
income.1,-income.5 = 10000 />800 <1500 / >750
-y1+income.2 + 1.6*S1-S2 - 1.28*L1+L2 = 0 ! income.2 only is an x-variable
```

The *x*-variable list in the first alternative may contain several *x*-variables. Variables in the second alternative may be *x*-variables and *z*-variables, i.e. nonnegative variables that are needed to define a linear programming problem. If the coefficient is one, it can be omitted. *Z*-variables are always global, i.e., they do not relate to domains. If domain specific *z*-variables are needed, they should be created explicitly. For instance, if in a goal programming problem there are target levels for both *savo* and *karelia,* then the problem paragraph should contain separate slack and surplus variables for both domains:

```
prob
data=savo:
income.1 -savosurplus.1 + savoslack.1 =   800000
```

```
....
data=karelia:
income.1 -kareliasurplus.1 + kareliaslack.1 =   600000
...
/
```

RHS Range

A specification for a RHS range is some of the following types:

```
= 100
>100
<200
>100 <200
<200 >100 ! equivalent to the previous one, no special order for '>' and '<'
```

The RHS's to be used when the problem is solved are selected with **`solve`** command described in section 2.7.1.

Chapter 4.8 describes how the system manager can develop own methods for generating RHS's. These methods can use the range specifications given in **`problem`** paragraph as parameters.

2.6.3 Objective

The objective function and the type of the problem is given as follows:

```
coef1*variable1+coef2*variable2 + ...    max
 or
coef1*variable1+coef2*variable2 + ...    min
```

If there were several domain specifications in the previous domain specification line, the first domain applies. For instance:

```
problem
all: data=south: data=north
incomed.1,-incomed.5=0
presentvalue max
/
```

Now the domain for the objective variable is 'all:'

The objective row can be anywhere in the **`problem`** paragraph and it can belong to any domain. It is not necessary to have objective row at all. If no objective function is given, JLP just finds a feasible solution when it is asked to solve the problem (it is possible to ask JLP to find a feasible solution even if objective function is included)

2.6.4 Using different domains on the same row

It is assumed that the *x*-variables on an objective or constraint row are all in the same domain. For instance, if it is required that variable `volume.1` should be equal in Savo

and Karelia, then this constraint can be expressed as follows using extra *z*-variables to define global domain specific *x*-variables:

```
data=Savo:
savovolume.1-volume.1=0  ! this defines volume.1 in Savo as a global
                         !   z- variable
data=Karelia:
kareliavolume.1-volume.1=0
savovolume.1-kareliavolume.1=0 ! this constraint can be after any domain
!              specification as it contains only global z-variables
```

2.7 Solution

2.7.1 Selecting the problem to be solved

JLP starts solving a problem when it gets command **`solve`**. If several RHS's are given in the **`problem`** paragraph, then the appropriate RHS can be selected as follows:

```
solve    ! Solves the problem corresponding to first right-hand side.
solve 3  ! Solves the problem corresponding to third right-hand side.
         ! If for a constraint the are not 3 RHS's, the last one is used.
         ! If no constraint contains 3 RHS's, return to read new commands.
solve +1 ! Solves the problem corresponding to the next right-hand side.
         ! The RHS counter must be initilized with 'solve' or e.g. 'solve 5'
```

If the number of ranges in a constraint line is less than the number of column given in the **`solve`** command, then the last range is used. If no constraint line has enough ranges, no problem is solved.

If the system manager has written an own subroutine to generate RHS's (see Chapter 4.8), this subroutine is called when solve command is given with an option starting with 'm'. For instance:

```
solve/myown 3
```

If no objective function was given in the **`problem`** paragraph, **`solve`** will find a feasible solution. JLP will find only a feasible solution even if the objective function was included, if **`solve`** command is replaced with **`feasible`** command. The syntax for **`feasible`** command is the same as for **`solve`**.

Timing comparisons are meaningful only if **`solve`** is given with option **`/i`** that forces JLP *not* to use the previous solution as the starting point.

2.7.2 Printing options

This section describes JLP commands controlling how the solution is printed. An interpretation of the shadow price and marginal cost variables printed is in the next

section (2.7.3), a mathematical description is given in Chapter 6.4. Chapter 4.10 describes how the system manager can write an own report writer.

Printing rows and *x*-variables

After solving a problem, JLP prints the solution (if `printlevel`>0). Values of rows and *z*-variables are always printed. User can determine with **`show`** command what else is printed. The same solution can be printed with different **`show`** options using **`recall`** command. The format of the **`show`** command is as follows:

```
show(options) variable_list
```

The most important options are (see Reference Manual for more details, and how to negate the following options):

```
/nox      ! print no x-variables
/noxfirst ! print no x-variables automatically after solution,
          ! print x-variables information only with recall command
          ! as specified with other show options
/all      ! print all x-variables (default)
/prob     ! print variables used in problem
/cost     ! compute cost of decrease and cost of increase for x-variables
            (default)
/nocost   ! costs are not computed
/int      ! compute the integer approximation
/domains  ! start paragraph that defines domains that are used when
            computing x-variables (in addition to domains used in the problem)
/nodom    ! do not use extra printing domains
```

The computation of cost of decrease and cost of increase of *x*-variables may take quite a lot of time. As a rough approximation the time used to compute the values of *x*-variables and costs is

$$\frac{\text{(number of } x\text{-variables)*(number of domains)}}{\text{number of rounds through units in optimization}} \times \text{(optimization time)}$$

The main part of time is spent in computing costs (this part of the software is new, currently it is not well optimized and tested). The user may wish that this information is not computed. The purpose of `/noxf` option is to allow the user first look the rows of the solution, and then get a more detailed output with **`recall`** if the solution is interesting.

The integer approximation is computed so that only the schedule with largest weight is applied in each unit. No integer optimization is done, and the integer approximation does not generally satisfy the constraints.

The optional variable list in the **`show`** command tells what variables are printed in addition to the variables appearing in the problem.

Section 2.3.4 contains an example of the use of **show** both with **/dom** option and variable list. Another example:

```
show/nodom/dom !/nodom clears previous domains /dom tells that new ones follow
owner=private: ! remember colon
owner=public & site = wilderness:
/
```

Reprinting the last solution with other options

JLP prints the solution automatically after solving the problem using the current **show** options except if /noxf option is in effect. If the **show** options are changed, or if the printing parameters have changed, or if the user just wants to see the results again, the solution can be printed again with **recall** command. If /noxf option is in effect, then the current **show** options are used for the first time at **recall.**

Printing weights and shadow prices of schedules

After solving a problem, information about weights and shadow prices (marginal values) of schedules can be printed using **sched** command. This command has the following options:

```
sched          ! print all basic schedules (schedules used in the solution)
sched n        ! print at most n schedules
sched/all      ! print also values of nonbasic schedules
sched/all n    ! print at most n schedules
sched/all>95   ! print all schedules whose shadow price > 95% from the
               ! value of the basic schedules of the unit
sched/all>95 n ! print at most n schedules
```

Example:

```
jlp>sched 100
value% of unit: % is from sum of unit values  33459072.9

  unit  value%  sched     %     sched   %
    1  0.217174    8  100.0000
    2  0.208857    4  100.0000
    3  0.015557    2  100.0000
...
   82  0.187835   40  100.0000
   83  0.262362   21  58.80239   27  41.19760
   84  0.176998   16  100.0000
   85  0.074584    4  100.0000
   86  0.198073    3  100.0000
   87  0.162058   17  100.0000
   88  0.153418    2  67.96054    3  32.03945
   89  0.112786    1  100.0000
```

The unit and sched columns tell the unit number and schedule numbers for basic schedules. The '%' column is the weight of the schedule multiplied by 100. There can be several basic schedules in a unit. The value% column tells how many per cents is the shadow price of the unit from the sum of shadow prices of all units. Thus the value%

column adds up to 100. The shadow price of unit 1 in the example is 0.00217174*33459072.9 = 72664.4.

The printing format is different with the `sched/all` option:

```
jlp>sched/all>99.5 150
  unit sched  value% of s   share%   value% of unit
    1    8    100.0000    100.0000    0.217174
    2    4    100.0000    100.0000    0.208857
...
    6    2    100.0000    100.0000    0.068647
    7    1    100.0000    100.0000    0.026881
    8    2    99.92227                0.011231
    8    4    100.0000    100.0000
    9    1    100.0000    100.0000    0.184652
...
   83   12    99.55764                0.262362
   83   21    100.0000    58.80239
   83   27    100.0000    41.19760
```

This option prints all schedules on separate lines. The `share%` column is the weight of the schedule multiplied by 100 (= `%` column in the first format). The column '`value% of unit`' tells how many % is the shadow price of the unit from the sum of shadow prices of all units (= `value%` column of the previous format). Column '`value% of s`' tells how many percent the shadow price of the schedule is from the shadow price of the unit. This is at least as great as the specified printing limit. For all basic schedules this figure is 100. For rejected schedules, the '`value% of s`' may be over 100. For instance, let us solve the same problem as above after transformation:

```
> xtran
> if unit=2.and.s=4 then reject  ! this was a basic schedule above
> /
```

We will then get a slightly different solution and:

```
jlp>sched/all>99.5 80
value% of unit: % is from sum of unit values  33458479.3

  unit sched value% of s    share%   value% of unit
    1    8    100.0000    100.0000    0.217178
    2    3    100.0000    100.0000    0.207087
    2    4    100.8566    rejected
    3    2    100.0000    100.0000    0.015557
```

2.7.3 Marginal analysis of the solution

The dual problem of an LP problem can be used to analyze marginal changes of the objective function caused by slight modifications of the original problem. Chapter 6.4 describes in more detail the mathematical basis of the marginal (dual) analysis of the problems solved by JLP. This section indicates how to interpret the marginal price information JLP computes.

A marginal change of the objective function has the following meaning. Assume that a constant in a problem has value ξ, and the objective function has the value z_0. If z_{0+} is

the value of the objective function when the problem is solved replacing constant with a new value $\xi+\varepsilon$, then $(z_0-z_{0+})/\varepsilon$ is the marginal change in the objective function. In *linear* programming, $(z_0-z_{0+})/\varepsilon$ is independent of ε provided that ε is so small that the current basis does not change. The marginal changes of the objective function may be called, depending on the context, marginal prices, or shadow prices, or reduced costs. In forest management planning problems where both the number of treatment units and the number of simulated schedules are relatively large, and the schedules follow the same logic of forest growth, the marginal prices may change quite little even if the basis will change.

Shadow price of a utility constraint

The shadow price of a constraint is the marginal change of the objective function when the RHS of the constraint is increasing. The effect of decreasing the constraint is the opposite. JLP prints automatically the shadow prices of the utility constraints. The shadow prices are for the lower or upper bound depending which one is active (character 'L' or 'U' indicates this in the printed solution). Note that for an equality constraint (lower bound and upper bound are equal), either the lower or the upper bound is active. The following table shows how the sign of the shadow price (π) is determined:

active bound	the objective function is maximized	minimized
lower bound	$\pi < 0$	$\pi > 0$
upper bound	$\pi > 0$	$\pi < 0$
no active bound	$\pi = 0$	$\pi = 0$

The signs can be heuristically inferred as follows. If the lower bound is active, then increasing the lower bound will make the constraint more restrictive, and the objective function will become worse, i.e., smaller for maximization and greater for minimization.

The shadow price of the objective row is set to be one. This is in accordance with the equivalent problem formulation where the objective is always to maximize z_0 subject to the constraint that

z_0 – (the initial objective row) = 0.

The shadow price of this constraint would always be one.

Shadow price of an *x*-variable

The shadow price of an *x*-variable x_k is the shadow price of the constraint (1.3) or (1.10) that defines x_k as a sum over schedules. A natural way to interpret the shadow price of an *x*-variable is that it is the marginal utility of a unit of the *x*-variable obtained from other sources and used for satisfying the constraints of the problem. Alternatively, marginal change in x_k may result from a marginal change in a coefficient x_k^{ij} of a basic schedule *j* in unit *i*.

The shadow price μ_k of x_k can be expressed in terms of the shadow prices of the utility constraints as follows (see Chapter 6.4):

$$\mu_k = a_{0k} - \sum_{t=1}^{r} a_{tk}\pi_t \qquad (2.1)$$

where a_{tk} is the coefficient of x_k on row *t* and π_t is the shadow price of constraint *t*.

Note that if x_k is present only on one row *t* and with coefficient one (e.g. the constraint is like: `final_volume > 1000`), then

$$\mu_k = -\pi_t, \qquad (2.2)$$

i.e., the marginal changes in the objective function are opposite if we get an extra unit of quantity *k* from outside or if we require that the treatment units produce one unit more. If x_k does not have a nonzero coefficient in any binding utility constraint, its shadow price is zero (which is not printed).

When interpreting the shadow prices of *x*-variables, it should be kept in mind that effect of an extra unit of x_k obtained from another source is taken into account only through the explicit constraints and objective function. No implicit meaning or implications are taken into account. For instance, in the example in section 2.3.1, the net present value variable `npv.0` was maximized subject to smoothness constraints for incomes:

```
> prob
> income.2-income.1>0
> income.3-income.2>0
> income.4-income.3>0
> income.5-income.4>0
> npv.0 max
> /
```

The shadow prices of incomes were:

```
     x-variable            shadow
                           price

_____________________________________
     income.1   . . . .  -0.2052712
     income.2   . . . .  -0.0094556
     income.3   . . . .  0.12518680
     income.4   . . . .  0.04850047
     income.5   . . . .  0.04103964
```

This does not really mean that keeping the problem unchanged, the net present value would decrease if we will get more income during first period. But the problem formulation did not express the direct effect of income to present value. Extra income during first period would be just used in constraints, and in this case extra unit of `income.1` would make the constraints more difficult to satisfy. But we can change the objective function to take into account the direct relation between income and present value (3% interest rate, 10 year subperiods, incomes in the middle of subperiods, income variables are per year incomes):

```
> prob
...
> 0.228107*npv.5 + 2.644386*income.5 + 3.55383*income.4 + >
> 4.7760557*income.3 + 6.4186195*income.2 + 8.6260878*income.1  max
> /
```

Note that $0.228107=1/1.03^{50}$, $2.644386= 10/1.03^{45}$, etc. If JLP solves this problem, the same solution is obtained but, the shadow prices will be (shadow prices are divided by 10 to transform the per year scale of incomes to absolute scale) :

```
     x-variable          shadow
                         price/10

_____________________________________
     income.1   . . . 0.842081784
     income.2   . . . 0.640916123
     income.3   . . . 0.490124112
     income.4   . . . 0.360233488
     income.5   . . . 0.268542446
```

Thus `1` mark of income after 5 years will increase `npv.0` by 0.842 marks. Price 8.42 is smaller than the coefficient 8.63 of `income.1` in the definitions of `npv.0`. This is in accordance with the fact that the shadow price of `income.1` in the first formulation was negative.

The shadow prices of incomes can further be converted into interest rates as follows (see e.g. Lappi and Siitonen 1985). Let π_t be the shadow price of income at time t, and let $r_t=\pi_t/\pi_{t+1}$, then the internal rate of interest, i_t, between t and $t+1$ is $i_t = r_t^{1/d} - 1$. The internal rates of interest computed from the above shadow prices are:

$i_{01} = 3.5\%,\ i_{12} = 2.8\%,\quad i_{23} = 2.7\%,\quad i_{34} = 3.1\%,\quad i_{45} = 3.0\%$

The information obtained with the second objective function can be computed from results obtained for the first objective function. These relations will not be developed further here. The purpose of the above example of the analysis of shadow prices is to emphasize that the solution of a linear programming problem can be properly interpreted only if the basic properties of linear programming are understood.

Cost of decrease or increase of an *x*-variable

The optimal solution provides weights w_{ij} that can be used to compute the value of an aggregated *x*-variable x_k as

$$x_k = \sum_{i=1}^{m} \sum_{j=1}^{n_i} x_k^{ij} w_{ij}. \qquad (2.3)$$

The options of **`show`** command determine what *x*-variables are computed. Let ξ_k denote the value thus obtained. If we add a constraint that requires that x_k should have a value different from ξ_k. the objective function will generally change (even if the shadow price of x_k would be zero).

The cost of decrease tells how many units the objective function will change if x_k is required to decrease by one unit, and the cost of increase tells how many units the objective function will change if x_k is required to increase by one unit. The costs are expressed as positive values, so for a maximization problem, the cost is marginal decrease and for a minimization problem, a marginal increase in the objective function.

It may be that when a constraint is added that requires that x_k deviates from the observed value ξ_k, the resulting problem is infeasible. The corresponding cost can then be defined to be infine. Thus the `'INF'` printout of JLP can be interpreted either as '*infinite*' or '*infeasible*'. If the objective row in a maximization problem consists of a single *x*-variable, the cost of increase for that variable is automatically infinite.

For a basic *x*-variable x_k (i.e., x_k appears in a binding constraint or on the objective row) the cost of decrease or increase is mathematically related to the shadow price of the variable but it is equal to the shadow price only in special cases (generally only when x_k appears alone on one row). The cost of changing the value of x_k and the shadow price of x_k are based on different concepts of 'changing the problem slightly'. In the former analysis a constraint is added, and the latter analysis a constraint is modified. The cost of changing the value of a nonbasic *x*-variable may be easier to interpret than the cost of changing the value of a basic *x*-variable. For a basic *x*-variable, the interactions of the additional constraint with the original constraints may not be self–evident.

Reduced cost of a nonbasic *z*-variable

The reduced cost of a nonbasic *z*-variable tells how many units the objective function will change if the *z*-variable is forced to increase by one unit (from zero). These costs are always printed (if printing is allowed at all).

Shadow price of a treatment unit

The shadow price δ_i of the area constraint (1.4) or (1.11) for unit *i* is called the shadow price of the treatment unit *i*. If the area of unit *i* would increase by α%, then the change in the objective function would be $\alpha\delta_i/100$. The increase of the area by α% means that the coefficients x_k^{ij} for all *x*-variables *k* and for all schedules *j* in unit *i* are increased by α%. Marginal changes in the nonbasic schedules do not really have effect on the optimal value of the objective function, but it is easier to think that all schedules are changed.

The analysis of the dual problem reveals that the value of the objective function is:

$$z_0 = \sum_{i=1}^{m} \delta_i + \sum_{t=1}^{r} c_t^* \pi_t, \qquad (2.4)$$

where c_t^* is the active bound (either c_t or C_t).

Thus the shadow prices of the units do **not** generally add up to the solution. If the shadow price of a unit is negative in a maximization problem or positive in a minimization problem (and the unit is so small that the marginal analysis is valid), then a better solution would be obtained without the unit (thus the unit should be immediately sold to someone who does not understand linear programming).

The shadow price of a treatment unit is equal to the shadow price of any of the basic schedules in the unit.

Shadow price of a treatment schedule

The shadow price of a schedule is not really a shadow price of a constraint in the problem. The shadow price λ_{ij} of schedule *j* in unit *i* is here defined as:

$$\lambda_{ij} = \sum_{k=1}^{p} \mu_k x_k^{ij} \qquad (2.5)$$

For all basic schedules λ_{ij} is equal to δ_i, the shadow price of unit *i*. For a nonbasic schedule *j*, the difference $\delta_i - \lambda_{ij}$ is the marginal (reduced) cost of forcing schedule *j* into the solution. For a minimization problem, the cost computed as $\lambda_{ij} - \delta_i$ would

express the cost as a nonnegative quantity (costs are assumed here to be always positive)

2.7.4 Input parameters of the optimization

There are some parameters that determine how the optimization is done. The current values can be seen with **`parin`** command. Thereafter new values can be given in format 'parameter=value', and **`parin`** paragraph is ended with '/'. For instance:

```
jlp>parin
  start_mode  0.0000     0=norm 1=cont old 2=ffeas
  start_unit  0.0000
    maxvisit  0.0000
      invert   100.0     after given # of changes of basis
      trace1  0.0000     step # to start
      trace2  0.0000     step # to stop
        tole   1.000     coefficient for tolerances
        wmin  0.0000     = 0 change basis as you can , else =1
parin>invert=200
parin>/
jlp>
```

The user may wish to modify following parameters (other parameters are needed in tests). Note that if the effect of different options on the optimization time is studied, then **`solve`** command should be given in form **`solve/i`** so that optimization starts always from equal situation (otherwise the key schedules of the previous problem are used as the starting point).

invert

The basis matrix is reinverted after `invert` changes in the basis. Default for `invert` is 100. A reinversion of the basis takes time but it will remove rounding errors accumulated during the stepwise changes of the basis. After finding a solution, JLP inverts the basis if the basis has changed more often than 10% of the value of `invert`.

wmin

If `wmin=0`, then JLP enters a variable into basis even if its value will become zero (i.e. the variable will be a degenerate basic variable). If `wmin=1` , then variables with value zero are not entered. Changing the value of `wmin` may help if there are problems in the optimization (see Chapter 5.3)

tole

JLP tries to figure out what is the range of rounding errors. If the estimated tolerance is too small, JLP may get into trouble in computations. If the estimated tolerance is too

large, JLP may fail to reach the solution (the obtained solution is anyhow reasonable, but not necessarily optimal). The tolerances estimated by JLP are multiplied by parameter `tole`. This way the estimates can be corrected, if JLP runs into trouble or if the user feels that JLP does not find the optimum. See Chapter 5.3 for more information.

2.7.5 Output parameters of the optimization

JLP collects information about different steps of the optimization. The summary statistics can be seen with **`parout`** command. For instance:

```
jlp>parout
        nonfeasible constraint     0.
             unit last visited    69.
          rounds through units    15.
         improvements in units  1839.
       changes of key schedule   988.
                     w enters  1005.
          slack/surplus enters   192.
                     z enters     0.
                     w leaves   993.
          slack/surplus leaves   204.
                     z leaves     0.
                basis changes  1197.
     changes after reinversion     0.
     reinversions of the basis    12.
```

3. REFERENCE MANUAL (FILE *jlp.hlp*)

The up to date reference manual is stored in file *jlp.hlp* which is also used by the on-line help. With **help** command, JLP lists all lines starting with '*'. With '**help** *key* ' command, JLP lists the entry between '*****key' and ';'. Changes made after the printing of this manual will be indicated by '##' in file *jlp.hlp*.

Current modules are:

***batch *buff *buflevel *cdata *cform *command line *constants *ctran *cvar *dir *do *domain *dtran *duplicate *end *end do (enddo) *feasible *files *help *helpfile *include *init *integer approximation *keepc *keepx *list *make *mela *mrep *outfile *outlevel *ownread *own1 *own2 *parin *parout *path *pause *printlevel *problem *read *recall *reject *report *save *saveform *schedules *show *solve *split *system *time *title *transformations *unsave *values *variables *write *xdata *xform *xtran *xvar**

The current help file is listed below:

```
** file jlp.hlp ***
 (SYS.DEP) = the property is system dependent (see Part 4 for details)
 (not cmd) = the keyword is not a JLP command
;
*batch - JLP is running in batch mode.
Use as first command in batch mode. In batch mode, the run is terminated with
fatal errors, and prompts are not printed when reading commands.
;
*buff  - (SYS.DEP.) Calls user written interface subroutine 'buff'.
'Buff' can be used to make an interface that sends commands to JLP and reads
and interprets the results. The subroutine template provided with JLP just
prints the output buffer (controlled by 'buflevel')  and reads commands from
the terminal.
see also: buflevel, ownr, Chapter 4.11
;
*buflevel -(SYS.DEP.) Gives the amount of output send to the output buffer.
Usage: bufl i ! i= 0, 1, 2,....  Larger values of i indicates more output to
the internal output buffer:
0 = no output (default)
1 = only solution and problem definition
2 - 8 more and more output
```

Use only if your own routines have the full control. If buflevel is given negative value, then the subroutine 'ownwri' is called for each line to be printed. A template for 'ownwri' is given in file 'jlpint.src'. Currently the optimization algorithm prints information about how the optimization proceeds only to the terminal.

see also: printlevel, outfile, outlevel
;

***cdata** - Defines the names of the c-data files.

Usage: cdata file1,...,filen

It is recommended that names of text (ASCII) files end with ".cda", and names of binary files end with '.cdb'. (Files saved in the JLP-format end with '.cdj'.) During reading, JLP adds the directory specification given by 'path' command to the names. If 'xform m' is in effect, then the user subroutines may or may not utilize the names in cdat command (SYS.DEP.).

see also: cform, cvar, xdata, xform, save, path
;

***cform** - Defines the format of the c-data.

Usage: cform form ! where

```
form =  *     if c-data can be read with FORTRAN '*' format
        b     if c-data are in binary files
    (8f10.0)  any FORTRAN format
```

If cform is not given, JLP assumes that cform is the same as xform. All variables given in 'cvar' are read with one FORTRAN read statement. If xform is 'm' then cform is also assumed to be 'm' (i.e. cform command is not used to determine what subroutines JLP calls, it can be used to carry information to data access subroutines (SYS.DEP.)).

see also: cdata, cvar, xform
;

***command line** (not cmd) - Syntax of a command line.

A command line can contain spaces and tabs. The 'command' in a command line is the initial nonblank part of the line. Commands must be in lower case. If the last character of a line is '>' then the logical command line continues to the next physical line (record). Commands can be read from terminal (or input stream in batch mode) or from files using 'include' command. Command line starting with '*', '!', or ';' are comments, and the rest of line following '!' is also a comment. If a file or a part of file is included using 'list' command, then all lines are treated as comments. Names of commands are checked as long as the name is uniquely determined (usually four characters are significant). The rest of the command name is ignored. In options (e.g. '/all') usually only the first character is significant, except for the negations '/nooptio' where usually three characters are significant (i.e. 'no'+ the option character). The '/' options must follow command name without

a space. If the significant part of a name is longer than usually, it is indicated in this help file. A paragraph is section of command lines starting with a command and ending with '/' (e.g. problem paragraphs and transformation paragraphs). Note: 'show' command starts a paragraph only with '/dom' option.

see also: include, list, help
;

***constants** - Gives values for (constant) variables.

Usage: variable_list= value_list ! These constants can then be used in transformations. Example:

```
const price1,-price4=2,2*3.1,4
```

If dtran-, ctran-, or xtran- transformations calculate sums over data files, units or schedules, initial values (usually zeros), can be given with 'const' command (or using appropriate 'if ... then var=0' transformation). Current values of constants (or any variables) can be seen with 'values' command. Xdat creates automatically constants from the file names given in 'xdat' command.

see also: values, variables, xdat
;

***ctran** - Starts paragraph defining transformations made for c-variables.

Ctran-transformations are made in order to get variables that can be used as parameters in xtran-transformations or to define domains in problem definitions. The default is that all output variables are stored. If not all output variables need to be stored, then variables stored when data are read in are given in 'keepc' command, and output variables to be stored in later transformations should be given in 'make' command.

Examples:

```
ctran
if distance.gt. 200 then
 harvestcost=2
else
 harvestcost=1.5
end if
/
```

see also: transformations, xtran, make, keepc
;

***cvar** - Defines c-variables that are read from cdat files.

Usage: cvar variable_list ! Cvar list must include at least 'ns' which tells the number of schedules in each file. The default is that all cvar-variables are stored in cmat matrix. If only a subset needs to be stored, the stored variables are given with 'keepc' command.

Example:

```
cvar c1,-c6,ns
```

see also: keepc, xvar, variables
;

***dir** (not cmd) - How to define directory for input data?

Directory of data files can be given with 'path'

```
see also: path, files
;
```

***do** - Starts a loop.

A loop ends with 'end do' or 'enddo'.

Usage:

```
do n
 (commands)
end do    !or enddo
```

Example:

```
solve        ! The RHS counter must be initialized before using 'solve +1'.
do 10
solve +1     ! If there are not enough RHS's, iteration terminates
             ! without error.
end do
;
```

***domain** (not cmd) - Subset of units used in problem or report.

A domain is a subset of units that can be defined with d- and c-variables. Domains can be used in 'problem' paragraph or in 'show/domain' paragraph.

Examples:

```
data=Savo & owner=private:
unit=23:
all:
```

```
see also: problem, show, variables
;
```

***dtran** - Starts a paragraph defining transformations made for d-variables.

Usage:

```
dtran
(transformations)
/
```

Dtran-transformations are made in order to get variables that can be used as parameters in ctran- or xtran-transformations or to define domains in problem definition. When data are read in, dtran-transformations are made always as JLP starts reading new cdat- and xdat- files. In later transformations, JLP remembers the original file sturucture, and dtran-transformations are made when first unit of a cdat- and xdat- file is in turn. Dtran-transformations remain in effect also when using data saved in JLP format. The automatically created 'data' variable can be used in transformations.

Examples:

```
dtran
if data=NorthKarelia then harvestcost=2
else harvestcost=1.5
/
```

If you have given xdat-command 'xdat north.xda, south.xda' you can make dtran-transformations as:

```
if data=south then ...
```

Output variables of dtran-transformations are not stored anywhere, they are just computed again when needed, and new transformations are appended to

previous ones. All dtran transformations can be cleared (unlike computed ctran or xtran transformations) as follows:

```
dtran
clear
/
```

see also: xtran, transformations, variables, xdat, save
;

***duplicate** - Defines transformations describing duplication of schedules.

Usage:

```
dupl
(transformations)
/
```

If dupl- transformations determine a nonzero value for variable 'duplicate' in a schedule, then JLP makes that many NEW copies of the schedule. The total number of copies will thus be duplicate+1. Thereafter JLP makes xtran-transformations for each copy. Before computing xtran-transformations, JLP assigns the number of the copy (starting from zero) to the variable 'duplicate'.

Example:

```
dupl   ! duplicate all schedules with clearcutting during first period
if clearcut.1.gt.0 then duplicate=1
/
xtran   ! separate manual and harvester clearcuttings
if clearcut.1.gt.0 .and. duplicate=1 then
manpower=10*cutvolume.1
harvestertime= 2.5*cutvolume.1
else
manpower=70*cutvolume.1
harvestertime=0
end if
/
```

Thereafter there can be problems with constraints for manpower and harvestertime even if original data did not separate the two harvest method.
;

***end** - (SYS.DEP.) Return to the main program.

The standard main program prints the output buffer, and stops. In user JLP-implementation 'end' can be used to get the control to interface level.
;

***end do** or **enddo** - End of the do loop

see also: do
;

***feasible** - Finds a feasible solution.

The syntax for selecting RHS is the same as for 'solve'. If the solution is thereafter asked with 'solve' command, JLP starts directly from the feasible solution. If no objective was defined in 'problem' paragraph, JLP finds a feasible solution also with 'solve' command.

see also: problem, solve
;

***files** (not cmd) (SYS.DEP.) - Opening of old or new files.

The way new files are created is determined by $LIST and $VERSIONS options in file *jlp.par*. The $LIST option determines the carriage control keyword used when opening output ASCII files. If $VERSIONS option is set, then JLP adds a '_(nro)' -version number when opening a new file with the same name as an existing file. $READONLY option specifies an possible nonstandard keyword when opening existing files for reading. See file *jlp.par* for more details.
;

***help** - How to get on-line-help?

Usage: help ! List '*' lines in this help file.

help keyword ! List the help module for keyword. Keywords that are not commands are followed by (not cmd). A keyword must be written so far that it can be uniquely determined (first match is always printed). Modules or features that are dependent on the implementation of JLP are indicated by (SYS.DEP.). The system manager should edit these modules. If the significant part of the command is longer than 4 characters, it is indicated.

Current commands (significant part underlined):

batch	buff	buflevel	cdata	cform
const	ctran	cvar	do	dtran
dupl	end	enddo	feasible	help
helpfile	include	init	keepc	keepx
make	mrep	outfile	outlevel	own1
own2	ownread	parin	parout	path
pause	printlevel	problem	read	recall
report	save	sched	show	solve
split	system	time	title	unsave
values	write	xdata	xform	xtran
xvar				

(SYS.DEP.): own1 and own2 are replaced by commands given in jlp.par.

```
help         is equivalent to:      list/all jlp.hlp/*
help key     is equivalent to       list jlp.hlp/*key:;
```

see also: helpfile, command line, list
;

***helpfile** - Changes the help file.

Usage: helpf file !(5 characters required: 'helpf') New helpfile is 'file'. The default is *jlp.hlp*. Command 'helpf' without *file* makes *jlp.hlp* the current help file. The help file contains cells starting with '*keyword' and ending with ';'. The user can freely edit the help file.

see also: help
;

***include** - The command interpreter will read commands from a file.

Usage:

```
include  filename          ! The whole file is included
incl file/addr1:addr2      ! The first line included starts with addr1
                           ! and the last line included starts with addr2.
incl file/addr             ! Only the line starting with addr is included.
```

```
incl/all file/addr1:addr2 ! All matching sections are included.
incl/a  file/addr         ! All lines starting with addr are included.
incl ?opt.sav ?opt.def    ! If file 'optdat.sav exists then include it.
                          ! If it doesn't, include opt.def.
```

Include-files (including list-files) can be nested to 6 levels.

see also: list
;

***init** - Gets a fresh start.
;

***integer approximation** (not cmd)

If '/integer' option of the 'show' command is in effect, JLP computes the values of x-variables resulting when for each unit only the schedule with largest weight is applied.

see: show, recall
;

***keepc** - Defines c-variables to be stored in memory when reading data.

Usage: keepc variable_list ! (5 characters significant: 'keepc').

When data are read in, the default is that all cvar-variables (variables read from cdat-files) and output variables of ctran-transformations are saved in the memory. If only a subset of those variables are needed, give them with keepc-command. See 'make' for storing output variables of ctran-transformations defined after reading data.

see also: cvar, make, variables
;

***keepx** - Defines x-variables to be stored in memory when reading data.

Usage: keepx variable_list ! (5 characters significant: 'keepx')

When the data are read in, the default is that all xvar-variables (variables read from xdat-files) and output variables of xtran-transformations are saved in the memory. If only a subset of those variables are needed, give them with keepx-command. See 'make' for storing output variables of xtran-transformations defined after reading data.

see also: xvar, make, variables.
;

***list** - Lists files or parts of them.

List will be used exactly as include-command except all lines are treated as comments.

Usage:

```
list  filename            ! The whole file is listed
list file/addr1:addr2     ! The first line listed starts with addr1
                          ! and the last line listed starts with addr2.
list file/addr            ! Only the line starting with addr is listed.
list/all file/addr1:addr2 ! All matching sections are listed.
list/a  file/addr         ! All lines starting with addr are listed.
```

Short headers of source files (e.g. jlpsub.src etc.) can be listed as follows:

list/all jlpsub.src/*=:**

Longer headers can be listed:

list/all jlpsub.src/*=:***

The short header of a named module (e.g. 'ilfind') can be listed:

list jlpsub.src/*=ilfind:**

On line help is implemented using 'list' command.

see also: include, help
;

***make** - Makes new variables when data are already read in.

Usage:
make ! Make all defined new variables.

make variable_list ! Make only variable_list variables.

New variables are defined in ctran- or xtran-transformations. The default is that the output variables of transformations defined after reading data are stored, except special x-variables 'split' and 'duplicate'. If no variable_list is given, all new output variables are stored. Variable_list should contain both c- and x-variables that are needed later. If all new variables are stored, then 'make' is not necessary: new variables are automatically created at 'problem' or 'solve' commands.

see also: dtran, ctran, xtran, keepc, keepx, save
;

***mela** (not cmd) - MELA/ JLP relation.

For JLP, Mela is an implementation of the special data format 'xform m', and report generator implemented behind command 'mrep' or option 'show/mrep' command. For more information, consult Markku Siitonen, The Finnish Forest Research Institute.
;

***mrep** (SYS.DEP.) - Calls user's own special report writer.

If JMAKE option $MREP in *jlp.par* is in effect, JLP calls subroutine 'mrep'. If option 'show/mrep' is in effect, then report is always generated with this generator instead of the standard report writer. For JLP, user report generators 'report' and 'mrep' work exactly in the same way. It is intended that 'report' could be a general report writer and 'mrep' a report writer associated with 'xform m'.

see also: show, recall, solve, report
;

***outfile** (SYS.DEP.) - Opens (or closes) a file for additional output.

Usage: outfile File ! Open output file 'File'
outfile ! Close the current output file.

Depending on the $VERSIONS option in file *jlp.par*, file may be opened with version number added to the name. Option:

outf/s ! (SYS.DEP.) Additional output is written to unit NUOUT defined in *jlp.par* without opening the file first. It is assumed that the file is/will be opened by the operating system or the main program.

see also: outlevel, printlevel, write, files
;

***outlevel** - Amount of output to be written to outfile.

Usage: outlevel i ! where i is:

```
0     = no output to outfile (default)
1     = only solution and problem definition
2 - 8 = more and more output
```

The outlevel-parameter works exactly as printlevel-parameter, except currently the optimization algorithm prints information about how the optimization proceeds only to the terminal (controlled by 'printlevel').

see also: outfile, printlevel, buflevel, buff
;

***ownread** (SYS.DEP.) - Replaces terminal input with an own subroutine.

When JLP would normally read commands from input terminal, it will instead call the user subroutine ownrea. Input from include files is not affected. This may be useful when building an own interface.

see also: buff, buflevel
;

***own1** (SYS.DEP.) - Executes a user defined command.

When getting a command with a name given in $OWN1 option in *jlp.par*, JLP will call user subroutine own1 that may do something useful.

see also: own2
;

***own2** (SYS.DEP.) - Executes another user defined command.

Command name is given in $OWN2, and subroutine own2 is called.

see also: own1
;

***parin** - Lists and defines input parameters of optimization.

JLP lists first current parameters. Thereafter JLP expects a paragraph defining new values for input parameters of the LP-algorithm. User may wish to change the 'invert' parameter that tells how often the basis is reinverted. Other parameters are of interest when there are difficulties in the optimization.

Examples:

```
parin
invert=200 ! reinvert the basis after 200 changes.
/
```

```
parin
/ ! List only parameters, '/' ending the paragraph is necessary.
```

see also: parout
;

***parout** - Lists output parameters of the JLP optimization algorithm.

The output parameters are related only to the technical details of the optimization.

see also: parin

;

***path** - Defines directory for input data files.

Usage: path directory ! Directory is added to cdat-, xdat-, and unsave-file names. Note that command line

path <inventory.dat>

means that the command line continues to the next line. The command line ends where it is intended if '!' is put to the end of line:

path <inventory.dat> !

If JMAKE option $READONLY is properly defined in *jlp.par*, the directory can belong to another user who has granted reading rights (SYS.DEP.)

see also: xdat, cdat, unsave
;

***pause** - Pause until <ret>.

May be useful for following the program flow when commands are read from include files. Has no effect in batch mode.

see also: batch
;

***printlevel** - Determines the amount of printed terminal output.

Usage: printl i ! i= 0 or 1 or 2 or ..... Larger values of i indicates more output:

0 = no output
1 = only solution and problem definition
2 - 8 more and more output

see also: outlevel, outfile, buflevel
;

***problem** ! Starts problem definition paragraph.

Examples:

```
problem
x1>0        / > 110 />200 />300    ! defines several lower bounds
x2>0   / >120 <200                  ! both upper and lower bound
x3+2*x1-x3 >0     / >130            ! linear combination of x-variables
zvar-2*x3=0                         ! zvar is a z-variable
x4=0 / =140
x5>0 / >150
x6-x2 max                          ! objective row, max or min
pml=1: pml=2:       !defines domains for the following constraints
x1,-x3, x5 =0 /=100 ! the constraint can be defined for a variable list
/                   ! End of problem definition.
```

The domain specifications can be made using constants, d- and/or c-variables. There can be any number of domain specifications, domains need not be hierarchical. Either object variable or constraints may be missing. If there is no object variable, JLP just finds a feasible solution with 'solve'. The objective row can be anywhere. If the problem is solved with 'solve/m', then the user subroutine 'next' is used to compute actual RHS (SYS.DEP.). NOTE: After reading a problem paragraph, JLP computes the smallest and largest possible value for each row. Thus 'problem' can be used to compute the range of x-variables without solving the problem.

see also: solve, feasible, variables
;

***read** ! Reads data.

JLP reads data automatically at 'problem' command if data are not yet ready. Before reading data you must give xdat-,xvar-, cdat-, cvar- commands, and ctran- and xtran- transformations, if needed.

see also: keepc, keepx
;

***recall** - Prints again the last solution.

If you have made new variables and/or you have used commands 'printlevel', 'outlevel', 'buflevel' or 'show' then you will get more or less output than when you solved the problem. If option 'show/noxf' is in effect, the other options of 'show' are used only with 'recall'.

see also: printl, outl, bufl, show
;

***reject** (not cmd) - Rejecting schedules in the optimization.

Example:

```
xtran
if unit.eq.2.and.s.eq.3 then reject ! Reject schedule 3 in unit 2.
if herbicide>0 then reject          ! Reject schedules using herbicides.
/
```

Rejected schedules are ignored in the optimization, they are not deleted from the data. Rejection of schedules is implemented with a special variable 'reject' which gets value -1 for rejected schedules and value 0 for accepted schedules. If schedules are rejected for the first time during the session, the default is that all schedules are accepted. If schedules have been rejected earlier during the session, and later xtran-transformations define rejections, then the schedules rejected earlier remain rejected unless the new xtran-transformations specify the acceptance explicitly by giving value 0 for the variable 'reject'. Example:

```
xtran
reject=0                    ! Cancel earlier rejections.
if biocontrol>0 then reject ! Reject schedules with biological weed control.
/
```

It is possible define constraint that a variable needs to be always e.g. zero using a constraint in the problem paragraph, e.g.:

```
problem
herbicide=0
...
```

Rejection with 'reject' is computationally more efficient.
;

***report** ! (SYS.DEP.) Calls own general report writer.

Calls subroutine 'repo' provided by the system manager. If option 'show/repo' is in effect, then report is always generated with this generator.

see also: mrep
;

***save** - Saves the data in JLP format.

Usage: save File ! If file is not given, JLP uses name:'jlpsave'. If there are unsaved data, then data are saved immediately. If the data are not yet read in or are already saved, then saving is done when reading the data or creating new variables.

Options:

save/later ! If there are unsaved data, the saving is done after the next transformations. This may be useful when the data exceed the memory reserved so that JLP knows to write the data directly into a named file instead of a scratch file.

```
Save makes 3 files:
file.xdj = x-data file
file.cdj = c-data file
file.sav = the text file containing data definitions.
```

The saved data and all definitions can be read in using: 'include file.sav'

see also: write, saveform
;

***saveform** (not cmd) - The JLP format of saved files.

The file *file.sav* contains const-,xdat-, keepx-, keepc-, dtran- and unsave-commands that are needed to read in data stored in JLP-format. The file contains also the history of the file as comments.

The first record of *file.cdj* is:

ml, nht,maxrec,nfiles,(mll(i),i=1,nfiles) ,

where

```
ml     = the total number of units
nht    = the total number of real*4 variables in xdata
maxrec = all records in file.xdj and file.cdj contain less than
         maxrec numbers (4 bytes each)
nfiles = number of xdat and cdat files used to make save-files.
         saved files still differentiate original xdat- and cdat files
mll(1)...mll(nfiles)= number of units in each original file
```

The next records contain all saved c-variables. The first variable in each record is integer*4 variable telling the length of the rest of the record (i.e. read()n, (cmat(j), j=iprev+1,iprev+n)). JLP packs cdata in as large record as the MAXREC parameter in *jlp.par* allows, but when reading the data no special structure is assumed except that variables of each unit are in order given by keepc-command.

Each records of *file.xdj* contains first the number of xmat-numbers stored in the record. A record contains only complete units. JLP packs as many units as MAXREC allows into one record, but on input records can contain less units.
;

***schedules** - Prints weights and shadow prices of schedules and units.

Can be used after solving a problem. The command has the following options:

```
sched        ! Print all basic schedules (schedules used in the solution).
sched n      ! Print at most n schedules.
sched/all    ! Print also shadow prices of nonbasic schedules.
```

```
sched/all n  ! Print at most n schedules (basic + nonbasic).
sched/all>95 ! Print all schedules whose shadow price > 95% from the
             ! value of the basic schedules of the unit
sched/a>95 n ! Print at most n such schedules
```

The system manager may put these printings under 'mrep' or 'repo' (SYS.DEP.).
;

***show** - How the x-variables are printed after solving a problem.

Usage: show(/options) (variable_list)

Options:

```
  /nox      ! Print no x-variables.
  /noxfirst ! Print no x-variables automatically after solution,
            ! print x-variables information only with recall command
            ! using current show options
            ! Significant part of the option is nonstandard: /noxf
  /xfirst   ! Negate the /noxf option.
  /all      ! Print for each domain all x-variables (default).
  /prob     ! Print for each domain all x-variables used in problem
  /notwice  ! Do not print x-variable if it is on a constraint
              row alone (i.e. without z-variables or other x-variables,
              and thus the value can  be seen from output for rows).
  /twice    ! Print also duplicate information (default).
  /cost     ! Print cost of decrease and increase for x-variables (default).
            ! Computation of costs may take much time.
  /nocost   ! Costs are not computed.
  /inte     ! Print for x-variables the integer approximation obtained by
              using in each unit the schedules with largest weight.
  /nointe   ! Do not print the integer approximation (default).
  /domain   ! Start paragraph that defines domains that are used when
              computing x-variables (in addition to domains used in the
              problem). The domains are added to the domains given
              with earlier /domain definitions. The previous domains
              are first cleared if /nodom is also used (i.e. /nodom/dom).
              Domains are defined in the same way as in problem paragraph
              (remember ':' at the end), but only one definition per
              line is allowed.
  /nodom    ! Do not use extra printing domains.
  /repo     ! (SYS.DEP.) Use report generator 'repo'.
  /norepo   ! Do not use report generator 'repo' (default).
  /mrep     ! (SYS.DEP.) Use report generator 'mrep'.
  /nomrep   ! Do not use report generator 'mrep' (default).
```

show varlist ! print for each domain all x-variables in problem + varlist-variables.

New 'show' options will be in effect in the next 'solve' or 'recall' command.

Several options can be in the same 'show' command (e.g. show/repo/mrep/pro).

If the standard report writer is bypassed (i.e. '/repo' or '/mrep' or both are in effect), then other options (except '/twice') determine what quantities are available in user report generator.

see also: recall, mrep, report, solve
;

***solve** - Solves an LP-problem.

Usage:

```
solve    ! Solves the problem corresponding to first right-hand side.
solve 3  ! Solves the problem corresponding to third right-hand side.
         ! If for a constraint there are not 3 RHS's, the last one is used.
         ! If no constraint contains 3 RHS's, return to read new commands.
solve +1 ! Solves the problem corresponding to the next right-hand side,
         ! useful in do-loops.
solve +3 ! Solves the problem with RHS: previous + 3
```

```
options:

solve/i  ! Initializes the vector of key schedules, useful if
         ! you compare the solution times using different options
solve/m(text) ! (SYS.DEP.) generate RHS with user subroutine 'next'
```

After solving the problem, the solution is automatically printed with the current options of 'show' command (unless 'show/noxf' is in effect). The solution can be reprinted with 'recall'. Schedules information can be printed with 'sched'.

see also: problem, show, do, recall, report, mrep, sched, feasible
;

***split** - Splitting a unit into parts.

A unit can be split into parts that inherit different schedules. How units are split is determined in xtran-transformations with variable 'split'.

Example:

```
xtran
if unit.eq.10.and. s.ge.3.and.s.lt.7 then split=1.25
if unit.eq.10.and.s.ge.7 then split=2.40
oldunit=unit   ! Old unit numbers can be saved this way.
olds=s         ! Old schedule numbers can be saved this way.
/
```

Now schedules 3-6 are put to part 1 that is 25% of the original unit. Schedules 7- are put to part 2 that is 40% of the original unit. The unspecified schedules 1 and 2 remain in the original unit (part=0), and their share is 100-25-40%= 35%. It is required that the variable 'split' gets consecutive values 1,2,.., and that not all schedules are put to these parts so that some schedules are left to the original unit (part=0). The default is that all x-variables stored in xmat matrix are multiplied with the corresponding share proportion. If only part of x-variables should be split among parts (e.g. a x-variable like 'harvestmethod' should remain unchanged), then the variables that should be multiplied with the share can be determined with command:

split variable_list

Alternatively, the variables that are NOT multiplied with the share can be given with command:

split/no variable_list

see also: xtran, duplicate
;

***system** (SYS.DEP.) - Sends a command to system level.

Usage: system command ! Executes the one-line FORTRAN statement given in JMAKE option $SYSTEM in file *jlp.par*. The example given in *jlp.par* can be used to send command to the system level in VAX-VMS (e.g.: 'syst dir' will print the current VAX directory).
;

***time** (SYS.DEP.) Measures time.

If $SECNDS option in *jlp.par* is in effect, 'time' prints the time from the first time-command and from the previous time-command. If $CPU option in *jlp.par* is also in effect, the cpu- time is also printed. The time for solving a problem and doing the after-solution computations is automatically printed.
;

***title** - Defines title used when printing results.

Usage: title text

Note that you can get text into output file by entering comments.

User report writer can use the title for any purpose (SYS.DEP.).
;

***transformations** (not cmd)

The syntax of transformations is basically the standard FORTRAN syntax.

For instance:

```
x5=sin(x2**2+sqrt(ln(x4-2)))
if x3+x2=4 .or. sin(x3)>0.5 then    ! parentheses are not necessary
x7=x5-7
else
x4=x3**2.2+tan(x5)
end if                              ! if ... then can not be nested
```

Arithmetic operations and functions:

```
"    = raise to integer power (-1)"2=1 | **   = raise to real power
abs  = absolute value                  | atan = arctan
cos  = cosine of radians               | cosd = cosine of degrees
exp  = exp                             | int  = integer part
log  = natural logarithm               | log10   = log base 10
mod(x1,x2) = remainder mod x2          | ran(x1)= random with seed x1
sin  = sinus (angle in radians)        | sind = sinus, angle in degrees
sqrt = square root                     | tan  = tangent, angle in rad
tanh = tanh                            | x1=swap(x2) = change x1 and x2
max(x1,x2,x4)= maximum                 | min(x1,2,x4)=  minimum
```

Logical functions:

```
.gt. .lt. .ge. .le. .eq. .ne. .and. .or. .not.
 >     <   >=   <=   =          &
```

Current 'own' functions:

npv(interest_percent,income1,time1,...,incomen,timen) = net present value

see manual for %-loops and special uses of 'then' and 'else'
;

***unsave** - Gets data from saved JLP files (not generally needed).

Usage: unsave cdat xdat

If JLP has saved data in JLP format at 'save' command, then JLP automatically creates the correct 'unsave' command into the '.sav'-file.

User is not expected to give this command explicitly, implicitly this command is implied by: 'include file.sav'.

see also: save, path
;

***values** - Prints current values of variables.

Usage: values variable_list

Variable_list may contain d-, c- and x-variables and constants. For d-, c- and x-variables the value is for the last xdat-file, for the last unit, and for the last schedule, respectively. This command can be used to see the current values of constants, or to print the results if transformations are used to compute summary information over files, units, or schedules.

see also: constant, dtran, ctran, xtran
;

***variables** (not cmd) - Description of data variables.

Data variables are constants, d-variables, c-variables or x-variables. Constants, d-, c-, and x-variables differ in the way how they get their values and how they can be used. Constants are given values by 'constant' command or are created by xdat-command (e.g. 'xdat south.xda, north.xda' creates constants 'south' and 'north' with values 1 and 2). D-variables get new values when the data file changes. A d-variable 'data' gets automatically the number of the data file, and other d-variables are defined by 'dtran'-transformations. C-variables (class variables) are read or made by 'ctran'-transformations for each calculation unit, and x-variables are read or made by 'xtran'-transformations for each treatment schedule. When data are read in or when making transformations, all variables are put in the same vector so that transformations can access variables from different levels. You should not use the same names for constants, d-, c- and x-variables. JLP does not check this. Constants, d-variables and c-variables can be used to define domains for constraints or domains for printed results (see 'show/dom'). Constants and d-variables can be used as parameters in 'ctran'-, and 'xtran'-transformations, and c-variables can be used as parameters in 'xtran'-transformations. C-variables need to include variable with name 'ns' which tells the number of treatment schedules for each unit. Variable names must start with a letter A-Z or a-z (not with 'ÄÅÖäöå') and cannot contain characters '!"=*/:%-' (allowed characters include e.g. '#' and '.'). A list of variables is formed by separating variable names with commas. A list (sublist) of several variables with consecutive variable names can given with a ',-' construction:

var1,-var125,costa,-costx

The predefined variables are:

```
data      = the number of data file to be read in (d-variable)

unit      = the number of calculation unit (c-variable)
ns        = number of schedules in unit  (c-variable)
s         = the number of the schedule (x-variable)
duplicate = x-variable used to duplicate schedules (see: duplicate)
split     = x-variable used to split units into part (see: split)
```

```
reject    = x-variable having value -1 for rejected schedules (0 otherwise)

see also: cdat, cvar, ctran, xdat, xvar, dtran, constant
;
```

***write** - Writes the current xmat and cmat-matrices.

Usage:

```
write file          ! Writes xmat to binary file file.xdb and cmat to
                      binary file file.cdb
write/*  file       ! Write xmat to file file.xda and cmat to file
                      file.cda with free format
write/(8f8.0) file  ! writes xmat and cmat to files file.xda and file.cda
                      using FORTRAN format
```

Stored c-variables of one unit are written with one write-statement (i.e. into one record unless implied otherwise by the format), and stored x-variables of each schedule are written with one write statement. Names of written variables are printed (to terminal/outfile/buffer).

The current data can be saved also with this 'write' command, but the user needs to give the proper commands for reading the data again (compare with 'save')

see also: values, save
;

***xdata** - Gives the names of x-data files.

Usage: xdata file1,...,filen

It is recommended that names of text files end with '.xda', names of binary files end with '.xdb'. Files saved in JLP format end with '.xdj'. Directory specification given by 'path' command is automatically added to the name. Constants with names file1,...,filen (exluding extension) are created and given values 1,...,n. JLP keeps track of original file structure with variable 'data'. Thus transformations may contain 'if data=file1 then' statements and domain specifications contain 'data=file1' parts.

If xform = 'm', then the user subroutines may interpret names file1,... without a connection to physical files (SYS.DEP.).

also: xform, xvar, cdata, save, cvar, path
;

***xform** - Defines the format for reading xdat files.

Usage: xform form ! where

```
form =  *     if xdat files can be read with FORTRAN '*' format
        b     if x-data are in binary files
    (8f10.0)  any FORTRAN format
        m     data are read with user subroutines:
              minit, mgetc, mgetx, mfinit  (SYS.DEP.)
```

All variables given in 'xvar' are read with one FORTRAN read statement.

see also: cform, write
;

***xtran** - Defines transformations made for x-variables.

Xtran-transformations are made in order to get variables that can be used in problem definitions. For each schedule, the values of constants, d-, and c-

variables can be used. The default is that all output variables are stored (except 'split' and 'duplicate'). If not all output variables should be stored, then variables stored when data are read in are given in 'keepx' command, and output variables to be stored in later transformations should be given in 'make' command. Xtran- transformations are computed into the xmat-matrix, and they cannot be cancelled.

Examples:

```
xtran
cost=harvestcost*harvestvolume
/
```

If linear transformations are needed in LP-problems, they can be specified either in xtran transformation or written explicitly in problem paragraph. E.g. the following two problems are equivalent:

```
1)
  xtran
  diff.1=income.2-income.1
  /
  prob
  diff.1>0
  ...
  /

2)
  prob
  income.2-income.1>0
  ...
  /
```

If same linear transformations are used in several problems, it is more efficient to do them just once in xtran transformations. On the other hand, if linear transformations are written explicitly in 'problem' paragraph, JLP can compute the shadow prices of the element x-variables ('income.2' and 'income.1' in the above example).

Xtran-transformations are used for splitting a unit into parts (see 'split'), and for rejecting schedules in optimization (see 'reject').

see also: transform, keepx, ctran, make, split, duplicate, problem
;

***xvar** - Defines x-variables to be read in.

Usage: xvar variable_list

The default is that all xvar-variables are stored. If only a subset needs to be stored, the stored variables are given with 'keepx' command.

Example:

```
xvar income.1,-income.6,volume.1,-volume.6
```

If variable with name 'reject' is among xvar-variables, it is interpreted to indicate schedules that are rejected. Value -1 means that the schedule is rejected and value 0 that it is not rejected. The values of 'reject' can later be changed with xtran transformations.

see also: keepx, cvar, variables, xtran, reject
;

```
** end of file jlp.hlp ***
```

4. SETTING UP THE WORKING ENVIRONMENT

This part describes how the system manager (called here 'user') can build an executable program from the source files provided and install JLP into a larger management planning system.

4.1 Building JLP

4.1.1 Compiling and linking JLP (file readme.jlp)

File *readme.jlp* contains information what files are included, and how to set up the working environment using JMAKE precompiler. File *readme.jlp* will be updated to correspond changes made after the printing of this manual. Current content of *readme.jlp*:

```
********************* File readme.jlp:
This file includes general information about:

A. Files included in the JLP-package

and how to:

B. Compile and link JMAKE
C. Make own interface subroutines
D. Modify file jlp.par
E. Run JMAKE
F. Compile and link JLP
G. Test JLP
and
H. Revisions of JLP after June 1, 1992.

A. Files included in the JLP-package
====================================

The following files are included in JLP-package (on DOS or Macintosh
diskettes, file names are always in lower case):

1: readme.jlp  - this file

2: jmake.f     - source for the JMAKE precompiler

3: jlp.par     - file containing system options and data parameters

4: jlp.hlp     - help file for on-line help and reference

          JLP source files

5: jlp.src     - file containing main program and interface subroutine
6: jlp2.src    - subroutines accessing common data areas
7: jlpsub.src  - general subroutines
8: jlpopt.src  - optimization subroutines
9: jlpint.src  - templates for interface subroutines

          Test files:

10: test.in    - commands for a test run, use:"include test.in"
```

```
11: test.xda   - x-data for test
12: test.cda   - c-data for test
13: test.out   - output from the test run
```

B. Compiling and linking JMAKE

1) Edit in the first program line of the file jmake.f the values of parameters n5 and n6 according to the system defaults:

```
* n5= unit for terminal input
* n6= unit for terminal output
      parameter (n5=5,n6=6)
```

2) Change the extension ".f" of the file jmake.f if it is more convenient in your system.

3) Compile jmake.f

4) Link jmake

C. Make your own interface subroutines

File jlpint.src contains templates for interface subroutines. If user specific interface subroutines are needed, then the user should make own versions of the subroutines into a different file.

D. Modify file jlp.par

Edit the file jlp.par, and save it with a different name if you want to keep original jlp.par unchanged. The file jlp.par contains information about the system specific features and size parameters for declaring variables and vectors of JLP. File jlp.par contains three types of parameters. Parameter lines starting with "$$" give general information for the JMAKE precompiler. Lines starting with "$" tell how certain system dependent features can be included in the programs. Other noncomment lines (lines starting with "*" are comments) are parameters for defining FORTRAN parameters, variables and vectors.

E. Run JMAKE

Run then program JMAKE that creates final source files (with file name extension given with parameter "$$EXT"in jlp.par). If the default directory already contains a file with the corresponding name, JMAKE asks if the file should be replaced. If answer "Y" is given, then JMAKE tries to write the new final source file, and an error occurs in some systems (e.g. OS/2), or the old file is just replaced (e.g. in Macintosh), or the new file will be the newest version of the file (e.g. in VAX/VMS).

JMAKE creates files (assuming that $EXT -parameter in file jlp.par is F):

```
jlp.f
jlp2.f
jlpsub.f
jlpopt.f
jlpint.f  - if not removed from $FILES statement in jlp.par.
+ other files specified in $FILES statement in jlp.par.
```

F. Compile and link JLP

Compile program files created by JMAKE and other files not precompiled with JMAKE.

Link. If you have written your own main program, that file must linked before the object file resulting from jlp.src. The interface subroutines replacing templates in jlpint.src must be linked before jlpint.
If parameters in file jlp.par are changed, run JMAKE again. It is safest to let JMAKE precompile all files again, even if not all files are generally affected by changes of parameters in jlp.par.

```
G. Test JLP
===========

Copy JLP program and test files in the same directory. Run JLP for a test
problem: give as first JLP-command:

incl test.in/*:*

The output should look similar to to contents of file test.out.


H. Revisions of JLP after printing of the manual
================================================

This section will tell what changes are made in JLP package after printing of
the manual.

******** end of file readme.jlp
```

4.1.2 Parameter file *jlp.par*

The programs are written trying to follow the FORTRAN-77 standard. Some common nonstandard features are useful. Options of the JMAKE precompiler determine if nonstandard features are included, and what is the syntax of the nonstandard features. All system specific features and editable size parameters for declaring variables and vectors of JLP are transmitted in file *jlp.par*.

Current contents of *jlp.par* as used in Language Systems FORTRAN 3.0 running in Macintosh Quadra 700:

```
******   file jlp.par
**** user can edit only the right-hand sides of the parameters
*
* Precomiler parameters
* =====================
*
$$EXT = .f           ! File name extension for source files, e.g.
*                      ".f",".for" or ".ftn".
$$! = T              ! Compiler interprets text after '!' as a comment.
*                      This parameter has effect only in lines
*                      generated by JMAKE.
$$DOUBLE = DOUBLE PRECISION
*                      Data type used in calculations, e.g. REAL*10.
*                      Precision less than real*8 is not recommended.
$$SOLTYPE = DOUBLE PRECISION !
*                      Data type for accessing the results.
$$DEFINITIONS=jlp2.src
*                    ! Files containing global definitions,
*                      user can/must change the first file only if
*                      JMAKE is used to precompile other programs. If own
*                      files included, separate with commas, e.g:
*                      $$DEFINITIONS=jlp2.src,owndef.src
*                      The definitions can be at the beginning of ordinary
*                      source file (as jlp2.src is) that is also precompiled
*                      with JMAKE.
$$FILES = jlpint.src
*                      User source files that use global JLP variables or
*                      variables defined in user
*                      or JMAKE filtering options. Initially file jlpint.src is
*                      included here. If user subroutines replace all the
*                      subroutine templates there, remove jlpint.src.
*                      If several files separate with commas,e.g:
```

```
*                     $$FILES =  FILE1.SRC,FILE2.SRC
*                     If all files do not fit to one line,
*                     give several $$FILES -lines.
*                     If there are no files put '*' as first character:
*                     *$$FILES (possible if JMAKE is used for other programs)
*
*
* Options
* ========
*
* Option is in effect:              $OPTION = T
* Some options require additional information.
* In that case the syntax is:       $OPTION = T = TEXT
* Option is not in effect:          $OPTION = F (= TEXT)
*
*options in JLP:
*
$READONLY = T = READONLY
*                     ! Keyword in OPEN statement used by JLP to open
*                       files for reading. In multiuser systems, a user
*                       with reading rights can acces files in other users'
*                       directories (e.g. in VMS this allows also a shared
*                       acces). It is always safer to open files with this
*                       option, as it prevents accidental modifications
*                       of files. This keyword is nonstandard Fortran.
*                       In IBM Fortran/2 this option would be
*                       ACTION=READ.
*
$LIST =  T = LIST    ! If this option is in effect, JLP opens output
*                       text files with the nonstandard keyword
*                       CARRIAGECONTROL='(text)'
*                       In some systems one may have trouble with the
*                       carriage control characters of standard Fortran
*                       text files (e.g. a program may  read in characters
*                       you don't see in the editor or printing).
*
$SUPPRESS = T = $    ! Format that suppresses carriage return in output.
*                       This is used to print prompts (e.g. 'jlp>') that
*                       indicate that JLP waits for input.
*                       In IBM Fortran/2 this format is: \
*
$VERSIONS = T        ! Different systems work differently when a program
*                       tries to create a new file with name of an existing
*                       file (e.g. VMS creates new version of the file,
*                       some systems just delete the old file, and in
*                       some systems an error occurs). If this option is
*                       in effect JLP creates version numbers when creating
*                       new files. For instance if JLP should open a new
*                       file with name "output.jlp" and a file with
*                       that name exists, JLP opens the file with name
*                       "output_2.jlp". Parameter MAXVER given below
*                       determines the maximum number of versions.
*
$SYSTEM = F =  call lib$spawn(inp(ial:lop))
*                      This option (if in effect) tells what JLP should
*                      do at JLP command 'system'. Character variable
*                      inp contains the command line, ial is the first
*                      nonblank position after 'system ', and lop is
*                      the last nonblank character of the command line.
*                      In VMS 'call lib$spawn(inp(ial:lop))' sends the
*                      command line after 'system ' to the system level
*                      (e.g. JLP command 'system dir' then prints the
*                      names of files in the current directory)
*
$OLDMFORM = F       ! Old versions of subroutines for reading data are
*                      included. These subroutines are used  used when
*                       'xform m' is given as the format.
*                      See the manual for more details.
*
$MREP = F           ! Own report generator subroutine MREP included.
*                      Invoked by jlp-command 'mrep'. See the manual.
*
```

```
$SECNDS=T= SECNDS(0.)
*                    Timing function available in the system. The function
*                    should return the elapsed time measured
*                    from any fixed point in any units.
*
$CPU =F =          ! Timing function  measuring time used by cpu.
*

$INIT1=F=          ! First JLP command executed when JLP starts.
*                    For instance, if data files are always in
*                    directory disk1:[data], this could be
*                    $INIT1=T= path disk1:[data]
*
$INIT2=F=          ! Second command executed when JLP starts. If
*                    more than two initialization commands are needed,
*                    the commands can be stored in a file, and
*                    included with INIT1 option, e.g.:
*                    $INIT1 =T = include init.in
$OWN1=T=own1       ! Command for calling user subroutine own1(inp,errors).
*                    File jlpint.src contains a template and more information.
$OWN2=T=own2       ! Command for calling user subroutine own2(inp,errors).
*                    File jlpint.src contains a template and more information.
*
$DUMP=F            ! This option is used for printing information
*                    for tracing errors in the optimization algorithm.
*                    Ordinary user should have this option always off.
*
* Parameters:
* ===========
*
*  If e.g. parameter MAXNX is too small, JLP gives an error message:
*  "*PAR* increase MAXNX"
*
N5 = 5         ! Unit for terminal input.
N6 = 6         ! Unit for terminal output.
*
MAXXMA=900000 !Size of the vector used for xdata.
*               If xdata exceed the memory reserved, JLP is slower.
*               So put MAXXMA as large as possible.
MAXCMA=2000   ! Size of the vector used to store c-data,
*               at least (number of units) * (number of c-variables)
*
MAXREC=8191   ! Maximum number of real*4 variables in one record
*               of an unformatted file. MAXREC has effect only if
*               parameter MAXXMA is so small that the whole data
*               can not be stored in memory, or when the data
*               is saved in JLP format using 'save' command.
*               Optimal value is dependent how the speed of
*               reading depends on the record size.If data does
*               not always fit to memory, MAXREC should be
*               at most 1/3 of MAXXMA, but probably e.g. 1/20
*               of MAXXMA is better. MAXREC should be at least so large
*               that one record can hold the x-data for any
*               calculation unit
*
MAXNX=100     ! Max. number of x-variables
MAXXS=300     ! Max. number of x-variables computed and printed after 'solve'.
*               If the integer solution is not printed, then this should be
*               (number of domains) x (number of x-variables printed). If the
*               integer solution is printed, then this should be twice as much.
MAXXDX=1700   ! If cost of decrease and increse computed,
*               this should be at least:
*       (# of domains) x (# of x-variables printed) x (# of basic x-variables)
MAXNR=40      ! Max. number of rows in a problem (area constraints
*               are not counted)
MAXNXP=60     ! Max number of x-variables in a problem definition,
*               rows including a single x-variable without
*               a coefficient are not counted.
MAXML=1000    ! Max. number of calculation units.
MAXMV=200     ! Max. number of shcedules in one unit
MAXSPL=20     ! Max number of parts in a unit when a unit is split
MAXSPT=100    ! Max. total number of parts in all split units
```

```
MAXCOM=30     ! Max. number of domain combinations
MAXDOM=30     ! Max. number of domains
MAXNC1=30     ! Max. number of c-variables
MAXNZ=50      ! Max number of z-variables
MAXDAF=30     ! Max number of data files used in xdat -  command
MAXVER=3      ! Max. number of file versions if $VERSIONS = T
*
* unit numbers used by jlp files (change if these conflict with
* unit numbers used in own subroutines):
*
NUSAVX = 33   ! Unit for saving data
NUSAV2 = 34   ! Unit for rewriting save file
NU1    = 35   ! Unit for several JLP files
NU2    = 36   !    "
NUOUT  = 37   ! Unit for additional output
*
* units for included files:
NF1 = 41
NF2 = 42
NF3 = 43
NF4 = 44
NF5 = 45
NF6 = 46
*
* units the user can use in own interface routines e.g.
* for reading data and report writer:
NUOWN1=51
NUOWN2=52
*
LCOMLI=600  ! Max  length of the command (including continuation lines)
LLINE=130   ! Max. length of a command  record
LPROBL=200  ! Max length of command line in problem-paragraph
*
**text buffers
*
LENINC =1640  ! Length of the input buffer 'INC'
LININC = 100  ! Max. number of lines of the input buffer
*
LENOUT =2640  ! Length of the output buffer 'OUT'
LINOUT = 100  ! Max. number of lines of the output buffer
*
LENDTR = 800  ! Length of buffer 'DTR' for d-transformations
LINDTR =  50  ! Max number of lines in the buffer for d-transformations
*
LENLOO = 400  ! Length of the buffer 'LOO' storing DO-loops
LINLOO =  60  ! Max. number of lines in the buffer
*
LENCON = 160  ! Length of buffer 'CON' for constant definitions
LINCON =  10  ! Max. number of lines in the buffer
*
LENPRO =1024  ! Buffer 'PRO' for constraint definitions (without rhs)
*               Max. number of lines = MAXNR
*
LENSDO=400    ! Buffer 'SDO' for storing show/domain definitions
LINSDO=40     ! Max. number of lines.
**end of text buffers
*
LVARNA=32   ! Length of character variables used for variable names
LFORM =130  ! Max. length of formats xform and cform
LFILNA=50   ! Max. length of file names
LDOMNA=40   ! Max. length of domain specifications
LPATHN=40   ! Length of character variables used for PATH
NDTRAN=300  ! Length of compiled d-transformations
NCTRAN=300  ! Length of compiled c-transformations
NXTRAN=300  ! Length of compiled x-transformations
NDUTRA=300  ! Length of compliled dupl-transformations
NSDTRA=200  ! Length of transformations defining show/domains
NINT  =50   ! Max. number of intermediate results in transformations
NPARA =100  ! Max. number of constants in transformations
NIDOUT=100  ! Max. number of output variables in d-transformations
NICOUT=100  ! Max. number of output variables in c-transformations
NIXOUT=100  ! Max. number of output variables in x-transformations
```

```
MAXRHS=5    ! Max. number of rhs's in problem-command,
*       note that  your own NEXT subroutine may generate more rhs's
LEVELP=3    ! Default value for printlevel
LEVELO=1    ! Default value for outlevel
*
* Own parameters can be added here. For instance, if you need
* additional unit numbers, it is a good idea to determine them
* here so it is easier to prevent conflicting numbers. See manual for
* how to use JMAKE in own subroutines. An example: integer
* parameters NUOWN3, NSIZ a real parameter DELTA, a double precision
* parameter DDELTA and character parameter TEXT can be defined by
* deleting '*' in the following lines:
*NUOWN3 = 77
*NSIZ   = 123
*DELTA  = 1.3 ! JMAKE assumes the first character convention of Fortran
*DDELTA = 1.2D0 ! Double precision parameters should include '.' and 'D'.
*TEXT   = 'Help, Help'
** JMAKE will generate the corresponding parameter statements,
**  if the program contains a section:
*needs:
*NUOWN3,NSIZ,DELTA
*DDELTA,TEXT
*end:
** end of file jlp.par
```

4.1.3 Features of standard FORTRAN not used

Some FORTRAN compilers do not implement all standard features. And some companies seem to interpret the standard differently. In order to avoid difficulties with less general compilers, the following features were not used:

- character and numeric data in the same common area
- alternative entry points in subroutines
- alternative return addresses
- same character variable on both sides of an assignment statement

4.2 Output Files in non-VMS Environment

New files are opened by **`save`**, **`outfile`** and **`write`** commands. Operating systems work in different ways when a program tries to open a new file with a name of an existing file. The VMS operating system just creates a new file with a new version number. In the UNIX operating system an error occurs. Using LS- FORTRAN in Macintosh the new file replaces (and thus deletes) the old file. If option `$VERSIONS` is set to `T` in file *jlp.par*, then JLP appends version '_n' to the file name (before file name extension) if there is a file with the given name. The version number will be one higher than the highest existing version. The first version does not have a version number. If the version number would be higher than `MAXVER` parameter given in *jlp.par*, then an error occurs.

For input files defined by **`cdat`**, **`xdat`** or **`unsave`** commands, JLP expects to get the full file names, i.e. JLP does not try to figure out what version might be in question. If data

are stored in the internal format using **save** command, then the **unsave** command is written into the '.sav' file with the correct version numbers.

4.3 Sending a Command to the System Level

While using JLP interactively, the user may need to interrupt the JLP session to do something at the system level (e.g. copy files). In a modern windows based operating system (e.g. in Macintosh), the system level can be accessed easily. If you are using a simple VAX-VMS terminal, you can set the following JMAKE option to `T`:

```
$SYSTEM = F =  call lib$spawn(inp(ial:lop))
```

Thereafter `system` command can be used to send the command line to the operating system:

```
system dir          ! get directory
syst edit file.in   ! edit file 'file.in'
```

In operating systems other than VMS, you may replace the call to `lib$spawn` with a call to another system routine. The argument '`inp(ial:lop)`' contains the command line after the **system** command.

4.4 Creating Own Timing Subroutine

In the version of *jlp.par* listed above it is assumed that the function `SECNDS` provided both by VAX FORTRAN and Language Systems FORTRAN is used for timing. If the system does not support `SECNDS` then you may make your own timing subroutine into a source file linked with JLP. For instance, in IBM FORTRAN/2, an corresponding timing function might be:

```
      function secs()
      integer*2 hh,mm,ss,hd
      call gettim(hh,mm,ss,hd)
      is=hh*3600+mm*60+ss
      secs=is+hd/100.
      return
      end
```

To use this function, change `$SECNDS` option into:

```
$SECNDS=T= SECS()
```

Elapsed time can be measured in JLP using **time** command. The time used in the optimization phase is also measured automatically. If `$CPU` option is in effect (and corresponding function provided), also elapsed cpu-time is measured.

4.5 Management of Programs with JMAKE Precompiler

JMAKE is a general purpose precompiler used to manage global parameters, global variables (stored in common areas), lengths of character variables and system dependent options.

JMAKE is case sensitive.

4.5.1 Accessing JLP global parameters and variables

JLP is designed so that all JLP subroutines and subroutines written by the user can access all global variables and parameters of JLP (henceforth term 'variable' is used to refer to both variables and parameters). Because the standard FORTRAN does not recognize global variables, JMAKE precompiler was made to manage global variables in a transparent way. JMAKE generates necessary definitions of variables and common areas for all variables that the subroutine needs.

Editable parameters are given in file *jlp.par*, and other global variables are in the file given in `$$DEFINITIONS` statement in *jlp.par* (currently in file *jlp2.src*). JMAKE precompiles all files given in `$$FILES` statement in *jlp.par* and all files listed in `$FILES` section in files given in `$$DEFINITIONS` statement in *jlp.par* (with this a little complicated system JMAKE can hide definitions that the user is not allowed to change). JMAKE generates definitions for global variables listed in 'needs:' sections of the file. A '`needs:`' section looks like:

```
*needs:
*KEEPCL,KEEPXL,LISTXS,TITLE,LIST,VNAME
*BATCH,INPUT,LEVEL,LEVEL2,LEVEL3,NOUT,NOUT2
*BMAT
*end:
```

It is possible to edit the output file of JMAKE and make it the new input file of JMAKE by changing the file name extension into '`.src`'. In order to avoid confusion with file names, this is not recommended except in case when corrections are accidentally made to '*.f*' file.

pThe user should **define all variables** in subroutines using JLP global variables, so that the compiler will print an error message if the user is trying to define a local variable having the same name as a JLP global variable. Note that in order to make proper definitions of common areas, JMAKE generates also variables not included in the 'needs:' section. The user can not rely that these additional definitions generated will remain the same in future versions of JLP.

4.5.2 Using JMAKE to manage own data structures

When writing own subroutines linked with JLP, the user may need to define own global parameters and variables. It is recommended that the user will manage his/her own global parameters and variables with JMAKE precompiler.

Parameters can be defined either by adding parameters directly into *jlp.par* or defining them in the same way as variables (see below). Here *jlp.par* refers to the parameter file of JMAKE (recall that the parameters can be in any file). Own variables can be defined as follows:

1) Add to `$$DEFINITIONS` statement in *jlp.par* the name of the file that contains the JMAKE definitions (that file can be ordinary source file as *jlp2.src* is).

2) Define the parameters, variables and common areas at the beginning of the file (later called *definitions section*) given in `$$DEFINITIONS` statement.

All lines in definitions section start with '*'. A comment line starts with '**'. Character '!' starts an end-of-line comment.

A definitions section must first contain `$FILES` subsection that looks like:

```
*$FILES ! files to be precompiled
*jlpsub.src
*jlpopt.src
*jlp2.src
*jlp.src
*$END
```

If no files are specified here (recall that these files can be given also in `$FILES` section of *jlp.par*) , this section contains only `*$FILES` and `*$END` lines. Then the definitions section may contain a parameter section like:

```
*::PARAMETER
*MAXOPN =7          ! Max. number of simultaneous open include files
**                    comment
*MAXNC=MAXD+8       ! MAXD must be defined earlier in jlp.par
*MAXNV=MAXNC+4      ! total number of variables
*LCHAR=1300         ! parameter used later to specify the length of character
**                    variable
*RPAR=1.58          ! real parameters can also be given
*DPAR=1.67D0        ! double precision parameters must contain both '.' and 'D'
*TXT = 'Message'    ! Character parameters are also allowed
```

pParameters can be equally well given in *jlp.par* as in `'*::PARAMETER'` section in definitions file.

Thereafter definitions can contain sections as:

```
*::VTYPE        ! VTYPE can be any variable type recognized by the compiler
*CFC            ! Variable doing something useful
```

```
**AML           ! comment
*MV(-2:MAXNC)   ! MAXNC needs to be a parameter defined earlier
```

If e.g. variable MV is needed in somewhere (it is in 'needs:' list or it is required to build a common area properly), JMAKE generates:

```
      PARAMETER (MAXD=100)        ! this comes from jlp.par
      PARAMETER (MAXNC=MAXD+8)    ! from *::PARAMETER section
      VTYPE    MV(-2:MAXNC)       ! MAXNC needs to be a parameter defined
```

Thus if a variable is needed, JMAKE generates automatically all the parameters needed.

If *jlp.par* contains a JMAKE $$-parameter like:

```
$$VTYPE= CTYPE*8    ! CTYPE*8 is a variable type known to the compiler
```

then JMAKE replaces the type VTYPE with type CTYPE*8:

```
     CTYPE*8    MV(-2:MAXNC)       ! MAXNC needs to be a parameter defined
```

There are no assumptions for variable types used in definitions section, thus all types accepted by the compiler can be used.

A special treatment is given for '*::CHARACTER' section which may look like:

```
*::CHARACTER
*100 VNAME(MAXD)
*LCHAR APUNIM          ! LCHAR is a parameter defined earlier
```

If VNAME and APUNIM are needed, JMAKE will generate

```
      PARAMETER         (MAXD=100)
      CHARACTER*100     VNAME(MAXD)
      CHARACTER*1300    APUNIM
```

Note that statement

```
      PARAMETER         (LCHAR=1300)
```

will be generated only if it is needed for other purposes in addition to specifying the length of APUNIM. A parameter determining the length of a character variable must be given literally, i.e. , definition

```
*LCHAR=LC1+LC2
```

is not allowed.

Common areas are defined in *::COMMON subsection as follows

```
*::COMMON
*JLPDAT ML,MV,NSTICLA,>
* IFREE,LMEM,ILINK1,LOCREJ,IXAP
*JLPXMA XMAT,CMAT
```

The first name is the name of the common area. Character '>' at the end of line indicates that the items in the next line belong to the same common area. If a variable in a common are is needed, then JMAKE will generate definitions for all the variables and the definition for the common area. JMAKE splits the lines in the definition of the common in the same way as splitted in the `*::COMMON` subsection, so the line can not be too long (JMAKE gives an error message if line is too long). JMAKE also generates `SAVE` statement for each common it creates, so commons created by JMAKE are static also in systems where default is that commons are dynamic.

There can be several definitions for the same common area. JMAKE will generate the definition containing variables needed in the subroutine (of course variables given in different definitions can not be used in the same subroutine). This way different subroutines can share the same working areas. JLP uses a common `JLPWRK` this way. The user can also use this common in report writer but not in subroutines used in transformations and reading the data into the program.

JMAKE can be used to generate also definitions for local variables. Variables will automatically be local if they are not contained in any common.

The definitions section ends with:

```
*::END
```

4.5.3 Using JMAKE precompiler options

If the user is making programs that should be used in different operating systems, then the precompiler options of JMAKE might be useful. Assume that *jlp.par* contains e.g. option:

```
$MREP = F    ! option is not in effect
```

or

```
$MREP = T    ! option is in effect
```

Then a program may contain section

```
*IF MREP
         call ownsub(par1,par2)
         write(n6,*)'kukuu'
*END
```

or section

```
*IF MREP
         call ownsub(par1,par2)
*ELSE
         write(n6,*)'kukuu'
*END
```

or section

```
*IF NOT MREP
         call ownsub(par1,par2)
*END
```

JMAKE will then comment out the lines according to the value (T/F) of option $MREP. No ordinary comment starting with '*' is allowed in '*IF ... *END' section. Options can be associated with a text string that can be used to transmit system dependent features into the code. For instance, assume that *jlp.par* contains:

```
$SECNDS=T= SECNDS(0.)
```

Then the program may contain:

```
*IF SECNDS REPLACE ??
       TIME=??
*ELSE
       TIME=0
*END
```

String defining what must be replaced if option is in effect can be anything (or contain even spaces). This is useful if JMAKE is used to precompile the output file of JMAKE where the original string (e.g. '??') has been replaced with e.g. 'double precision'.

4.5.4 Using JMAKE in other programs

JMAKE does not contain JLP specific assumptions. Thus it can be used in any program. The following changes are needed if JMAKE is used in other programs :

1) Change the default name of the parameter file determined in file *jmake.f* (this is not necessary as JMAKE asks if the default parameter file should be replaced with some other file).

2) Make the corresponding parameter file. At least $$FILES and $$DEFINITIONS statements must be different from *jlp.par*.

3) Make a definitions section to each file listed in $$DEFINITIONS statement in the parameter file.

4) Make 'needs:' section to each subroutine where global parameters or variables are needed.

5) If compiling options are needed, make corresponding '*IF *option* ... *END' sections.

4.6 Using JLP Data Structures and Subroutines

This section describes some general properties of those JLP data structures and subroutines that the user may need in writing own interface, data input and report generator subroutines. All variables and parameters mentioned can be accessed using 'needs:' construction of JMAKE.

4.6.1 Listing headers of subroutines with JLP

The purpose of this chapter is to introduce some possibilities how the user can add extra properties to JLP. More detailed (and updated) information is found in source files. JLP can be used to extract the summary headers of subroutines from the source files. Each subroutine has a short header (containing the `subroutine` or `function` statement and the purpose of the subroutine), and a longer header containing more information. Both headers starts with '*='. A short header ends with '**' and a long header ends with '***'. The short headers of all subroutines in file *jlpsub.src* can thus be printed as follow:

```
jlp>list/all jlpsub.src/*=:**
```

The headers in other files can be listed similarly (in addition to *jlpsub.src*, *jlpint.src* may be of special interest).

The longer forms of all headers can be listed as follows:

jlp>list/all jlpsub.src/*=:***

The listing of short headers in file *jlpsub.src* included:

```
*=jnewf=== file jlpint.src ========================
      subroutine jnewf(iunit,form,name,name2,errors)
*   Opens a new file (possibly a new version).
**
```

The long header of this specific module can be printed as follows:

```
jlp>list jlpsub.src/*=jnewf:***
```

The whole module `jnewf` can be listed as follows:

```
jlp>list jlpsub.src/*=jnewf:*=
```

4.6.2 Changing JLP subroutines

File *jlpint.src* contains subroutine templates whose purpose is to help the user to write own special subroutines for data access, transformations, report writer etc. Also the main program in file *jlp.src* can be replaced with custom main program. It is recommended that before making changes, the corresponding modules are copied into an own file, and this file is linked before files provided by JLP files so that standard routines will be replaced.

4.6.3 JLP data variables

As described in Chapter 2.5, JLP puts *d-*, *c-*, *x-* variables in the same vector 'V' when JLP read data or makes transformations. The variable names are stored in character vector

'VNAME'. The user can not assume any specific order of V-variables, except that variables created by an **xvar**, **cvar**, or **const** command and in one **dtran**, **xtran**, or **ctran** transformation paragraph are consecutive (this can be used in %-loops in transformations).

Variable lists

JLP refers to a subset of variables using integer vector called *variable list* having the following structure. For instance a variable list `listxs` is defined:

```
integer listxs(-1:MAXNX)
```

where `MAXNX` is a global JLP parameter. Element (-1) tells the maximum number of elements (i.e. `listxs(-1)=MAXNX`). Element (0) tells the actual number of elements (i.e. `0•listxs(0)•MAXNX`). Element `i`, `0•i•listxs(0)` refers to an element in `V`-vector, the name of the variable is `VNAME(listxs(i))`.

The user may need following subroutines for handling variable lists:

```
*=ilapp=== file jlpsub.src ======================
      subroutine ilapp(ix,list,errin,errors)
* Appends variable ix into a variable list 'list'.

*=ilfind=== file jlpsub.src ============================
      subroutine ilfind(ix,list,ilout)
* Finds the position of variable ix from a variable list.

*=ilmerg=== file jlpsub.src ==========================
      subroutine ilmerg(list1,list2,list3,errin,errors)
* Merges variable lists list1 and list2 into list3

*=ilput=== file jlpsub.src ======================
      subroutine ilput(ix,list,errin,errors,ilout)
* Puts an element ix to a list if it is not there.

*=ilret=== file jlpsub.src ========================================
      subroutine ilret(ix,list)
* Removes an element ix from a variable list and puts it into reserve
```

The user may need e.g. the following subroutines that treat also the names of variables:

```
*=jname=== file jlpsub.src ==============================
        subroutine jname(inp,names,nxres,nx,list,errors)
* Finds numbers of variables and makes new variable names.

*=joutl=== file jlpsub.src ==================
      subroutine joutl(level,buf,list,name)
*  Print names of variables in a variable list.

*=mlist=== file jlpsub.src ========================
      subroutine mlist(ch,ial,lop,nimi,nx,mul,errors)
* Makes a variable list.

*=mtja=== file jlpsub.src ==================
      function mtja(nimi,nx,xni)
* Finds the number of a variable with name xni.
```

Special variables

There are some special variables used for handling transformations etc. These variables should not generally be used for other purposes. JLP does not generally try to check if these variables are misused, as there are legal ways to handle these variables in nonstandard way (e.g. rejection variable 'reject' can be read directly from data). The global parameters for variable numbers and the names of the special variables are:

`*IVDATA`	'`data`' variable
`*IVUNIT`	'`unit`' variable
`*IVS`	'`s`' variable (current schedule)
`*IVONE`	number of variable having value 1.
`*IVDUPL`	number of '`duplicate`' variable
`*IVSPLI`	number of '`split`' variable
`*IVNS`	number of variable '`ns`'
`*IVREJ`	number of variable '`reject`'

How these variables are treated is described in Chapter 2.5.

4.6.4 Accessing stored *c*- and *x*-data

The user may want to access the stored *c*-variables and *x*-variables e.g. in her/his own report writer. Variable list `KEEPCL` tells what variables are stored as *c*-variables in simple vector `CMAT` defined as '`real CMAT(MAXCMA)`', where `MAXCMA` is a global parameter given in *jlp.par*. The number of stored *c*-variables is thus `KEEPCL(0)`. The first `KEEPCL(0)` elements of `CMAT` are the *c*-variables for the first unit, and so on up to the last unit `ML`. The name of first stored *c*-variable is `VNAME(KEEPCL(1))`, etc.

Variable list `KEEPXL` tells what are stored *x*-variables. Storage of *x*-variables is more complicated, because JLP is designed to be able to handle *x*-data that exceed the memory, and because JLP generates temporary *x*-variables for linear combinations of *x*-variables appearing on the rows of a linear programming problem. *X*-variables can be accessed by calling subroutine `jstun` for each unit started:

```
*=jstun=== file jlp2.src =======
      subroutine jstun(ic,ranac)
* Makes x-data ready for unit ic.
**
* Reads data from disk if necessary.
* Updates LISTV0 so that variable KEEPXL(ix) for schedule is
* can be accessed using statement function:
* x(is,ix)=XMAT(LISTV0+(is-1)*NXDD+ix).
* An equivalent (more complicated but clearly faster)
* way to access several x-variables in the same schedule is to compute
* the base addres for each schedule is as follows:
* isbas = LISTV0 + (is-1) * NXDD
* or if all schedules are acceses in order by defining starting
* value of isbas and adding NXDD for each schedule.
```

```
* Thereafter x-variable KEEPXL(ix) can be accessed with statement function:
* x2(ix)=XMAT(isbas + ix)
* Note: KEEPXL, XMAT, LISTV and NXDD are globals variables
*       accessed with 'needs:'
* input parameters:
         integer ic
* ic    = unit
         logical ranac
* ranac = .true. if units are accessed in any order (i.e.
*         not necessarily in order 1,2,...,ML.
*         If data does not fit to the memory, it is recommended
*         that even with ranac=.true. the unit numbers in consecutive
*         calls are in increasing order (units may be missing) so that
*         work file needs not to be rewinded repeatedly.
*********************************************************
```

Thereafter `keepxl` variables can be accessed with either of the statement function described above in the header of `jstun`. A global function subroutine is not used in JLP, because satement functions work much faster.

4.6.5 Text buffers

Text is stored in text buffers. Each buffer has a three character name called later 'bufnam' e.g. bufnam='DTR'. A text buffer is a single character variable to which all text lines are packed. The name of the variable is bufnam//'BUF', e.g. 'DTRBUF'. The length of the variable is determined by JMAKE parameter given in file jlp.par. the name of the length parameter is 'LEN'//bufnam e.g. 'LENDTR'. Associated with each buffer is a link vector with name 'LNK'//bufnam (e.g. 'LNKDTR') which tells the size of the buffer used to prevent overflow, and links to the first character in each line. The maximum number of lines in a buffer is given by a parameter with name 'LIN'//bufnam, e.g. 'LINDTR'. The buffer name is stored in the buffer variable so that the buffer subroutines can generate error messages if parameters are too small. The user may also use the following buffer subroutines:

```
*=bufapp=== file jlpsub.src ========================
      subroutine  bufapp(inp,le,txtbuf,lnktxt,errors)
*   Adds string inp to standard buffer txtbuf.

*=bufio=== file jlpsub.src ============================
      subroutine bufio(what,line,L,errors)
*   Sends commands to JLP and gets the JLP output.
*   Handles command buffer 'INC' and output buffer 'OUT'.

*=bufpri=== file jlpsub.src ==========================
      subroutine  bufpri(nu,txtbuf,lnktxt)
* Prints the contents of text buffer txtbuf into a file.
```

For more information about text buffers, see the long headers of the above subroutines, especially of the subroutine `buffapp`.

4.6.6 String manipulation

The user may use the following string manipulation subroutines:

```
*=adjul2=== file jlpsub.src ==============================================
      subroutine adjul2(inp)
* Adjusts a character variable to the left, i.e. removes initial blanks

*=chi5=== file jlpsub.src ===========
      character*5 function chi5(i,il)
* Returns integer i as character*5.

*=chr8=== file jlpsub.src ====================
      character*8 function chr8(a)
* Returns real value as a character*8 variable.

*=chr10=== file jlpsub.src ==========================
      character*10 function chr10(a)
* Returns double precision a as character*10 variable.

*=len1=== file jlpsub.src ========================
      function len1(str)
*  Returns the position of first nonblank character.

*=len2=== file jlpsub.src ================================
      function len2(str)
* Returns the length of str when trailing blanks are ignored

*=nexlim=== file jlpsub.src ============
      function nexlim(inp,ial,lop,limit)
* Finds the next limiter.

*=repl=== file jlpsub.src ==========================
      subroutine repl(jono,jono1,jono2,lkm1,lkm2,lop)
*  Replaces substring with another string.

*=jrepl=== file jlpsub.src ================================
       subroutine jrepl(jono1,i1,i2,lop,jono2,le2)
*replaces the substring jono1(i1:i2) by string jono2(1:le2)
```

4.6.7 Printing subroutines

JLP prints almost all results using subroutine `jout` that prints a character line (character variable) to terminal, output file and output buffer according to the current options of printing (determined by **`printlevel`**, **`outlevel`**, **`outfile`**, **`buflevel`**). (Currently the optimization algorithm prints information about how the optimization proceeds only to the terminal.) The user can also call this and other printing subroutines:

```
*=jout=== file jlp2.src ==============================
      subroutine jout(ilevel,buf)
* Outputs a line into screen and/or file and/or buffer.

*=jouti=== file jlpsub.src =============
      subroutine jouti(level,buf,ivec,n)
*          Outputs an integer vector.

*=joutl=== file jlpsub.src ====================
      subroutine joutl(level,buf,list,name)
*  Print names of variables in a variable list.
```

4.6.8 Transformation subroutines

JLP handles all transformations (**dtran**-, **ctran**-, **xtran**-, **dupl**-, and **parin**-transformations and definitions of domains) with the same subroutines. Transformations are first compiled with subroutine `compi`:

```
*=compi=== file jlpsub.src ========================================
      subroutine compi(teku,nteku,nimi,nxres,nx,x,nint,npfrst,nxtot,
     6 jono,errors,ixoutl)
*  Compiles a transformation line jono into vector teku.
```

Compiled transformations are then made for variables stored in vector `x` with subroutine `muun`:

```
*=muun=== file jlpsub.src ===========
         subroutine muun(x,teku)
*  Computes compiled transformations.
```

The user can use these transformation routines for own purposes.

If there are no defined transformations, subroutine `muun` can be called safely (i.e. with immediate return) if the vector of compiled transformations (`teku`) is properly initialized (otherwise unpredictable problems with memory will occur).

4.7 Creating Own Transformation Subroutines

It is possible to add own functions that can be used in transformations exactly as the predefined functions. Own functions can be added by editing function **`ifunc`** and subroutine **`func`** in file *jlpint.src*. JLP global parameters and variables can be used but they are not generally necessary.

To show how this can be done, function *npv* is included as an example. Transformation defined as:

```
present_value=npv(3,100,0,50,2,-70,10)
```

will calculate the net present value using 3% interest rate when there is instant income 100, income 50 after 2 years and payment 70 after 10 years. There can be any number of (income,time) - pairs in the function call, and any of the arguments can be a variable.

A new function can be defined by editing function `ifunc` and subroutine `func` properly:

```
*=ifunc=== file jlpint.src ========================================
        function ifunc(name)
* Defines function names for own functions and returns their number.

*=func=== file jlpint.src ==============
      subroutine func(teku,x)
*  Compute the value of an own function.
```

4.8 User Designs for RHS Generation

The user defines constraints in the **problem** paragraph in form:

```
volume =1000 / >100 <1000 / >0
```

Then **solve** *r* command tells JLP to use r^{th} set of RHS's, or **solve** +*r* tells JLP to use the set of RHS's with number: previous_number + *r*. If many sets of RHS's are used in a systematic way it is tedious to write all the combinations into the problem paragraph. If the **solve** command is given with option starting with '/m', e.g.:

```
solve/mmethod r
```

or

```
solve/mstandard +r
```

then the subroutine *next* is used to generate RHS's:

```
*=next=== file jlpint.src ==================
      subroutine next(method,ir,errors)
* Gets new upper and lower bounds for JLP.
```

The whole option is transmitted to *next* as a character variable *method* and can be used as input parameter for specifying the method for generating the RHS. The lower and upper bounds given in the **problem** paragraph can be used as parameters for defining new RHS's. Subroutine *next* contains the code for a method 'mstandard' which generates RHS's exactly in the same way as the standard interface without explicit *method*. The standard interface does not use subroutine *next*, so the user can safely edit it.

4.9 User Defined Data Input

If the format for *x*-data is given by 'xform m' (where 'm' stands for 'my_own'), then both *x*-data and *c*-data are read in using user defined subroutines. If 'xform m' is in effect, JLP opens files and reads records as follows (transformations etc. are made as described in Chapter 2.5):

```
call minit   - initializes reading
do ifi=1, (number of xdat files)
     call mopen              ! Open ifi^th cdat and xdat file
                             ! get the number of treament units in file
     do iu=1, (number of units)
          call mgetc         ! read values of cvar variables of the unit
                    do is = 1, ns    ! ns = number of schedules in the unit
                    call mgetx ! read values of xvar variables from xdat file
          end of loop over schedules
```

```
      end of loop over units
  end of loop over files
  call mfinit  - open files can be closed etc.
```

Terms 'open a file' and 'reading variables' mean that such operations are done in the user subroutines that work similarly as if files were opened and records read. For the user, the essential fact is in what place in the loop structure each subroutine is called.There does not need to be a one-to-one connection between the logical and physical operations. For instance, xdat file names can be area codes of a data base system, and *c*-variables and *x*-variables may be stored in the same data base. Or, files can be opened in the `mgetc` subroutine. It is also possible that treatment schedules are simulated *in place*. JLP does not change values of *c*-, and *x*-variables (unless modified by **`ctran`** and **`xtran`** transformations), so it is possible that `mgetc` and `mgetx` give only the changing values. This may be handy if data contain several levels of hierarchy (e.g., state, coynty, village, farm).

File *jlpint.src* contains subroutine templates that the user can use as starting point when defining own subroutines, or as dummy subroutines in case no special input subroutines are needed:

```
*=minit=== file jlpint.src ============================
       subroutine minit(errors)
* Initilizes everything for reading data with 'xform m'

*=mopen=== file jlpint.src ========================================
        subroutine mopen(mlfil,errors)
* Initializes reading of new data, called for each element of xdat-list

*=mgetc=== file jlpint.src ========================
      subroutine mgetc()
* reads the c-variables of the next calculation unit

*=mgetx=== file jlpint.src ================
       subroutine mgetx()
* Rreads the x-variables of the next schedule.

*=mfinit=== file jlpint.src ======================
       subroutine mfinit(errors)
* Cleans everything after reading data with 'xform m'
```

The provided subroutine templates work in the same way as if '`xform b`' and '`cform *`' would be in effect.

4.10 Writing Own Report Writer

If the printing options provided by JLP (**`show`**, **`sched`**) are not enough, or the results are needed in binary form for further analysis, the user can write his own report writer. Using JLP subroutines and global variables, a report writer can have access to the following variables:

1) termination status of the problem

2) RHS's used in the solution
3) values of rows (utility constraints + objective function)
4) shadow prices of utility constraints
5) values of (aggregated) *x*-variables (including *x*-variables not used in the problem definition)
6) shadow prices of *x*-variables included in the problem
7) cost of forcing *x*-variables to have smaller or greater value they obtained according to the solution (x-variables may or may not have been used in the problem definition)
8) values of *z*-variables used in the problem definition
9) reduced costs of nonbasic *z*-variables
.....
10) weights of schedules in the solution
11) shadow prices of units (= shadow prices of basic schedules)
12) shadow prices of nonoptimal schedules (reduced cost for forcing nonbasic schedules into the solution)

JLP prints quantities 1) – 9) automatically after solving each problem (according to the current options of **`show`** command). How the user can replace or augment this report is described in the next section. JLP prints quantities 10) – 12) connected with schedules with **`sched`** command. How these reports can be replaced or augmented is described in the section thereafter.

4.10.1 General part of the report writer

The general report JP prints after each solution can be replaced or augmented by editing the subroutine template `repo`:

```
*=repo=== file jlpint.src =================
      subroutine repo(inp,errors)
* subroutine template for own report writer
```

If the user writes a command line starting with 'repo' then JLP calls subroutine 'repo'. The whole command line is transmitted as an input character variable to the subroutines, so that the user can specify in the command line all necessary printing options. If option '`/repo`' of command **`show`** is in effect, then the report is generated always with `repo` instead of the standard JLP report writer. The command line transmitted to `repo` is in this case the **`solve`** command line, and can not be used so easily to transmit report writer options. The provided template for `repo` prepares basically the same report as JLP usually does but in a slightly simplified format.

If JMAKE option `$MREP` is in effect, then JLP will call subroutine `mrep` exactly in the same ways as `repo` is called. That is, if a command line starts with 'mrep' then JLP calls

subroutine 'mrep'. And if option '/mrep' of command **show** is in effect, then the report is generated with mrep instead of the standard JLP report writer. If both option '/repo' and option '/mrep' are in effect, then JLP calls first repo and thereafter mrep. Report writer repo is intended for a general purpose report writer, and mrep for report writer for special data structures (e.g. MELA system), i.e. for the case when the data are read in with 'xform m'.

Because the use of mrep is identical to the use of repo, there is no separate subroutine template for subroutine mrep (one can start making mrep from a copy of repo where the subroutine name is changed into mrep).

The header of repo contains a list of those global variables that are possibly needed. The options of **show** command determine what global variables are actually computed by JLP. For instance, the integer approximation is computed only if '/int' option is in effect, and cost of decrease and increase is computed only if '/cost' option is in effect. The shadow prices of *x*-variables are not computed into global variables, because they are fast to compute with subroutine jpix when needed:

```
*=jpix=== file jlp2.src ====================
       subroutine jpix(idom,iv,ipres,pix)
* computes the shadow price for an x-variable
```

4.10.2 Report writer for schedule information

The user may want to treat the schedule information (items 10-12 above) differently than command **sched** allows. A subroutine template showing how to access the necessary global variables is in subroutine own1:

```
*=own1=== file jlpint.src ==========================================
      subroutine own1(inp,errors)
* Subroutine template for own command given in OWN1 option in jlp.par.
* Currently includes template for report writer replacing
* sched command and showing how to access c- and x-data.
**
```

Subroutine own1 can be accessed with a command given in file *jlp.par* (see section 4.1.2). The default command name is **own1**. The user may wish to combine all report writing procedures into subroutine repo and/or subroutine mrep described above.

The shadow prices of schedules (including shadow prices of units) are not computed into a global vector. They can be accessed with subroutine jpis:

```
*=jpis=== file jlp2.src ===================
      subroutine jpis(iunit,is,spsc)
* Computes the shadow price of an schedule.
```

4.11 Creating Own Interface

JLP is designed so that the user can easily create totally new interface with menus and buttons etc. on the provided command based interface. This can be done using input and output buffers. There are three main strategies for building an own interface. Because JLP controls command input and printed output independently, it is possible to choose the input method from one strategy and output method from another.

4.11.1 Main program interface calling JLP

The provided main program in file *jlp.src* is very simple. It basically just calls subroutine `jlpin` that contains the standard JLP interface. Thus the user can write an own main program that will replace the standard main program.

The main program must (here the program calling JLP subroutine `jlpin` is called *main program*, it can also be a subroutine) define an character variable for error messages and a variable for receiving output:

```
      character*80 errors
      character*78 outlin  ! the length can be also e.g. 80
* errors must initially be empty
      data errors/' '/
```

The main program can communicate with `jlpin` using subroutine `bufio:`

```
*=bufio=== file jlpsub.src ==============================
      subroutine bufio(what,line,L,errors)
*     Sends commands to JLP and gets the generated output
**
* INPUT:
* what = 'in'  adds line to command buffer INC
*      = 'in/clear' clears command buffer
*      = 'out' gets a line from output buffer OUT
*      = 'out/clear' clears output buffer
```

The main program can put a package of commands to the command buffer using `'in'` as `what` parameter of `bufio`. If last command put to the buffer is `'end'` then the control can be obtained back to the main program (otherwise control remains in JLP, usually JLP would wait input from the terminal). For instance:

```
* output is put to the output buffer:
      call bufio('in','buflevel 2',L,errors)
      if(errors(1:1).ne.' ')goto 999 ! errors are checked there
*      error messages start always in column one, it is faster to test
*      only first character
      call bufio('in','end',L,errors)
      if(errors(1:1).ne.' ')goto 999
```

JLP can then be asked to execute the commands:

```
      call jlpin(errors)
```

If parameter **buflevel** has been >0, the output has been send to the output buffer that can be printed e.g. as follows:.

```
10      call bufio('out',buf,L3,errors)
        if(L3.lt.0)goto 20
         if(l3.gt.0) write(n6,*)buf(1:L3)
       goto 10
20     (new commands)
```

After solving a linear programming problem, an own report generator can be accessed either directly from the main program or via JLP (e.g. with JLP command **report**).

4.11.2 Interface in a subroutine called by JLP

If JLP gets command **buff** it calls subroutine `buff`:

```
*=buff=== file jlpint.src ================================
         subroutine  buff(inp,errors)
* An example of an interface operating through the buffer.
```

The subroutine template written to subroutine `buff` is handling similar interface as the main program interface described in the previous section. Commands are read from the terminal with prompt `'bufin>'` and they are put into the command buffer. When string `'//'` is encountered, control returns to the calling subroutine `jlpin` and stored commands are executed. If **buff** is the last command put to the buffer, control returns back to this subroutine. If `buflevel` is given a positive value, then output goes to output buffer that can treated in this subroutine first.

The main program provided will give control directly to subroutine `buff`, if JMAKE option `$INIT1` in *jlp.par* is given value `'buff'`.

It depends on the structure of the interface and on what other tasks the interface is controlling if it is easier to build the interface into main program (or a subprogram) that calls `jlpin`, or if it is better to build the interface into subroutine `buff` that is called by `jlpin`.

4.11.3 Replacing terminal input and buffer output

The interface structures described in the two previous sections are based on the idea that the interface is intelligent, i.e., the interface knows what it is striving at so that it can send to JLP command packages that accomplish major tasks. But an interface may be just an other way of sending commands to JLP and printing the results. For instance, the user may want to send commands using buttons or menus and get results to different windows. In such an interface the main thing is that terminal input and output (FORTRAN `read` and `write` statements) must be replaced with some other operations. The JLP package provides the following tools for this.

Replacing terminal input

If JLP gets command **ownread**, then the terminal input (reading from unit n5) is replaced by call to subroutine ownrea:

```
**=ownrea=== file jlpint.src==================================
         subroutine ownrea(line)
* An example of own input function that replaces terminal input.
```

The provided template for ownrea just reads the command line from the terminal.

The command **ownrea** affects only reading from the terminal, i.e., **include** command can still be used to get input from files.

Command **ownrea** will toggle, i.e., giving another **ownrea** terminal input is used again.

Replacing buffer output

The output buffer provides an way to replace terminal output. If **printlevel** is set to zero, and **buflevel** is given a positive value, then nothing is printed to the terminal and all output goes to the output buffer. The output buffer can then be handled in the main program after returning to the main program after command **end,** or in the subroutine buff after giving the control to subroutine buff by command **buff.** If the **buflevel** is given a negative value, then instead of putting a line into the output buffer, JLP calls subroutine ownwri:

```
*=ownwri=== file jlpint.src ================================
         subroutine ownwri(line)
* An example of own output function replacing buffer output.
```

With ownwri the output can be handled line by line. It may be easier to make input and output co-operate smoothly, if entries ownrea and ownwri are put to the same subroutine.

A possible use for ownwri is to get better scrolling properties on the screen than obtained by unqualified writing to the standard terminal unit.

4.12 Adding Own Commands to JLP

The user may add two commands to the JLP commands as follows (on request arrangements for more commands can be easily made). The names of commands can be given by giving proper values for JMAKE options $OWN1 and $OWN2 in *jlp.par*. Let us call the commands **own1** and **own2** (as is the default given in *jlp.par*). When these commands are encountered, JLP calls user subroutines own1 and own2:

```
*=own1=== file jlpint.src ==========================================
```

```
      subroutine own1(inp,errors)
* Subroutine template for own command given in OWN1 option in jlp.par.
* Currently includes template for report writer replacing
* sched command and showing how to access c- and x-data.
**
* INPUT:  inp    = the whole command line (extra blanks are removed)

*=own2=== file jlpint.src ==========================================
      subroutine own2(inp,errors)
* Subroutine template for own command given in OWN2 option in jlp.par.
```

These subroutines get the whole command line as the input, so all command options etc. can be implemented by interpreting the command line properly. As all JLP global parameters and variables can be accessed using 'needs:' construction of JMAKE, the user may do whatever she/he wants in these subroutines.

5. ERRORS AND TROUBLESHOOTING

5.1 Syntax Errors

If JLP encounters an illegal command in batch mode, the program terminates (returns to the main program) with the proper error message. In interactive mode (default) all open *include* files are closed, the error message is printed, and the control is given to the input terminal. Note that only the significant part of a command is interpreted, and e.g. 'printleuvel 2' does not cause an error.

JLP prints warning messages in case no error has occurred but the result of a JLP command may be different than the user may expect. For instance, if JLP is asked to solve a problem without an objective function, JLP will print:

```
*W* no objective variable, finding feasible
```

The author is expecting feedback from the users to improve the error and warning messages, and how to deal with error situations.

5.2 Dimensions of Vectors

JLP tries to check the ranges of character substrings and array indexes. If an overflow would occur, JLP prints an error message telling what parameter should be increased. For example the error message for parameter `MAXNX` is:

```
*PAR* increase MAXNX
```

The parameter `MAXNX` in file *jlp.par* should then be increased and JLP rebuilt as described in Chapter 4.1. It is possible to continue the current session with other commands. However, if the error message comes in form:

```
*F*PAR* increase MAXSPL
```

then the data areas are out of order, and the current session can be continued only after **`init`**. It is recommended that JLP source files are compiled without range checking option, unless the user suspects that JLP fails in the range checking (which is, in theory, possible). Programs compiled without range checking are smaller and faster.

5.3 Problems in the Optimization

A major difficulty in a nontrivial numerical algorithm is that unavoidable rounding errors may prevent the algorithm from finding the solution within a reasonable accuracy. Even if JLP has solved all the test problems, there are certainly problems where JLP fails. In case of difficulties, and before consulting the author, the user should:

i) use 'printlevel 9' to get all the diagnostic output that might explain the cause of the problem,

ii) try to solve modified problems, e.g., by adding a constraint at a time, to see when the problems arise.

iii) modify **`parin`** parameters `tole, invert` and/or `wmin` (see section 2.7.4).

5.3.1 Degeneracy due to linear dependency

A basic variable in a linear programming problem is called degenerate if its value is zero. Degeneracy can cause unstable behavior. There are two types of degeneracy problems that have been addressed in the design of JLP.

First degeneracy situation arises when some constraint rows are linear combinations of others. An example:

```
> prob
> income.2-income.1=0
> income.3-income.2=0
> income.4-income.3=0
> income.5-income.4=0
> income.5-income.3=0
> npv.0 max
> /
```

Now the last constraint '`income.5-income.3`' is a linear combination (sum) of the two previous constraints. JLP keeps all constraints (including equality constraints) nonbinding as long as they are satisfied up to the tolerance computed from the minimum and maximum value of each *x*-variable. Thus in the above sample problem the constraint for '`income.5-income.3`' will not become binding, and following solution is obtained:

```
row                                    value       shadow      lower     upper
                                                   price       bound     bound
_____________________________________________________________________________
  1) income.2-income.1  . . . . .  0.00000000 -0.2052712         0.000000     L
  2) income.3-income.2  . . . . .  0.00000000 -0.2147269         0.000000     L
  3) income.4-income.3  . . . . .  0.00000000 -0.0895401         0.000000     L
  4) income.5-income.4  . . . . .  0.00000000 -0.0410396         0.000000     L
  5) income.5-income.3  . . . . .  0.00000000 0.00000000         0.000000
  6) npv.0 . . . . . . . . . . . . 33459072.7 1.00000000            max
```

The order of the last two constraints were then changed.

```
> prob
> income.2-income.1=0
> income.3-income.2=0
> income.4-income.3=0
> income.5-income.3=0
> income.5-income.4=0
> npv.0 max
> /
```

The last constraint is nonbinding also this time, and following results are obtained:

```
row                                    value       shadow      lower     upper
                                                   price       bound     bound
_____________________________________________________________________________
  1) income.2-income.1  . . . . .  0.00000000 -0.2052712         0.000000     L
  2) income.3-income.2  . . . . .  0.00000000 -0.2147269         0.000000     L
  3) income.4-income.3  . . . . .  0.00000000 -0.0485004         0.000000     L
  4) income.5-income.3  . . . . .  0.00000000 -0.0410396         0.000000     L
  5) income.5-income.4  . . . . .  0.00000000 0.00000000         0.000000
  6) npv.0 . . . . . . . . . . . . 33459072.7 1.00000000            max
```

Thus the results look different depending on the order of constraints. Note the relations between the shadow prices of this and the previous problem: -0.0485004-0.0410396= -0.089540. The shadow prices for *x*-variables look the same in both cases.

If the computed tolerance range for constraints is too small, then linear dependencies may remain undetected, and JLP may behave in an unstable way, and may or may not find the solution. If the tolerance range is too wide, then JLP will get a reasonable solution but the solution is not exact in the sense that a constraint that should be binding is not. See section 2.7.4 for how to change the default tolerance.

5.3.2 Degeneracy when lower bound = minimum

Suppose that the simulated alternatives contain alternatives with herbicide treatments (x-variable `herbicide >0`) , and those alternatives are economically favorable. If we set a constraint '`herbicide=0`' then this constraint will become binding and will get a nonnegative shadow price. Thus the algorithm takes a schedule with `herbicide>0` as a basic schedule, even if the weight of such a schedule is zero. (Earlier versions of JLP had difficulties in finding the solution in this case.)

A faster way to implement such constraints would be to reject unacceptable alternatives in **`xtran`**- transformations:

```
xtran
if herbicide>0 then reject
/
```

If a constraint is forced this way, then no shadow price is obtained.

6. LINEAR PROGRAMMING ALGORITHM

In this part, the mathematical background of JLP algorithm is briefly described. The domain structure has effect only on the way different variables are accessed and not in the basic optimization algorithm as such. Thus the algorithm is described without a reference to the domains. The realization of the domain structure is then described at the end of the part. The reader is assumed to be familiar with basic linear programming concepts (see e.g. Luenberger 1973).

6.1 Problem Formulation

Let us first restate the problem definition from Chapter 1.2 in a slightly different form (see Chapter 1.2 for interpretation of the symbols)

$$\text{Max or Min } z_0 = \mathbf{a}_0' \mathbf{x} + \mathbf{b}_0' \mathbf{z} \tag{6.1}$$

subject to:

$$c_t \le \mathbf{a}_t' \mathbf{x} + \mathbf{b}_t' \mathbf{z} \le C_t, \quad t = 1,..., r \tag{6.2}$$

$$x_k - \sum_{i=1}^{m} \sum_{j=1}^{n_i} x_k^{ij} w_{ij} = 0, \quad k = 1,\ldots, p \tag{6.3}$$

$$\sum_{j=1}^{n_i} w_{ij} = 1, \quad i = 1,\ldots, m \tag{6.4}$$

$$w_{ij} \ge 0 \quad \text{for all } i \text{ and } j \tag{6.5}$$

$$z_k \ge 0 \quad \text{for } k = 1,\ldots, q \tag{6.6}$$

Vectors **z**, and **x** are:

$$\mathbf{x} = \begin{pmatrix} x_1 & \ldots & x_p \end{pmatrix}' \tag{6.7}$$

$$\mathbf{z} = \begin{pmatrix} z_1 & \ldots & z_q \end{pmatrix}' \tag{6.8}$$

Constraints (6.2) can be written in matrix form as:

$$\mathbf{c} \le \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{z} \le \mathbf{C} \tag{6.9}$$

The problem is easier to understand (and define) if the constraints including the aggregate x_k-variables and their definitions are presented separately, as above. An equivalent problem would be obtained by substituting the definitions of x-variables directly into the objective (6.1) and constraints (6.2) (as is the formulation of Dantzig and Van Slyke 1967). Note that without a loss of generality we might assume that on each row t all coefficients a_{tk} are zero except possibly one coefficient is one. For instance, if some row t contains

$$2x_2 + 3x_2$$

then this linear combination can be replaced by a new variable x_{p+1} for which we define:

$$x_{p+1}^{ij} = 2x_2^{ij} + 3x_2^{ij}$$

It is more natural for the user to define problems without artificial new x-variables, but computationally a more efficient algorithm is obtained by making new variables for linear combinations of x-variables. These variables are called 'temporary x-variables' in JLP output. The mathematical basis of JLP is here described assuming that there can be several x-variables on each row. It is also indicated how computations will simplify if there can be only one x-variable on each row without a coefficient (i.e. with coefficient 1). This formulation is called **one-x formulation**.

Any standard linear programming algorithm can be used to solve the problem, at least after writing any constraint t of form (6.2) as two separate constraints, one for the lower bound and the other for the upper bound, or as an equality constraint in case c_k=C_k. However, to solve the problem efficiently, the special features of the problem should be taken into account.

JLP applies the following techniques:

(i) Generalized upper bound technique (see Dantzig and Van Slyke 1967) is used to handle the area constraints (6.4).

(ii) Using the revised simplex method (used also by Dantzig and Van Slyke 1967), the algorithm makes small local steps, i.e. without having the whole tableau in the memory.

(iii) The basic unit in the optimization is one treatment unit, thus the algorithm applies a kind of decomposition technique.

(iv) An upper bound technique is used to handle simultaneously both the lower and upper bound.

6.2 Generalized Upper Bound Technique

6.2.1 Basic idea: key variables

The generalized upper bound technique is the most important special feature of the algorithm. The number of constraints in the problem is $m + 2r$, and generally m is large and r is small. As the speed and memory requirements of a linear programming computer program depend mainly on the number of constraints, the original problem may take quite much time and memory. Applying the generalized upper bound technique for the area constraints and ordinary upper bound technique for the upper bounds, the effective number of constraints is r.

The basic idea of the generalized upper bound technique is that the area constraint (6.4) for treatment unit i

$$\sum_{j=1}^{n_i} w_{ij} = 1$$

will be automatically satisfied if we select from each unit i a schedule $J(i)$, and write $w_{iJ(i)}$ in terms of the other weights:

$$w_{iJ(i)} = 1 - \sum_{j \neq J(i)} w_{ij} \tag{6.10}$$

Constraint (6.3) defining variable x_k, k=0,...p can then be written without variables $w_{iJ(i)}$:

$$x_k - \sum_{i=1}^{m} \left[\sum_{j \neq J(i)} x_k^{ij} w_{ij} + x_k^{iJ(i)} \left(1 - \sum_{j \neq J(i)} w_{ij} \right) \right] = 0 \tag{6.11}$$

or

$$x_k - \sum_{i=1}^{m} \sum_{j \neq J(i)} \left(x_k^{ij} - x_k^{iJ(i)} \right) w_{ij} = \sum_{i=1}^{m} x_k^{iJ(i)} \tag{6.12}$$

The area constraints (6.4) can be dropped from the problem, since (6.10) automatically guarantees that they are satisfied. However, for each unit i, the nonnegativity constraint $w_{iJ(i)} \geq 0$ will become:

$$\sum_{j \neq J(i)} w_{ij} \leq 1, \quad i = 1,\ldots,m \tag{6.13}$$

Thus the number of constraints (nonnegativity constraints are not counted) is the same as in the original formulation, the area constraints were just changed into constraints (6.13) . However, if the schedules *J(i)* are chosen at each stage of the solution process so that $w_{iJ(i)}$ would be a basic variable (i.e. $w_{iJ(i)} >0$), then these new constraints are never active. Thus the working basis can be formed without having basic variables corresponding to these constraints.

The problem definition uses inequality constraints. As the matrix algebra of linear programming is based on equalities, artificial surplus or slack variables are usually introduced to make inequalities formally into equations. JLP treats nonbinding constraints without surplus and slack variables by adjusting the dimension of the basis matrix according to the number of active constraints. This can make the algorithm faster if there are several nonbinding constraints.

Let us first describe how the optimization proceeds at any stage after finding a feasible solution. How to obtain a feasible solution is described later. A stage of optimization can be described as follows:

For each unit *i* there is an *key schedule* *J(i)* for which $w_{iJ(i)} >0$. Variables $w_{iJ(i)}$ ("key variables" of Dantzig and Van Slyke 1967) are implicit basic variables they are not included in the working basis. Let **s** denote the sum of *x*-variables over the key schedules, i.e.:

$$\mathbf{s} = \sum_{i=1}^{m} \mathbf{x}^{iJ(i)} \quad , \tag{6.14}$$

where

$$\mathbf{x}^{ij} = \left(x_1^{ij} \quad \dots \quad x_p^{ij} \right)' \tag{6.15}$$

There are *R* binding utility constraints, $0 \le R \le r$, for each binding utility constraint *t* either the lower bound c_t or the upper bound C_t is active. Let us denote the *R*-vector of the active bounds by $\mathbf{c}_b$. Assume for simplicity that the binding constraints are the *R* first.

Correponding to the *R* binding utility constraints, there are *R* basic variables among *w*- and *z*-variables (these variables form the "working basis" of Dantzig and Van Slyke 1967). Let the number of basic basic *z*-variables be *Q*. Assume for simplicity that the basic *z*-variables are the *Q* first. Let $P = R - Q$ be the number of basic *w*-variables (in addition to the implicit basic variables $w_{iJ(i)}$). These *w*-variables are called explicit basic *w*-variables, and the corresponding schedules are called explicit basic schedules. Let us index the explicit basic schedules by *u*, and denote the unit and schedule for explicit

basic schedules by *ij(u)*, *u*=1,...,*P*. Note that there can be more than one explicit basic schedule in the same treatment unit. Denote further:

$$\mathbf{w} = \left(w_{ij(1)}, \quad \dots \quad , w_{ij(P)}\right)' \tag{6.16}$$

$$\mathbf{d}_u = \mathbf{x}^{ij(u)} - \mathbf{x}^{iJ(u)}, \; u = 1, \dots, P, \tag{6.17}$$

$$\mathbf{D} = \left(\mathbf{d}_1 \quad \dots \quad \mathbf{d}_P\right). \tag{6.18}$$

Thus the current value of $\mathbf{x}$ is:

$$\mathbf{x} = \mathbf{s} + \mathbf{D}\mathbf{w} \tag{6.19}$$

Let us decompose $\mathbf{A}$, $\mathbf{B}$, $\mathbf{b}_0$, and $\mathbf{z}$ separating binding and nonbinding constraints and basic and nonbasic variables:

$$\mathbf{z} = \begin{pmatrix} \mathbf{z}_b \\ \mathbf{z}_n \end{pmatrix}, \text{ where } \mathbf{z}_n = \mathbf{0} \tag{6.20}$$

$$\mathbf{b}_0 = \begin{pmatrix} \mathbf{b}_{0b} \\ \mathbf{b}_{0n} \end{pmatrix} \tag{6.21}$$

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_b \\ \mathbf{A}_n \end{pmatrix} \tag{6.22}$$

$$\mathbf{B} = \begin{pmatrix} \mathbf{B}_{bb} & \mathbf{B}_{bn} \\ \mathbf{B}_{nb} & \mathbf{B}_{nn} \end{pmatrix}. \tag{6.23}$$

The current value of the objective function is :

$$z_0 = \mathbf{a}_0' \mathbf{x} + \mathbf{b}_0' \mathbf{z} = \mathbf{a}_0' \mathbf{s} + \mathbf{a}_0' \mathbf{D}\mathbf{w} + \mathbf{b}_{0b}' \mathbf{z}_b \quad , \tag{6.24}$$

where the current values of $\mathbf{w}$ and $\mathbf{z}_b$ can be solved using the assumption that the R first utility constraints are binding:

$$\mathbf{A}_b \mathbf{x} + \mathbf{B}_{bb} \mathbf{z}_b = \mathbf{c}_b, \text{ or} \tag{6.25}$$

$$\mathbf{A}_b \mathbf{s} + \mathbf{A}_b \mathbf{D}\mathbf{w} + \mathbf{B}_{bb} \mathbf{z}_b = \mathbf{c}_b, \text{ or} \tag{6.26}$$

$$\mathbf{A}_b \mathbf{D}\mathbf{w} + \mathbf{B}_{bb} \mathbf{z}_b = \mathbf{c}_b - \mathbf{A}_b \mathbf{s} \text{ , or} \tag{6.27}$$

$$\left(\mathbf{A}_b \mathbf{D} \quad \mathbf{B}_{bb}\right) \begin{pmatrix} \mathbf{w} \\ \mathbf{z}_b \end{pmatrix} = \mathbf{c}_b - \mathbf{A}_b \mathbf{s} \text{ , or} \tag{6.28}$$

$$\begin{pmatrix} \mathbf{w} \\ \mathbf{z}_b \end{pmatrix} = \left(\mathbf{A}_b \mathbf{D} \quad \mathbf{B}_{bb}\right)^{-1} \left(\mathbf{c}_b - \mathbf{A}_b \mathbf{s}\right). \tag{6.29}$$

The matrix $(\mathbf{A}_b\mathbf{D} \quad \mathbf{B}_{bb})$ is the current (working) basis matrix of the problem.

6.2.2 Entering variable

There can be three different possibilities to improve the current solution:

i) A new schedule *j* for some unit *i* enters into the solution (more precisely: weight w_{ij} enters into the the solution).

ii) A nonbasic *z*-variable enters into the solution

iii) A binding constraint becomes nonbinding (the slack or surplus variable of a binding constraint enters into the solution).

New schedule enters

Let us consider what will happen if schedule *j* for some unit *i* enters into the solution with weight λ. Let $\mathbf{w}_+$ denote the new values of the weights of the current explicit basic schedules, let $\mathbf{z}_{b+}$ be the new values of the basic *z*-variables, and let $\mathbf{d}^*$ denote the difference:

$$\mathbf{d}^* = \mathbf{x}^{ij} - \mathbf{x}^{iJ(i)} \tag{6.30}$$

New value of the *x*-vector is denoted as $\mathbf{x}_+$ and is obtained as:

$$\mathbf{x}_+ = \mathbf{s} + \mathbf{D}\mathbf{w}_+ + \lambda\mathbf{d}^* \tag{6.31}$$

Binding constraints remain satisfied if (see Eq. 6.27):

$$\mathbf{A}_b\mathbf{D}\mathbf{w}_+ + \lambda\mathbf{A}_b\mathbf{d}^* + \mathbf{B}_{bb}\mathbf{z}_{b+} = \mathbf{c}_b - \mathbf{A}_b\mathbf{s} \tag{6.32}$$

or

$$(\mathbf{A}_b\mathbf{D} \quad \mathbf{B}_{bb})\begin{pmatrix}\mathbf{w}_+\\ \mathbf{z}_{b+}\end{pmatrix} = \mathbf{c}_b - \mathbf{A}_b\mathbf{s} - \lambda\mathbf{A}_b\mathbf{d}^* \tag{6.33}$$

Hence:

$$\begin{pmatrix}\mathbf{w}_+\\ \mathbf{z}_{b+}\end{pmatrix} = (\mathbf{A}_b\mathbf{D} \quad \mathbf{B}_{bb})^{-1}(\mathbf{c}_b - \mathbf{A}_b\mathbf{s} - \lambda\mathbf{A}_b\mathbf{d}^*) \tag{6.34}$$

or

$$\begin{pmatrix}\mathbf{w}_+\\ \mathbf{z}_{b+}\end{pmatrix} = \begin{pmatrix}\mathbf{w}\\ \mathbf{z}_b\end{pmatrix} - \lambda(\mathbf{A}_b\mathbf{D} \quad \mathbf{B}_{bb})^{-1}\mathbf{A}_b\mathbf{d}^* \tag{6.35}$$

Denote

$$\mathbf{H} = \begin{pmatrix} \mathbf{A}_b\mathbf{D} & \mathbf{B}_{bb} \end{pmatrix}^{-1} = \begin{pmatrix} \mathbf{H}_x \\ \mathbf{H}_z \end{pmatrix}, \tag{6.36}$$

where $\mathbf{H}_x$ contains P first rows and $\mathbf{H}_z$ Q last rows of $\mathbf{H}$.

Then

$$\mathbf{x}_+ = \mathbf{s} + \mathbf{D}\mathbf{w}_+ + \lambda\mathbf{d}^* = \mathbf{x} + \lambda\left(-\mathbf{D}\mathbf{H}_x\mathbf{A}_b + \mathbf{I}\right)\mathbf{d}^*, \text{ and} \tag{6.37}$$

$$\mathbf{z}_+ = \mathbf{z}_b - \lambda\mathbf{H}_z\mathbf{A}_b\mathbf{d}^* \tag{6.38}$$

Thus the new value of the objective function is

$$z_{0+} = z_0 + \lambda\left(\mathbf{a}_0'\left(-\mathbf{D}\mathbf{H}_x\mathbf{A}_b + \mathbf{I}\right) - \mathbf{b}_b'\mathbf{H}_z\mathbf{A}_b\right)\mathbf{d}^*, \text{ or} \tag{6.39}$$

$$z_{0+} = z_0 + \lambda\mathbf{v}_x'\mathbf{d}^*, \text{ where} \tag{6.40}$$

$$\mathbf{v}_x' = \mathbf{a}_0' - \mathbf{v}_c'\mathbf{A}_b, \text{ where} \tag{6.41}$$

$$\mathbf{v}_c' = \mathbf{a}_0'\mathbf{D}\mathbf{H}_x + \mathbf{b}_b'\mathbf{H}_z = \begin{pmatrix} \mathbf{a}_0'\mathbf{D} & \mathbf{b}_b' \end{pmatrix}\mathbf{H} \tag{6.42}$$

is the vector of shadow prices of the active constraints (more precisely, this vector is the shadow price vector at the solution).

If $\mathbf{v}_x'\mathbf{d}^* > 0$, or $\mathbf{v}_x'\mathbf{x}^{ij} > \mathbf{v}_x'\mathbf{x}^{iJ(i)}$, then the solution will improve if schedule j is put into the solution. Thus the value of a schedule in a unit can be computed using the marginal prices of x-variables. A nonbasic schedule can enter into solution if its value is greater than the value of the key schedule.

In the one-x formulation all elements on each row of $\mathbf{A}_b$ are zeros except possibly one element is one. Thus the computations simplify considerably. Each row of matrix $\mathbf{A}_b\mathbf{D}$ needed in the above formulas is either zero or is obtained by picking a row of **D.** In computing the pricing vector $\mathbf{v}_x$, we note that $\mathbf{a}_0$ is either zero or contains one in some position, and postmultiplication of vector $\mathbf{v}_c$ by $\mathbf{A}_b$ just adds the elements indicated by the columns of $\mathbf{A}_b$. Thus the total number of computations needed to compute $\mathbf{v}_x$ is very small.

New z-variable enters

Let us consider what will happen if a new z-variable, e.g. z_{Q+1} enters into the solution. Let λ be the new value of z_{Q+1}. Let $\mathbf{b}_b^*$ denote the coefficient (column) vector of z_{Q+1} in the binding constraints, and let b_0^* denote the coefficient of z_{Q+1} on row 0. The binding constraints remain satisfied if:

$$\mathbf{A}_b\mathbf{D}\mathbf{w}_+ + \mathbf{B}_{bb}\mathbf{z}_{b+} + \lambda\mathbf{b}_b^* = \mathbf{c}_b - \mathbf{A}_b\mathbf{s}\,. \tag{6.43}$$

We see that the equation is otherwise as Eq. (6.32) but $\mathbf{A}_b\mathbf{d}^*$ is replaced by $\mathbf{b}_b^*$. Thus

$$\begin{pmatrix}\mathbf{w}_+\\ \mathbf{z}_{b+}\end{pmatrix} = \begin{pmatrix}\mathbf{w}\\ \mathbf{z}_b\end{pmatrix} - \lambda\begin{pmatrix}\mathbf{A}_b\mathbf{D} & \mathbf{B}_{bb}\end{pmatrix}^{-1}\mathbf{b}_b^*\,. \tag{6.44}$$

In the case of the entering schedule, we had to take into account the direct effect of entering schedule on the *x*-variables. Now the values *x*-variables are changed only through the changed weights of schedules in the basis. Thus

$$\mathbf{x}_+ = \mathbf{s} + \mathbf{D}\mathbf{w}_+ = \mathbf{x} - \lambda\mathbf{D}\mathbf{H}_x\mathbf{b}_b^*, \text{ and} \tag{6.45}$$

$$\mathbf{z}_{b+} = \mathbf{z}_b - \lambda\mathbf{H}_z\mathbf{b}_b^* \tag{6.46}$$

The new value of the objective function is

$$z_{0+} = z_0 + \lambda b_0^* - \lambda\left(\mathbf{a}_0'\,\mathbf{D}\mathbf{H}_x + \mathbf{b}_b'\,\mathbf{H}_z\right)\mathbf{b}_b^*, \text{ or} \tag{6.47}$$

$$z_{0+} = z_0 + \lambda\left(b_0^* - \mathbf{v}_c'\,\mathbf{b}_b^*\right), \text{ where} \tag{6.48}$$

The shadow price vector $\mathbf{v}_c$ is given in (6.42).

Thus the objective function will increase if $b_0^* - \mathbf{v}_c'\,\mathbf{b}_b^* > 0$. If $b_0^* - \mathbf{v}_c'\,\mathbf{b}_b^* < 0$ then $\mathbf{v}_c'\,\mathbf{b}_b^* - b_0^*$ is the *reduced cost* that would result if the *z*-variable would be forced to the solution.

Slack or surplus variable enters

Assume that for some constraint *t* the upper bound C_t is binding and the lower bound c_t is strictly less than C_t. Then it may happen that when dropping the constraint, and letting the value of the row to decrease, the objective function may increase. An equivalent description for this is that the so called slack variable of constraint t enters to the solution. (JLP does not actually use slack and surplus variables, but they are useful for describing the situation when a binding constraint becomes nonbinding.) Thus the above analysis for the entering *z*-variable applies. The slack variable of constraint *t* is a *z*-variable so that

$$\mathbf{a}_t'\,\mathbf{x} + \mathbf{b}_t'\,\mathbf{z} + \text{slack}_t = C_t \tag{6.49a}$$

i.e. it has coefficient one on row t and zero on other rows. The objective function can be inreased by relaxing the constraint *t* if element *t* of $\mathbf{v}_c$ is negative.

Similarly, if constraint t is at the lower bound c_t and $c_t < C_t$, we should consider entering the surplus variable for constraint t into the solution. The surplus variable of constraint t is a z-variable so that

$$\mathbf{a}_t' \mathbf{x} + \mathbf{b}_t' \mathbf{z} - \text{surplus}_t = c_t \tag{6.49b}$$

i.e., it has coefficient -1 on row t and zero on other rows. Thus the objective function can be inreased by relaxing the constraint t if element t of $\mathbf{v}_c$ is positive.

6.2.3 Leaving variable

When a new variable enters into the solution, the objective function increases in proportion to the new value λ of the entering variable. The new value will be increased until some basic variable becomes zero. That variable then leaves the basis. Three cases may occur:

(i) The weight $w_{iJ(i)}$ of the key schedule of some unit i (i may or may not be the same unit for an entering schedule) becomes zero. Note that $w_{iJ(i)}$ is not formally a basic variable of the modified problem. This is equivalent to the case that an implicitly treated constraint $\sum_{j \neq J(i)} w_{ij} \leq 1$ becomes binding.

(ii) The weight w_{ij} of an explicit basic schedule will leave the basis.

iii) A basic z-variable leaves the basis

(iv) A nonbinding utility constraint t, $R<t\leq r$ will become binding (at lower or upper bound).

To determine which of the three cases occurs, we need to compute the critical value λ^* in each case. Let us first present in a unifying formalism how the w-, z-, and x-variables change when a new variable enters:

$$\mathbf{w}_+ = \mathbf{w} + \lambda \mathbf{r}_w \tag{6.50}$$

$$\mathbf{x}_+ = \mathbf{x} + \lambda \mathbf{r}_x \tag{6.51}$$

$$\mathbf{z}_+ = \mathbf{z} + \lambda \mathbf{r}_z \tag{6.52}$$

where

$$\mathbf{r}_w = \begin{cases} -\mathbf{H}_x\mathbf{A}_b\mathbf{d}^*, & \text{if a new schedule enters} \\ -\mathbf{H}_x\mathbf{b}_b^*, & \text{if a new } z\text{ - variable enters} \\ t^{\text{th}} \text{ column of } -\mathbf{H}_x & \text{if surplus variable of constraint } t \text{ enters} \\ t^{\text{th}} \text{ column of } \mathbf{H}_x & \text{if slack variable of constraint } t \text{ enters} \end{cases} \tag{6.53}$$

$$\mathbf{r}_x = \begin{cases} \left(-\mathbf{D}\mathbf{H}_x\mathbf{A}_b + \mathbf{I}\right)\mathbf{d}^*, & \text{if a new schedule enters} \\ -\mathbf{D}\mathbf{H}_x\mathbf{b}_b^*, & \text{if a new } z\text{ - variable enters} \\ t^{\text{th}} \text{ column of } -\mathbf{D}\mathbf{H}_x & \text{if surplus variable of constraint } t \text{ enters} \\ t^{\text{th}} \text{ column of } \mathbf{D}\mathbf{H}_x & \text{if slack variable of constraint } t \text{ enters} \end{cases} \tag{6.54}$$

$$\mathbf{r}_z = \begin{pmatrix} \mathbf{r}_{zb} \\ \mathbf{r}_{zn} \end{pmatrix}, \text{ where} \tag{6.55}$$

$$\mathbf{r}_{zb} = \begin{cases} -\mathbf{H}_z\mathbf{A}_b\mathbf{d}^*, & \text{if a new schedule enters} \\ -\mathbf{H}_z\mathbf{b}_b^*, & \text{if a new } z\text{ - variable enters} \\ t^{\text{th}} \text{ column of } -\mathbf{H}_z & \text{if surplus variable of constraint } t \text{ enters} \\ t^{\text{th}} \text{ column of } \mathbf{H}_z & \text{if slack variable of constraint } t \text{ enters} \end{cases} \tag{6.56}$$

and all elements of $\mathbf{r}_{zn}$ are zero except if a z -variable enters then the corresponding element is one (e.g. if z_{Q+1} enters then first element of $\mathbf{r}_{zn}$ is one).

We need then consider the following cases:

The weight of a key schedule becomes zero

The weight $w_{iJ(i)}$ of a key schedule becomes zero when the weights of basic schedules of unit i sum up to one, i.e., the implicit constraint $\sum_{j \neq J(i)} w_{ij} \leq 1$ becomes binding.

Denote by T_i the index set of explicit basic schedules from treatment unit i (i.e., $j \in T_i$ means that $w_{ij} > 0$ and $j \neq J(i)$). Let r_{wij} denote the corresponding element of $\mathbf{r}_w$.

Let us first concider the case that the entering variable is not weight w_{ij} in unit i. We note first that the weight $w_{iJ(i)}$ cannot become zero if there are no explicit basic schedules in unit i (i.e., T_i is empty). If there are explicit basic schedules in unit i (i.e. T_i is not empty), then the weights of the explicit basic schedules sums up to one if

$$\sum_{j \in T_i} \left(w_{ij} + \lambda^* r_{wij}\right) = 1, \text{ or} \tag{6.57}$$

$$\lambda^* = \left(1 - \sum_{j \in T_i} w_{ij}\right) \Big/ \sum_{j \in T_i} r_{wij} \tag{6.58}$$

If the entering variable is weight w_{ij} in unit *i*, the weights of the previous explicit basic schedules and the weight of the entering schedule sum up to one if

$$\sum_{j \in T_i} \left(w_{ij} + \lambda^* r_{wij} \right) + \lambda^* = 1, \text{ or} \tag{6.59}$$

$$\lambda^* = \left(1 - \sum_{j \in T_i} w_{ij} \right) \Big/ \left(1 + \sum_{j \in T_i} r_{w\,ij} \right) \tag{6.60}$$

If there were no explicit basic schedules in the unit of the entering schedule, then the above equation says simply that $w_{iJ(i)}$ becomes zero if $\lambda^*=1$.

An explicit basic schedules leaves

The weight of an explicit basic schedule, w_{ij} , becomes zero, if $r_{wij}<0$ and

$$w_{ij} + \lambda^* r_{wij} = 0, \text{ or} \tag{6.61}$$

$$\lambda^* = -w_{ij} / r_{wij} \tag{6.62}$$

A basic *z*-variable leaves

A basic *z*-variable z_{bk} becomes zero if $r_{zk} < 0$ and

$$z_{bk} + \lambda^* r_{zk} = 0, \text{ or} \tag{6.63}$$

$$\lambda^* = -z_{bk} / r_{zk} \tag{6.64}$$

A nonbinding constraint becomes binding (a slack or surplus variable leaves)

Let Z_t denote the current value of a nonbinding constraint row *t*:

$$Z_t = \mathbf{a}_t' \mathbf{x} + \mathbf{b}_t' \mathbf{z} \quad \text{, and} \tag{6.65}$$

$c_t < Z_t < C_t$. The new value of the row, denoted as Z_{t+}, will be

$$Z_{t+} = Z_t + \lambda \left(\mathbf{a}_t' \mathbf{r}_x + \mathbf{b}_t' \mathbf{r}_z \right). \tag{6.66}$$

If $\mathbf{a}_t' \mathbf{r}_x + \mathbf{b}_t' \mathbf{r}_z < 0$, the constraint will reach the lower bound c_t when λ gets value

$$\lambda^* = \left(c_t - Z_t \right) / \left(\mathbf{a}_t' \mathbf{r}_x + \mathbf{b}_t' \mathbf{r}_z \right). \tag{6.67}$$

Similarly, if $\mathbf{a}_t' \mathbf{r}_x + \mathbf{b}_t' \mathbf{r}_z > 0$, the constraint will reach the upper bound C_t when λ gets value

$$\lambda^* = (C_t - Z_t) / (\mathbf{a}_t' \mathbf{r}_x + \mathbf{b}_t' \mathbf{r}_z). \qquad (6.68)$$

Note that the elements of $\mathbf{r}_z$ corresponding to nonbasic *z*-variables are zero except for an entering *z*-variable. The smallest value of λ^* computed in Eqs. 6.58, 6.60, 6.62, 6.64, 6.67, and 6.68 will be the new value of the entering variable (weight w_{ij}, *z*-variable, slack/surplus variable) and it determines which is the leaving basic variable (a key variable, an explicit basic variable, a *z*-variable, or an implicit slack/surplus variable of a nonbinding constraint). Thereafter we need to update the problem description, i.e. the list of key schedules, the list of explicit basic schedules, and **s**, **D**, $\mathbf{B}_{bb}$, **H**, **w**, **z**, **x,** $\mathbf{v}_c$, and $\mathbf{v}_x$.

6.2.4 Updating step

There are three different types of entering variables (treating slack and surplus variables as one category), and four different types of leaving variables. Thus there are twelve different combinations. The overall updating step can be combined by applying the following operations:

The weight of a key schedule becomes zero

The updating steps are simple, if weight w_{ij} enters the solution and the weight $w_{iJ(i)}$ of the key schedule of the same unit *i* leaves the solution, and there are no explicit basic schedules for the unit (i.e., w_{ij} will become 1 and $w_{iJ(i)}$ was 1). We first update **s** (:= denotes assignmet operation):

$$\mathbf{s} := \mathbf{s} - \mathbf{x}^{iJ(i)} + \mathbf{x}^{ij} \qquad (6.69)$$

Then we set *J*(*i*) := *j* in the list of key schedules. **D**, $\mathbf{B}_{bb}$, **H**, $\mathbf{v}_x$, and $\mathbf{v}_c$ will remain the same. New **w**, **z**, and **x** can be computed using Eqs. (6.29) and (6.19).

If $w_{iJ(i)}$ becomes zero for a unit *i* having explicit basic schedules, then the updating can be done as follows. We select any explicit basic schedule *j'* in unit *i* to become the new key schedule. Vector **s** is updated similarly as in (6.69). If there are other explicit basic schedules in the unit (in addition to the new key schedule), the columns of **D** for other schedules in the unit are changed to correspond to the new key schedule. The inverse **H** of the basis can be updated accordingly by standard pivot operations. *J(i)* is set to be *j'*. Thereafter we proceed as if it where the column of **D** corresponding to the schedule *j'* that is leaving the basis.

A column of the basis is changed

A colum of the basis is changed when the entering variable is either *w*- or *z*-variable and the leaving variable is either *w*- or *z*-variable. If the leaving variable is the weight $w_{iJ(i)}$) of the key schedule (i.e. an implicit basic variable), it was described in the previous section what steps are taken to transform the situation to correspond the case that the leaving variable is w_{ij} of an explicit basic schedule.

If the leaving variable is w_{ij} for an explicit basic schedule, the corresponding column of **D** is dropped. If the leaving variable is a *z*-variable, then the corresponding column of $\mathbf{B}_{bb}$ is dropped. If the entering variable is a *z*-variable z_k then the coefficient vector

$$\begin{pmatrix} b_{1k} \\ b_{2k} \\ \vdots \\ b_{Rk} \end{pmatrix}$$

is included in $\mathbf{B}_{bb}$. If the entering variable is w_{ij} then vector $\mathbf{d}^* = \mathbf{x}^{ij} - \mathbf{x}^{iJ(i)}$ is joined to matrix **D**. Thereafter the inverse of the basis **H** is updated using standard pivot operations. For computing the inverse, the basis is treated as a single matrix whose column is changed. Logical separation between $\mathbf{B}_{bb}$ and **D** is done with link lists.

A row is added to the basis

If either a *z*- variable or w_{ij} of an explicit basic schedule is entering the basis and a new constraint *t* ,

$$c_t \le \mathbf{a}_t' \mathbf{x} + \mathbf{b}_t' \mathbf{z} \le C_t \tag{6.70}$$

t>*R*, becomes active, then the dimension of the basis is inreased by one. Then coefficient rows $\mathbf{a}_t'$ and $\mathbf{b}_t'$ that has been in the nonbasic (lower) part of **A** and **B** in (6.22) and (6.23) are moved to the basic (upper) part. If the entering variable is a *z*-variable z_k then the corresponding column vector of coefficients is included in $\mathbf{B}_{bb}$. If the entering variable is w_{ij} then vector $\mathbf{d}^* = \mathbf{x}^{ij} - \mathbf{x}^{iJ(i)}$ is joined to the matrix **D**. Thus the basis matrix $(\mathbf{A}_b\mathbf{D} \quad \mathbf{B}_{bb})$ is updated by adding both a new row and a new column to it. The inverse basis **H** can be updated using the matrix formula (CRC ... 1981)

$$\begin{pmatrix} c & \mathbf{d}' \\ \mathbf{b} & \mathbf{A} \end{pmatrix}^{-1} = \begin{pmatrix} h & h\mathbf{d}'\mathbf{A}^{-1} \\ -h\mathbf{A}^{-1}\mathbf{b} & \mathbf{A}^{-1} + h\mathbf{A}^{-1}\mathbf{b}\mathbf{d}'\mathbf{A}^{-1} \end{pmatrix}, \tag{6.71}$$

where

$$h = 1/\left(c - \mathbf{d}'\mathbf{A}^{-1}\mathbf{b}\right) \tag{6.72}$$

A row is dropped from the basis

If the implicit slack or surplus variable of constraint t is entering the solution (a binding constraint t becomes nonbinding), and either a z- variable or w_{ij} of an explicit basic schedule is leaving the basis, then we reduce the dimension of the basis by removing a column and a row. If w_{ij} is leaving the solution the corresponding column of the matrix **D** is dropped. If the leaving variable is a z-variable, then the corresponding column of $\mathbf{B}_{bb}$ is dropped. Thereafter the row t is classified as a nonbinding both in matrix **A** and **B**. The inverse of the basis is updated using the matrix formula (this can be derived from formulas given in CRC ... 1981):

If

$$\begin{pmatrix} \mathbf{B} & \mathbf{c} \\ \mathbf{d}' & e \end{pmatrix}^{-1} = \begin{pmatrix} \mathbf{X} & \mathbf{y} \\ \mathbf{z}' & u \end{pmatrix} \tag{6.73}$$

then

$$\mathbf{B}^{-1} = \mathbf{X} - \tfrac{1}{u}\mathbf{y}\mathbf{z}' \tag{6.74}$$

Two rows of the basis are changed

If the implicit slack or surplus variable of constraint t is entering the solution (a binding constraint t becomes nonbinding), and the implicit slack or surplus variable of an other constraint is leaving the basis, then we interchange the status of the corresponding rows in matrices **A** and **B**. The inverse of the basis $(\mathbf{A}_b\mathbf{D} \quad \mathbf{B}_{bb})$ is then obtained by standard (row) pivot operations.

Computations after changing the basis

After updating **s**, $\mathbf{A}_b$, **D**, $\mathbf{B}_{bb}$ and **H**, new values of **w**, **z**, **x**, $\mathbf{v}_c$ and $\mathbf{v}_x$ are computed using Eqs. 6.29. 6.19, 6.42 and 6.41. Then JLP tries to improve the solution by entering a new z-variable, a slack/surplus variable of a binding constraint or a schedule.

6.3 Optimization Algorithm

The preceeding chapters described briefly the mathematical basis of the generalized upper bound method as applied in JLP. This chapter describes some properties of the implementation of the method.

6.3.1 Minimization

The algorithm is described above for the case where we want to maximize the objective function. If the problem is defined initially as a minimization problem, an equivalent

maximization problem is obtaint by changing the signs of coefficients on the objective row. The signs needs to be taken into account only in Eqs. 6.42, 6.41 and 6.48.

6.3.2 Summary of the algorithm

The following symbols are used in addition of symbols defined in Chapters 6.1 or 1.2:
g = the current (temporary) objective row (after finding feasible g=0)
L_n = list of nonbinding constraints
L:=L $\cup$ {t} means that t is added to the list L

Finding a feasible solution

JLP finds a feasible solution by maximizing or minimizing each constraint row until it will reach the feasible range [c_t, C_t]. In the following it is summarized how the algorithm is used to find the feasible solution.

Initialization: Get lower and upper bounds, and get for each unit the key schedule (for first problem with the data, key schedules are just different schedule numbers, thereafter key schedules are obtained from the previous solution), compute **s** (which is also the initial value of **x**) using key schedules.

0. Set g:=0; L_n:={ }

1. If g=r , then EXIT, FEASIBLE FOUND
else
g:=g+1; L_n:= $L_n \cup$ {g};
If constraint g is satisfied go to 1

2. Find an entering variable when row g is maximized ($\mathbf{a}_g'\mathbf{x} + \mathbf{b}_g'\mathbf{z} < c_g$) or minimized ($\mathbf{a}_g'\mathbf{x} + \mathbf{b}_g'\mathbf{z} > C_g$). If no variable can enter, then EXIT, PROBLEM INFEASIBLE

3. Find the leaving variable, make one optimization step.

4. If constraint g is satisfied, then go to 1, else go to 2

The reason for adding constraint g into the list of nonbinding constraints in same time as we begin to maximize or minimize row g is to prevent the possibility that the row that is smaller than the lower bound (greater than the upper bound) and will become greater than the upper bound (smaller than the lower bound) in one optimization step. The algorithm became more efficient than the basic version described above with the following modification. Each time a new constraint g is started, all constraints g+1,....,r

are inspected if they are already in the feasible range, and satisfied constraints are added to the list of nonbinding constraints to prevent them to become unsatisfied when making constraint g feasible. Also such option to the algorithm was tested that temporary lower or upper bounds were used for constraints g+1,....,r to prevent them deviate more from their feasible ranges. No clear speed advantage was found, and this option is no more available.

Finding optimal solution

After finding a feasible solution, the optimum value for row g=0 can be found simply as follows:

1. Find an entering variable. If no variable can enter, then EXIT, SOLUTION.
2. Find the leaving variable, make one optimization step.
3. Go to step 1.

There are different possible stategies for finding the next entering variable. JLP is using the following one.

How JLP selects the entering variable

A linear programming algorithm has found the solution, if the current solution cannot be increased by any entering variable. If several variables can enter, the solution will be found if any strategy is used to select the entering variable. Selection strategy affects of course the speed of the algorithm. JLP selects the entering variable initially and after each change of the basis according to the following priority order (i.e. the entering variable is selected from the highest possible category):

i) z-variables
ii) Slack or surplus variables
iii) Weights of schedules.

If several z-variables (slack/surplus variables) can enter, then the z-variable (slack/surplus variable) resulting in highest marginal change in the objective function is chosen. Units are visited in order when it is checked if a weight w_{ij} can enter into the solution. The values of all schedules in a unit are computed, and schedule with largest value is entered into the solution if its value is greater than the value of the key schedule. If a weight w_{ij} enters, then next time JLP computes prices of schedules it starts from unit i+1. If i+1 is greater than the number of units, then the first unit will be the next unit. If no schedule can enter in the unit where last weight entered the solution, then it is known that the optimum has been found.

If, after entering w_{ij} into the solution, JLP would return to the same unit i for calculating the prices of schedules, JLP would find the optimum for the current unit. In the language of decomposition algorithms: we would find the optimum for a subproblem. In test problems, it was found slightly more efficient to go to the next unit i+1 after entering a weight w_{ij} .

If there are no x-variables in the problem, then JLP just never reaches the phase iii) where prices of schedules are computed.

6.4 Dual Analysis

It may give insight to the problem if we analyze how the primal problem and the dual problem are related. This analysis will also indicate how to desribe marginal properties of the solution.

6.4.1 Primal problem

Let us first rewrite the 'standard' problem formulation by separating the lower bound and upper bound constraints:

$$\text{Max} \quad z_0 = \mathbf{a}_0' \mathbf{x} + \mathbf{b}_0' \mathbf{z} \tag{6.75}$$

subject to:

$$\sum_{k=1}^{p} a_{tk} x_k + \sum_{k=1}^{q} b_{tk} z_k \le C_t, \quad t = 1,\ldots,r \tag{6.76}$$

$$-\sum_{k=1}^{p} a_{tk} x_k - \sum_{k=1}^{q} b_{tk} z_k \le -c_t, \quad t = 1,\ldots,r \tag{6.77}$$

$$x_k - \sum_{i=1}^{m} \sum_{j=1}^{n_i} x_k^{ij} w_{ij} = 0, \quad k = 1,\ldots,p \tag{6.78}$$

$$\sum_{j=1}^{n_i} w_{ij} = 1, \quad i = 1,\ldots,m \tag{6.79}$$

$$w_{ij} \ge 0 \quad \text{for all } i \text{ and } j \tag{6.80}$$

$$z_k \ge 0 \quad \text{for } k = 1,\ldots,q \tag{6.81}$$

6.4.2 Dual problem

The dual problem is first defined using new symbols for the dual variables. It is then indicated at the end of the chapter how the dual variables are related to quantities computed when the primal problem is solved.

Let Φ_t, k=1,...,r be the shadow prices for upper bound constraints (6.76), let ϕ_t, t=1,...,r be the shadow prices for lower bound constraints (6.77), let μ_k, k=1,...,p, be the shadow prices of constraints (6.78), and let δ_i, i=1,...m be the shadow prices of constraints (6.79). The dual problem is then:

$$\text{Min} \sum_{t=1}^{r} C_t \Phi_t - \sum_{t=1}^{r} c_t \phi_t + \sum_{k=1}^{p} 0 \mu_k + \sum_{i=1}^{m} \delta_i \tag{6.82}$$

or after dropping the third term (which is zero)

$$\text{Min} \sum_{t=1}^{r} C_t \Phi_t - \sum_{t=1}^{r} c_t \phi_t + \sum_{i=1}^{m} \delta_i \tag{6.83}$$

subject to:

$$\sum_{t=1}^{r} a_{tk} (\Phi_t - \phi_t) + \mu_k = a_{0k}, \quad k = 1, \ldots, p \tag{6.84}$$

$$\sum_{t=1}^{r} b_{tk} (\Phi_t - \phi_t) \geq b_{0k}, \quad k = 1, \ldots, q \tag{6.85}$$

$$-\sum_{k=1}^{p} x_k^{ij} \mu_k + \delta_i \geq 0, \quad \text{for all } i \text{ and } j \tag{6.86}$$

$$\Phi_t \geq 0 \text{ for } t=1,...,r \tag{6.87}$$

$$\phi_t \geq 0 \text{ for } t=1,...,r \tag{6.88}$$

Because (6.78) and (6.79) are equality constraints, the corresponding dual variables μ and δ are free variables. Because x_k-variables are free in the primal problem, the corresponding constraints in the dual (6.84) are equality constraints.

Optimization of the objective variable of the primal problem can be described as a process of finding a feasible solution for the dual problem. For instance, if the pricing rule (6.40) tells that a schedule j' in unit i could be included in the solution, this indicates that constraint (6.86) in the dual problem is not satisfied. After the schedule (precisely weight w_{ij}) enters the solution of the primal, the constraint is satisfied.

6.4.3 Relations between primal and dual problems

Let us then consider what kind of relations there are between shadow prices and variables in the primal problem in the optimal solution.

Shadow price of an *x*-variable

For an original utility constraint t either the lower bound c_t or the upper bound C_t is active. Thus either $\Phi_t=0$ or $\phi_t=0$ or both are zero. Let us write that

$$\pi_t = \Phi_t - \phi_t \,. \tag{6.89}$$

Then the shadow prices of *x*-variables can be obtained from (6.84) :

$$\mu_k = a_{0k} - \sum_{t=1}^{r} a_{tk}\pi_t, \quad k = 1,..,p \tag{6.90}$$

Shadow price of a unit

Constraint (6.86) can be written as:

$$\delta_i \geq \sum_{k=1}^{p} x_k^{ij}\mu_k, \quad \text{for all } i \text{ and } j \tag{6.91}$$

Equality holds, if the weight w_{ij} in the primal problem is a basic variable ($w_{ij}>0$), i.e., the shadow price of a unit is equal to the value of any basic schedule calculated using the shadow prices of the *x*-variables. Multiplying with w_{ij} we get an equality for each w_{ij} :

$$w_{ij}\delta_i = \sum_{k=1}^{p} w_{ij}x_k^{ij}\mu_k, \quad \text{for all } i \text{ and } j \tag{6.92}$$

If add up over all all schedules in unit *i*, we get

$$\delta_i = \sum_{k=1}^{p} x_k^{i}\mu_k, \quad \text{for all } i \tag{6.93}$$

where

$$x_k^i = \sum_{j=1}^{n_i} w_{ij}x_k^{ij} \,. \tag{6.94}$$

Thus the shadow price of a unit can be calculated using any of the basic schedules

Reduced cost of a nonbasic schedule

The reduced cost for forcing schedule *j* in unit *i* into the solution is:

$$\delta_i - \sum_{k=1}^{p} x_k^{ij} \mu_k . \tag{6.95}$$

Reduced cost for a nonbasic *z*-variable

Reduced cost for a *z*-variable z_k is obtained from (6.85) and (6.89) as:

$$\sum_{t=1}^{r} b_{tk}(\Phi_t - \phi_t) - b_{0k} = \sum_{t=1}^{r} b_{tk}\pi_t - b_{0k}, \quad k = 1,\ldots, q \tag{6.98}$$

Objective function of the dual

The objective function of the dual (6.83) is equal to:

$$z_0 = \sum_{t=1}^{r} C_t\Phi_t - \sum_{t=1}^{r} c_t\phi_t + \sum_{i=1}^{m} \delta_i$$

$$= \sum_{t=1}^{r} C_t\Phi_t - \sum_{t=1}^{r} c_t\phi_t + \sum_{i=1}^{m}\sum_{k=1}^{p} x_k^i \mu_k \qquad \text{(from 6.93)}$$

$$= \sum_{t=1}^{r} C_t\Phi_t - \sum_{t=1}^{r} c_t\phi_t + \sum_{k=1}^{p} x_k \mu_k \tag{6.96}$$

$$= \sum_{t=1}^{r} C_t\Phi_t - \sum_{t=1}^{r} c_t\phi_t + \sum_{k=1}^{p} x_k \left(a_{0k} - \sum_{t=1}^{r} a_{tk}\pi_t \right) \qquad \text{(from 6.90, use then 6.89)}$$

$$z_0 = \sum_{k=1}^{p} x_k a_{0k} + \sum_{t=1}^{r} \Phi_t \left(C_t - \sum_{k=1}^{p} a_{tk} x_k \right) + \sum_{t=1}^{r} \phi_t \left(\sum_{k=1}^{p} a_{tk} x_k - c_t \right) \tag{6.97}$$

If there are no *z*-variables in the problem, then for each *t* either Φ_t is zero (the upper bound constraint *t* is nonbinding) or $C_t - \sum_{k=1}^{p} a_{tk} x_k$ is zero, and similarly, ϕ_t is zero (the lower bound constraint *t* is nonbinding) or $\sum_{k=1}^{p} a_{tk} x_k - c_t$ is zero. Thus if there are no *z*-variables, the last two terms term in (6.97) are always zero.

Computation of the shadow prices

The nonzero values of π_t, t=1,...,r, i.e., the shadow prices of utility constraints, are obtained from vector $\mathbf{v}_c$ in Eq. (6.42). During the solution process $\mathbf{v}_c$ is always up to date. If π_t>0, then $\Phi_t = \pi_t$, and if π_t<0, then $\phi_t = -\pi_t$.

During the optimization, the shadow prices of x-variables are calculated from the shadow prices of utility constraints using (6.41). If there are linear combinations of x-variables on the rows of the problem, these shadow prices are for the temporary x-variables presenting the linear combinations. After solving the problem, the shadow prices of the original x-variables can be computed using (6.90).

During the optimization process, values of schedules are computed using (6.40) when it is determined if a schedule should enter into the solution. The prices are not stored. After finding the solution, the shadow price of a unit i (precisely, the shadow price of the area constraint for unit i) can be computed applying (6.91, with equality sign) to the key schedule of the unit (or to the explicit basic schedules). Thereafter the reduced cost of a nonbasic schedule can be computed with (6.95).

Formula (6.48) used to check if a z-variable could enter into the solution is essentially the same as the reduced cost of a nonbasic z-variable given in (6.98). Values computed with (6.48) are not stored, beacause they are trivial to recompute with (6.98) after finding the solution.

The fact that the optimization of the primal problem is essentially a r-dimensional problem (r is the number of the utility constraints) is reflected in the dual problem so that after knowing the shadow prices of the utility constraints, other dual variables can be directly computed from them.

6.4.4 Cost of changing values of x-variables

In this section we study the marginal changes in the objective function, if we add a constraint to the problem that requires that an x-variable x_k gets a value slightly different from the value computed with the weights w_{ij} obtained from the optimal solution. Variable x_k may or may not be present in the original problem.

Assume that according to the optimal solution x_k has the value:

$$x_k = \xi_k \tag{6.99}$$

Assume then that the problem is modified by adding a constraint

$$x_k = \xi_k + \varepsilon \tag{6.100}$$

or a constraint

$$x_k = \xi_k - \varepsilon, \tag{6.101}$$

where ε is a small positive constant. The constraint (6.100) is called *constraint for increase* and the relative change of the objective function (i.e. the change in z_0 divided by ε) is called the *cost of increase*. Similarly, constraint (6.101) is *constraint for decrease* , and the corresponding relative change is *cost of decrease*. If x_k was not present in the problem, then we need to add also a constraint of type (6.78) that defines x_k. This constraint is treated implicitly as before. The changes in the objective function can be analyzed as follows.

If ε is small enough, then a solution satisfying the new additional constraint (6.100) or (6.101) can be reached in one step from the current optimal solution by entering a new *w*-, *z*- or slack/surplus variable into the basis. As the number of constraints is increased by one, no variable is leaving the basis. The optimal entering variable can be chosen by applying the same formulas that are used to determine the entering variable during the optimization (in this discussion 'entering variable' refers to a variable that *could* be entered into the basis, actually no computations are made to change the basis). If a potential entering variable y gets value λ, (λ>0) then the change in the objective function is $\alpha(y)\lambda$, where $\alpha(y)$ is computed with Eq. (6.40) if y *is* a *w*-variable (i.e. $\alpha = \mathbf{v}_x' \mathbf{d}^*$), and with Eq (6.48) if y is a *z*-variable or a slack/surplus variable (i.e. $\alpha = b_0^* - \mathbf{v}_c' \mathbf{b}_b^*$, or α= ± an element of $\mathbf{v}_c$) Because we start from the optimal solution with less restrictive constraints, $\alpha(y)$ is always nonpositive (when the objective function is maximized).

The corresponding changes in the variable x_k can be analyzed using the same formulas treating x_k as the objective function. When computing the price vector $\mathbf{v}_c' = (\mathbf{a}_0' \mathbf{D} \quad \mathbf{b}_b')\mathbf{H}$ using Eq. (6.42) we note that $\mathbf{a}_0' \mathbf{D}$ is a vector with elements $x_k^{ij} - x_k^{iJ(i)}$ and $\mathbf{b}_b$ is zero. The change in x_k can be expressed as $\beta(y)\lambda$, where $\beta(y)$ can be zero, positive or negative.

The constraint for increase (6.100) will be satified if $\beta(y)$>0 and

$$\beta(y)\lambda = \varepsilon, \quad \text{or} \tag{6.102}$$

$$\lambda = \varepsilon / \beta(y). \tag{6.103}$$

The objective function will change by the amount:

$$\alpha(y)\lambda = \varepsilon\,\alpha(y)/\beta(y)\,. \qquad (6.104)$$

Thus the optimal entering variable is such that $-\alpha(y)/\beta(y)$ is as small as possible. Ratio $-\alpha(y)/\beta(y)$ is the marginal cost of forcing x_k to increase.

Similarly, the optimal entering variable for the constraint for increase (6.101) is such that $\beta(y)<0$ and $\alpha(y)/\beta(y)$ is as small as possible. Ratio $\alpha(y)/\beta(y)$ is the marginal cost of forcing x_k to decrease.

If the optimal solution is not unique, then it is possible that cost of decrease or decrease is zero (i.e., $\alpha(y)$ may be zero for an entering variable y for which $\beta(y)$ is nonzero). If ξ_k is the largest value that x_k can have, then $\beta(y)$ is never greater than zero, and the problem with the additional constraint is infeasible. Thus the cost of increase is not defined (or can defined to be infinite). Similarly, ξ_k may be the smallest value x_k can have. The JLP printout `'INF'` can thus interpreted to mean that the problem with the corresponding additional constraint is *infeasible* or that the cost is *infinite*. Generally, the cost of increase is different from the cost of decrease. For instance, if the net present is maximized, then the requirement to decrease cuttings in the first period costs usually more than the requirement to increase cuttings.

The cost of increase and decrease can be computed also for domain specific x-variables (even if the domains were not used in the problem). If the entering variable y is a z- or slack/surplus variable, then ratio $\alpha(y)/\beta(y)$ is the same in each domain.

The shadow price of an x-variable x_k tells what is the marginal change of the objective function, if the problem remains the same and we get an extra unit of x_k. If the shadow price of an x-variable is zero (i.e. the x-variable is present only in a nonbinding constraint) or an x-variable does not appear in the problem, then a marginal change in the x-variable does not cause any change in the objective function. The cost of change gives a different view of the marginal properties of the solution. The main difference is that the shadow prices are computed assuming that only the right hand sides of constraints change, while when the cost of change is computed, a constraint is added.

The cost of changing the value of a nonbasic x-variable is easier to interpret than the cost of change of a basic x-variable (a basic x-variable is an x-variable present on the objective row or in a binding utility constraint). For a basic x-variable, the additional constraint for change may intervene with the previous constraints in a way that may be not seen directly. The mathematical connections between shadow prices and costs of changing the values of x-variables will not be analyzed further in this report.

6.5 Domains

Domains are not assumed to have any specific structure. For instance domains can overlap in any manner. Thus it did not seem to be possible to apply decomposition techniques to take advantage of the domain structure in optimization algorithm as such. The domains are taken into account in the computation process as follows.

All *x*-variables are stored in one matrix without domain information. When the problem and domains are defined in a **`problem`** paragraph, JLP classifies treatment units into domain combinations. All units in a domain combination belong exactly to the same domains (a unit can belong to any number of domains). A logical vector is created for each domain combination telling what constraints apply in the domain combination. When JLP is dealing with a unit, the loops go over the constraints that apply in that domain combination.

Domain structure for constraints can be taken into account using any linear programming program by defining coefficients x_k^{ij} so that they are zero outside the domain. This would lead to a system where the same non-zero numbers are stored several times and many zeros are also stored. JLP stores coefficients x_k^{ij} only once and does not store extra zeros.

Concluding Remarks: Future Developments

It was a very difficult to decide at what point to stop the development of the JLP program and make a first version for a general use. There are several possible extensions, and some of them would be quite easy to implement. It is best that users decide what extensions are included in future versions of the program. For instance, following additions might interest some users:

a) Objective function could be nonlinear with respect to the *x*-variables. This would quite easy to implement. Other nonlinear properties would require more work.

b) Now all transformation definitions are cleared after transformations are computed (except `dtran` which are computed always *in place*). If definitions could be saved, it would be possible to have parameters in definitions and change their values with **`constant`** command without needing to retype (or recall from include file) transformation definitions.

c) It is not currently possible to merge several data sets saved in JLP-format. Is there need for it?

d) It is not currently possible to edit afterwards problem paragraph or transformation definitions. This is not a great shortage in modern computer environments where one can switch to an other editor program and then return to JLP and get edited commands using **`include`** command.

e) JLP algorithm is built so that dimension of the basis (the number of binding constraints) can change any time. Thus it would be easy to add to JLP the property that the user could add or remove constraints from the previous problem, and the optimization would start from the previous solution. Now the algorithm is utilizing the previous solution only by using the set of the key schedules of the previous solution. An earlier version of the program included these capabilities, and it was found that in small problems no significant improvement in speed was obtained. The situation may be different in large problems.

References

CRC Standard mathematical tables. 25th Edition. CRC Press, Boca Raton, Florida. 613 p.

Dantzig, G. B. and VanSlyke, R. M. 1967. Generalized upper bounding techniques. Journal of Computer and System Sciences 1:213-226.

Dykstra, D. P. 1984. Mathematical programming for natural resource management. McGraw-Hill. New York. 318 p.

Kilkki, P. 1987. Timber management planning. 2nd edition. Silva Carelica 5. 160 p. University of Joensuu. Faculty of Forestry.

Lappi, J. and Siitonen, M. 1985. A utility model for timber production based on different interest rates for loans and savings. Silva Fennica 19(3):271-280.

Luenberger D. G. 1973. Introduction to linear and nonlinear programming. Addison-Wesley, Reading. Mass. 356 p.

Press, W. H., Flannery, B. P., Teukolsky, S. A. and Vetterling, W. T. 1986. Numerical recipes: The art of scientific computing. Cambridge University Press. 818 p.

Siitonen, M. 1983. A long term forestry planning system based on data from the Finnish national forest inventory. Forest Inventory for improved Management, Proc. of the IUFRO Subject Group 4.02 Meeting in Finland, September 5-6 1983. Univ. of Helsinki. Department of Forest Mensuaration and Management. Res. Notes 17: 195-207.

Steuer, R.E. 1986. Multiple criteria optimization: Theory, computation, and application. John Wiley. New York. 546 p.

List of Commands

Current commands (significant part is underlineed):

batch	buff	buflevel	cdata	cform
const	ctran	cvar	do	dtran
dupl	end	enddo	feasible	help
helpfile	include	init	keepc	keepx
make	mrep	outfile	outlevel	own1
own2	ownread	parin	parout	path
pause	printlevel	problem	read	recall
report	save	sched	show	solve
split	system	time	title	unsave
values	write	xdata	xform	xtran
xvar				

Note that the modules of the Reference Manual (Part 3) are listed at the beginning of it.

Index

Valid JLP commands are underlined.