**Juha Lappi**

# **J3.0 Setup and Development Guide**

**Version 3.0 2021**

**t**

[**J3.0 Setup Guide** 1](#_Toc471214734)

[1. Introduction 2](#_Toc471214735)

[2. Files of the release package 2](#_Toc471214736)

[2.1. Documentation (+j.par) 2](#_Toc471214737)

[2.2 Executable J programs for Windows 3](#_Toc471214738)

[2.3. Jfig: showing figures in Windows 3](#_Toc471214739)

[3. Setting up the J environment 3](#_Toc471214740)

[3.1. Using ready executables in Windows 3](#_Toc471214741)

[4. The source code 4](#_Toc471214742)

[5. The precompiler 5](#_Toc471214743)

[6. Compiling and linking J 6](#_Toc471214744)

[6.1. Using Gfortran 6](#_Toc471214745)

[6.2 Using Intel Fortran 7](#_Toc471214746)

[6.3 Using another compiler 7](#_Toc471214747)

## 1. Introduction

This setup guide contains all essential information to setup J environment.

## 2. Files of the release package

This chapter describes the files distributed with the release package. The release package is available from github.com/###RELEASE###. There are three executable Windows programs included in the package: J, Jfig and J\_precompiler. J program itself is distributed in four different executable Windows versions. The source of all these programs is also distributed. Jfig is showing figures in Windows environment. So the source of it cannot be directly used in other environments. Note that figures can be also produced by R scripts which are made by graphics functions of J. This way J graphics can be utilized in other environments. The files are packed in two zip files, j\_text.zip and j\_exe.zip. J-text.zip contains documentation, source code and example files, i.e. all files which can be distributed with e-mail. j\_exe.zip contains exe- and dll files which cannot be distributed with e-mail.

### 2.1. Documentation (+j.par)

The packed file j\_text.zip contains files

J3.0\_userguide.pdf

J3.0\_setup\_development.pdf (this file)

Lappilempinen.pdf The paper Juha Lappi & Reetta Lempinen (2014) A linear programming algorithm and

software for forest-level planning problems including factories, Scandinavian Journal of Forest

Research, 29:sup1, 178-184, DOI: 10.1080/02827581.2014.886714

jlp92.pdf Old (1992) manual of JLP software, explains the principles of the JLP algorithm used also in J

jhelp.txt help file used by help() –function. The user can edit this and make also alternative help files for different purposes.

j.par file which can be used to define number of objects and initialization commands.

Makefiles:

makefile\_release

makefile\_debug

makefile\_release\_mac

Source code:

j\_modules.f90

own1\_mod.f90

own2\_mod.f90

own3\_mod.f90

j\_ownmod.f90

own1.f90

own2.f90

own3.f90

j\_utilities.f90

fletcherd.for

jlp.f90

matsub.for

jsysdep\_gfortran.f90

j\_sysdep\_intel.f90

j.f90

Word files J3.0\_userguide.docx J3.0\_setup\_development.docx

are available on request.

### 2.2 Executable J programs for Windows

The zip file j\_exe.zip ###Github#YKSIvaiERIKSEEN### contains four different versions of the executable Windows J3.0 software:

J3.0.exe Release version compiled with Gfortran

j3.0\_debug.exe Debug version compiled with Gfortran

J3.0\_intel.exe Release version compiled with Intel Fortran

J3.0\_intel\_debug.exe Debug version compiled with Intel Fortran.

The release versions are much faster than debug versions, so they should be primarily used. Debug versions should be used only when the execution of the release version terminates in a Fortran error without telling what was the source line causing the termination. The differences between Intel and Gfortran versions are not clear. With some Fortran errors, the error messages of Intel Fortran debug version are more specific. It is recommended that the program is run in the command window so that the error messages remain visible and do not just disappear.

If Linux and Mac executables are needed they can be provided.

### 2.3. Jfig: showing figures in Windows

The graphics created by J can be seen in Windows environment using Jfig program. Jfig and J change data with file jfig.jfig and change messages about availability of a new figure with file jflag.jflag . Note that graphics can be created by J and then it can be shown using only R by using r-> option and show->0 options in J graphics functions.

There are two files in the packed file jfig.zip:

jfig.f90 the source file for Intel Fortran

jfig.exe the Windows executable

The source file is not commented, but the use of graphics and window handling subroutines can be easily detected from the code. I hope that someone could make a program for other programming environments.

File . j\_examples.zip contains examples.

## 3. Setting up the J environment

The J working environment can thus be set up e.g. as follows. It is assumed that the user first makes a special directory for J software called J directory.

### 3.1. Using ready executables in Windows

1. Load and unzip the documentation file j\_doc.zip into J directory. If you need to change the number of objects or if you want J to run automatically certain initialization commands put j.par into the active working directory. If you want to use J’s online help, put copy of jhelp.txt to the active directory.

2. Load j\_exe.zip ###Github#YKSIvaiERIKSEEN### from github.com/###RELEASE### and unzip it to J directory. Make shortcuts for those executables you intent to use. Edit the properties of the shortcuts as explained in the user’s guide. Make copies of the shortcuts into the active working directory.

3. If you are making figures and want to see them directly in Windows, load and unzip jfig.zip in J directory. Make a shortcut of Jfig.exe to the active directory, and edit the properties of the shortcut only with respect to ‘start in’ part (jfig is not running in console window, so its properties need not to be edited).

4. Open Command Prompt. Set the directory to the active directory (if you need to change the disk, write e.g. c: , going upwards in the directory write cd .. and then write e.g. cd projectA\subprojectB). Start the proper J executable by writing e.g. J3.0.lnk. Or alternatively you can set the path to the folder containing J executable and run J in your working directory. If you are using graphics start jfig directly by double clicking the shortcut (there is no need to run jfig in the Command Prompt). Start working.

### 4. The source code

The source code of J is available from the GitHub repository github.com/##jREPO###. Repository contains the following files (in the order they are used in makefile for Gfortran)

j\_modules.f90 file for defining function names, option names and object type names for basic J

own1\_mod.f90 file for defining function names, option names and object type names for own1 package

own2\_mod.f90 the same for own2 package

own3\_mod.f90 the same for own3 package

j\_ownmod.f90 modules putting basic J and own package definitions together

own1\_.f90 file containing own1 functions and subroutines they need

own2.f90 the same for own2 package

own3.f90 the same for own3 package

j\_utilities.f90 utility functions, can be used also by own functions, interfaces are in j\_modules.f90

fletcherd.for file containg Fletcher’s subroutines

jlp.f90 subroutines related to jlp functions

matsub.for mathematical subroutines obtained from other open sources and written in Fortran 77

jsysdep\_gfortran.f90 system dependent subroutines for Gfortran

jsysdep\_intel.f90 system dependent subroutines for Intel compiler

jsysdep\_unknown.f90 system dependent subroutines for other compilers which just produce error messages

#Note: only one of these sysdep –files is needed

j\_.f90 file containing the main program and j-functions

## 6. The precompiler

The purpose of the precompiler is to add the needed Fortran ‘use’ -statements when the user’s code contains references to global J variables or utility functions or subroutines defined in the J’s core modules. Specifically, for global J variables the precompiler is producing statements like

use j\_globalsmod, only: j\_v ! real variable associated with each named object

The previous line is produced whenever the user is writing code containing reference to the vector j\_v. If the code contains the reference to e.g. subroutine j\_printtitle() then the precompiler will generate code

use getmod, only: j\_printtitle ! write !!title of object iv to unit nu

Note: the comment above originates from file j\_modules.f90 where !!title is in form %%title. Tag %% is used to mark keywords which can be searched in all source files. Available keywords can be seen at the beginning of file j\_modules.f90. It is reasonable to find only places where a subroutine or variable is defined, not all places where they are used. That is why the precompiler changed %%title into !!title.

Initially the precompiler was used to add j\_ -prefix to global variables and utility subroutines. The user can add her/his own prefix to variables and subroutines stored in her/his own modules. If the prefix should not be changed, then the same prefix should be both for the input and ouput, and the prefix change does not have any effect.

If a module should not be handled with j\_precompiler, then ‘!nopre!’ at the end of the line at the module name line indicates this. By default all use –lines are deleted from the input file unless it has also !nopre! at the end of the line. A reason for having ‘!nopre!’ is that subroutines or variables with the same names can be stored in several modules. This may be reasonable if stand and tree level simulators have the same upper level structure with same subroutine and variable names (e.g. thin, clearcut, basal\_area), they just access different lower level subroutines and data structures.

There are nine files for the precompiler available from the GitHub repository github.com/##jPrecompilerREPO###.:

j\_precompiler.f90 the source of the precompiler

j\_pre.txt the command file for the precompiler when the the compiler is Gfortran

j\_pre\_intel.txt the command file for intel compiler

j\_pre\_unknown.txt the command file for some other compiler

Makefile\_debug the gfortran makefile for debug version

Makefile\_debug\_mac the gfortran makefile for debug version for Mac OS X or Linux

Makefile\_release the gfortran makefile for faster release version

Makefile\_release\_mac the gfortran makefile for faster release version for Mac OS X or Linux

j\_precompiler.exe the Windows executable

The first line of j\_precompiler.f90 shows how it can be compiled and linked with Gfortran.

If doing own packages in Windows environment, the j\_precompiler.exe provided in j\_precompiler.zip can be used directly.

When running the precompiler answer with *<return>* if commands are in j\_pre.txt. If commands are in j\_pre\_intel.txt (or in j\_pre\_unknown), give that name to the question. The only difference between the command files is that j\_pre.txt asks the precompiler to precompile j\_sysdep\_gfortran.f90, j\_pre\_intel.txt asks to precompile j\_sysdep\_intel.f90 and j\_pre\_unknown.txt asks to precompile j\_sysdep\_unknown.f90. The sysdep –files contain currently three subroutines which are system dependent. First subroutine, jsleep() puts the computer into sleep, and its used only if ;incl-function has wait-> option. The showdir() and setdir() subroutines are accessed under j-functions showdir() and setdir().

#Note: after running the precompiler first time and after making the needed sysdep file, command file j\_pre.txt can be used even if it generates an unnecessary sysdep file. Thus after the first use, the answer to the first question can be *<return>*.

#Note: the precompiler command file j\_pre.txt is compatible with Makefile\_release and Makefile\_debug

The program j\_precompiler.exe compiled with Gfortran returns error message :

*Note: The following floating-point exceptions are signalling: IEEE\_DENORMAL*

The reason for this message is unknown but it does not seem to indicate anything serious and the precomiler compiled with Intel Fortran does not produce any warnings.

The precompiler can also be used the make indentations for any f90 source files. By answering the question by ‘+’, the precompiler just makes indentations using input and output files in j\_pre.txt (the prefixes are just ignored, but they must be present). If the command file is different than j\_pre.txt the file name must be given immediately after ‘+’ sign. The indentation run is done separately, and the indentation can be done for any f90 files (and perhaps for f95). Strange ways to put several parts of code into the same line using ‘;’ can confuse the program. Using ‘do ...;…;endo’ and ‘if … then;… endif’ structures in a single line are allowed. In addition to indentation, the precompiler also adds comments showing how each section started. When any section ends so that the indentation should move on step left, the start of the section is included as a comment. This helps to understand e.g. how nested if..then..endif and do..enddo sections are made up. Initially the J source code did not have proper indentations, so I made the precompiler to do the indentations. In future it may be better to keep the indentations clean all the time.

## 6. Compiling and linking J

The following steps are needed, if J is used as such in other environments than Windows or when starting to make own packages in any environment.

### 6.1. Using Gfortran

1. Load and unzip the documentation file j\_doc.zip into J directory. If you need to change the number of objects or if you want J to run automatically certain initialization commands put j.par into the active working directory. If you want to use J’s online help, put copy of jhelp.txt to the active directory.

2. Download J\_precompiler source code files from github.com/##jPrecompilerREPO### to the J directory.

In Windows environments, steps 3 and 4 can be bypassed.

3. Delete the j\_precompiler.exe.

4. Compile and link J\_precompiler using command: gfortran j\_precompiler.f90 -o j\_precompiler.

5. Download J source code files from github.com/##jREPO### to the J directory.

6. Run j\_precompiler, answer to the first question is *<return>* and second question ‘*a*’.

7. Run Gfortran command ‘make –f makefile\_release’, ‘make –f makefile\_Debug’, ‘make –f makefile\_release\_mac’, or ‘make –f makefile\_Debug\_mac’.

With Mingw Fortran the commands are ‘mingw32-make –f makefile\_release’, etc.

8. Start using J.

#Note: the only difference between makefiles for Mac Os X/Linux and Windows is that Window version make the output with the extension ‘.exe’.

### 6.2 Using Intel Fortran

With Intel Fortran, file j\_pre\_intel.txt is used as the command file for the precompiler. The files which are output files in j\_pre\_intel.txt are compiled and linked. Specifically, file J-sysdep\_intel.f90 is the proper sysdep –file to be used..

### 6.3 Using another compiler

With other compilers we do similar things as desprovidedcribed for Gfortran. The differences are:

1. Study your compiler to see if there are functions for putting the computer into sleep, accessing the current working directory and changing the current working directory. Then edit the file J\_sysdep\_unkown.f90 accordingly, and if the compiler does not provide these functions let the file be as it is.

2. When running J\_precompiler, the correct answer to the first question is j\_pre\_unknown.txt.

3. Makefile\_release and Makefile\_debug show what files should be compiled and linked. The only difference is that J\_sysdep\_gfortran.f90 needs to be replaced by J\_sysdep\_unknown.f90.

### 6.4 Making J with smoothing splines

Owing to license uncertainty, the smoothing splines are not included in the basic distribution. If the user thinks that she/he can use smoothing spline routines which are available in Netlib and Dmoz, the user can edit the beginning of the file matsub.f as described in that file.

**Juha Lappi**

# **User’s guide to own J3.0 packages**

**Version 3.0 2017**

**Natural Resources Institute Finland  
Suonenjoki Unit**

[**User’s guide to own J3.0 packages** 1](#_Toc471200049)

[1. Introduction 2](#_Toc471200050)

[2. Making own packages 2](#_Toc471200051)

[3. The structure of the J program 2](#_Toc471200052)

[4. J objects and object types 4](#_Toc471200053)

[5. Defining own functions 11](#_Toc471200054)

[5.1 Using J utilities 12](#_Toc471200055)

[5.2 Starting an own function 12](#_Toc471200056)

[5.3 Accessing options 12](#_Toc471200057)

[5.4 Using j\_commands from own functions 13](#_Toc471200058)

[6. Defining own object types 13](#_Toc471200059)

[7. Defining own data format 14](#_Toc471200060)

[8. Defining own interface 15](#_Toc471200061)

[9. Error handling 15](#_Toc471200062)

[10. Global module variables 16](#_Toc471200063)

[11. Example function for making own functions 16](#_Toc471200064)

## 1. Introduction

Starting from J3.0 J is open source software. The purpose of this document is to guide the J user to manage the software package and to add own functions, object types and options. The files defining own functions, object types and options are called own packages. The own packages are written in Fortran 90. In addition also files written in Fortran 77 can be used. The user needs to read both the code and this document together. It is assumed that the user already has read the *J Setup Guide* as well as the *J User’s guide* and also its preface.

### Manual conventions

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

Expressions in the **J** source (Fortran) code will be written in the Euphemia font.

## 2. Making own packages

When starting to do own J packages set up the working environment as described in *J Setup Guide*. Setup guide also includes instructions for precompiling and compiling J. When making own functions in Windows environment the user can use directly the exe version of J\_precompiler. When making own function in other environments, the user needs to compile and link the precompiler.

Select one of the three own packages as your own own-package (the other two own packages can be used to link others’ own packages with your package). Let us assume that you selected own2-package. Start editing own2\_mod.f90 and own2\_.f90 as will be explained later. Compile and link. The global variables in own packages have all prefix own1\_ , own2\_ or own3\_. In this manual the generic name own*X*\_ is used when own1, own2 or own3 must be used.

It may be that a safer approach to develop own packages is to edit the two own files under different name so that reloading all files from the source file depository will not accidentally replace edited own files. Then the script file of the precompiler and the makefiles of Gfortran (or other compiler) need to be edited accordingly. Also these files should be saved under different names.

There are two ways to combine two own packages which have been developed using the same name. First, easier way is just rename one of the packages. This can be done by changing all occurrences of e.g. ‘own2\_’ into ‘own3\_’. The other possibility is to merge the functions, options and object types in an evident way.

## 3. The structure of the J program

The main program is doing initialization of data structures by calling subroutine jinit(), which checks that the core J and own packages do not have conflicting names for functions, object types and options. It also initializes the J objects, and defines all the default objects. If file j.par is available it reads from it how many named objects there can be during the session. The main program is also calling the initialization subroutines of own packages.

Thereafter J is just executing J functions. First, if the current directory has file j.par, then the main program executes the code

call j\_command(";incl('j.par')")

which is asking J to include initialization commands stored in file ‘j.par’. Note that the argument looks exactly as what the user can give at the sit> command prompt. This is always the case, the arguments of subroutine j\_command() are ordinary J commands. Thereafter the main program is going to sit> prompt by

call j\_command('sit()')

The sit() function driving the sit> prompt is also an ordinary J function.

There are three subroutines whose interaction is carrying out the core tasks of J. These subroutines are

1. j\_getinput(): this subroutine (in j\_utilities.f90) is getting the next input line from the terminal or from hierarchical include files and makes the preprocessing called ‘input programming’. This subroutine can be called from everywhere, and also the own functions can call it.

2. j\_compiler(): this subroutine (in j\_utilities.f90) is interpreting an input line and adds the interpretation to an integer vector. Not all input lines obtained by j\_getinput() are interpreted, especially input paragraphs are interpreted directly by J functions without trying to ‘compile’ them.

3. dotrans(): this recursive subroutine scans the interpreted integer vector and executes all J-functions stored in the integer vector.

The main command loop in the sit> prompt subroutine is using j\_getinput() to get next command, it then uses j\_compiler() to interpret it and then dotrans() is used to execute the interpreted code.

The structure of the dotrans() subroutine is as follows:

recursive subroutine dotrans(iob,ioi) !compute transformation stored in object iob starting from element ioi

j\_v(j\_ivrecursion)= j\_v(j\_ivrecursion) + 1 !$Recursion=$Recursion+1

io=ioi

3000 continue !

if(j\_err)then

call j\_debugerr(iob,io)

j\_v(j\_ivrecursion) = j\_v(j\_ivrecursion) – 1 ! !$Recursion=$Recursion-1

return

endif

goto (10,20,30,40,50,60, …,2740) j\_o(iob)%i(io)

!note: control comes to this point if j\_o(iob)%i(io)==0

j\_v(j\_ivrecursion) = j\_v(j\_ivrecursion) - 1. ! !$Recursion=$Recursion-11

return !

10 call func1(iob,io)

goto 3000

…

2740 call func274(iob,io)

goto 3000

The argument iob tells that the interpreted functions are stored in the integer vector j\_o(iob)%i associated with object iob. The function index is stored in the element io. The duty of each function is to update io. If the function index is zero, then recursion level is decreasing and the control is returned to the calling J –function (which may be the sit> prompt). If the function returns error (j\_err=.true.), then the j\_debugerr() prints the source corresponding to this point of the interpreted code, all open include files are closed and control is returned to the calling J function. The error condition remains until the control comes to sit> prompt which sets j\_err into .false. and the user can continue from the sit> prompt.

The standard structure of the interpreted code is such that the element io+1 tells the number of arguments (=narg), then the following narg elements are the indexes of the arguments and then comes the object index of the output. Thus the function needs to update the io as follows:

io=io+j\_o(iob)%i(io+1)+3

In almost all J functions the update of io is done this way explicitly within the subroutine. It is recommended that the user will update io using a recently written subroutine startfunc(), which has also the number of arguments as an output as well as pointer to the arguments. If io is not updated before return under an err condition then J is not able to generate proper error message (a wrong line causing the error is printed)

The arithmetic functions and some special functions do not have the number of arguments just after the function index, and the update of io is done differently. Three special functions which are generated if j\_compiler() encounters a function in the any of the three own function set. The code in dotrans() is:

2640 io=io+1; call own1\_funcs(iob,io);goto 3000

2650 io=io+1; call own2\_funcs(iob,io);goto 3000

2660 io=io+1; call own3\_funcs(iob,io);goto 3000

That is the interpreted code has first the J function index for own1, own2, or own3, and then the next element is the function index within the corresponding own function set. The structure of own1\_funcs(), own2\_funcs() and own3\_funcs() is the same as the structure of dotrans(), i.e., the element j\_o(iob)%i(io) is the function index of the own function etc.

As dotrans() is a recursive subroutine, it means that when a J-function is executed it can do other J functions either calling directly dotrans() or calling utility subroutines or functions which are calling dotrans(). J variable $Recursion contains the current recursion level. The sit> prompt J function invoked by call j\_command('sit()') in the main program is at the recursion level 1. Thus the lowest value $Recursion what the user can encounter is two (try by giving print($Recursion) at the sit> prompt).

Some properties of J functions are defined in j\_modules.f90. These definitions are same as the user needs to define for own function, thus they will be explained as it is explained how the user can add own functions.

## 4. J objects and object types

A basic component of J environment is a J object. Logically each J object has the following components: name, type, title, real value, two integer vectors, real vector, double precision vector, vector of character\*1 characters. The name, type, title and the real value are dynamically allocated at the initialization phase and the number of objects can be determined in file j.par. The vector components of an object are allocated when they are needed.

The object names are stored in a text object Names, and the names are defined and accessed using utility subroutines designed for text objects. Once an object name is defined it cannot be deleted or changed. In addition to named objects, there are unnamed temporary objects which can store intermediate results in calculations.

The real value of an object is stored into vector j\_v. All arithmetic operations do operate using these values. All numerical constants appearing in calculations are also stored in vector j\_v, thus the size of j\_v is equal to the number of named objects + number of temporary objects + allocated space for numeric constants. If there would be more numeric constants than is the space reserved, more space is allocated during the execution.

Vectors consisting of vectors can be defined in Fortran 90 using derived data types. In j\_modules.f90 there are definitions

type j\_basicobject ! defines basic J-object types

real, dimension(:),allocatable ::r ! real vector associated with each object

integer, dimension(:),allocatable ::i ! integer vector associated with each object

integer, dimension(:),allocatable::i2 !second integer vector, contains usually object indexes of subobjects

double precision, dimension(:),allocatable::d ! double precision vector associated

character\*1,dimension(:), allocatable::ch ! character\*1 -vector associated with each object

end type j\_basicobject

!j-objects:

type(j\_basicobject),dimension(:),allocatable,target:: j\_o

The dynamically allocatable objects are allocated in initialization as

allocate(j\_o(1:j\_nv))

The component vectors of an object iv are referred as j\_o(iv)%r, j\_o(iv)%i, j\_o(iv)%i2, j\_o(iv)%d and j\_o(iv)%ch. See Fortran manuals how derived data types work. Allocation of j\_o does not allocate the vectors %r, %i, %i2, %d and %ch associated with the objects. These are allocated when they are needed and they are deallocated as they are not needed.

#Note: the keyword ’target’ means that pointers can point to the parts of an object.

#First j\_namedv objects have a name. Objects j\_namedv+1…j\_nv are temporary objects for intermediate results.

For each object iv the type of iv is given by the vector element j\_otype(iv). The value j\_otype(iv) is the index of an object type. Each object type has also a name. Technically all object types are similar general J objects. The object type just tells the intended use of the objects and it also indicates how the vectors associated with the object are organized.

New objects with a given name and type are generated with j\_getobject() subroutine. This subroutine just generates object name (if name does not exist) and puts the object type into j\_otype vector. Real variables can be generated with just j\_getobject(). Otherwise the allocation of object components must be done separately. Different object types are generated with subroutines whose name start with j\_def e.g. j\_deftext. These def –subroutines are first calling j\_getobject().

The first argument of j\_getobject() and any of def subroutines is the index of the master object. The second argument is character string telling the last part of the object name. E.g. call j\_getobject(iout,’%mean’,…) generates an object whose name is obtained by concatenating the name of object with index iout and ’%mean’. If the first argument is zero, then the second argument tells directly the object name.

The following object type names stored in character vector j\_objecttypes have general interest (it is also mentioned which J functions usually generate such objects):

'REAL' = real variable, generated usually with arithmetic statements

'CHAR' =character constant or character variable

'LIST' = object list, generated by J function list()

'MATRIX' = matrix, generated by matrix() function or by other functions

'TRANS' = transformation object, generated by trans() function

'STORE' = storage for variables, generated by store()

'STEMSPLINE' = stem curve spline produced by stemspline()

'TEXT' = text object produced e.g. by text() function

'DATA' = data object produced by data()

'PROBLEM' = problem object produced by problem()

'FIGURE' = figure object produced by plotyx(), draw(), drawline(), or drawclass()

'SMOOTH' = smoothing spline produced by smooth()

'REGR' =regression object produced by regr()

'BITMATRIX' = bit matrix produced by bitmatrix()

'TRACESET' = object created by trace()

'LAASPOLY' = object created by laaspoly()

'TAUTSPLINE' = object created by tautspline()

The index of each J object type is a global parameter which starts with ‘j\_ip’, e.g. j\_ipregr. Thus to test if an object iv has type 'LIST' is done as: if(j\_otype(iv).eq.j\_iplist)then… For own object types such associated parameters do not exist, and the user must obtain the index by using function j\_objecttype\_index(objecttype) as will be explained later.

Object title is reference to a character constant which contains the title of the object. Specifically the global vector j\_otitle contains indexes of the title character constants, i.e. j\_otitle(ivobject) is the index (in the vector j\_o) of the character constant which is the title for object with index ivobject. Value zero indicates that no title is given.

The structure of some J objects and the utilities which can be used with these objects are as follows.

#### 'REAL' = real variable

For a real variable only the value of vector j\_v is used. The real variable –object is generated with j\_getobject() having the type argument as j\_ipreal.

#### 'CHAR' =character constant or character variable

Character constant is an object whose name directly tells the value of the constant. The names are stored in a text object with index j\_ivnames. The ‘ –character is around the name. Character variable refers to a character constant. Both are presented with an object whose %i –vector has four values, as follows:

j\_o(ivchar)%i(1): the start of the character string in character\*1 vector j\_o(j\_ivnames)%ch

j\_o(ivchar)%i(2): the end of the character string in character\*1 vector j\_o(j\_ivnames)%ch

j\_o(ivchar)%i(3) =0 if the object is character constant otherwise it is the index of character constant to which the object (character variable) refers to.

j\_o(ivchar)%i(4) the unit number of a file which is associated with the character constant (zero indicates that no file is associated). This is used in read() and write() functions. The association of a character variable with a file is indicated via the corresponding character constant.

Utility subroutines for character variables and constants:

logical function j\_ischarconst(iv) : returns .true. if iv is character constant. Note that expression j\_otype(iv).eq.j\_ipchar tells if iv is either character constant or character variable.

subroutine j\_getchar(iv,buffer,le) : gets the character variable or constant into a Fortran character string ‘buffer’. Argument le returns the length of the string. Because the character constants are stored into a vector of character\*1 strings, standard character operations cannot be directly used before putting the string into a standard Fortran character string. If iv is not character constant or character variable, then the real value j\_v(iv) is put into the buffer as a character\*8 string.

subroutine j\_getchar2(iv,buffer,le): as j\_getchar() but if iv is not of character type then an error occurs.

integer function j\_iounit(iv) Gives the unit number of the associated file of a character constant or character variable. Returns 0 is no file is associated.

#### 'LIST' = object list, generated by list() function

List object (e.g. iv) has a simple structure. j\_o(iv)%i(0) tells the actual size of list and then the elements j\_o(iv)%i(1)… j\_o(iv)%i( j\_o(iv)%i(0) ) contain the list. There can be unused elements in the j\_o(iv)%i - vector. The allocated upper bound for a list object can be obtained using intrinsic Fortran function UBOUND.

A list object can be generated with several subroutines:

j\_deflist2(iv,name,list,ivout) ! define a list object (ivout), and put list to it. The name of the list is either the argument name (if iv =0) or generated by concatenating the name of object with index iv and argument name (if iv > 0).

j\_deflist3(iv,name,list0,ivout) !allocates list object with size list0, but put it as empty

j\_deflist4(iv,name,inp,ivout) !makes a list from character variable inp which contains the names of objects separated with commas

j\_deflistopt(iv,name,list0,list,ivout) !allocates list object with size list0, and put list into it

#This is used to get list object from an option

j\_defmergelist(iv,name,list,list2,ivsingle,ivout) ! merging two lists+ ivsingleinto a list object

The following function tells the length of the list (number of elements):

function j\_lenlist(iob) ! length of list, -1 if not a list

There are also several functions which can be used to put elements to a list object. If the size of the list object is not large enough, the list object is automatically expanded. The functions available (the function returns the position into which the object index is put):

function j\_putlist2(i,ivlist) ! put i into list object ivlist if it is not yet in the list

function j\_putlist2plus(i,ivlist) ! put i into list object ivlist, in addition i is put after the list.

#function putlist2plus() is not probably needed by the own package programmer

function j\_putlist2b(i,ivlist) ! put i into list object, if i is list expand it, and put all

function j\_putlist3(i,ivlist) ! append i into list object

The following functions can be used to test whether an object is in a list object:

function j\_inlist2(i,ivlist) ! returns index of i in a list object ivlist

function j\_inlist2b(i,ivlist) ! as j\_inlist2(), but if(ivlist.le.0) --> error

#Note, there are also utilities which work with lists which are not stored in list object, i.e. which are integer (0:\*)-vectors where element 0 tells the number of objects in the list. These are:

function j\_putlist0(i,list) ! put i into list, no bound checking

function j\_putlist(i,list) ! put i into allocatable list, bound checking and increasing size of list if needed

function j\_inlist(i,list) ! is i in list, length given in list(0)

function j\_inlist3(i,list,list0) ! is i in list, length given in list0 , not in list(0)

subroutine j\_differ(list1,n1,list2,n2,list3,n3) !picks to list3 elements from list1 elements which are not in list2

function j\_ndiffer(list1,n1,list2,n2) !number of elements of list1 which are not in list2

subroutine j\_union(list1,n1,list2,n2,list3,n3) !the union of list1 and list2 put to list3

function j\_nunion(list1,n1,list2,n2) !size of union of list1 and list2

subroutine j\_uniondif(list1,n1,list2,n2,list3,n3,list4,n4) !list4: list1+list2-list3

#### 'MATRIX' = matrix, generated by J function matrix() or by other J functions

Matrix object iv contains the following components

j\_o(iv)%i(1) = number of rows

j\_o(iv)%i(2) = number of columns

j\_o(iv)%i(3) = number of elements (number of rows times number of columns)

j\_o(iv)%i(4) = type of the matrix, j\_matreg is a regular matrix, j\_matclass is matrix produced by classify function

j\_o(iv)%i(5) =1 if the matrix can be expanded (does not work yet)

j\_o(iv)%r(1:ne) = elements of the matrix in row order

If type of the matrix is j\_matclass then the matrix contains additional elements.

A matrix is defined by the utility function:

subroutine j\_defmatrix(iv,name,ndim1,ndim2,itype,expand,ivout) ! defines a matrix object

The following functions and subroutines work with matrices:

logical function j\_expandable(iv) ! is matrix expandable

subroutine j\_putmatrix(ivmat,irow,icol,val) ! puts value val into row irow and column icol

real function j\_getmatel(ivmat,irow,icol) ! returns element (irow,icol) from matrix object ivmat

#Note j\_putmatrix() and j\_getmatel() do not check whether arguments are legal.

#There are several J functions which work with matrices and which are generated either with explicit J functions (e.g. t(a) returns the transpose of a) or by writing arithmetic expressions involving matrices (e.g. a(2,3)=7).

#### 'TRANS' = transformation object, generated by trans() function

A transformation object iv has the following structure:

j\_o(iv)%i(0) = the length of the interpreted code

j\_o(iv)%i(1: ) contains the interpreted code

j\_o(iv)%i2(1) = index of list object containing the input variables

j\_o(iv)%i2(2) = index of list object containing the output variables

j\_o(iv)%i2(9) = j\_ivarg =index of variable used as an argument for the transformation set (see user’s guide)

j\_o(iv)%i2(10) = j\_ivresult = index of variable used as a result for the transformation set (see user’s guide)

j\_o(iv)%i2(11) = text object storing the source code

Other elements of i2 are used e.g. in a simulator

A transformation set is defined using j\_deftrans() subroutine.

j\_compil() is a subroutine that is used to interpret an input line and put the interpreted code into an integer vector. It is however recommended that the J function trans() is used to define trasformations as is explained in the J function example().

#### 'STORE' = storage for variables, generated by store()

Probably the J user should use the store only through J functions store() and load().

#### 'STEMSPLINE' = stem curve spline produced by stemspline()

Probably the user should use stem splines only through stemspline() and value() functions.

#### 'TEXT' = text object produced e.g. by text() function

A text object is defined using

subroutine j\_deftext(iv,name,lines,leng,ivout)

Where leng is the length of character\*1 fork of the object, and lines is the (current) maximum number of lines. In j\_deftext() the text object is allocated as

allocate( j\_o(ivout)%i(0:lines+1),j\_o(ivout)%ch(1:leng))

where ivout is the text object.

In a text object ivtext element j\_o(ivtext)%i(0) tells the current amount of the lines in the text object. Element j\_o(ivtext)%i(linenum) refers to the first character in line linenum. If nlines is the total number of lines in the text object, then j\_o(ivtext)%i(nlines+1) refers to the first character of the next new line.

A line into a text object can be put using following subroutines

subroutine j\_puttext(iv,text)

where iv is the index of the text object and text is the character string put into the text object. If either j\_o(iv)%i or j\_o(iv)ch does not have enough space it is first automatically expanded.

subroutine j\_puttext2(iv,text)

This subroutine stores also line numbers into j\_o(iv)%i2. This is used to handle ;goto addresses in input programming.

subroutine j\_putcleantext(iv,text)

This removes blanks and tabs etc from text before putting text into the text object iv.

subroutine j\_putnewcleantext(iv,text,iline)

puts the cleaned text into the text object if it is not already there. Returns the line number of the existing or inserted text in argument iline.

A line from text buffer can be obtained by these two subroutines.

subroutine j\_getline(iv,line,buffer,le) ! get line from text object iv into buffer, le will be the length

subroutine j\_getline2(iv,line,buffer,le) ! as j\_getline() but fills end of buffer with blanks

Line number of text object consisting of string name can be obtained using function

function j\_line(iv,name) !get the index of the first line having content equal to name in a text object iv, not found =>0

function j\_line2(ivtext,ivchar) ! as j\_line() but now the input text is a character variable or character constant.

Number of lines in a text object can be obtained using

integer function j\_nlines(iv) !number of lines in a text object iv

#### 'DATA' = data object produced by data()

A data object is a compound object which contains only links to the component objects. A data object is defined by j\_defdata() subroutine. A user probably does not need it in own packages as data objects can be made by J function data(). A user probably needs, however, to access the components of a data object. These can be accessed using the following integer functions (see options of the data() function):

The matrix containing the data: integer function j\_data\_matrix(iv)

The list of variables stored in the data: integer function j\_data\_keep(iv)

Transformation set for prolog->: integer function j\_data\_prolog(iv)

Trasformation set corresponding to maketrans->: integer function j\_data\_maketrans(iv)

Trasformation set corresponding to trans->: integer function j\_data\_trans(iv)

Trasformation set corresponding to epilog:->: integer function j\_data\_epilog(iv)

The list of variables in the data (keep variables + output variables of trans()): integer function j\_data\_vars(iv)

The subdata of the data set: integer function j\_data\_sub(iv)

The nobsw –variable: integer function j\_data\_nobsw(iv)

The upper data set: integer function j\_data\_up(iv)

The obs- variable (the variable storing the observation number): integer function j\_data\_obs(iv)

The obsw –variable: integer function j\_data\_obsw(iv)

The nowswcum –variable: integer function j\_data\_nobswcum(iv)

As shown in the J function example, data sets can be accessed in any function using first j\_getdatasets() subroutine before clearing options and then using j\_getdataset() and j\_nextobs() subroutines.

#### 'PROBLEM' = problem object produced by problem()

A problem object is created by problem() function: The problem objects are used by jlp() function. A writer of an own package probably does not need to access directly problem objects. See users guide what subobjects are created by problem() function.

#### 'FIGURE' = figure object produced by plotyx(),draw(),drawline(), or drawclass()

Figure objects will be explained later when someone wants to write own graphics functions.

#### 'SMOOTH' = smoothing spline produced by smooth()

A smoothing spline object is created by J function smooth(). A smoothing spline can be used either using J function value(), or using function j\_valuesspl(). If a smoothing spline is accessed within an own function, it is recommended that the function j\_valuesspl() is used.

#### 'REGR' =regression object produced by regr()

A regression object is used in J function value() to compute values of the regression function. See user’s guide how to access other properties of a regression object. These functions show also how a regression object can be used directly.

#### 'BITMATRIX' = bit matrix produced by bitmatrix()

If iv is a bitmatrix object, then j\_o(iv)%i(1) is the numbers of rows in the matrix and j\_o(iv)%i(2) is the number of columns. J function value() shows how an individual element can be accessed using j\_ibittest() function.

#### 'TRACESET' = object created by ;trace()

Traceset object may not have general interest for writers of own packages.

#### 'LAASPOLY' = object created by laaspoly()

A laaspoly object can be used using crkd() function in matsub.f as shown in J function value().

#### 'TAUTSPLINE' = object created by tautspline()

A tautspline object can be used with J function value or using directly using ppvalu() function located in file matsub.f. Also if using ppvalu() directly, how it should be used can be seen in value() function (search for ‘ppvalu’).

## 5. Defining own functions

First, the user must decide which of the three possible own packages (s)he is developing. Provided in the distribution package are skeletons for such packages. Assume that two users have developed the same own package (lets say that own1\_ package), and these packages should be put to work together. Then the other user just needs to change the name of her package into ‘own2\_’ (for instance). This happens by changing the names of files ‘own1\_mod.f90’ and ‘own1\_.f90’ into ‘own2\_mod.f90’ and ‘own2\_.f90’ and by replacing all occurrences of ‘own1\_’ into ‘own2\_’ in these files.

Let us assume that we are now developing own2\_ package.

First thing when developing a package is to give name to the package. This is done be editing at the beginning of file own2\_mod.f90 line

character\*15 :: own2\_title='own2 20.12.2014'

Characters before first space identify the package, the second part is just printed when starting J.

A new function can be added as follows. Alternatively, you can edit one of the existing skeleton functions. First the file own2\_mod needs to be edited.

1. increase the number of functions in line

integer, parameter :: own2\_nfunctions = 4

2. give then name to the function (J initialization checks that the name is free)

3. give minimum and maximum number of arguments. That the number of arguments is legal is checked when interpreting the J script. No error is generated when the number of actual arguments is one and the minimum is larger than one because the actual argument can be a list of objects during the time of execution.

4. Define in own1\_nnamedfuncarg the functions whose arguments must be named objects. Define in own1\_nnamedfuncarg the number of such functions.

### 5.1 Using J utilities

There are several utility subroutines and functions which can be used when dealing with J objects. These are stored in file j\_utilities.f90 and their interfaces are stored in j\_modules.f90. Two of these utility subroutines are explained here and others are explained together with the object types which they are dealing with. Several utility subroutines are explained only in file J\_utilities.f90.

### 5.2 Starting an own function

Let us assume that the user is developing function example2 in own2 package provided by the distribution.

All compiled functions except the arithmetic functions have the same structure. All functions start as:

subroutine j\_function(iob,io)

where iob is the object whose i vector contains the compiled (interpreted) code. The element j\_o%i(io) contains the function index, element j\_o%i(io+1) contains the number of arguments, then the arguments follow, and after arguments is the output. If no output was given, the output is Result, and object whose index is j\_ivresult. Thereafter is the index for next function, or zero, which means that control returns to the calling program.

If the function output cannot be the same as one of the arguments or option arguments or one of the input or output variables of transformation arguments the function should be started with

call j\_checkoutput(iob,io)

if(j\_err)return

It may happen that if the function is not prepared to have the same output as one of arguments or option arguments, the function can cause a system crash if \_checkoutput(iob,io) is not used.

Then it is recommended that there is call to j\_startfunction(). See own2\_example() in own2 package and j\_startfunction() documentation in file j\_utilities.f90.

Each function must update the location variable io. The new value must be

io=io+j\_o(iob)%(io+1)+3

Subroutine j\_startfunction() is updating io.

#Note: In most J functions j\_startfunction() is not used because this subroutine was done quite recently.

### 5.3 Options

There are two kinds of options in J: regular options and code options. A regular option can access the arguments of the option, and these option arguments are just accessed once when starting a J function. A code option is an option which stores a piece of code that can be computed several times within the function. A typical code option is func-> option used e.g. in draw() function. This option defines the function to be drawn and the code of this function is computed several times in draw() function.

Own options are defined in ownX\_options in file ownX\_mod.f90. In ownX\_namedoptarg are listed those options whose arguments must be named objects. In own1\_newvar are listed such options whose named arguments can be objects which are not known beforehand. In own3\_codeoptions are listed the names of code options.

The structure of the option in the compiled code is similar to the structure of a function. If function has an option, let us say, data->, then j\_linkopt(j\_mdata) refers to the point in the current code where the option starts. First element is the number of option arguments, then there are the arguments. If the option arguments are numeric values, then possible numeric calculations are done before the value is transmitted to the option, e.g. xrange->(h%min-1,h%max+1). The option values can be accessed using j\_linkopt(), as done in most J functions. It is, however, recommended that the options are accessed using the recent generic j\_getoption() subroutine. Options in the base J can be accessed using the option index as the second argument. The option index is j\_m + the option name. E.g the index of option xrange-> is j\_mxrange. Own options can be accessed only using the name of the option as the second argument. The interface to the generic subroutine j\_getoption() is defined in file j\_modules.f90. Through the j\_getoption() interface one can access subroutines j\_getoption\_name() and j\_getoption\_index() subroutines in j\_utlities.f90. See J function example() in j\_.f90 and own2\_example() function in own\_2.f90 for more information.

For code options j\_linkopt2(option\_index) returns the index of the element where the code for the option starts. Most base J functions are using j\_linkopt2() to access code options. It is recommended that new J users use the same generic j\_getoption() subroutine as is used for regular options. The last optional argument returns the address at which the code for the option starts. See subroutines j\_getoption\_name() and j\_getoption\_index() subroutines in j\_utlities.f90 and example() function in j\_.f90 for an example.

After the information of all options needed in a function is obtained, the options must be cleared before using the recursive dotrans() subroutine. Otherwise the options would be present in function executed later. There are two subroutines for clearing the options. Subroutine j\_clearopt() must be used if options are accessed using j\_linkopt() directly or if data sets are accessed with j\_getdatasets(), and j\_clearopt2() can be used if all options are accessed through j\_getoption() subroutine. Subroutine j\_clearopt2() warns if there were options present which were not checked with j\_getoption(). It is a good strategy to use j\_clearopt2() at the beginning of the function if the function does not recognize any options. This will print warnings about useless options and prevents useless options from being transmitted to consecutive functions.

### 5.4 Using j\_commands from own functions

It is possible to use J commands within own functions using j\_command() utility subroutine. The argument for the subroutine is a character string in form which could be given at sit> prompt. The default is that the options are cleared with j\_clearopt() or j\_clearopt2() before starting to use j\_command(). If this has not happened an ‘\*j\*’ error occurs. It is important to put after each j\_command() call the code ‘if(j\_err)return’.

If it is certain that the command given as the argument of the j\_command() does not clear options and it does not use options, then j\_command() can be used in form j\_command(inp,.true.) where the second optional argument indicates that no error occurs even if there are options available when the subroutine is called. Currently in the basic J this is utilized in utility subroutine j\_deflist4() which creates a list object from character string.

Several lines of J code can be run as shown in the example() function in j\_.f90.

## 6. Defining own object types

Own object types can be declared in file own*X*\_mod.f90. If a new object type is declared then parameter ownX\_nobjecttypes is increased by one. Then the new name of the object type is added in the data statement for vector ownX\_objecttypes.

The index of object types in the base J system can be accessed through global j\_ip parameters (e.g. j\_iptext) as described above. The index of an own object type can be obtained with j\_objecttype\_index() function. New objects with a given name and type can be generated with j\_getobject() subroutine. This subroutine just generates object name (if name does not exist) and puts the object type into j\_otype vector. The allocation of object components must be done separately. It is recommended that own object types are defined using similar def- subroutines as are def subroutines for many J object types.

If an object with own object type is deleted, then subroutine ownX\_del() is called so that possible subobjects can be deleted. The subroutine definition is:

recursive subroutine ownX\_del(iv,iotype) !deletes subobjects of compound own-object

where iv is the index of the object and iotype is the type of the object according to the numbering of own object types. If the object has %i, %i2, %r , %d or %ch components the subroutine ownX\_del() does not need to deallocate these, they are deallocated by j\_del() which is deleting J objects and which is calling ownX\_del().

Specifically, if iv2 is a subobject of object iv, then it can be deleted by

call j\_del(iv2)

The subroutines ownX\_del() and j\_del() are defined as recursive subroutines so also subobjects can be compound objects whose subobjects need to be deleted.

## 7. Defining own data format

Data objects are created by J function data() (see user’s guide). The format for reading data variables can be binary, default text format $ or any given Fortran format. The format is given in form-> option. Two hierarchical data sets can be created by one data function using sub –options to define how subdata can be read in. If the first word in the value of the form-> or subform-> option is the same as the first word of ownX\_title defined in the beginning of own*X*\_mod.f90 (see the beginning of Chapter 5), then the data values can be read using subroutines in own*X*\_.f90 from any kind of file structure or data base.

In the file own2\_.f90 there are subroutines that read the data and subdata exactly as using the default J data formats.

There are two subroutines for reading the upper level data.

subroutine own*X*\_open()

subroutine own*X*\_getobs()

Subroutine ownX\_open() is used to open the data file given in in-> option. Of course it is not necessary that the data file is given in in-> option but I cannot see any reason why in-> option would not be used for that purpose. It is possible that the full file name is not given in in-> option. Global character variable j\_data\_in\_c has the value which is given in in-> option. Global integer variable j\_data\_in\_lenc gives the length of the name.

It is recommended that utility subroutine j\_openread() is used to open the file and it is recommended that j\_data\_nu is used to store the unit number.

Subroutine own*X*\_getobs()

The subroutine can utilize global pointer j\_data\_read\_ which is pointing to the variables given in read-> option. The number of read variables is given by j\_data\_nread.

Note that there is no subroutine for closing the data file. It is assumed that subroutine ownX\_getobs is closing the file when data are read in. If j\_openread() was used to open the file, it must be closed using j\_closeunit(). The fact that all data are read in is transmitted to the calling program setting global logical variable j\_data\_eof=.true.

If subdata is read in in the same data() function call, then there are two subroutines available

subroutine own*X*\_opensub()

subroutine own*X*\_getsubobs()

It is possible that the data for the subdata is read in from the same file as the upper level data. It is also possible that the subdata are in a different file than the upper level data but the subdata file is opened already in subroutine ownX\_open() or the subdata file has standard format for subdata, and the subdata file is handled as any J subdata. Subroutine ownX\_opensub() is called only if subform-> refers to the name of the own package. If ownX\_opensub() is called, then global logical variable j\_data\_subopen is set to .true., otherwise it is .false..

Subroutine ownX\_opensub() can refer to j\_data\_subin\_c (=name of the file) and j\_data\_subin\_lenc (=length of the name). Global variable j\_data\_subnu is reserved for the unit number of the subfile, and it is recommended that it is used.

As for each upper level unit a given number of subunits is read in, it should not happen that end of file condition appears when reading subdata. If this happens an error condition arises.

When end of file condition is encountered in ownX\_getobs() and j\_data\_subopen is .true., then also the subdata file (unit j\_data\_subnu) needs to be closed.

## 8. Defining own interface

The standard interface with sit> prompt is an ordinary J function (called sit()) which is called automatically during initialization unless the control is not taken by some user function called in j.par. The sit() function can be found in j\_.f90 by searching for ‘sit(iob,io)’. It is possible that the user starts to run J under her own interface by defining a similar own function which is called either from j.par or later. It is possible that the user interface is used only for some time, and then control is returned to sit> prompt.

A user can start using e.g. a graphical interface for some tasks. It is recommended, however, that the user does not make a trivial copy of the sit> interface with her own prompt in order to hide that user is actually using standard J interface.

## 9. Error handling

Global logical variable j\_err is used to transmit the information that a so serious error has happened that the control must be returned to sit> prompt. There are two ways how the error condition can arise in own function. First the own function may realize that there is something wrong in arguments of the function or in the options etc. Then the own function/subroutine must write a proper error message, set j\_err into .true. and return to the calling subroutine. If the own function has been initialized using j\_startfunction(), then the error handling system of J can trace the location of the code which has caused the trouble.

Second, the own function may call a utility subroutine (either a general J utility subroutine or an own utility subroutine) which may set j\_err=.true.. When calling such subroutines which may return the error condition, then the own function should always test if j\_err is .true.. If the error condition has appeared, the own function may write its own error message, but often it is not needed, as the utility subroutine can often tell what went wrong clearly enough (e.g. the argument objects of the subroutine do not have correct type). But it is important that after the error condition appears, the control must be returned immediately to the upper level (otherwise e.g. Fortran access violation may terminate the whole computing session).

There are two ways set j\_err into .true., either by writing j\_err=.true., or by writing call j\_crash(). By default, both work similarly. But if J variable $Crash has a nonzero value, then j\_crash() is causing a division by zero condition which means that the execution stops with an Fortran error. If the current J version is a debug version, then the calling sequence of Fortran subroutines can be seen. This can be useful if the error condition appears in an utility subroutine which can be called using different calling sequencies and it is not evident which was the calling sequence which caused the trouble. Usually using j\_crash() is useful mainly during the development phase. The final own functions should be able to check errors without such a brutal trick as with causing system error.

## 10. Global module variables

Global variables and vectors which are stored in modules and which can be used in any J or own subroutines are defined in file j\_modules.f90. To access these variables a use statement needs to be in the definition section of a subroutine. The precompiler generates the necessary use statements, so the user can just directly use the variables without knowing in what modules they are located. All global J variables start with the prefix j\_. It is recommended that similar prefix-module system is used also in own packages. The precompiler can be used to change an existing prefix, so it does not cause any trouble if two such own packages are put to work together which initially have the same prefix.

There are 6 modules in j\_modules.f90 which are used to store global variables:

module lenmod !the lengths of character variables

module j\_mod ! defines e.g. functions, options and object types of basic J

module errmod ! includes only j\_err() which transmits the error condition upwards

module j\_globalsmod ! global variables which also own functions can access

module j\_specialmod ! variables for special purposes, not intended for own functions

module compimod ! module used by j\_compiler(), not intended for own functions

It is not recommended that the programmer of own functions use variables stored in j\_specialmod or in j\_compimod. Module j\_mod is similar as modules own1\_mod, own2\_mod and own3\_mod, which are needed to define own functions, options and object types.

## 11. Example function for making own functions

**J** function example() demonstrates how to make own functions and use J features. Source code for function example() can be found in file j\_.f90.

[=]example([x1,…,xn][,func->] [,data->][,trans->][,filter->][,reject->]

[,transafter->][,title->])

Output : If output is given, then the output will be a column vector consisting of averages

Arguments: The averages are computed for all argument variables, and if no arguments are given, for all variables, i.e., the variables in the datasets (i.e. the keep variables of the data set plus output variables of the transformation set associate with the data set) and output variables of trans-> transformations.

Options:

func function which is computed and printed

data data sets used

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

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

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

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

Examples of the example() function are given in chapter 12 of jex.txt

If func-> option is given then it defines a function whose value is computed. The values of input variables can be given at funcinput> prompt in form x1,x2=1,sin(3), i.e. in a similar form as many variables can be defined in a single command line of J. If error occurs when this input line is in an include file control is returned to the sit> prompt. If example function is used at sit> prompt, the will get new possibility to define the input. After getting the input variables (it is not checked if input variables are getting new values), the value of the function defined in the func-> option is computed. Then new values of the input variables can be given at funcinput> prompt. Computation of the function values is stopped by giving ‘/’.

Then the example function gives the user the possibility to define a transformation set and to compute it. The transformations are asked at ‘calc>’ prompt. The end of transformations is defined by ‘/’ (giving it directly will bypass this phase). If there is error either in defining or computing the transformations, the control returns to ‘sit>’ prompt. Otherwise the values of input and output variables are printed.

If this ‘calc’ phase is done successfully, then the function starts to compute averages of variables in datasets (all datasets merged). The datasets can be given in data-> option. If the option is not given the last defined data set is used.

If title-> option is given, then the title will be associated with the output.

2. Files of the package

This chapter describes the files distributed with the package. There are three executable Windows programs included in the package: J, Jfig and J\_precompiler. J program itself is distributed in four different executable Windows versions. The source of all these programs is also distributed. Jfig is showing figures in Windows environment. So the source of it cannot be directly used in other environments. Note that figures can be also produced by R scripts which are made by graphics functions of J. This way J graphics can be utilized in other environments. When making own functions in Windows environment the user can use directly the exe version of J\_precompiler. When making own function in other environments, the user needs to compile and link the precompiler.

2.1. Documentation (+j.par)

The packed file j\_doc.zip contains files

J3.0\_userguide.pdf

J3.0\_ownpackages.pdf (this file)

lappilempinen.pdf The paper Juha Lappi & Reetta Lempinen (2014) A linear programming algorithm and

software for forest-level planning problems including factories, Scandinavian Journal of Forest

Research, 29:sup1, 178-184, DOI: 10.1080/02827581.2014.886714

jlp92.pdf Old (1992) manual of JLP software, explains the principles of the JLP algorithm used also in J

jhelp.txt help file used by help() –function. The user can edit this and make also alternative help files for different purposes.

j.par file which can be used to define number of objects and initialization commands.

#Note j.par is not really documentation but it has to be somewhere

Word files for J3.0\_userguide.pdf and J3.0\_ownpackages.pdf are available on request.

2.2 Executable J programs for Windows

The zip file j\_exe.zip contains four different versions of the executable Windows J3.0 software:

J3.0.exe Release version compiled with Gfortran

j3.0\_debug.exe Debug version compiled with Gfortran

J3.0\_intel.exe Release version compiled with Intel Fortran

J3.0\_intel\_debug.exe Debug version compiled with Intel Fortran.

The release versions are much faster than debug versions, so they should be primarily used. Debug versions should be used only when the execution of the release version terminates in a Fortran error without telling what was the source line causing the termination. The differences between Intel and Gfortran versions are not clear. With some Fortran errors, the error messages of Intel Fortran debug version are more specific. It is recommended that the program is run in the command window so that the error messages remain visible and do not just disappear.

If Linux and Mac executables are needed they can be provided.

2.3. Jfig: showing figures in Windows

The graphics created by J can be seen in Windows environment using Jfig program. Jfig and J change data with file jfig.jfig and change messages about availability of a new figure with file jflag.jflag . Note that graphics can be created by J and then it can be shown using only R by using r-> option and show->0 options in J graphics functions.

There are two files in the packed file jfig.zip:

jfig.f90 the source file for Intel Fortran

jfig.exe the Windows executable

The source file is not commented, but the use of graphics and window handling subroutines can be easily detected from the code. I hope that someone could make a program for other programming environments.

File . j\_examples.zip contains examples.

5. Setting up the J environment

The J working environment can thus be set up e.g. as follows. It is assumed that the user first makes a special directory for J software called J directory.

3.1. Using ready executables in Windows

1. Load and unzip the documentation file j\_doc.zip into J directory. If you need to change the number of objects or if you want J to run automatically certain initialization commands put j.par into the active working directory. If you want to use J’s online help, put copy of jhelp.txt to the active directory.

2. Load and unzip j\_exe.zip to J directory. Make shortcuts for those executables you intent to use. Edit the properties of the shortcuts as explained in the user’s guide. Make copies of the shortcuts into the active working directory.

3. If you are making figures and want to see them directly in Windows, load and unzip jfig.zip in J directory. Make a shortcut of Jfig.exe to the active directory, and edit the properties of the shortcut only with respect to ‘start in’ part (jfig is not running in console window, so its properties need not to be edited).

4. Open Command Prompt. Set the directory to the active directory (if you need to change the disk, write e.g. c: , going upwards in the directory write cd .. and then write e.g. cd projectA\subprojectB). Start the proper J executable by writing e.g. J3.0.lnk. Or alternatively you can set the path to the folder containing J executable and run J in your working directory. If you are using graphics start jfig directly by double clicking the shortcut (there is no need to run jfig in the Command Prompt). Start working.