

Juha Lappi

J -users' guide

Version 0.9.4 February, 2005

**Finnish Forest Research Institute
Suonenjoki Research Station**

Lappi, Juha. 2004. **J-users' guide. Version 0. 9.4.** Finnish Forest Research Institute, Suonenjoki Research Station 102 pp.

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

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

Author's address: Juha Lappi, Finnish Forest Research Institute, Suonenjoki Research Station, Juntintie 154, FIN-77600 SUONENJOKI, Finland

Publisher: The Finnish Forest Research Institute Metla, Unioninkatu 40 A, FIN-00170 HELSINKI, Finland

Metla project number: 3285

(URL: <http://www.metla.fi/products/J/J-userguide.pdf>.)

Copyright 2003 Finnish Forest Research Institute. All Rights Reserved.

The **J** users' guide is provided without warranty of any kind. It may include inaccuracies or errors. The author may make improvements and/or changes in the products at any time. These changes will be incorporated in the new editions of the **J** users' guide. The most recent users' guide is available as a Word document at download pages. The pdf file available at the public web page is updated less frequently.

The distribution versions of **J** software may deviate in some details from the general documentation presented in the **J** users' guide.

The names of companies and their products appearing in the **J** users' guide are trademarks or registered trademarks of their respective holders

Conditions of use of the software

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

The software is available at the web page <http://www.metla.fi/products/J/> which contains the conditions of use, user registration, and download page for registered users.

Recent changes

Changes after baseline version 0.6

V=0.9.4

Density function pdf() and cumulative distribution function cdf (currently only for normal distribution)

There are new debugging tools(see ;trace ;traceoff, tracetest and trace in Tracing variables chapter)

Using only one index to indicate a matrix element is now allowed only for column vectors (earlier one index indicated element number when matrix was ordered in row order).

Matrix function has form option which makes matrix function to read the matrix row by row, and it makes possible to add comments on the rows.

Functions trans and simulator store now by default all the source code into a text object source%.. Option source->0 indicates that source is not saved. In case of error in executing transformations the line generating the error is printed. Printing a text object with print function has now option row-> which make it possible to print only one line from a text object or a range of lines. Function write can be used to write a text object into a file. In case of error J tells also how many lines it has read from a include file. See error debugging for utilizing these new features.

If only one value is given in values option of the matrix function, then all elements get this value. If there is diagonal option then the matrix is made diagonal. Matrix functions can also used to set values for a existing matrix using values option.

There is new submatrix function which makes a new matrix from a previous matrix.

Arithmetic functions with several output variables are removed. Instead there is possibility to do arithmetic operations using lists. See list arithmetics.

Width option for draw and drawline functions.

In smooth function weight option is replaced by variance option.

If the readfirst option of data function is given without arguments, then the first line in the data file is interpreted as a header which is printed.

New sort and print options in stemspline function

Function drawline can have matrix arguments.

If b is a general object assignment a= b makes a copy of b into object a. Function show has now options yrange->, xrange-> and color-> which make it possible to show the same figure object in different ways. Smooth-function has new options maxiter and iterations. Stat-function has new option nobs to get the number of observations, and the functionality of other options have been improved New transfer option for handling data (transforming only accepted observations).

The operation of J is made easier by adding new function up() by which the console window is scrolled to point of last ;incl command. ;incl has new option clear-> which determines how much space is reserved to output (if there is not space enough up() does not go up enough). New function sqrt2 which works with nonpositive values.

Derivatives of any function can be computed using der- function. There is no more any default for x variable in draw function. New option any-> for len function and for index function for matrices. The value function and result function can pick the

function object from an object list if the first argument is list and there is index option which indicates which function object is used.

Elementsum has options row-> and column-> which make it possible to sum column element or row elements.

V-0.9.3

weight-> option in stat –function. Some bugs corrected. More stem curves functions (not yet reported here). Having only one point in drawline –function makes it possible to place text anywhere in the graph. The text is transmitted also to a R script using r-> option. If a comment line in an include file starts with '!', it will be printed. The data set given in data option in jlp-function can be also the schedules data (that is also the output of simulate-function). If a input line ends at '+', '*', '=', '(' then the next line is also interpreted as continuation line (in addition to the ','). New version of jfig.exe. Functions ask and askc have now option exit which makes it possible to terminate reading commands from a file.

V-0.9.2

Several bugs are corrected, and recovery from user errors improved. sign-function added. err-option in stempolar function to provide error debugging in case of problems. Similar option will be added to several other functions (user requests welcome). Setting a feasible range in smooth function improved. Constraint option in smooth changed into wish option. New function getobs by which one can access an complete observation in a data set.

V-0.9.1

Functions save and unsave for saving objects into binary files and loading them back.

V-0.9.0

Principles for controlling printing defined. Not yet completely fulfilled. See chapter 'Controlling output'.

New option repeatdomains-> in problem function and option z-> in the jlp function.

V-0.8.9

Function jlp has new options area-> and notareavars-> which make it possible to use schedule data which contain per area values. Jlp() has also options trans-> and subtrans-> which make it possible to link transformations to cdata or xdata temporarily. Function value(bit_matrix,row,col) has new option any-> which indicates value to return 0 when trying to access out of range row or column. Function simulate has option selector-> which makes it possible to use different simulators in different stands depending on the stand conditions. There is new function difference() which is making a list from a list by dropping elements of another list.

V-0.8.8

Some bugs corrected. Jlp-function returns value for logical variable Started_jlp which indicates if jlp inquiry functions can be used.

V-0.8.7

Some bugs corrected, and recovery from some user errors improved.

V-0.8.6

New function text which can be used to make text objects, and ;incl function can then use the text object similarly as a file. Detection of syntax errors in problem function improved.

V-0.8.5

New function result, which helps to use transformations as functions

V-0.8.4

In graphics or classify functions the distance between main ticks is taken into explicit options `dx-` `dy->` and `dz->` instead of using third argument in `xrange`, `yrange` and `zrange` to define `dx`, `dy` and `dz`. `trans-` function has now new options `arg->` and `result->`, and value function is changed to utilize these which makes definition of function like transformation sets more natural.

V-0.8.3

Closures-function improved, draw function has now new options `y->` and `points->`, `y-` option makes it possible to draw `x` as a function of `y` and `points` option makes it possible to determine the number of points included in the graph.

V-0.8.2

A new bitmatrix function 'closures' for making neighbourhoods

V-0.8.1

New integer option for functions `weight%schedw` and `weight%schedcum` for accessing the integer approximation for the JLP solution. New function `integerschedcum` and `integerschedw` for the same purpose.

V-0.8.0

Improved control for printing, new options for smoothing splines, new functionality for `index()`, `value()`, new object type bitmatrix, temporary association of transformation set with a data set (`trans->` option with all functions using data) stem curve functions. It is possible to write figure objects into a file which can be used by R software to generate publication level graphics. (This manual is not completely up to date, I try to update it soon). This version requires new `jfig.exe` to show figures.

V-0.7.8

It is possible to define `j.par` so that the io-window will close by terminating the program with `end` command.

V-0.7.8

Bugs corrected (second use of the same simulator failed, nested `;()`; then structures did not work, parsing of one transformation type failed, `'!` comment in input programming lines did not work)

V-0.7.7

Bugs corrected (in `dot()`, `which()`, extra space in `if()` then). If the output of arithmetic operation or numeric function has been a general object, the object is first deleted, and the output gets proper type (real variable). In earlier versions the object had to be deleted explicitly before a general object could become again a real variable.

V-0.7.6

`, ;elseif();then` did not work, corrected.

function `jlp()` has option `tole->` which was not described in the manual:

A corrected version of training file `jex.txt`.

V0.7.5

Bugs corrected. A bug corrected also in `jfig.exe`.

`print` function has now also `data-` option which makes it possible to print all observations from a data object (including those made by transformations)

The name of variable `Buffer` used in writing is changed into `$Buffer` which directly indicates that it is not put into the list of output variables.

V0.7.4

Tree level simulation is possible in a simulator using `treevars` option. New matrix functions `elementsum` and `dotproduct`. Additional spaces in some special contexts caused errors. Now I have tried to correct these. A bug in the optimization related to rejection of schedules is corrected.

V0.7.3 October 2003

Option matrix added to trans and simulator. Assignment a=b works also for matrices.
Bugs corrected.

V0.7.2 October 2003

Function branch added for the simulator

Functions showdir and setdir. added

Set-up of J (file j.par) changed.

V0.7.1 October 2003

Function classify and properties added

list elements can be accessed (see Shortcuts for object lists)

V0.7 October 2003

Graphics requires new version of the jfig-program.

Bit functions added. Filter option added to jlp-function. Function drawline added.

Color and style options added also to draw function. Function value can be used to interpolate matrices and to return a value from transformations. Bugs corrected.

V0.6.5 September 17, 2003

Stability of the optimization improved. The jlp-function has now option
sparse->

which will indicate the use of sparse matrix routines in the optimization. In problems with many domains this may improve the computation time significantly. With dense problems sparse computations are slower

V0.6.04 August 21, 2003.

The stability of the optimization algorithm improved.

len(list) returns the number of elements in the list

New function: sort.

V0.6.03 August 8, 2003.

Tolerance parameters of jlp-function fine tuned.

The debug option of jlp can now have several values which indicate turning on and off

the debug mode., e.g., debug->(0,100,200) means that debug is initially on then off at pivot step 100 and then again on at step 200.

V0.6.02 August 4, 2003.

Computation in the lp-function jlp changed into double precision

V0.6.01 July 29, 2003 some bugs corrected and recovery from user errors improved

Table of contents

	Conditions of use of the software.....	2
	Recent changes.....	3
	Table of contents.....	7
1	Introduction.....	15
	1.1 Hardware requirements.....	16
	1.2 Software requirements.....	16
	1.3 Set up of J.....	16
	1.4 During the first use.....	17
	1.5 Manual conventions.....	18
2	J objects.....	18
	2.1 Object names.....	18
	2.2 Copying object: a=b.....	19
	2.3 Deleting objects: delete().....	19
	2.4 Object types.....	19
	Real variables and constants.....	19
	Character constants and variables.....	20
	Text object.....	20
	Logical values.....	20
	Object lists.....	21
	Matrices and vectors.....	21
	Transformation set.....	21
	Simulator.....	21
	Data set.....	21
	Problem definition object.....	22
	Figure object.....	22

	Function objects.....	22
	Storage for variables.....	22
	Bitmatrix.....	22
	Trace set.....	22
2.5	Objects created automatically and default names.....	23
3	Command input and output.....	24
3.1	Input line and input paragraph.....	24
3.2	Input programming.....	25
3.2.1	Addresses in input programming.....	25
3.2.2	Changing "... " sequences.....	25
3.2.3	Input programming commands and control structures.....	26
	;incl([file][,from->]).....	26
	;return.....	27
	;do(i,start,last[,step]).....	27
	;enddo.....	27
	;if(...).....	28
	;if(value);then.....	28
	;elseif(value);then.....	28
	;endif.....	28
	;goto('adr').....	28
3.2.4	Utilizing object lists: @list and @list(elem).....	29
3.2.5	Shortcuts for implicit object lists: x1...x5, ?%x1.....	30
3.2.6	Key shortcuts.....	30
3.2.7	Defining a text object with text function and using it in ;incl.....	31
	=text().....	31
3.3	Immediate operations starting with '!.....	31
3.4	Controlling output.....	31
4	J transformations.....	32
4.1	Structure of general J functions.....	32
4.2	Common options.....	33

	in->	33
	data->	34
	trans->	34
	err->	34
4.3	J function for defining transformation sets: trans()	34
4.4	Executing a transformation set explicitly: call()	35
4.5	Using a transformation set as a function: result()	36
4.6	Using a transformation set as a function with an argument: value()	36
5	Arithmetic computations	37
5.1.1	Standard numeric expressions	38
5.1.2	Logical and relational expressions	38
5.1.3	Arithmetic functions	38
	sqrt, exp, log, log10, abs	39
	Real to integer conversion	39
	min, max	39
	sign	39
	dot(c1,...,cn,x1,...,xn)	39
	which(cond1,value1,...,condn,valuen[,valuedef])	39
	Trigonometric functions, argument in radians	39
	Trigonometric functions, arguments in degrees	40
	Inverse trigonometric functions, result in radians	40
	Inverse trigonometric functions, result in degrees	40
	Hyperbolic functions	40
5.1.4	Probability distributions	40
	pdf(x[,mean][,sd])	40
	cdf(x[,mean][,sd])	40
5.1.5	Random numbers	40
	ran(seed)	40
	rann(seed)	41

5.1.6	Special numeric functions.....	41
	npv(interest,income1,...,incomen,time1,...,timen).....	41
5.1.7	List arithmetics	41
5.2	Derivatives	42
6	Matrix computations	42
6.1	Defining a matrix: matrix()	42
6.1.1	Matrix functions.....	43
	setmatrix(matrix,value [,diagonal->]).....	43
	t(a).....	44
	inverse(a)	44
	dotproduct(a,b[,limit1][,limit2])	44
	elementsum(a[,limit1][,limit2][,row->][,column->]).....	44
	submatrix(a[,row->][,column->])	45
	nrows(a)	45
	ncols(a)	45
	len(a[,any->]).....	45
	index(val, a[,any->])	45
	sort(a,key->(key1[,key2])).....	46
7	Transformation control structures	46
7.1	If	47
	if().....	47
	if()then	47
	elseif()then	47
	else	47
	endif.....	47
7.2	Loops	47
	do(i,start,end[,step]).....	47

	enddo	47
	cycle.....	47
	exitdo	48
7.3	Return from a transformation set.....	48
	return.....	48
	errexit(arg1,...,argn).....	48
7.4	Using addresses in transformation sets.....	48
7.4.1	Address in transformation set	48
	goto('address')	48
	jump('address').....	48
	back.....	49
8	IO-functions	49
	print(arg1,...,argn[,maxlines->][[data->]][row->])	49
	read(file,format,var1,...,varn).....	50
	write(file,format,val1,...,valn).....	50
	write(file,'t',t1,val1,t2,val2,...,tn,valn) ! case[2/5]	51
	write(file,'w',w1,val1,w2,val2,...,wn,valn) ! case[3/5].....	51
	write(file,text_object) ! case[4/5].....	51
	Writing into \$Buffer ! case[5/5]	51
	close(file).....	52
	ask(var1,...,varn[,default->][,q->][exit->]).....	52
	askc(chvar1,...,chvarn[,default->][,q->][exit->])	53
9	Data sets.....	53
9.1	Creating a data object: data().....	53
9.2	Modifying an existing data set: editdata()	55
9.3	Linking hierarchical data: linkdata()	56
9.4	Combining two observations in same class: crossed()	56

9.5	Utility functions for data sets	57
9.5.1	Extracting values of class variables: values()	57
9.5.2	Number of observations: nobs()	57
9.5.3	Getting an observation from a data set: getobs()	58
9.6	Data set object	58
10	Statistical functions	59
10.1	Basic statistics:)	59
10.2	Classifying data: classify()	60
10.3	Linear regression: regr()	62
10.3.1	Computing the regression function: regr()	62
10.3.2	Using the regression object: value(),coef(),se(),mse(),rmse(),r2(), nobs(), len()	62
10.4	Smoothing spline: smooth()	63
10.4.1	Smoothing spline directly from data	63
10.4.2	Smoothing spline from classified data	64
11	Linear programming (JLP functions)	65
11.1	Problem definition: problem()	65
11.2	JLP problem definition object	66
11.3	Solving a problem: jlp()	67
11.4	Inquiry functions for the JLP solution	70
	weights()	70
	unit(i)	70
	schedcum(i)	70
	schedw(i)	70
	weight(i)	70
	partweights()	70
	partunit(i)	71
	partschedcum(i)	71
	partschedw(i)	71
	partweight(i)	71
	price%unit(iunit)	71

	weight%schedcum(sched)	71
	price%schedcum(sched)	72
	price%schedw(iunit,sched).....	72
	weight%schedw(iunit,sched[,integer->]).....	72
	integerschedw(iunit)	72
	integerschedcum(iunit)	73
12	Simulator.....	73
	12.1 Defining a simulator.....	73
	12.1.1 Simulator definition: simulator()	73
	12.1.2 Special functions used in a simulator.....	75
	next(node1,...,nodem)	75
	branch(node1,...,nodem)	76
	cut().....	76
	loadtrees()	76
	12.2 Using a simulator: simulate ()	77
13	Plotting figures.....	79
	Scatterplot: plotyx()	80
	Drawing a function: draw().....	80
	Drawing line through points: drawline().....	82
	Drawing class information: drawclass()	83
14	Stem curves	84
	14.1 Stem splines.....	84
	=stemspline(h1,...,hn,d1,...,dn[,sort->] [print->]).....	84
	stempolar(stemspline,angle[,origo->][,err->])	84
14	Utility functions	85
	14.1 Working directory	85
	showdir().....	85
	setdir(charval).....	85

14.2	Timing functions	85
	secsnds()	85
	cpu()	85
14.3	List functions	85
	=list(obj1,...,objn[,mask->])	85
	=merge(obj1,...,objn)	86
	=difference(list1,list2)	86
	index(object,list[,any->])	86
	len(list[,any->])	87
14.4	Getting value from an object: value(object,xvalue)	87
	14.4.1 Interpolating a regular matrix: value(matrix,x)	88
	14.4.2 Interpolating a classify-matrix: value(cl_matrix,xvalue)	88
	14.4.3 Using a spline: value(spline,xvalue)	89
	14.4.4 Getting values from a transformation set: value(tr_set,xvalue)	89
	14.4.5 Getting value of a list variable	90
14.5	Inverse function: valuex(object,yvalue)	90
	14.5.1 Height of diameter using stemspline: valuex(stemline,diameter)	90
	valuex(stemline,diameter)	90
14.6	Interpolating points: interpolate()	91
	interpolate(x0,x1[,x2],y0,y1[,y2],x)	91
14.7	Integrating a function	91
	14.7.1 Integrating stem curve to get stem volumes	91
	integrate(stem_spline,h1,h2)	91
14.8	Bit functions	91
	setbits(ind, bit1,...,bitn)	91
	clearbits(ind,bit1,...,bitn)	92
	getbit(ind,bit)	92
	=getbitch(ind[,from][,to])	92
	=bitmatrix (nrows[,colmax][,in->] [colmin->][,func->])	93
	value(bitmatrixobj,row[,col][,->any])	94

	setvalue (bitmatrixobj,row[,col],value).....	94
	nrows(bitmatrixobj)	94
	ncols(bitmatrixobj)	94
	=closures(bitmatrixobj).....	94
14.9	Defining crossed variables: properties().....	95
14.10	Storing values of variables	96
	=store(var1,...,varn)	96
	load(storage)	96
14.11	Saving object into files	96
	save(filename,obj1,...,objn)	96
	unsave(filename).....	96
15	Error debugging and handling.....	96
15.1	Errors detected by J	97
15.2	Tracing variables	97
	;trace(obj1,...,objn [,min->],[,max->][out->][,level->] [errexit->])	97
	;traceoff(obj1,...,objn).....	98
	trace(obj1,...,objn [,min->],[,max->][,level->].....	99
	[errexit->])	99
	tracetest(traceset)	99
	An example of tracing functions.....	99
15.3	J does not work correctly.....	101
	debug().....	101
16	Acknowledgements.....	101
17	References.....	101
18	Index	102

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 will supersede previous linear programming software JLP (Lappi 1992) and the unpublished Jakta software.

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 **Hardware requirements**

J is allocating dynamically memory for data structures. Stack size defined at link time determines both the memory need and size of data structures which can be handled during a **J** session. See *Set up of J* for details.

1.2 **Software requirements**

The current version of **J** is developed under Windows XP. It is running also at least under Windows 2000 and Windows NT 4.0.

1.3 **Set up of J**

J is written in Compaq Visual Fortran 6.6 using Fortran 90. **J** is allocating dynamically memory for all objects. It is using stack to store objects. The stack size is set up in the linking time. If there is not enough stack size available, an error occurs during the execution. It is possible change the size of the stack with EDITBIN program. See Visual Fortran help for 'stack size'. Later possibly also versions with different stack sizes will be available

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
```

or

```
!2000
```

where the number gives the maximum number of named objects. The default is 2000. If the first character is '*' then the io-window will remain open if the program is terminated with end command or by a Fortran error. This is the default if there is no `j.par` file. If the first character is '!' then the program terminates completely at end command or at a Fortran error.

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

Program `j.exe` is an ordinary console application. See the following section how to modify the I/O window on the first run.

1.4 *During the first use*

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

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

```
sit>
```

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

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

```
sit>a=7.7  
sit>print(a)  
The result should look like  
a= 8.800000  
sit>
```

Edit then the first command by using the arrow keys into `a=8.8` and submit the command, as well as the print command. Study also the copy and paste possibilities under the Dos icon.

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.

1.5 *Manual conventions*

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

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

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

There are several limitations in the first versions of the **J** software. These limitations are indicated by '***', and they will be hopefully removed from later versions.

2 **J** objects

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

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

2.1 *Object names*

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

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

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

```
print(Names)
```

2.2 **Copying object: a=b**

A copy of object can be made by the assignment statement

```
a=b
```

#Note several objects cannot be copied in one statement as in arithmetic assignment.

2.3 **Deleting objects: delete()**

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

```
delete(obj1,...,objn)
```

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

Note 1: Objects can equivalently (since **J** version 0.7.7) be deleted by giving command

```
obj1,...,objn=0
```

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

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

2.4 **Object types**

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

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

Real variables and constants

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

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

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

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

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

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

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

Character constants and variables

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

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

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

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

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

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

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

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

Text object

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

Logical values

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

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

Object lists

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

Matrices and vectors

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

See *Matrix computations*.

Transformation set

A transformation set groups several operation commands together so that they can be used for different purposes by **J** functions and **J** objects. A transformation set contains the interpreted transformations and possibly the lists of input and output variables. A transformation set also has an associated argument result variables. The result variable is used in result function, and bot argument and result variables are used in value function. For more details see *J function for defining transformation sets: trans()*.

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

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

Simulator

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

Data set

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

Problem definition object

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

Figure object

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

Function objects

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

Storage for variables

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

Bitmatrix

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

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

Trace set

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

*****Coming object types**

J will contain at least following object types currently used in Jakta but not yet implemented in **J**:

-covariance matrix

-interpolating spline function

-stem curve parameters of the nonparametric stem curve model of J. Lappi

2.5 **Objects created automatically and default names**

The following objects are created automatically at start-up:

Names	text object containing the names of all objects
Pi	real variable having value 3.14... .
Result	real variable used to store the result if output variable is not given
Obs	the default real variable used to indicate the number of observation within a data set
\$Cursor\$	transformation object used to run the command level
\$Cursor2\$	another implicit transformation set
\$Val\$	transformation object used to extract values of mathematical statements, used, e.g., in input programming
\$Data\$	default data set name for a new data set created by data-function
Data	a list object used to indicate current data sets
LastData	a list 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 commands .
Duplicate	a special variable used in data function when duplicating observations
\$Buffer	a special character object used by the write function

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

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

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

Object	created by	explanation
Tracevars	;trace	cumulative list of all objects used in all ;trace functions
Tracestatus	;tracef	row vector corresponding to Tracevars list indicating if tracing code is generated
Tracelevel	;trace	vector indicating the tracing level for variables having the tracing code.
Tracecount	;trace	counts of changes

Traceminstatus	;trace	idicates if minimum checking is effective
Tracemin	;trace	minimum values
Tracemaxstatus	;trace	idicates if maximum checking is effective
Tracemax	;trace	maximum values

3 Command input and output

3.1 *Input line and input paragraph*

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

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

** I may drop continuation with '>' in future (it is inherited from JLP)

When entering input lines from the keyboard, previous lines given from the keyboard can be accessed and edited using the arrow keys. Input lines can be also pasted from the clipboard using the right button of the mouse. Using Enter key of the keyboard one can copy text from the **J** window into the clipboard. Console applications of Compaq Fortran do not provide copy and paste using <cntrl>c and <cntrl>v.

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

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

Examples of input paragraphs:

```

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

```

3.2 ***Input programming***

The purpose of the input programming is to read or generate **J** commands or input lines needed by **J** functions. The names of input programming commands start with semicolon ';'. There can be no space between ';' and the following input programming function. The syntax of input programming commands is the same as in **J** functions, but the input programming functions cannot have an output. There are also controls structures in the input programming. An input paragraph can contain input programming structures.

3.2.1 **Addresses in input programming**

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

```

;ad1:

see: ;incl ;goto

```

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

3.2.2 **Changing "..."** sequences

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

```

directory='D:\j\'
name='areal'
extension='svs'

```

then

```

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

```

is equivalent to

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

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

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

are translated into

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

With `"..."` substitution one can define general macros which will get specific interpretation by giving values for character and numeric parameters, and numeric parameters can be utilized in variable names or other character strings. In transformation sets one can shorten computation time by calculating values of expressions in the interpretation time instead of doing computations repeatedly. E.g. if there is in a data set transformation

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

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

3.2.3 Input programming commands and control structures

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

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

Includes lines from a file or from a text object.

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

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

Examples:

```
;
```

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

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

Note2.. See Chapter 'Input from a text object' how to include commands from a text object.

;return

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

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

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

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

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

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

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

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

;enddo

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

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

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

Examples (in a include file):

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

```
call(g)
```

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

```
;if(...)
```

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

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

Examples:

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

```
;if(value);then
```

```
...
```

```
;elseif(value);then
```

```
...
```

```
;endif
```

Picks several lines into input based on some condition.

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

```
;goto('adr')
```

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

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

Example, in an include file:

```
;goto('ask')
...
;ask:
*what to do next: task1, task2 or end
```

```
askc(ad)
;goto(ad)
;task1:
...
;return
;task2:
...
;return
;end:
;return
```

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

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

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

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

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

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

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

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

which is equivalent to:

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

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

Note: Lists can be also used to define pointers to single variables. E.g. a general method defined in an include file (a macro) can refer to a variable using e.g. `@arg`.

Then we can give specific interpretation to the variable giving at the command level `arg=list(temperature)`, and if we then include the macro from an include file then all reference using [@arg](#) refer to the variable `temperature`. Of course we can make the `arg` list to point to several variables by defining it to be a list of several objects..

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

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

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

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

are equivalent to

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

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

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

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

```
print(;%x1)
```

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

3.2.6 Key shortcuts

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

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

give the same result as

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

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

```
"I"
```

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

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

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

```
=text( )
```

```
...  
//
```

output: a text object

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

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

Examples:

3.3 *Immediate operations starting with ';'*

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

3.4 *Controlling output*

It is quite difficult to design the amount of printing logically in environment like **J**. There should be enough output to see that **J** is doing what it should do. But using input programming one can generate efficiently huge amount of commands which could easily cause very much printing. I have now made a preliminary plan how to control the amount of printing, but I have not yet implemented this plan consistently.

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: 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.

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

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

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

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

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

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

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

4.2 **Common options**

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

in->

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

data->

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

If the user has defined a 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 `editdta` function). In all function which are using data sets, `trans->` option defines a transformation set which is used in this function. An example:

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

err->

The `err->` option indicates an 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 `stempoilar` function, but it will be included in many other functions.

4.3 J function for defining transformation sets: trans()

Transformation sets are created with the `trans` function.

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

...

/

options:

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

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

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

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

`source` If value zero is given for the option, then a text object containing the source code is not generated.

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

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

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

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

call(tr)

argument: a transformation set

`call` function can be used at the command level or within transformation set.

Defining transformation sets which are called within other transformation sets one can use some transformation as subroutines. But the transformation sets do not yet have any system for argument passing, thus all objects within transformations are global objects. Using input programming one can define transformations which get specific

interpretation after giving values to character variables and object lists. But these transformations must be interpreted first with `trans` function before they can be used.

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

Example:

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

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

```
call(tr)
```

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

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

result(tr, [,result->])

argument: a transformation set

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

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

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

** Later I'll add some kind of argument passing system.

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->])

argument:

tr a transformation set

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

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

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

See value function for more details

5 Arithmetic computations

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

output_variables=arithmetic expressions

matrix element= arithmetic expression

arithmetic expression

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

Result= arithmetic expression.

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

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

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

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

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

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

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

5.1.1 Standard numeric expressions

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

unary minus

*** integer power

** real power

* multiplication

/ division

+ addition

- subtraction

The reason for having a different integer power is that it is faster to compute.

5.1.2 Logical and relational expressions

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

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

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

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

5.1.3 Arithmetic functions

The arithmetic functions return single real value.

sqrt, exp, log, log10, abs

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

Real to integer conversion

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

min, max

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

sign

`sign(val)` returns 1 if $val \geq 0$ otherwise returns -1.

`dot(c1,...,cn,x1,...,xn)`

dot product, $c1*x1+\dots+cn*xn$, see also matrix function `dotproduct`

`which(cond1,value1,...,condn,valuen[,valuedef])`

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

Trigonometric functions, argument in radians

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

Trigonometric functions, arguments in degrees

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

Inverse trigonometric functions, result in radians

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

Inverse trigonometric functions, result in degrees

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

hyperbolic functions

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

5.1.4 Probability distributions

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

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

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

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

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

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

5.1.5 Random numbers

```
ran(seed)
```

Returns a uniform random number between 0 and 1. Seed determines the exact sequence. If there are no arguments, then the seed will be system generated. If `ran` is called first with `seed` and then without `,` then previous `seed` applies.

rann(seed)

Returns normally distributed random number with mean zero and variance 1

***currently a quick and dirty generator

5.1.6 Special numeric functions

npv(interest,income1,...,incomen,time1,...,timen)

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

5.1.7 List arithmetics

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

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

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

```
alist=list(a1,a2,a3)
blist=list(b1,b2,b3)
clist=list(c1,c2,c3)
@alist=1,2,3
@blist=4,5,6
clist=alist+blist !list=list+list
print(@clist)
c1= 5.000000
c2= 7.000000
c3= 9.000000
clist=alist+5 !list plus real_value
write($,$,@clist)
 6.000000      7.000000      8.000000
clist=-alist !negative
write($,$,@clist)
-1.000000     -2.000000     -3.000000
clist=blist-alist !subtract
write($,$,@clist)
 3.000000      3.000000      3.000000
clist=2*alist !list =real_value * list
write($,$,@clist)
 2.000000      4.000000      6.000000
clist=alist*blist ! element by element multiplication
write($,$,@clist)
 4.000000     10.000000     18.000000
cval=alist*blist ! if output is real variable then dot product is
computed
print(cval)
cval= 32.000000
```

******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.

******Previous functions `multcl`, `multcll`, `multll`, `multlll` are removed because they are now unnecessary.

5.2 *Derivatives*

d1,...,dn=der(x1,...,xn)

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

Example

```
da,db=der(a,b)
```

```
f=a*exp(-b*x)
```

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

6 Matrix computations

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

The following matrix functions are currently available:

6.1 *Defining a matrix: matrix()*

```
matrix(nrows,[ncols][,in->][diagonal->] [,form->]  
[,values->])
```

```
(values if in-> option is present)
```

```
/
```

Generates a matrix.

arguments:

`nrows` number of rows

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

options:

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

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

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

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

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

Examples:

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

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

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

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

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

6.1.1 Matrix functions

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

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

##Note, if m is matrix then substitution

M=1

makes M into a real variable.

t(a)

Computes the transpose of a matrix

argument: a matrix object

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

$h=b*t(a)$

inverse(a)

Compute the inverse of a matrix

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

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

Computes the dot product of two vectors

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

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

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

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

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

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

Computes the sum of elements of vector

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

`elementsum(a,n)` computes the sum using n first element.

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

options:

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

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

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

Takes a submatrix from a matrix

argument: a matrix

options:

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

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

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

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

nrows(a)

Returns the number of rows in a matrix

argument: a matrix object

Note: nrows works also for text objects,

ncols(a)

Returns the number of columns in the matrix

argument: a matrix object

len(a[,any->])

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

argument: a matrix object

option

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

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

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

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

arguments.

val a real value
a matrix object

option

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

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

Note2. when the first argument is list, then index function returns the position of an object in an object list, see list functions

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

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

argument: a matrix object

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

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

Note2: If there is no output, then the argument matrix is sorted in place.

Note3: The argument can be the data matrix of a data object. The data object will remain a valid data object.

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

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

7 Transformation control structures

Within **J** transformations, there can be similar controls structures as in the input programming. The difference is that these will remain as part of the transformation set. Only the '`if()output=...`' structure is allowed at the command level, other are possible only within a transformations set.

7.1 **If**

if()

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

```
if( )output=...
```

or

```
if( )func( )
```

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

Groups of transformations can depend on conditions using structure:

```
if( )then
```

```
...
```

```
elseif( )then
```

```
...
```

```
else
```

```
...
```

```
endif
```

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

7.2 **Loops**

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

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

```
enddo
```

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

```
cycle
```

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

exitdo

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

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

7.3 *Return from a transformation set*

return

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

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

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

errexit (arg1 , ... , argn)

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

Example:

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

7.4 *Using addresses in transformation sets*

7.4.1 Address in transformation set

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

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

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

goto ('address')

purpose: to continue execution of transformations from the given address

jump ('address')

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

back

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

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

Example:

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

8 IO-functions

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

Print values of variables or information about objects.

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

options:

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

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

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

For simple objects, all the object content is printed, for complicated 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, var1, ..., varn)

Reads real variables from a file

arguments:

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

format:

'b' binary format

'(...)' a Fortran format

\$ or '*' the * format of Fortran

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

write(file, format, val1, ..., valn) ! case[1/5]

Writes real values to a file or to the console

arguments:

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

format:

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

A character expression, with the following possibilities:

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

A Fortran format statement, e.g. (`~the values were ~,4f6.0`)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Examples:

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

close(file)

Closes an open file

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

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

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

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

options:

`default` , default values for the asked variables

`q` text used in asking

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

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

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

Examples: (two first are equivalent):

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

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

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

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

options:

default default character strings

q text used in asking

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

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

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

9 Data sets

9.1 *Creating a data object: data()*

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

A data set is created by a `data` function (all options are optional)

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

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

Options:

`read` variables read from the input file

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

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

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

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

`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.

`subdata` the name of the lower level data set to be created.

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

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

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

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

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

`duplicate->(duplicates-transformations,duplicate-transformations)` The two transformation set arguments describe how observations in the `subdata` will be duplicated. The first transformation set should have `Duplicates` as an output variable so that the value of `Duplicates` tells how many duplicates are made (0= no

duplication). The second transformation set defines how the values of subdata variables are determined for each duplicate. The number of duplicate is transmitted to the variable `Duplicate`. These transformations are called also when `Duplicate=0`. This means that when there is the `duplicate` option, then all transformations for the subdata can be defined in the duplicate transformation set, and `submaketrans` is not necessary.

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

`oldobs`: This works similarly with respect to the initial `obs` variable as `oldsubobs` works for initial `obs` variable.

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

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

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

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

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

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

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

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

`editdata(data_set,trans->)`

argument: `data_set` , a data set object

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

Changes the transformation set associated with the data set

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

9.3 **Linking hierarchical data: linkdata()**

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

Links hierarchical data sets.

options:

`data` the name of the upper level data set

`subdata` the name of the lower data set

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

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

`obs` 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%obs_variable_of_the_upper_data`

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

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

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

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

output: an data set

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

do `c=1`, number of classes

do `i=first observation in class`, last observation in class

`$Stage=1`

call transformation set `tr`

```

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

```

9.5 *Utility functions for data sets*

9.5.1 Extracting values of class variables: **values()**

=values(variable[,data->])

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

output: A columns vector getting different values

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

option: `data` gives the data sets searched

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

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

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

9.5.2 Number of observations: **nobs()**

nobs(dataset)

Gets the number of observations in a data set.

argument: a data set

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

An example of `nobs` and `index`:

Fast:

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

Fast:

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

Slow:

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

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

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

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

9.6 *Data set object*

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

parts:

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

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

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

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

`trans (O)` link to transformations done for each observation

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

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

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

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

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

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

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

10 Statistical functions

10.1 *Basic statistics:*)

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

Computes and prints basic statistics from data sets.

arguments: variables for which the statistics are computed.

options:

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

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

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

If the values of the `min` option are variables, then the minima are stored into these variables.

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

`mean` means are stored

`var` variances are stored

`sd` standard deviations are stored

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

`nobs` gives variable which will get the number of accepted observations

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

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

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

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

`# stat` prints min, max, means, sd and sd of the mean computed as $sd/\sqrt{\text{number of observations}}$

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 **Classifying data: classify()**

```
classify([var1,...,varn][,data->],x->[,xrange->][,dx->]
```

```
[,classes][,z->][,zrange->][,dz->][,mean->]
```

```
[trans->][,filter->][reject->][transafter->])
```

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

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

arguments: variables for which class means are computed.

options:

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

`x` the first variable defining classes

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

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

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

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

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

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

`mean` if z variable is given, class means are stored in a matrix given in the `mean` option. (** details fixed later)

`trans`

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

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

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

#Operation:

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

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

10.3 *Linear regression: regr()*

10.3.1 Computing the regression function: **regr()**

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

Computes a linear regression function.

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

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

options:

`data` data sets used

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

`trans`

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

`reject`

`transafter`

10.3.2 Using the regression object:

value(),coef(),se(),mse(),rmse(),r2(), nobs(), len()

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

```
value(regr_object[,x1,...,xn])
```

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

```
coef(regr_object,xvar)
```

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

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

```
se(regr_object,xvar)
```

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

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

`mse(regr_object)`

Returns the MSE of the regression.

`rmse(regr_object)`

Returns the RMSE of the regression

`r2(regr_object)`

Returns the R^2 of the regression

`nobs(regr_object)`

return number of observations used to compute the regression

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

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

option

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

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

***Jakta contains also nonlinear regression. This will be also available in **J**.

10.4 ***Smoothing spline: smooth()***

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

10.4.1 **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: dependent variable, independent variable

options:

`data` data sets used

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

`modeldf` effective degrees of freedom used for model parameters, if not given then the generalized cross validation value is minimized, and the 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. Deafuld is `degree->3`.

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

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

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

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

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

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

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

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

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

10.4.2 Smoothing spline from classified data

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

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

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

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

options:

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

`modeldf, degree:` see above

`wish->(x1,y1,w1,...,xn,yn,wn)`

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

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

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

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

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

*****Jakta contains also regression splines. Perhaps these are not needed as **J** has smoothing splines.**

11 Linear programming (JLP functions)

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

11.1 *Problem definition: problem()*

`=problem([repeatdomains->)`

...

/

Define a lp problem for `jlp` function.

output: a problem definition object

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

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

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

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

Note 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. In the optimization algorithm, the objective row is taken from its arbitrary position in the problem definition to become row 0.

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:

11.2 ***JLP problem definition object***

generated with: problem

used in: jlp

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

parts:

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

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

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

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

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

All coefficients of the constraints in a packed format.

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

11.3 **Solving a problem: `jlp()`**

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

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

options:

`problem` problem definition generated by `problem` function

`data:` data sets describing the stand (management unit) data or the schedules data. The data sets must be linked to schedule data either using sub-options in the `data` function or using `linkdata` function. 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 is 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.

***Currently only one data set allowed

`sparse` sparse matrix routines are used, should be used in large problems with many domains

`tole` the default tolerances are multiplied with the value of the `tole` option (default is thus one). Smaller tolerances means 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` or `tole->0.01`.

`trans` defines a transformation set which is executed for each `cdata` record after doing transformations permanently associated with the `cdata`.

`subtrans` defines a transformation set which is executed for each `xdata` record after doing transformations permanently associated with the `xdata`.

`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.

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

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

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

`report` the standard written output is written into the file given in `report` option

(.e.g. report->'result.txt'. The file remain open and can be written by several jlp-functions or by additional write functions. Use close function to close it explicitly if you want to open it with other program.

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

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

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

jlp is generating output (amount is dependent on the `print` option) plus a JLP-solution stored in special data structures which can be accessed with special **J** functions described below. In addition `jlp` creates a list `zvars%problem` which stores the variables among `vars%problem` which are interpreted to be z-variables. Note that the z-variables get directly the optimal values, and they can be accessed directly without any access functions. 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 condition 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

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

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 come from the x-data. It does not make any sense to have on a problem row only c-variables, but they can be constraints like

$\text{vol\#1} - \text{vol\#0} > 0$

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

11.4 *Inquiry functions for the JLP solution*

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

weights()

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

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

unit(i)

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

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

schedcum(i)

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

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

schedw(i)

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

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

weight(i)

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

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

partweights()

Returns the number of schedules which have nonzero weight in the solution but so that the whole unit is not assigned to the schedule. In a linear programming problem there is usually only one schedule in each unit in the solution i.e. with a nonzero

weight. Binding constraints bring in the solution schedules with weight between zero and one. The schedules can be access with `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 I'm not sure if I'll keep this

partunit(i)

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

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

partschedcum(i)

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

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

partschedw(i)

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

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

partweight(i)

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

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

price%unit(iunit)

Returns the shadow price of the unit `iunit`.

argument: unit (stand) number

Note: if active rhs's are nonzero, then the prices of all units do not sum up to the value of the objective function). see Lappi (1992).

weight%schedcum(sched)

Returns the weight of a schedule.

argument: cumulative schedule number

option:

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

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

price%schedcum(sched)

Returns the shadow price of a schedule .

argument: cumulative schedule number

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

price%schedw(iunit,sched)

Returns the shadow price of a schedule within an unit

arguments:

`iunit` the number of the unit

`sched` the schedule number within the unit

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

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

arguments:

`iunit` the number of the unit

`sched` the schedule number within the unit

option:

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

integerschedw(iunit)

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

arguments:

`iunit` the number of the unit

integerschedcum(iunit)

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

arguments:

`iunit` the number of the unit

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-> ] [ ,keeperperiod-> ]
[ ,treevars-> ]
( simulator definition )
/
```

output: a simulator

options:

`periods` number of simulation periods

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

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

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

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

`nper=5`

```

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

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

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

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

```

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

The `next` function tells that the argument nodes will be entered for the next period. The next function can have several arguments and it can be in any place in the node section. The whole node section is always computed before going to next period. The `next` and `branch` function accumulate the branching nodes during the execution time.

Note: 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 computation 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 `keeperperiod` level can be reached. If `next` functions are dependent on logical conditions it may happen that during the simulation the `keeperperiod` level is not reached. An error results in this case only if there is no branches reaching the `keeperperiod` 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 ,... , node_m)

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`)

***The logic of J simulator is different than in the Jakta simulator. In Jakta there is for each node a condition statement describing when the node is entered. In the J simulator there is within the current period description what nodes are entered next or what sister nodes to make. J simulators are more efficient than Jakta simulators, and it is also easier to describe simulators which just contain some fixed predefined schedules (e.g. thin at age 70 and age 80 clear cut at age 90).

***Jakta simulator contains the possibility to simulate growth at plot and tree level. Plot level will be later available also in J.

loadtrees()

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

Example:

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

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

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

12.2 *Using a simulator: simulate ()*

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

Simulate schedules

output: data set storing simulated schedules

arguments: one or more simulator objects

options:

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

selector If there are several simulator argument, there must be selector option which determines which simulator is selected for the current stand. The selector option has one or two arguments. The first argument gives the transformation set which determines which simulator is selected. The second argument, if present, gives the name of list object which tells which simulator is selected. The default is Selected. See an example below.

keep The variables stored in the schedules data (output). If keep-variables are not given, then the output variables of the simulator are stored. **Note**: variables with names starting with '\$' are not counted as output variables. **Note 2**. If there is no output, option is ignored.

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

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

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

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

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

`nobs` 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 `treevars` option in the simulator definition.

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

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

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

```
age#0=40
vol#0=50
simulate(js)
kukuu1
grow/period      1  age=      50  vol=      75
kukuu2
grow/period      2  age=      60  vol=     100
kukuu3
grow/period      3  age=      70  vol=     125
kukuu4
grow/period      4  age=      80  vol=     150
kukuu5
grow/period      5  age=      90  vol=     175
thin/period      5  age=      90  vol=      90
clear/period     5  age=       0  vol=       0
thin/period      4  age=      80  vol=      75
kukuu5
grow/period      5  age=      90  vol=     100
clear/period     4  age=       0  vol=       0
kukuu5
grow/period      5  age=      10  vol=      25
thin/period      3  age=      70  vol=      60
kukuu4
grow/period      4  age=      80  vol=      85
kukuu5
grow/period      5  age=      90  vol=     110
thin/period      5  age=      90  vol=      51
clear/period     5  age=       0  vol=       0
thin/period      2  age=      60  vol=      45
kukuu3
grow/period      3  age=      70  vol=      70
kukuu4
```

```

grow/period      4  age=      80  vol=      95
kukuu5
grow/period      5  age=      90  vol=     120
thin/period      5  age=      90  vol=      57
clear/period     5  age=       0  vol=       0
thin/period      4  age=      80  vol=      42
kukuu5
grow/period      5  age=      90  vol=      67
clear/period     4  age=       0  vol=       0
kukuu5
grow/period      5  age=      10  vol=      25

```

Example of using the simulator with data:

```

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

```

```

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

```

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

```

select=trans()
if(standtype.eq.1)then
Selected=list(js)
else
Selected=list(js2)
endif
/
simsdata=simulate(js,js2,data->data,selector->select,unitdata->cdat)

```

13 Plotting figures

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

then **J** is waiting the disappearance of `jfig.jfig` (i.e. clicking on the figure window) before it writes a new `jfig`-file.

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

The first graphics **J** functions are:

Scatterplot: `plotyx()`

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

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

output: a figure object, default `Figure`

arguments: `y`-variable and `x`-variable

options:

`data` data sets

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

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

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

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

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

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

Drawing a function: `draw()`

```
draw(func->[,x->][, xrange->][,dx->]
```

```
[,yrange->][,dy->][,y->]
```

```
[points->] [,append->][,style->][,width->]
```

```
[ ,color->][ ,mark->][ ,show->][ ,r->]
```

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

output: a figure object, default: `Figure`

options:

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

If `y` is functions of `x`:

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

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

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

If `x` is function of `y` :

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

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

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

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

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

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

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

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

`mark->` mark put to some points on the line

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

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

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

Drawing line through points: drawline()

drawline(x1[,...xn][,y1][,...yn] [,append->][,style->]

[,width->][,color->][,mark->][position->]

[r->][,show->])

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

output: a figure object, default: Figure

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

options:

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

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

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

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

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

0 (default), text is centered

1, text is below

2 text is left

3 text is up

4 text is right

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

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

Note: 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 start up 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->[,sd->][,se->][,dx][,append->][,style->]
```

```
[,color->][,mark->][r->][,show->])
```

Plots class information produced by classify function (class means standard deviations, standard errors)

```
show(fig[,r->][,xrange->][,yrange->][color->])
```

Shows a previously made figure

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

options:

xrange->(xmin,xmax) defines a new x-range for the figure. If xmin=xmax=0, then the minimum and maximum x-coordinates used in any subfigure are used. The new range will become property of the figure object.

sd draw +- class standard deviation around class mean

se draw +- class standard error (sd/sqrt(n)) around class mean

yrange->(ymin,ymax) defines a new y-range for the figure. If ymin=ymax=0, then the minimum and maximum y-coordinates used in any subfigure are used. The new range will become property of the figure object.

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

color-> gives the color code for the whole figure which will bypass any color codes given in subfigures. If no argument is given, the drawing is done in black. This is useful if we want to see figures in colors then we must turn everything into black and white when producing figures for publications. The color codes used in the subfigures will remain unchanged.

***Currently only one argument allowed, later several figures can be overlaid.

14. Stem curves

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

13.1 *Stem splines*

```
=stemspline(h1,...,hn,d1,...,dn[,sort->] [print->])
```

output: an interpolating cubic spline, designed specially for stem curves by Carl Snellman. The resulting spline is tested to see if it oscillates (which can happen with splines). If the spline predicts that the diameter at the center point of two knot heights give by the arguments is over 0.4 cm larger than either of the bracketing diameters, then a new knot point is added having diameter which is equal to the maximum of the diameters.

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` function or `integrate` function.

```
stempolar(stemspline,angle[,origo->][,err->])
```

Compute the diameter at polar coordinate angle (in degrees) using a stemspline object.

options:

origo_> gives the baseline when computing the angle, default is 0

err-> if there is error in obtaining the polar coordinate diameter, then err option defines transformation set which is called before returning from stempolar function.

14 Utility functions

14.1 *Working directory*

The current working directory can be seen or changed.

showdir ()

Prints the current working directory

setdir(charval)

Sets the current working directory. The argument is a character variable or character constant.

14.2 *Timing functions*

There are two timing functions which can be used to measure the computation time. There are two versions of each, without argument, and with an argument

secnds ()

purpose: first call gives the elapsed time since midnight in seconds, later the elapsed time since last call to secnds-function.

secnds (t)

purpose: gives the elapsed time since midnight -t.

cpu ()

purpose: first call gives the cpu time since starting the program in seconds, later the cpu time since last call to cpu-function.

cpu (t)

purpose: gives the total cpu time -t

14.3 *List functions.*

=list(obj1,...,objn[,mask->])

Defines an object list

output: a list object

arguments: 0-n objects (need not exist before)

options:

`mask` defines which objects are picked from the argument list, value 0 indicates that the object is dropped, positive value indicates how many variables are taken, negative value how many object are dropped (thus 0 is equivalent to -1). `mask` option is useful for creating sublists of long lists.

Note 1. If an argument does not exist beforehand, it is first created as a real variable.

Note 2: The same object may appear several times in the list. (see `merge`)

Note 3. There may be zero arguments, which result in an empty list (see example below)

Examples:

```
all=list() ! empty list
sub=list()
;do(i,1,nper
period#"i"=list(ba#"i",vol#"i",age#"i",harv#"i")
sub#"i"=list(@period#"i",mask->(-2,1,-1))
all=list(@all,@period#"i") !note that all is on both sides
sub=list(@sub,@sub#"i")
;end do
```

=merge(obj1,...,objn)

Defines a list dropping multiple references to the same object

arguments: objects or lists

If an argument is a list, then it is not necessary to expand it using `@`-operator, even if it can be expanded and the result is the same.

Note: arguments of `merge` need to be known beforehand (unlike in `list` function).

=difference(list1,list2)

Defines a list dropping from list `list1` all objects found in list `list2`

arguments: lists

index(object,list[,any->])

Gets the index of a variable in a list, usually in the `keep` list of a data set (the column number in the data matrix)

arguments:

`object` object name

`list` : a list object

options:

`any`: accept also that variable is not in list (output=0) without error condition

Note 1: if the second variable is not a list, error occurs. If the variable is not in the list, index gets value 0. A error condition is obtained if `any` option is not present

Note 2. It is faster to get the value of the index within input programming or outside the transformation set so that it must not be searched repeatedly.

Note 3. See *Inquiry functions for data object* for an example how to utilize `index` function.

len(list[,any->])

Returns the number of elements in the list

argument: a list object

option

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

Note: `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: The value of a specific list element variable can be obtained using `value` function

14.4 **Getting value from an object: value(object,xvalue)**

If a **J** function generates an object containing parameters for a special function then the `value` function can be used to generate values from the object. The general form of the `value` function is

value(object,xvalue [options])

where `xvalue` is the value used as the argument for the object which can be used as a function. For most cases the object can be used also otherwise. The output is a single numeric value.

If the first argument is a list and there is option `index->` then the object is picked from the list.

There are the following special cases.

14.4.1 Interpolating a regular matrix: value(matrix,x)

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

Interpolates linearly rows of matrices.

arguments:

matrix a matrix

xvalue value for which a row must be interpolated

options:

row Gives the y-row for interpolation if there are more than two rows in the matrix and only one row needs to be interpolated.

#The first row of the matrix defines the knot points. It is assumed that knot points are in increasing order. If there are only two rows in the matrix, then the second row defines the values at the knot points. If there are more than two rows then a vector is generated by interpolating each row from 2 to nrows(matrix), unless there is row option

Note: Also extrapolation is allowed, i.e. the argument can be smaller than the first knot point or larger than the last knot point.

Example:

```
sit>a=matrix(3,4,in->)
10, 20, 30, 40
15, 16, 18, 20
20, 40, 60, 80
/
sit>v=value(a,35)
sit>print(v)
v is matrix(          2 ,          1 )
 19.00000
 70.00000
sit>c=value(a,15,row->2)
sit>print(c)
c= 15.50000
```

*** Later quadratic and cubic interpolation, as well of interpolating two dimensional matrices will be available.

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

value(cl_matrix,xvalue)

Interpolating a matrix produced by classify function.

output: a real variable

arguments:

`cl_matrix` a matrix produced by `classify` function

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

14.4.3 Using a spline: `value(spline,xvalue)`

`value(spline,xvalue)`

Gets values from a smoothing spline or a stem curve spline

arguments:

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

`xvalue` argument of the spline.

**later other types of splines will be available

14.4.4 Getting values from a transformation set: `value(tr_set,xvalue)`

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

arguments:

`tr_set` a transformation set generated by `trans` function

`xvalue` argument which is put into the argument variable (default Arg)of the transformation set

options

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

`result` , defines the variable whose value is the result of the function, default is the result variable associated with the transformation set (and default for that variable is Result)

Note The original value of the argument variable is remains unchanged.

Example:

```
s=trans(input->,arg->x,result->h)
```

```
h=sin(x+z+1)
```

```
/
```

Then `y=value(s,3)` is equivalent to

```
xold=x
```

```
x=3
```

```
call(s)
```

```
y=h
```

```
x=xold
```

This form of the value function is useful e.g. in filter, reject or func options or when transformations are needed to get numeric values into options.

Note 2. A transformation set can be used also as a function using result function, if the transformation set does not use a argument whose value need to bypassed simultaneously.

14.4.5 Getting value of a list variable

value(list,index)

arguments:

list a variable list generated by list function

index index of the variable

Example:

```
alist=list(a,b,c)
```

```
b=6.7
```

Then value(alist,2) returns 6.7

*** Later there will be more function objects accessed using value function

14.5 *Inverse function: valuex(object,yvalue)*

An inverse function gives the value of the x-variable for which the function obtains a give value. Currently the only inverse function implemented is:

14.5.1 Height of diameter using stemspline: **valuex(stemspline,diameter)**

valuex(stemspline,diameter)

height (in m) of a given diameter

arguments:

stemspline stemspline object generated with stemspline or stemspinej

diameter diameter (in cm) for which the height is obtinaed

14.6 *Interpolating points: interpolate()*

`interpolate(x0,x1[,x2],y0,y1[,y2],x)`

If argument x2 and y2 are given then computes the value of the quadratic function at value x going through the three points, otherwise computes the value of the linear function at value x going through the two points.

arguments: numeric values

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

14.7 *Integrating a function*

*** Later there will be several for of `integrate` function. Currently the only one is:

14.7.1 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 $\pi*0.25*value(stem_spline,h)^2$ and the result is then divided by 10000 to get the result in dm³ (in stem splines both height and diameter are in cm)

14.8 *Bit functions*

In some applications we may need several indicator variables to indicate if some property is present. In large data sets would be waste of space to store a separate variable for each indicator. J has special bit functions for packing several indicators in the same variable. One variable can store 32 indicators, and it is also possible to store more indicators using variable lists. When bits are stored into variables, then these variables can be included in data sets. There is also a special bit matrix object which is created with `bitmatrix` function. The bit patterns can be read from files, or set by `setvalue` function. Bits in a bit matrix can be obtained with `value` function and the matrix can be printed with `print` function.. There are following bit functions available.

`setbits(ind, bit1,...,bitn)`

Sets one or more bits on.

arguments:

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

Note: argument `ind` is used both as input and output

`bit1,...,bitn` bit positions to be put on

The bit positions of a real variable are numbered 1,...,32, the bit positions of a variable list are numbered from 1 to 32*(number of variables in the list).

clearbits(ind,bit1,...,bitn)

Sets one or more bits off.

arguments:

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

Note: argument `ind` is used both as input and output

`bit1,...,bitn` bit positions to be put off

The bit positions of a real variable are numbered 1,...,32, the bit positions of a variable list are numbered from 1 to 32*(number of variables in the list).

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

getbit(ind,bit)

Gets the value of a bit position.

arguments:

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

`bit` bit position to read

function: If the bit is on, then the value of the `getbit` function is 1 (*True*), otherwise zero (*False*). the bit positions of a real variable are numbered 1,...,32, the bit positions of a variable list are numbered from 1 to 32*(number of variables in the list).

Note: The `getbit` function can be directly used in logical statements, e.g.,
`if(getbit(ind,8))then`

=getbitch(ind[,from][,to])

Gets indicators into a text line, '1' indicating a bit which is on, and '0' a bit which is off.

Output: text object

argument

ind a real variable or variable list

if there are no other arguments, then all 32 bit positions are put into the output, if there is an additional argument, then this indicates the number of bits read, and if there are two additional arguments the first indicates the starting position and the second indicates the last position.

Example:

```

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

```

=bitmatrix (nrows[,colmax][,in->] [colmin->][,func->])

Output: a bit matrix object.

arguments

nrows number of rows in the bit matrix, value -1 indicates that this is indicated by the number of records in the file given in in-> option.

colmax upper limit of column index, default=1, value -1, indicates that this is obtained from the column indices read from the file given in in-> option.

options

in-> indicates that the bit pattern is read from the input paragraph or from the file. If in-> option is not given then all bits are initially zeros until changed with setvalue function.

colmin-> gives the lower limit of the column index

func-> when read the column indices, they can be transformed first using the function give 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 to pack the matrix and acces 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 nonzero values indicate that the bit is set into one.

nrows(bitmatrixobj)

returns the number of rows in a bitmatix

ncols(bitmatrixobj)

returns the number of column in a bitmatix

=closures(bitmatrixobj)

To get neighborhoods indicated by a bitmatrix

output: a bitmatrix

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

#All the neighbors of a given point are not necessarily neighbors if they are located at opposite sides of a point. Closures function will generate all such neighborhoods where all points are neighbors.

Example:

If points are located

1 2

3 4

5 6

Then this can be firstdescribed

```

ne=bitmatrix(6,6,in->)
4,1,2,3,4
4, 2,1,3,4
6 ,3,1,2,4,5,6
6, 4,1,2,3,5,6
4 ,5,3,4,6
4,6,3,4,5
/

```

Note that the 'focus' point is given as first in each line, but the neighbors can be in any order. Then commands

```

ne2=closures(ne)
print(ne2)

```

will produce output

```

ne2 is      2 x (      1 :      6 ) bitmatrix:
111100
001111

```

**The algorithm in closures is not well tested

14.9 ***Defining crossed variables: properties()***

A data object is describing several subjects by defining for each subject a set of variables associated with them. If there are a few named subjects then it may be useful to have separate subject specific variables (constants) which define properties of the subjects. These kinds of variables can be defined with properties function.

```

properties(var1,...,varn[,print->])

```

```

subject1, vall,...,valn

```

```

...
/

```

Defines subject specific constants.

Arguments: generic names of variables

Options: _____ print : are the values printed (to check that they have been read correctly)

Input paragraph following `properties` function has a line for each subject, where first is the name of the subject, and then values for all argument variables. The properties function then defines variables having first the generic variable name, then '%' and then the subject name.

Example:

```

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

```

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

14.10 **Storing values of variables**

=store(var1,...,varn)

Stores the values of variables.

Output: a storage object

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

load(storage)

Loads back the values of variables.

argument: a storage object created by `store`.

*** Now only values of real variables can be stored. If there is need to store general objects, it is quite easy to make `store` capable of handling these.

14.11 **Saving object into files**

save(filename,obj1,...,objn)

arguments:

filename character variable or character constant, the name of the file

obj1,...,objn named objects

Note: There can be several save commands which all save into the same file, the file remains open after each save command.

Note 2. The file can be closed by using `close(filename)`

*** Currently the only compound objects (i.e. objects having links to other objects, e.g. data set or transformation set) which can be save are:

-list of real variables

-regression object

.

unsave(filename)

argument:

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

Note: 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 exist, then it will be replaced.

Note2: If `unsave` is tested using it in the same run as the save functions, the file must be closed first with `close` function.

15 Error debugging and handling

15.1 **Errors detected by J**

If **J** detects an error then the following information is provided:

-**J** prints the current and previous input line as generated by the input programming.
 -**J** will close all open include files and it tells how many lines it has read from these files. Usually, but not always, the last line read has caused the error. E.g. `;do` loops are read first before starting to interpret lines within the loops. So if the error is within the loop, then last line read is later than the error line.

If an error occurs when computing interpreted transformations within a transformation set or a simulator, and saving of source code is not denied by `source->0`, then **J** also prints the source code causing the error. The source code around the line can be seen by using `print` function (possibly with `row` option) or by writing the whole source code into a file using `write` function.

*** In some cases **J** may print the next source line. Please tell about these cases.

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

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

*****J** is in principle protected against trying to put data above the allocated dimension limits, but there will certainly be problems in the first versions.

15.2 **Tracing variables**

It is possible, using special tracing functions, to track changes of variables, define minimum and maximum values for variables, and test if all required variables get values within any phase of a **J** run.

```
;trace(obj1,...,objn [,min->],[,max->][out->][,level->]  
[errexit->])
```

Start generating tracing information for objects.

arguments: **J** objects (usually real variables)

options:

`min->` gives lower bound for a variable. If a variable gets smaller value then the value and the source line are written, and if `errexit` option is present then an error condition occurs.

`max->` gives upper bound for a variable. If a variable gets smaller value then the value and the source line are written, and if `errexit` option is present then an error condition occurs.

out-> gives a name for an trace set object which will be generated and which can be used in tracetest function to test that all objects obj1,...,objn have got values in a section of a J run.

level-> defines the level of tracking, possible values are

0 means that tracing code is generated but it is now deactivated, it can be activated later with trace function.

1 indicates that the changes of objects are counted but not automatically written unless ranges given in min or max option is violated.

2 indicates that all changes are written.

If min, max or out option is present then default is 1 otherwise 2.

errexit-> if value smaller than min or value greater than max occurs, then an error occurs.

Note. When an object is used as an argument of ;trace function, then each time when the object is an output of any function or arithmetic statement, then the transformation interpreter adds a call to tracing function. What exactly happens in this tracing function is dependent on the current values of tracing parameters which are initially set by ;trace function but which can be later modified, also within an transformation set, using trace function.

;trace function creates following objects:

Tracevars = a cumulative object list of all objects used in all ;trace functions. Even if generation of tracing code is stopped for an object, the object remains in the same place in Tracevars list.

Tracestatus = row vector corresponding to Tracevars list indicating if tracing code is generated (value 1) or not (value 0).

Tracelevel = row vector telling the current value of tracing level.

Tracecount = row vector showing counts of changes, used e.g. by tracetest function-

Traceminstatus= row vector indicating if minima are given

Tracemin = contains the given minimum values

Tracemaxtatus= row vector indicating if maxima are given

Tracemax = contains the given maximum values

Note. Current Tracevars list and current values of the trace parameter vectors can be seen by printing. It is also possible to change the parameter values directly, but it is recommended that ;traceoff and trace functions are used to change the values.

;traceoff(obj1,...,objn)

Stop generating tracing code for objects.

Note: The objects remain in the Tracevars list but the values in Tracestatus vector are changed into zero.

```
trace(obj1,...,objn [,min->],[,max->][,level->]
[errexit->])
```

Change the tracing parameters for objects. The meaning of arguments and options is like in ;trace function. The differences are:

1. Arguments of trace function must be previous arguments of ;trace function.
2. Trace set can be defined only in ;trace function (using out-> option)
3. trace function can be an function within a transformation set, but ;trace function is just done directly and it will not regain part of the transformations set. With trace function one can program dynamic tracing and debugging strategies.

```
tracetest (traceset)
```

Test that all objects in a trace set object have been changed since last call of the tracetest function ort from the beginning. If not all objects have been changed, an error occurs.

Argument: An trace set object generated by the out option of the ;trace function.

Note: tracetest function is useful mostly in two different cases:

1. if the values of some variables are determined in complicated control structures which may contain 'holes', i.e. with some combinations of input variable values an intended output variable does not get any value at all.
2. When defining a simulator it can very easily happen that all intended output variables do not get values in all nodes. See simulator function how to utilize tracetest function.

An example of tracing functions

Define a transformation set tr as follows:

```
tr=trans()
a=1
;trace(a,b) !start generating tacing 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 8.000000 in tr at line 2 :
a=7+1
b got value 5.000000 in tr at line 3 :
b=2+3
b got value 3.000000 in tr at line 5 :
b=3
trace(a,level->0) ! stop tracing a even if it
call(tr)
b got value 5.000000 in tr at line 3 :
```

```

b=2+3
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) ! stop tracing a
call(tr)
b got value  5.000000   in tr at line      3 :
b=2+3
b got value  3.000000   in tr at line      5 :
b=3

```

Start checking that b is at least 3.

```

trace(b,min->4,errexit->)
call(tr)
b got value  3.000000   in tr at line      5 :
b=3
*err* transformation set=tr, *source= source%tr
error on source row      5:
b=3

```

An example of using tracetest. Define first trace set outvars and transformation set tr2:

```

;trace(x1,x2,out->outvars) ! define trace set outvars
tr2=trans()
if(a.gt.2)then
x1=5
x2=4
elseif(a.lt.3)
x2=7
endif
/

```

Define a and execute transformations:

```

a=7
call(tr2) ! now both x1 and x2 get new values
tracetest(outvars) ! nothing happens

```

But call then transformations using a=2:

```

a=2
call(tr2) ! now x1 is not updated
tracetest(outvars) ! comment helps to find the place if error occurs
*tracecount for x1 is zero
*err* transformation set=$Cursor$
**input line:tracetest(outvars) ! comment helps to find the place if
error occurs
*closing inc-file 'trace.txt'
  after reading          34 lines from          34

```

See simulator for another example of tracetest.

15.3 ***J does not work correctly***

Several functions have an `debug->` option even if it is not described in the above manual. With this option **J** writes extra information about how it proceeds. User may try this option before consulting J. Lappi.

Error messages starting with `'*j*'` indicate programming errors which should be reported to the author.

If there is in transformations or at the command level function

```
debug( )
```

then special debugging mode is entered. Debug information can be understood only by the author. Debugging can be made conditional by having argument

```
debug(t.gt.0)
```

This is actually equivalent to

```
if(t.gt.0)debug( )
```

Debugging is put off by

```
debug(0)
```

16 Acknowledgements

Function `jlp` is using linear algebra subroutines contained in quadratic programming software `Bqp` made by Prof. R. Fletcher, University of Dundee. Discussions with Reetta Lempinen helped to fix the properties and the syntax of the software. She also introduced me to the programming techniques of Fortran 90. Even Bergseng started to use the linear programming modules on a premature stage and Even has detected a large number of errors. Jarmo Saarikko and Olavi Kurttio made the web distribution of the software. Jarmo Saarikko also cleaned up considerably the first version of this manual.

17 References

Dantzig, G.B and VanSlyke, R.M. 1967. Generalized upper bounding techniques. *Journal of Computer and System Sciences*. 1:213-226.

Lappi, J. 1992. *JLP - A linear programming package for management planning*. Finnish Forest Research Institute Research papers. 414, 134 p.

R Development Core Team, 2004. *R: A language and environment for statistical computing*. ISBN 3-900051-00-3. <http://www.R-project.org>

Fletcher, R. 1996. Dense factors of Sparse matrices. Dundee Numerical Analysis Report NA/170

18 Index

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

;do()	27	dotproduct()	44
;enddo	27	draw().....	80, 83
;goto	28	drawline().....	82
;if().....	28	editdata()	55
;incl()	26	elementsum()	44
;return	27	enddo.....	47
abs().....	39	err->	34
acos()	40	errexit()	48
acosd()	40	exitdo	48
acotan().....	40	exp().....	39
acotand().....	40	Figures	79
Address in a transformation set	48	File	50, 52
Address in input programming	25	format.....	50
asin().....	40	getbit().....	92
asind().....	40	getbitch().....	92
ask().....	52	getobs()	58
askc()	53	goto().....	48
atan().....	40	if()	47
atand().....	40	in->.....	33
back	49	Index in list	86
Basic statistics	59	index().....	86
Binary format	50	Input programming	25
Bit functions	91	int()	39
bitmatrix().....	93	Integer	39
branch()	76	integerschedcum().....	73
Buffer	51	integerschedw().....	72
call()	35	integrate	84
ceiling()	39	inverse function.....	90
Character constant	20	Inverse trigonometric functions	40
Character variable.....	20	inverse()	44
classify()	60	j.par	16, 83
clearbits().....	92	jfig.....	79
close().....	52	jfig.jfig	79
cos().....	39	JLP	65, 70
cosd().....	40	jlp()	67
cosh().....	40	jump()	48
cot()	39	Key shortcuts	30
cotd()	40	len(list).....	87
cpu()	85	len(matrix)	45
cut()	76	Linear programming	65
cycle	47	linkdata().....	56
Data set	53, 58	List arithmetics	41
data()	53	list().....	85, 90
data->.....	34	Lists.....	85
debug().....	101	load().....	96
Degrees.....	40	loadtrees()	76
delete().....	19	log().....	39
difference()	86	log10().....	39
Directory.....	85	Logical expressions.....	38
do()	47	Logical values	20
dot()	39	Matrix	88

Matrix computations.....	42	setvalue().....	94
matrix().....	42	Shortcuts.....	30
max().....	39	showdir().....	85
Merge lists.....	86	simulate().....	77
merge().....	86	Simulator.....	73
min().....	39	simulator().....	73
ncols().....	45	sin().....	39
next().....	75	sin(x).....	81
nint().....	39	sind().....	40
nobs().....	57	sinh().....	40
npv().....	41	smooth.....	89
nrows().....	45	smooth().....	63
Object names.....	18	Smoothing spline.....	63
Option.....	33	sort().....	46
partschedcum().....	71	sqrt().....	39
partschedw().....	71	sqrt2().....	39
partunit().....	71	stat.....	59
partweight().....	71	Statistical functions.....	59
partweights().....	70	stemspline.....	84, 89, 91
plotyx().....	80	store().....	96
price% <i>schedcum</i> ().....	72	submatrix ().....	45
price% <i>schedw</i> ().....	72	t()44	
price% <i>unit</i> ().....	71	Tab.....	50
print().....	49	tan().....	39
problem().....	65	tand().....	40
properties().....	95	tanh().....	40
Radians.....	39	;if().....	28
ran().....	40	Timing.....	85
Random numbers.....	40	trans().....	89
rann().....	41	Transformation set.....	21
read().....	50	Transpose.....	44
regr().....	62	unit().....	70
Regression.....	62	unsave().....	96
Relational expressions.....	38	value.....	84
result().....	36	value().....	36, 88, 94
return.....	48	values().....	57
save().....	96, 97	valuex().....	90
Scatterplot.....	80	weight% <i>schedcum</i> ().....	71
schedcum().....	70	weight% <i>schedw</i> ().....	72
schedw().....	70	weight().....	70
secsnd().....	85	weights().....	70
setbits().....	91	which().....	39
setdir().....	85	write().....	50
setmatrix().....	43		