Select the gray area and click the right button
– change photo
– from file

Notice! The image can only be scaled from the top corners. The placing of the bottom edge cannot be changed.

Insert or change photo here

[Natural resources and bioeconomy studies XX/20XX]

J User's Guide

J Software version 3.0

Juha Lappi and Reetta Lempinen

Kommentoinut [LR(1]: Yleistä:
- Teksti on nyt siirretty tähän pohjaan, mutta en ole sen jälkeen lukenut tätä läpi. Virheitä voi siis löytyä.

- Kiitokset avaamisprosessiin osallistumisesta jätin omavaltaisesti pois, koska en usko Luken niitä hyväksyvän julkaisuunsa

- Ehdotan tulevaan kehitykseen viittaavien kohtien siirtämistä erilliseen dokumenttiin, joka laitetaan githubiin ("repoon").

- Minulla olevassa koodiversiossa data()-funktiossa ei toiminut useamman tiedoston lukeminen, tästä pitänee laittaa huomautus (Changes from previous versions?, data()-funktion esittely) tai muuttaa in-> option kuvausta.

- User's Guide to own J3.0 packages -dokumentti pitää kanssa katsoa läpi, mutta sitä ei onneksi tarvitse vääntää tähän raportti pohjaan. Johonkin pitänee laittaa maininta mistä dokumentti löytyy (githubista)

Puuttuvaa:
- Dokumentin sisäiset linkitykset (näitä olen merkannut keltaisella korostuksella)

- Hakemisto & hakemisto-tagit

- logistic()-funktio

- Pitäisikö kaikista funktioista olla johdonmukaisesti output, arguments ja options -kuvaukset? Jonkin verran niitä lisäilin, mutta en kaikkialle (tarmo loppui)

muotoili: englanti (Yhdysvallat)

muotoili: englanti (Yhdysvallat)

Luke
NATURAL RESOURCES
INSTITUTE FINLAND

[Natural resources and bioeconomy studies XX/20XX]

Report title to be written here

Report specifier (if any)

Firstname Lastname, Firstname Lastname,
Firstname Lastname and Firstname Lastname

Natural Resources Institute Finland, Helsinki 20XX

muotoili: englanti (Yhdysvallat)

Partner logos (if any)

Recommended citation:

Author, X., Author, X. & Author, X. 20XX. Title of publication : Report specifier (if any). Natural resources and bioeconomy studies xx/2020. Natural Resources Institute Finland. Helsinki. xx p.

Recommended citation for specific articles:

Author, X., Author, X. & Author, X. 20XX. Article title in publication: Author, X., Author, X. & Author, X. 20XX. Title of publication : Report specifier (if any). Natural resources and bioeconomy studies xx/2020. Natural Resources Institute Finland. Helsinki. p. xx–xx.

Kommentoinut [LR(2]: Ohjeen mukaan julkaisutoimitus täyttää tämän ja seuraavan sivun

Main author, ORCID ID, https://orcid.org/

ISBN: xxx-xxx-xxx-xxx-x (Print)
ISBN: xxx-xxx-xxx-xxx-x (Online)
ISSN 2342-7647 (Print)
ISSN 2342-7639 (Online)
URN: http://urn.fi/URN:ISBN:xxx-xxx-xxx-xxx-x

Authors: Firstname Lastname and Firstname Lastname
Publisher: Natural Resources Institute Finland (Luke), Helsinki 2016
Year of publication: 20XX
Cover photo:
Printing house and: publishing sales: Juvenes Print, http://luke.juvenesprint.fi

Summary (and/or abstract)

Juha Lappi[1)] and Reetta Lempinen[2)]

[1)]Herralantie 10 B 14, 77600 Suonenjoki, Finland, juha.lappi.sjk@gmail.com

[2)]Natural Resources Institute (Luke), 80100 Joensuu, Finland. reetta.lempinen@luke.fi

Please write the first paragraph here.

The J User's Guide is provided without warranty of any kind. It may include inaccuracies or errors. The authors may make improvements and/or changes in the products at any time. These changes will be incorporated in the new editions of the J User's Guide.

The distribution versions of J software may deviate in some details from the general documentation presented in J User's Guide.

The names of companies and their products appearing in the J User's Guide are trademarks or registered trademarks of their respective holders.

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

muotoili: suomi

muotoili: suomi

Kentän koodi muuttunut

muotoili: suomi

Kommentoinut [LR(3]: Mitäs tähän laitteaisiin?

Kommentoinut [LR(4]: Onko näiden paikka tässä vai missä?

Contens

Preface

The most important change in the new J3.0 is that the software is made open source. Initially I made the software so that only I could understand the structure and the code. When Reetta Lempinen started to work with me in version 2.0 incorporating factory problems, I learned what are the weakest points in the software if it should be maintained by a group of people. But as I was always able to tell Reetta how the software worked, only very slight improvements in the documentation etc. were made.

When I realized that my retirement age is approaching, I started to worry how the software could be maintained after my retirement. In 2014 when we were looking at how J could be utilized in Me-la software, I encountered old JLP concept 'own function', which was the method for allowing users of JLP to add own (arithmetic) functions if they had access to the source code. Adding these two things together I realized how J should be restructured so that it could be better maintained and how different users could add their own J functions, object types and options.

I was then convinced that the best way to utilize these improvements would be to make the soft-ware open source. This would also allow me to continue my work with J after my retirement. So I started to make necessary changes that different users could start to maintain and develop J source code. This meant changes in the file structure and data structures and additional documentation. By changing the data structures, J also became significantly faster (main reason was that I made allocatable pointers were changed into allocatable arrays). I made also a precompiler which allows the users access global module ~~global~~ variables without knowing where they are. To separate the global variables from user's own variables, they are separated by j_ -prefix.

At least in this initial phase of the open J project, we did not try to comment all existing J functions. The purpose is to give the user enough information that she/he can make own functions, object types and options and access old object types and options. Old J functions do not show good examples how to make J functions, as I made just for this open J project general purpose utility functions for starting a function and accessing options, and these are not utilized in the old code. Old J functions utilize directly the information of how the interpreted J code is organized. 'User's guide to own J3.0 packages' describes how the user can develop and maintain J code.

The previous version utilized some subroutines of IMSL package and from the 'Numerical Recipes in Fortran' book. These are now replaced mainly by open access routines from Lapack and Ranlib packages from the Netlib library. Roger Fletcher allowed to distribute his linear programming sub-routines in the open J package (see the Conditions of the use for the conditions).??

The users are invited to develop the new open J software. This first version could be better documented, but I think that it is according to the open source ideas that software can be opened even if it is not so completely clean. I try to bear the shame for the poor documentation.

Suonenjoki. dd mm 2021

Juha Lappi

Changes from previous versions

Changes from version 2.1 to version 3.0

See Preface to version 3.0 for the most important general changes in the structure and distribution of the software. The file 'User's guide to own J3.0 packages' describes the structure of the code. This section just lists some details of the changes and additions.

Owing to unclear license conditions, the smoothing splines have been removed from the basic distribution. The user can generate a version of J including the smoothing splines using the procedure described in 'J3.0 Setup Guide'.

The naming of subobjects and associated objects has been changed. Earlier the subobjects and associated objects were named e.g. as `rows%problem`. Now the order has been changed into `problem%rows`. The current practice follows standard way to name subobjects, and helps to understand the code related to J-objects. Sorry for the inconvenience.

If a line ends with '>' then the next line is continuation line and the '>' character will also be part of the total input line. Thus now '>' works similarly as other continuation marks, and a line can also end with the option mark '->'.

Functions `ask()` and `askc()` can be used to ask only one real or character value, respectively.

`jlp()` function has new options (`refac->`, `warm->`, `slow->`, `finterval->`, `fastrounds->`, `fastpercent->` and `swap->`). If the problem does not contain z –variables, the integer approximation is also printed, i.e., the solution where in each divided unit the schedule with largest weight is applied. If the algorithm encounters corrupted problem vectors due to rounding errors, a recovery procedure is applied instead of stopping the solution. The capacity constraints with zero capacity are treated in a special way which makes the algorithm more stable. The period of each constraint can now be given in problem definition. If the periods are given, then the optimization is utilizing the tree structure of schedules. Unfortunately, this seems not to improve solution times e.g. with five periods, but it may be useful with large number of periods.

The variable names for variables stored in data file can be given at the beginning of the file using read option without arguments. The `readfirst->` option in data function has increased capabilities.

The way the user can see objects when a part of the name has been specified (using `?` to specify unknown part of the object name) has been changed.

An include file name can be given at the command line used to start J.

The default for arguments of `stat()` function is all variables of the data set plus output variables of `trans->` transformations. The means, standard deviations etc can be put into vectors using output for the function. Standard deviations of the means can also produces as the output.

The `read()` function has a new option `eof->` transmitting if end of file is reached. The write function can write a whole matrix. Note that `read()` function can read also whole matrices.

The " " sequences can contain format specifier. If `write()` function has only two arguments and the second argument is character constant or variable, the character is written to the file

indicated by the first argument. When this is combined with the previously mentioned " " sequence format addition, this will substantially reduce the need to use `$Buffer` when text and numbers need to be put on the same output line. See jex.txt, Chapter 3 Command input and output and Chapter 8 IO-functions for examples.

In figures the `mark->` option can now refer also to numeric value and not only character constant or character variable. In the `drawclass()` function there is a new option `area->` which makes it easy to overlay histogram and density function.

`Write()`–function can now write matrices and vectors.

The `;incl()` and `read()` functions have new option `wait->` by which one can make client-server applications. See chapter J as a server.

New functions: `logistic()`, `ranbin()` (binomial random variable), `rannegbin()` (negative binomial variable), `ranpoi()` (random Poisson variable), `select()` (random selection of individuals), cumulative distribution function of the chi-square distribution (`chi2->` option in `cdf()` function), `qr()` (qr decomposition of a matrix, useful for detecting linear dependencies), `eigen()` (computes eigenvectors and eigenvalues), `varcomp()` (computing variance and covariance components), `help()` (prints online help from jhelp.txt or from file defined by the user). Function `thisfile()` returns the name of the current include file. See file jex.txt for an application. Function `nonlin()` can do nonlinear regression. `xkf()` function has now new option `form->` and the number of records written is given as the output. New function `envelope()` computes the convex hull of points (this may be useful when illustrating supply areas of factories, see jlp.txt). The function `exist()` has a new option `object->` which allows to inquire if an object with a given name exists. New functions for interpolation: `plane()` computes the equation of plane through three points, and computes the z-value of any (x,y)-point. Function `bilin()` uses four (x,y)-points to compute bilinear interpolation for any (x,y)-point. When writing a matrix with `write()` -function, the range of rows can be given with `rows->` option. When drawing lines through points using `drawline()` -function and one matrix arguments or two vector arguments, the number of lines can be limited using `maxlines->` option.

In linear programming problems with very large number schedules there may be shortage of virtual memory. In J3.0 there are two features which allow to double the size of schedules data for given amount of virtual memory. See `nobs->` option of `data()` function and `swap->` option of `jlp()` function.

There is now special way to handle warnings. Search for 'warnings'.

Function `example()` shows different things for the programmers of the own packages.

Changes from version 2.0 to version 2.1.

Domains can be defined also for constraints in optimization problems including factory variables.

Function `jlp()` has new options (`zmatrix->`, `max->`, `min->`, `rhs->`, `rhs2->`) for the definition of large problems including only z-variables (ordinary linear programming problems). There are also several new objects for accessing the solution of a problem. Some naming conventions of generated objects have been changed. See Chapter 11.10 Objects for the JLP solution.

In function `xkf()` the order of output fields has changed.

The lines of a text object can be accessed, see Chapter 3.2.8.

Function `delete()` can now be used to delete files in addition to deleting objects.

Function `print()` has new option `file->` for redirecting output into a file.

Function `print()` has new option `form->` for printing a matrix .

If `amat` is a matrix, then all elements can be se with simply assignment statement, e.g. `amat=0`. Previously such assignment statement made `amat` a real variable and substituted the value to this variable.

New matrix function: `elementprod()`.

Object names can contain special symbols (e.g. `+-*/()`) if they are closed within '`[`' and and '`]`'.

When making a data object from a file, `nobs->` option can be used to limit the number of records read, and new automatic variables `Record` and `Subrecord` (if option `subread->` is present) get the number of input record which can be used when rejecting observations.

The variable `Result` (which is the output variable if no output is given) is not put into the list of output variables of the transformation. This has effect e.g. in the simulator.

Continuation line cannot start with '`*`' or '`!`'

The documentation is improved. Several bugs are corrected, and testing of user made errors is improved, so that user errors do not cause crash so easily.

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.

1. 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 has superseded the previous linear programming software JLP (Lappi 1992).

Most users are interested in applying J in linear programming problems of forest management planning. 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). After version 2.1 it is possible to solve large ordinary linear programming problems without `problem()` function using `zmatrix->` option in `jlp()` function. To access the weights of optimal treatment schedules and to get them into files requires use of J-transformations (chapter 4.) inquiry functions (chapter 11.11), IO-functions (chapter 8), loops (chapter 7.2). An example is given in chapter 11.

Kommentoinut [LR(5]: Tarkistettava lukujen numeroinnit, linkitykset lukuihin

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 binary versions of J are developed under Windows 7. They are compiled as 32-bit applications and they are running also at least under Windows XP, Windows 7 and Windows 10. Binary versions have also been compiled for Macintosh and Linux. This manual is written mainly for Windows users, but it is indicated how J works differently in other environments.

Kommentoinut [LR(6]: 7 vai 10?

J is written in Fortran 90 and compiled with gfortran. There are both release and debug versions available. See User's guide to own J3.0 packages for more details.

All versions are ordinary console applications, see chapter 1.3. how to modify the I/O window on the first run.

If execution of J terminates unexpectedly, the console window disappears without outputting information about possible fault. It is thus recommended that J is run in Command Prompt so that the program window does not disappear. When the debug version is run in command prompt window the reason of the crash is printed as well as the source code line causing the error. When the release version is run in the command prompt window, the problematic source code line is not printed. See chapter 17 for more information of error handling. 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. If `j.par` is not available, the default number of objects is 5000.

Thereafter there can be in `j.par` file any number of J commands executed directly (e.g. you can give symbolic names for color indexes and line types used in graphics, or you can give shortcuts for commands which are handy e.g. when including repeatedly certain sections from include files). If you want to go directly into a specific J-application, you can put into `j.par` the corresponding include command.

If J is started from command prompt, there can be an include file name in the command line. This include file is run after commands read from `j.par` if it is available.

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. Or alternatively you can set the path to the folder containing J executable and run J in your working directory. Copy also the file `j.par` into each working directory.

It is recommended that J is run in the Command Prompt window even if it can be run also directly from the program shortcut.

Edit first the properties of the I/O window if you are using J directly or from the Command Prompt window. The properties of the I/O window can be changed by right-clicking the-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).

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

```
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 icon at the upper left corner.

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
```

1.5. Typographical conventions

In the description of J functions, optional arguments or options are indicated by [ ]. Alternative 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 an 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 explicit output is necessary, then '=' is put in front of the function name. Notation '`[=]`' means that the output can be given but it is not necessary. In most cases the default output `Result` is then used: In some cases no output is then generated (this will be indicated). For functions that return real values or matrices which can be used directly in arithmetic or matrix operations, the existence of output is not indicated.

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

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

1.6. J help

J help can be accessed using `help()` function. After command

```
help()
```

J prints all keywords of the default help file `jhelp.txt`. After command

```
help('key')
```

J prints the help related to the key word 'key'. When these commands are given with file-> option, then help is printed from the file indicated, e.g.

```
help(file->'ownhelp.txt')
help('key',file->'ownhelp.txt')
```

The structure of the help file is simple and the file can freely be edited by the user. The structure is similar to include file, and in fact most include files can be printed with help function. A section in the help file starts with

```
;key:
```

and the section ends with

```
;return
```

If no keyword is given, then all keywords are printed.

Note 1: If ;return is part of the help section and should be printed add to the line ! (after some spaces). help() function ;return does not accept any comments.

Note 2: In J3.0 the jhelp.txt is still rudimentary.

1.7. Customizing J: own packages

Starting from J3.0 J user is able to manage the software package and to add own functions, object types and options. The files defining own functions, object types and options are called own packages.

J function example() demonstrates how to use J features in own functions. See documentation in User's guide to own J3.0 packages for details.

2. J objects

2.1. Object names

Object names start with letter or with '$'. Object names can contain any of symbols '#.%§\_'. J is using '%' to name objects related to some other objects. E.g. function `stat(x1,x2,mean->)` will store means of variables `x1` and `x2` into variables `x1%mean` and `x2%mean`. Objects with name starting with '$' are not stored in the automatically created lists of input and output variables when defining transformation sets. Starting from J2.1 also the variable `Result` which is the output variable, if no output is given, is not put into these lists.

Object names can contain special characters (e.g. `+-*=()`) if these are closed within '`[`' and '`]`', e.g. `a[2+3]`.

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: Other objects except matrices can equivalently be deleted by giving command

```
obj1,…,objn = 0
```

This is because the output objects of any functions are first deleted before defining them anew. Usually an object is automatically deleted if the object name is as an output object for other functions.

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

Note 3: Starting from version J3.0 deleting a compound object deletes also such subobjects which have no meaning when the main object is deleted. But e.g. if a data object is deleted then the as-sociated transformation object is not deleted as the transformation can be used independently.

Note 4: Also files can be deleted with `delete(file)`. See IO-functions for details.

Note 5: If the user has defined own new compound objects in the open source J3.0 software she needs to define the associated delete function. See User's guide to own J3.0 packages for more information.

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 and in User's guide to own J3.0 packages.

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.

2.4.1. Real variables and constants

A real variable is a named object associated with a single real value. The value can be directly de-fined 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.

2.4.2. Character constants and variables

Character constants are generated by closing text within apostrophe signs ('). Character constants are used e.g. 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 constants within character constants.

2.4.3. 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.

2.4.4. 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 non-zero 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=    1.00000
```

2.4.5. 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.

2.4.6. 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.

2.4.7. 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 trans-formation set are done separately for each observation.

There is an implicit transformation set `$Cursor$` which is used to run the command level. The name `$Cursor$` may appear in error messages when doing commands at command level. An-other 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 `$Cursor2$`.

2.4.8. Simulator

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

2.4.9. 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.

2.4.10. Problem definition object

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

2.4.11. 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 sub-figure 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.

2.4.12. Function object

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

2.4.13. 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()` functions to deal with the storage.

2.4.14. 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. See bit functions.

2.4.15. 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.
`Arg`	Default argument variable when a transformation set is used as a function.
`Record`	Variable getting the record number when reading data files in `data()` function.
`Subrecord`	Variable getting the record number for subdata when reading hierarchical data in `data()` function.
`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 object referring to the last data set made, used as default data set.
`$`	Object name used to indicate console and '`*`' format in reading and writing command.
`x#`	Variable used when drawing functions.
`Selected`	Variable used to indicate the simulator selected.
`Printinput`	Variable used to specify how input lines are printed.
`Printoutput`	Variable used to indicate how much output is printed.
`Duplicate`	A special variable used in data function when duplicating observations.
`$Buffer`	A special character object used by the write function.
`$Input0$`	Text object used for original input line.

`$Input1$` Text object for input line after removing blanks and comments.

`$Input2$` Text object for input line after interpreting "-sequencies.

`$Debug` Variables used to put debugging mode on.

`$Recursioion~~n~~`
Variable telling recursion level of transformations.

`$Crash` Variable used to tell error handling system to crash the system when obtaining illegal values for some variables. This can be used to trace the calling sequence of subroutines in case of *j* -errors.

`$Warnings` Variable telling the current number of printed warnings. If 30 warnings are printed then the printing of warnings is stopped. Printing of warnings can be reactivated by giving `$Warnings=0`.

`Testoptions` If `Testoptions=1`, then it is checked that J-functions clear options properly. May help in debugging errors of J-function.

`Accepted` When using functions dealing with data, `Accepted` gives the number of accepted observations.

`$1` Variable having value 1. Needed in regression context to indicate the regressor for intercept.

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 `;trace()` function whenever it is called:

`Tracevars` Cumulative list of all objects used in all ;trace functions.

`Tracestatus`
Row vector corresponding to `Tracevars` list indicating if tracing code is generated.

`Tracelevel`
Vector indicating the tracing level for variables having the tracing code.

`Tracecount`
Counts of changes.

`Traceminstatus`
Indicates if minimum checking is effective.

`Tracemin` Minimum values.

`Tracemaxstatus`
: Indicates if maximum checking is effective.

`Tracemax`
: 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. The continuation character is kept as a part of the input line. One input record can contain 256 characters, and an input line can contain 2048 characters (this can be increased if needed).

Starting from version J3.0, if continuation is indicated by '>' then '>' will also remain in the logical input line.

Starting from J2.1, the continuation line cannot start with '`*`' or '`!`' because these are reserved to indicate comments.

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 upper left icon, select Edit, and then select 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.

Kommentoinut [LR(7]: Intel-kääntäjällä käännettyä exeä ei taideta saada aikaiseksi, kun minullakaan ei enää ole sitä tällä uudella koneella.

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 para-graph 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 out-put. 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='area1Z
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.

If value of the expression within a "" sequence is an integer then the value is dropped in the place without the decimal point and without any spaces, otherwise its value is presented in form which is dependent on magnitude of the value. After J3.0 the format can be explicitly specified within [] before the numeric value. Eg. text can be put into a figure as

```
fig =
drawline(5,5,mark->'y="[f5.2]coef(reg,x1)"*x1+"[f5.2]coef(reg,1)"')
```

See file `jex.txt` and Chapter 8 for an example.

3.2.3. Input 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->][,wait->])

Includes lines from a file or from a text object.

Argument:

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

Options:

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

`wait` J waits until the include file can be opened. Useful in client server applications. See chapter J as a server.

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.

Note 3: When editing the include file with Notepad ++, it is reasonable to set the language as Fortran freeform. This shows the control structures etc quite nicely.

;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][,print->])

Generates a sequence of input lines in a loop.

Arguments:

`I`	index variable
`start`	initial value
`last`	final value
`step`	step (optional, default is 1).

Options:

`print`	If present then the generated input is printed during each iteration. The input is printed in a cleanded form, i.e., spaces and tabs are removed, and all continuation lines are presenten on a single line.

;enddo

Closes a `;do()` loop.

Note 1: There can be 6 nested `;do` loops.

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

Note 3: It is not recommended to use `;do` loops in the console input even if this is possible.

Note 4: There is not yet ;exitdo. Use ;if(0);then .....;endif to bypass the rest of the ;do -block..

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(expr)

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

Argument:

`expr` a 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 info in file debug.txt')
;if(expr);then
...
;elseif(expr);then
...
;endif
```

Picks several lines into input based on some condition.

Argument:

`expr` 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:

`'adr'` 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 (it works like calling a subroutine). 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 re-places 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 functions 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)
```

muotoili: suomi

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. commands

```
l = list('?%mean')
print(l)
```

will print all variables whose name end with '`%mean`', and commands

```
l = list('x1%?')
print(l)
```

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

3.2.8. Accessing text object lines

Text object lines can be accessed similarly as list elements. If `textobj` is a text object, then the row `irow` of the text object can be dropped into the input lines using `@textobj(irow)`. E.g. if `prob%rows` is the text object describing problem rows in a problem object `prob`, then row `3` of the problem can be printed using

```
print(@prob%rows(3))
```

Text object lines can be printed in the same line with numeric variables by dropping text object lines into the format in a `write()` function. Eg. if the ouput of the `jlp()` function is `outp`, the both the problem row and the value of the constraint can be written:

```
write($,'(~@prob%rows(3)=~,f8.2)',outp%rows(3))
```

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.

3.3.1. 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 recognizes 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 argu-ments

! here the value of the argument of cos is obtained by 'computing' the val-ue of real variable c.

stat(D,H,min->,max->)  ! Here arguments must be variable names

plotyx(H,D,xrange->(int(D%min,5), ceiling(D%max,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 uses 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 trans-formation 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 other functions.

Kommentoinut [LR(8]: Siirto repoon

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 trans-formations 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 exists 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 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 `tr%input` and `tr%output` lists are created for input and output variables (`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 `tr%source` if option `source->0` is not given.

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

Kommentoinut [LR(9]: Siirto repoon

4.4. Executing 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:

`tr` 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:

`tr` 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: 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 an 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 do 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. 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.

5.2. 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.

5.3. Arithmetic functions

Kommentoinut [LR(10]: Näissä ei ole output, aruments, options -kuvauksia, mutta eivät välttämättä tarvitsekaan.

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.

Minimum and maximum

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

[=]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, argument in radians

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

Inverse trigonometric functions, result in radians

```
asin(x)
acos(x)
atan(x)
acotan(x)
```

Inverse trigonometric functions, result in degrees

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

Hyperbolic functions

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

[=]gamma(x)

Gives the value of gamma function for the argument x.

5.4. Probability distributions

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

Kommentoinut [LR(11]: Johdonmukaisuuden vuoksi funktioista voisi olla output, arguments ja options -kuvaukset?

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).

[=]cdf(x,df,chi2->]

Returns the cumulative distribution for the chi-square distribution with `df` degrees of freedom. If the difference of 2log-likelihood between a full model and restricted model is `x` and the full model has `k` parameters more than the restricted model, then

```
print(1-cdf(x,k,chi2->)
```

gives the p-value for testing if the restricted model deviates significantly from the full model.

[=]bin(k,n,p)

Gives the binomial probability that there will be `k` successes in `n` independent trials when in a single trial the probability of success is `p`.

[=]negbin(k,myy,theta)

Gives the probability that a negative binomial random variable has value `k` when the variable has mean `myy` and variance `myy+theta*myy**2`. Note that `negbin(k,n*p,0)=bin(k,n*p)`. Sorry for the parameter inconsistency with `rannegbin()`.

5.5. Random numbers

Random number generators are taken from Ranlib library of Netlib. Additional generators are straightforward to add if needed.

[=]ran()

Returns a uniform random number between 0 and 1.

[=]rann([mean][,sd])

If no arguments are given, the function returns normally distributed random number with mean zero and variance 1.

Arguments:

`mean`	the mean of the random number
`sd`	the standard deviation of the variable, default is 1

[=]ranpoi(myy)

Returns random Poisson variate.

Kommentoinut [LR(12]: Repoon koodien yhteyteen

Kommentoinut [LR(13]: Johdonmukaisuuden vuoksi funktioista voisi olla output, arguments ja options -kuvaukset?

Kommentoinut [LR(14]: Repoon koodien yhteyteen

Arguments:

`myy` mean and variance of the distribution

[=]ranbin(n,p)

The function returns random number distributed according to the binomial distribution.

Arguments:

`n` number of trials

`p` probability of success

[=]rannegbin(r,p)

The function returns random number distributed according to the negative binomial distribution.

Arguments:

`r` the number of failures

`p` probability of success

The result is the number of successes before `r`'th failure. `ranbin(r,1)` returns always 1.7e37 and `ranbin(r,0)` returns always 0.

Note 1: there are different ways to define the negative binomial distribution. In this definition a Poisson random variable with mean λ is obtained by letting r go to infinity and defining $p= \lambda /(\lambda+r)$. The mean μ of this definition is p*r/(1-p) and the variance is V=p*r/(1-p)2. Thus given μ and V, r and p can be obtained as follows: $p=1- \mu /V$ and $r= \mu^2/(V- \mu)$. This is useful when simulating 'overdispersed Poisson' variables. Sorry for the inconsistency of parameters with function `negbin()`.

Note 2: `r` can also have a noninteger values. This is not in accordance with the above interpretation of the distribution, but it is compatible with interpreting negative binomial distribution as a compound gamma-Poisson distribution and it is useful when simulating overdispersed Poisson distributions.

=select(k,n)

The function returns a column vector with `n` elements indicating random selection of `k` elements out of `n` elements.

5.6. Special numeric functions

Kommentoinut [LR(15]: Funktiooille output, arguments, options

[=]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`.

[=]logistic(x)

Returns the value of the logistic function 1/(1+exp(-x)). This can in principle computed by the transformation, but the transformation will produce an error condition when the argument -x of the exp-function is large. Because the logistic function is symmetric, these cases are computed as 1-1/(1+exp(x)). Because the logistic function can be needed in the nonlinear regression, also the derivatives are implemented. Eg when `f=logistic(a*(x-x0))`, then the derivatives can be obtained with respect to the parameters a and x0.

5.7. 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.

Note 1: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
```

Note 2: 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.

Kommentoinut [LR(16]: kehitysideat repoon koodin yhteyteen

5.8. 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)
```

Kommentoinut [LR(17]: output, arguments, .options luvussa esiteltyihin funktioihin

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 multi-plied.

The following matrix functions are currently available:

6.1. Defining matrix()

Standard case

[=]matrix(nrows[,ncols][,in->][,diagonal->] [,row->][,form->] [,values->][,matrix->])
(values if in-> option is present without an argument)
/

Generates a matrix.

Arguments:

`nrows` number of rows (can be later obtained using `nrows()` function)

`ncols` number of columns, default is 1 (that is vectors are assumed to be column vectors). Number of columns can be seen using `ncols()` function.

Options:

`in` If `in->` option is present alone then the values are given in a input paragraph. If an argument iis presented, then matrix is reaf from the file given

`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`~~`row` 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.

form With form-'b' the file given in in-> is single precision binary, 'B' indicates double precision binary. EI TOIMINe vielä

`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.

`matrix` The matrix is made from submatrices. ncols and nrows indicate the numbers of submatrix columns nd rows.

Note: If values are not given through **in->** or **values->** options then the elements will be zero.

Examples:

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

Starting from J2.1 all elements of a matrix can be set with simply assignment statement, e.g. `amat=0`. Previously such assignment statement made `amat` a real variable and substituted the value to this variable.

Making a matrix from a list of variables

=matrix(list)

Argument:

`list` list of real variables

This makes a column vector consisting of the current values of the real vriables.

6.2. Matrix functions

=submatrix(matrix,[row->][,column->])

Takes a submatrix from a matrix

Argument:

`matrix` input matrix

Options:

`row` A single argument gives the row to be taken. A range ow rows is indicated in the form row->(firstrow,-lastrow)

`column` A single argument gives the column to be taken. A range ow rows is indicated in the form column->(firstcolumn,-lastcolumn)

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

Kommentoinut [LR(18]: argumetns, options -osiot

Puts all elements equal to the given value, or all the diagonal elements if `diagonal->` option is present. This function is now useful only with diagonal-> option because all elements of a matrix can now be given a fixed value using simple assignment. e.g. amat=11.

{=]t(a)

Computes the transpose of a matrix

Argument:

`a` 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` 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:

Kommentoinut [LR(19]: limit1, limit2

`a, b` 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 elements.

`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:

Kommentoinut [LR(20]: limit1, limit2

`a` a 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)

[=]elementprod(a,b)

Computes the matrix where each element is the product of the corresponding elements of the argument matrices.

Arguments:

`a,b` matrices having the same number of elements, they can also be intermediate results of matrix operations (e.g. `c=elementprod(a,c+d)`). If `a` and `b` are real numbers, their ordinary product is formed.

[=]submatrix(a[,row->][,column->][,diagonal->])

Takes a submatrix from a matrix-

Argument:

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

Note: 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.

[=]mean(a[,row->][,column->][,var->][,sd->][,sum->]

[,min->][,max->])

Computes mean or other properties from specified matrix elements-

Argument:

`a` 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)`.

`var` sample variance of specified elements is stored in variable `output%var`.

`sd` sample standard deviation of specified elements is stored in variable `output%sd`.

`sum` sum of specified elemets is stored in variable `output%sum`.

`min` minimun of specified elemets is stored in variable `output%min`

`max` sample variance of specified elemets is stored in variable `output%max`

Note: 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.

muotoili: Fontti: (Oletus) Courier New

[=]nrows(a)

Returns the number of rows in a matrix.

Argument:

`a` a matrix object

Note: `nrows()` works also for text objects.

[=]ncols(a)

Returns the number of columns in the matrix.

Argument

`a` a matrix object

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

Returns the number of elements in the matrix (=nrows*ncols).

Argument:

`a` a matrix object

Option

`any` function 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->])

Returns the position of a number in a matrix (usually vector). Note, the matrices are stored in row order.

Arguments:

`val` a real value

`a` 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]))

Kommentoinut [LR(21]: option-kuvaus

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

Argument:

`a` 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 vari-able names. Currently index()function can be used to get the proper column number of the data matrix.

Kommentoinut [LR(22]: Repoon koodien seuraksi.

[=]qr(a)

Makes qr decomposition of matrix `a`. This can be used to study if columns of a are linearly dependent. J prints a matrix which indicates the structure of the upper diagonal matrix R in the qr decomposition. If column k is linearly dependent on previous columns the k'th diagonal element is zero. If output is given, then it will be the r matrix. Due to rounding errors diagonal elements which are interpreted to be zero are not exactly zero. Explicit r matrix is useful if user thinks that J has not properly interpreted which diagonal elements are zero.

Argument:

`a` a matrix object

[=]eigen(a)

Computes eigenvectors and eigenvalues of a square matrix. the eigenvectors are stored as columns in matrix *`output`*`%matrix` and the eigenvalues are stored as a row vector *`output`*`%values`.

Argument:

`a` a square matrix object

[=]envelope(a[,nobs->])

Computes the convex hull of points.

Output:

(nvertex+1, 2) matrix of the coordinates of the convex hull, where nvertex is the number of verteces. The last point is the same as the first point, so the transpose of the matrix can be directly used in `drawline()` function.

Argument:

`a` (n ,2) matrix of point coordinates

Option:

`nobs` gives the number of points if not all points of the input matrix are used

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()

Kommentoinut [LR(23]: argumentti: expr

The one line if-statement.

`if()j_statement...`

Groups of statements 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 transfers 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 another 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)

Kommentoinut [LR(24]: arguments-osio

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 whether 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')

Kommentoinut [LR(25]: arguments-osio

Continues execution of transformations from the given address.

jump('address')

Computes 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->]
[,file->][,func->][,debug->])

Print values of variables or information about objects.

Arguments:

`arg1,...,argn`
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.g. `row->(10,-15)`).Note that `nrows()` function can be used to get the number of rows.

`file` the file name as a character variable or a character constant. Redirects the output of the `print()` function to given file. After printing to the file, the file remains open and must be explicitly closed (`close('file')`) if it should be opened in a different application.

`form` when a matrix is printed, the format for a row can be given as a Fortran format, e.g. `form '(15f6.2)'` may be useful when printing a correlation matrix.

`debug` the associated real variable part is first printed, and thereafter the tow associated two integer vectors, the real vector and the double precision vector

`func` all functions available are printed

Note: 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][,eof->var] [,wait->])

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

Arguments:

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

`format`

muotoili: Fontti: (Oletus) Courier New

Kommentoinut [LR(26]: Poisto?

`'b'` unformatted (binary) data
`'bn'` unformatted, but for each record there is integer for the size of the record. Does not work when reading matrices.
`'bis'` binary data consisting of bytes, each value is converted to real value (the only numeric data type in J). This works only when reading matrices.
`'(....)'` a Fortran format. Does not work when reading matrices.
`$` the * format of Fortran

`obj1,…,objn`
J objects

Options:

`eof` Defines the variable which indicates the end of file condition of the file. If the end of the file is not reached the variable gets the value 0, and when the end of file is reached then the variable gets value 1 and the file is closed without extra notice. Note that when `eof->` option is not present and the file ends then an error condition occurs and the file is closed.

`wait` J is waiting until the file can be opened. Useful in client-server applications. See chapter J as a server.

Note 1: Use `ask()` or `askc()` to read values from the terminal when reading lines from an include file.

Note 2. When reading matrices, their shapes need to be defined earlier with `matrix()` function.

`write(file,format,val1,…,valn[,tab->][,rows->])! case[1/6]`

Writes real values to a file or to the console. If `val1` is a matrix then this matrix (or usually vector) is written or at most as many values as given in the `values->` option.

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

Kommentoinut [LR(27]: Suunnitelmat repoon

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 or there is a matrix argument. If they are not, then just the real value which is anyhow associated with each J object is printed (usually it will be zero). If the `val1` argument is a matrix, then all values are printed.

`val1,…,valn`
real values

Options:

`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. If there are no decimals after the decimal point also the decimal point is dropped.

`rows` If `val1` is a matrix or a vector and `rows->` has one argument then at most as many values are written as given in this option, if there are two arguments then the option gives the range of written rows in the form rows->(row1,-row2). If the upper limit is greater than the number of rows, no error is produced, all available rows are just written.

`write(file,'t',t1,val1,t2,val2,…,tn,valn[,tab->]) ! case[2/6]`

Tabulation format. 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.

Arguments:

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

`'t'` tabulation format

`t1,val1,t2,val2,…,tn,valn`
KUVAUS

Kommentoinut [LR(28)]: Kaipaa kuvausta

Options:

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

Note. Variable names from a variable list can be written using `@`–construction. E.g. if `xl` is a variable list, then both the name and value of element i can be written:

```
write($,'w',1,'@xl(i)=',8,@xl(i))
```

`write(file,'w',w1,val1,w2,val2,…,wn,valn[,tab->]) ! case[3/6]`

Width format: 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).

Arguments:

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

`'w'` width format

`w1,val1,w2,val2,…,wn,valn`
KUVAUS

muotoili: suomi
Kommentoinut [LR(29]: Kaipaa kuvausta
muotoili: suomi

Options:

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

Note. Variable names from a variable list can be written using `@`–construction. E.g. if `xl` is a variable list, then both the name and value of element `i` can be written:

```
write($,'w',8,'@xl(i)=',7,@xl(i))
```

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'.

write(file,text_object) ***! case[4/6]***

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

Arguments:

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

`text_object`
J text object

write(file,character) ***! case[5/6]***

When second argument is a character constant or character variable referring to a character constant, then it is written to the file (or to the screen is file is `$)`. Note character constant can contain "" -sequences, and after J3.0 these can contain also format specifier.

Arguments:

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

`character` a character constant or a variable referring to character constant

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

Examples:

```
write('out.txt','sin("[f4.4]a" is "[f5.3]sin(a)", believe or not')
dir='d:\j\'
write('"dir"example.out','(~the values were ~,4f4.0)',@values)
```

Writing into $Buffer ! case[6/6]

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.

close(file)

Closes an open file.

Argument:

`file` a character variable or constant telling the name of an open file. The file has been created and opened with `write()`, `print()` or `save()` functions.

exist(name[,object->])

Tests if a file or object with a given name exists.

Argument:

`name` a character variable or constant telling the name of the file or an object

Option:

`object` indicates that the existence of an object is inquired

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

delete(file1,…,filen)

Deletes an existing file or files. If the file is open it will be closed prior to deleting. If the file does not exist, an error will be returned.

Argument:

`file1,…,filen`
character variables or constants telling the names of the files.

Note: Function `delete()` is also used to delete objects. For details, see Deleting objects.

`ask([var][,default->][,q->][,exit->])`

Ask values for a variable while reading commands from an include file.

Argument:

`var` 0 or one real variable (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.

Note: If there are no arguments, then the value is asked for the output variable, otherwise for the argument. 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, 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[,default->][,q->][,exit->])`

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

Arguments:

`chvar1` 0 or one character variable (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.

Note: 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, 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). It is recommended that two linked data objects are created with one `data()` function call only in case the data is read from a single file where subdata observations are stored immediately after the upper data observation. 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

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

Output:

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

Options:

`read` variables read from the input file or the name of the list containing all variables to be read in. If no arguments are given and there is no `readfirst->` option then the variables to read in are stored in the first line of the data file separated with commas. Also the ... -shortcut can be used. If no arguments are given and there is `readfirst->` option then the variable names are read from the second line. See Chapter 13 in file jex.txt for examples.

`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. Format `'bs'` indicates that the file is opened with access='stream'. Format `'d'` indicates direct access in Intel Fortran application and format `'d4'` indicates direct access in Gfortran application (Intel and Gfortran interpret 'recl' key word differently in 'open' statement). Binary files written with Pascal application in Windows can read in using either `'bs'` or `'d'`. Any general Fortran format can be given as character constant or variable (e.g. `'(4f4.1,1x,f4.3)'`).

Kommentoinut [LR(30]: Jaetaan varmaankin pelkästään gfortranilla käännettynä

`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 and the first line starts with `'!'` or with `'*'` then the first line is printed (a text header). If no variables are given and the first line does not start with `'!'` or with `'*'` then the

first line is a J-command line which is executed. The intended use for this is that the first line can contain both the names and values associated with the whole file. E.g the first line can be `region,simulation_time=10,2016.` See Chapter 13 in file jex.txt for examples.

`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. Option `filter->` can utilize automatically created variable `Record` which tells which input record has been just read. If observations are rejected, then the `Obs`-variable has as its value number of already accepted observations+1.

`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. Option `reject->` can utilize automatically created variable `Record` which tells which input record has been just read. If observations are rejected, then the `Obs`-variable has as its value number of already accepted observations+1.

`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` a 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 `'obs_variable%obsw'`.

`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` There are two uses of this option. First, a data set can be created without reading from a file or from the following input paragraph 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. Second, if `read->` option is present `nobs->` option can be used to indicate how many records are read from a file and what will be the number of observations. Currently `reject->` or `filter->` can not be used to reject records (consult authors if this would be needed). If there are fewer records in file as given in `nobs->` option, an error occurs. There are three reasons for using `nobs->` option this way. First, one can read a small sample from a large file for testing purposes. Second, the reading is slightly faster as the data can be read directly into proper memory area without using linked buffers. Third, if the data file is so large that a virtual memory overflow occurs, then it may be possible to read data in as linked buffers are not needed. If virtual memory overflow occurs without using this option and the data file is schedules data for `jlp()` function, then probably also `swap->` option is needed in `jlp()`. **Note**: In case nobs-> option is present and read-> option is absent either maketrans-> or keep-> option (or both) is required.

`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 (or perhaps not). Note that the buffers are not needed if number of observations is given in `nobs->`.

`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)`.

`disk` The content of *`output`*`%matrix` is stored into a direct access file *`output`*`%matrix.bin` and the information needed to open the file (the record length and the number of records) is stored into file *`output`*`%matrix%info.txt`. The data can be used exactly as before, the only difference is that functions using the data are slower. If the data function is given in the same form in a new J-run and the files exist, J-askes if these files are used or replaced with new files. See load().

Kommentoinut [LR(31]: Muistiinpanoista löytyi merkintä. "datafunktion nobs-option kohdalla:
Jos on nobs ja ei ole read-> , niin vanhan version mukaa piti olla maketrans. Ny pitää olla maketrans tai keep (tai molemmat)"

Meniskö tämä näin?

`subdisk` Works for subdata exactly as `disk->` for the upper level data

`rfhead` when reading data from a text file, the first line can contain a header which is printed but othewise ignored

`rfcode` The data file can contain also J-code which is first executed. Note the code can be like `var1,var,x1...x5=1,2,3,4,5,6,7,` which give the possibility to define variables which describe the data set.

If there are both `rfhead->` and `rfcode->` then `rfhead->` is excuted first. `rfhead->` and `rfcode->` replace `readfirst->` option which was too complicated.

`rfsubhead` works for subdata similarly as `rfhead->` for data.

`rfsubcode` works for subdata similarly as `rfcode->` for data

Note 1: `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 2: See common options section for how data sets used in other J functions will be defined.

Note 3: All `read->` variables are treated as real variables.

Note 4: 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.

Note 5: When reading the data the `obs->`variable (default `Obs`) can be used in `maketrans->` transformation and in `reject->` option and `filter->` option, and the variable refers to the number of observation in resulting data object. The variable `Record` gets the number of the read record in the input file, and can be used in `maketrans->` transformations and in `reject->` and `filter->` options. If `subdata->` option is given, variable `Subreject` gets the number of record in the sub file, and it can be used in `submaketrans->` transformations and in `subreject->` option and in `subfilter->` option.

Note 6: Options `nobs->100`, `reject->(Record.gt.100)` and `filter->(Record.le.100)` result in the same data set, but when reading a large file, the `nobs->` option is faster as the whole input file is not read.

Note 7: If no observations are rejected, `obs` variable and `Record` variable get the same values.

Note 8: If virtual memory overflow occurs, see `nobs->` option.

9.2. Modifying an existing data set: editdata( )

`editdata(dataset,trans->)`

Argument:

`data` 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 ma-trix 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 upper level data set object

`subdata` the lower data set object

`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.

Kommentoinut [LR(32]: Repoon

`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 `obs`_variable_of_the_upper_*`data`*`%obsw`.

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. Making new data from components: newdata()

`=newdata(arg1,…,argn[,read->][,maketrans->][,obs->][,filter->][,reject->])`

Makes a new data set from existing data sets, matrices or using transformations

Arguments:

Data sets or matrices having equal numbers of observations or rows.

Options:

`read` gives names for all columns of all matrix arguments

`maketrans` transformation set generating new variables. All output variables whose names do not start with '$' are included.

`obs` name of the observation variable, default 'Obs'. Note that this variable is not included in the data matrix but is generated when using the data.

`filter` condition for including observations in the new data

`reject` condition for rejecting observations in the new data

muotoili: Kappaleen oletusfontti

~~9.4.~~9.5. Combining two observations in same class: crossed( )

=crossed(data->,class->,trans->,keep->,dummy->[,sym->])

Output:

a 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`. `sym->` option defines that for observations i and j a new observation is formed only for i and j not both for i and j, and j and i. 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
```

This function is used in stem curves modeling to form products of residuals of different trees in the same stand.

~~9.5.~~9.6. Utility functions for data sets

~~9.5.1.~~9.6.1. 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:

`variable` 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 `nrows()` function.

***Later there will be different ways to utilize the obtained values in connection of data sets.

Kommentoinut [LR(33]: repoon

Note 3. The `values()` function can be utilized e.g. in generating domains for all different owners or regions found in data.

~~9.5.2.~~9.6.2. Number of observations: nobs( )

[=]nobs(dataset)

Gets the number of observations in a data set.

Argument :

`dataset` 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',
dat%matrix(i,"index(x4, dat%keep)")- dat%matrix(i-1,
"index(x4,dat%keep)"))
enddo
```

Fast:

```
inx4=index(x4, dat%keep)
```

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

Slow:

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

9.6.3. Getting group vectors from grouped data: classvector( )

classvector(class->[,data->],obs[,trans->])

~~9.5.3.~~9.6.4. 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.

Arguments:

`dataset` a data set

`obs` row number in the data matrix of the dataset

Option:

`trans` if option is present, these transformations are also executed.

~~9.6.~~9.7. 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:

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

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

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

Kommentoinut [LR(34]: Repoon

Kommentoinut [LR(35]: Ovatko kaikki käyttäjän ulottuvilla vai onko jotkut J:n sisäisiä?

Kommentoinut [LR(36]: Repoon

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

`var1,…,varn`
: variables for which the statistics are computed, the default is all variables in the data (all variables in the data matrix plus the output variables of the associated transformation set) and all output variables of the `trans->` tranformations.

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 `x1%pien` and `x2%pien`. 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.

Note 1: `stat()` function prints min, max, means, sd and sd of the mean computed as sd/sqrt(number of observations)

Note 2: 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:

`var1,…,varn`
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.

Note: 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->[,minobs->][,nobs->]
[,xrange->][,dx->]
[,classes->][,z->][,zrange->][,dz->][,mean->][,trans->]
[,filter->][,reject->][,transafter->][,print->])

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

minobs-> minimum number of observations in one class. Classes are merged so that this can be obtained.

Output:

A matrix containing class information (details given below)

Arguments:

`var1,…,varn`
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

`minobs` minimum number of observation in a class, obtained by merging classes. Does not work if z is given

`nobs` total number of accepted observations, default nobs->Nobs. If nobs is smaller than minobs and nobs is not given, an error occurs.

`xrange` `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 given without arguments, J variables x%min and x%max exist, and if they exist they are use, and if they do not exist an error occusr. Note that these can be generate with stat(min->,max). 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 (z variable) defining classes in two dimensional classification.

`zrange` `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

`classes` number of classes, has effect if dx is not defined in ~~`xrange`~~`dx->`. The default is `classes->7`. If z is given then, there can be a second argument, which gives the number of classes for z, the default being 7.

`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.

`print` by setting `print->0` the classification matrix is not printed

Operation:

If z variable is not given then first column in printed output and the first row in the output matrix (if given) contains class means of the x variable. In the output matrix the last element is zero. Second column an TARKASTA VOISIKO VAIHTAArow 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.

Kommentoinut [LR(37]: dx-> ?

Kommentoinut [LR(38]: dx-> ?

muotoili: Fontti: (Oletus) Courier New

muotoili: Kappaleen oletusfontti

Muotoiltu: J optio

muotoili: Fontti: (Oletus) Courier New

muotoili: Kappaleen oletusfontti

Muotoiltu: Sisennys:Vasen: 0 cm, Ensimmäinen rivi: 0 cm

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->][,step->][,trans->]
[,filter->][,reject->][,transafter->][,var->][,corr->])

Computes a linear regression function.

Output :

A regression object, utilized through `value(), coef(), se(), rmse(), mse()` and `r2()` functions. If no output is given, output is NOT generated into `Result`

Arguments:

`y, x1,…,xn` y-variable, x-variables (not including constant term)

Options:

`data` data sets used

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

`step` A backward stepwise regression is applied until the absolute t-values of all regressors (excluding the intercept) are larger than the argument of the 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.

`var` if this option is present, the variance-covariance matrix of parameter estimates is stored in matrix *`output`*`%var`.

`corr` if this option is present, the correlations of the parameter estimates are put to the nondiagonal elements of *`output`*`%corr` and standard errors are put to the diagonal elements.

Function `regr()` is using Lapack routines from Netlib to solve the normal equations..

Note 1 If the dependent variable or any of the regressors gets a value whose absolute value is greater than 1.7d19, then the observation is rejected

Note 2. All regressors which are constants for all observations or which are linearly related to some other regressor are dropped.

Note 3. If any-> option is present in coef-or se-functions, then these functions give zero values and do not produce error when applied to dropped regressors-

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])

Output:

the value of the regression function

Arguments:

`regr_object`
regression object created with `regr()` function

`x1,...,xn` the values of the independent variables. .

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[,any->])

Output:

the value of the coefficient of a x-variable

Arguments:

`regr_object`
regression object created with `regr()` function

`xvar` x variable .

Options:

`any` output is zero, if x-variable is not a regressor in the regression. Useful if a regressor is dropped in the stepwise regression or due to degeneracy.

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

[=]se(regr_object,xvar)

Output:

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

Arguments:

`regr_object`
regression object created with `regr()` function

`xvar` x variable .

Options:

`any` output is zero, if x-variable is not a regressor in the regression. Useful if a regressor is dropped in the stepwise regression or due to degeneracy.

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

[=]mse(regr_object)

Output:

the MSE of the regression

Argument:

`regr_object`
regression object created with `regr()` function

[=]rmse(regr_object)

Output:

the RMSE of the regression

Argument:

`regr_object`
regression object created with `regr()` function

[=]r2(regr_object)

Output:

the R2 of the regression

Arguments:

`regr_object`
regression object created with `regr()` function

[=]nobs(regr_object)

Output:

the number of observations used to compute the regression

Argument:

`regr_object`
regression object created with `regr()` function

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

Output:

the number of parameters in the regression (including intercept)

Argument:

`regr_object`
regression object created with `regr()` function

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.

Kommentoinut [LR(39]: Repoon

10.6. Nonlinear regression: nonlin()

[=]nonlin(y,f[,data->],trans->,par->,der->[,parmax->] [,parmin->][,maxiter->][,slow->][,tole->][,var->] [,corr->][print->)

Nonlinear regression is computed using the modified Gauss-Newton algorithm based on computing iteratively linear degression with first order Taylor series. Lapack routines are used for solving normal equations (as in linear regression).

Output:

regression object, can be used in `mse()`, `rmse()`, `se()` and `r2()` functions. If no output is given, output is not generated into `Result`.

Arguments:

`y` dependent variable

`f` function

`data` data sets used (default last data generated)

`tran` transformation set defining `f` and derivatives (see below)

`par` parameters of the function and possibly also initial values. If initial values are not given, the current values of the parameter variables are used. Initial values need to be given to all parameters if they are given at all.

`der`	derivatives of the function with respect to the parameters. These can be computed using the `der()` function.
`parmax`	maximum values of the parameters. If `parmax->` is present the maximum needs to be given to all parameters.
`parmin`	minimum values of the parameters. If `parmin->` is present the maximum needs to be given to all parameters.
`maxiter`	maximum number of iterations. One iteration is done when MSE is computed with one set of values of the parameters. It is counted as one iteration when new linear regression is computed or when the step size is reduced after the linear regression. Default is 100.
`slow`	determines using SSE when the algorithm has converged. If slow is negative, the iteration stops when the decrease of SSE is smaller than abs(slow)%. If slow is positive then the iteration stops when decrease in RMSE is smaller than slow. Default slow=-0.1.
`tole`	Iteration stops if the suggested change is less than tole% of the current standard error of each parameter estimate. Default tole=0.1
`var`	If option is present then the variance covariance matrix is stored into matrix output%var
`corr`	If this option is present, the correlations of the parameter estimates are put to the nondiagonal elements of output%corr and standard errors are put to the diagonal elements.
`print`	Determines whether printing is produced. With `print->0` no printed output is produced, with `print->1` only solution is printed and with `print->2` information on iterations is printed.

Kommentoinut [LR(40]: Lisätty muistiinpanoista: "nonlin : print->0, ei tulosta mitään, print->1 tulostaa vain ratkaisun print->2 tulostaa tietoa iteraation etenemisestä."

Function `nonlin()` is using modified Gauss-Newton method to compute the parameter estimates and their variance covariance matrix. The linear regressions involved in the method are done similarly as in function `regr()`.

Example:

```
tr=trans()
da,db,dc=der(a,b,c)
f=a+b*(1-exp(-c*t)) ! t is variable in the data
/
nl=nonlin(y,f,trans->tr,par->(a,b,c,0,10,0.1),der->(da,db,dc))
```

Thereafter functions `mse()`, `rmse()`, `se()` and `r2()` can be used similarly as in linear regression.

10.7. Smoothing spline: smooth( )

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

NOTE: Owing to unclear license conditions of gcvspl, the function is removed from the basic distribution. See 'Users guide to J3.0 own packages' how to get it back.

10.7.1. 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,param_index)`.

Arguments:

`x`	dependent variable
`y`	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. Default is `degree->3`.
`wish`	`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.

Note 1: 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

Note 1: 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.

10.7.2. 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`
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` see above (`smooth()` function)

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

`maxiter` gives 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: `min->` and `max->` options do not yet work for smoothing data.

The function is based on the GCVSPL package of H.J Woltring (see Woltring 1986) which is loaded from DMOZ page https://www.dmoz.org/Computers/Programming/Languages/Fortran/Source_Code/Statistics_and_Econometrics/

Kommentoinut [LR(41]: Pidetäänkö tässä dokumentissa, vai siiretäänkö erilliseen tai own packages dokumenttiin? Eivät ole käytettävissä julkaistavassa J3.0:ssa.

10.8. Variance and covariance components

Function `varcomp()` can be used to compute variance and covariance components (covariances of random effects) when there are no other fixed parameters than the mean. The function is using the 'analysis of variance' estimator (see p. 474 in Searle 1971). The variance components are computed using the model:

$y_{ij}=m+b_i+e_{ij}$,

where m is fixed mean, b_i is random effect in class i and eij is residual error in individual j in class i. Covariances of random class effects and residual errors between different variables are estimated by computing variance components of sums of variables and utilizing the equation var(x+y)=var(x)+var(y)+2cov(x,y), i.e.

cov(x,y)=0.5*(var(x+y)-var(x)-var(y)

[=]varcomp(var1,…,varn,class->clvar[,cov->][,corr->] [,data->][,filter->][,reject->][,trans->])

where

Output:

If output is given, then the average values of variables are stored into vector *output*`%mean`, and the gls estimates of the mean values are stored into vector *output*`%glsmean.`

Arguments:

`var1,...,varn`
variables for which variance and possible covariance components are computed

Options:

`class` gives the variable defining classes. It is assumed that the class is changing when the class variables changes

`cov` covariance components are computed. Option requires that output is given. The covariance matrix of class effects is stored into the matrix *output*`%varb` and the covariance matrix of residual errors is stored into matrix *output*`%varw.`

`corr` correlations of random effects and residual errors are computed. Option requires that output is given. The correlation matrix of class effects is stored into the matrix

output%corrb and the correlation matrix of residual errors is stored into matrix *output*%corrw.

data->, filter->, reject-> and trans-> options work as in other functions dealing with 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 11.12 JLP examples.

11.1. Optimization problem without factories

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:

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

subject to the following constraints:

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

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

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

$$w_{ij} \geq 0 \quad \text{for all } i \text{ and } j \quad (5)$$

$$z_k \geq 0 \quad \text{for } k = 1, \ldots, q \quad (6)$$

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:

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

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:

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

subject to:

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

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

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

$$w_{ij} \ge 0 \quad \text{for all } i \text{ and } j \tag{12}$$

$$z_k \ge 0 \quad \text{for } k = 1,\ldots,q \tag{13}$$

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.

Ordinary linear programming problems are problems which contain only z-variables.

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 follow

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

subject to the following utility constraints

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

and technical constraints

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

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

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

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

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

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

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

$$x_k^{ij} \ge 0, k \in \mathbf{K} \quad (21)$$

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*

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.

In typical problems the utility constraints including x_{kf} are of form $x_{kf} \leq C$ 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_{kf}^{i}. 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 (1997)

Solution algorithm for the optimization problem including factories is described in Lappi and Lempinen (2014).

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 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). If the same domain definition is in several places is slightly inefficient in computations, e.g. `jlp()` function computes and prints the values of x-variables for each domain definition even if the same values have been computed and printed for earlier occurrence of the domain definition.

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+  !both lower and upper bound is possible
(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.

The objective row must be the first row. The objective must always be present. If the purpose is to just get a feasible solution without objective, this can be obtained by minimizing a z-variable which does not otherwise appear in the problem (remember `z->` option in the `jlp()` function.

In problems having large number of variables in a row it is possible to give the coefficients as a vector and variables as a list e.g.

ESIMERKKI

In problems with x-variables it is possible to maximize or minimize the objective without any constraints. In factory problems this would also be quite straightforward to implement, but it

Kommentoinut [LR(42]: Lisätty

Kommentoinut [LR(43]: Tähän yksinkertainen esimerkki vektori*lista määrittelystä

does not come as a side effect of computations as in the case of maximization of x-variables, and thus it has not been implemented. The maximization of a factory objective without constraints can be obtained by adding to the problem constraints which require that the amounts of transported timber assortments to different factories are greater than or equal to zero.

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, which are x-variables and which are factory variables (see Lappi 1992).

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

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.

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. The problem obtained by taking multiple domains in domain definition rows into account is called 'expanded problem'.

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.

Starting from J3.0 it is also possible to specify the period of the row for each row containing x-variables. The period is given between two '#' signs at the beginning of the row, e.g.

```
#2# income.2-income.1 >0
```

If the row contains x variables from several periods, the period of the row is the last period of the x variables. If the period is given for some rows containing x variables, it must be given for all except for the objective row. The period of the objective is assumed to be the last period as having any other period for the objective would not make any sense. If wrong period is given for a row, J computes the correct solution but not as efficiently as with correct periods.

If periods are given for rows, J is utilizing the tree structure of schedules in the optimization. This leads to smaller amount of additions and multiplications as the computation of the value

of a branch of the tree can for each node utilize the value of branch before that node. Unfortunately this was not more efficient e.g. in test problems with five periods.

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.

Kommentoinut [LR(44]: Repoon

Note 2: If optimization problem includes factories (see chapter 11.2 Optimization problem including factories), there have to be variables in the objective function or at least in one constraint row. Example of problem definition including factories can be found in chapter 11.12 JLP Examples.

Note 3: An ordinary linear programming problem contains only z-variables.

Note 4: It is not necessary to define `problem()` function if the problem includes only z-variables. In `jlp()` function you can use `zmatrix->` option instead of `problem->` option. For more information see chapter 11.8 Solving a large problem with z-variables: jlp( ).

Note 5: If the problem contains harvest/area constraints for several domains, it saves memory, if the constraints are written in form

```
harvest < area_of_domain*constant
```

instead of

```
(1/area_of_domain)*harvest < constant.
```

The latter formulation takes the number of domains times more memory than the former formulation.

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.

Note: all objects created by `problem()` function can be printed using commands `L=list(`*`problem`*`%?)` and `print(L)`

Parts:

`problem``%rows`
: is a text object containing the rows of the problem definition. Note: `jlp()` function creates vector *`output`*`%rows` containing values of constraint rows in solution.
Note: After version 3.0 J duplicates the rows if a row follows a domain specification row having several domain definitions.

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

A domain definition appears in the text object only once even if it has been in several places in the problem definition (see the `repeatdomains->` option above) If there are no domain definitions in the problem, then *problem*`%domains` contains row '`All`'.

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

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

problem`%rhs`
column vector containing the lower bound for each row. **Note**: a technical value is given for objective row. Recall the note for problem%rows.

problem`%rhs2`
vector containing the upper bounds for each row. **Note**: a technical value is given for objective row.

All coefficients of the constraints are in a packed format which the user cannot access..

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( )

There are two versions of `jlp()` function call: one with `problems->` option for problems defined by `problem()` function and the other with `zmatrix->` option for large ordinary linear programming problems with z-variable coefficients defined by matrix.

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

```
[=]jlp(problem->[,data->][,z->][,trans->][,subtrans->]
[,tole->][,subfilter->][,subreject->][,class->]
[,area->][,notareavars->][,print->][,report->]
[,maxiter->][,refac->][,slow->][,warm->][,finterval->]
[,fastrounds->][fastpercent->][,swap->][,test->][,debug->]
[,memory->])
```

Kommentoinut [LR(45]: Lisätty

Output:

Necessary for factory problems, otherwise optional. If output is given then function generates several matrices and lists associated with the solution (e.g. the values of the constraint rows, the shadow prices of the rows, the values of the z-variables, the reduced costs of z-variables, the sums of all x-variables of the data in all domains and their shadow prices, lists telling how problem variables are interpreted. See Objects for the JLP solution for more detailed description.

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.

Kommentoinut [LR(46]: Repon puolelle

`trans` transformation set which is executed for each unit.

`subtrans` transformation set which is executed for each schedule. **Note**: the subtrans transformations can utilize the variables in the unit data and the output variables of `trans->` transformations. **Note**: transformations already associated with cdata and xdata are taken automatically into account and they are executed before transformations defined in `trans->` or `subtrans->` options.

*** Later we may add the possibility to have several data sets (note that several files can be read into one data object in the data()function)

Kommentoinut [LR(47]: Repon puolelle

`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`.

`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, `9=` the pivot steps and the point of code where pivoting is done and the value of objective function are written to fort.16 file (or similar file depending on the operating system). The value `9` can be used in the case where jlp seems to be stuck. From fort.16 file one can then infer at what point debugging should be put on. Some cycling situations are now detected, so it should be rather unlikely that jlp is stuck.

`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).

`refac` after refac pivot steps the basis matrix is refactorized. The default value is 1000. New option since J3.0. Actually refactorization was present in the first version of J but it had to be dropped because the refactorization corrupted some times the factors of the basis matrix. The reason was found and corrected for J3.0. The reason for misbehavior of the refactorization algorithm of Fletcher was such that it never caused problems in ordinary linear programming problems for which Fletcher designed his algorithms.

`slow` if improvement of the solution in one round through all units is smaller than value given in `slow->` option, then J terminates under condition 'slow improvement'. New option since J3.0. Earlier slow improvement is computed from the tolerances of the problem, and if the `slow->` option is not present these tolerances are still used. If `slow->` option gives negative value, then the absolute value of the option indicates per cent change. Note that in large problems the solution is often very long time quite close the actual optimum, and hence the optimization time can be decreased with rather low loss in accuracy using the `slow->` option. If `slow->` option is give value zero it is equivalent to omitting the option and hence the slow improvement is determined from the tolerances.

`warm` If option is present in ordinary problems with x-variables then the key schedules of previous solution are used as the starting values of the key schedules. In factory

problems also the previous key factories are used as starting values. If there is no previous solution available or the dimension of the key schedules vector (number of treatment units) or the dimensions of the key factories matrix (number of treatment units and number of factories) do not agree with the current problem, warm option is ignored. Thus it is usually safe to use the option always, the only exception is that the factories and number of units do agree with the previous problem even if factories or schedule data are changes. The warm start may reduce solution time perhaps 40-80%.

`finterval` In factory problems the transportations to new factories are checked if the round number is divisible with finterval. The default value is 3. When there have not been improvements during the last round, the value is changed into 1.

`fastrounds`
The shadow prices are used to select an active set of schedules which are considered as entering schedules. fastrounds gives the number of rounds using the same active set. The default is 10. When there have not been improvements during the last round, all schedules are used as the active set.

`fastpercent`
A schedule belongs to the active set if its shadow price is at least fastpercent % of the shadow price of the best schedule. The default is 90.

`swap` If option is present, the schedules data matrix is written to a direct access binary file. This option may help if virtual memory overflow occurs. The data needs to be used from the file until all inquiry function calls are computed. Thereafter the data can be loaded again into memory using function `unswap()`. If this does not happen ordinary functions using this data object work a little slower. If a new problem is solved with the `swap->` option, there is no need to `unswap()` before that. In an Intel Fortran application the swap-> option is given without arguments. In a Gfortran application the option must be given in form swap->4. If there is shortage of virtual memory, read note 5 for `problem()` function before starting to use `swap->`.

Kommentoinut [LR(48]: Vain gfortran käännös?

`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`.

`memory` gives the amount of memory in millions of real numbers that can be used to store data needed in solving the problem. In factory problems also the xdata stored in a direct access file are loaded into memory as much as possible. It is not possible to figure out how large number memory option can give, so it must be determined with experimentation. Using `disk->` option in `data()` function and `memory->` option in `jlp()` function makes it in principle possible to solve arbitrary large problems. In practise the ability of double precision numbers cannot store accurately the needed quantities in very huge problems.

Kommentoinut [LR(49]: Lisätty. Oliko tämä mukana julkaistavassa versiossa?

Function `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 and if output is given then several output objects are created (see Objects for the JLP solution).

Note 1: a feasible solution (without an objective) can be found by minimizing a z-variable (remember `z->` option), or by maximizing a unit variable (which is constant for all schedules in a unit).

Note 2: If virtual memory overflow occurs, see first Note 5 for `problem()` function and then description of option `swap->`.

11.8. Solving a large problem with z-variables: jlp( )

When solving problems including only a large number of z-variables, it is possible to feed the coefficients as a matrix with `zmatrix->` option. Unit and schedule data (c- and x-variables) are not allowed when `zmatrix->` is used.

=jlp(zmatrix->,max->|min->[,rhs->][,rhs2->][,tole->]
[,print->][,maxiter->][,test->][,debug->])

Output:

Function `jlp()` generates output row vectors *output*`%zvalues`, *output*`%redcost` and output column vectors *output*`%rows`, *output*`%shprice` *output* is the name of the output.

Options:

`zmatrix`	Matrix containing coefficients of z-variables for each constraint row.
`max`	Vector containing coefficients of z-variables for the objective row of a maximization problem. Either `max->` or `min->` option has to be defined but not both.
`min`	Vector containing coefficients of z-variables for the objective row of a minimization problem. Either `max->` or `min->` option has to be defined but not both.
`rhs`	Vector containing lower bound for each constraint row. Value 1.7e37 is used to indicate the absence of the lower bound in a row. Either or both of the bound options (`rhs->`, `rhs2->`) has to be defined.
`rhs2`	Vector containing upper bound for each constraint row. Value 1.7e37 is used to indicate the absence of the upper bound in a row. Either or both of the bound options (`rhs->`, `rhs2->`) has to be defined.

Other options described above in chapter Solving a problem: jlp( ).

Note. When `zmatrix->` option is used, the solution is not automatically printed. Use jlp solution objects to access solution. For more information see chapter 11.10 Objects for the JLP solution.

Some refinements may be possible which make solution of zmatrix-problems faster. If this option will get some interest, we may try these improvements.

Kommentoinut [LR(50]: Repon puolelle

An example of the usage of `zmatrix->` option can be found in chapter 11.12 JLP Examples.

If an ordinary jlp-problem is solved using `zmatrix->` option, the solution takes much longer. For instance, a problem with 200 treatment units and 127264 schedules and 9 utility constraints took less than 1 second cpu time using `problem->` and `data->` options, but over 16 minutes when solved with `zmatrix->` option. A similar problem with 300 treatments units and 181640 schedules took 58 minutes with `zmatrix->` and 1.2 seconds with our algorithm utilizing the generalized upper bound technique.

11.9. JLP output

`jlp()` function prints information about problem, data and solution during problem solving. Messages are here described in order of appearance (self-explanatory messages are not included).

`tole = value`
The current value of `tole->` option. Value = 1.0 if option is not defined.

`number of domains = value`
The number of domains in problem definition.

`number of domain occurrences = value`
The number of domain definitions in problem definition, printed only if same domain definitions appear in several places of the problem definition.

`number of constraint rows in problem definition= value`
Note that when there are several domain definitions in a same row in the problem definition the number of rows is obtained by expanding the domain definitions and the subsequent rows.

Number of x-variables is printed for problems with data, and number of xkf and ykf-variables are printed for factory problems.

If in factory problem a constraint tells that the amount of timber assortment transported to a given factory has an upper bound but no lower bound is given, then the lower bound zero is assumed. The constraints affected will be printed. In principle amounts transported to factories will also otherwise be nonnegative, but for some reason which we do not completely understand the algorithm will reach slightly better objective values when the lower bounds are generated and these will become active.

```
***row, min,max,initial value, tolerance
rowindex  min_value  max_value  initial_value  tolerance_value
```

The smallest possible value (min_value) and the largest possible value (max_value) of each problem row including x-variables. Initial value is the value of the row when key schedules are selected. Factory variables (transported amounts to factories) and z-variables are ignored when determining minimum and maximum values. Minimum and maximum values can be used to detect infeasible rows at start, if there are only x-variables in the row and min_value > upper bound or max_value < lower bound of the constraint.

For rows containing only factory variables minimum and maximum values are omitted, but initial value (key schedules and key factories selected) and tolerance are printed.

Note: Row index 0 refers to the objective row.

`preoptimization round n improved infeasibility by value`
To find an initial solution close to feasible solution integer programming is applied prior to the linear programming algorithm. The preoptimization is done by going through all units once (before J3.0 twice)

`initial values`
`rowindex value`
Values after preoptimization for the rows including x–variables.

`*round,nonfeasible rows,pivots,obj` *`round rows pivots value`*
The number of pivot operations (pivots) performed, the value of the temporary objective function (value) and the number of nonfeasible rows (rows) in current solution after each round in optimization before reaching feasibility. The temporary objective function estimates the amount of infeasibility in the current solution. In one round optimization looks over all treatment units once.

`*round,pivots,obj` *`round pivots value`*
The number of pivot operations (pivots) performed and the value of the objective function (value) in current solution after each round in optimization.

`regular return`
JLP has found the solution and looked through all z-variables, residuals, slack and surplus variables, schedules and factory variables without improvement in the value of the objective function.

`slow improvement`
JLP has looked over all treatment units and in the last round found only a slight improvement, which does not significantly improve the value of the objective function. Note also the slow-> option for `jlp()` function.

`number of basic residuals (=nonbinding constraints)` *`number`*
`number of explicit basic schedules` *`schedules`*
`unit=` *`unit`* `key=` *`key_schedule`* `basic sched=` *`schedule`* `weight=` *`weight`*
The total number of basic schedules (schedules) and the weights (weight) of the basic schedules (schedule) in each split unit (unit). The weight of key schedule (key_schedule) in split unit can be calculated as follows:
weight of the key schedule = 1 – sum of the weights of the basic schedules in the unit.

In factory problems:

`number of split transportations`

Note: the number of basic residuals. the number of explicit basic schedules and the number of split transportations add up to the number of constraints.

Notes about the solution report

The double precision value of the objective function in the solution report may slightly differ from the single precision value of variable Objective (see 11.10 Objects for the JLP solution below).

If the lower and upper bounds of a constraint row have same value it is printed in the middle of columns 'lower bound' and 'upper bound'. The active lower or upper bounds are marked with 'L' and 'U' correspondingly.

Note: Owing to rounding errors, the constraint row value may be (slightly) illogical, e.g. the amount of timber assortment transported to a certain factory may be negative in a capacity constraint, or it may be tiny positive, and this can indicate that nothing is transported to that factory. Transportations to factories are computed differently in `xkf()` function, and there should appear no negative amounts of timber.

x-variables section of the report includes all x-variables in the data and in the transformations related to the data.

The column 'integer appr.-opt.' prints the difference between the integer approximation and the optimum. The integer approximation is obtained by selecting for each unit the schedules with largest weight.

11.10. Variables and objects for the JLP solution

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). This is single precision value and it may slightly differ from double precision the value of the objective function in the solution report.

If the `jlp()` function is used with `problem->` option, and there are z-variables in the problem, the z-variables get the optimal z-values found in the solution.

If the problem contains x-variables and there is only one domain, the x-variables get directly the sums of x-variables over schedules in the solution. These values will be changed in any operations with xdata, thus using *`output`*`%xsum` matrix is safer.

Objects created when there is output for the `jlp()` *function.*

Note 1: All objects can be printed with commands `L=list('output%?')` and `print(L)`

Note 2: column vectors can be accessed without specifying the column index (=1), but row vectors are matrices with one row and the row index (=1) must be given when accessing the values of the vector.

output`%rows`
: column vector containing values of the constraint rows. Note: `problem()` function creates a text object *problem*`%rows` containing the rows of the problem definition. The vector is generated if there are constraint rows in the problem.

Note 3: Starting from J3.0 the rows of the text object *problem*`%rows` and rows present in *output*`%rows` are in one to one correspondence except the first row of *problem*`%rows` refers to the objective row which is not present *output*`%rows`

Note 4: The value of the objective function is stored to the variable `Objective`.

output`%shprice`
: columns vector containing shadow prices of the constraint rows. The vector is generated if there are constraint rows in the problem.

output`%rhs`
: column vector telling the lower bounds of constraints. This is not equal to *problem*`%rhs` because objective row is included in *problem*`%rhs`. The vector is generated if there are constraint rows in the problem.

output`%rhs2`
: column vector telling the upper bounds of constraints. This is not equal to *problem*`%rhs2` because objective row is included in *problem*`%rhs2`. The vector is generated if there are constraint rows in the problem.

Objects created when there is output and `data->` *option*

output`%xvars`
: List of all variables found in xdata, the variables of cdata used in problem definition and output variables of subtrans –transformations. Note: if a variable is already in the data and the same variable is an output of subtrans-transformation, it appears twice in *output*`%xvars` but everything is still otherwise in order.

output`%xsum`
: Matrix which contains as many rows as there are domains in the problem definition. In each row there are sums of the x-variables over optimal schedules in the same order as the x-variables are in the list *output*`%xvars`. Note: domains can be seen using `print(`*problem*`%domains)`, and the number of domains can be accessed using `nrows()` function. If there is only one domain, the x-variables get also directly the sums of x-variables, but these values are easily lost when making operations with the data.

output`%xprice`
: Matrix which contains as many rows as there are domains in the problem

definition. In each row there are shadow prices of the x-variables in the same order as the x-variables are in the list *output*%xvars. The shadow price of a x-variable tells how many units the objective function would change if we get one (small) unit of the x-variable from an outside source. Shadow prices are nonzero only for those x-variables which appear in binding constraint rows.

output%xvarsproblem
: List of all x-variables found in the problem. These are among variables in *problem*%vars.

output%domains
: Column vector telling to which domain each constraint row belongs. The domain definition of constraint row irow can be printed with print('@*problem*%domains(*output*%domains(irow))'). The vector is generated if there are constraint rows in the problem.

output%problemrows
: Starting from J3.0 this is no more needed as problem rows and solution rows correspond each other.

Objects created when there is output and z-variables

output%zvalues
: row vector containing values of the z-variables . Note that if the jlp() function is used with problem-> option, and there are z-variables in the problem, the z-variables get the optimal z-values found in the solution.

output%redcost
: row vector containing reduced costs of the z-variables.

Object created when there is output and z-variables are defined using problem

output%zvars
: List of z-variables found in the problem. Note: in J2.0 this list was named as zvars%*problem*. The current naming is more logical as classification of problem variables is done in jlp() function, not in problem function. Variables in *output*%zvars are among variables in *problem*%vars.

Object created in factory problems (output required)

output%factories
: list stores factories found in optimization problem. In version J2.0 the list was named factories%*problem*.

output%xkyk
: list stores (transported) variables found in optimization problem. In version J2.0 the list was named xkyk%*problem*.

Note 5: The indexes written by the xkf() function correspond to the locations of factories and x-variables in these lists.

11.11. Inquiry functions for the JLP solution

The following J functions can access the most recent solution.

[=]weights()

Output:

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)

Output:

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

Argument:

`i` numeric value between 1 and `weights()`

[=]schedcum(i)

Output:

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

Argument:

`i` numeric value between 1 and `weights()`

[=]schedw(i)

Output:

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

Argument:

`i` numeric value between 1 and `weights()`

[=]weight(i)

Output:

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

Argument:

`i` numeric value between 1 and `weights()`

[=]partweights()

Output:

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 `part()` 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 we are not sure if wel keep this

Kommentoinut [LR(51]: Repon puolelle

[=]partunit(i)

Output:

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

Argument:

`i` numeric value between 1 and `partweights()`

[=]partschedcum(i)

Output:

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

Argument:

`i` numeric value between 1 and `partweights()`

[=]partschedw(i)

Output:

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

Argument:

`i` numeric value between 1 and `partweights()`

[=]partweight(i)

Output:

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

Argument:

`i` numeric value between 1 and `partweights()`

[=]price%unit(iunit)

Output:

Returns the shadow price of the unit iunit.

Argument:

`iunit` 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->])

Output:

Returns the weight of a schedule.

Argument:

`sched` 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)

Output:

Returns the shadow price of a schedule.

Argument:

`sched` 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)

Output:

Returns the shadow price of a schedule within a unit

Arguments:

`iunit` the number of the unit

`sched` the schedule number within the unit

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

Output:

Returns 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)

Output:

Returns the within-unit schedule number of the schedule which has the largest weight within the unit

Argument:

`iunit` the number of the unit

[=]integerschedcum(iunit)

Output:

Return the cumulative schedule number of the schedule which has the largest weight within the unit

Argument:

`iunit` the number of the unit

[=]xkf(file[tole->][,print->])

Prints for each unit the amount of variables transported to each factory. Output consists of four numeric values:

- index of the unit
- index of the forest variable
- index of the factory
- the transported amount.

Factory and forest variable indexes refer to the elements of *`output`*`%factories` and *`output`*`%xkyk` lists. The file is closed automatically after writing. The file can be open before `xkf()` function (so one can write e.g a header to this file). In this case `xkf()` writes just after the previous text. The file can be read into a data object following the example below. Note: if one-line header is written into the file, `data()` function requires `readfirst->` option.

Output:

Number of records written

Argument:

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

Options:

`tole` only such transported amounts are printed which are greater than the value given in tole. The default value is zero. Note that the mathematics of the linear programming works so that there are values which are practically but not exactly zero, so it may be a good strategy not to print these practically zero values. It may be that these close to zero amounts do not indicate reasonable transportation routes.

`print` indicates that the timber assortment names and factory names are also printed to the file

An example of using `xkf()` function:

```
;xkf:
xkf('fout.txt')
xkfdata=data(in->'fout.txt',read->(iunit,xk,fact,amount))
outf=matrix(len(f%factories),len(f%xkyk))
trf=trans()
outf(fact,xk)=outf(fact,xk)+amount
/
stat(amount,data->xkfdata,trans->trf)
! stat() function makes that data is gone through and
! trf-transformations are done for each observation
print(outf)
An example of using xkf() function in domain problems:
xkf('fout.txt')
xkfdata=data(in->'fout.txt',read->(iunit,xk,fact,amount))
ndom=nrows(newprob%domains)
;do(id,1,ndom)
doutf"id"=matrix(len(f%factories),len(f%xkyk))
;enddo
trf=trans(input->)
getobs(cdata,iunit)!cdata is the unit data object
;do(id,1,ndom)
if(@newprob%domains(id))doutf"id"(fact,xk)=
doutf"id"(fact,xk)+amount
;enddo
/
stat(amount,data->xkfdata,trans->trf)
print(doutf?)
```

Note: `xkf()` computes timber transportations differently from the LP solution that is computed in capacity constraints for the general report or for *`output`*`%rows` matrix; thus the results may differ slightly owing to rounding errors.

11.12. JLP examples

11.12.1. Example 1: Definition of a problem including factories

See include file jlp.txt found in j_examples.zip for almost real life example.

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 Lahti%xcoor 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((@sawmill(i)%xcoor-sxcoor)***2 + (@sawmill(i)%
ycoor-sycoor)***2)
!use the factory price and the discounting factor
;do(j,1,p)
util%%sawlog#"j"%%@sawmill(i)=( @sawmill(i) sprice%-cost) *df#"j"
;enddo
;enddo
/

trans.util%%pulp%%pulpfactory=trans()
;do(i,1,len(pulpfactory))
cost=costpkm*sqrt((@pulpfactory(i)%xcoor%-sxcoor)***2 +
(@pulpfactory(i)% ycoor-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

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

11.12.2. Example 2: A problem with z-variables

Version 1: Coefficients of z-variables defined in `problem()` function.

```
luen=problem()
3*x1+x2+3*x3==max
2*x1 +x2 +x3 <2
x1+2*x2 + 3*x3<5
2*x1+2*x2+x3<6
x2<1.7E7
/

jlp(problem->luen,z->)
```

Version 2: Coefficients of z-variables defined with `zmatrix->` option.

```
a=matrix(4,3,in->)
2,1,1
1,2,3
2,2,1
0,1,0
/

obj=matrix(1,3,in->)
3,1,3
/
```

```
ub=matrix(4,in->) !note column vector can be read like this
2,5,6,1.7E7
/

out=jlp(zmatrix->a,max->obj,rhs2->ub)

print(out%zvalues)
print(out%redcost)
print(out%shprice)
print(Objective)
```

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 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 are 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
!one can kill or make new trees so number may change
ntrees#"t"= ntrees#"t-1"
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:

`simulator1,…,simulatorn`
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 arguments, 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 [*the name of the output-variable*] `//'%unitdata'`

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

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

```
age#0=40

vol#0=50
```

Kommentoinut [LR(52]: Tarkistettava yhtenäinen käytäntö (hipsuilla vai ilman)

Kommentoinut [LR(53]: Onko hakasulul osa nimeä?

```
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 or also directly from menu. The file can be edited to get proper legends etc. The R figure can be saved in several formats.

Note: The axes can be dropped from a R figure by adding axes=FALSE, ann=FALSE to the initial plot R function. This way J/R can be used to draw any geometric graphs.

The graphics J functions are:

Scatterplot: `plotyx()`

[=]plotyx(yvar,xvar[,data->][,mark->][,xrange->][,dx-]
[,yrange->][,dy->][,append->][style->][,width->][,color->]
[,show->][,r->])

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

Output:

a figure object, default `Figure`

Arguments:

`yvar` y-variable

`xvar` x-variable

Options:

`data` data sets

`mark` character or numeric value used to plot observations, default `'.' Mark '/'` means that no mark is plotted. This is useful with `style->` option.

`xrange` `xrange->(xmin,xmax)` defines the minimum and maximum of the x-axes. Default is the observed minimum and maximum of x-values. If an explicit range is given, it remains part of the figure object, and points falling outside the given

Kommentoinut [LR(54]: Kuuluvatko uloimmat hipsut syntaksiin?

range will be outliers in subsequent subfigures appended to the figure. If no explicit range is given, the printing range is modified if new observations fall outside the previous range. It is possible to give only xmin or both xmin and xmax.

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

`yrange` `yrange->(ymin,ymax)` works similar to `xrange`

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

`style` points are connected with lines drawn with style given in `style->` option. `0`=no line (default), `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)

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

`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.

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 utilized through value function, e.g. `func->(value(ss,y))` where `ss` is transformation object and `y` variable getting value in `ss`.

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

`xrange` `xrange ->(xmin,xmax[,xmin2,xmax2])` defines minimum and maximum of the x-axes. 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

If `x` is function of `y`:

yrange	yrange->(ymin,ymax[,ymin2,ymax2]) defiens 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 (default), 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 (character or numeric value) 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.
delete	If fle name is given in r-> option and the file with that name already exists, then it is deleted without asking, if this option is given.

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][,maxlines->]
[,append->][,style->][,xrange->][,yrange][,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:

`x1,…,xn,y1,…yn`
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:

`maxlines` if points are given in a matrix or in two vectors, then this option gives the number of points used (thus actually the number of lines is maxlines-1).

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

`xrange` `xrange->(xmin,xmax)` gives the range of the x-variable

`yrange` `yrange->(ymin,ymax)` gives the range of the y-variable

`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` character or numeric value 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` and 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->])

Kommentoinut [LR(55]: Tarkistettava yhtenäinen välilyöntikäytäntö läpi dokumentin

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` `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` `yrange->(ymin,ymax)` defines the y-range for the figure.

`histogram`
histogram is produced. This is the default if classify function did not have any arguments.

`area` the histogram is scaled so that the area under the histogram is one making overlaying the histogram and a density function easy.

`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 (character or numeric value) 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.

Kommentoinut [LR(56]: Hipsuilla vai ilman? yhtenäinen käytäntö läpi manuaalin

`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:

`fig` a figure object to be written into filer `jfig.jfig` so that the program `Jfig` can then show the figure.

Options:

`xrange` `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` `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.

Kommentoinut [LR(57]: Repon puolelle

14. Stem curves, splines and volume functions

J will contain many tools for handling stem curves. Currently there are the following functions available.

Kommentoinut [LR(58]: Repon puolelle...

=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->])

Kommentoinut [LR(59]: = vai [=]

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])

Kommentoinut [LR(60]: = vai [=]

Volume functions of Laasasenaho (1982).

Output:

Volume in litres.

Arguments:

`species` `1` =Scots pine, `2`= Norway spruce, `3` and `4` = birch, `9`=larch (not available when dbh is the only measured dimension)

`dbh` diameter at breast height, cm

`d6` diameter at 6m, cm,

`h` height, m

=laaspoly(species,dbh[,d6],h)

Polynomial stem curve 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` diameter at breast height, cm

`d6` diameter at 6m, cm,

`h` height, 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.

Kommentoinut [LR(61]: Repon puolelle

=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()` function.

The taut spline algorithm is published by de Boor (1978) on pages 310-314. The source code was loaded from Netlib.

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.

Argument:

`charval` a character variable or character constant.

15.2. Current include file

The name of the current include file is returned as a character variable by:

[=]thisfile()

This is useful when defining shortcuts for commands that include sections from an include file. Using this function the shortcuts work even if the name of the include file is changed. See file `jex.txt` for an application.

15.3. 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()

Output:

the elapsed time since midnight in seconds

[=]secnds(t)

Output:

the elapsed time since midnight `-t`.

Argument:

`t` seconds

[=]cpu()

Output:

the first call gives the cpu time since starting the program in seconds

[=]cpu(t)

Output:

the total cpu time `-t`

Argument:

`t` seconds

15.4. List functions

[=]list(obj1,…,objn[,mask->])

Defines an object list

Output:

a list object

Arguments:

`obj1,…,objn`
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 objects 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

Output:

a list object

Arguments:

`obj1,…,objn`
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. The output can exceptionally be one of the inputs.

[=]difference(list1,~~list2~~obj2)

Defines a list dropping from list `list1` obj2 if it is not a list and all objects found in ~~list~~ obj~~list~~2, if it is alist.

muotoili: Fontti: (Oletus) Courier New

muotoili: Kappaleen oletusfontti

Output:

a list object

Arguments:

`list1` list

~~,~~obj~~list~~2 list or objects

[=]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).

Output:

index of the variable `object` in the list

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. An 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.

[=]len(list[,any->])

Output:

the number of elements in the list

Argument:

`list` 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

15.5. 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.

Output:

a single numeric value.

Arguments:

`object` J object which can be used as a function. If `object` is a list and there is option `index->` then the object is picked from the list.

`xvalue` the value used as the argument for the object which can be used as a function

Options:

`options` see special cases below.

There are the following special cases.

15.5.1. Interpolating a regular matrix: value(matrix,x)

[=]value(matrix,xvalue[,row->])

Interpolates linearly rows of matrices.

Output:

a single numeric value or matrix of interpolated rows.

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.

Kommentoinut [LR(62]: Repon puolelle

15.5.2. Interpolating a classify-matrix: value(cl_matrix,xvalue)

[=]value(cl_matrix,xvalue)

Interpolating a matrix produced by `classify()` function.

Output:

a real value

Arguments:

`cl_matrix` a matrix produced by `classify()` function

`xvalue` value of the x variable used for computing class means.

15.5.3. Using a spline: value(spline,xvalue)

[=]value(spline,xvalue)

Gets values from a smoothing spline or a stem curve spline

Output:

a real value

Arguments:

`spline` a spline generated by `smooth()` or `stemspline()` function

`xvalue` argument of the spline.

**later other types of splines will be available

Kommentoinut [LR(63]: Repoon

15.5.4. Getting values from a transformation set: value(tr_set,xvalue)

[=]value(tr_set,xalue[,arg->][,result->])

Output:

a real value or J object returned by transformation

Kommentoinut [LR(64]: Omasta päästä, onko oikein?

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 an argument whose value need to be bypassed simultaneously.

15.5.5. Getting value of a list variable

`[=]value(list,index)`

Output:

a real value or J object

Kommentoinut [LR(65]: Omasta päästä,, onko näin

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

Kommentoinut [LR(66]: Repoon

15.6. 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:

15.6.1. 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

15.7. 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.

Output:

a real value

Arguments:

`x0,x1,x2,y0,y1,y2,x`
numeric values

Note. The argument `x` need not be within the interval of given x values (thus the function also extrapolates).

15.8. Interpolating using a plane: plane()

[=]plane(x1,x2,x3,y1,y2,y3,z1,z2,z3,x,y]

The function computes the equation of plane going through the three points (`x1,y1,z1`), etc and computes the value of the z-coordinate in point (`x,y`). The three points defining the plane cannot be on single line.

Output:

a real value

Arguments:

`x1,x2,x3,y1,y2,y3,z1,z2,z3,x,y`
numeric values

15.9. Bilinear interpolation: bilin()

[=]bilin(x1,x2,y1,y2,z1,z2,z3,z4,x,y]

`z1` is the value of function at point (`x1,y1`), `z2` is the value at point (`x1,y2`), `z3` is the value at (`x2,y1`) and `z4` is the value at (`x2,y2`): the function is using bilinear interpolation to compute the value of the z-coordinate in point (`x,y`). The point (`x,y`) needs not be within the square defined by the corner points, but it is good if it is. See Press et al. ? (or Google) for the principle of bilinear interpolation.

Kommentoinut [LR(67]: Täsmennys

Output:

a real value

Arguments:

`x1,x2,y1,y2,z1,z2,z3,z4,x,y`
numeric values

15.10. Integrating a function

*** Later there may be several forms of integrate() function. In j3.0 the only form of `integrate()` function is:

Kommentoinut [LR(68]: Repoon

15.10.1. Integrating stem curve to get stem volumes

`[=]integrate(stem_spline,h1,h2)`

Output:

volume (dm3) 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 π*0.25*`value(stem_spline,h)`^{2} and the result is then divided by 10000 to get the result in dm3 (in stem splines both height and diameter are in cm)

15.11. 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

Note: 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

Note 1: 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 2: giving value zero to a real variable clears all bits.

[=]getbit(ind,bit)

Gets the value of a bit position.

Output:

If the bit is on, then the value of the `getbit()` function is 1 (True), otherwise zero (False).

Arguments:

`ind` a real variable or a list of real variables (do not expand by `@`)

`bit` bit position to read

Note 1: 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 2: 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:

```
a=0
setbits(a,2,5,7)
v=getbitch(a)
print(v)
 v is text object:
 01001010000000000000000000000000
```

=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)

Output:

the number of rows in a bitmatrix

Argument:

`bitmatrixobj`
: an object created by `bitmatrix()` function

[=]ncols(bitmatrixobj)

Output:

the number of column in a bitmatrix

Argument:

`bitmatrixobj`
: an object created by `bitmatrix()` function

=closures(bitmatrixobj)

To get neighborhoods indicated by a bitmatrix

Output:

a bitmatrix

Argument:

`bitmatrixobj`
A symmetric square (1:n,1:n) bitmatrix where ith row indicates all the neighbors of ith point.

Note 1: 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
```

Note 2: The algorithm in closures is not well tested.

15.12. 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:

`var1,…,varn`
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 subject name, then '`%`' and then the generic variable name.

Example:

```
properties(capacity,xkoor,ykoor)
rauma, 100, 64,78
pori, 30, 67,89
/
```

Defines variables `rauma%capacity`, `rauma%xkoor`, `rauma%ykoor`, `pori%capacity`, etc.

15.13. Storing values of variables

=store(var1,…,varn)

Stores the values of variables.

Output:

a storage object

Arguments:

`var1,…,varn`
variables to be stored

Note: A variable list may be again nice when defining the arguments.

load(storage)

Loads back the values of variables.

Argument:

`storage` 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.

Kommentoinut [LR(69]: Repoon

15.14. Storing matrix

Kommentoinut [LR(70]: Lisätty

store(matrix)

Stores a matrix into a direct acces file *matrix*.bin and the accompanying information file *matrix*%info.txt. The matrix can be used from the disk similarly as earlier.

Argument:

`matrix` matrix object to be stored

load(matrix)

Loads the data from direct access file back to memory.

Argument:

`matrix` matrix object to be loaded from the file *matrix*.bin and the accompanying information file *matrix*%info.txt.

15.15. 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)`

Note 3: 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)

Output:

the list of all objects loaded (default for the output is `Result` as usually)

Argument:

`filename` character variable or character constant, the name of the file created by `save()`.

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 names of the first ten objects will be printed, as well as the total number of replaced objects.

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. J as a server

Using the `wait->` option in `read()` or `;incl()` functions one can make J as a server which makes certain computational tasks and provides them to the client application. Of course J can be also client application but probably this is not often needed. Lets show how J can be used to calculate the sums of two variables and to provide the answer to the client. It is supposed that J is started first. There can be several number pairs in a batch. The `j.par` file could look like this:

```
!5000
!the file input.txt is used to transmit the numbers from the client
!when J is started there may be an old input.txt available which is
first deleted.
if(exist('input.txt'))delete('input.txt')
calc=trans())
do(i,1,10000000)
read('input.txt',$,x1,x2,eof->finito,wait->)
print(x1,x2)
!if(x1.lt.-1.e30)return there may be condition for stop computations
if(finito)then
! input.txt  is closed automatically
delete('input.txt')
close('output.txt')

cycle
endif
write('output.txt',$,x1+x2)
enddo
/

call(calc)
```

The client application is writing the file input.txt and closes it. Then it is trying repeated to open the file `output.txt`. When it can open it, it reads the results, and writes new input.txt.

If the client is not transmitting just numbers but also J commands, the `j.par` file could look like:

```
!5000
;if(exist('input.txt')delete('input.txt')
;do(i,1,10000000)
print('waiting for input.txt')
;incl('input.txt',wait->)
;if(ok);then
delete('input.txt') ! now the client can see that input.txt
! is deleted and it can read the possible output files
;else
print('everything is not ok')
whatnext=ask(q->'0=end 1=ready to continue with new input.txt')
;if(whatnext.eq.0)end
```

```
;endif
;enddo
```

17. Error debugging and handling

17.1. User errors detected by J

If J detects an error made by the user then the following information is provided:

- J prints the current 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. The source code of a transformation set transf is stored into text object *`transf`*`%source`. Usually the error line can be seen in the include-file, but if input programming methods are used then the generated source code and include file are not equal.

After an error the control returns to `sit>` level.

Note: when error message starts with '`*j*`, then it is an error made by J, see next subchapter.

If the limit for the maximum number of named objects is encountered, then J stops. See Set up of J how to proceed.

When J is started again several times it may be wise to run J from the command prompt regularly as the history of the working process can be seen from the command prompt window, and arrow keys can be used to access input lines from earlier runs.

The error printed in the command prompt window can be 'insufficient virtual memory'. This is not an error of J, J just tries to allocate more memory than there is available. On a Windows 32-bitoperating system a Win32 application cannot use more than 2 GB of memory. This is by design from Microsoft. Possible solutions to this problem are:

1. if there are many open applications, close unnecessary applications (if the cause of the problem is the total available virtual memory, not the 2 GB of memory limit of one application)
2. delete unnecessary J-objects
3. reduce the size of the data
4. when reading large data files, use `keep->` option to store only the necessary variables in the data matrix
5. when making new variables use in `data()` function use `trans()` option instead of `maketrans->` option so that the new variables are not stored in the data matrix
6. increase the size of the virtual memory.

Note: J is in principle protected against trying to put data above the allocated dimension limits, but there may be errors, in principle at least.

17.2. J does not work correctly

Error messages starting with '*j*' indicate programming errors which should be reported to the authors. After version 3.0, they may result also from programming errors of own-packages. Often the error is actually caused by the error of the user, but J does not check user input carefully enough at higher level but the error is detected at lower level utility subroutines. If the user defines `$Crash=1` then system crash is caused and the calling sequence can be seen if J is run in command window and it is compiled with such options that error debugging reports are produced.

If J crashes and J window disappears, run the debug version of J on command prompt window and send to the authors the debug information printed out to the console. Select first the proper disk (e.g. C:) and select then the working directory using cd-command (e.g. cd work1/sub1). If the exe-file is in the working directory write the name of the exe file (the debug version), and if the shortcut is in the directory type the shortcut name and add '.lnk' extension before sending the command. The shortcut name cannot contain spaces (which the system may have generated).

Some 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 authors.

17.3. 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:

`obj1,…objn`
: 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.

Arguments:

`obj1,…objn`
J objects

Kommentoinut [LR(71]: kappaleen muotoilulle pitää tehdä jotain

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:

`traceset` 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:

```
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= tr%source
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()` function for another example of `tracetest()`.

18. 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. Antti Mäkinen and Jussi Rasinmäki from Simosol company have helped to make the code more portable by using different compiler than used originally for the development.

19. 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. 1997. Dense factors of sparse matrices. In: Approximation Theory and Optimization. Eds. M.D. Buhmann and A. Iserles. Cambridge University Press. ISBN 0 521 58190 7

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, nonparametric stem-curve prediction method. Can. J. For. Res. 36:1017-1027

Lappi, J & Lempinen, R. 2014. A linear programming algorithm and software for forest level planning problems including factories. Scandinavian Journal of Forest Research. DOI: 10.1080/02827581.2014.886714

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.

Woltring, H.J. 1986.A Fortran package for generalized, cross-validatory spline smoothing and differentiation. Adv. Eng. software 8(2):104-113.

20. Index

In the following index J functions and statement are in lower case, other entries are in the upper case..

luke.fi

Natural Resources Institute Finland
Viikinkaari 4
FI-00790 Helsinki, Finland
tel. +358 29 532 6000