Juha Lappi
Reetta Lempinen

J -users' guide 2.0

Version 2.0 2013

Finnish Forest Research Institute
Suonenjoki Unit

Lappi, Juha and Lempinen, Reetta. 2013**. J-users' guide 2.0**. Finnish Forest Research Institute, Suonenjoki Unit 118 pp.

The **J** users' guide is published only electronically and is made available for public access in the Internet

Keywords: forestry analysis, linear programming, data analysis, matrix computations, simulation, optimization

Authors' addresses: Juha Lappi, Finnish Forest Research Institute, Suonenjoki Unit, Juntintie 154, FI-77600 SUONENJOKI, Finland
Reetta Lempinen, Finnish Forest Research Institute, Joensuu Unit, Yliopistokatu 6, FI-80100 JOENSUU, Finland

Authors' email: juha.lappi@metla.fi, reetta.lempinen@metla.fi

Publisher: The Finnish Forest Research Institute Metla, Jokiniemenkuja 1, BOX 18 FI-01301 VANTAA, Finland
Metla project number: 3002

(URL: http://mela2.metla.fi/mela/j/manuals/J2.0_userguide.pdf)

Conditions of use of the software

There are two license types for the **J** software. The use of **J** software is free for research and teaching purposes (academic license). To use J for commercial or production purposes or for practical forest planning requires commercial license with annual license fee.

The software is available from 22.4.2013 at the web page http://mela2.metla.fi/mela/j/index.htm which contains the conditions of use, user registration, and download page for registered users.

Changes from previous versions

Changes from version 1.0.3 to version 2.0.

J software is able to solve factory problems. In a factory problem, the transportations costs of different timber assortments at specified time periods and capacities of factories at the same time periods are included in the problem definition. For instance, the net present value can be maximized subject to capacity constraints and sustainability constraints. See `jlp` function for more details.

Multiple input files can be defined in a `data` function call.

Migration to a new IDE and operating system. J is now compiled with Intel(R) Visual Fortran Composer XE 2013 under Windows 7.

Changes in older versions can be found in Version 1.0.3 manual, which is available at the web page http://mela2.metla.fi/mela/j/manuals/J1.0.3_userguide.pdf.

Table of contents

Introduction

J is a general program for doing different tasks in data analysis, matrix computations, simulation and optimization. It is intended to be used mainly in different forestry related applications. It will supersede previous linear programming software JLP (Lappi 1992).

Most users are interested in applying J in linear programming. Linear programming functions and examples are described in chapter 11. J version 2.0 is introducing factory problems where transportation costs and factory capacities can be taken into account. Factory problems are also described in chapter 11. Shortest route to linear programming problems is to read basics of command generation programming from chapter 3.2, at least ;`incl`-function from chapter 3.2.3. Forestry LP-problem requires also use of `data`-function (chapter 9.1) and usually also `linkdata`-function (chapter 9.3). LP-problems are defined with `problem`-function (chapter 11.5) and the problems are solved with `jlp`-function (chapter 11.7). To access the weights of optimal treatment schedules and to get them into files requires use of J-transformations (chapter 4.) inquiry functions (chapter 11.8), IO-functions (chapter 8.), loops (chapter 7.2). An example is given in chapter 11.

J was used to do all computations in the stem curve paper of Lappi (2006). Not all J functions needed in this method are reported in this guide. Those interested to apply the method should contact J. Lappi.

J is operated using text command lines, but it contains tools which make this kind of operation mode more efficient, e.g. input can be included from files so that a part of the input lines is reinterpreted, input lines can be generated using loop constructs etc. These properties are called here as input programming.

1.1 *System requirements*

The current version of **J** is developed under Windows 7. It is compiled as 32-bit application and it is running also at least under Windows XP.

J is written in Fortran 90 and compiled with Intel(R) Visual Fortran Composer XE 2013. There are both release and debug versions available.

The release version is an ordinary console application, See section 1.3.how to modify the I/O window on the first run.

If execution of the release version of **J** terminates unexpectedly, the console window disappears without outputting information about possible fault. **J** debug version can be used for troubleshooting the crash problems. The debug version is a QuickWin application, which in case of crash displays message box with run time error message. Compared to the release version, the debug version has bigger size and the execution takes more time.

J is allocating dynamically memory for data structures. Both the stack and the heap are used for the automatic and temporary array allocations. If data sets are large, **J** puts automatic and temporary arrays on the heap, otherwise they are put on the stack.

1.2 *Set up of J*

The maximum number of available objects cannot be changed during a **J** session. It is determined during the initialization. When **J** is started it tries to read first file `j.par` from the default directory (see 'During the first use' chapter): The first line must look like

```
*2000
```

where the number gives the maximum number of named objects. The default is 5000.

Thereafter there can be any number of **J** commands executed directly (e.g. you can give symbolic names for colour indexes and line types used in graphics).

1.3 *During the first use*

It is reasonable to have the exe version in one folder, and to make shortcuts into all working folders. Edit the properties of the shortcut (right click the shortcut icon) so that the starting directory is the working directory. Copy also the file `j.par` into each working directory.

When the program is started, there appears a prompt, possibly after initialization commands read from `j.par`.

```
sit>
```

Edit first the properties of the I/O window (release version). The properties of the I/O window can be changed by right-clicking the dos-icon at the upper left corner. It is reasonable to make the screen buffer rather large (large height) so that the whole history of the **J** session can be seen (this is done in the layout sheet of the shortcut properties). The default height of the I/O window is also probably too small. The width should be at least 81. If you would like to use mouse in copy and paste, put quick edit option on. Also the colours of the text and background of the **J** window should be made healthier for eyes (dark text, bright background).

In the debug version console window properties cannot be modified.

To see that **J** is running properly, give your first commands at `sit>` prompt:

```
sit>a=7.7
sit>print(a)
```
The result should look like
```
a=  7.700000
sit>
```

Edit then the first command by using the arrow keys into `a=8.8` and submit the command, as well as the print command. Study also the copy and paste possibilities under the Dos icon. In the debug version you can find copy and paste commands in the Edit menu.

All input lines entered or generated by input programming at `sit>` prompt are called commands. Commands are either input programming commands (*input commands*) or commands that define operations in the **J** working environment (*operation commands*). Input commands and operation commands may read and interpret more input lines before returning control to the command level.

It is most convenient to develop **J** applications using include files. There is now available an include file `jex.txt` on the download web page which gives several examples and exercises.

The working environment of **J** consists of named objects, temporary objects, constants, functions, arithmetic operations and text paragraphs. Operation commands define simple arithmetic operations or more complicated operations on the data structures. Operation commands are defined using a transformation language. In addition to operation commands, the same transformation language is used to define transformation sets which are computed as a group and which can be linked in different ways to data structures or other transformation sets.

1.4 *Exiting J*

To exit **J** program and close console window, just give `end` command:

```
sit>end
```

This command also closes automatically **J** console window.

1.5 *Manual conventions*

In the description of **J** functions, optional arguments or options are indicated by [ ]. Some elements may be necessary if some other optional elements are present. These are described by detail. If there is no output for a operation command line, the object `Result` is used as the default output. In many cases there is no output object, and a possible output given is ignored. If an output is necessary, then '=' is put in front of the function name. Specific implementation details of functions are given after '#'.

Expressions in the **J** language will be written in the `Courier` font.

This manual contains very few examples. More examples are given in the accompanying include file jex.txt.

Possible future changes are indicated by '***'. Users needing these improvements should tell their hopes.

2 J objects

There are several object types in the **J** environment. All objects except real and character constants have names which can be used to refer to the object. Some of the objects are elementary objects. Other objects are compound objects consisting of elementary objects and possibly also of an object specific part. The named elements of a compound object can be accessed also directly. Some element objects are automatically created by the function which creates the compound object. Some elements may be named independently, and the compound object just contains links to the element objects.

There are both named and unnamed objects in a **J** workspace. Named objects are created by giving output objects to **J** functions, and also by giving object names as arguments for such **J** functions and options which accept unknown arguments. Real numeric constants encountered in **J** commands are put into a vector which is used in the same way as named real variables. Intermediate results of **J** commands and transformations are stored into unnamed objects.

2.1 *Object names*

Object names start with letter or with '`$`'. Object names can contain any of symbols '`#.%$_`'. **J** is using '`%`' to name objects related to some other objects. E.g. function `stat(x1,x2,mean->)` will store means of variables `x1` and `x2` into variables `mean%x1` and `mean%x2`. Objects with name starting with '`$`' are not stored in the automatically created lists of input and output variables when defining transformation sets.

Names of objects having a predefined interpretation start with capital letter. The user can freely use lower or upper case letters. **J** is case sensitive.

All objects known at a given point of a **J** session can be listed by command:

```
print(Names)
```

2.2 *Copying object: a=b*

A copy of object can be made by the assignment statement `a=b`.

2.3 *Deleting objects: delete()*

When an object with a given name is created, the name cannot be removed. With `delete` function one can free all memory allocated for data structures needed by general objects:

`delete(obj1,…,objn)`

After deleting an object, the name refers to a real variable (which is initialized by the `delete` function into zero).

Note 1: Objects can equivalently be deleted by giving command

```
obj1,…,objn=0
```

Note 2: One can see how much memory each object is using with `print(Names)`.

*** Currently, deleting a compound object will not delete the (named) element objects. Thus e.g. deleting a data set will not delete the data matrix. It is now also possible to delete a named element of a compound object and thus corrupt the compound object (this will or will not properly realized by the function which is using the compound object).

2.4 *Object types*

The following description describes shortly different object types available in **J**. More detailed descriptions are given in connection of **J** functions which create the objects.

Object types may change during a **J** session. If the final object type is not yet known during the interpretation time, the object is first created as a real variable.

Real variables and constants

A real variable is a named object associated with a single real value. The value can be directly defined at the command level, or the variable can get the value from data structures. E.g.

```
a=sin(2.4)
```

```
h=data(read->(x1…x4))
```
! x1,x2,x3,x4 are variables in the data set, and get their values when doing operations for the data.

All numeric constants appearing in transformations will be stored as real constants.

Intermediate results in arithmetic calculations are stored into unnamed real variables.

Note 1: All objects have also an associated real value. In order to make arithmetic operations fast, the argument types in simple arithmetic functions are not checked. If a general object is used as an argument in an arithmetic operation, then the real value associated with the object is used. This will usually prevent the program to stop due to Fortran errors, but will produce unintended results.

Note 2: In this manual 'variable' refers to a **J** object whose type is real variable.

Character constants and variables

Character constants are generated by closing text within apostrophe signs (').
Character constants are used in I/O functions for file names, formats and to define text to be written. E.g.

```
a=data(in->'file1.dat',read->(x1,x2))
```

Apostrophe character (') within a character constant is indicated with (~) (if the character ~ is not present in the keyboard, it can be produced by <Alt>126, where numbers are entered from the numeric keyboard), e.g.

```
write('output.txt', '(~kukkuu=~,4f7.0)', sqrt(a))
```

Character variables get character constants as their values. An example of a character variable definition:

```
cvar='file1.dat'
```

After defining a character variable, it can be used exactly as the character constants.

Note: The quotation mark (") has special meaning in the input programming. See *Input programming* how to use character variables within character constants.

Text object

Text object is an object which can store several lines of text. Several **J** functions create associated text objects. **J** function `text` can be used to create text objects directly. All the names of **J** objects are also stored in a text object called `Names`. The number of lines in a text object can be obtained with `nrows` function and the total number of characters can be obtained with `len` function.

Logical values

There is no special object type for logical variables. Results of logical operations are stored into temporary or named real variables so that 0 means *False* and 1 means *True*. In logical tests all nonzero values will mean *True*. Thus e.g. `if(6)b=7` is legal statement, and variable `b` will get value 7. E.g.

```
sit>h=a.lt.b.and.b.le.8
sit>print(h)
h=   1.000000
```

Object lists

An object list is a list of named **J** object. See *Shortcuts for implicit object lists* and *List functions* for more details. Object lists can be used also as pointers to objects, see e.g. the selector option of the simulate function.

Matrices and vectors

Matrices and vectors are generated with the `matrix` function or they are produced by matrix operations, matrix functions or by other **J** functions. E.g. the `data` function is producing a data matrix as a part of the compound data object. Matrix elements can be used in arithmetic operations as input or output in similar way as real variables.

See *Matrix computations*.

Transformation set

A transformation set groups several operation commands together so that they can be used for different purposes by **J** functions and **J** objects. A transformation set contains the interpreted transformations. For more details see ***J** function for defining transformation sets: trans()*.

Transformation sets can be called using `call` function, so that all transformations defined in the set are done once. Function result also calls transformations but is also returning a value. When transformation sets are linked to data objects, then the transformations defined in transformation set are done separately for each observation.

There is an implicit transformation set `$Cursor$` which is used to run the command level. Another transformation set `$Val$` which is used to take care of the substitutions of "-sequences in the input programming. Some **J** functions use also implicitly transformations set `$Cursor2$`

Simulator

A simulator is a transformation set with a few additional parameters. The simulator is described in the *Simulator* chapter.

Data set

Data set is a compound **J** object linking together data matrix, variable names, transformations and links to other data sets in a data hierarchy. Data set object is described in chapter *Data sets*.

Problem definition object

Problem definition object is produced by the `problem` function, and it is described in *Linear programming*.

Figure object

Graphic functions produce figure objects. Each figure object can consist of several subfigures. Each figure object stores information of x- and y axes, the range of all x- and y-values, and for each subfigure information of the ranges of x and y in the subfigure plus the subfigure type and the needed data values. See *Plotting figures* for more information.

Function objects

Different **J** functions can produce function objects which need several associated parameters and which can be used through `value` function.

Storage for variables

Especially in a simulator it may happen that a set of variables have certain values but the same variables are used for other purposes for some time and then one would like to get the previous values. There is special **J** object used to store the values, and special `store` and `load` function to deal with the storage.

Bitmatrix

A bitmatrix is an object which can store in small memory space large matrices used to indicate logical values. A bitmatrix object is produced by `bitmatrix` function or by closures function from an existing bitmatrix. Bitmatrix values can be read from the input stream or file or set by `setvalue` function. The values of bitmatrix elements can be accessed with `value` function.

Note: Also ordinary real variable can be used to store bits. Se bit functions

Trace set

Trace set is an object created by `;trace` function which is used by `tracetest` function to test if a set of variables has been updated. See chapter 'Tracing variables'

2.5 *Objects created automatically and default names*

The following objects are created automatically at start-up:

Names	text object containing the names of all objects
Pi	real variable having value 3.14… .
Result	real variable used to store the result if output variable is not given
Obs	the default real variable used to indicate the number of observation within a data set
$Cursor$	transformation object used to run the command level
$Cursor2$	another implicit transformation set
Val	transformation object used to extract values of mathematical statements, used, e.g., in input programming
$Data$	default data set name for a new data set created by data-function
Data	a list object used to indicate current data sets
LastData	a list objectreferring to the last data set made, used as default data set
$	object name used to indicate console and '*' format in reading and writing commands .

Duplicate	a special variable used in data function when duplicating observations
$Buffer	a special character object used by the write function

The following names are used as default names for objects created by **J** functions:

Figure	the default figure object created with graphics functions
T	the default variable for the period number in a simulator
Unit	the variable for the observation number in the unit data made by the `simulate` function

The following objects are created by some functions whenever these functions are called.

Object	created by	explanation
Tracevars	;trace	cumulative list of all objects used in all ;trace functions
Tracestatus	;trace	row vector corresponding to Tracevars list indicating if tracing code is generated
Tracelevel	;trace	vector indicating the tracing level for variables having the tracing code.
Tracecount	;trace	counts of changes
Traceminstatus	;trace	indicates if minimum checking is effective
Tracemin	;trace	minimum values
Tracemaxstatus	;trace	indicates if maximum checking is effective
Tracemax	;trace	maximum values

3 Command input and output

3.1 *Input line and input paragraph*

J reads input records from the current input channel which may be terminal, file or a text object. When **J** interprets input lines, spaces between limiters and function or object names are not significant. In input programming function start with ';' which is part of the function name (and there can thus be no space immediately after ';'). If a line (record) ends with ',' ,'+','*,'-','('='=' or with '>', then the next record is interpreted as a continuation record. All continuation records together form one *input line*. If the continuation is indicated by other continuation characters than the '>', the continuation character is kept as a part of the input line. If continuation is indicated by '>' then '>'

will not remain in the logical input line. One input record can contain 256 characters, and an input line can contain 2048 characters (this can be increased if needed)

Note . '/' cannot be used as last character indicating the continuation of the line because it can be legal last character indicating the end of an input paragraph.

When entering input lines from the keyboard, previous lines given from the keyboard can be accessed and edited using the arrow keys.

To copy text from the **J** window into the clipboard right-click the title bar, select Edit, and in the context menu click Mark. Next click and drag the cursor to select the text you want to copy and finally press Enter (or right-click the title bar, select Edit, and in the context menu click Copy).

To paste text from the clipboard into the **J** command line right-click the title bar, select Edit, and in the context menu click Paste. Console applications of Intel Fortran do not provide copy and paste using <cntrl>c and <cntrl>v.

All input lines starting with '*' will be comments, and in each line text starting with '!' will also be interpreted as comment (`!debug` will put a debugging mode on for interpretation of the line, but this debug information can be understood only by the author). If a comment line starts with '*!', it will be printed.

Many **J** functions interpreted and executed at the command level need or can use a group of text lines as input. In these cases the additional input lines are immediately after the function. This group of lines is called *input paragraph.* The input paragraph ends with '/', except the input paragraph of text function end with '//' as a text object can contain ordinary input paragraphs. It may be default for the function that there is input paragraph following. When it is not a default, then the existence of the input paragraph is indicated with option `in->` without any value. An input paragraph can contain input programming commands; the resulting text lines are transmitted to the **J** function which interprets the input paragraph.

Examples of input paragraphs:

```
tr=trans()
a=log(b)
write($,'(~sinlog is=~,f4.0)',sin(a))
/
b=matrix(2,3,in->)
1,2,3
5,6,7
/
```

3.2 *Input programming*

The purpose of the input programming is to read or generate **J** commands or input lines needed by **J** functions. The names of input programming commands start with semicolon ';'. There can be no space between ';' and the following input programming function. The syntax of input programming commands is the same as in **J** functions,

but the input programming functions cannot have an output. There are also controls structures in the input programming. An input paragraph can contain input programming structures.

3.2.1 Addresses in input programming

The included text files can contain addresses. Addresses define possible starting points for the inclusion or jump addresses within an include file. An address starts with semicolon (`;`) and ends with colon (`:`). There cannot be other text on the address line. E.g.

```
;ad1:
```

see: `;incl ;goto`

Note: The definition of a transformations set can also contain addresses. These addresses start with a letter and end also with colon (`:`).

3.2.2 Changing "…" sequences

If an original input line contains text within quotation marks, then the sequence will be replaced as follows. If a character variable is enclosed, then the value of the character variable is substituted: E.g.

```
directory='D:\j\'
name='area1'
extension='svs'
```

then

```
in->'"directory""name"."extension"'
```

is equivalent to

```
in->'D:\j\area1.svs'
```

If the expression is not a character variable then **J** interprets the sequence as an arithmetic expression and computes its value. Then the value is converted to character string and substituted into the place. E.g. if `nper` is variable having value 10, then lines

```
x#"nper+1"#"nper"=56
chv='code"nper"'
```

are translated into

```
x#11#10=56
chv='code10'
```

With `"…"` substitution one can define general macros which will get specific interpretation by giving values for character and numeric parameters, and numeric

parameters can be utilized in variable names or other character strings. In transformation sets one can shorten computation time by calculating values of expressions in the interpretation time instead of doing computations repeatedly. E.g. if there is in a data set transformation

```
x3="sin(Pi/4)"*x5
```

Then evaluation of `sin(Pi/4)` is done immediately, and the value is transmitted to the transformation set as a real constant.

3.2.3 Input programming commands and control structures

The input programming has its own commands and control structures which will deliver text lines to the command level.

`;incl([file][,from->])`

Includes lines from a file or from a text object.

Argument: file name (character constant or character variable) or a text object, if omitted, then the same file is used as in the previous `;incl()`.

Option: `from` gives the starting address for the inclusion, address is given without starting '*;*' and ending '**:**'.

Examples:

```
;incl('file.txt')
;incl('file2.txt',from->'ad1')
;incl(from->'ad2')
```

Note 1: Include files can be nested up to 4 levels.

Note 2: See Chapter '*Defining a text object with text function and using it in ;incl*' how to include commands from a text object.

`;return`

Closes the current include file, and changes the input channel to the upper include file or to the console

Note 1: The include file can be open simultaneously in a text editor during the **J** session if you open the file first with the text editor. If you want to include sections from a changed file, remember to save the changes before include. It is a handy to have after each `;return` the text which can be used to include the previous section. E.g. in file `jlp.inc`:

```
;test:
…
;return
;incl('jlp.inc',from->'test')
```

Then after editing the test section, copy the `;incl`-line into clipboard and drop it into **J** session by clicking the right button of the mouse.

Note 2: Transformation set can also contain `return` statement (without '`;`') which stops execution of transformations in the transformation set.

`;do(i,start,last[,step])`

purpose: to generate a sequence of input lines in a loop

Arguments: index variable, initial value, final value, step (optional, default is 1)

`;enddo`

purpose: to close a `;do()` loop

Note 1: There can be 6 nested `;do` loops. It is not recommended to use `;do` loops in the console input even if this is possible.

Note 2: Form `;end do` is also accepted.

Examples (in a include file):

```
;do(i,1,5,1)
per"i"=i*10
;enddo
g=trans()
;do(i,1,5,1)
per"i"="i"*10
;enddo
/
call(g)
```

It is necessary to use `"i"*10` in the input paragraph of `trans`-function. If the transformation line is `per"i"=i*10`, then during calling `g`-transformations, the value of variable `i` is the same (`5`) for each transformation line generated by the `;do` loop.

`;if(…)`

Generate input based on some condition. The text after the condition may be input command or operation command or text within an input paragraph.

Argument: logical or arithmetic statement producing zero (False) or nonzero value (True)

Examples:

```
;if(Feasible);incl('report.inc',from->'summary')
;if(debuglevel.gt.2)print('Note: debug information in file
debug.txt')
```

`;if(value);then`

…

`;elseif(value);then`

…

`;endif`

Picks several lines into input based on some condition.

The argument for `;if` or `;elseif` is a logical or arithmetic statement (or variable) producing zero (*False*) or nonzero value (*True*)

`;goto('adr')`

Start reading input from another place in an include file or include text object. Jumping is allowed only forward.

Argument: character constant or character variable, the address without starting '`;`' and ending '`:`'

Example, in an include file:

```
;goto('ask')
…
;ask:
*what to do next:  task1, task2 or end
askc(ad)
;goto(ad)
;task1:
…
;return
;task2:
…
;return
;end:
;return
```

Note: Software specialists do not recommend using `goto` structures, but e.g. the structure of the example above may be useful. If you want to use several sections from a file, you can define a driver include file which contains just `;incl( ,from->)` – lines

3.2.4 Utilizing object lists: @list and @list(elem)

There is special object list object in **J.** An object list is generated with `list` function, e.g.

```
xvar=list(vol#1,ba#1,dbh#1)
```

Thereafter @`xvar` in any place of the input line is equivalent to `'vol#1,ba#1,dbh#1'`.

The names of individual variables in a list can be accessed using @`xvar(elem)` where `elem` is a numeric expression obtaining a value between 1 and `len(xvar)`.

There is subtle difference between expanding whole list using . @`xvar` and accessing the names of individual variables in a list using @`xvar(elem)`. When **J** expands the whole list, it first interprets the whole transformation line as if @`xvar` would be a single argument, and then finally it just replaces the index of the argument by all indices of the elements in the list. When **J** encounters @`xvar(elem)` then the value of `elem` is first computed and then the name the corresponding variable in the list is dropped into the same place before interpreting the line (i.e. **J** proceeds as in interpreting "-sequences). Thus we may have:

```
alist=list(a,b)
@alist(1)#@alist(2)=@alist(1)*@alist(2)
```

which is equivalent to:

```
a#b=a*b
```

Some function can have lists as their arguments and some options can have lists as their values. In those cases the name of the list object must be used without '@'. See chapter *List functions* for more details about lists

Note: Lists can be also used to define pointers to single variables. E.g. a general method defined in an include file (a macro) can refer to a variable using e.g. @arg. Then we can give specific interpretation to the variable giving at the command level `arg=list(temperature)` , and if we then include the macro from an include file then all reference using `@arg` refer to the variable `temperature`. Of course we can make the `arg` list to point to several variables by defining it to be a list of several objects.

3.2.5 Shortcuts for implicit object lists: x1…x5, ?%x1

Many functions can have several arguments, and also an option can refer to many objects. There are some shortcut notations for referring to several objects.

If several objects have increasing numeric or character end in their names, then implied object lists can be formed using the '. . .' construct. E.g.

```
stat(x4...x10)
stat(vara...vard)
```

are equivalent to

```
stat(x4,x5,x6,x7,x8,x9)
stat(vara,varb,varc,vard)
```

All variables having a common part in their name can be referred using '`?`' to indicate the unspecified part. E.g. command

```
print(mean%?)
```

will print all variables whose name start with '`mean%`', and command

```
print(?%x1)
```

will print all variables whose name end with '`%x1`' (e.g. `mean%x1, min%x1`, etc.)

3.2.6 Key shortcuts

It is possible to define key shortcuts using character variables. If the whole input line consists of the name of a single character variable, then the value of the character variable is taken as the input line. E.g. input lines

```
I=';inc(~j.inc~)'
I
```

give the same result as

```
;inc('j.inc')
```

Note: the character variable `I` can be utilized according to the rules of the input programming equivalently as

```
"I"
```

Key shortcut are handy when one is repeatedly including the same part from an include file when testing e.g. a simulator.

3.2.7 Defining a text object with text function and using it in ;incl

Text objects are created as a side product by many **J** functions. Text objects can be created directly by the text function which works in a nonstandard way. The syntax is:

```
=text()
…
//
```

Output: a text object

The input paragraph ends exceptionally with '`//`' and not with '`/`'. The lines within the input paragraph of text are put literally into the text object (i.e. even if there would be input programming functions or structures included).

If the text object is used as an argument of ;`incl()` then everything goes as if the lines would be included from a file. Using text objects this way makes it possible to define text macros in the same file as other commands.

3.3 *Immediate operations starting with ';'*

Currently there are two special commands (`;trace` and `;traceoff`) which looks like input programming commands, but which can be characterized as 'immediate operations' or 'interpreter directives'. There may be more such commands in the future: The `;trace` command tells that the transformation interpreted should start to generate special tracing information among all the commands or transformations given thereafter, and `;traceoff` tells to stop to generate such code. See Error debugging chapter for more information.

3.4 *Controlling output*

It is quite difficult to design the amount of printing logically in environment like J. There should be enough output to see that J is doing what it should do. But using input programming one can generate efficiently huge amount of commands which could easily cause very much printing.

First we must separate printing related to the input programming or input paragraphs and printing output of jlp functions. There are two global variables controlling either of these, `Printinput` and `Printoutput`. The default value for both of these variables is `2`. Value `0` indicates no printing, value `1` less than default and values>`2` indicate more printing. Value `10` indicates debugging type of printing. For each jlp function, there will be available `print->` option which will locally guide printing.

4 J transformations

Most operation commands affecting **J** objects can be entered directly at the command level or packed into transformation object. In both cases the syntax and working is the same. A command line can define arithmetic operations for real variables or matrices, or they can include functions which operate on other **J** objects. General **J** functions can have arithmetic statements in their arguments or in the option values. In some cases the arguments must be object names. In principle it is possible to combine several general **J** functions in the same operation command line, but there may not be any useful applications yet, and possibly some error conditions would be generated.

Definition: A numeric function is a **J** function which returns a single real value. These functions can be used within other transformations similarly as ordinary arithmetic functions. E.g. `weights()` is a numeric function returning the number of schedules having nonzero weight in a JLP-solution. Then `print(sqrt(weights())+Pi)` is a legal transformation.

4.1 *Structure of general J functions*

The general (non arithmetic) **J** functions are used either in statements

func(arg1,…,argn,opt1->value1,….,optm->valuem)

or

output=func(arg1,…,argn,opt1->value1,….,optm->valuem)

If there is no output for a function in a statement, then there can be three different cases:

i) The function does not produce any output (if an output would be given, then **J** would just ignore it)

ii) The function is producing output, and a default name is used for the output (e.g. `Result` for arithmetic and matrix operations, `Figure` in graphic functions).

iii) The function is a sub expression within a transformation consisting of several parts including other function or arithmetic operations. Then the output is put into a temporary unnamed object which is used by upper level functions as an argument (e.g. `a=inverse(b)*t(c)`)

If the value of an option is not a single object or numeric constant then it must be enclosed in parenthesis.

Note 1: It is useful to think that options define additional argument sets for a function. Actually an alternative for options would be to have long argument lists where the position of an argument determines its interpretation. Hereafter generic term 'argument' may refer also to the value of an option.

Note 2: When **J** is interpreting a function, it is checking that the option names and the syntax are valid, but it is not checking if an option is used by the function. Also when executing the function, the function is reacting to all options it recognises but it does not notice if there are extra options, and these are thus just ignored.

An argument for a **J** function can be either functional statements producing a **J** object as its value, or a name of **J** object. Some options can be without any argument (indicating that the option is *on*). Examples:

```
a=sin(cos(c)+b)    ! Usual arithmetic functions have numeric values as arguments,
!   here the value of the argument of cos is obtained by 'computing' the value of real variable c.
stat(D,H,min->,max->)    ! Here arguments must be variable names
plotyx(H,D,xrange->(int(min%D,5), ceiling(max%D,5)))  ! arguments of the function are variables, arguments of option xrange are numeric values
c=inverse(h+t(g))  ! The argument can be intermediate result from matrix computations.
```

If it is evident if a function or option should have object names or values as their arguments, it is not indicated with a special notation. If the difference is emphasized, then the values are indicated by `val1,…valn`, and objects by `obj1,…,objn`, or the names of real variables are indicated by `var1,…,varn`.

There are some special options which do not refer to object names or values. Some options define a small one-statement transformation to be used to compute something repeatedly. E.g.

`stat(D,H,filter->(sin(D).gt.cos(H+1))` ! only those observations are accepted which pass the filter

`draw(func->(sin($x)+1),x->$x,xrange->(0,10,1))` ! the `func` option transmits the function to be drawn not a single value.

4.2 *Common options*

There are some options which are used in many **J** functions. Such options are e.g.

`in->`

If a J function needs to read some data or text, then the source is given in `in->` option. If there is no value for option, then the source is the following input paragraph. If the value is a character constant or a character variable, then the source is the file with that name.

`data->`

If the function is using data sets, the data sets are given in `data->` option. All data sets will be treated logically as a single data set. If a **J** function needs to access data, and the `data->` option is not given then **J** is used default data which is determined as follows.

If the user has defined an object list `Data` consisting of one or more data sets, then these will be used as the default data set. E.g.

```
Data=list(dataa,datab)
```

When a data set is created, it will automatically become the only element in `LastData` list. If the `Data` list has not been defined and there is no `data->` option, then the `LastData` dataset will be used.

`trans->`

When a data set is created with data function, `trans->` option defines a transformation set which is permanently associated with the data set (unless the association is changed with `editdata` function). In all functions which are using data sets, `trans->` option defines a transformation set which is used in this function. An example:

```
tr=trans()
xy=x*y
/
stat(xy,trans->tr)
```

`err->`

The `err->` option indicates a transformation set which will be called if an error occurs within a J function. In this transformation set one can e.g. print information about values of variables etc. Currently this option is present only in `stempolar` function, but it will be included in many other functions.

4.3 *J function for defining transformation sets: trans( )*

Transformation sets are created with the `trans` function.

```
=trans([input->][,matrix->][,arg->][,result->]
                [,local->][,source->])
```

...

`/`

options:

`input` If present without any values, it indicates that if there are any arguments in transformations which are not yet known they will be created as real variables during interpretation time. If there are variables given as values for the option, then these variables will be created as real variables (and no error occurs when referring to these variables). But if there are other unknown arguments, an error occurs.

`matrix` The objects given in the option are interpreted to be matrices even if the objects do not yet exist or if they are not matrices at the interpretation time. These objects can be used in statements referring to elements of the matrices e.g. `a(1,i)=b(i)`. Arithmetic operations for the whole matrices (e.g. `a=b`) will also be properly interpreted). The matrices need not be otherwise defined in the interpretation time. The actual type and dimensions will be checked during execution time. If a matrix already exist in the interpretation time, it need not be indicated in the matrix option.

`arg` If the transformation set is defining a function to be used in the `value` function then `arg` option gives the name of the variable used as the argument. Default is variable `Arg`. The `arg` variable permanently associated with a transformation set can be temporally bypassed by giving `arg` option in `value` function.

`result` If the transformation set is defining a function to be used in value function or in result function then result option gives the name of the variable defining the output of the function. Default is variable `Result`. The result variable associated with the transformations set transformation can be temporally bypassed by giving result option in value or result functions.

`local` Gives object names which are intended to be used only locally in the transformation set. These objects will in fact be global objects but each object name will have prefix formed from the transformation set name and '`\`'. Eg

```
tr=trans(local->(a,b))
```

will make objects `tr\a` and `tr\b`.

`source` If value zero is given for the option, then a text object containing the source code is not generated. Source code is used to generate debugging information when errors occur.

Each line in the input paragraph is read and interpreted and packed into a transformation object, and associated `input%`*`tr`* and `output%`*`tr`* lists are created (*tr* indicating the output of the `trans` function). Objects having names starting with '`$`' are not put into the input or output lists. The source code is saved in a text object `source%tr` if option `source->0` is not given.

*** Now there can be only one argument variable. If there is need for more argument variables, we can allow more than one

4.4 *Executing a transformation set explicitly: call()*

Interpreted transformations in a transformation set can be automatically executed by other **J** functions or they can be executed explicitly using `call` function.

`call(tr)`

Argument: a transformation set

`call` function can be used at the command level or within transformation set. Defining transformation sets which are called within other transformation sets one can use some transformation as subroutines. But the transformation sets do not yet have any system for argument passing, thus all objects within transformations are global objects. Using input programming one can define transformations which get specific interpretation after giving values to character variables and object lists. But these transformations must be interpreted first with `trans` function before they can be used.

Note: A transformation sets can be used recursively, i.e. a transformation can be called from itself. The depth of recursion is not controlled by **J**, so going too deep in recursion will eventually lead to a system error.

Example:

```
tr=trans(input->level)   !level will be initialized as zero
write( $,'(~recursion level~,f)',level)
level=level+1
call(tr)
/
```

Try it from the command level (it may take a while to reach the bottom):

```
call(tr)
```

4.5 *Using a transformation set as a function: result()*

Interpreted transformations in a transformation set can be used as a function returning a single numeric value using `result` function.

`result(tr [,result->])`

Argument: a transformation set

Option: `result` , defines the variable whose value is the result of the function, default is the result variable associated with the transformation set (the default result variable is `Result`)

Note: A transformation sets can be used recursively, i.e. a transformation can be called from itself. The depth of recursion is not controlled by **J**, so going too deep in recursion will eventually lead to a system error.

Note 2: There is no argument passing for changing the values of variables used in the transformation set as input variables. There is argument passing system for function value which also returns a value from a transformation set.

Example:

```
sit>s=trans(input->)
trans>Result=a+b
trans>f=Result+1
trans>/
sit>a,b=1,3
sit>print(result(s),result(s,result->f))
 =  4.000000
 =  5.000000
```

4.6 *Using a transformation set as a function with an argument: value()*

Interpreted transformations in a transformation set can be used as a function returning a single numeric value using `value` function.

`value(tr, xvalue[,arg->][,result->])`

Arguments:

`tr` a transformation set

`xvalue` value put into the argument variable before calling the transformation set options

`arg` variable used as the argument variable, it bypasses the argument variable associated with the transformation set

`result` , defines the variable whose value is the result of the function, default is the result variable associated with the transformation set

See `value` function for more details

5 Arithmetic computations

An arithmetic expression is a statement producing single real value. Arithmetic statements working with real variables have any of the three forms

output_variables=arithmetic expressions

matrix element= arithmetic expression

arithmetic expression

An arithmetic expression (statement producing a single real value) without an output variable is converted into statement

`Result=` arithmetic expression.

The rules for handling the case where there can be several output variables or arithmetic expressions are as follows:

If there are several output variables and one arithmetic expression, then each output variable obtains the value of the expression. E.g.

```
y1...y4=sin(x1)
```

If there are equal number of output variables and expressions, then each expression defines a assignment.. E.g.

```
y1...y3=1,2,sqrt(20)
```

If there are several output variables and more than one values but the number of output variables and values no not match, an error occurs.

Note: a copy of a general object can also be made with an assignment statement. Only one object can be copied in one statement

5.1.1 Standard numeric expressions

An arithmetic expression consisting of ordinary arithmetic operations is formed in standard way. The operations are in the order of their precedence:

unary minus

`***` integer power

`**` real power

* multiplication

`/` division

`+` addition

- subtraction

The reason for having a different integer power is that it is faster to compute and a negative value can have an integer power but not a real power.

5.1.2 Logical and relational expressions

There are following (Fortran style) relational and logical operations (alternative notation for relational comparisons):

`.eq.`	==	equal to
`.ne.`	<>	not equal
`.gt.`	>	greater than
`.ge.`	>=	greater or equal
`.lt.`	<	less than
`.le.`	<=	less or equal
`.not.`		negation
`.and.`		conjunction
`.or.`		disjunction
`.eqv.`		equivalent.
`.neqv.`		not equivalent

The relational and logical expressions produce value 1 for *True* and value 0 for *False.*

Note: Testing equivalence can be done also using 'equal to' and 'not equal', as the same truth value is expressed with the same numeric value.

5.1.3 Arithmetic functions

The arithmetic functions return single real value.

`sqrt, exp, log, log10, abs`

`sqrt(x)`	square root, sqrt(0) is defined to be 0
`sqrt2(x)`	sign(x)*sqrt(abs(x))
`exp(x)`	*e* to power *x*
`log(x)`	natural logarithm
`log10(x)`	base 10 logarithm
`abs(x)`	absolute value

Real to integer conversion

`nint(x)` nearest integer value
`nint(x,modulo)` returns modulo*nint(x/modulo) ,e.g.
`nint(48,5)=50; nint(47,5)=45;`
`int(x)` integer value obtained by truncation
`int(x,modulo)` returns modulo*int(x/modulo), e.g. int(48,5)=45
`ceiling(x)` smallest integer greater than or equal to x.
`ceiling(x,modulo)` returns modulo*ceiling(x/modulo), e.g. `ceiling(47,5)=50`.
`floor(x)` greatest integer smaller than or equal to x.
`floor(x,modulo)` returns modulo*floor(x/modulo), e.g. `floor(47,5)=45`.

min, max

`min(x1,...,xn)` minimum
`max(x1,...,xn)` maximum

sign

`sign(val)` returns 1 if val≥0 otherwise returns -1.

dot(c1,…,cn,x1,…,xn)

dot product, c1*x1+…cn*xn, see also matrix function `dotproduct`

which(cond1,value1,…,condn,valuen[,valuedef])

Takes first value for which the condition is true. If no condition is true then the `valuedef` is given, and if there is no `valuedef` argument then the initial value of the output is unchanged (producing probably unintended result, if which is used within another expression).

Trigonometric functions, argument in radians

`sin(x)`
`cos(x)`
`tan(x)`
`cot(x)`

Trigonometric functions, arguments in degrees

`sind(x)`
`cosd(x)`
`tand(x)`
`cotd(x)`

Inverse trigonometric functions, result in radians

`acos(x)`
`asin(x)`

```
atan(x)
acotan(x)
```

Inverse trigonometric functions, result in degrees

```
acosd(x)
asind(x)
atand(x)
acotand(x)
```

Hyperbolic functions

```
sinh(x)
cosh(x)
tanh(x)
```

5.1.4 Probability distributions

pdf(x[,mean][,sd])

Returns the density function of normal distribution. Default values for mean is 0 and for sd 1 (if sd is given then the mean must be also given even if it is zero)

** Later there will be other distributions specified by option.

cdf(x[,mean][,sd])

Returns the cumulative distribution function for normal distribution. Default values for mean is 0 and for sd 1 (if sd is given then the mean must be also given even if it is zero)

** Later there will be other distributions specified by option.

5.1.5 Random numbers

ran()

Returns a uniform random number between 0 and 1.

rann()

Returns normally distributed random number with mean zero and variance 1

***currently a quick and dirty generator

***Later possibility to determine the seed for the sequence will be added.

5.1.6 Special numeric functions

`npv(interest,income1,…,incomen,time1,…,timen)`

Returns net present value for income sequence `income1,...,incomen`, occurring at times `time1,…,timen` when the interest percentage is `interest`.

5.1.7 List arithmetics

We can do arithmetic operations for several variables using lists List arithmetics work very much like matrix algebra, the difference is that arguments and results are in named real variables.

***The list arithmetics has replace previous functions multcl,multcll,multll,multlll.

List arithmetics is easier to understand using examples (see jex.txt):

```
 alist=list(a1,a2,a3)
 blist=list(b1,b2,b3)
 clist=list(c1,c2,c3)
 @alist=1,2,3
 @blist=4,5,6
 clist=alist+blist !list=list+list
 print(@clist)
 c1=   5.000000
 c2=   7.000000
 c3=   9.000000
 clist=alist+5  !list plus real_value
 write($,$,@clist)
   6.000000       7.000000       8.000000
 clist=-alist   !negative
 write($,$,@clist)
  -1.000000      -2.000000      -3.000000
 clist=blist-alist !subtract
 write($,$,@clist)
   3.000000       3.000000       3.000000
 clist=2*alist  !list =real_value * list
 write($,$,@clist)
   2.000000       4.000000       6.000000
 clist=alist*blist  ! element by element multiplication
 write($,$,@clist)
   4.000000       10.00000       18.00000
 cval=alist*blist  ! if output is real variable then dot product is
computed
 print(cval)
 cval=   32.00000
```

**There cannot yet be several list arithmetic operations in the same line. It would be possible to extend the list arithmetic also that way that elements of lists could be matrices.

5.2 *Derivatives*

`d1,…,dn=der(x1,…,xn)`

The `der` function computes derivatives of a function with respect to one or several arguments using analytical derivation rules. The function is given in the next line.

Example

```
da,db=der(a,b)
f=a*exp(-b*x)
```

There is available a macro file which is using the `der` function and ordinary linear regression to compute nonlinear least squares regression.

6 Matrix computations

If the matrix dimensions agree, then matrix addition, subtraction and multiplication can be defined using standard arithmetic operations. If in addition either argument is scalar, then the scalar is added into each element. If in multiplication either argument is a scalar, then each element is multiplied.

The following matrix functions are currently available:

6.1 *Defining a matrix: matrix( )*

`matrix(nrows[,ncols][,in->][,diagonal->] [,form->]`
`[,values->])`
(values if in-> option is present)
/

Generates a matrix.

Arguments:

`nrows` number of rows

`ncols` number of columns, default is 1 (that is vectors are assumed to be column vectors).

Options:

`in` If `in` option is present then the values are given in a input paragraph

`diagonal` This option indicates that the matrix is diagonal. For diagonal matrix the values given in values option or in the next input paragraph refer only to the diagonal vector.

`form` The option indicates that each row is read separately. At the end of the row there can be comments and if error occurs during reading, the input line causing the error is printed. If form option is not given all the matrix values are read in one read statement.

`values` The values are given within the option, transformations can be used to define the values. If only one value is given then this value is given for all elements, otherwise so many elements are filled in row order as there are values.

If values are not given through `in` or `values` options then the elements will be zero.

Examples:

```
b=matrix(2,4,in->,form->)

1,2,3,4  ! there can be comments if there is form option
5,6,7,8
/
c=matrix(3,values->(sin(1),sin(2),cos(Pi))  !vector
```

Example of a **J** session defining matrix and its elements separately

```
a=matrix(2,2)
ta=trans()
do(i,1,2)
a(i,i)=i
enddo
/
call(ta)
```

Matrices are used as arguments for some **J** functions. Arithmetic operations `+`,`-` and `*` work also for matrices. A copy of another matrix is obtained by assignment (e.g. `a=b`). Matrix elements can be used both as input and output in transformations. Using `matrix` option in `trans` function, matrices can be used in definition of transformation set before actual matrices are created.

6.1.1 Matrix functions

`setmatrix(matrix,value [,diagonal->]`)

Puts all elements equal to the given value, or all the diagonal elements if `diagonal->` option is present.

##Note, if M is matrix then substitution

```
M=1
```

makes M into a real variable.

`t(a`)

Computes the transpose of a matrix

argument: a matrix object

Note: transpose function can be used within a compound transformation, e.g.

`h=b*t(a)`

`inverse(a)`

Compute the inverse of a matrix

Argument: a square matrix or a scalar, for a scalar argument inverse return the reciprocal of the value

`dotproduct(a,b[,limit1][,limit2])`

Computes the dot product of two vectors

Arguments: `a` and `b` are matrix objects, which are considered as vectors made by putting rows after each other.

`dotproduct(a,b,n)` computes the dot product using `n` first element.
`dotproduct(a,b,first,last)` computes the dot product using elements from `first` to `last`

Note 1 : `dotproduct` using only part of elements is useful e.g. in a simulator where simulations are done at tree level, and tree vectors reserve space for all potential trees.

Note 2: If `a` and `b` are column vectors, then `dotproduct(a,b)` is equivalent to `t(a)*b`.

`elementsum(a[,limit1][,limit2][,row->][,column->])`

Computes the sum of elements of vector

Argument: `a` is matrix object, if column or row option is not given and the matrix is a general matrix (i.e. both dimensions>1) the matrix is considered as a vector made by putting rows after each other.

`elementsum(a,n)` computes the sum using `n` first element.
`elementsum(a,first,last)` computes the sum using elements from `first` to `last`

options:
`row` gives the row whose elements are added (`limit1` and `limit2` can be used to specify a part of the row vector)

`column` gives the columns whose elements are added (`limit1` and `limit2` can be used to specify a part of the row vector)

`submatrix(a[,row->][,column->])`

Takes a submatrix from a matrix

Argument: a matrix

Options:

`row` if only one value is given then this row is taken, two values indicate a range of rows, the second must be negative of the upper bound, e.g. `row->(3,-5)`.

`column` if only one value is given then this column is taken, two values indicate a range of columns, the second value must be negative of the upper bound, e.g. `column->(3,-5)`

`diagonal` indicates that the a column vector is made by picking the diagonal elements from the whole matrix or from the row range indicated by the `row` option.

**The syntax of row and column options is prepared to the case where one can pick individual rows and columns. Currently only one row (column) or range of consecutive rows (columns) is supported.

`nrows(a)`

Returns the number of rows in a matrix

Argument: a matrix object

Note: `nrows` works also for text objects,

`ncols(a)`

Returns the number of columns in the matrix

Argument: a matrix object

`len(a[,any->])`

purpose: return the number of elements in the matrix (=nrows*ncols)

Argument: a matrix object

Option

`any len` returns value-1 if argument is not legal object for `len` (without `any->` an error occurs)

Note: `len` works also for text objects, returning the number of characters in a text object, and for a list it returns the number of elements in the list, and for regression object number of parameters.

`index(val, a[,any->])`

purpose: to locate the position of a number in a matrix (usually vector). Note, the matrices are stored in row order.

Arguments:

`val` a real value

`a` matrix object

Option

`any` indicates that it will be searched between which two elements in the matrix `val` is, it is assumed that the matrix is in increasing order. Let i denote the output of the function. Then i is the index of element such that `val` ≥ ith element in the matrix. If `val` is smaller than the first element, then the output will be 0.

Note 1: without any option, an error occurs if `val` is not found in the matrix.

Note 2. when the first argument is list, then index function returns the position of an object in an object list, see list functions

`sort(a,key->(key1[,key2]))`

Makes a new matrix obtained by sorting all matrix columns according to one or two columns.

Argument: a matrix object

Absolute value of `key1` and the value of `key2` must be legal column numbers. If `key1` is positive then the columns are sorted in ascending order, if `key1` is negative then the columns are sorted in descending order. If two keys are given, then first key dominates. It is currently assumed that if there are two keys then the values in first key column have integer values.

Note 1: If `key2` is not given and `key1` is positive, then the syntax is: `sort(a,key->key1)`

Note 2: If there is no output, then the argument matrix is sorted in place**.**

Note 3: The argument can be the data matrix of a data object. The data object will remain a valid data object.

***later there will be sort function for data object so that the key variables can be given using variable names. Currently index-function can be used to get the proper column number of the data matrix.

*** Other matrix functions, e.g., computation of eigenvectors will become in later versions.

7 Transformation control structures

Within **J** transformations, there can be similar controls structures as in the input programming. The difference is that these will remain as part of the transformation

set. Only the '`if()output=…`' structure is allowed at the command level, other are possible only within a transformations set.

7.1 *If*

`if()`

The one line if-statement can have one of the following forms:

`if()output=…`

or

`if( )func()`

A transformation is done depending on the truth value of the condition

Groups of transformations can depend on conditions using structure:

`if()then`

….

`elseif()then`

…

`else`

….

`endif`

There can be 4 nested `if()then` structures. If-then structures are not allowed at command level.

7.2 *Loops*

The loop construction in **J** looks as follows:

`do(i,start,end[,step])`

`enddo`

Within a do –loop there can be `cycle` and `exitdo` statements

`cycle`

The cycle statement transfer the control to the `enddo` statement (i.e. to the next iteration)

`exitdo`

The `exitdo` statement transfers the control to the next statement after `enddo`.

There can be 8 nested loops. `Do` loop is not allowed at command level.

7.3 *Return from a transformation set*

`return`

At `return` the execution of transformations in the current transformation set stops. The control returns to the point where the transformation was called, e.g., to command level, or to the function going through the data, or to an other transformation set.

Note 1: A return is automatically put to the end of a transformation set.

Note 2: Notice the difference between `return` and the input programming command `;return` which closes an include file and thereafter input lines are read from an upper include file or from the terminal.

`errexit(arg1,…,argn)`

Stops executing transformations in a transformation set or in a simulator, and returns control to the command level closing all open include files similarly as if J detects an error. The values of arguments are printed. Useful in connection of testing if arguments have legal values.

Example:

```
if(si.le.0)errexit('illegal value of variable si',si)
```

7.4 *Using addresses in transformation sets*

7.4.1 Address in transformation set

A transformation line within a transformation set can have an address. An address is an alphanumeric expression ending with colon, e.g.

```
ad1: write($,'t',1,'kukuu')
```

Addresses can be utilized in `goto` and `jump` functions. Note the difference between addresses of input programming and addresses within transformation sets: addresses of input programming start with the semicolon (`;`).

`goto('address')`

purpose: to continue execution of transformations from the given address

`jump('address')`

purpose: to compute transformations within an internal subroutine starting with the address and ending with **`back`**.

`back`

Returns control to the next transformation line following `jump`.

Note: It is not recommended to use `goto` according to modern computation practices. It may be reasonable to use short internal subroutines using `jump`. Defining the subroutines as separate transformation sets and using `call` is an alternative which seems to be equally fast to compute.

Example:

```
s=trans()
i=0
goto('koe')
write($,'t',1,'here')
koe:write($,'t',1,'this',7,i)
write($,'t',1,'that')
i=i+1
if(i.lt.4)goto('koe')
jump('jump')
write($,'t',1,'after jump subroutine')
return
jump:write($,'t',1,'in subroutine')
back
/
```

8 IO-functions

`print(arg1,…,argn[,maxlines->][data->][row->])`

Print values of variables or information about objects.

Arguments: arguments can be any **J** objects or values of arithmetic or logical expressions

Options:

`maxlines` the maximum number of lines printed for matrices, default 100.

`data` data sets. If data option is given then arguments must be ordinary real variables obtained from data.

`row` if a text object is printed, then the first value given in the row option gives the first line to be printed. If a range of lines is printed, then the second argument must be the negative of the last line to be printed (e.q. `row->(10,-15))`. Note that `nrows` function can be used to get the number of rows.

For simple objects, all the object content is printed, for complicates objects only summary information is printed. `print(Names)` will list the names, types and sizes of all named **J** objects. The printing format is dependent on the object type.

*** The generated output does not look yet nice

`read(file,format[,obj1,…,objn])`

Reads real variables or matrices from a file. If there are no objects to be read, then a record is bypassed.

Arguments:

`file` the file name as a character variable or a character constant

format:

`'b'` unformatted (binary) data

`'bn'` unformatted, but for each record there is integer for the size of the record

`'bi'` binary format (without record structure), e.g. created with Matlab.

`'bis'` binary data consisting of bytes, each value is converted to real value (the only numeric data type in J)

`'(….)'` a Fortran format

`$` or `'*'` the * format of Fortran

Note: Use `ask` or `askc` to read values from the terminal when reading lines from an include file.

`write(file,format,val1,…,valn[,tab->])` ! case[1/5]

Writes real values to a file or to the console

Arguments:

`file`: variable `$` (indicating the console), or the name of the file as a character variable or a character constant, or variable $`Buffer`

format:

`$` indicates the '*' format of Fortran, works only for numeric values.

A character expression, with the following possibilities:

A format starting with '`b`' will indicate binary file. Now '`b`' indicates ordinary unformatted write, later there will be other binary formats

A Fortran format statement, e.g. `(~the values were ~,4f6.0),` with this format pure text can be written by having no object to write (e.g. `write('out.txt','(~kukuu~)')`).

For these formats, other arguments are supposed to be real variables or numeric expressions. If they are not, then just the real value which is anyhow associated with each **J** object is printed (usually it will be zero).

`tab` if format is a Fortran format then, `tab` option indicates that sequences of spaces are replaced by tab character so that written text can be easily converted to Ms Word tables.

`'t'` tabulation format, then the write -function is

`write(file,'t',t1,val1,t2,val2,…,tn,valn[,tab->])` ! case[2/5]

Positive tab position values indicate that the value is written starting from that position, negative tab positions indicate that the value is written up to that position. The values can be either numeric expressions or character variables or character constants. Tab positions can be in any order.

`tab` option indicates that sequences of spaces are replaced by tab character so that written text can be easily converted to Ms Word tables.

`'w'` width format, then the `write` function is

`write(file,'w',w1,val1,w2,val2,…,wn,valn[,tab->])` ! case[3/5]

Positive `w`-value indicates that the value is right-justified into field of that length, negative `w`-values indicate that the value is left-justified. The value can be either numeric or character expression.

In both '`t`' and '`w`' format with integer w-value, numeric values are converted into character expression with 8 characters. This special formatting drops unnecessary decimal points, leading and ending zeros, and will give as much precision as can be obtained using 8 characters. If less than 8 characters are needed, then one can use shorter fields than 8 characters.

A decimal `w`-value works similarly as f-format of Fortran, thus `w`-value 8.2 is equivalent to f8.2. For technical reasons, the format with zero decimals but with decimal point included must be given with `w`-value having decimal part .01, e.g. `w` value 5.01 is equivalent to f5.0. Note that writing with zero decimal using e.g. `5,nint(value)` will drop also the decimal point (corresponding to `I` format of Fortran).

When first write to a file is done, then the file will be opened. If the file already exists then **J** asks if the old file can be deleted. Note that before answering you can rename the file. In that case the old file will be saved even if you answer '`y`'

`tab` option indicates that sequences of spaces are replaced by tab character so that written text can be easily converted to Ms Word tables.

`write(file,text_object)` ! case[4/5]

A text object can be written into a file using this form of write function.

`Writing into $Buffer` ! case[5/5]

If variable `$Buffer` is used as the `file` argument, then different write –function calls can put information on the same line. Writing into `$Buffer` has the following logic. Other parts of **J** consider `$Buffer` as real variable. The output buffer can be initialized by giving value zero to `$Buffer` (i.e. giving command `$Buffer=0`), this is the situation initially. One can write onto the buffer using `$,'(...)', 't', or 'w'` -formats. `$` and `'(... )'` formats will also initialize the buffer first, so only `'t'`, and `'w'` formats can be used to collect buffer in several parts. After writing into the buffer, the real variable `$Buffer` gets the current length of the output. The current output buffer can be written into file using either

```
write(file,$Buffer)
```

or `$Buffer` can be used similarly as character variables in writing with '`w`' or '`t`' format, e.g.

```
write($,'t',1,$Buffer,$Buffer+2,'kukuu')
```

In the above first `$Buffer` indicates the current content of the buffer. In the tab value `$Buffer+2` indicates that the tab position is two characters past the buffer length.

Note: You can put character information into the format (to put apostrophe within character constant use (~),see *Character constants and variables*).

Examples:

```
dir='d:\j\'
write('"dir"example.out','(~the values were ~,4f4.0)',@values)
```

`close(file`)

Closes an open file

Argument: character variable or constant telling the name of an open file. The file has been created and opened with write or save functions.

`exist(filename)`

Tests if a file exist

Argument: character variable or constant telling the name of the file

Function returns value 1 (True) if the file exists and 0 (False) if the file does not exist

`ask(var1,…,varn[,default->][,q->][,exit->])`

Ask values for variables while reading commands from an include file.

Arguments: 0-n real variables (need not exist before)

Options:

`default` default values for the asked variables

`q` text used in asking

`exit` if the value given in this option is read, then the control returns to command level similarly as if an error would occur. If there is no value given in this option, then the exit takes place if the text given as answer is not a number.

If there are no arguments, then the value is asked for the output variable, otherwise for the arguments. The value is interpreted, so it can be defined using transformations.

Response with carriage return indicates that the variables get the default values. If there is no `default` option or the `default` option has fewer values than there are arguments, then the previous value of the variable is maintained (which is also printed as the `default` value in asking)

Examples: (two first are equivalent):

```
a=ask(default->8)
ask(a,default->8)
print(ask()+ask()) ! ask without argument is a numeric function
```

`askc(chvar1,…,chvarn[,default->][,q->][,exit->])`

Asks values for character variables when reading commands from an include file.

Arguments: 0-n character variables (need not exist before)

Options:

`default` default character stings

`q` text used in asking

`exit` if the character constant or variable given in this option is read, then the control return to command level similarly as if an error would occur.

If there are no arguments, then the value is asked for the output variable, otherwise for the arguments.

Response with carriage return indicates that the variable gets the default value. If there is no `default` option or the `default` option has fewer values than there are arguments, then the variable will be unchanged (i.e. it may remain also as another object type than character variable).

9 Data sets

9.1 *Creating a data object: data( )*

Data sets are created with the `data` function. Two linked data sets can be created with the same function call (using option `subdata` and options thereafter in the following description). A data set can be modified with `editdata` function. Data sets can be linked also afterwards with the `linkdata` function.

A data set is created by a `data` function

```
d=data(read->[,in->][,form->][,maketrans->]
 [,readfirst->][,trans->][,keep->][,obs->]
 [,filter->][,reject->] [,subdata->][,subread->]
 [,subin->][,subform->][,submaketrans->]
 [,subkeep->][,subobs->][,nobsw->][,nobswcum->][,obsw->]
 [,duplicate->][,oldsubobs->][,oldobsw->][,nobs->]
 [,buffersize->][,par->])
```

Output: Data set to be created If there is no output then the default is `$Data$`.

Options:

`read` variables read from the input file

`in` input file or list of input files. If no file given, data is read from the following input paragraph. If either of `read` or `in` option is given, then both options must be present.

`form` format, default is '*' format of Fortran, this can be indicated explicitly by `$`, '`b`' is binary. Any general Fortran format can be given as character constant or variable (e.g. `'(4f4.1,1x,f4.3)'`.

`maketrans` transformations computed for each observation when reading the data

`readfirst` variables read from the first line of the input file, if no variables are given, then anyhow first line is read and printed (a text header)

`trans` transformation set associated with the data set when data set is used later, does not have effect in making the data, and can be given later with `editdata` function.

`keep` variables kept in the data set, default: all `read` variables plus the output variables of `maketrans` transformations.

`obs` variable which gets automatically the observation number when working with the data, variable is not stored in the data matrix, default: `Obs`. When working with hierarchical data it is reasonable to give `obs` variable for each data set.

`filter` logical or arithmetic statement (nonzero value indicating True) describing which observations will be accepted to the data set. `Maketrans`-transformations are computed before using filter.

`reject` logical or arithmetic statement (nonzero value indicating True) describing which observations will be rejected from the data set, if filter option is given then reject statement is checked for observations which have passed the filter.

`subdata` the name of the lower level data set to be created. This option is not allowed, if there are multiple input files defined in option `in`.

`subread`,….`subobs` sub data options similar as `read`…`obs` for the upper level data.

(`subform->'bgaya'` is the format for the Gaya system)

`nobsw A` variable in the upper data telling how many `subdata` observations there is under each upper level observation, necessary if `subdata` option is present.

`nobswcum` A variable telling the cumulative number of subdata observations up to the current upper data observation but not including it. This is useful when accessing the data matrix one upper level unit by time, i.e., the observation numbers within upper level observation are `nobswcum+1`,…,`nobswcum+nobsw`

`obsw` variable in the subdata which automatically will get the number of observation within the current upper level observation, i.e. `obsw` variable gets values from 1 to the value of `nobsw`-variable, default is '`obsw%`*`obs_variable`*'.

`duplicate->(duplicates-transformations,duplicate-transformations)` The two transformation set arguments describe how observations in the subdata will be

duplicated. The first transformation set should have Duplicates as an output variable so that the value of Duplicates tells how many duplicates are made (0= no duplication). The second transformation set defines how the values of subdata variables are determined for each duplicate. The number of duplicate is transmitted to the variable Duplicate. These transformations are called also when Duplicate=0. This means that when there is the `duplicate` option, then all transformations for the subdata can be defined in the duplicate transformation set, and `submaketrans` is not necessary.

`oldsubobs` if there are duplications of sub-observations, then this option gives the variable into which the original observation number is put. This can be stored in the subdata by putting it into `subkeep` list, or, if `subkeep` option is not given then this variable is automatically put into the `keep` list of the subdata.

`oldobsw` This works similarly with respect to the initial `obsw` variable as oldsubobs works for initial `obs` variable.

`nobs` Number of observations, if known beforehand. Currently a data set can be created from the air by using `nobs` option and `maketrans` transformation, which can use `obs` variable as argument. Creation of data set this way is indicated by the presence of `nobs` option and absence of `in` and `read` options.

`buffersize` the number of observations put into one temporary working buffer. The default is 10000. Experimentation with different values of `buffersize` in huge data sets may result in more efficient `buffersize` than is the default.

`par` additional parameters for reading. If `subform` option is '`bgaya`' then par option can be given in form `par->(ngvar,npvar)` where `ngvar` is the number of nonperiodic x-variables and `npvar` is the number of period specific x-variables for each period. Default values are `par->(8,93)`

***In future the use of `nobs` option will make `data`-function faster also when reading data from a file.

`data` function will create a data set object, which is a compound object consisting of links to data matrix, etc. see *Data set object*.

Note 1. See common options section for how data sets used in other **J** functions will be defined.

Note 2: All `read` variables are treated as real variables.

Note 3: The `in` and `subin` can refer to the same file, or if both are without arguments then data are in the following input paragraph. In this case `data` function read first one upper level record and then `nobsw` lower level records.

9.2 *Modifying an existing data set: editdata( )*

`editdata(data_set,trans->)`

Argument: data_set , a data set object

Option: `trans` , gives the transformation to be done for each observation when dealing with the data. If removing existing transformation without a new one, give `trans->`, or `trans->0`

Changes the transformation set associated with the data set

***Coming ways to change old data sets, make new data sets from old ones, get observation matrix from matrix made by other means. It will be possible to keep the data in a file in cases there is shortage of memory.

9.3 *Linking hierarchical data: linkdata( )*

`linkdata(data->,subdata->,nobsw->[,obsw->])`

Links hierarchical data sets.

options:

`data` the name of the upper level data set

`subdata` the name of the lower data set

`nobsw` the name of variable telling the number of lower level observations for each upper level observation, now `nobsw` must be an existing variable.

***It will be later possible to link data when the class variable is in the subdata.

`obsw` variable which will automatically get the number of lower level observation within each upper level observation. If not given, then this variable will be `obsw%`*`obs_variable_of_the_upper_data`*

Note 1: In most cases links between data sets can be either made using sub-options of `data` function or `linkdata` function. If there is need to duplicate lower level observations, then this can be currently made only in `data` function. Also when the data for both the upper level and lower level data are read from the same file, then `data` function must be used.

Note 2: When using linked data in other functions, the values of the upper level variables are automatically obtained when accessing lower level observations. Which is the observational unit in each function is determined which data set is given in `data` option or defined using `Data` list.

9.4 *Combining two observations in same class: crossed( )*

`=crossed(data->,class->,trans->,keep->,dummy->)`

Output: an data set

For each class defined by the class variable, each observation pairs form a new observation in the output data set. Assume that crossed is called with `trans->tr` and `dummy->same`. The algorithm can be described

do c=1, number of classes
do i=first observation in class, last observation in class
$Stage=1
call transformation set tr
do j=first observation in class, last observation in class
if(i==j)then
 same=1
else
 same=0
endif
$Stage=2
call transformation set tr
make new observation in the output data storing variables defined in keep->
enddo over j
enddo over i
enddo over classes

9.5 *Utility functions for data sets*

9.5.1 Extracting values of class variables: values( )

=values(*variable[,data->]*)

Gets all different values of a variable in one or several datasets into a vector.

Output: A columns vector getting different values

Argument: a data set variable (either stored in the data matrix or generated with the associated transformations).

Option: `data` gives the data sets searched

Note 1. the values found will be sorted in an increasing order

Note 2. After getting the values into a vector, the number of different values can be obtained using `ncols` function.

***Later there will be different ways to utilize the obtained values in connection of data sets. Now the `values` function can be utilized e.g. in generating domains for all different owners or regions found in data.

9.5.2 Number of observations: nobs( )

`nobs(dataset)`

Gets the number of observations in a data set.

Argument : a data set

Note: `index` function described in *List functions* chapter is needed when doing transformations using the data matrix of a data object

An example of `nobs` and `index`:

Fast:

```
do(i,2,nobs(dat))
write('outfile.dat','b',
matrix%dat(i,"index(x4,keep%dat)")-matrix%dat(i-1,
"index(x4,keep%dat)"))
enddo
```

Fast:

```
inx4=index(x4,keep%dat)
do(i,2,nobs(dat))
write('outfile.dat','b',
matrix%dat(i,inx4)-matrix%dat(i-1,inx4))
enddo
```

Slow:

```
do(i,2,nobs(dat))
write('outfile.dat','b',
matrix%dat(i,index(x4,keep%dat))-matrix%dat(i-1,index(x4,keep%dat)))
enddo
```

9.5.3 Getting an observation from a data set: getobs( )

`getobs(dataset,obs[,trans->])`

Get the values of all variables associated with observation `obs` in data set `dataset`. First all the variables stored in row `obs` in the data matrix are put into the corresponding real variables. If a transformation set is permanently associated with the data set, these transformations are executed. Then if there is `trans->` option present, these transformations are also executed-.

9.6 *Data set object*

Data set is a compound **J** object created by the `data` function. A data set is linking together data, variable names, case names (coming later), transformations, links to other data sets. In the following (A) indicates that the part is created automatically, (N) that the part is necessary and the user can give the name for the part, and (O) indicates that the part may or may not exist. The name of the data set is indicated by *data.*

parts:

`matrix`%*data* (A) matrix containing the data values

`keep`%*data* (A) variable list telling the variables in the data (columns names)

***later: *cases (O)*: link to case names

prolog (O) link to initialization transformations done before starting to handle the data

trans (O) link to transformations done for each observation

epilog (O)= link to transformations done after last observation

vars%*data(A)* variable list merging keep%*data* and output%*trans*

obs (N) link to variable which will obtain the observation number automatically (default: `Obs )`

up (O)= link to an upper level data set whose subdata this is (e.g. stand data for the tree data)

sub (O)= link to the lower level subdata (e.g. schedule data for the stand data)

nlink (O)=link to the variable telling the # of lower level observations

Note: `matrix`%*data* , `keep`%*data* and vars%*data* are named element objects which can be accessed also directly.

10 Statistical functions

10.1 *Basic statistics: stat()*

stat(var1,…,varn[,data->][,weight->][,min->]
[,max->][,mean->][,var->][,sd->][,sum->]
[,nobs->][,filter->][,reject->][,trans->]
[,transafter->])

Computes and prints basic statistics from data sets.

Arguments: variables for which the statistics are computed.

Options:

`data` data sets , see section *Common options* for default

`weight` gives the weight of each observations if weighted means and variances are computed. The weight can be given in form of transformation or it can be a variable in the data set

`min` defines to which variables the minima are stored. If the value is character constant or character variable, then the name is formed by concatenating the character with the name of the argument variable. E.g. `stat(x1,x2,min->'pien%')` stores minimums into variables `pien%x1` and `pien%x2`. The default value for `min` is `'min%'`. If the values of the `min` option are variables, then the minima are stored into these variables.

`max` maxima are stored, works as `min`

`mean` means are stored

`var` variances are stored

`sd` standard deviations are stored

`sum` sums are stored, (note that sums are not printed automatically)

`nobs` gives variable which will get the number of accepted observations, default is variable '`nobs`'. If all observations are rejected due to filter or reject option, then an error occurs unless `nobs` option is given (utilizing the nobs variable the user can control how the execution continues)

`trans` transformation set which is executed for each observation. If there is a transformation set associated with the data set, those transformations are computed first.

`filter` logical or arithmetic statement (nonzero value indicating True) describing which observations will be accepted. `Trans`-transformations are computed before using filter.

`reject` logical or arithmetic statement (nonzero value indicating True) describing which observations will be rejected, if filter option is given then reject statement is checked for observations which have passed the filter.

`transafter` transformation set which is executed for each observation which has passed the filter and is not rejected by the `reject` option.

`stat` prints min, max, means, sd and sd of the mean computed as sd/sqrt(number of observations)

**If the value of a variable is greater than or equal to 1.7e19, then that observation is rejected when computing statistics for that variable.

Example:

```
stat(area,data->cd,sum->bon20,filter->(site.ge.18.5))
stat(ba,data->cd,weight->area)
stat(vol,weight->(1/dbh***2))
```

10.2 Covariance matrix: cov()

cov(var1,…,varn[,data->][,weight->][,filter->]
[,reject->][,trans->][,transafter->])

Computes variance –covariance matrix .

Arguments: variables for which the variances and covariances are computed.

Options:

`data` data sets , see section *Common options* for default

`weight` gives the weight of each observations if weighted means and variances are computed. The weight can be given in form of transformation or it can be a variable in the data set

`trans` transformation set which is executed for each observation. If there is a transformation set associated with the data set, those transformations are computed first.

`filter` logical or arithmetic statement (nonzero value indicating True) describing which observations will be accepted. `Trans`-transformations are computed before using filter.

`reject` logical or arithmetic statement (nonzero value indicating True) describing which observations will be rejected, if `filter` option is given then reject statement is checked for observations which have passed the filter.

`transafter` transformation set which is executed for each observation which has passed the filter and is not rejected by the `reject` option.

**Currently the `cov` function does not print the matrix, it can be printed using `print` function.

10.3 Correlation matrix: corr()

corr(var1,…,varn[,data->][,weight->][,filter->]
[,reject->][,trans->][transafter->])

Computes the correlation matrix. Arguments and option are as in the previous `cov` function. If a variable has zero variance, the correlation with the same variable is defined to be one and correlations with other variables are zero.

10.4 *Classifying data: classify( )*

***classify*([*var1*,…,*varn*][,*data->*],*x->*[,*xrange->*][,*dx->*]**
[,*classes*][,*z->*][,*zrange->*][,*dz->*][,*mean->*]
[,*trans->*][,*filter->*][,*reject->*][,*transafter->*])

Classifies data with respect to one or two variables, get class frequencies and means of argument variables

Output: a matrix containing class information (details given below)

Arguments: variables for which class means are computed.

Options:

`data` data sets used, if option is not given default data sets are used

`x` the first variable defining classes

`xrange->(min,max)` defines the range of x variable and class width if several values of x variable are put into the same class. If `xrange` is not given all values of the `x` variable define its own class.

`dx` defines the class width for a continuous x variable. If `dx` is not given, range is divided into 7 classes.

`classes` number of classes, has effect if dx is not defined in `xrange`

`z` the second variable defining classes in two dimensional classification.

`zrange->(min,max)` defines the range and class width for a continuous z variable.

`dz` defines the class width for a continuous z variable.

`mean` if `z` variable is given, class means are stored in a matrix given in the mean option

`trans` transformation set which is executed for each observation. If there is a transformation set associated with the data set, those transformations are computed first.

`filter` logical or arithmetic statement (nonzero value indicating True) describing which observations will be accepted

`reject` logical or arithmetic statement (nonzero value indicating True) describing which observations will be rejected, if `filter` option is given then `reject` statement is checked for observations which have passed the filter.

`transafter` transformation set which is executed for each observation which has passed the filter and is not rejected by the `reject` option.

#Operation:

If z variable is not given then first row both in printed output and in the output matrix (if given) contains class means of the x variable. In the output matrix the last element is zero. Second row shows number of observations in class, and the last element is the total number of observations. Third row shows the class means of the argument variable. The fourth row in the output matrix shows the class standard deviations, and the last element is the overall standard deviation.

If z variable is given the first column shows the class means of z variable.

10.5 *Linear regression: regr( )*

10.5.1 Computing the regression function: regr()

`regr(y,x1,…,xn[,data->][,noint->][,trans->]`
`[,filter->][,reject->][,transafter->])`

Computes a linear regression function.

Output : a regression object, utilized through `value`, `coef`, `se`, `rmse`, `mse`,`r2` functions

Arguments: y-variable, x-variables (not including constant term)

Options:

`data` data sets used

`noint` intercept is not included (default is to include)

`trans` transformation set which is executed for each observation. If there is a transformation set associated with the data set, those transformations are computed first.

`filter` logical or arithmetic statement (nonzero value indicating True) describing which observations will be accepted

`reject` logical or arithmetic statement (nonzero value indicating True) describing which observations will be rejected, if `filter` option is given then `reject` statement is checked for observations which have passed the filter.

`transafter` transformation set which is executed for each observation which has passed the filter and is not rejected by the `reject` option.

10.5.2 Using the regression object: value(),coef(),se(),mse(),rmse(),r2(), nobs(), len()

When a regression object has been created with `regr` function, it can be utilized using the following functions.

`value(regr_object[,x1,…,xn])`

Computes the value of the regression function. If the regression object is the only argument, then the current values of the independent variables are used. If the values of the independent variables are given as arguments, they are used. They must be in the same order as in the `regr` function which created the object.

`coef(regr_object,xvar)`

Gives the value of the coefficient of a x-variable.

Note: `coef(regr_object,1)` returns the intercept

`se(regr_object,xvar)`

Gives the estimated standard error of the coefficient of a x-variable.

Note: `se(regr_object,1)` returns the standard error of the intercept

`mse(regr_object)`

Returns the MSE of the regression.

`rmse(regr_object)`

Returns the RMSE of the regression

`r2(regr_object)`

Returns the R^2 of the regression

`nobs(regr_object)`

return number of observations used to compute the regression

`len(regr_object[,any->])`

return the number of parameters in the regression (including intercept)

Option

`any len` returns value-1 if argument is not legal object for `len` (without `any->` an error occurs)

*** Functions for accessing F and p values will be added when needed.

10.6 *Smoothing spline: smooth( )*

A smoothing spline can be computed with `smooth` function. This function is using an algorithm gcvspl from Netlib

10.6.1 Smoothing spline directly from data

For small data sets smoothing spline can be computed using each value of the x variable as a knot point.

`=smooth(y,x,[data->][,variance->][,modeldf->]`
`[,degree->][,wish->])`

Output: a smoothing spline object, can be used through `value(output,x)`, and parameters of the fit can be accessed by `param(output,pram_index)`

Arguments: dependent variable, independent variable

Options:

`data` data sets used

`variance` Variance of each observation (weight will be inverse of variance). Can be a variable or statement function.

`modeldf` effective degrees of freedom used for model parameters, if not given then the generalized cross validation value is minimized, and the effective degrees of freedom is obtained as an output parameter which can be accesses through `param(output,3)`

`degree` degree of polynomial used, feasible values are 1,3, … corresponding to linear, cubic, etc functions..If even value is given then it is turned into the nearest lower value. Deafult is `degree->3`.

`wish->(x1,y1,w1,…,xn,yn,wn)` gives wishes for the points through which the spline should go. For each triplet (xi.yi,wi) and artificial data point with x value xi and y value yi and weight wi is added to the data. The larger is the weight the closer the smoothing spline will be to the point. Weight 1 is the weight for one observation.

The parameters of the fitting are printed, and they can be accessed through `param` function:

`param(output,1)`=Generalized Cross Validation Value

`param(output,2)` = Mean Squared Residual

`param(output,3)`=Estimated df for the model (=modeldf , if this option is given)give

`param(output,5)`=Estimated true MSE

`param(output,6)`=Gauss Markov variance

`param(output,7)`= number of data points

*** Currently this does not work if the same x-value appears several times. In that case the smoothing spline can be computed by first classifying the data.

10.6.2 Smoothing spline from classified data

For large data sets the smoothing spline can be computed by first computing class means using classify function, and the computing the smoothing spline using class means as data point.

`=smooth(class_matrix[,variance->][,modeldf->][,degree->]`
`[,wish->][,min->][,max->][,maxiter->][,iterations->])`

Output: a smoothing spline object, can be used through `value` function

Arguments: `class_matrix` is a matrix of class means generated by classify function.

Options:

`variance` Variance of each observation (weight will be inverse of variance). Can be a variable or statement function. It is taken automatically into account that the variance of the class mean of the y variable is inversely proportional to the number of observations in the class.

`modeldf, degree` see above (`smooth` function)

`wish->(x1,y1,w1,…,xn,yn,wn)` see above (`smooth` function)

`min->(fmina [,fminb])` The required lower bound for the function. If only one `value (fmina)` is given then after obtaining the initial smoothing spline it is checked if the value of spline if smaller than `fmina` and if it is, the y-value of the point is replaced with `fmina`, and the smoothing spline is computed again. It may, however, be that the values of the smoothing spline are not smaller than `fmina`. If value `fminb` is given (`fminb<fmina`), then the y values are replaced with `fmina`-(iteration_count-1)*(`fmina-fminb`), and the procedure is repeated until four iterations.

`max->(fmaxa [,fmaxb])` The required upper bound for the function. Works as `min` option.

`maxiter` give the maximum number of iterations to get the function to obey min or max constraints, default is 6.

`iterations` gives the variable which obtains the used number of iterations. Can be useful to stop automated iterations to look more closely to problematic cases.

**Note that `min` and `max` options do not yet work for smoothing data.

11 Linear programming (JLP functions)

JLP is a linear programming package described in Lappi (1992). J is designed to substitute this package. The linear programming J functions are called JLP functions. JLP functions are 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 manual, 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.

See Lappi (1992) for the background of the linear programming as used in **J**.

J optimization example with output explained is introduced in chapter 11.9 JLP examples.

11.1 *Optimization problem*

Mathematically the optimization problems considered can be described as follows. Let us first define a linear programming problem without assuming domains for constraints. An optimization problem can be presented as:

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

subject to the following constraints:

$$c_t \le \sum_{k=1}^{p} a_{tk} x_k + \sum_{k=1}^{q} b_{tk} z_k \le C_t, \quad t = 1, \ldots, r \tag{2}$$

$$x_k - \sum_{i=1}^{m} \sum_{j=1}^{n_i} x_k^{ij} w_{ij} = 0, \quad k = 1, \ldots, p \tag{3}$$

$$\sum_{j=1}^{n_i} w_{ij} = A_i, i = 1, \ldots, m \tag{4}$$

$$w_{ij} \ge 0 \quad \text{for all } i \text{ and } j \tag{5}$$

$$z_k \ge 0 \quad \text{for } k = 1, \ldots, q \tag{6}$$

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 per unit area 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
A_i	=	area of unit i

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

The constraints of form (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 (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. The constraint (3) can be equivalently written as:

$$x_k = \sum_{i=1}^{m} \sum_{j=1}^{n_i} x_k^{ij} w_{ij}, \quad k = 1,\ldots,p \tag{7}$$

The less intuitive form is used in (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 (3) or in (7), or to constants x_k^{ij}.

Constraints (4) are so called area constraints saying that the areas under different schedules add up to the total area of the stand. If coefficients x_k^{ij} are expressed as the total amount in the unit (instead per area), then w_{ij}'s are proportions and each area is 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. (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:

$$\text{Max or Min} \quad z_0 = \sum_{k=1}^{p} a_{0k} x_{kD_0} + \sum_{k=1}^{q} b_{0k} z_k, \tag{8}$$

subject to:

$$c_t \le \sum_{k=1}^{p} a_{tk} x_{k D_t} + \sum_{k=1}^{q} b_{tk} z_k \le C_t, \quad t = 1,\ldots,r \tag{9}$$

$$x_{kD_t} - \sum_{i \in D_t} \sum_{j=1}^{n_i} x_k^{ij} w_{ij} = 0 \quad , \quad k = 1,\ldots,p, \qquad t = 1,\ldots,r \qquad (10)$$

$$\sum_{j=1}^{n_i} w_{ij} = A_i, i = 1,\ldots,m \qquad (11)$$

$$w_{ij} \geq 0 \quad \text{for all } i \text{ and } j \qquad (12)$$

$$z_k \geq 0 \quad \text{for } k = 1,\ldots,q \qquad (13)$$

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.

The user of JLP functions needs only to define objective function (1) or (8) and the utility constraints (2) or (10), and J takes care of the other constraints utilizing the special structure of the problem.

11.2 *Optimization problem including factories*

In a factory problem, the transportations costs of different timber assortments at specified time periods and capacities of factories at the same time periods are included in the problem definition. For instance, the net present value can be maximized subject to capacity constraints and sustainability constraints.

Mathematically the optimization problems including factories can be defined as follows

$$\text{Max or Min } z_0 = \sum_{k=1}^{p} a_{0k} x_k + \sum_{k=1}^{q} b_{0k} z_k + \sum_{k=1}^{p} \sum_{f=1}^{F} \alpha_{0kf} x_{kf} + \sum_{k=1}^{p} \sum_{f=1}^{F} \beta_{0kf} y_{kf} \qquad (14)$$

subject to the following utility constraints

$$c_t \leq \sum_{k=1}^{p} a_{tk} x_k + \sum_{k=1}^{q} b_{tk} z_k + \sum_{k=1}^{p} \sum_{f=1}^{F} \alpha_{tkf} x_{kf} + \sum_{k=1}^{p} \sum_{f=1}^{F} \beta_{tkf} y_{kf} \leq C_t, \quad t = 1,\ldots,r \; (15)$$

and technical constraints

$$x_k - \sum_{i=1}^{m} \sum_{j=1}^{n_i} x_k^{ij} w_{ij} = 0, \quad k = 1,\ldots,p \qquad (16)$$

$$\sum_{j=1}^{n_i} w_{ij} = A_i, i = 1,\ldots,m \qquad (17)$$

$$x_{kf} - \sum_{i=1}^{m} x_{kf}^{i} = 0, \quad (k, f) \in \mathbf{R} \tag{18}$$

$$y_{kf} - \sum_{i=1}^{m} \gamma_{kf}^{i} x_{kf}^{i} = 0, \quad (k, f) \in \mathbf{B} \tag{19}$$

$$\sum_{f=1}^{F} x_{kf}^{i} - \sum_{j}^{n_i} x_{k}^{ij} w_{ij} = 0, \quad i = 1, \ldots, m, \ k \in \mathbf{K} \tag{20}$$

$w_{ij} \geq 0, \ i = 1,...,m, \ j = 1,...,n_i$**,** $z_k \geq 0 \quad \text{for } k = 1, \ldots, q$

$x_{kf}^{i} \geq 0, (k, f) \in \mathbf{R}$

$x_{k}^{ij} \geq 0, k \in \mathbf{K}$ (21)

where m, n_i, w_{ij}, x_k^{ij}, x_k, z_k, a_{tk}, b_{tk}, r, A_i as described in the chapter 11.1 *Optimization problem* and

α_{tkf}	=	fixed real constants for t= for t=0,...,r, k=1,...,p, f=1,...,F
β_{tkf}	=	fixed real constants for t= for t=0,...,r, k=1,...,p, f=1,...,F
x_{kf}^{i}	=	x_k -variable transported from unit i to factory f
y_{kf}	=	utility obtained when a forest variable k is transported to factory f taking into account the transportation costs
γ_{kf}^{i}	=	utility when one unit of forest variable k is transported from treatment unit i to factory f, the transportation cost is taken into account
F	=	number of factories
R	=	set of (k, f) such that $\alpha_{tkf} > 0$ or $\beta_{tkf} > 0$ for some t
B	=	set of (k, f) such that $\beta_{tkf} > 0$ for some t
K	=	set of such k that $\alpha_{tkf} > 0$ or $\beta_{tkf} > 0$ for some t and f

Note: Domains cannot be defined for constraints including factory variables.

The meaning of different constraints:
Constraints (16) and (17) have the same meaning as constraints (3) and (4) in the chapter 11.1 Constraint (5) states that forest variable k assigned to factory f is obtained by adding up all standwise assignments of variable k into factory f. Constraint (19) tells that transportation cost of forest variable k to factory f is obtained by summing up standwise transportation costs. Constraint (20) tells that all of forest variable k is transported to factories. Note that constraint (21) is not standard linear programming constraint, because it constrains the values of the problem coefficients, not variables. Note that taking into account constraint (19), $\beta_{tkf} y_{kf} = \sum_{i=1}^{m} \beta_{tkf} \gamma_{kf}^{i} x_{kf}^{i}$.

Thus multiplying γ_{kf}^{i} for each k and f by a constant and dividing each β_{tkf} by the same constant we can get an equivalent problem. Thus we can assume without loss of

generality that each β_{tkf} is one. This assumption is made in J but the formulas are presented below without this assumption.

Usually y_{kf} -variables appear in the objective row as a part of the definition of the net present value. When trying to understand the formulas, it might be easier to consider that y_{kf} is the total discounted transportation cost for variable *k* and γ^i_{kf} is the discounted per unit transportation cost when variable *k* is transported from unit *I* to factory *f*. When the also the utility of having variable *k* transported to factory *f* is taken into account in y_{kf} and γ^i_{kf} we get more efficient computations and more compact problem definition in J. Note that we can have different factory groups for different timber assortments by setting properly zeroes to alfas and betas.

In typical problems the utility constraints including x_{kf} are of form $x_{kf} \le C$ which states that the capacity of factory *f* has a upper bound *C* for a period-specific timber assortment.

The user of JLP functions must specify the objective (14) and the utility constraints (15) and give information how the program can compute coefficients γ^i_{kf} . The program takes care automatically of the constraints 17-20. In problems including factories you can define only maximization problems. The minimization problem can be turned into maximization by multiplying the objective function by -1.

11.3 *Solution algorithm*

Function `jlp` is using the algorithm of Lappi (1992), based on the generalized upper bound technique of Dantzig and VanSlyke (1967). Function is using linear algebra subroutines of Prof. R. Fletcher based on Fletcher (1996)

Solution algorithm for the optimization problem including factories is described in Lappi and Lempinen (2013).

11.4 *J functions related to JLP*

In order to use JLP functions user should be familiar with at least `data`, `linkdata`, `trans` and `print` functions.

11.5 *Problem definition: problem( )*

`=problem([repeatdomains->])`

…

`/`

Define a lp problem for `jlp` function.

Output: a problem definition object

Option: `repeatdomains`, if this is option is given then the same domain definition can be in several places of the problem definition, otherwise having the same domain in different places causes an error (as this is usually not what was intended)

The problem definition paragraph can have two types of lines: problem rows and domain rows. Examples of problem definitions showing the syntax.

```
pr=problem()   !ordinary lp-problem
7*z2+6*z3-z4==min
2*z1+6.1*z2 >2 <8+b
(a+log(b))*z5-z8=0
-z7+z1>8
/
prx=problem()  ! timber management planning problem
All:
npv.0==max
sitetype.eq.2: domain7:
income.2-income.1>0
/
```

In the above example `domain7` is a data variable. Unit belongs to domain if the value of the variable `domain7` is anything else than zero.

Currently the objective row must be the first row. Function: problem interprets the problem paragraph, and extracts the coefficients of variables in the object row and in constraint rows. The coefficients can be defined using arithmetic statements utilizing the input programming "-sequence or enclosing the coefficient in parenthesis. The right hand side can utilize arithmetic computations without parenthesis. The values are computed immediately. So if the variables used in coefficients change their values later, the problem-function must be computed again in order to get updated coefficients. Note that a problem definition does not yet define a JLP task. Final interpretation is possibly only when the problem definition and simulated data are linked in a call to `jlp` function. At the problem definition stage it is not yet known which variables are z-variables and which are x-variables (see Lappi 1992).

Note that '<' means less or equal, and '>' means greater or equal. The equality is always part of linear programming constraints.

Note also that problem definition rows are not in one-to-one relation to the constraint rows in the final lp problem. A problem definition row may belong to several domains, thus several lp-constraint rows may be generated from one problem definition row.

Domain definitions describe logical or arithmetic statements indicating for what management units the following rows apply. Problem will generate problem definition object, which is described below:

Note 1: Only maximization is allowed in problems including factories. To change a minimization problem to a maximization problem, multiply the objective function by −1.

*** We may later add the possibility to define also minimization problems.

Note 2: If optimization problem includes factories (see chapter *11.2 Optimization problem including factories*), there have to be y_{kf} variables in the objective function or at least in one constraint row. Example of problem definition including factories can be found in chapter *11.9 JLP Examples*.

11.6 *JLP problem definition object*

generated with: `problem`

used in: `jlp`

JLP problem is a compound object created by `problem`-function (similar to JLP problem definition) for defining lp-problems.

parts:

`rows%`*problem* is a text object containing the rows of the problem definition

`domains%`*problem* is a text object containing the domain definitions or names used in the problem

`domainvar%`*problem* variable list containing the variables used in the domain definitions

`vars%`*problem* variable list containing the variables used in the problem definition

`rhs%`*problem* vector containing the lower bound for each row

`rhs2%`*problem* vector containing the upper bounds for each row

All coefficients of the constraints are in a packed format.

Note: the rhs- vectors can be modified (by arithmetic or matrix operations) before and between calls of the `jlp`-function utilizing the same problem definition.

11.7 *Solving a problem: jlp( )*

A lp problem defined by `problem` function can be solved using `jlp` function:

```
jlp(problem->[,data->][,z->][,sparse->][,tole->]
 [,subfilter->][,subreject->] [,class->]
 [,area->][,notareavars->][,print->][,report->]
 [,maxiter->][,test->][,debug->])
```

options:

`problem` problem definition generated by `problem` function

`data`: data set describing the stand (management unit) data or the schedules data. The unit data set must be linked to schedule data either using sub-options in the `data` function or using `linkdata` function. Following the JLP terminology, the unit data is called cdata, and the schedule data is called xdata. The `jlp` function tries if it can find a subdata for the data set given. If it finds, it will assume that the data set is the unitdata. If subdata is not found, it tries to find the upper level data. If it finds it, then it assumes that the data set given is the schedules data. If `data` is not given, then the problem describes an ordinary lp-problem, and all variables are z-variables. If `data`-option is given but no variable found in problem is in the schedules data set, then an error occurs.

z If the data option is given then the default is that there are no z-variables in the problem. The existence of z-variables must be indicated with z option (later the user can specify exactly what are the z variables, but now it is not possible). The reason for having this option is that the most jlp-problems do not have z variables, and variables which J interprets as z-variables are just accidentally missing from the data sets.

******* Later we may add the possibility to have several data sets

`sparse` sparse matrix routines are used, should be used in large problems with many domains *** not tested properly and cannot be used in factory problems

`tole` the default tolerances are multiplied with the value of the `tole` option (default is thus one). Smaller tolerances mean that smaller numerical differences are interpreted as significant. If it is suspected that jlp has not found the optimum, use e.g. `tole->0.1` ,`tole->0.01` or `tole->10.` Also in case of `jlp` reporting that solution is getting worse increase the value of the `tole` option e.g.set `tole->10.`

`subfilter` logical or arithmetic statement (nonzero value indicating True) describing which schedules will be included in the optimization. If all schedules are rejected, an error occurs. Examples: `filter->(.not.clearcut)` , `filter->(ncuts.ge.5)`,`filter->harvest` (which is equivalent to: `filter->(harvest.ne.0)`). If the `subfilter` statement cannot be defined nicely using one statement, the procedure can be put into a transformation set which can be then executed using value function.

`subreject` logical or arithmetic statement (nonzero value indicating True) describing which schedules will not be included in the optimization. If `subfilter` is given then test applied only for such schedules which pass the subfilter test. If the `subreject` statement cannot be defined nicely using one statement, the procedure can be put into a transformation set which can be then executed using `value` function.

`class class->(cvar, cval)` Only those treatment units where the variable `cvar` gets value `cval` are accepted into the optimization. The units within the same class must be consecutive.

`area` gives the variable in cdata which tells for each stand the area of the stand. It is then assumed that all variables of cdata or xdata used in the problem rows are expressed as per are values. In optimization the proper values of variables are

obtained by multiplying area and per area values. Variables of cdata used in domain definitions are used as they are, i.e. without multiplying with area. Variables which are not treated as per area values are given with the `notareavars` option.

`notareavars` If `area->` option is given then this option gives variables which will not be multiplied with area.

`print` of output printed, `1 =>` summary of optimization steps, `2=>` also the problem rows are printed, `3=>` also the values of x-variables are printed.

`report` the standard written output is written into the file given in report option (.e.g. `report->'result.txt'`). The file remain open and can be written by several jlp-functions or by additional write functions. Use `close` function to close it explicitly if you want to open it with other program.

`maxiter` maximum number of rounds through all units (default 10000).

`test` If option is present then jlp is checking the consistency of the intermediate results after each pivot step of the algorithm. Takes time but helps in debugging.

`debug` determines after which pivot steps jlp starts and stops to print debugging information to fort.16 file. If no value given, the debugging starts immediately (produces much output, so it may be good to use step number which is close to the step where problems started (print variable `Pivots` at the error return). `debug->(ip1,ip2,ip3)` indicates that debugging is put on at pivot step `ip1`, off at pivot `ip2` and the again on at pivot `ip3`.

jlp is generating output (amount is dependent on the `print` option) plus a JLP-solution stored in special data structures which can be accessed with special **J** functions described below. In addition `jlp` creates three lists: `zvars%`*problem*, `factories%`*problem* and `xkyk%`*problem*.

`zvars%`*problem* stores the variables among `vars%`*problem* which are interpreted to be z-variables. Note that the z-variables get directly the optimal values, and they can be accessed directly without any access functions.

`factories%`*problem* and `xkyk%`*problem* lists stores factories and x_k variables found in optimization problem including factories.

The variables receiving the status of the problem are:

`Feasible` logical variable (i.e. gets value 1 if problem is feasible, zero otherwise)

`Optimal` logical variable for indicating if the solution is optimal

`Unbounded` logical variable for indicating if the solution is unbounded

`Started_jlp` logical variable telling if `jlp`-function initialized data structures so that inquiry functions can be used. Note that even if the problem is infeasible, these

inquiry functions return the current status of the problem solution. If the inquiry functions are used when they cannot yet be used, an error conditions occurs.

`Pivots` number of pivot operations, can be used to set a good value for debug-option in case of trouble.

`Objective` value of the objective function, for non-feasible problem -9.9, for unbounded problem either 1.7e37 (for maximization) and -1.7e37 for minimization

The logic of `jlp`-function is the same as in the old JLP software. There is one difference which makes the life a little easier with J. In J the problem definition can use c-variables which are defined in the stand data. These are used similarly as if they would become from the x-data. It does not make any sense to have on a problem row only c-variables, but there can be constraints like

```
vol#1-vol#0>0
```

where `vol#0` is the initial volume, i.e. a c-variable, and `vol#1` is the volume during first period. In old JLP these initial values had to be put into the x-data.

11.8 *Inquiry functions for the JLP solution*

The following **J** functions can access the most recent solution.

`=weights()`

Gives the number of schedules which have nonzero weight in the solution.

Note. this is usually used in combination with `unit`, `schedcum`, `schedw` and `weight` functions.

`unit(i)`

Returns the unit number for the i'th schedule having a nonzero weight,

Argument: `i` is numeric value between 1 and `weights()`

`schedcum(i)`

Returns the cumulative schedule number (observation number in the subdata) for the i'th schedule having a nonzero weight,

Argument: `i` is numeric value between 1 and `weights()`

`schedw(i)`

Returns the within unit schedule number for the i'th schedule having a nonzero weight.

Argument: `i` is numeric value between 1 and `weights()`

`weight(i)`

Returns the weight (proportion) for the i'th schedule having a nonzero weight,

Argument : i is numeric value between 1 and weights()

`partweights()`

Returns the number of schedules which have nonzero weight in the solution but so that the whole unit is not assigned to the schedule. In a linear programming problem there is usually only one schedule in each unit in the solution i.e. with a nonzero weight. Binding constraints bring in the solution schedules with weight between zero and one. The schedules can be access with `par`t functions.

Note. `partweights()` is usually used in combination with `partunit`, `partschedcum`, `partschedw` and `partweight` functions.

*** currently `partweights(unit)` gives also the number of partweights in the unit, but I'm not sure if I'll keep this

`partunit(i)`

Returns the unit number for the i'th schedule having weight between zero and one.

Argument: `i` is numeric value between 1 and `partweights()`

`partschedcum(i)`

Returns the cumulative schedule number for the i'th schedule having weight between zero and one

Argument: i is numeric value between 1 and `partweights()`

`partschedw(i)`

Returns the within-unit schedule number for the i'th schedule having weight between zero and one.

Argument: i is numeric value between 1 and `partweights()`

`partweight(i)`

Returns the weight for the i'th schedule having le having weight between zero and one.

Argument: i is numeric value between 1 and `partweights()`

`price%unit(iunit)`

Returns the shadow price of the unit i`unit`.

Argument: unit (stand) number

Note: If active rhs's are nonzero the shadow prices of a units (precisely, the shadow prices of the area constraints for units) do not generally add up to the solution. For more detailed information see Shadow price of a treatment unit in Lappi (1992).

`weight%schedcum(sched[,integer->])`

Returns the weight of a schedule.

Argument: cumulative schedule number

option:

`integer` : the weight will be 1 for that schedule within the unit which has the largest weight and zero otherwise

Note: `weight(i)` and `partweight(i)` return only nonzero values (precisely, weights for basic schedules, which are nonzero except for degenerate basic schedules), but `weight%schedcum()` return also zero weights

`price%schedcum(sched)`

Returns the shadow price of a schedule.

Argument: cumulative schedule number

Note: for all schedules in the basis, the value of the schedule is the same as the value of the unit given by `price%unit(unit)`.

`price%schedw(iunit,sched)`

Returns the shadow price of a schedule within an unit

Arguments:

`iunit` the number of the unit

`sched` the schedule number within the unit

`weight%schedw(iunit,sched[,integer->])`

purpose: numeric function returning the weight of a schedule within an unit

Arguments:

`iunit` the number of the unit

`sched` the schedule number within the unit

Option:

`integer` the weight will be 1 for that schedule within the unit which has the largest weight and zero otherwise

`integerschedw(iunit)`

purpose: numeric function returning the within-unit schedule number of the schedule which has the largest weight within the unit

Arguments:

`iunit` the number of the unit

`integerschedcum(iunit)`

purpose: numeric function returning the cumulative schedule number of the schedule which has the largest weight within the unit

Arguments:

`iunit` the number of the unit

`xkf(file)`

purpose: prints for each unit the amount of x_k variables transported to each factory. Output consists of four numeric values: index of the unit, index of the factory, index of the forest variable and the transported amount. Factory and forest variable indexes refer to the elements of `factories%`*problem* and `xkyk%`*problem* lists.

Arguments:

`file` variable `$` (indicating the console), or the name of the file as a character variable or a character constant.

11.9 *JLP examples*

Example: Definition of a problem including factories

Stand data file `sdata.dat` includes coordinates and number of schedules for each stand (coordinates are purely fictive and just an example).

`sdata.dat:`

```
1,2,3
5,6,2
```

```
2,3,3
4,6,2
```

Schedule data file `xdata.dat` describes the amounts of timber produced in each period if schedule is applied. The first three values represent the mount of saw log in periods 1, 2 and 3. The last three values represent amount of pulp correspondingly.

`xdata.dat:`

```
5342,4885,12,82,28,18
856,48965,42,782,87,596
0,0,45,4878,145,568
89,7,456,78,513,181
520,30,840,8,7,60
58,654,370,6,68,40
4584,564,578,516,20,54
452,70,16,35,37,39
25,3,8,21,39,37
36,35,34,8,19,45
```

Stand and schedule data are read with `data` function and linked together with `linkdata` function. Sawmills and pulp factories are introduced with `list` function. Coordinates, capacity and factory price of saw logs or pulp wood for each saw mill or pulp factory are defined with `properties` function.

```
!read in stand data
sdata=data(read->(sxcoor,sycoor,ns),in->'sdata.dat')
!read in schedule data
xdata=data(read-> (sawlog#1,sawlog#2,sawlog#3,pulp#1,pulp#2,pulp#3),
in->'xdata.dat')
linkdata(data->sdata,subdata->xdata,nobsw->ns)

p=3 ! number of periods
pl=10 ! length of period

r=1.05   !1.05  ! 1+interest
! define discounting factors
;do(i,1,p)
df#"i"=1/r**(-pl/2+i*pl)
;enddo

costpkm=0.011  !transportation cost per km

sawlog=list(sawlog#1...sawlog#"p")
pulp=list(pulp#1...pulp#"p")

sawmill=list(Lahti,Keuruu,Kotka)
! define variables xcoor%Lahti etc
properties(xcoor,ycoor,scapacity,sprice)
Lahti,50,100,2000,60
Keuruu,100,20,3000,55
Kotka,40,200,5000,70
/

pulpfactory=list(Oulu,Varkaus)
properties(xcoor,ycoor,pcapacity,pprice)
Oulu,20,400,3000,45
```

```
Varkaus,30,200,5000,40
/
```

Transformations are defined to compute the factory- and period-specific utility ((factory price – transportation cost) * discounting factor) which is then used in the objective function of the problem definition. Computing is implemented with input programming loops. The outer loop iterates over the factories (saw mills or pulp factories) and the inner loop iterates over the periods

```
trans.util%%sawlog%%sawmill=trans()
! each variable text1%%text2%%text3 needs to have associated
! transformation trans.text1%%…
!compute the transportation cost from coordinates
;do(i,1,len(sawmill))
cost=costpkm*sqrt((xcoor%@sawmill(i)-sxcoor)***2+
(ycoor%@sawmill(i)-sycoor)***2)
!use the factory price and the discounting factor
;do(j,1,p)
util%%sawlog#"j"%%@sawmill(i)=(sprice%@sawmill(i)-cost) *df#"j"
;enddo
;enddo
/

trans.util%%pulp%%pulpfactory=trans()
;do(i,1,len(pulpfactory))
cost=costpkm*sqrt((xcoor%@pulpfactory(i)-sxcoor)***2+
(ycoor%@pulpfactory(i)-sycoor)***2)
;do(j,1,p)
util%%pulp#"j"%%@pulpfactory(i)=(pprice%@pulpfactory(i)-cost) *df#"j"
;enddo
;enddo
/
```

The problem definition consists of the objective function and the constraints. The transformations defined above are used in the objective function, which is always the first row in the problem. The period- and factory-specific capacity constraints are also generated with input programming loops. There could be different capacities and factory prices for different periods, but in this example the same capacity and price applies for all periods.

```
!define the optimization problem
newprob=problem()
! objective function
util%%sawlog%%sawmill+util%%pulp%%pulpfactory==max
!capasity constraints
;do(i,1,len(sawmill))
;do(j,1,p)
sawlog#"j"%%@sawmill(i)<scapacity%@sawmill(i)
;enddo
;enddo
;do(i,1,len(pulpfactory))
;do(j,1,p)
pulp#"j"%%@pulpfactory(i)<pcapacity%@pulpfactory(i)
;enddo
;enddo
/
```

Finally solve the problem with `jlp` function

```
jlp(data->sdata,problem->newprob)
```

More examples will be available from 22.4.2013 at the web page http://mela2.metla.fi/mela/j/manuals/.

12 Simulator

12.1 *Defining a simulator*

J includes a simulator language as a slight extension of the ordinary transformations. Using simulator language in the `simulator` function, one can define a simulator. Simulations are done using `simulate` function which links a simulator and data sets. Optimal (or reasonable) treatment schedules can then be selected using JLP-functions. The structure of the simulator function is:

12.1.1 Simulator definition: simulator()

=simulator(periods->[,period->][,keepperiod->]
[,treevars->])
(*simulator definition*)
/
Output: a simulator

Options

`periods` number of simulation periods

`period` variable indicating the period during simulation, default `T`

`keepperiod` each node up to period `keepperiod`-1 must have at least one next function, default is the total number of periods. This option is not transmitted to simulate which has keepperiod which shows how many periods are actually simulated, but there is need for this option only if keepperiod is used in simulation and the simulator defines branches which do not reach all periods.

`treevars` gives the tree variables used in the simulation They can be used in the simulator if they were vectors. The `simulate` function will actually make these vectors for the simulation time. The `loadtrees` function will put the values of those tree variables which are in the tree data set linked to the stand data

The first part of the simulator paragraph is an initialization part, then there are definitions of nodes in any order, and definitions of sub module sections in any order. Branching structure is defined using `next` and `branch`. `Next` function tells which nodes are entered in the next period, branch will add nodes to the current period. The structure of the simulator can be best understood best by a simple example.

```
nper=5
js=simulator(periods->nper,period->P)
next(grow)
if(age#0.ge.50)next(thin)
if(age#0.gt.70)next(clear)
;do(t,1,nper)
;trace(age#"t",vol#"t",out->outvars#"t")
grow::t   ! node header consist of generic node name and the period
            !  number
age#"t"=age#"t-1"+10
vol#"t"=vol#"t-1"+25
jump('gr#"t"')
write($,'w',12,'grow/period',5,P,6,'age=',8,age#"t",5,'vol=',8,vol#"t
")
next(grow)
if(age#"t".ge.50)next(thin)
if(age#"t".eq.50)branch(thin2)
if(age#"t".ge.70)next(clear)
!test that all output variables got values
tracetest(outvars#"t")

thin::t
age#"t"=age#"t-1"+10
vol#"t"=0.6*vol#"t-1"
write($,'w',12,'thin/period',5,P,6,'age=',8,age#"t",5,'vol=',8,vol#"t
")
next(grow)
thin2::t
age#"t"=age#"t-1"+10
vol#"t"=0.6*vol#"t-1"
write($,'w',12,'thin2/period',5,P,6,'age=',8,age#"t",5,'vol=',8,vol#"
t")
next(grow)
tracetest(outvars#"t")

clear::t
age#"t"=0
vol#"t"=0
write($,'w',12,'clear/period
',5,P,6,'age=',8,age#"t",5,'vol=',8,vol#"t")
next(grow)
tracetest(outvars#"t")

sub
gr#"t":write($,'t',1,'kukuu"t"')
back
endsub
;enddo
/
```

The nodes are identified both by the generic node (treatment) name and the period. The nodes can be defined in any order, e.g. one can define first all growth nodes and the all thin nodes. It is not necessary to define all nodes for all periods, e.g. for initial periods there can be more treatment options. If there is a fixed number of treatment programs defining all the thinning times, then it may be useful not to define generic thinning nodes but thinning_at_age_80_in_program_1 type of nodes.

The `next` function tells that the argument nodes will be entered for the next period. The `next` function can have several arguments and it can be in any place in the node

section. The whole node section is always computed before going to next period. The `next` and `branch` function accumulate the branching nodes during the execution time.

Note 1: When adding nodes with `next` it is not tested if the nodes are already present (if this will cause difficulties in practice I may add such testing, possibly conditional on some test option)

Note 2: The `branch` function serves similar purpose as `next` function. The difference is that `next` function adds nodes to the next period but `branch` is adding nodes to the current period. It is tested if the nodes are already in the node list.

The sub sections contain start address`...back` subsections to which one can jump from any node. If there are period dependent computations in these 'subroutines', then also the starting addresses must be period specific.

There is no default action with respect to the `period` variable. It may be useful not to use period variable e.g. in the input programming `;do` loops as the index (`t` in the above example) in order not to confuse the period in the simulator definition and in the simulations.

Note 3. The simulator function checks during the simulator generation that all argument nodes of `next`-functions are defined. It also checks that there are `next` functions so that at least the `keepperiod` level can be reached. If `next` functions are dependent on logical conditions it may happen that during the simulation the `keepperiod` level is not reached. An error results in this case only if there is no branches reaching the `keepperiod` level.

Note 4. The above example show how `;trace` and `tracetest` can be used to check that all output variables get values at each node for each period. After testing the simulator properly, these can be commented out.

12.1.2 Special functions used in a simulator

`next(node1,…,nodem)`

Adds nodes to the list of children nodes of the current node.

see the `simulator` function above

`branch(node1,…,nodem)`

Adds nodes to the list of sister nodes of the current node.

see the `simulator` function above

It is first tested if the nodes are already in the list of sister nodes because the `next` command can have put the nodes into the list during the previous period at the mother node.

`cut()`

Removes all children nodes generated by previous `next` function calls (a regret function for `next`)

`loadtrees()`

Loads initial state tree variables into the tree vectors. What variables are loaded is determined within the `simulate` function. During the initialization phase of the simulate function it is checked which variables are both in the `treevars` option of the simulator and in the tree data set linked to the stand data, and the values are put into the initial positions of the tree variable vectors.

Example:
Assume that `d#0` gives the initial diameter, and there are initially `ntrees#0` trees in a stand. The one can defined diameters for next periods e.g.

```
loadtrees()
;do(t,1,nper)
growth::t
ntrees#"t"= ntrees#"t-1"   ! one can kill or make new trees so number may change
do(tree,1,ntrees#"t")
d#"t"(tree)=1.04*d#"t-1"(tree)
enddo
…;enddo
```

Note. It is possible to do tree level simulations without `loadtrees()` and tree data set: one can generate trees from the stand variables.

12.2 *Using a simulator: simulate ( )*

`simulate(simulator1[…,simulatorn][,data->]`
`[,selector->][,keep->][,keepperiod->][,obs->]`
`[,obsw->][,unitdata->][,unitdataobs->][,nobsw->])`

Simulate schedules

Output: data set storing stimulated schedules

Arguments: one or more simulator objects

Options:

`data` data set for stand data , if no data then the simulation is done using the values of variables as they are during the computation (in this case no output data set is generated (the simulator can of course e.g. write output files). If the simulator is using such tree variables (given in `treevars` option) which come from tree data, then these must be in the subdata linked to the data set.

`selector` If there are several simulator argument, there must be `selector` option which determines which simulator is selected for the current stand. The `selector`

option has one or two arguments. The first argument gives the transformation set which determines which simulator is selected. The second argument, if present, gives the name of list object which tells which simulator is selected. The default is `Selected`. See an example below.

`keep` The variables stored in the schedules data (output). If keep-variables are not given, then the output variables of the simulator are stored.

Note 1: variables with names starting with '`$`' are not counted as output variables.

Note 2. If there is no output, option is ignored.

`keepperriod` Each node in the simulated tree at the `keepperiod` level determines a schedule. The whole subtree below the node is visited before generating the schedule. The default for `keepperiod` is the value of the `periods` option of the simulator.

`obs` if output: the schedule data variable indicating the cumulative schedule number in the schedule data (output). Default is '`Sched`'.

`obsw` the schedule data variable indicating the schedule number within the current unit. Default [name of the obs-variable]//'`%`'/[name of the obs variable of the data set]

`unitdata` the data set containing all variables in the input data plus the `nobsw` variable (cdata of old JLP). The output schedule data is linked to the `unitdata` set so that thereafter `unitdata` data set can be used as the input data for the optimization (`jlp` function). The default is '`unitdata%`'//[the name of the output-variable]

`unitdataobs` the variable in the `unitdata` indicating the observation number. Default is '`Unit`'.

`nobsw` the variable in the unitdata indicating the number of schedules in each unit (used to link output to unitdata). Default `Nsched`

`maxtrees` maximum number of trees in one stand, default 100. This option has meaning only if there was `treeevars` option in the simulator definition.

`buffersize` Schedules are temporarily stored in linked buffers. `Buffersize` option gives the number of schedules in one buffer. It may be useful to experiment different values in large simulations. Default is 10000.

Note: output will linked to `unitdata` in the same way as subdata to data in data-function or in `linkdata` function.

E.g. the simulator js defined above can be used as follows:

```
age#0=40
vol#0=50
simulate(js)
kukuu1
grow/period     1  age=      50 vol=      75
```

```
kukuu2
grow/period     2  age=      60 vol=     100
kukuu3
grow/period     3  age=      70 vol=     125
kukuu4
grow/period     4  age=      80 vol=     150
kukuu5
grow/period     5  age=      90 vol=     175
thin/period     5  age=      90 vol=      90
clear/period    5  age=       0 vol=       0
thin/period     4  age=      80 vol=      75
kukuu5
grow/period     5  age=      90 vol=     100
clear/period    4  age=       0 vol=       0
kukuu5
grow/period     5  age=      10 vol=      25
thin/period     3  age=      70 vol=      60
kukuu4
grow/period     4  age=      80 vol=      85
kukuu5
grow/period     5  age=      90 vol=     110
thin/period     5  age=      90 vol=      51
clear/period    5  age=       0 vol=       0
thin/period     2  age=      60 vol=      45
kukuu3
grow/period     3  age=      70 vol=      70
kukuu4
grow/period     4  age=      80 vol=      95
kukuu5
grow/period     5  age=      90 vol=     120
thin/period     5  age=      90 vol=      57
clear/period    5  age=       0 vol=       0
thin/period     4  age=      80 vol=      42
kukuu5
grow/period     5  age=      90 vol=      67
clear/period    4  age=       0 vol=       0
kukuu5
grow/period     5  age=      10 vol=      25
```

Example of using the simulator with data:

```
dat=data(read->(age#0,vol#0),in->)
10,5
45,50
10,5
/

simdata=simulate(js,data->dat,unitdata->cdat)
```

Example of using the selector option (this also demonstrates a use of object list as a pointer).

```
select=trans()
if(standtype.eq.1)then
Selected=list(js)
else
Selected=list(js2)
```

```
endif
/
simsdata=simulate(js,js2,data->data,selector->select,unitdata->cdat)
```

13 Plotting figures

There is clearly a need to make graphs within **J**. On the other hand, it is not reasonably to try to include professional level graphs routines in a program like **J**. The purpose is to make **J** as efficient as possible in solving large lp-problems. Graphic windows take much memory, and it is complicated to use a large text I/O window in the same program which is using graphic windows. Some simple graphics has been organized in **J** as follows. The graph functions of **J** produce figure objects, which are automatically, or with special `show` function written into temporary working file `jfig.jfig`. There is an accompanying program `jfig` which waits for the appearance of `jfig.jfig`. When the file is ready, it reads it and plots the figure. One can then copy the figure as a bitmap into e.g. Word. When the user will click with the mouse on the figure, `jfig` will delete `jfig.jfig`. When **J** has written file `jfig.jfig`, it continues execution. But if **J** is asked to make a new figure and file `jfig.jfig` exists, then **J** is waiting the disappearance of `jfig.jfig` (i.e. clicking on the figure window) before it writes a new jfig-file.

Publication level graphics can be created using free R software. Using `r->` option in graphic functions the figure is written into a text file which can be loaded into R using source("file") command. The file can be edited to get proper legends etc. The R figure can be saved as a postscript file.

The first graphics **J** functions are:

Scatterplot: `plotyx()`

`plotyx(`*`yvar`*`,`*`xvar`*`[,data->][,mark->][,xrange->][,dx-]`
`[,yrange->][,dy->][,append->][,show->])`

Makes a scatter plot figure with the help of `jfig` program.

Output: a figure object, default `Figure`

Arguments: `y`-variable and `x`-variable

Options:

`data` data sets

`mark` character used to plot observations, default '`+`'

`xrange->(xmin,xmax)` , minimum and maximum of the x-axes , and the distance between major ticks, default, the observed minimum and maximum of x-values, and for `dx` the default is 10% of the x-range. It is possible to give only `xmin` or only `xmin` and `xmax`.

`dx` the distance between major ticks, the default is 10% of the x-range.

`yrange->(ymin,ymax)` , similar as `xrange`

`append` the figure is appended to the output figure object

`show` , `show->0` indicates that the figure is not shown (can be shown later after adding more sufigures)

Drawing a function: `draw()`

`draw(`*`func`*`->[,x->][,xrange->][,dx->][,yrange->][,dy->]`
`[,y->][,points->][,append->][,style->][,width->]`
`[,color->][,mark->][,show->][,r->])`

Draws a curve into a figure object shown with the help of `jfig`- program

Output: a figure object, default: `Figure`

Options:

`func` describes the function to be drawn, e.g. `func->(sin(x))`, transformations objects can be utlized through value function, e.g. `func->(value(ss,y))` where `ss` is transformation object and `y` variable getting value in `ss`.

If `y` is functions of x:

`xrange->(xmin,xmax[,xmin2,xmax2])` , minimum and maximum of the x-axes , and the distance between major ticks, for `dx` the default is 10% of the x-range. If `xmin2` and `xmax2` are given, the function is drawn within this range not for the complete range of x axes.

`dx` the distance between major ticks, for `dx` the default is 10% of the x-range

`x` variable which defines the x-axes for the curve.

If `x` is function of y :

`yrange->(ymin,ymax[,ymin2,ymax2])` , minimum and maximum of the y-axes. If `ymin2` and `ymax2` are given, the function is drawn within this range not for the complete range of y axes.

`dy` the distance between major ticks, for `dy` the default is 10% of the y-range.

`y` variable which defines the y-axes for the curve

`points` number of points generated, linear interpolation between the points, default 100 if `dx` or `dy` is not given, 10 points in each `dx` or `dy` section.

`append` the figure is appended to the output figure object

`style` style of the line, `0`=no line, `1`= is solid line, `2` = dashed line , `3`= dotted line, `4` = dashdot line, these values work also with `r->` option.

`width` width of the line, default is `1`, has effect only in the R-version of the figure

`color` `1`=black, `2`=red, `3`=green,`4`=blue,`5`=turquoise,`6`=purple,`7` (or greater)=yellow (the same colors apply when transporting to R)

`mark` mark put to some points on the line

`show` , `show->0` indicates that the figure is not shown (can be shown later after adding more subfigures)

`r` with no argument this implies that the figure is written to file 'jfig.r' which can be loaded into R using function call:source("jfig.r"), if the argument is given, then it defines the file name.

Note. The other line types except the solid line do not show up properly when the figure is shown with Jfig program if the number of points is large (i.e. line segments are short). With R these are displayed properly.

Drawing line through points: `drawline()`

`drawline(`*`x1[,…,xn][,y1][,…,yn]`*` [,append->][,style->] [,width->][,color->][,mark->][position->][r->] [,show->])`

Draws a polygon connecting points (`x1`,`y1`), (`x2`,`y2`) etc into a figure object shown with the help of `jfig` program. If only one point is given, then text given in `mark` – option is placed at that point.

Output: a figure object, default: `Figure`

Arguments: The x and y coordinates of the points. If there is only one argument which is a matrix object having two rows, then the first row is assumed to give the x values and the second row the y values. If there are two matrix (vector) arguments, then the first matrix gives the x-values and the second matrix gives the y-values.

Options:

`style`, style of the line, `0`=no line, `1`= is solid line, `2`= dashed line , `3` = dotted line, `4`= dashdot line

`width` width of the line, default is `1`, has effect only in the R-version of the figure

`color` `0`=black, `1`=red, `2`=green,`3`=blue,`4`=purple

`mark` mark put to the corner points on the line. If only one point given, then text to be placed at the point in position indicated by the position option.

`position` if only one point given, then the option indicated how the text given in mark is placed with respect to the point. The interoperation is:

`0` (default), text is centered
`1`, text is below
`2` text is left
`3` text is up
`4` text is right

`r` with no argument this implies that the figure is written to file 'jfig.r' which can be loaded into R using function call:source("jfig.r"), if the argument is given, then it defines the file name.

`show` , `show->0` indicates that the figure is not shown (can be shown later after adding more subfigures)

Note 1: if `style->0 a`nd there is `mark->` then only the points are shown.

Note 2: If you like to have symbolic names for colors and styles you can define these nicely by putting definitions into the startup file j.par.

Note 3: The position codes are the same as in R, and the output of text tries to imitate the R output so that one could put legends on the graph already in J and use R just to draw final figures.

Drawing class information: `drawclass()`

`drawclass(`*`matrix,x->[`*`,xrange->][,yrange->]`
`[,histogram->][,freq->][,sd->] [,se->][,dx->][,append->]`
`[,style->][,color->] [,mark->][,r->][,show->])`

Plots class information produced by `classify` function (class means standard deviations, standard errors)

Arguments:

`matrix` a matrix containing class information (produced by `classify` function)

`x` variable which defines the x-axes

Options:

`xrange->(xmin,xmax)` defines a new x-range for the figure. If `xmin=xmax=0`, then the minimum and maximum x-coordinates used in any subfigure are used. The new range will become property of the figure object.

`yrange->(ymin,ymax)` defines the y-range for the figure.

`histogram` histogram is produced

`freq` histogram is for produced for counts (default percentages)

`sd` standard errors of class means are drawn

`se` class standard deviations are drawn

`dx` the distance between major ticks, for `dx` the default is 10% of the x-range

`append` the figure is appended to the output figure object

`style` style of the line, `0`=no line, `1`= is solid line, `2` = dashed line , `3`= dotted line, `4` = dashdot line, these values work also with `r->` option `r` with no argument this implies that the figure is written to file 'jfig.r' which can be loaded into R using function call:source("jfig.r"), if the argument is given, then it defines the file name.

`color` gives the color code for the whole figure which will bypass any color codes given in subfigures. If no argument is given, the drawing is done in black. This is useful if we want to see figures in colors then we must turn everything into black and white when producing figures for publications. The color codes used in the subfigures will remain unchanged.

`mark` mark put to some points on the line

`r` with no argument this implies that the figure is written to file 'jfig.r' which can be loaded into R using function call:source("jfig.r"), if the argument is given, then it defines the file name.

`show` `show->0` indicates that the figure is not shown (can be shown later after adding more subfigures)

`show(fig[,r->][,xrange->][,yrange->][color->])`

Shows a previously made figure

Argument : a figure object to be written into filer `jfig.jfig` so that the program Jfig can then show the figure.

Options:

`xrange->(xmin,xmax)` defines a new x-range for the figure. If `xmin=xmax=0`, then the minimum and maximum x-coordinates used in any subfigure are used. The new range will become property of the figure object.

`sd` draw +- class standard deviation around class mean

`se` draw +- class standard error (sd/sqrt(n)) around class mean

`yrange->(ymin,ymax)` defines a new y-range for the figure. If `ymin=ymax=0`, then the minimum and maximum y-coordinates used in any subfigure are used. The new range will become property of the figure object.

`r` with no argument this implies that the figure is written to file 'jfig.r' which can be loaded into R using function call:source("jfig.r"), if the argument is given, then it defines the file name.

`color` gives the color code for the whole figure which will bypass any color codes given in subfigures. If no argument is given, the drawing is done in black. This is useful if we want to see figures in colors then we must turn everything into black and white when producing figures for publications. The color codes used in the subfigures will remain unchanged.

***Currently only one argument allowed, later several figures can be overlaid.

14 Stem curves, splines and volume functions

J will contain many tools for handling stem curves. Currently there are the following functions available.

14.1 *Stem splines*

```
=stemspline(h1,…,hn,d1,…,dn
  [,sort->][,print->])
```

Output: an interpolating cubic spline, designed especially for stem curves by Carl Snellman. To prevent oscillation (which can happen with splines) two knots are merged if the distance between heights is less than 8cm (this can be made an option if needed). The resulting spline is tested to see if it oscillates. For each knot interval, the value of the spline is computed at 1/3 and 2/3 point of the knot interval. If the larger of these predicted diameters is larger than 0.4 cm + largest of the endpoint diameters, a new knot is added with the diameter value equal to 0.7*the larger endpoint diameter+0.3*the smaller endpoint diameter. If the smaller of the tested diameters is smaller than the smaller endpoint diameter-0.4 cm or it is smaller than 0.4 cm a new knot is added with diameter value 0.7*the smaller endpoint diameter+0.3*the larger end point diameter.

Arguments: `h1,…,hn`, the heights of measured diameters (in m), `d1,…,dn`, the diameters (cm).

Options:

`sort` the default is that the heights are increasing, if not then `sort` option must be given

`print` If print option gets value `2` then only the problem cases are printed, if less than 2, then nothing is printed (unless an error occurs)., with value `3` or greater the knot points are printed.

The resulting spline can be utilized using `value`, `integrate` or `stempolar` functions.

=stempolar(stemspline,angle[,origo->][,err->])

Compute the diameter at polar coordinate angle (in degrees) using a `stemspline` object.

Arguments

`stemspline` a stemspline object (produced by `stemspline` function)

`angle` polar coordinate angle (in degrees)

options:

`origo` gives the baseline when computing the angle, default is `0`

`err` if the is error in obtaining the polar coordinate diameter, then `err` option defines transformation set which is called before returning from `stempolar` function.

=laasvol(species,dbh[,d6][,h])

Volume functions of Laasasenaho (1982).

Arguments

`species` `1` =Scots pine, `2`= Norway spruce, `3` and `4` = birch, `9`=larch (not available when `dbh` is the only measured dimension)

`dbh` and `d6` (diameter at 6m) are in cm, height (`h`) is in m and result is in litres.

=laaspoly(species,dbh [,d6],h)

Polynomial stem curve of Laasasenaho (1982).

Arguments

`species`: `1` =Scots pine, `2`= Norway spruce, `3` = birch, `5`=aspen, `6`=alder, and `9`=larch (not available when `dbh` is the only measured dimension)

`dbh` and `d6` (diameter at 6m) are in cm, height (`h`) is in m

The curve can then be used using `value` function, e.g.

```
curve=laaspoly(species,dbh,h)
d6=value(curve,6) ! diameter at 6 m.
```

Functions providing volume integrals of the curves and height of given diameter will be added on the request.

=tautspline(x1,…,xn,y1,…,yn
[,par->][,sort->][,print->])

Output: an interpolating cubic spline, which is more robust than an ordinary cubic spline. To prevent oscillation (which can happen with splines) the function adds automatically additional knots where needed.

Arguments: `x1,…,xn`, the x values, `d1,…,dn`, the y values. There must be at least 3 knot point, i.e. 6 arguments.

Options:

`par` gives the parameter determining the smoothness of the curve. The default is zero, which produces ordinary cubic spline. A typical value may 2.5. Larger values mean that the spline is more closely linear between knot points.

`sort` the default is that the x's are increasing, if not then sort option must be given

`print` if print option is given, the knot points are printed (after possible sorting).

The resulting spline can be utilized using `value`

The taut spline algorithm is taken from: de Boor (1978).

15 Utility functions

15.1 *Working directory*

The current working directory can be seen or changed.

showdir()

Prints the current working directory

setdir(charval)

Sets the current working directory. The argument is a character variable or character constant.

15.2 *Timing functions*

There are two timing functions which can be used to measure the computation time. There are two versions of each, without argument, and with an argument

`secnds()`

purpose: first call gives the elapsed time since midnight in seconds

`secnds(t)`

purpose: gives the elapsed time since midnight -`t`.

`cpu()`

purpose: first call gives the cpu time since starting the program in seconds

`cpu(t)`

purpose: gives the total cpu time -`t`

15.3 *List functions.*

`=list(obj1,…,objn[,mask->])`

Defines an object list

Output: a list object

Arguments: 0-n objects (need not exist before)

Options:

`mask` defines which objects are picked from the argument list, value `0` indicates that the object is dropped, positive value indicates how many variables are taken, negative value how many object are dropped (thus 0 is equivalent to -1). `mask` option is useful for creating sublists of long lists.

Note 1: If an argument does not exist beforehand, it is first created as a real variable.

Note 2: The same object may appear several times in the list. (see `merge`)

Note 3: There may be zero arguments, which result in an empty list (see example below)

Examples:

```
all=list()  ! empty list
sub=list()
;do(i,1,nper
period#"i"=list(ba#"i",vol#"i",age#"i",harv#"i")
sub#"i"=list(@period#"i",mask->(-2,1,-1))
all=list(@all,@period#"i") ! note that all is on both sides
sub=list(@sub,@sub#"i")
;end do
```

=merge(obj1,…,objn)

Defines a list dropping multiple references to the same object

Arguments: objects or lists

If an argument is a list, then it is not necessary to expand it using `@`-operator, even if it can be expanded and the result is the same.

Note: arguments of merge need to be known beforehand (unlike in `list` function).

=difference(list1,list2)

Defines a list dropping from list `list1` all objects found in list `list2`

Arguments: lists

index(object,list[,any->])

Gets the index of a variable in a list, usually in the keep list of a data set (the column number in the data matrix)

Arguments:

`object` object name

`list` : a list object

Options:

`any` accept also that variable is not in list (output=0) without error condition

Note 1: if the second variable is not a list, error occurs. If the variable is not in the list, index gets value 0.A error condition is obtained if `any` option is not present

Note 2. It is faster to get the value of the index within input programming or outside the transformation set so that it must not be searched repeatedly.

Note 3. See chapter 9.5.2 for an example how to utilize index function for an example how to utilize `index` function.

len(list[,any->])

Returns the number of elements in the list

Argument: a list object

Option

`any len` returns value-1 if argument is not legal object for `len` (without `any->` an error occurs)

Note 1: `len` works also for text objects, returning the number of characters in a text object, and for a matrix it returns the number of elements in the matrix.

Note 2: The value of a specific list element variable can be obtained using value function

15.4 *Getting value from an object: value(object,xvalue)*

If a **J** function generates an object containing parameters for a special function then the `value` function can used to generate values from the object. The general form of the value function is

`=value(object,xvalue[options])`

where `xvalue` is the value used as the argument for the object which can be used as a function. For most cases the object can be used also otherwise. The output is a single numeric value.

If the first argument is a list and there is option `index->` then the object is picked from the list.

There are the following special cases.

15.4.1 Interpolating a regular matrix: value(matrix,x)

`=value(matrix,xvalue[,row->])`

Interpolates linearly rows of matrices.

Arguments:

`matrix` a matrix

`xvalue` value for which a row must be interpolated

Options:

`row` Gives the y-row for interpolation if there are more than two rows in the matrix and only one row needs to be interpolated.

#The first row of the matrix defines the knot points. It is assumed that knot points are in increasing order. If there are only two rows in the matrix, then the second row defines the values at the knot points. If there are more than two rows then a vector is generated by interpolating each row from 2 to `nrows(matrix)`, unless there is `row` option

Note: Also extrapolation is allowed, i.e. the argument can be smaller than the first knot point or larger than the last knot point.

Example:

```
sit>a=matrix(3,4,in->)
10, 20, 30, 40
15, 16, 18, 20
20, 40, 60, 80
/
sit>v=value(a,35)
sit>print(v)
 v is matrix(           2 ,           1  )
   19.00000
   70.00000
sit>c=value(a,15,row->2)
sit>print(c)
c=   15.50000
```

*** Later quadratic and cubic interpolation, as well of interpolating two dimensional matrices will be available.

15.4.2 Interpolating a classify-matrix: value(cl_matrix,xvalue)

`=value(cl_matrix,xvalue)`

Interpolating a matrix produced by `classify` function.

Output: a real variable

Arguments:

`cl_matrix` a matrix produced by `classify` function

`xvalue` value of the x variable used for computing class means.

15.4.3 Using a spline: value(spline,xvalue)

`=value(spline,xvalue)`

Gets values from a smoothing spline or a stem curve spline

Arguments:

`spline` a spline generated by `smooth` or `stemspline` function

`xvalue` argument of the spline.

**later other types of splines will be available

15.4.4 Getting values from a transformation set: value(tr_set,xvalue)

`=value(tr_set,xalue[,arg->][,result->])`

Arguments:

`tr_set` a transformation set generated by `trans` function

`xvalue` argument which is put into the argument variable (default `Arg`) of the transformation set

Options

`arg` variable used as the argument variable, it bypasses the argument variable associated with the transformation set

`result` defines the variable whose value is the result of the function, default is the result variable associated with the transformation set (and default for that variable is `Result`)

Note 1: The original value of the argument variable is remains unchanged.

Example:

```
s=trans(input->,arg->x,result->h)
h=sin(x+z+1)
/
```

Then `y=value(s,3)` is equivalent to

```
xold=x
x=3
call(s)
y=h
x=xold
```

This form of the `value` function is useful e.g. in `filter`, `reject` or `func` options or when transformations are needed to get numeric values into options.

Note 2: A transformation set can be used also as a function using result function, if the transformation set does not use a argument whose value need to bypassed simultaneously.

15.4.5 Gettting value of a list variable

`=value(list,index)`

Arguments:

`list` a variable list generated by `list` function

`index` index of the variable

Example:

```
alist=list(a,b,c)
b=6.7
Then value(alist,2) returns 6.7
```

*** Later there will be more function objects accessed using `value` function

15.5 *Inverse function: valuex(object,yvalue)*

An inverse function gives the value of the x-variable for which the function obtains a given value. Currently the only inverse function implemented is:

15.5.1 Height of diameter using stemspline: valuex(stempline,diameter)

=**`valuex(stemspline,diameter)`**

Output: height (in m) of a given diameter

Arguments:

`stemspline`	stemspline object generated with `stemspline`
`diameter`	diameter (in cm) for which the height is obtained

15.6 *Interpolating points: interpolate()*

`interpolate(x0,x1[,x2],y0,y1[,y2],x]`

If arguments `x2` and `y2` are given then computes the value of the quadratic function at value `x` going through the three points, otherwise computes the value of the linear function at value `x` going through the two points.

Arguments: numeric values

Note. The argument `x` need not be within the interval of given x values (thus the function also extrapolates).

15.7 *Integrating a function*

*** Later there will be several forms of `integrate` function. Currently the only one is:

15.7.1 Integrating stem curve to get stem volumes

=integrate(stem_spline,h1,h2)

Output: volume (dm^3) of stem segment

Arguments:

`stem_spline` a stem spine generated with `stemspline` function

`h1` lower limit of the stem segment, in metres

`h2` upper limit of the segment, in metres

Note 1: The upper limit must be smaller or equal to the to the last height argument given in `stemspline`.

Note 2: This form of the integrate function does not integrate the value of the stem curve but actually $\pi*0.25*value(stem\_spline,h)^2$ and the result is then divided by 10000 to get the result in dm^3 (in stem splines both height and diameter are in cm)

15.8 *Bit functions*

In some applications we may need several indicator variables to indicate if some property is present. In large data sets would be waste of space to store a separate variable for each indicator. J has special bit functions for packing several indicators in the same variable. One variable can store 32 indicators, and it is also possible to store more indicators using variable lists. When bits are stored into variables, then these variables can be included in data sets. There is also a special bit matrix object which is created with `bitmatrix` function. The bit patterns can be read from files, or set by `setvalue` function. Bits in a bit matrix can be obtained with `value` function and the matrix can be printed with `print` function.. There are following bit functions available.

setbits(ind, bit1,…,bitn)

Sets one or more bits on.

Arguments:

`ind` a real variable or a list of real variables (do not expand by `@`))

Note: argument `ind` is used both as input and output

`bit1,…,bitn` bit positions to be put on

The bit positions of a real variable are numbered 1,…,32, the bit positions of a variable list are numbered from 1 to 32*(number of variables in the list).

`clearbits(ind,bit1,…,bitn)`

Sets one or more bits off.

Arguments:

`ind` a real variable or a list of real variables (do not expand by `@`))

Note: argument `ind` is used both as input and output

`bit1,…,bitn` bit positions to be put off

The bit positions of a real variable are numbered 1,…,32, the bit positions of a variable list are numbered from 1 to 32*(number of variables in the list).

Note: giving value zero to a real variable clears all bits.

=**`getbit(ind,bit)`**

Gets the value of a bit position.

Arguments:

`ind`: a real variable or a list of real variables (do not expand by @)

`bit` bit position to read

function: If the bit is on, then the value of the `getbit` function is 1 (*True)*, otherwise zero *(False*). the bit positions of a real variable are numbered 1,…,32, the bit positions of a variable list are numbered from 1 to 32*(number of variables in the list).

Note: The `getbit` function can be directly used in logical statements, e.g., `if(getbit(ind,8))then`

`=getbitch(ind[,from][,to])`

Gets indicators into a text line, '`1`' indicating a bit which is on, and '`0`' a bit which is off.

Output: text object

Argument

`ind` a real variable or variable list

If there are no other arguments, then all 32 bit positions are put into the output, if there is on additional argument, then this indicates the number of bits read, and if there are two additional arguments the first indicates the starting position and the second indicates the last position.

Example:

```
sit>a=0
sit>setbits(a,2,5,7)
sit>v=getbitch(a)
sit>print(v)
 v is text object:
 01001010000000000000000000000000
sit>
```

=bitmatrix (nrows[,colmax][,in->][,colmin->][,func->])

Output: a bit matrix object.

Arguments

`nrows` number of rows in the bit matrix, value `-1` indicates that this is indicated by the number of records in the file given in `in->` option.

`colmax` upper limit of column index, default=`1`, value `-1`, indicates that this is obtained from the column indices read from the file given in `in->` option.

Options

`in` indicates that the bit pattern is read from the input paragraph or from the file. If `in->` option is not given then all bits are initially zeros until changed with `setvalue` function.

`colmin` gives the lower limit of the column index

`func` when read the column indices, they can be transformed first using the function given in `func` option. The column index read from the data is put into the default argument variable '`x#`'.

Examples:

```
a=bitmatrix(3,4,in->)
2,1,4
0
1,1
/
```

The first number tells how many bits are set for the row, then there are the column indices.

```
a=bitmatrix(3,4,in->,colmin->0) ! now 0 is legal column index
2,0,4
0
1,1
/
```

Note: If the matrix would be very sparse and large, then it is possible to use `index` function to pack the matrix and access also the matrix. I give an example when this packing is needed.

=value(bitmatrixobj,row[,col][,any->])

The value of a bit in a bit matrix can be obtained using `value` function

Output: real value 1 or 0

Arguments

`bitmatrixobj` an object created by `bitmatrix` function

`row` row index

`col` column index

Option:

`any` if `row` or `col` is out of range (`row`<1 or `row`>`nrows(bitmatrixobj)`, or `col`<colmin, or `col`>colmax), the default is that an error condition occurs, but `any->` option indicates that the value zero (False) is returned. This option is handy when bitmatrix e.g. describes domains, then it is not necessary that each stand belongs to some domain.

setvalue(bitmatrixobj,row[,col],value)

Sets `bitmatrixobj`(`row`,`col`)=`value`
All nozero values indicate that the bit is set into one.

=nrows(bitmatrixobj)

returns the number of rows in a bitmatrix

=ncols(bitmatrixobj)

returns the number of column in a bitmatrix

=closures(bitmatrixobj)

To get neighborhoods indicated by a bitmatrix

Output: a bitmatrix

Argument: A symmetric square (1:n,1:n) bitmatrix where ith row indicates all the neighbors of ith point.

#All the neighbors of a given point are not necessarily neighbors if they are located at opposite sides of a point. Closures function will generate all such neighborhoods where all points are neighbors.

Example:

If points are located
1 2
3 4
5 6
Then this can be first described

```
ne=bitmatrix(6,6,in->)
4,1,2,3,4
4, 2,1,3,4
6 ,3,1,2,4,5,6
6, 4,1,2,3,5,6
4 ,5,3,4,6
4,6,3,4,5
/
```

Note that the 'focus' point is given as first in each line, but the neighbors can be in any order. Then commands

```
ne2=closures(ne)
print(ne2)
```

will produce output

```
ne2 is            2  x (           1 :           6 ) bitmatrix:
111100
001111
```

**The algorithm in closures is not well tested

15.9 *Defining crossed variables: properties( )*

A data object is describing several subjects by defining for each subject a set of variables associated with them. If there are a few named subjects then it may be useful to have separate subject specific variables (constants) which define properties of the subjects. These kinds of variables can be defined with properties function.

`properties(`*`var1,…,varn[,print->]`*`)`

```
subject1, val1,…,valn
…
/
```

Defines subject specific constants.

Arguments: generic names of variables

Options:

`print` are the values printed (to check that they have been read correctly)

Input paragraph following `properties` function has a line for each subject, where first is the name of the subject, and then values for all argument variables. The properties function then defines variables having fist the generic variable name, then '`%`' and then the subject name.

Example:

```
properties(capacity,xkoor,ykoor)
rauma, 100, 64,78
pori, 30, 67,89
/
```

Defines variables `capacity%rauma,xkoor%rauma,ykoor%rauma,capacity%pori`, etc.

15.10 *Storing values of variables*

`=store(var1,…,varn)`

Stores the values of variables.

Output: a storage object

Note: A variable list may be again nice when defining the arguments.

`load(storage)`

Loads back the values of variables.

Argument: a storage object created by `store`.

*** Now only values of real variables can be stored. If there is need to store general objects, it is quite easy to make `store` capable of handling these.

15.11 *Saving object into files*

`save(filename,obj1,…,objn)`

Arguments:

`filename` character variable or character constant, the name of the file

`obj1,…,objn` named objects

Note 1: There can be several save commands which all save into the same file, the file remains open after each save command.

Note 2. The file can be closed by using `close`(filename)

*** Currently the only compound objects (i.e. objects having links to other objects, e.g. data set or transformation set) which can be saved are:
-list of real variables
-regression object

`unsave(filename)`

Argument:

`filename` character variable or character constant, the name of the file created by `save()`.

Output will be the list of all objects loaded (default for the output is `Result` as usually)

Note 1: All objects saved in the file are loaded, i.e., without taking into account if they are saved with one or more `save` functions. If an object to be unsaved already exists, then it will be replaced. The number of replaced objects will be printed.

Note2: If `unsave` is tested using it in the same run as the save functions, the file must be closed first with `close` function.

16 Error debugging and handling

16.1 *Errors detected by J*

If **J** detects an error then the following information is provided:

- **J** prints the current and previous input line as generated by the input programming.
- **J** will close all open include files and it tells how many lines it has read from these files. Usually, but not always, the last line read has caused the error. E.g. `;do` loops are read first before starting to interpret lines within the loops. So if the error is within the loop, then last line read is later than the error line.

If an error occurs when computing interpreted transformations within a transformation set or a simulator, and saving of source code is not denied by `source->0`, then **J** also prints the source code causing the error. The source code around the line can be seen by using print function (possibly with row option) or by writing the whole source code into a file using `write` function.

After an error the control return to `sit>` level.

If the limit for the maximum number of named objects is encountered, then **J** stops. See *Set up of J* how to proceed.

***J** is in principle protected against trying to put data above the allocated dimension limits, but there will certainly be problems in the first versions.

16.2 *Tracing variables*

It is possible, using special tracing functions, to track changes of variables, define minimum and maximum values for variables, and test if all required variables get values within any phase of a J run.

`;trace(obj1,…,objn [,min->][,max->][,out->][,level->] [,errexit->])`

Start generating tracing information for objects.

Arguments: **J** objects (usually real variables)

Options:

`min` gives lower bound for a variable. If a variable gets smaller value then the value and the source line are written, and if `errexit` option is present then an error condition occurs.

`max` gives upper bound for a variable. If a variable gets smaller value then the value and the source line are written, and if `errexit` option is present then an error condition occurs.

`out` gives a name for an trace set object which will be generated and which can be used in `tracetest` function to test that all objects `obj1,…,objn` have got values in a section of a **J** run.

`level` defines the level or tracking, possible values are
`0` means that tracing code is generated but it is now deactivated, it can be activated later with `trace` function.
`1` indicates that the changes of objects are counted but not automatically written unless ranges given in `min` or `max` option is violated.
`2` indicates that all changes are written.
If `min`, `max` or `out` option is present then default is 1 otherwise 2.

`errexit` if value smaller than `min` or value greater than `max` occurs, then an error occurs.

Note. When an object is used as an argument of `;trace` function, then each time when the object is an output of any function or arithmetic statement, then the transformation interpreter adds a call to tracing function. What exactly happens in this tracing function is dependent on the current values of tracing parameters which are initially set by `;trace` function but which can be later modified, also within an transformation set, using `trace` function.

`;trace` function creates following objects:

`Tracevars` = a cumulative object list of all objects used in all `;trace` functions. Even if generation of tracing code is stopped for an object, the object remains in the same place in `Tracevars` list.
`Tracestatus` = row vector corresponding to `Tracevars` list indicating if tracing code is generated (value 1) or not (value 0).
`Tracelevel` = row vector telling the current value of tracing level.
`Tracecount` = row vector showing counts of changes, used e.g. by `tracetest` function
`Traceminstatus`= row vector indicating if minima are given
`Tracemin` = contains the given minimum values
`Tracemaxtatus`= row vector indicating if maxima are given
`Tracemax` = contains the given maximum values

Note. Current `Tracevars` list and current values of the trace parameter vectors can be seen by printing. It is also possible to change the parameter values directly, but it is recommended that `;traceoff` and trace functions are used to change the values.

`;traceoff(obj1,…,objn)`

Stop generating tracing code for objects.

Note: The objects remain in the `Tracevars` list but the values in `Tracestatus` vector are changed into zero.

`trace(obj1,…,objn [,min->],[,max->][,level->]`
`[,errexit->])`

Change the tracing parameters for objects. The meaning of arguments and options is like in `;trace` function. The differences are:
1. Arguments of trace function must be previous arguments of `;trace` function.
2. Trace set can be defined only in `;trace` function (using `out->` option)
3. `trace` function can be an function within a transformation set, but `;trace` function is just done directly and it will not remain part of the transformations set. With `trace` function one can program dynamic tracing and debugging strategies.

`tracetest(traceset)`

Test that all objects in a trace set object have been changed since last call of the `tracetest` function ort from the beginning. If not all objects have been changed, an error occurs.

Argument: An trace set object generated by the out option of the `;trace` function.

Note: `tracetest` function is useful mostly in two different cases:
1. if the values of some variables are determined in complicated control structures which may contain 'holes', i.e. with some combinations of input variable values an intended output variable does not get any value at all.

2. When defining a simulator it can very easily happen that all intended output variables do not get values in all nodes. See `simulator` function how to utilize `tracetest` function.

An example of tracing functions

Define a transformation set `tr` as follows:

```
tr=trans()
a=1
;trace(a,b) !start generating tracing code
a=7
b=2
;traceoff(a) !stop generating tracing code for a
a=4
b=3
/
```

Executing `tr` we get:

```
sit>call(tr)
a got value    7.000000      in tr at line            2  :
a=7
b got value    2.000000      in tr at line            3  :
b=2
b got value    3.000000      in tr at line            5  :
b=3
```

We can drop tracing of `a` even if the tracing code remains in transformation set `tr`.

```
trace(a,level->0)
call(tr)
b got value    2.000000      in tr at line            3  :
b=2
b got value    3.000000      in tr at line            5  :
b=3
```

Start checking that `b` is at least `3`.

```
trace(b,min->4,errexit->)
call(tr)
b got value    3.000000      in tr at line            5  :
b=3
*err* transformation set=tr, *source= source%tr
error on source row            5:
b=3
```

An example of using `tracetest`. Define first trace set `outvars` and transformation set `tr2`:

```
;trace(x1,x2,out->outvars)  ! define trace set outvars
tr2=trans()
if(a.gt.2)then
x1=5
x2=4
```

```
elseif(a.lt.3)
x2=7
endif
/
```

Define `a` and execute transformations:

```
 a=7
 call(tr2)  ! now both x1 and x2 get new values
 tracetest(outvars) ! nothing happens
```

But call then transformations using `a=2`:

```
 a=2
 call(tr2) ! now x1 is not updated
 tracetest(outvars) ! comment helps to find the place if error occurs
 *tracecount for x1 is zero
 *err* transformation set=$Cursor$
 **input line:tracetest(outvars) ! comment helps to find the place if
error occurs
 *closing inc-file 'trace.txt'
   after reading            34  lines from           34
```

See simulator for another example of `tracetest`.

16.3 *J does not work correctly*

Several functions have a `debug->` option even if it is not described in the above manual. With this option **J** writes extra information about how it proceeds. User may try this option before consulting J. Lappi.

Error messages starting with '`*j*`' indicate programming errors which should be reported to the author.

If there is in transformations or at the command level function

`debug()`

then special debugging mode is entered. Debug information can be understood only by the author. Debugging can be made conditional by having argument

```
debug(t.gt.0)
```

This is actually equivalent to

```
if(t.gt.0)debug()
Debugging is put off by
debug(0)
```

17 Acknowledgements

Function `jlp` is using linear algebra subroutines contained in quadratic programming software Bqpd made by Prof. R. Fletcher, University of Dundee. Even Bergseng

started to use the linear programming modules on a premature stage and Even has detected a large number of errors. Jarmo Saarikko and Olavi Kurttio made the web distribution of the software. Jarmo Saarikko also cleaned up considerably the first version of this manual.

18 References

Dantzig, G.B and VanSlyke, R.M. 1967. Generalized upper bounding techniques. Journal of Computer and System Sciences. 1:213-226.

de Boor, C 1978. A practical guide to splines. Applied mathematical sciences. Vol. 27. Springer-Verlag, New York, 392 p.

Dykstra, D. P. 1984. Mathematical programming for natural resource management. McGraw-Hill. New York. 318 p.

Fletcher, R. 1996. Dense factors of Sparse matrices. Dundee Numerical Analysis Report NA/170

Kilkki, P. 1987. Timber management planning. 2nd edition. Silva Carelica 5. University of Joensuu. Faculty of Forestry. 160 p.

Lappi, J. 1992. JLP - A linear programming package for management planning. Finnish Forest Research Institute Research papers. 414, 134 p.

Lappi, J. 2006. A multivariate, nonparameteric stem-curve prediction method. Can. J. For. Res. 36:1017-1027

Lappi, J. 2006. A multivariate, nonparameteric stem-curve prediction method. Can. J. For. Res. 36:1017-1027

Lappi, J & Lempinen, R. 2013. A linear programming algorithm and software for forest level planning problems including factories. Manuscript submitted to Scandinavian Journal of Forest Research

R Development Core Team, 2004. R: A language and environment for statistical computing. ISBN 3-900051-00-3. http://www.R-project.org

Steuer, R.E. 1986. Multiple criteria optimization: Theory, computation, and application. John Wiley. New York. 546 p.

19 Index

In the following index **J** functions and statement are in lower case, other entries are in the upper case..