User Defined Variables in the Optimisation Shell INVERSE

(FOR VERSION 3.11)

Igor Grešovnik

Ljubljana, 27 September, 2005

Contents:

5. USER-DEFINED VARIABLES

User-defined variables are used to store different types of data in the shell. Results of various operations and algorithms can be stored in these variables for further use in the solution procedure.

Another important use of user-defined variables is for transferring data between different modules, operations and algorithms of the shell. A typical example of that is passing data between optimisation algorithms and the function which performs direct analyses. Variables with a pre-defined meaning are used for this task.

Variable types that are currently implemented in the shell are scalar, vector, matrix, file and field.

5.1.1.1 How this Chapter is Organised

In the first sub-chapter some basic concepts of user-defined variables and functions for variable manipulation are explained. Understanding these concepts can help the user to understand the behaviour of the supporting interpreter's and calculator's functions. However, it is not absolutely necessary to read through this chapter to comprehend the next ones. Therefore if you find the chapter boring and dry, simply skip it. You can return back later when you will have some insight about what everything is about, or when you feel the need for clearing some conceptual things.

The first part of the first sub-chapter explains what user-defined variables of the shell actually are. The second part offers somehow more practical information about how to address parts of the data stored in user-defined variables in the argument blocks of the user-defined functions. The third part explains the concepts of operations which affect groups of data objects of a given type. Storing groups of objects in a single variable and performing operations on them is a strong feature of the shell. An experienced user can easily take the advantage of this feature.

In the second sub-chapter a rough overview over the basic functions for variable manipulation is made. The examples refer to matrix variables, but similar functions exist for most of the variable types. The user is advised to read this sub-chapter to get a compact view on the subject.

The third sub-chapter describes the functions which enable an advanced way of setting components of the variables. The chapter is in that place because the described functions are used for several variable types. The user is advised to skip the chapter and return to it when needed.

The following sub-chapters include description of functions for handling of different types of variables. The first of these sub-chapters is dedicated to matrix variables. Many things described here are similar for other variable types. Especially the treatment of vector, scalar and field variables is in many terms the same. The user can therefore take this as an representative example which can ease the introduction to the treatment of other variable types. In many cases it will happen, for example, that the user will not need to read a description of a specific function for vector manipulation if he is already familiar with the appropriate function for matrix manipulation.

The last sub-chapter is dedicated to variables with a pre-defined meaning. These variables provide the necessary data links between different modules and operations of the shell. They are of great importance for setting optimisation and inverse problems.

Finally, let me conclude with a ***useful hint***. Maybe it is not bad at all to start at the chapter about matrix variables. At least you will quickly get a feeling about which information you miss to get a clear insight. Have a nice reading!

5.2 Important Note: Naming Recommendations

Although it is currently possible to name two variables of different types by the same name, this should be avoided in *Inverse* command files. The variable system will undergo substantial changes in the future, and backward compatibility will only be provided for scripts that follow this recommendation. Beside that, variable names should be restricted to those which can be used in C and other language. Names should consist of alphabetic characters ('A' to 'Z' and 'a' to 'z'), underscores ('_') and digits ('0' to '9') and should begin either by an alphabetic character or an underscore.

5.3 Basic Concepts of User-defined Variables

Each type of the shell's user-defined variables has its own set of file interpreter's and expression evaluator's functions for manipulating variables of that type. Basic manipulation includes creating, copying, renaming and deleting variables. Beside that, for each type of variables there exists a specific set of operations typical for that type, i.e. algebraic operations and setting or obtaining components for vector and matrix variables.

5.3.1 Tables of Elements

Each user-defined variable can hold a multi-dimensional table of elements of a specific data type. Number of dimensions of this table is called ***rank*** of a variable. Variable's element tables can be thought of as a recursive tables, where the basic level holds a specific number of sub-tables, each of which again holds a number of sub-tables, etc., until the last level which holds a table of elements of a specific type. Numbers of sub-tables or element which level holds are called *variable dimensions*, and the umber of levels (or dimensions) of a variable is called ***variable rank***. Variables with rank 0 can hold only one element. Variables with rank greater than 0 can hold as many data elements as is the product of variable dimensions.

Example:

Suppose we have a vector variable *v* with rank 3 and dimensions 2, 3 and 2. The variable can hold 3*2*2=12 elements which are organised in the following way:

Table v: contains 3 sub-tables (because the first dimension is)

Sub-table v[1]: contains 2 sub-tables (because the second dimension is 2)

Sub-table v[1,1]: contains 2 elements (because the third dimension is 2)

Element *v[1,1,1]*

Element *v[1,1,2]*

Sub-table v[1,2]: contains 2 elements (because the third dimension is 2)
Element *v[1,2,1]*
Element *v[1,2,2]*
Sub-table v[2]: contains 2 sub-tables (because the second dimension is 2)
Sub-table v[2,1]: contains 2 elements (because the third dimension is 2)
Element *v[2,1,1]*
Element *v[2,1,2]*
Sub-table v[2,2]: contains 2 elements (because the third dimension is 2)
Element *v[2,2,1]*
Element *v[2,2,2]*
Sub-table v[3]: contains 2 sub-tables (because the second dimension is 2)
Sub-table v[3,1]: contains 2 elements (because the third dimension is 2)
Element *v[3,1,1]*
Element *v[3,1,2]*
Sub-table v[3,2]: contains 2 elements (because the third dimension is 2)
Element *v[3,2,1]*
Element *v[3,2,2]*

The variable *v* containd three sub-tables, each of which contains two sub-sub tables, each of which contains two vectors. Variable lazout is also shwn in Table 1.

Table 1: Variable *v* with dimensions 3*2*2.

Level 1:	*variable* ***v***					
Level 2:	***v[1]***		***v[2]***		***v[3]***	
Level 3:	***v[1,1]***	***v[1,2]***	***v[2,1]***	***v[2,2]***	***v[3,1]***	***v[3,2]***
Elements:	***v[1,1,1]*** ***v[1,1,2]***	***v[1,2,1]*** ***v[1,2,2]***	***v[2,1,1]*** ***v[2,1,2]***	***v[2,2,1]*** ***v[2,2,2]***	***v[3,1,1]*** ***v[3,1,1,2]***	***v[3,2,1]*** ***v[3,2,2]***

In the case of variables that hold elements which themselves contain tables of elements, e.g. vectors and matrices, the same variable can contain elements of different dimensions. Exept at variables with a pre-defined meaning, it is completely upon user's will which are the dimensions of elements stored in a specific variable. However, this freedom is seldomly used because it decreases the level of organisation.

In the case of multi-dimensional tables, each sub-table of the same level holds the same number of sub-tables or elements, which equals the dimension of that level. Of course, some elements can contain no data.

In the above example, the first level (i.e. the variable itself) contains three sub-tables of the second level (because the first dimension is 3), i.e. *v[1]*, *v[2]* and *v[3]*. Each of these contains two sub-tables of the third level (because the second dimension is 2), i.e. *v[1]* contains *v[1,1]* and *v[1,2]*, *v[2]* contains *v[2,1]* and *v[2,2]* and *v[3]* contains *v[3,1]* and *v[3,2]*. And each of these sub-tables of the third level contains two elements because the third dimension is 2.

5.3.1.1 Addressing Variable Elements

Special conventions exist about referencing shell's user-defined variables and their elements in the argument blocks of the file interpreter's functions.

First of all, each variable has its name. Variables of different types can have the same names, e.g. we can have both matrix and vector variable names *a1*. Usual rules apply for variable names. They are strings which consist of letters and numbers, while the first character of a name must always be a letter.

Since variables can hold multi-dimensional tables of elements, we must specify which element of the variable's element table is in question. This is done by indices which specify the position of an element in the element table. Indices must be listed in square brackets and be separated by spaces or commas. They specify, by turns, from which sub-table of a specific level the element must be taken. Variable name followed by a list of indices in square brackets form the ***specification of an element***. Spaces are allowed between the variable name and index list.

Example:

Let us have a vector variable *v* of rank 3 and dimensions 3*2*2. The specification *v[3,1,2]* refers to the second vector of the first sub-sub-table of the third sub-table of the variable *v*.

Variables of rank 0 can hold only one element, therefore there is no need to specify indices when addressing their only element. Element specification can in this case be only a variable name, or eventually we can put empty square brackets behind the name (e.g. *v1, v1[ ]*).

5.3.1.2 Addressing Variables and Variable Element Sub-tables

Some functions perform operations on whole sub-tables of elements rather than on individual elements of variables. Specification of variables element sub-tables is similar to specification of individual arguments. The only difference is that few last indices are not specified, i.e. less indices are usually listed than the variable rank. It is always considered than the last indices are missing. A sub-table spcified this way consists of elements which have fixed first indices as specified in the specification, while the rest indices vary in their range (from 1 to the appropriate dimension). As usual, the precedent indices change faster while elements are listed by order.

Example:

Suppose we have a variable *v* of rank 3 and dimensions 3*2*2. The specification *v[2]* then refers to a sub-table of rank 2 and dimensions 2*2 with elements *v[2,1,1], v[2,1,2], v[2,2,1]* and *v[2,2,2]*. The specification *v[2,1]*refers to the first sub-table of this table, which contains elements *v[2,1,1]* and *v[2,1,2]*.

A sub-table specification can also refer to a whole element table of a variable. In this case the specification consists only of a variable name, which can be eventionally followed by empty square brackets, i.e. *v* or *v[ ]*.

Some operations perform not on data elements or element sub-tables, but on whole variables. The appropriate functions require ***variable specifications*** in their argument blocks. These consist only of variable names and may not be followed by square brackets, not even empty ones.

5.3.1.3 Addressing Variables Using String Objects

When addressing variables, their element sub-tables or individual elements, elements of string variables can be referred to instead of variable names. Such references must consist of the hash sign (#) followed by specification of a string object that is used for variable name. Object specification must include square brackets, even if they are empty in case of a zero-rank string variable, to avoid ambiguities.

Example:

The following commands create a zero-rank vector variable named *v1* and set its only vector element to $[1.1, 2.2, 3.3]^T$. Instead of stating variable name directly, a reference to a string object *s[ ]* that has been defined in the preceding line is used in the **setvector** command. Although *s* is a zero-rank variable, square brackets must follow variable name in string element specification:

setstring *{ s v1 }*
setvector *{ #s[ ] 3 { 1.1 2.2 3.3 } }*

5.3.1.4 Operations on Variable Sub-tables

Some general rules apply to operations on the element sub-tables of variables. These rules are mostly concerned with the dimension compatibility. The file interpreter's functions which operate on variable sub-tables can be divided into several sub-groups.

5.3.1.4.1 Functions that just Perform Operations on Elements (Simple Unary operations)

These functions iterate over indices of the element sub-tables, take the appropriate elements one by one and perform an operation on them. Sometimes it is important to know which is the order in which elements are taken, because the operations which are performed on the elements can depend on the state of other elements of the sub-table.

The operation begins with the element that has all indices of the sub-table set to 1. indices are then incremented in turns, starting with the last one. When a certain index reaches its range (i.e. the appropriate dimension of the sub-table), it is set to 1 and its preceding index is incremented. This is repeated until all indices reach their range.

Example:

Suppose that we have a variable *v* of dimensions 3*2*2, and we perform an unary operation named *unopsimp* on the elements of its sub-table *v[2]*. The code which does that is

unopsimp *{ v[2] }*

and the operation is performed on the elements in the following order:

v[2,1,1], v[2,1,2], v[2,2,1], v[2,2,2].

The first index is fixet since it was given in the specification, while the rest two indices change alternately in the above described way.

5.3.1.4.2 Functions that Perform Operations on Elements and Store Results in Another Sub-table (Unary Operations Which Store Results)

These operations usually leave the elements on which they act unchanged, but store the results of the performed operations in the elements of another table (there can be exceptions).

The order in which operations are performed is the same as in the previous case. The difference is that another element sub-table must be specified in which the results of operations are stored. The result table must be of the same dimensions than the one on which the operations are performed. The results of operations on elements of the first sub-table are then stored in the appropriate elements of the result sub-table, i.e. in the elements with the same indices.

Usually, the elements of the result sub-table don't need to be initialised. While the results are stored in the table, the elements are appropriately initialised if necessary.

Instead of properly specifying the result dabble of the appropriate dimensions, we can only specify the variable name. If a variable with that name does not yet exist, it is created before the beginning of the operation. It is created with the element table of the same dimensions than the dimensions of the element sub-table on which the operation is performed. If a whole variable table is specified (not a sub-table) which exists, but does not have the right dimensions, it is deleted and created anew with the right dimensions. If a result sub-table is specified with wrong dimensions, this is an error and the operation will not be performed.

Specifications of variables or sub-tables on which the operation is performed and in which the results are stored are arguments of the file interpreter's function that performs the operation. Usually, the specification of the sub-table on which the operation is performed, is the first argument, and the specification of the sub-table in which the results are stored, is the second argument of such functions.

Example:

Let's have a variable *v* of dimensions 3*2*2, a variable *r1* with dimensions 4*2*2, a variable *r2* with dimensions 3*2. Let's say we want to perform the operation *unop* on the sub-table *v[3]* and that we would like to use one of the variables *r1* or *r2* for storing results of the operation. We can to that different ways:

unop *{ r1[2] v[3] }*

is legal since the dimensions of the sub-tables *r2[2]*and *v[3]* match. The operation is performed in turns on elements *v[3,1,1], v[3,1,2], v[3,2,1]* and *v[3,2,2]* and the results of the operation are stored in elements *r1[3,1,1], r1[3,1,2], r1[3,2,1]* and *r1[3,2,2]*, accordingly.

unop *{ r2 v[3] }*

is also legal because the first specification specifies the whole variable table. Since the dimensions do not match, variable *r2* is first deleted and then created anew with dimensions 2*2 that match the dimensions of *v[3]*. The situation is similar if we perform

unop *{ r1 v[3] }*

We can not, however, perform the operation

unop *{ r1[1] v[3] }*

since *r1[1]* is a sub-table of the variable *r1* and it does not have the same dimensions than *v[3]*.

5.3.1.4.3 Operations on Pairs of Variable Elements (Binary Operations)

Most of binary operations on pairs of elements of variable sub-tables store their results in a third table of elements. For this result table the same results apply as in the chapter about unary operations which store their results.

There are two possibilities for the element tables on which the operation is performed. They can either be of the same dimensions or one table has a single element. In the second case, the operation combines the single element with all elements of the other sub-table.

At functions which perform binary operations, the specification of the result sub-table (or variable), and specifications of the sub-tables on which the operation is performed, are arguments of these functions. Usually the specifications of the sub-tables on which the operation is performed, are the first two arguments, and the specification of the result sub-table is the third argument of such functions. If the appropriate operations are not commutative, the order of the first two arguments matters, too.

Example:

Let us say there are variables *v1* of dimensions 3*2*2, *v2* of dimensions 5*2*2, *v3* of dimensions 4*2, *r1* of dimensions 2*2*2, and *r2* of dimension 2*3, and a binary operation named *binop*. Then we can perform the following operations:

binop *{ r1[2] v1[2] v2[4] }*

is valid since all sub-tables have the same dimensions.

binop *{ r1[ ] v1[2] v2[4] }*

is also valid since the first argument specifies the whole element table of the variable *r1*. This table is not of the same dimension as should the result table be, therefore variable *r1* is first deleted and then created with write dimensions (2*2) before the operation is performed.

binop *{ r2[ ] v3[2,1,2] v2[5] }*

is also legal because *v3[2,1]* specifies a single element. This element is in turns combined with all four elements of the sub-table *v2[5]*, the operation is performed on such pairs and results are stored in the appropriate elements of variable *r2*. This variable is deleted and then created anew with dimensions 2*2 before the operation begins.

binop *{ r1[ ] v3 v2[1] }*

is not legal since the dimensions of sub-tables *v3* and *v2[1]* do not match.

5.4 Basic Operations on User-defined Variables

For every type of the user-defined shell's variables there is a set of interpreter's and calculator's functions for manipulation of these variables. Some of these functions are of a general character and are similar for almost all types of variables.

Among the most important are the file interpreter's functions for creating variables and setting their values. For matrix variables these are functions **newmatrix**, **dimmatrix** and **setmatrix**.

The **newmatrix** function creates a matrix variable with specific dimensions. The only argument of this function is the name of the matrix variable followed by the list of dimensions in a square bracket. If this bracket is empty or if it is omitted, a variable with rank 0 is created (such variable can hold a single element). If a matrix variable with a given name exists when the **newmatrix** function is called, it is first deleted and then created.

The **dimmatrix** function is similar than **newmatrix**, except that if a matrix variables of a given name already exists and has right dimensions, it is leaved untouched.

The **setmatrix** function initialises individual elements of a matrix variable according to the value given in its argument block. The first argument is the specification of a matrix element, i.e. a variable name followed by a list of indices). The second argument specifies the values that should be assigned to that matrix element. This includes matrix dimension and all components, although only the dimension or individual components can be given.

If the first argument of the **setmatrix** function is a complete element specification with a list of indices, a given element must already exist. This can be assured by an appropriate execution of the **newmatrix** or **dimmatrix** function. If the element specification is given without an index list and a matrix variable with a given name does not exist yet, it is first created with rank 0. This is the only case when the **setmatrix** command creates a matrix variable. In other cases the variable must exist before.

Similar functions than **newmatrix** and **dimmatrix** exist for other types of variables. They only have different name (string "matrix" is replaced by the appropriate type name). Similar functions to **setmatrix** exist for other types of variables, with appropriately different names and formats of the second argument which specifies the element values.

Other important general functions are these for copying, moving and printing variables or their elements. For matrix variables, the function **movematrixvar** renames a matrix variable. The first name is the old and the second argument is a new name of the variable. Function **copymatrixvar** copies a whole matrix variable to another matrix variable. Again, names of the copied and the result variables are the two arguments of the functions. The **movematrixvar** function first deletes the second variable if it already exists, while the **copymatrixvar** function deletes it only if it is not of the right dimensions, and the same applies for individual elements.

The **copymatrix** function copies sub-tables of elements of matrix variables to another element sub-tables. It behaves like functions which perform unary operations on element sub-tables and store results to another sub-tables. Similarly the **movematrix** function moves sub-tables of elements to another sub-tables. It also behaves like functions which perform unary operations on element sub-tables and store results to another sub-tables. Particularity of this function is that the elements on which operation is performed are also affected (they are deleted).

The **printmatrix** prints a sub-table of elements of a matrix variable. It behaves like a simple unary operation. The **printmatrixvar** function does the same, except that it prints all matrices of the variable's element table and also some additional data about the variable.

For each variable type there are usually one or more expression evaluator's functions which return specific information about the variable's elements. For matrix variables such functions are **getmatrix**, which returns matrix element's components or dimensions, and **getmatrixdim**, which returns dimensions or rank of a matrix variable.

For variables that contain numerical information (i.e. scalars, vectors or matrices), there are functions which set the components of these information for sub-tables of variables' elements. For matrix variables such function is **setmatrixcomponents** or shortly **setmatcomp**. The first argument of this function is a specification of a sub-group of elements on which the operation is performed. The second argument is a mathematical expression the value of which is evauated and assigned to each component of the group of matrices separately. Two special calculator's functions, namely **varindex** and **varcomponent** can be used in functions similar to **setmatcomp**. The first one returns a specific index of the matrix element, and the second one returns the specific component number of the matrix component which is currently being assigned.

5.5 Variables with Local Definition Scope

When larger command files are used and a part of the code is organized in blocks with a clearly defined functionality (e.g. user defined interpreter functions that perform specific tasks), it is sometimes beneficial to define and use local variables. The reason for this is that such isolated blocks of code often require use of auxiliary variables to temporarily store data and operate on it. Such blocks can often offer some general functionality and can be reused several times, therefore the creator can not know in advance where his functions will be used. Use of global variables could be dangerous in such instances because these variables can have some other function outside the block, and their use for auxiliary data storage inside the block would corrupt their contents and disable their functionality in the global scope.

For the above reasons, a mechanism of declaring local variables, which can be used only in their local definition scope, is provided. In this scope the local variable names hide the global ones. All operations on a variable with a name that is made local to this scope, will be performed on the local instance of a variable and will leave the global variable with the same name untouched. When a local scope is exited, the local variables are destroyed and global names are uncovered, so that any operation on a variable with a given name will from then on be performed on the global instance of the variable with a given name.

5.5.1 Definition of Local Scopes

The scopes in which local definitions apply must be explicitly defined by the user, as well as variable names which are made local to a specific scope. This is done by nested calls to pairs of functions **deflocvar** and **undeflocvar**.

The **deflocvar** function defines the beginning of a potentially nested local scope (block) and the variable names that are made local to this scope, i.e. which hide names of variables that are defined and used outside that scope. Arguments of this function are all names (interpreted as string arguments) are made local to the scope.

The **undeflocvar** function exits the local scope defined by the corresponding call to **deflocvar** and releases any storage related to use of local variables in this block. After a call to **undeflocvar**, the variables outside the local scope, which have names that were made local, are again accessible.

Definition of any number of nested local scopes is possible. The user must take care that every local scope defined by **deflocvar** is exited by the corresponding call to **undeflocvar**. **undeflocvar** always cancels the inner-most local scope. Local names defined in a nested local scope hide local names in all outer scopes that include this scope. When a nested scope is exited by **undeflocvar**, each local name defined in that block uncovers the name in the innermost scope where that name is defined as a local one. It is considered that all possible variable names are defined in the global scope. The global scope does not need to be defined by a combined call to **deflocvar** and **undeflocvar**.

There are some potential traps related with definition of local scopes. Similarly as one must take care of properly closing brackets that define code blocks in branches and loops, starts and ends of local definition blocks must be defined consistently by nested calls to function pairs **deflocvar** and **undeflocvar**, which enclose these blocks. A potential danger lies in the possibility that the corresponding **undeflocvar** is omitted, for example because of exiting a code block where this function is called before the call, which can be done by exiting the current interpretation level using the **exit** command. If some inner local scope is not exited when it is supposed to be, a call to **undeflocvar**

might undefine a nested scope instead of the one it should. A mechanism for discovering such situations is provided through the functions **marklocvar** and **checklocvar**. When a pair of these functions is called inside of the same local definition level, the **checklocvar** reports an error if the level of the local definition scope is unexpectedly changed, e.g. because of omitting a call to **undeflocvar** that should close a nested local scope, or simply because of a coding error where the user forgets to call thefunction.

Warning:

Only shell variables can be made local by the mechanisms provided by **deflocvar** and **undeflocvar**. All calculator variables are global.

5.5.2 deflocvar *{ varname1 varname2 … }*

Defines a new local variable definition scope and makes shell interpreter or calculator variable names *var1, var2* , etc., local to this scope.

The scope defined by this function lasts until it is canceled by a call to the **undeflocvar** function. Within the scope, all operations on variables named by local names *varname1*, *varname2*, etc., affect local instances of these variables. Such operations will not affect variables with the same name that were potentially initialized outside of this scope. The same rules as apply for operations on global variables also apply for operations on local ones. The call to **undeflocvar**, which ends the local definition scope, destroys all local variables that were initialized within this scope (i.e. the variables named *varname1*, *varname2*, etc., which were initialized within the scope). It also restores eventual global variables with these names. Memory space that was allocated to store these variables is released.

An arbitrary number of nested local definition scopes can be defined. A rule applies that local names from inner scopes hide all variables carrying these names, which are defined in the enclosing outer scopes. If some particular variable name is not defined within the inner-most scope, operations on a variable with that name will affect the variable in the first (i.e. inner-most) enclosing scope where this name is defined as a local name, or a global variable, if this name is not defined as local in any of the enclosing scopes.

Example:

In the example below a scalar variable *s1* is defined in the global scope and assigned value 1.1. Then a local scope is defined by a call to **deflocvar** with variable names *s1* and *s3* made local to this scope. Then a local scalar variable *s1* is defined and assigned value 1.2, a global scalar value *s2* (since the name s2 is not local to this scope) is defined and assigned value 2.2, and a local scalar variable *s3* is defined and assigned value 3.2. Values of these variables are printed before the **undeflocvar** function is called, which ends the local scope. Then all values are printed again. The **undeflocvar** removes the memory space that was allocated for local storage of *s1* and *s3* and uncovers these global names. When the scalar variable *s1* is printed out, the value 1.1 is printed because the print refers to the global *s1* which was not affected by the **setscalar** function called

within the local scope. Value 2.2 will be printed for a scalar variable *s2*, because the appropriate call to the **setscalar** function within the local scope initialized a gloval *s2* as *s2* was not defined as a local name for this scope. The third scalar variable that attempts to be printed, *s3*, does not exist at all, because it was initialized only within the local scope and its name was defined as local to this scope by the **deflocvar** function.

It should be noted that within the local scope *s1* could be initialized as a variable of some other type (e.g. as a vector variable) since the local *s1* is not related to the global *s1* in any sense.

```
setscalar { s1 1.1}
deflocvar { s1 s3 }
     setscalar { s1 1.2 }  *{ refers to local s1 }
     setscalar {s2 2.2 }  *{ refers to global s2 }
     setscalar { s3 3.2 }  *{ refers to local s2 }
     printscalar{ s1 }
     printscalar { s2 }
     printscalar { s3 }
undeflocvar{  }
printscalar{ s1 }  *{ value 1.1 – assigned outside the local
scope }
printscalar { s2 }  *{ value 2.2 – assigned in the local scope
to global s2 }
printscalar { s3 }  *{ global s3 not defined }
```

5.5.3 undeflocvar *{ }*

Ends the definition of the local scope whose beginning was defined by the last call to the **deflocvar** that has not yet been canceled by **undeflocvar**. If any local variables were allocated within this scope, they are destroyed and the memory space allocated for their storage is released. Variable names defined in enclosing scopes, which were hidden by local names to the scope that is being canceled, are uncovered.

5.5.4 marklocvar *{ }*

Remembers the level of the innermost local variable definition scope, within which the function is called. This function must be always called in combination with **checklocvar**, which must be called within the same local definition scope within which the corresponding **marklocvar** was called.

5.5.5 checklocvar *{ }*

Checks if the level of the innermost local variable definition scope in which the function was called, corresponds to the level of the local scope within which the

corresponding (preeeding) call to **marclocvar** was made. If it does not, it reports an error. The function also writes a note about the number of the nested level of the local scope in which it was called, to the standard output and to the output file.

5.6 Special Expression Evaluator's Functions varindex *and* varcomponent

Calculator's functions **varindex** and **varcomponent** are designed to support those file interpreter's functions that iterate over components of the variable's element sub-tables and assign them values specified by mathematical expressions. Such functions are, for example, **setmatomp**, **setveccomp**, **setscalcomp**, etc.

5.6.1 varindex *[indexnum]*

Returns a specific index (defined by *indexnum*) of the variable's element which is currently in the assigment procedure of a function like **setmatcomp**. The function can be called only in the specific situation where a component of a variable's element is being assigned by a function like **setmatcomp**.

If the value of *indexnum* is zero, the function returns the rank (i.e. number of dimensions) of the element table of the variable in the assignment procedure.

Example:

Let us have a matrix variable *m* of dimensions 2*3 and let us execute the command

setmatcomp *{ m[2] varindex[0] + 3 * varindex[2] }*

When the components of the matrix sub-table are assigned, the term *varindex[0]* has the value 2 since the rank of variable *m* is 2. The value of the expression *varindex[2]* depends on which matrix element is currently in the evaluation procedure, since it returns the second index of that element. For *m[2,1]* its value is 1, for *m[2,2]* its value is 2 and for *m[2,3]* its value is 3. Therefore, all components of *m[2,1]* will be assigned the value 2 + 3 * 1 = 5, all components of *m[2,2]* will be assigned the value 2 + 3 * 2 = 8, and all components of *m[2,3]* will be assigned the value 2 + 3 * 3 = 11.

5.6.2 varcomponent *[compnum]*

Returns a specific component index (defined by *compnum*) of the variable element's component which is currently in the assigment procedure of a function like

setmatcomp. The function can be called only in the specific situation where a component of a variable's element is being assigned by a function like **setmatcomp**.

If the value of *compnum* is zero, the function returns the number of dimensions of a variable element (e.g. 2 for matrices, 1 for vectors, 0 for scalars).

Example:

Let us have a matrix variable *m* of dimensions 3*2 where all its elements are 2 by 2 matrices. and let us execute the command

setmatcomp *{ m[3] 10 * varindex[1] + varindex[2] + 0.1 * varcomponent[1] + 0.01 * varcomponent[2] }*

The command affects matrix elements *m[3,1]* and *m[3,2]* and sets their components to the following values:

m[3,1]: {{31.11, 31.12}, {31.21, 31.22}}
m[3,2]: {{32.11, 32.12}, {32.21, 32.22}}

5.7 Matrix Variables

Matrix variables hold matrix objects. These are two-dimensional arrays of decimal numbers. Both dimensions of an array are also a part of a matrix object.

Elements of matrix variables can be empty (uninitialised) which means that they contain no data.

5.7.1 File Interpreter's Functions for Manipulation of Matrix Variables

5.7.1.1 **newmatrix** *{ varname < [dim1, dim2, ...] > }*

Creates a new matrix variable named *varname* with dimensions *dim1*, *dim2*, etc. It does not initialise variable elements. The rank of the created matrix variable equals the number of specified dimensions (*dim1*, *dim2*, etc.). If no dimensions are specified, a variable with rank 0 is created. If a matrix variable named *varname* already exists, it is firs deleted (together with its elements).

Explanation:

The shell's variables can hold multi-dimensional tables of objects of a given type (matrices in this case). Variable dimensions refer to the dimensions of such table, and variable rank is the number of these dimensions.

5.7.1.2 **dimmatrix** *{ varname < [dim1, dim2, ...] > }*

The same a **newmatrix**, except that if a matrix variable named *varname* with proper dimensions already exists, it does not delete it.

5.7.1.3 **setmatrix** *{ elspec matspec }*

Sets a matrix element specified by *elspec* to the values specified by *matspec*. The specification of a matrix element *elspec* consists of a variable name and an optional index list in square brackets, e.g. *m1[2,3]*. The index list is not necessary if the rank of the matrix variable is 0.

elspec must address an existing matrix element, except if no indices are specified. In this case a matrix variable of rank 0 is created before the matrix element is set.

matspec specifies the contents, which are assigned to a matrix element. Matrix dimensions and components are normally specified in *matspec*. Alternatively, only dimensions, individual components, or groups of components can be specified in *matspec*. If only dimensions are specified, they must be followed by an empty curly bracket.

If matrix dimensions are specified in *matspec*, then if the matrix element specified by *elspec* already exists, but has wrong dimensions, it is deleted and created again with the right dimensions. If matrix element is not yet initialised, it is created anew. If it exists and has the right dimensions, it is not changed before the components are read.

Just components can be specified in *matspec* only if the matrix element specified by *elspec* already exists with the right dimensions.

All matrix components can be specified in *matspec*, but alternatively only one or only few components can be specified. In this case the components which are not specified remain the same. If these components have not been specified before, they will have indefinite values.

The standard format of *matspec* is the following:

dim1 dim2 { {1 1 : comp$_{1,1}$ } {1 2 : comp$_{1,2}$ } ... {2 1 : comp$_{2,1}$ } ... }

where *dim1* and *dim2* are the number of rows and columns, respectively, and *comp$_{1,1}$*, *comp$_{1,2}$*, etc., are matrix components. In the brackets where we specify components, component indices (i.e. the row number and the column number) are specified on the left side of a colon.

Alternatively, whole rows are given in one bracket:

dim1 dim2 { {1: comp$_{1,1}$ comp$_{1,2}$ ... } {2: comp$_{2,1}$ comp$_{2,2}$ ... } ... }

or components are just listed by turns, listing row by row:

dim1 dim2 { comp$_{1,1}$ comp$_{1,2}$ ... comp$_{2,1}$ comp$_{2,2}$ ... ... }

If row and column numbers are specified with the components, then the order in which components are listed does not matter. Similarly, if only row numbers are specified, the order in which rows are listed does not matter.

All numbers, which occur in the matspec, can be given by mathemathical expressions that are evaluated in the expression evaluator or by expression evaluator's variables.

Examples:

Let us say that we have a matrix variable *m* of dimensions 2*3 and that we want to assign a 2 by 2 matrix to its element *m[1,3]*. Let the assigned matrix be a diagonal matrix with diagonal components set to 3.2. In a standard form this is done like this:

setmatrix *{ m[1 3] 2 2 {{1 1 : 3.2 } { 1 2 : 0 } { 2 1 : 0 } { 2 2 : 3.2 } }*

Instead individual components, whole rows can be listed in curlz brackets:

setmatrix *{ m[1 3] 2 2 {{1 : 3.2 0 } { 2 : 0 3.2 } }*

All components can be just listed without specifying their row and column numbers:

setmatrix *{ m[1 3] 2 2 { 3.2 0 0 3.2 } }*

When component numbers are given, the order in which components are specified does not matter. The same matrix can also be set like this:

setmatrix *{ m[1 3] 2 2 {{ 1 2 : 0 } { 2 2 : 3.2 } { 2 1 : 0 } {1 1 : 3.2 }}*

or like this:

setmatrix *{ m[1 3] 2 2 {{ 2 : 0 3.2 } {1 : 3.2 0 } }*

If the matrix element is already initialised with proper dimensions, only components can be specified with the **setmatrix** command. This can be illustrated with an example where the dimensions and components are specified separately:

setmatrix *{ m[1 3] 2 2 { } }*

setmatrix *{ m[1 3] { 3.2 0 0 3.2 } }*

With the first **setmatrix** command the matrix element dimensions are initialised, i.e. a matrix with two rows and two columns is created. With the second **setmatrix** command the componenets of the matrix element are set. We can furtherly split the setting of components into two parts:

setmatrix *{ m[1 3] 2 2 { } }*

setmatrix *{ m[1 3] { 3.2 0 } }*

setmatrix *{ m[1 3] { { 2: 0 3.2 } } }*

With the second **setmatrix** command, only the first two matrix components were set. This means that the whole second row remained unset. We set this row with the third **setmatrix** command.

All numbers in the **setmatrix** commands can be replaced by mathematical expressions or by calculator's variables. We can simply put *${expr}* or *$varname* in place of numbers, where *expr* is a mathematical expression that can be evaluated in the expression evaluator, and *varname* is a name of the expression evaluator's variable. This and the fact that we can set only dimensions and individual components by the **setmatrix** command enables additional matrix variables manipulation .

Example:

Let us have a matrix variable of rank 0 named *m*. The following code creates an exact copy of *m* named *mc* regardless of the dimensions of *m*:

```
if { ( getmatrix[“m”,0,1]>0 && getmatrix[“m”,0,2]>0 )
[
    setmatrix { mc ${getmatrix[“m”,0,1]} ${getmatrix[“m”,0,2]} { } }
    = {countrow:1}
```

```
        while { (countrow<=getmatrix["m",0,1])
        [
                = {countcol:1}
                while { (countrow<=getmatrix["m",0,2])
                [
                        setmatrix { mc  { {
                                $countrow $xountcol :
                                ${getmatrix["m",countrow,countcol]}
                                } }
                        }
                        = {countcol:countcol+1}
                ] }
        = {countrow:countrow+1}
] else
[
        newmatrix { mc }
] }
```

5.7.1.4 initmatrix *{ subspec matspec }*

Sets matrix elements contained in the element sub-table specified by *subspec*, to the value specified by *matspec*. The form of *matspec* is the same as for function **setmatrix**.

5.7.1.5 copymatrixvar *{ varname1 varname2 }*

Copies the matrix variable named *varname1* to the variable named *varname2*. If the second matrix variable does not yet exist, it is created first. If it exists, it is overwritten.

5.7.1.6 movematrixvar *{ varname1 varname2 }*

Moves the matrix variable named *varname2* to the variable named *varname1*. If a matrix named *varname2* already exists, it is overwritten. After the operation the matrix variable named *varname1* no longer exists.

5.7.1.7 deletematrixvar *{ varname }*

Deletes the matrix variable named *varname*, together with its elements. After the operation the matrix variable named *varname* no longer exists.

5.7.1.8 printmatrixvar *{ varname }*

Prints the information (dimensions and components) about all matrix elements contained in the matrix variable named *varname*, to the standard output. General information about the matrix variable (e.g. rank and dimensions) is also printed.

5.7.1.9 fprintmatrixvar *{ varname }*

The same as the **printmatrix** function, except that it prints to the programme's output file.

5.7.1.10 dprintmatrixvar *{ varname }*

The same as the **printmatrix** function, except that it prints to both the standard output and programme's output file.

5.7.1.11 copymatrix *{ subspec1 subspec2 }*

Copies a sub-table of matrix elements specified by *subspec1* to a sub-table specified by *subspec2*. Elements are copied one by one from the first sub-table to the appropriate places of the second sub-table. Dimensions of the sub-tables must be the same, except if the second specification (*subspec2*) refers to the whole element table of a matrix variable (i.e. the specification does not include an index list). In this case, the second variable is created anew if necessary. The old variable is in this case first deleted if it exists.

If the second sub-table already contains matrix elements, they are overwritten by the elements of the first sub-table.

5.7.1.12 movematrix *{ subspec1 subspec2 }*

Moves a sub-table of matrix elements specified by *subspec1* to a sub-table specified by *subspec2*. Elements are moved one by one from the first sub-table to the appropriate places of the second sub-table. Dimensions of the sub-tables must be the same, except if the second specification (*subspec2*) refers to the whole element table of a matrix variable (i.e. the specification does not include an index list). In this case, the second variable is created anew if necessary. The old variable is in this case first deleted if it exists.

If the second sub-table already contains matrix elements, they are overwritten by the elements of the first sub-table. After the operation, the first sub-table contains only empty (uninitialised) elements.

5.7.1.13 deletematrix *{ subspec }*

Deletes a sub-table of matrix elements specified by *subspec*. Matrices in the sub-table are deleted one by one. After the operation is performed, the sub-table contains only empty (non-initialised) elements.

If *subspec* specifies a whole table of elements of a matrix variable, the matrix variable itself is not deleted. Only its elements become empty (uninitialised).

5.7.1.14 printmatrix *{ subspec }*

Prints information about elements of a sub-table of matrices specified by *subspec*.

5.7.1.15 fprintmatrix *{ subspec }*

Prints information about elements of a sub-table of matrices specified by *subspec* to the shell's output file (*outfile*).

5.7.1.16 dprintmatrix *{ subspec }*

Prints information about elements of a sub-table of matrices specified by *subspec* to both the standard output and the shell's output file (*outfile*).

5.7.1.17 **setmatrixcomponents** *{ subspec expr }*, shortly **setmatcomp**

Sets the components of all matrices contained in the element sub-table specified by *subspec*, to the value of the expression *expr*. The expression *expr* is evaluated by the expression evaluator for each component separately. The function iterates over all matrices of the sub-table specified by *subspec* and over all components of these matrices and assigns them values specified by *expr*.

Two special expression evaluator's functions, **varindex** and **varcomponent** are designed for use with functions like **setmatcomp**. When the expressions *expr* is being evaluated for a specific component of a specific matrix element, the **varindex** function returns a specific index of the matrix element that is affected. The **varcomponent** function returns a specific component index (row or matrix number) of the component, which is currently in the evaluation procedure.

Setting matrix components by the **setmatcomp** function is much quicker than doing it by programming loops which iterate over matrices of a variable's sub-table and over their components. This is because we avoid iterative calls of interpreter's functions. Instead, only one interpreter's function is called to do the job, the iteration loops are implemented in advance in this function and only the expression *exp* is evaluated again and again. The evaluation of this expression is very quick because it is parsed and interpreted outside the iteration loops. An already parsed and interpreted expression is evaluated within the iteration loops.

Example:

Let us have a matrix variable *m* which contains 5*2*3 matrices of dimension 5*2. Let us assign values to all components of the sub-table *m[4]* in such a way that component values will equal 100 times the first index of the sub-table element plus 10 times the second index of the sub-table element plus 0.1 times the row number plus 0.01 times the column number of components. This is done by the command

setmatcomp *{ m[4] 100 * varindex[2] + 10 * varindex[3] + 0.1 * varcomponent[1] + 0.01 * varcomponent[2] }*

5.7.1.18 **setmatcompcond** *{ subspec (cond) expr }*

Does the same as **setmatcomp**, except that only those components of the matrix elements specified by *subspec* are set for which the condition *cond* is satisfied. *cond* is a mathematical expression that can also include calls to expression evaluator's functions **varindex** and **varcomponent**.

5.7.1.19 **matrixsum** *{ subspec1 subspec2 subspecres }*

Adds together matrices from element tables specified by *subspec1* and *subspec2* and stores the results into matrices on the element table specified by *subspecres*.

The aim of this function is to demonstrate how binary operations on sub-tables of matrix variables work.

5.7.1.20 **matrixdif** *{ subspec1 subspec2 subspecres }*

Subtracts matrices from element tables specified by *subspec1* and *subspec2* and stores the results into matrices on the element table specified by *subspecres*.

5.7.1.21 **matrixop** *{ spec = < operator > spec < operator spec > },* **matop**

Performs a matrix operation specified in the argument block. The operation specification consists of object specifications (matrix, vector and scalar objects) and operators.

This function performs some basic matrix and vector operations like multiplication and addition, evaluation of norms and determinants, matrix inversion, solution of systems of equations, etc.

One operation can be performed at a time. The argument block of the command is an expression which consists of object specifications (operands) and operators. Object specifications usually start with a two letter prefix which specifies the type of a specific object: s_ for scalars, v_ for vectors and m_ for matrices. This is followed by the specification of an element of a specific type, which is given by variable name and index list in square brackets. No spaces are allowed between the prefix which specifies object type and element specification. Element specification refers to a specific element of a user-defined variable. The operations can be performed only on individual elements (objects) within one call to the **matop** function, not on tables of elements.

Sometimes the object specification is a mathematical expression which can be evaluated in the expression evaluator. In this case, the expression must be in curly brackets.

Usually the operations which are performed produce some results which is stored in some object. In this case, in the argument block we gave first the specification of the object (variable element) into which the result is stored, followed by the assignment operator =.

The following operations can be performed by the **matrixop** function:

- ***matop{m_m0 = m_m1 + m_m2}*** : matrix *m0* becomes the sum of matrices *m1* snd *m2*.
- ***matop{m_m0 = m_m1 - m_m2}*** :
- ***matop{v_v0 = v_v1 + v_v2}*** : vector *v0* becomes the sum of vectors *v1* and *v2*.
- ***matop{v_v0 = v_v1 - v_v2}*** : vector *v0* becomes the difference between matrices *v1* and *v2*.
- ***matop{m_m0 = m_m1 * m_m2}*** : matrix *m0* becomes the product of matrices *m1* snd *m2*.
- ***matop{v_v0 = m_m1 * v_v2}*** : vector *v0* becomes the product of matrix *m1* and vector *v2*.
- ***matop{s_s0 = v_v1 * v_v2}*** : Scalar variable *s0* becomes the scalar product of vector variables *v1* and *v2*.

- ***matop{m_m0 = m_m1 * {expr}}*** : Matrix *m0* becomes matrix *m1* multiplied by the value of the expression *expr*.
- ***matop{v_v0 = v_v1 * {expr}}*** : Vector *v0* becomes vector *v1* multiplied by the value of the expression *expr*.
- ***matop{v_x = m_A solve v_b}*** : Vector *x* becomes the solution of the equation A x = b
- ***matop{m_m0 = transpose m_m1}*** : Matrix *m0* becomes the transpose of the matrix *m1*.
- ***matop{m_m0 = transpose v_v1}*** : Matrix *m0* becomes th etranspose of the vector *v1*.
- ***matop{m_m0 = invert m_m1}*** : matrix *m0* becomes the inverse of th ematrix *m1*.
- ***matop{s_s0 = norm m_m1}*** : Scalar *s0* becomes the euclidian norm of the matrix *m1* (the square root of the sum of squares of components).
- ***matop{s_s0 = norm v_v1}*** : Scalar *s0* becomes the euclidian norm of the vector *v1*.
- ***matop{m_m0 = normalize m_m1}*** : Matrix *m0* becomes the normalised matrix *m1* (its component are divided by its norm).
- ***matop{v_v0 = normalize v_v1}*** : Vector *v0* becomes the normalized vector *v1*.
- ***matop{m_m0 = identitymatrix {expr}}*** Matrix *m0* becomes the identity square matrix of dimension that equals the value of the expression *expr*.
- ***matop{m_m0 = zeromatrix {expr1} {expr2}}*** : .Matrix *m0* becomes the zero matrix of dimensions that equal the values of the expressions *expr1* and *expr2*.
- ***matop{m_m0 = randommatrix {expr1} {expr2}}*** : Matrix *m0* becomes a matrix of dimensions that equal the values of the expressions *expr* and *expr1* with random components between 0 and 1. If only one expression is given, *mo* becomes a square matrix of the appropriate dimension.
- ***matop{v_v0 = zerovector {expr}}*** : Vector *v0* becomes a yero vector of the dimension that equals the value of the expression *expr*.
- ***matop{v_v0 = randomvector {expr}}*** : Vector *v0* becomes a zero vector of the dimension that equals the value of the expression *expr*.

Examples
matop { m_m1[2,3] = transpose m_a0[1] }

5.7.2 Expression Evaluator's Functions for Manipulating Matrix Variables

5.7.2.1 **getmatrix** *[varname rownum colnum < elind1, elind2, ... >]*

Returns a specific matrix component. *varname* is the name of a matrix variable. *elind1*, *elind2*, etc., are the indices, which specify the matrix element on the variable's element table. *rownum* is the row number and *colnum* is the column number of the component, which is returned.

If the variable named *varname* is of rank 0, then no indices *elind1*, *elind2*, etc. need to be specified.

If *rownim* is 0, the function returns number of rows of the matrix if *colnum* is 1, or number of columns of the matrix if *colnum* is 2.

5.7.2.2 **getmatrixdim** *[varname dimnum]*

Returns the *dimnum*-th dimension of the matrix variable named *varname*. If *dimnum* is 0, it returns the rank of the variable, and if *dimnum* is –1, it returns the total number of elements contained in the variable (which equals product of all dimensions if rank is greater than zero), no matter if elements are initialised or not.

Warning:

If *dimnum* is –1, **getmatrixdim** does not report an error if matrix variable named *varname* does not exists. It returns 0 in this case, therefore this function is suitable for checking variable existence.

5.8 Vector Variables

Vector objects are one-dimensional arrays of decimal numbers. The dimension of an array is also a part of a vector object.

Elements of vector variables can be empty (uninitialised) which means that they contain no data. The dimension and components of such elements are not defined.

5.8.1 File Interpreter's Functions for Manipulating Vector Variables

5.8.1.1 **newvector** *{ varname < [dim1, dim2, ...] > }*

Does the same as **newmatrix**, but for vector variables.

5.8.1.2 **dimvector** *{ varname < [dim1, dim2, ...] > }*

Does the same as**dimmatrix**, but for vector variables.

5.8.1.3 **setvector** *{ elspec vecspec }*

This function is similar to the corresponding matrix function **setmatrix**. It sets a vector element specified by *elspec* to the values specified by *vecspec*. The specification of a vector element *elspec* consists of a variable name and an optional index list in square brackets, e.g. *v1[2,3]*. The index list is not necessary if the rank of the vector variable is 0.

elspec must address an existing vector element, except if no indices are specified. In this case a vector variable of rank 0 is created before the vector element is set.

vecspec specifies the contents which are assigned to the vector element. Vector dimension and components are normally specified in *vecspec*. Alternatively, only the dimension, individual components, or groups of components can be specified in *vecspec*. If only dimension is specified, it must be followed by an empty curly bracket.

If vector dimensions are specified in *vecspec*, then if the vector element specified by *elspec* already exists, but have a wrong dimension, it is deleted and created again with the right dimension. If vector element is not yet initialised, it is created anew. If it exists and has the right dimension, it is not changed before the components are read.

Just components can be specified in *vecspec* only if the vector element specified by *elspec* already exists with the right dimension.

All vector components can be specified in *vecspec*, but alternatively only one or only few components can be specified. In this case the unspecified components remain the same. If the components have not been specified before, their values will be indefinite.

The standard format of *vecspec* is the following:

dim { {1 : comp$_1$ } { 2 : comp$_2$ } { 3 : comp$_3$ } ... }

where *dim* is vector dimension (the number of components), and *comp$_1$*, *comp$_2$*, etc., are vector components. In the brackets where we specify components, the component numbers are specified on the left side of a colon.

Alternatively, the components are just listed by turns:

dim { comp$_1$ comp$_2$ comp$_3$ ... }

If the component numbers are specified with the components, then the order in which components are listed does not matter.

All numbers that occur in the *vecspec* can be given by mathemathical expressions that are evaluated in the expression evaluator or by expression evaluator's variables.

Examples:

Let us say that we have a vector variable *v* of dimensions 2*3 and that we want to assign a 4 dimensional vector to its element *v[1,3]*. Let the assigned vector components have values 1.1, 2.2, 3.3 and 4.4. In a standard form this is done like this:

setvector *{ v[1 3] 4 { {1 : 1.1 } { 2 : 2.2 } { 3 : 3.3 } { 4 : 4.4 } }*

All components can be just listed without specifying their component numbers:

setvector *{ v[1 3] 4 { 1.1 2.2 3.3 4.4 } }*

When component numbers are given, the order in which components are specified does not matter. The same vector can also be set like this:

setvector *{ v[1 3] 4 { { 3 : 3.3 } {1 : 1.1 } { 4 : 4.4 } { 2 : 2.2 } }*

If a vector element is already initialised with proper dimensions, only components can be specified by the **setvector** command. This can be illustrated with an example where the dimensions and components are specified separately:

setvector *{ v[1 3] 4 { } }*

setvector *{ v[1 3] { 1.1 2.2 3.3 4.4 } }*

With the first **setvector** command the vector element dimension is initialised, i.e. a vector with four components is created. With the second **setvector** command the componenets of the vector element are set. We can furtherly split the setting of components into two parts:

setvector *{ v[1 3] 4 { } }*
setvector *{ v[1 3] { 1.1 2.2 } }*
setvector *{ v[1 3] { { 3 : 3.3 } { 4 : 4.4 } } }*

With the second **setvector** command, only the first two vector components were set. The rest components were set by the third **setvector** command.

All numbers in the **setvector** commands can be replaced by mathematical expressions or by calculator's variables. We can simply put *${expr}* or *$varname* in place of numbers, where *expr* is a mathematical expression that can be evaluated in the expression evaluator, and *varname* is a name of an expression evaluator's variable. This and the fact that we can set only dimensions and individual components by the **setvector** command enables additional vector variables manipulation .

Example:

The following code creates a zero rank vector variable *tab* which holds a table of valuesof the exponential function on the interval [0,1] with step 1:

```
= {currentx: 0}
= {lastx: 10}
= {step: 1}
= {dim: 11}
setvector {tab  dim { } }
= {compnum : 1}
while { (currentx<=lastx)
[
        setvector {tab { { $compnum : exp[currentx] } }}
        = {currentx:currentx+step}
        = {compnum:compnum+1}
] }
```

5.8.1.4 initvector *{ subspec vecspec }*

Sets vector elements contained in the element sub-table specified by *subspec*, to the value specified by *vecspec*. The form of *vecspec* is the same as for function **setvector**.

5.8.1.5 copyvectorvar *{ varname1 varname2 }*

Does the same as **copymatrixvar**, but for vector variables.

5.8.1.6 movevectorvar *{ varname1 varname2 }*

Does the same as **movematrixvar**, but for vector variables.

5.8.1.7 deletevectorvar *{ varname }*

Does the same as **deletematrixvar**, but for vector variables.

5.8.1.8 printvectorvar *{ varname }*

Does the same as **printmatrixvar**, but for vector variables.

5.8.1.9 fprintvectorvar *{ varname }*

Does the same as **frintmatrixvar**, but for vector variables.

5.8.1.10 dprintvectorvar *{ varname }*

Does the same as **drintmatrixvar**, but for vector variables.

5.8.1.11 copyvector *{ subspec1 subspec2 }*

Does the same as **copymatrix**, but for vector variables.

5.8.1.12 movevector *{ subspec1 subspec2 }*

Does the same as **movematrix**, but for vector variables.

5.8.1.13 deletevector *{ subspec }*

Does the same as **deletematrix**, but for vector variables.

5.8.1.14 printvector *{ subspec }*

Does the same as **printmatrix**, but for vector variables.

5.8.1.15 fprintvector *{ subspec }*

Does the same as **fprintmatrix**, but for vector variables.

5.8.1.16 dprintvector *{ subspec }*

Does the same as **dprintmatrix**, but for vector variables.

5.8.1.17 setvectorcomponents *{ subspec expr }*, shortly **setveccomp**

This function is similar to its matrix equivalent **setmatcomp**. It sets the components of all vectors contained in the element sub-table specified by *subspec*, to the value of the expression *expr*. The expression *expr* is evaluated by the expression evaluator for each component separately. The function iterates over all vectors of the sub-table specified by *subspec* and over all components of these vectors and assigns them values specified by *expr*.

Two special expression evaluator's functions, **varindex** and **varcomponent** are designed for use with functions like **setveccomp**. When the expressions *expr* is being evaluated for a specific component of a specific vector element, the **varindex** function returns a specific index of the vector element that is affected. The **varcomponent** function returns the number of the component which is currently in the evaluation

procedure. It must be called with argument 1, although vector objects have only one dimension anyway (because of compatibility reasons).

Setting vector components by the **setveccomp** function is much quicker than doing it by programming loops which iterate over vectors of a variable's sub-table and over their components. This is because we avoid iterative calls to interpreter's functions. Instead, only one interpreter's function is called to do the job, the iteration loops are implemented in advance in this function and only the expression *exp* is evaluated again and again. The evaluation of this expression is very quick because it is parsed and interpreted outside the iteration loops.

Example:

Let us have a vector variable *v* which contains 6*3*4 vectors of dimension 10. Let us assign values to all components of the sub-table *v[3]* in such a way that component values will equal 10 times the first index of the sub-table element plus the second index of the subtable element plus 0.1 times the component number. This is done by the command

setveccomp *{ v[3] 10 * varindex[2] + varindex[3] + 0.1 * varcomponent[1] }*

5.8.1.18 setveccompcond *{ subspec (cond) expr }*

Does the same as **setveccomp**, except that only those components of the vector elements specified by *subspec* are set for which the condition *cond* is satisfied. *cond* is a mathematical expression that can also include calls to the expression evaluator's functions **varindex** and **varcomponent**.

5.8.1.19 vectorsum *{ subspec1 subspec2 subspecres }*

Does the same as **matrixsum**, but for vector variables. The aim of this function is mostly to demonstrate how binary operations on variable sub-tables work.

5.8.1.20 vectordif *{ subspec1 subspec2 subspecres }*

Does the same as **matrixdif**, but for vector variables.

5.8.2 Expression Evaluator's Functions for Manipulating Vector Variables

5.8.2.1 getvector *[varname compnum < elind1, elind2, ... >]*

Returns a specific vector component. *varname* is the name of a vector variable. *elind1*, *elind2*, etc., are the indices which specify the vector element on the variable's element table. *compnum* is the number of the component which is returned.

If the variable named *varname* is of rank 0, then no indices *elind1*, *elind2*, etc. do not need to be specified.

If *compnum* is 0, then the function returns the dimension (number of components) of the vector element.

5.8.2.2 **getvectordim** *[varname dimnum]*

The same as **getmatrixdim**, but for vector variables.

5.9 Scalar Variables

Scalar objects are simply decimal numbers.

Elements of scalar variables can be empty (uninitialised) which means that they contain no data, the same as elements of vector or matrix variables..

5.9.1 File Interpreter's Functions for Manipulating Scalar Variables

5.9.1.1 **newscalar** *{ varname < [dim1, dim2, ...] > }*

Does the same as **newmatrix**, but for scalar variables.

5.9.1.2 **dimscalar** *{ varname < [dim1, dim2, ...] > }*

Does the same as**dimmatrix**, but for scalar variables.

5.9.1.3 **setscalar** *{ elspec scalspec }*

Sets a scalar element specified by *elspec* to the value specified by *scalspec*. The specification of a scalar element *elspec* consists of a variable name and an optional index list in square brackets, e.g. *s1[2,3]*. The index list is not necessary if the rank of the scalar variable is 0.

elspec must address an existing scalar element, except if no indices are specified. In this case a scalar variable of rank 0 is created before the scalar element is set.

scalspec specifies the value which is assigned to the scalar object. It can be given as a number, as a mathematical expression that can be evaluated by the expression evaluator, or as an expression evaluator's variable. Expressions in place of numbers must be of the form *${expr}* where *expr* is a mathematical expression, and calculator's variables in place of numbers must be of the form *$varname*, where *varname* is the calculator's variable name.

5.9.1.4 **initscalar** *{ subspec scalspec }*

Sets scalar elements contained in the element sub-table specified by *subspec*, to the value specified by *scalspec*. The form of *scalspec* is the same as for function **setscalar**.

5.9.1.5 **copyscalarvar** *{ varname1 varname2 }*

Does the same as **copymatrixvar**, but for scalar variables.

5.9.1.6 **movescalarvar** *{ varname1 varname2 }*

Does the same as **movematrixvar**, but for scalar variables.

5.9.1.7 **deletescalarvar** *{ varname }*

Does the same as **deletematrixvar**, but for scalar variables.

5.9.1.8 **printscalarvar** *{ varname }*

Does the same as **printmatrixvar**, but for scalar variables.

5.9.1.9 **fprintscalarvar** *{ varname }*

Does the same as **printmatrixvar**, but for scalar variables.

5.9.1.10 **dprintscalarvar** *{ varname }*

Does the same as **dprintmatrixvar**, but for scalar variables.

5.9.1.11 **copyscalar** *{ subspec1 subspec2 }*

Does the same as **copymatrix**, but for scalar variables.

5.9.1.12 **movescalar** *{ subspec1 subspec2 }*

Does the same as **movematrix**, but for scalar variables.

5.9.1.13 **deletescalar** *{ subspec }*

Does the same as **deletematrix**, but for scalar variables.

5.9.1.14 **printscalar** *{ subspec }*

Does the same as **printmatrix**, but for scalar variables.

5.9.1.15 **fprintscalar** *{ subspec }*

Does the same as **fprintmatrix**, but for scalar variables.

5.9.1.16 **dprintscalar** *{ subspec }*

Does the same as **dprintmatrix**, but for scalar variables.

5.9.1.17 setscalarcomponents *{ subspec expr }*, shortly **setscalcomp**

This function is similar to its matrix equivalent **setmatcomp**. It sets the values of all scalars contained in the element sub-table specified by *subspec*, to the value of the expression *expr*. The expression *expr* is evaluated by the expression evaluator for each scalar separately.

A special expression evaluator's function **varindex** is designed for use with functions like **setscalcomp**. When the expressions *expr* is being evaluated for a specific scalar element, the **varindex** function returns a specific index of the scalar element that is affected.

It makes sense to use this function only with scalar variables with rank greater than zero, especially if they contain a large number of scalars. It is differently with matrix and vector variables where element themselves hold more components.

The **varcomponent** function can not be used with **setscalcomp** since scalars are simple objects without components.

Warning:

The **setscalcomp** function assigns values only to those scalars which are initialised. It has no effect if it is performed on a sub-table the elements of which are not initialised. Note that functions which create scalar variables like **newscalar** or **dimscalar** usually do not initialise the scalars on variable's element table.

Example:

Let us have a scalar variable *s* which contains 2*3*4 scalars. Let us assign values to all components of the sub-table *s[2]* in such a way that scalar values will equal 10 times the first index of the sub-table element plus the second index of the sub-table element. This is done by the command

setscalcomp *{ s[3] 10 * varindex[2] + varindex[3] }*

5.9.1.18 setscalcompcond *{ subspec (cond) expr }*

Does the same as **setscalcomp**, except that only those scalar elements specified by *subspec* are set for which the condition *cond* is satisfied. *cond* is a mathematical expression that can also include calls to expression evaluator's function **varindex**.

5.9.1.19 scalarsum *{ subspec1 subspec2 subspecres }*

Does the same as **matrixsum**, but for scalar variables. The aim of this function is mostly to demonstrate the use of binary operations performed on sub-table of variable's elements.

5.9.1.20 scalardif *{ subspec1 subspec2 subspecres }*

Does the same as **matrixdif**, but for scalar variables.

5.9.2 Expression Evaluator's Functions for Manipulating Scalar Variables

5.9.2.1 **getscalar** *[varname < elind1, elind2, ... >]*

Returns the value of a specific scalar object. *varname* is the name of a scalar variable. *elind1*, *elind2*, etc., are the indices which specify the scalar element on the variable's element table.

If the variable named *varname* is of rank 0, then no indices *elind1*, *elind2*, etc. do not need to be specified.

5.9.2.2 **getscalardim** *[varname dimnum]*

The same as **getmatrixdim**, but for scalar variables.

5.10 Field Variables

5.10.1 File Interpreter's Functions for Manipulating Field Variables

5.10.1.1 **newfield** *{ varname < [dim1, dim2, ...] > }*

Does the same as **newmatrix**, but for field variables.

5.10.1.2 **dimfield** *{ varname < [dim1, dim2, ...] > }*

Does the same as **dimmatrix**, but for field variables.

5.10.1.3 **setfield** *{ elspec fieldspec }*

Does the same as **setmatrix**, but for field variables.

5.10.1.4 **initfield** *{subspec fieldspec }*

Does the same as **initmatrix**, but for field variables.

5.10.1.5 **copyfieldvar** *{varname1 varname2}*

Does the same as **copymatrixvar**, but for field variables.

5.10.1.6 **movefieldvar** *{ varname1 varname2 }*

Does the same as **movematrixvar**, but for field variables.

5.10.1.7 **deletefieldvar** *{ varname }*

Does the same as **deletematrixvar**, but for field variables.

5.10.1.8 **printfieldvar** *{ varname }*

Does the same as **printmatrixvar**, but for field variables.

5.10.1.9 **fprintfieldvar** *{ varname }*

Does the same as **fprintmatrixvar**, but for field variables.

5.10.1.10 **copyfield** *{ subspec1 subspec2 }*

Does the same as **copymatrix**, but for field variables.

5.10.1.11 **movefield** *{ subspec1 subspec2 }*

Does the same as **movematrix**, but for field variables.

5.10.1.12 **deletefield** *{ subspec }*

Does the same as **deletematrix**, but for field variables.

5.10.1.13 **printfield** *{ subspec }*

Does the same as printmatrix, but for field variables.

5.10.1.14 **fprintfield** *{ subspec }*

Does the same as **fprintmatrix**, but for field variables.

5.10.1.15 **setfieldcomponents** *{ subspec expr }*

Does the same as **setmatrixcomponents**, but for field variables.

5.10.1.16 **setfldcompcond** *{ subspec (cond) expr }*

Does the same as **setmatcompcond**, but for field variables.

5.10.2 Expression Evaluator's Functions for manipulating Field Variables

5.10.2.1 **getfield** *{varname <elind1, elind2,...> rownum colnum }*

Does the same as **getmatrix**, but for field variables.

5.10.2.2 **getfielddim** *{varname dimnum }*

Does the same as **getmatrixdim**, but for field variables.

5.11 Counter Variables

Counter objects are integer numbers.

Elements of counter variables can be empty (uninitialised) which means that they contain no data, the same as elements of vector or matrix variables.

5.11.1 File Interpreter's Functions for Manipulating Counter Variables

5.11.1.1 **newcounter** *{ varname < [dim1, dim2, ...] > }*

Does the same as **newmatrix**, but for counter variables.

5.11.1.2 **dimcounter** *{ varname < [dim1, dim2, ...] > }*

Does the same as **dimmatrix**, but for counter variables.

5.11.1.3 **setcounter** *{ elspec countspec }*

Sets a counter element specified by *elspec* to the value specified by *countspec*. The specification of a counter element *elspec* consists of a variable name and an optional index list in square brackets, e.g. *cc[2,3]*. The index list is not necessary if the rank of the counter variable is 0.

elspec must address an existing counter element, except if no indices are specified. In this case a counter variable of rank 0 is created before the counter element is set.

countspec specifies the value which is assigned to the counter object. It can be given as a number, as a mathematical expression that can be evaluated by the expression evaluator, or as an expression evaluator's variable. Expressions in place of numbers must be of the form *${expr}* where *expr* is a mathematical expression, and calculator's variables in place of numbers must be of the form *$varname*, where *varname* is the calculator's variable name.

Warning:

countspec can also result in a decimal. In this case, it is first rounded and then assigned to the appropriate element of a counter variable.

5.11.1.4 **initcounter** *{ subspec countspec }*

Sets counter elements contained in the element sub-table specified by *subspec*, to the value specified by *countspec*. The form of *contspec* is the same as for function **setcounter**.

5.11.1.5 **copycountervar** *{ varname1 varname2 }*

Does the same as **copymatrixvar**, but for counter variables.

5.11.1.6 **movecountervar** *{ varname1 varname2 }*

Does the same as **movematrixvar**, but for counter variables.

5.11.1.7 **deletecountervar** *{ varname }*

Does the same as **deletematrixvar**, but for counter variables.

5.11.1.8 **printcountervar** *{ varname }*

Does the same as **printmatrixvar**, but for counter variables.

5.11.1.9 **fprintcountervar** *{ varname }*

Does the same as **fprintmatrixvar**, but for counter variables.

5.11.1.10 **dprintcountervar** *{ varname }*

Does the same as **dprintmatrixvar**, but for counter variables.

5.11.1.11 **copycounter** *{ subspec1 subspec2 }*

Does the same as **copymatrix**, but for counter variables.

5.11.1.12 **movecounter** *{ subspec1 subspec2 }*

Does the same as **movematrix**, but for counter variables.

5.11.1.13 **deletecounter** *{ subspec }*

Does the same as **deletematrix**, but for counter variables.

5.11.1.14 **printcounter** *{ subspec }*

Does the same as **printmatrix**, but for counter variables.

5.11.1.15 **fprintcounter** *{ subspec }*

Does the same as **fprintmatrix**, but for counter variables.

5.11.1.16 **dprintcounter** *{ subspec }*

Does the same as **dprintmatrix**, but for counter variables.

5.11.1.17 **setcountercomponents** *{ subspec expr },* shortly **setcountcomp**

This function is similar to its matrix equivalent **setscalcomp**. It sets the values of all counters contained in the element sub-table specified by *subspec*, to the value of the expression *expr*. The expression *expr* is evaluated by the expression evaluator for each counter separately.

A special expression evaluator's function **varindex** is designed for use with functions like **setcountcomp**. When the expressions *expr* is being evaluated for a specific counter element, the **varindex** function returns a specific index of the counter element that is affected.

It makes sense to use this function only with counter variables with rank greater than zero, especially if they contain a large number of counters. It is differently with matrix and vector variables where element themselves hold more components.

The **varcomponent** function can not be used with **setcountcomp** since counters are simple objects without components.

Warning:

The **setcountcomp** function assigns values only to those counters which are initialised. It has no effect if it is performed on a sub-table the elements of which are not initialised. Note that functions which create counter variables like **newcounter** or **dimcounter** usually do not initialise the counters on variable's element table.

Example:

Let us have a counter variable *c* which contains 2*3*4 counters. Let us assign values to all components of the sub-table *c[2]* in such a way that counter values will equal 10 times the first index of the sub-table element plus the second index of the sub-table element. This is done by the command

setcountcomp *{ c[3] 10 * varindex[2] + varindex[3] }*

5.11.1.18 **setcountcompcond** *{ subspec (cond) expr }*

Does the same as **setcountcomp**, except that only those counter elements specified by *subspec* are set for which the condition *cond* is satisfied. *cond* is a mathematical expression that can also include calls to expression evaluator's function **varindex**.

5.11.1.19 **countersum** *{ subspec1 subspec2 subspecres }*

Does the same as **matrixsum**, but for counter variables. The aim of this function is mostly to demonstrate the use of binary operations performed on sub-table of variable's elements.

5.11.2 Expression Evaluator's Functions for Manipulating Counter Variables

5.11.2.1 **getcounter** *[varname < elind1, elind2, ... >]*

Returns the value of a specific counter object. *varname* is the name of a counter variable. *elind1*, *elind2*, etc., are the indices which specify the counter element on the variable's element table.

If the variable named *varname* is of rank 0, then no indices *elind1*, *elind2*, etc. do not need to be specified.

5.11.2.2 **getcounterdim** *[varname dimnum]*

The same as **getmatrixdim**, but for counter variables.

5.12 Options

Options are object that can have two values:0 or 1. We also say that they are unset (value 0) or set (value 1). Usually they are used to control behaviour of various functions of the shell.

No special type of variables is prepared for carrying options. Rather than that, options are hold by counter variables. An option hold in a counter variable is considered to be 0 if the appropriate counter is 0, and 1 if the appropriate counter is different than zero.

If a counter that represents a specific option does not exist or is not initialised, the appropriate option is considered to be 0.

Actually, the file interpreter's and expression evaluator's functions for manipulating counter variables could be used for handling options. Nevertheless, there are a few functions that are created especially for setting options and checking their status. The main reason for that is maintaining greater clarity of the code in the command file.

5.12.1 File Interpreter's Functions for Handling Options

5.12.1.1 **setoption** *{ optspec }*

Sets the option specified by *optspec*. *optspec* is the specification of the counter element that represents the option. The appropriate counter element is set to 1.

5.12.1.2 clearoption *{ optspec }*

Clears the option specified by *optspec*. *optspec* is the specification of the counter element that represents the option. The appropriate counter element is set to 0 if it exists, otherwise nothing happens.

5.12.2 Expression Evaluator's Functions for Handling Options

5.12.2.1 getoption *[varname < elind1, elind2, ... >]*

Returns 1 if the specified option is set and 0 if it is not set. *varname* is the name of a counter variable. *elind1, elind2,* etc., are the indices which specify the counter element that holds the option, on the variable's element table. If the counter variable named *varname* is of rank 0, then no indices *elind1*, *elind2*, etc. do not need to be specified.

A specific option is considered to be set, if the appropriate counter element exists, is initialised and is different than zero. Otherwise, this option is considered to be unset.

5.13 Options

5.13.1.1 setoption *{ optspec }*

Sets the option specified by *optspec*. *optspec* is the specification of the counter element that represents the option. The appropriate counter element is set to 1.

5.13.1.2 clearoption *{ optspec }*

Clears the option specified by *optspec*. *optspec* is the specification of the counter element that represents the option. The appropriate counter element is set to 0 if it exists, otherwise nothing happens.

5.14 String Variables

String objects are character sequences of arbitrary length. They can not contain null characters, i.e. characters with ASCII code 0.

Elements of string variables can be empty (uninitialised) which means that they contain no data, the same as elements of vector or matrix variables.

setstrcomp *{ s[3] 10 * varindex[2] + varindex[3] }*

5.14.1 File Interpreter Functions for Manipulating String Variables

5.14.1.1 newstring *{ varname < [dim1, dim2, ...] > }*

Does the same as **newmatrix**, but for string variables.

5.14.1.2 dimstring *{ varname < [dim1, dim2, ...] > }*

Does the same as**dimmatrix**, but for string variables.

5.14.1.3 setstring *{ elspec strspec }*

Sets a string element specified by *elspec* to the value specified by *strspec*. The specification of a string element *elspec* consists of a variable name and an optional index list in square brackets, e.g. *s1[2,3]*. The index list is not necessary if the rank of the string variable is 0.

elspec must address an existing string element, except if no indices are specified. In this case a string variable of rank 0 is created before the string element is set.

strspec specifies the value (string), which is assigned to the string object. It can be specified directly or as reference to an existent string object. When specified directly, it can contain special character sequences and must be stated inside double quotes if it contains blank characters. When specified as reference to an existing string object, a copy of that object is created and assigned to the element specified by *elspec*. In this case, *strspec* consists of the hash character (#) followed by specification of the referenced string element.

Examples:

setstring *{ s[3 2] xyz }*
setstring *{ s1[] "This is my home.\n" }*
setstring *{ s2 #s[3 2] }*

5.14.1.4 initstring *{ subspec strspec }*

Sets string elements contained in the element sub-table specified by *subspec*, to the value specified by *strspec*. The form of *strspec* is the same as for function **setstring**.

5.14.1.5 copystringvar *{ varname1 varname2 }*

Does the same as **copymatrixvar**, but for string variables.

5.14.1.6 **movestringvar** *{ varname1 varname2 }*

Does the same as **movematrixvar**, but for string variables.

5.14.1.7 **deletestringvar** *{ varname }*

Does the same as **deletematrixvar**, but for string variables.

5.14.1.8 **printstringvar** *{ varname }*

Does the same as **printmatrixvar**, but for string variables.

5.14.1.9 **fprintstringvar** *{ varname }*

Does the same as **printmatrixvar**, but for string variables.

5.14.1.10 **dprintstringvar** *{ varname }*

Does the same as **drintmatrixvar**, but for string variables.

5.14.1.11 **copystring** *{ subspec1 subspec2 }*

Does the same as **copymatrix**, but for string variables.

5.14.1.12 **movestring** *{ subspec1 subspec2 }*

Does the same as **movematrix**, but for string variables.

5.14.1.13 **deletestring** *{ subspec }*

Does the same as **deletematrix**, but for string variables.

5.14.1.14 **printstring** *{ subspec }*

Does the same as **printmatrix**, but for string variables.

5.14.1.15 **fprintstring** *{ subspec }*

Does the same as **fprintmatrix**, but for string variables.

5.14.1.16 **dprintstring** *{ subspec }*

Does the same as **dprintmatrix**, but for string variables.

5.14.1.17 **printstring0** *{ elspec }*

Prints the value of the string element specified by *elspec* to programme standard output. Only bare string is printed without any accompanying comment, spaces or double quotes.

5.14.1.18 **fprintstring0** *{ elspec }*

Does the same as **printstring0**, except that it writes to the programme output file instead of standard output.

5.14.1.19 **fileprintstring0** *{ filespec elspec }*

Does the same as **printstring0**, except that it writes to the file specified by *filespec* instead of standard output.

5.14.1.20 **setstringcomponents** *{ subspec expr }*, shortly **setstrcomp**

This function is similar to its matrix equivalent **setmatcomp**. It sets individual characters of all strings contained in the element sub-table specified by *subspec*, to the value of the expression *expr*. The expression *expr* is evaluated by the expression evaluator for each component separately. The function iterates over all strings of the sub-table specified by *subspec* and over all characters of these strings and assigns them values specified by *expr*. These values must be integers between 1 and 255. They represent ASCII codes of the assigned characters.

Two special expression evaluator's functions, **varindex** and **varcomponent** are designed for use with functions like **setstrcomp**. When the expressions *expr* is being evaluated for a specific component of a specific string element, the **varindex** function returns a specific index of the string element that is affected. The **varcomponent** function returns the sequential number of the character which is currently in the evaluation procedure. It must be called with argument 1, although string objects have only one dimension anyway (because of compatibility reasons).

Example:

Let us create a string variable *s* which contains the string of the first four characters of the alphabet, i.e. "abcd". Then let us change this string so that it will contain the next four characters of the alphabet. This is done by the following sequence of commands:

setstring *{s abcd }*
= { last: getstring["s",getstring["s",0]] }
setstrcomp *{s last+varcomponent[1] }*

In the first line we create a string object *s* and initialize it to "abcd". In the second line ASCII code of the last character of the string is obtained and assigned to calculator variable *last*. Then characters of the string are set to new values so that the first character has ASCII code *last+1* and the consequent characters have ASCII codes incremented by one. String object *s* so becomes "efg".

5.14.1.21 **setstrcompcond** *{ subspec (cond) expr }*

Does the same as **setstrcomp**, except that only those string characters (places) specified by *subspec* are set for which the condition *cond* is satisfied. *cond* is a mathematical expression that can also include calls to expression evaluator's functions **varindex** and **varcomponent**.

5.14.1.22 **stringcat** *{ subspec1 subspec2 subspec3 }*

Concatenates strings specified by *subspec1* and *subspec2*, and stores resulting strings to elements specified by *subspec3*. This means that strings specified by *subspec2* are appended to strings specified by *subpspec1*. *supspec1, subspec2* and *subspec3* are specifications of subtables of elements of string variables. Operation of concatenation is performed on correspondent tripples of elements. Element sub-table specified by

subspec3 can be equivalent to the sub-table specified by *subspec1* or/and sub-table specified by *subspec2*.

5.14.1.23 **copystringpart** *{ subspec1 subspec2 from to }*

Copies parts of strings specified by *subspec1* to string elements specified by *subspec2*. *from* and *to* specify the first and the last character that is copied. If *from* is 0, strings are copied from the first character, and if *to* is zero, strings are copied to the last character. *subspec1* and *subspec2* specify sub-tables of string elements on which operation is performed. They can specify identical sub-tables, in this case original strings are replaced by copies of their parts.

5.14.1.24 **stringwrite** *{ elspec arg1 arg2 arg3 … }*

Prints on the string specified by *elspec* in the same way as function **write** prints to the programme standard output. *arg1*, *arg2*, etc. specify what is printed on the string. Their meaning is the same as the meaning of arguments of function **write**.

If *elspec* contains any indices, then the string element specified by *elspec* must exist before function execution. If *elspec* does not contain any indices, this is not necessary. If the specified element does not exist, a zero-rank string variable of the appropriate name is created first. Its only string element is then initialised to the contents printed according to arguments *arg1*, *arg2*, etc.

Execution of **stringwrite** owerwrites contents of the string specified by *elspec* if it has been initialised before.

Example:

Let us execute the following code:

```
newstring { errorstr[ 3] }
= { j : 1 }
while { (j<=3)
[
  stringwrite { errorstr[ $j ] "Error" ${1+0.1*j} " detected." }
  = { j:j+1 }
]}
errorstr[ 3] }
```

After execution of this code, elements of string vatiable *errorstr* have the following values:

errorstr[1]: "Error 1.1 detected."
errorstr[2]: "Error 1.2 detected."
errorstr[3]: "Error 1.3 detected."

5.14.1.25 **stringappend** *{ elspec arg1 arg2 arg3 … }*

Similar to **stringwrite**, except that the string specified by *elspec* is not just overwritten. What is printed according to arguments *arg1*, *arg2* etc., is appended at the

end of the string if it already exists. If the string element specified by *elspec* does not exist or is not initialised, this function acts exactly in the same manner as **stringwrite**.

Examples:

Let us execute the following commands:
newstring *{ s[2 3] }*
setstring *{ s[2 1] "/home/inverse/ex/opt/" }*
= *{ i: 2 }*
stringappend *{ s[2 1] "inv" ${i+1} ".cm" }*
After execution of this code, string *s[2 1]* will have value "*/home/inverse/ex/opt/inv3.cm*".

If we execute command
stringappend *{str1 "test" ${2+3} ".cm" }*
and string variable *str1* has not existed before, a new string variable of rank 0 *str1* is first created and its only element is set to "*test5.cm*".

5.14.1.26 **numtostring** *{ elspec num < numdigits > <avoidexp> }*

Converts number *num* to its string representation and assigns this string to the string element specified by *elspec*. If the string specified by *elspec* is already initialised, it is overwritten. An optional argument *numdigits* specifies the number of digits that are printed to the string (0 means default), and optional argument *avoidexp* specifies whether exponential notation should be avoided (see the note below).

The string element specified by *elspec* must exist before function execution if *elspec* contains any indices. If it does not, a zero-rank variable of the appropriate name is created first if it does not yet exist.

Note:

By default, this function avoids writing numbers in exponential notation (such as "1.3e-5", this number would be represented as "0.000013"). This is useful e.g. for preparing string representations that can be correctly interpreted by programs that do not understand the E notation, such as "Mathematica", however it may lead to inaccurate representation of small numbers (since the *numdigits* refers in this case to number of written digits after the decimal point, not number of written digits after the first non-zero digit as with exponential notation).

5.14.1.27 **appendnumtostring** *{ elspec num < numdigits > <avoidexp> }*

Similar to **numtostring**, except that the string specified by *elspec* is not just overwritten. String representation of *num* is appended at the end of the string if it already exists. If the string element specified by *elspec* does not exist or is not initialised, this function acts exactly in the same manner as **numtostring**.

See also description of **numtostring** and the note included in this description.

Example:
Let us execute the following code:
setstring *{ info[ ] "Pi: " }*
appendnumtostring *{ info[ ] ${2*arcsin[1]} 3 }*
After execution of this code, string *info[ ]* will have value "*Pi: 3.14*".

5.14.1.28 **stringtonum** *{ elspec varname }*

Converts string specifiet by *elspec* to a number and assigns it to the expression evaluator variable named *varname*. It is not necessary that the variable named *varname* already exists. The string specified by *elspec* must exist and contain a valid string representation of a number.

5.14.2 Expression Evaluator Functions for Manipulating String Variables

5.14.2.1 **getstring** *[varname which < elind1, elind2, ... >]*

Returns required information for a specific string object. *varname* is the name of a string variable. *elind1*, *elind2*, etc., are the indices which specify the string element on the variable's element table. *which* specifies which information should be returned. If which is 0, string length is returned. If it is greater than zero, an ASCII code of the character specified by *which* is returned.

If the variable named *varname* is of rank 0, then no indices *elind1*, *elind2*, etc. need to be specified.

5.14.2.2 **getstringdim** *[varname dimnum]*

The same as **getmatrixdim**, but for string variables.

5.15 File Variables

File objects are used in the shell to establish the connection with files on the disk. File objects hold basic information about files, i.e. the file pointer, the file name, the current position, and the mode in which the file is open. Each file object can be connected

with a physical file or not. Various shell's functions that deal with files shell access files through file objects.

Elements of file variables can be empty (uninitialised) which means that they contain no data and that they are not connected with any files.

5.15.1 File Interpreter's Functions for Manipulating File Variables

5.15.1.1 **newfile** *{ varname < [dim1, dim2, ...] > }*

Does the same as **newmatrix**, but for file variables.

5.15.1.2 **dimfile** *{ varname < [dim1, dim2, ...] > }*

Does the same as **dimmatrix**, but for file variables.

5.15.1.3 **setfile** *{ elspec filespec }*

Sets a file element specified by *elspec* to the value specified by *filespec*. The specification of a file element *elspec* consists of a variable name and an optional index list in square brackets, e.g. *f1[2,3]*. The index list is not necessary if the rank of the file variable is 0.

elspec must address an existing file element, except if no indices are specified. In this case a file variable of rank 0 is created before the file element is set.

filespec specifies the value of the file object. It consists of the file name (i.e. the name of the physical file) and optional mode in which the file is open:

filespec = filename < openmode >

openmode ia a string which defines how the file must be open. It consists of one or two letters, with the following possibilities: *r, w, a, r+, w+, a+* (Table 1).

Table 2: Meaning of different opening modes for files.

mode	meaning
r	Opens the file for reading. The file must have existed before if we want to open it in this mode. The file position is set to the beginning of the file (i.e. 1).
w	Opens the file for writing. If the file already exists, it is overwritten. If it does not exist, it is created anew.
a	Opens the file for appending. This option is similar to **w**, except that the old content of the file is not overwritten if it has existed before. The file position is set to the end of the file.
r+	The same as **r**, except that writing to the file is also possible.
w+	The same as **w**, except that reading from the file is also possible.
a+	The same as **a**, except that reading from the file is also possible.

The **setfile** function opens the file in the mode specified. When the file is not used any more, it should be removed by the *deletefile* function, which also closes the physical file connected to the file object.

If the **setfile** function is called with the *elspec* argument that specifies an existent and open file, then the result is unpredictable. Usually the file connected to that object is closed and re-open. An error message should also be launched.

Warning:

Files must be handled with care. Take into account the fact that only a limited number of files may be open simultaneously on a system, and especially the number of files that can be open simultaneously by a single program is quite limited. Therefore, if you need to deal with many files when solving some problem (e.g. when it is necessary to store different kinds of intermediate results into different files), take care that only a limited number of these files are actually open at a time. Close the file immediately when you don't need to access it any more. If there are iterative steps at which you need to write something to several files or read something from them, you might consider at every such step opening the first file, performing any necessary input/output operations on it, closing this file, and then proceed with the second, third and other files. On the other hand, successive I/O operations are slower if the file is closed and open between, therefore the described approach may not be the best one when successive input and output operations are frequent and there are only a few files involved.

5.15.1.4 **copyfilevar** *{ varname1 varname2 }*

Does the same as **copymatrixvar**, but for file variables.

5.15.1.5 **movefilevar** *{ varname1 varname2 }*

Does the same as **movematrixvar**, but for file variables.

5.15.1.6 **deletefilevar** *{ varname }*

Does the same as **deletematrixvar**, but for file variables. The files that are deleted are closed first.

5.15.1.7 **printfilevar** *{ varname }*

Does the same as **printmatrixvar**, but for file variables. The name of the physical file, the current position in the physical file and the opening mode of the file are printed for each file element.

5.15.1.8 **fprintfilevar** *{ varname }*

Does the same as **printmatrixvar**, but for file variables. The name of the physical file, the current position in the physical file and the opening mode of the file are printed for each file element.

5.15.1.9 **dprintfilevar** *{ varname }*

Does the same as d**printmatrixvar**, but for file variables. The name of the physical file, the current position in the physical file and the opening mode of the file are printed for each file element.

5.15.1.10 **copyfile** *{ subspec1 subspec2 }*

Does the same as **copymatrix**, but for file variables.

5.15.1.11 **movefile** *{ subspec1 subspec2 }*

Does the same as **movematrix**, but for file variables.

5.15.1.12 **deletefile** *{ subspec }*

Does the same as **deletematrix**, but for file variables. The files that are deleted are closed first.

5.15.1.13 **closefile** *{ subspec }*

Closes any open files connected to elements of file variables specified by *subspec*. The opening mode is deleted on the file elements. *subspec* can specify an element or sub-table of elements of a file variable.

5.15.1.14 **flushfile** *{ subspec }*

Flushes any open files connected to elements of file variables specified by *subspec*. The content of any related file buffers is updated in files, which ensures that the contents of write operations can not be lost if the program execution is broken for any reason.

5.15.1.15 **printfile** *{ subspec }*

Does the same as **printmatrix**, but for file variables.

5.15.1.16 **fprintfile** *{ subspec }*

Does the same as **fprintmatrix**, but for file variables.

5.15.1.17 **dprintfile** *{ subspec }*

Does the same as **dprintmatrix**, but for file variables.

5.15.2 Expression Evaluator's Functions for Manipulating File Variables

5.15.2.1 **getfile** *[varname dataid < elind1, elind2, ... >]*

Returns specific data about a specific file object. *varname* is the name of a file variable. *elind1*, *elind2*, etc., are the indices which specify the file element on the

variable's element table. If the variable named *varname* is of rank 0, then no indices *elind1*, *elind2*, etc. need to be specified.

dataid specifies which data about the file object must be returned. According to the value of *dataid*, the function returns the following:

dataid	what is returned by **getfile**
0	1 if the file is open, 0 otherwise.
1	1 if the file is open for reading, 0 otherwise.
2	1 if the file is open for writing, 0 otherwise.
3	The current position of the file; positions start with 1 and mean the positions in the file in bytes where the operations act.

5.15.2.2 getfiledim *[varname dimnum]*

The same as **getmatrixdim**, but for file variables.

5.16 Shell Variables with a Pre-defined Meaning

Some variables of the shell have a pre-defined meaning and are supposed to be used for a particular purpose. They are basically still user-defined variables and behave in many terms the same as other user-defined variables. They are distinguished in a sense that they are used to accomplish specific tasks. Besides, some operations that act on variables of specific types treat the pre-defined variables in a slightly different way, usually with some additional automatism.

Each pre-defined variable serves for specific tasks. Some of them store results of operations and algorithms without this should be specified by the user. These results are automatically available for further use after the appropriate operations are finished. The others serve for data exchange between different shell's functions, e.g. between optimisation algorithms and the function, which performs the direct analysis. Both kinds of variables are supposed to carry the information with exactly defined meaning, e.g. the current value of the objective function or the optimised values of parameters. Furtherly, we can find yet another group of pre-defined variables. These simply serve as variables on which certain operations act without the variable in use is explicitly specified. A typical example is the shell's output file (variable *outfile*) to which the output of the function **fwrite** (and many others) is directed, although this is never explicitly specified.

There is usually some additional automatism connected with the pre-defined variables. For example, when a pre-defined file *outfile* is open by the **setfile** function, the file is open for writing without this is explicitly specified in the argument block of the **setfile** command. When a pre-defined vector or matrix variable is set by the **setvector** or

setmatrix command, it is not necessary to specify the variable dimensions if the dimensions are known from what was done in the shell before.

To generalise the above statements, we can establish that there is usually some information related with the pre-defined variables which has more general meaning. Such information is for example the number of optimisation parameters. Dimensions of several pre-defined vector, matrix and scalar variables are bound to the number of parameter. Once the number of parameters is known to the shell, the appropriate dimensions of pre-defined variables that are bound to this information are also known and they need not to be specified in some functions which operate on variables.

Such general information as the number of parameters are stored in the shell in special places separated from the system of user-defined variables. These information either exist in advance (e.g. the information that the pre-defined file *outfile* should normally be open for writing) or are set when some related actions are undertaken. For example, the number of parameters is set when the appropriate dimension of any pre-defined variable related to this information is set (e.g. the dimension of the vector *parammom* or *paramopt*). The user can access (i.e. set and get) these information in two ways: through the dependent information about variables and through the appropriate expression evaluator's and file interpreter's functions.

5.16.1 Pre-defined Matrix, Vector and Scalar Variables

Matrix, vector and scalar pre-defined variables hold general information which is directly related to optimisation or inverse problems, like the initial, current and optimal values of parameters, experimental measurements and their errors, simulated measurements, the objective function and the constraint functions and their derivatives with respect to parameters, etc. It is obvious that groups of these variables are related through their dimensions. For example, the vector of (current) parameters has the same dimension as the gradient of the objective function.

General information related to these pre-defined variables are listed in Table 3. These information are not stored in the user-defined variables, but in the internal shell's variables. They can be obtained and set through special expression evaluator's and file interpreter's functions.

Table 3: general information related to some pre-defined matrix, vector, and scalar variables:

information	**meaning**
numparam	The number of optimisation parameters
numconstraints	The number of constraint functions
numobjectives	The number of the objective functions (usually equals 1)
nummeas	The number of measurements (applicable for inverse problems)

Some automatism is built in the shell regarding the pre-defined matrix, vector and scalar variable. This especially concerns the file interpreter's functions that assign values to variable elements or allocate variables. When a specific dimensions of any of these pre-defined variables or their elements is set, the appropriate internal shell's variable related to that dimension changes accordingly.

The opposite is also valid: When we call functions that set the values of variable elements, we do not need to specify dimensions that are determined by the shell's internal variables that have defined values.

Example:

Let's say that we set the vector of initial guesses *param0* with dimension 3:

setvector *{param0 3 { 2.43 5.34 6.444 }}*

Since this pre-defined vector has by definition the dimension *numparam*, the shell's internal variable *numparam* is automatically set to 3. The same effect would have, for example, the code

setvector *{paramopt 3 }*

since the pre-defined vector *paramopt* has by definition the same dimension (*numparam*). If we then set the vector of current parameter values *parammom*, we do not need to specify its dimension, although the vector variable does not yet exist and will be created anew. This is because the dimension of this vector, which is by definition *numparam*, is already known since it was set implicitly when the dimension of the vector *param0* or *paramopt* was set. Therefore, we can set the vector *parammom* simply by specifying components:

setvector *{parammom { 1.64 22.3 101.1 }}*

5.16.1.1 Counter Pre-defined variables

Pre-defined counter variables *calcobj*, *calcconstr*, *calcgradobj* and *calcgradconstr* are reserved for evaluation flags that are intended to specify whether a given type of response should be evaluated during the analysis or not, and afterwards to carry return information on whether the corresponding response has actually been evaluated. What concerns the command file, these flags are typically checked and set within the *analysis* block.

They are usually set by internal analysis functions according to what is expected to be calculated during a given analysis. These flags an be checked within the analysis block of interpreter code to determine which parts of evaluation code must be executed and which can eventually be skipped. If some part of the response can not be calculated then value of the corresponding flags should be set to 0. *calcobj* refers to evaluation of the objective function, *calcconstr* to evaluation of constraint functions, *calcgradobj* to evaluation of objective function gradient, and *calcgradconstr* to evaluation of constraint function gradients.

pre-defined scalar variable	**characterisation of the variable**

calcobj []	Intended to specify within the *analysis* block whether to calculate the objective function (not defined or defined with non-zero value) and to carry return information (after the analysis block is interpreted) whether the objective function has been calculated or not.
calcconstr []	Intended to specify whether to calculate constraint functions or not and to carry the corresponding return information.
calcgradobj []	Intended to specify whether to calculate objective gradients or not and to carry the corresponding return information.
Calcgradconstr []	Intended to specify whether to calculate constraint gradients or not and to carry the corresponding return information.

5.16.1.2 Scalar Pre-defined Variables

pre-defined scalar variable	characterisation of the variable
objectivemom [] < [numobjectives] >	Value(s) of the objective function(s) at the current parameter values.
objectiveopt [] < [numobjectives] >	Value(s) of the objective function(s) at the optimal values of parameters.
objective0 [] < [numobjectives] >	Value(s) of the objective function(s) at the initial guess.
constraintmom [numconstraints]	Values of the constraint functions at the current values of parameters.
constraintopt [numconstraints]	Values of the constraint functions at the optimal values of parameters.
constraint0 [numconstraints]	Values of the constraint functions at the initial guess.

5.16.1.3 Vector Pre-defined Variables

pre-defined vector variable	characterisation of the variable
parammom [] (numparam)	Current values of parameters.
paramopt [] (numparam)	Optimal values of parameters.
param0 [] (numparam)	Initial guess for parameters.
parammomold [] (numparam)	A copy of current parameter values (usually made when the current parameters are transformed).
transf [] (numparam)	Vector of transformed parameters.
direction [] (numparam)	Direction of a line search.
startpoint [] (numparam)	Initial point of the line search.
meas [] (nummeas)	Experimental measurements (used at inverse analyses)
sigma [] (nummeas)	Vector of measurement errors.
measexact [] (nummeas)	Exact measurements; Imaginary quantity that is used e.g. at monte carlo simulations with aim to estimate

	the influence of the measurement errors on the results.
measmom [] (nummeas)	Current values of simulated measurements.
measopt [] (nummeas)	Simulated measurements in the optimum (i.e. solution of the inverse problem).
meas0 [] (nummeas)	Simulated measurements at the starting guess (i.i. *param0*).
gradobjectivemom [] <[numobjectives]> (numparam)	Gradient of the objective function(s) at the current parameter values.
gradobjectiveopt [] <[numobjectives]> (numparam)	Gradient of the objective function(s) at the optimum.
gradobjective0 [] <[numobjectives]> (numparam)	Gradient of the objective function(s) at the initial guess.
gradconstraintmom [numconstraints] (numparm)	Gradients of the constraint functions at the current parameter values.
gradconstraintopt [numconstraints] (numparam)	Gradients of the constraint functions at the optimal values of parameters.
gradconstraint0 [numconstraints] (numparam)	Gradients of the constraint functions at the initial guess.
gradmeasmom [nummeas] (numparam)	Gradients of the simulated measurements at the current values of parameters.
gradmeasopt [nummeas] (numparam)	Gradients of the simulated measurements at the optimum.
gradmeas0 [nummeas] (numparam)	Gradients of the simulated measurements at the initial guess.

5.16.1.4Matrix Pre-defined Variables

pre-defined matrix variable	characterisation of the variable
der2objectivemom [] < [numobjectives] > (numparam,numparam)	Second derivatives (Hessian) of the objective function(s) at the current values of parameters.
der2objectiveopt[] < [numobjectives] > (numparam,numparam)	Second derivatives (Hessian) of the objective function(s) at the optimal values of parameters.
der2objective0[] < [numobjectives] > (numparam,numparam)	Second derivatives (Hessian) of the objective function(s) at the initial guess.
der2constraintmom [numconstraints] (numparam,numparam)	Second derivatives (Hessian) of the constraint functions at the current values of parameters.
der2constraintopt [numconstraints] (numparam,numparam)	Second derivatives (Hessian) of the constraint functions at the optimal values of parameters.
der2constraint0 [numconstraints] (numparam,numparam)	Second derivatives (Hessian) of the constraint functions at the initial guess.
der2measmom [nummeas] (numparam,numparam)	Second derivatives (Hessian) of the simulated measurements at the current values of parameters.
der2measopt [nummeas] (numparam,numparam)	Second derivatives (Hessian) of the simulated measurements at the optimal values of parameters.
der2meas0 [nummeas] (numparam,numparam)	Second derivatives (Hessian) of the simulated measurements at the initial guess.

5.16.2 File Interpreter's Functions for Setting Shell's Internal Data Related to Pre-defined Variables

5.16.2.1 setnumparam *{ val }*

Sets the number of parameters *numparam* to *val*. Val is a numerical argument, which means that it can be given as a number, as a mathematical expression in the form *${expr}* or as an expression evaluator's variable in the form *$varname*. If a non-integer value is given for *val*, it is rounded.

5.16.2.2 setnumobjectives *{ val }*

Sets the number of objective functions *numobjective* to *val*. By default, this number is set to 1. Val is a numerical argument, which means that it can be given as a number, as a mathematical expression in the form *${expr}* or as an expression evaluator's variable in the form *$varname*. If a non-integer value is given for *val*, it is rounded.

5.16.2.3 setnumconstraints *{ val }*

Sets the number of constraint functions *numconstraints* to *val*. Val is a numerical argument, which means that it can be given as a number, as a mathematical expression in the form *${expr}* or as an expression evaluator's variable in the form *$varname*. If a non-integer value is given for *val*, it is rounded.

5.16.2.4 setnummeas *{ val }*

Sets the number of measurements *nummeas* to *val*. Val is a numerical argument, which means that it can be given as a number, as a mathematical expression in the form *${expr}* or as an expression evaluator's variable in the form *$varname*. If a non-integer value is given for *val*, it is rounded.

5.16.3 Expression Evaluator's Functions for Accessing Shell's Internal Data Related to Pre-defined Variables

5.16.3.1 getnumparam *[ ]*

Returns the number of parameters *numparam*.

5.16.3.2 getnumobjectives *[ ]*

Returns the number of objective functions *numobjectives*.

5.16.3.3 getnumconstraints *[ ]*

Returns the number of constraint functions *numconstraints*.

5.16.3.4 getnummeas *[ ]*

Returns the number of measurements *nummeas*.

5.16.4 Pre-defined File Variables

The pre-defined file variables represent files which have a specific meaning in the shell, e.g. the shell's output file or the input and output file of the direct analysis. Some operations act exclusively in these pre-defined files. For example, the **fwrite** function prints its output to a pre-defined file *outfile*.

The pre-defined file variables should have rank 0, although this is not always necessary. However, in all cases only the first element of the file variable's element table can be involved in operations that act on pre-defined files. The user should avoid defining pre-defined variables with rank greater than zero.

5.16.4.1 File Pre-defined Variables

pre-defined file variable	characterisation of the variable
infile []	Shell-s input file.
outfile []	Shell-s output file.
aninfile []	The simulation (direct analysis) input file.
anoutfile []	The simulation (direct analysis) output file.