4 OPTIMISATION SHELL “INVERSE”

4.1 Aims and Basic Structure of the Shell

4.1.1 Basic Ideas

As mentioned in the introductory part, the main purpose of the optimisation shell “Inverse” is to utilize a finite element method based simulation code for solving inverse and optimisation problems. The philosophy of the shell[4]-[6] follows the idea that two naturally distinct parts can be recognized in the solution scheme of optimisation problems (Figure 4.1).

One part of the solution procedure is the solution of a direct problem at given optimisation parameters. This comprises numerical solution of the equations that govern the system of interest. In the scope of this work, this is performed by a finite element simulation, referenced in chapter 2 and schematically shown in a dashed frame on the right-hand side of Figure 4.1.

It is regarded that when a set of optimisation parameters is given, the system is completely determined in the sense that any quantity required by the optimisation algorithm can be evaluated. This usually refers to the value of the objective and constrained function and possibly their derivatives for a given set of parameters. Evaluation of these quantities is referred to as direct analysis[1] and is shown in the larger dashed frame in Figure 4.1.

[1]The notion of direct analysis is not used in a uniform way in the literature. Some authors use this term to denote merely the numerical simulation and sometimes the term is not defined strictly. In the present work the term “direct analysis” refers strictly to evaluation of the relevant quantities (e.g. the value of the objective and constraint functions) at a given set of optimisation parameters and includes all tasks that are performed as a part of this evaluation.

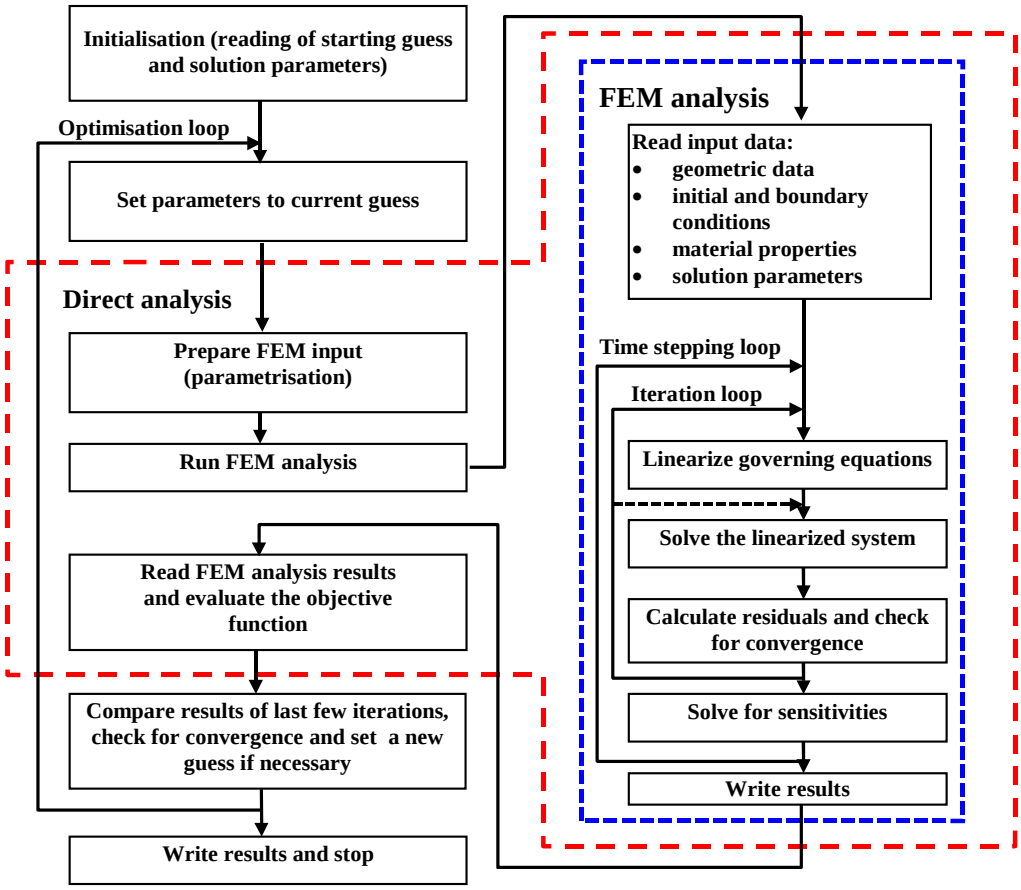

Figure 4.1: Typical solution scheme for an optimisation problem.

In order to utilise an existing simulation environment for solution of inverse and optimisation problems, the optimisation shell performs tasks on the left-hand side of Figure 4.1. Taking into consideration merely the solution scheme as shown in this figure, these tasks can be further divided into two parts. The part not included in the larger frame obviously represents an optimisation algorithm in its most basic sense (i.e. as was treated in the previous chapter). The part included in the frame represents those tasks of the direct analysis that are not performed by the simulation environment. This part can be viewed as an interface between the optimisation algorithm and the simulation.

The above discussion indicates that two basic elements of an optimisation system, i.e. optimisation algorithms and simulation tools, can be implemented as physically separate parts. This is one of the key ideas followed by the present work and is clearly reflected in separate and independent treatment in chapters 2 and 3. It

must be emphasised that the above statements do not exclude dependency between optimisation algorithms and solution algorithms for the direct problem applied in specific cases. It is evident that close correlation between different numerical algorithms applied in the solution scheme of a specific problem is not excluded and was actually stressed at the end of the previous chapter. For example, whether analytical derivatives are provided by the simulation module or not usually plays a crucial role in defining the optimisation algorithm whose use will result in the most effective overall solution of the problem. This however does not affect physically separate treatment or implementation of either algorithm.

The scheme in Figure 4.1 is restricted to a solution process of a specific problem. When an optimisation system is treated, it is also important how the problem is defined and which solution strategies can be applied by using the system. In this respect it is significant to consider individual tools and algorithms implemented within the system, operational relations and interfaces between the parts of the system, which enable synchronous function, and finally the user interface.

Figure 4.2 outlines the operation of the presented optimisation system. It consists of the optimisation shell and the finite element simulation environment. In the solution scheme, the shell performs the tasks on the left-hand side of Figure 4.1, while the simulation environment is employed in solution of the direct problem, which corresponds to the tasks shown in the right-hand side of the figure.

The direct problem solved within the optimisation loop is determined by the values of the optimisation parameters. The problem is determined when boundary conditions, geometry, constitutive relations, etc. are known. These represent input data for numerical simulation. A transformation between optimisation parameters and the input data must therefore be defined. This transformation is referred to as parametrisation and is shown in Figure 4.1 as the first task of a direct analysis. This task is typically performed by the shell.

Optimisation parameters usually affect only a part of the simulation input data[1]. It is therefore advantageous to prepare a skeleton of the direct problem in advance and use it as a template for parametrisation. Most conveniently this skeleton is a definition of a direct problem at a specific set of optimisation parameters. It is usually created using pre-processing facilities of the simulation environment as shown in Figure 4.2.

The optimisation shell changes the affected input data according to the values of optimisation parameters. In the figure, this is done by updating the simulation input file, but can also be done directly by manipulating the data structure of the

[1] In this respect we talk more specifically about parametrisation of individual components, e.g. parametrisation of shape (or domain), parametrisation of material models, etc.

simulation programme, provided that the shell and the simulation unit can share data structures in the memory. After the simulation is performed, the shell reads the results and evaluates the quantities required by the optimisation algorithm. The shell must therefore be able to access the necessary simulation results. In the figure, results are accessed through the simulation output file, but a direct access can also be implemented.

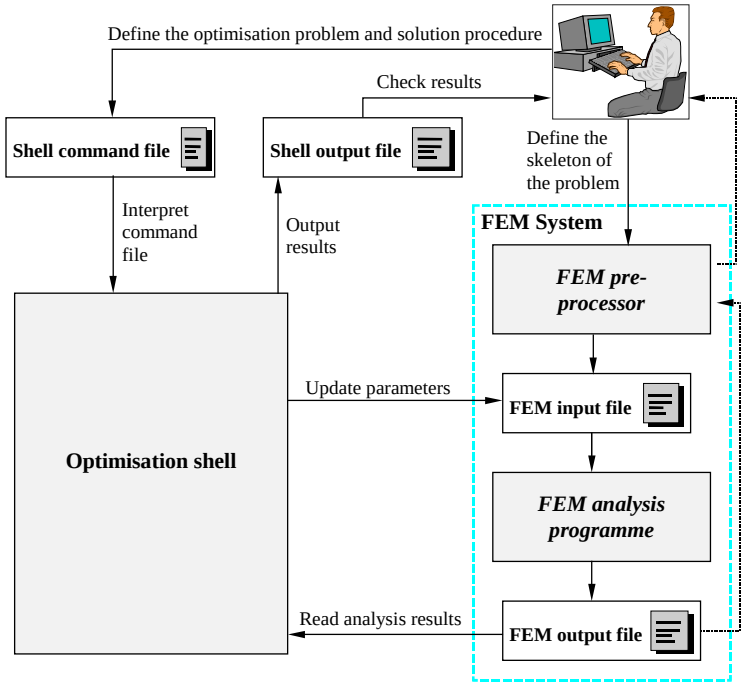

Figure 4.2: Operation scheme of the optimisation system.

Changing input data for numerical simulation, running the simulation and reading its results are performed by interface utilities of the shell. These are direct analysis tasks performed by the shell and are shown within the left-hand side of the larger frame in Figure 4.1. The shell can also perform a certain amount of processing of simulation results. It is not strictly defined which parametrisation and result processing tasks are performed by the shell and which by the simulation environment. This depends mostly on the capabilities of the simulation pre- and post-processing modules. The shell should permit employment of available capabilities in the simulation environment if these are convenient for performing the relevant tasks.

The optimisation shell includes implementation of various optimisation algorithms and other tools which can serve in the solution of optimisation problems. These tools are accessible through the shell user interface, which is separated from the user interface of the simulation environment. The current user interface is implemented through the shell input file in which the user defines the problem, and

the shell output file, where the shell writes its results (Figure 4.2). Unlike traditional simulation input files, which consist of various sets of data written in some prescribed format, the shell input file consists of commands that are interpreted by its built-in interpreter. It is therefore commonly referred to as the shell command file.

4.1.2 Requirements Taken into Consideration

Before continuing with description of the optimisation shell, it is appropriate to mention some requirements which affect its design[1]-[3],[7]. These requirements will be referenced later in the text in order to justify certain design aspects.

The basic demand for a good optimisation system is flexibility. It must be possible to apply the system to a large variety of problems that can possibly appear. This concerns definition of the problem itself as well as definition of the solution strategy. On one hand this flexibility is determined by the set of tools for solution of different subproblems, which are offered by the system. On the other hand the conceptual structure of the system should not impose any fundamental restrictions on the way how different tools can be combined to solve complex problems.

Somehow conflicting with flexibility is the demand for simplicity of use. Logical structure of the system is a prerequisite for avoiding conflicts induced by these two demands. A system is easy to be applied for certain types of problems if the user is required to provide only that information necessary to define the particular problem and if the requirements are set by the system in a clear way. It is obvious that this can be achieved only on a case to case basis, so that all particularities can be taken into account. The system must be structured hierarchically, so that high level easy-to-use tools for particular problems can be implemented by templates built on the lower-level basis. These tools can introduce additional concepts, but these should apply locally and should not affect the underlying system, which can be still applied independently and should retain generality.

A group of requirements is related to the economy of the system development. This essentially means the ability of achieving the best possible effect with limited development resources. The effect is measured in terms of applicability of the system to various problems the potential user can be faced with and in terms of effort needed for problem definition and computer time needed for problem solution. Beside logical structure of the system, economy of development is mostly related to its portability, modularity and openness.

Portability means that the system can be easily transferred from one computational platform to another. A portable system can be developed in any homogeneous or heterogeneous computer environment so that its transfer to a different environment requires minimal additional development effort. It is most

easily achieved by using portable development tools. Due to portability reasons most of the system was developed in ANSI C[25],[26]. Compilers based on the ANSI C standard are available on most existing computer architectures and are usually implemented in a strict enough manner that it is possible to transfer programmes between different platforms without major modifications. Use of non-standard libraries has been avoided as much as possible in the shell development. Where system dependent details could not be completely avoided, they were captured in isolated and clearly distinguished locations which are easy to identify and modify when transfer to a new platform is performed. One of the development principles that were taken into account is also that turning off system dependent details should affect as little functionality as possible. In this way detrimental effects of non-standard behaviour of any system part can usually be avoided to a great extent.

The modular structure of the system also has beneficial effect on economy of development. In a modular system, tools that constitute its functionality are implemented in separate units. Development of these units must be as independent as possible, so that development and change of specific tools does not affect and is not affected by existent structure and functionality. Modularity is best achieved by imposing a limited number of clear general rules on the system and providing a simple implementation interface for development of new modules. This interface must be such that it does not restrict the range of tools which can potentially be added to the system. It must be possible to apply this interface to existing general tools which were not primarily developed to be included in the system. Optimisation algorithms provide a good example of these principles. The main concern of an optimisation algorithm is to effectively locate a constrained local minimiser to a given accuracy, i. e. with as few function evaluations and housekeeping operations as possible. If the algorithm is used as a part of a complex optimisation system, a number of additional implementation details must be solved such as interaction with the simulation environment. However, this should not affect development of the algorithm itself, because the aim of the algorithm remains the same. Additional requirements such as interaction with simulation environments must be overcome by the implementation interface, which is used at the final stage when the already implemented algorithm is built into the system.

Openness of the system includes two aspects. The first aspect regards the definition of the problem. In this respect openness means that the user can easily access various built-in utilities and employ external programmes using the available interfacing utilities when defining a solution strategy for the problem to be solved. This has strong impact on the flexibility of the system. The development aspect of openness means that existing functionality can be directly employed when developing additional tools in the system. The shell can therefore be easily integrated with other programmes and different modules can be developed independently and merged together. This facilitates development of higher level and more case specific tools on the basis of lower-level utilities. Openness of the system is to a large extent conditional on its modularity.

4.1.3 Operation Synopsis

The optimisation shell “Inverse” operates on the basis of an input (or command) file, in which the user defines what the shell should do. The problem to be solved is not defined in a descriptive way as is usually the case with simulation programmes, but as a set of instructions for the shell, which ensures sufficient flexibility of the user interface[5],[6].

Figure 4.3 shows how parts of the optimisation system interact in the solution procedure. Any action of the optimisation shell is triggered by the corresponding command in the command file. The shell file interpreter[15] reads commands one by one and runs internal interpreter functions that correspond to them. The built-in optimisation algorithms and other utilities such as mathematical tools (function approximation, matrix operations etc.) or interfacing with the simulation, are accessed through the interpreter functions. Each command in the command file has its own argument block through which arguments can be passed to corresponding functions. The argument block is enclosed in curly brackets that follow the command.

The shell includes a general built-in function that performs the direct analysis (Figure 4.3). Optimisation algorithms and some other utilities such as tabulating functions call this function for evaluation of necessary quantities such as values of the objective and constraint functions. Actually there is an additional interface function between this function and any calling algorithm (not shown in the figure). This function covers specificity of the algorithm regarding input and output of the direct analysis. This includes formats of function arguments prescribed by the specific algorithm. It also concerns the fact that different algorithms require different data to be evaluated, e.g. some of them require derivatives and the others do not.

The general analysis function runs interpretation of a specific block in the shell command file, referred to as the analysis block. This block is so interpreted every time the direct analysis is performed and is therefore used for user definition of the direct analysis. Since the complete interface with the simulation environment is accessible through interpreter commands, the user can precisely define how the numerical simulation at specific values of optimisation parameters is performed and how data is transferred between the shell and simulation environment. The analysis block is physically an argument block of the *analysis* command. When this command is encountered by the interpreter, the position of its argument block is stored so that the block can be interpreted any time by the general analysis function.

There must exist a data link for transferring input and output parameters of the direct analysis between the calling algorithm and user definition in the analysis block. The data is passed through function arguments between the algorithm and the direct analysis function called by that algorithm. The data link between this function

and user definition is established through pre-defined variables. These variables uniquely define the place where particular input and output data of the direct analysis is stored. The internal analysis functions automatically update input data obtained by the algorithm (e.g. values of optimisation parameters) on the appropriate pre-defined location. After interpretation of the analysis block it retrieves the data to be returned to the calling algorithm (e.g. values of the objective and constraint functions) from the locations defined for this purpose. The user can access these locations through interpreter functions for accessing variables. The user must ensure in the analysis definition that analysis results are correctly evaluated and stored to the appropriate locations, where they can be retrieved by the analysis function and returned to the algorithm[18].

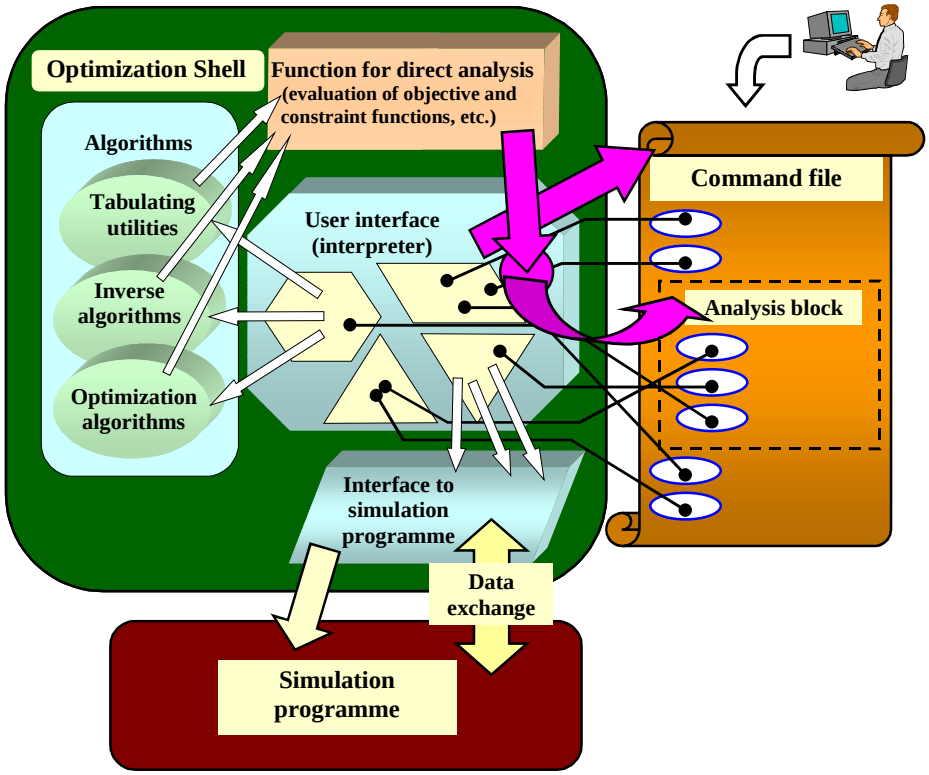

Figure 4.3: Structure and operation of the optimisation shell.

Beside interpreter commands for accessing various built-in tools, there are also commands for controlling the flow of interpretation, such as branching and looping commands, for example. Programming capabilities of the interpreter are supplemented by a system of user-defined variables[17] and a system for evaluation of

mathematical expressions[16]. The shell can therefore be programmed in a way that resembles a high level programming language[1].

The shell design allows the user to interact with the solution process at several levels. All tools and algorithms provided by the shell are run from the command file. Their output results can be stored in user-defined variables, and so used as input for other built-in algorithms. The available utilities can therefore be easily combined as necessary when applied to the solution of more complex problems. This feature is further enhanced by programming capabilities of the interpreter, which makes the shell very flexible and applicable to a large variety of problems.

At first sight the consequence of such a flexible user interface implemented through the interpreter is that the optimisation system can not be made easy to use. It might seem that solution of any optimisation problem requires detailed programming of the solution process, which is only assisted by built-in utilities.

This is not an entirely correct impression. For any set of similar problems it is possible to implement a high level interface in such a way that definition of the problem and the solution procedure require a minimum amount of user interference. Such an interface can be built by templates written for the shell interpreter. User interaction can be reduced merely to insertion of input data, which can eventually be assisted by an external user interface.

Such high level interfaces inevitably restrict the range of problems that can be solved by the system. Their use is adequate when the optimisation system is used for highly specialised purposes. Another way of making the system easier to use is to introduce high level functions that perform complex tasks or combination of groups of tasks, which appear steadily in a larger group of related problems. This can result in a hierarchical structure of utilities where the user can decide which level to use. High level tools make the use of the system easier without imposing a priori restrictions on flexibility. New higher level commands can be created by combination of existing lower level utilities using the shell interpreter. In this way interventions in the shell source code are avoided, which reduces the level of skill necessary for implementation of such tasks.

By implementation of hierarchically structured sets of lower and higher level commands, the two fundamentally conflicting demands for flexibility and simplicity of use can be compromised. Currently the most urgent problem with the shell is that many necessary sets of high level specialised commands are not as yet implemented, which is especially expressive at interfacing the simulation environments.

[1] In this respect file interpreter commands are also referred to as functions. This is sometimes better to be avoided in order to avoid ambiguity and confusion of interpreter commands and internal functions of which the shell consists.

4.2 Function of the Shell

The discussion in the previous sections was centered around the basic concepts of the shell. In order to make these things less abstract, a few more details regarding the shell function are given in the present section. Some details will be cleared from the user point of view. The intention of this section is however not to serve as a user reference, but merely to give more insight into how previously described concepts are reflected when the shell is applied to the solution of problems. Detailed reference of the existing functionality of the shell exist in the form of manuals available on the Internet[12],[14]-[22].

4.2.1 Basic File Interpreter Syntax

The basic file interpreter syntax is simple:

```
command1 { arguments } command2 { arguments } …
```

When a shell command file is interpreted, the interpreter simply searches for commands, locates their argument blocks and passes control to the appropriate functions that are in charge of execution. For each interpreter command there exists an appropriate function installed in the file interpreter system. These functions are usually just an interface between commands in the command file and those functions which really do the job, and are referred to as interpreter functions in this text. Interpreter functions take care of the correct transfer of arguments from the argument block in the command file and for imposing additional syntax and other rules imposed by the optimisation shell. Such two stage arrangement makes it possible to easily incorporate functions and modules that were not primarily developed for use in the shell. Separation of the concepts imposed by the shell and those implied by a specifically incorporated function or module is achieved in this way. The two stage calling arrangement is evident from Figure 4.3.

The file interpreter is supported by the system for evaluation of mathematical expressions or expression evaluator. This is an independent system of the shell. Its capabilities are accessed by the file interpreter functions for treatment of their arguments. The basic functionality offered by the expression evaluator is evaluation of mathematical expression with the ability of defining new variables and functions. For the interpreter itself the most important use of the expression evaluator is evaluation of conditions in branching and looping commands. This enables the interpreter to be used as a programming language.

Control of the interpretation flow[13][15] is implemented through branching and looping interpreter commands and through the function definition utility. All related

functionality is treated as a part of the file interpreter. The syntax of these utilities is described below.

The *if* command interprets a block of code in the basis of the value of a condition. Its syntax is the following:

```
if { (condition) [ block1 ] else [ block 2 ] }
```

If the condition in round brackets is true (that is non-zero), then the block of code *block1* is interpreted, otherwise the block *block2* is interpreted. The condition is evaluated by the expression evaluator.

The *while* commands repeatedly interprets a block of code. The block is being interpreted as long as the condition remains satisfied (i.e. the value of the condition expression is non-zero). The syntax is the following:

```
while { ( condition ) [ block ] }
```

The condition in the round bracket is evaluated by the expression evaluator in each iteration of the loop before the code block in square brackets is interpreted. The first evaluation of the condition expression as zero causes exit of the loop and interpretation is continued after the *while* command argument block. Typically the code block contains commands that affect the value of the condition expression, so that after a certain number of iterations the value of the expression becomes zero and the loop exits.

Similar to the while command is the *do* command, except that the condition expression is evaluated after interpretation of the code block and therefore the block is interpreted at least once. Its syntax is the following:

```
do { [ block ] while (condition) }
```

Beside standard branching and looping commands, the ability of defining new interpreter commands is relevant. This is referred to as the function definition utility and is also implemented through an interpreter command. This utility enables implementation of commands that perform combined tasks by employing existing commands. Higher level commands can therefore be implemented without interference in the shell source code. Commands defined by the function definition utility behave in a similar manner to the built-in commands.

New interpreter commands are defined by using the *function* command. Its syntax is the following:

```
function { funcname ( arg1 arg2 … ) [ defblock ] }
```

funcname is the name of the new function, *arg1*, *arg2*, etc. are names of function arguments, and *defblock* is the definition block of the new command. The *function* command makes the interpreter install a new command, which includes storage of the function argument list and position of the definition block. When the newly defined command is encountered later during interpretation, its definition block is interpreted. Occurrences of arguments in the definition block are replaced by actual arguments prior to interpretation. The replacement is made on a string basis, so that the meaning of arguments is not prescribed by the definition of the new function[15]. In the definition block, the arguments must be marked by argument names preceeded by the hash sign (‘#’). The interpreter can in this way recognise occurrences of arguments and replace them by actual arguments stated in the argument block of the called command.

A clear example[13] of how the function definition utility can be used is given by the implementation of the for loop through the interpreter. This can be done in the following way[1]:

```
1. function { for ( begin condition end body )
2. [
3.   #begin
4.   while { ( #condition )
5.   [
6.     #body
7.     #end
8.   ] }
9. ] }
```

The function requires four arguments. *begin* is the code block interpreted before the loop is entered. *condition* is the looping condition that is checked before every iteration of the loop. It must be an expression that can be evaluated in the expression evaluator. *body* is the code segment that is interpreted in the loop, and *end* is the code segment that is interpreted after the loop. Using the newly defined function *for*, the following code will print numbers from 1 to 5 to the standard output:

```
for { ={i:1}   i<=5   ={i:i+1}
  { write { $i "\n" } }
}
```

When the interpreter encounters the command for, it replaces formal arguments in the definition block (lines 2 to 8 in the definition segment of the code) with actual arguments and interprets the definition block. The resulting code that is actually interpreted is then as follows:

[1]Line numbers simply enable referencing portions of code. In the command file lines are not numbered.

```
={i:1}
while { ( i<=5 )
[
  write { $i "\n" }
  ={i:i+1}
] }
```

4.2.2 Expression Evaluator

The expression evaluator (succinctly referred to as the calculator)[16] is an independent shell module. Support to control of the interpretation flow is one of its basic tasks, therefore the interpreter and the expression evaluator are inseparably connected.

The calculator contains a set of built-in mathematical functions and operators, which can be arbitrarily combined with variables and numbers to form expressions. The calculator system currently supports only scalar variables. The syntax for forming mathematical expressions is standard and is described in detail in [16].

The file interpreter commands = and *$* serve for user interaction with the expression evaluator.

The syntax of the = command is the following:

```
= { varname: expression }
```

The expression is first evaluated by the calculator and its value is assigned to the calculator variable named *varname*.

The *$* commands calculator variables and functions in terms of expressions. Definition of a variable has the following syntax:

```
$ { varname : expression }
```

The expression is not evaluated at execution of this command. It is assigned to the variable as an expression that defines how the value of the variable is calculated. If the value of any part of the defining expression is changed, this affects the value of the variable. It is not even necessary that all calculator variables and functions that appear in the expression are defined at the time the *$* command is interpreted. The value of the variable becomes defined as soon as all variables and functions appearing in the expression are defined.

Definition of new expression evaluator functions have the following syntax:

$ { funcname [arg1, arg2, …] : expression }

The defining expression contains formal arguments *arg1*, *arg2*. After the definition, the function can be used in the same way as built-in expression evaluator functions. Function evaluation consists of evaluation of the defining expression after replacement of formal arguments by the values of actual arguments.

The definition of new calculator functions may, as definition of variables, include variables and functions that are not yet defined. The use of functions = and *$* is illustrated by the following example:

```
1. ${ a: cubesum[b,c] }
2. ${cubesum[x,y]: (x+y)^3 }
3. ={ b: 1 }
4. ={ c: 2 }
5. write { $a }
```

The first line defines a new calculator variable *a* as *cubesum[b,c]*. Neither the variables *b* and *c* nor the function *cubesum* are defined at the point of execution of this line. The function of two variables *cubesum* is defined in line 2 as the third power of the sum of its arguments. In line 3 the variable *b* is defined and assigned the value 1, and in line 4 the variable *c* is defined and assigned the value 2. The value of the variable *a* is defined after this because the values of all terms of its defining expression are defined. The last line writes the value of the expression evaluator variable *a*, which is $cubesum(b,c) = (b+c)^3 = (1+2)^3 = 27$.

New expression evaluator functions can also be defined using the interpreter by the *definefunction* command[15] with the following syntax:

```
definefunction { funcname [defblock] }
```

Evaluation of a function defined in this way includes interpretation of its definition block *defblock*. Additional calculator and interpreter functions, which can be used in the definition block, facilitate definition of how the function is evaluated. The file interpreter function *return* is used for final specification of the value which the function evaluates. Its syntax is

```
return { expression }
```

The function defined by the *definefunction* command is evaluated to the value of the *expression* given at interpretation of the *return* command in the function definition block.

The function definition is also facilitated by two pre-defined expression evaluator functions. *numargs* is evaluated to the number of actual arguments passed at function evaluation while *argument* is evaluated to values of specific arguments that are passed.

Use of the *definefunction* can be illustrated by the following example, where an expression evaluator function *Sumation*, which evaluates to the sum of its arguments, is defined[13]:

```
definefunction { Sumation
[
  ={retsum:0}
  ={indsum:1}
  while { (indsum<=numargs[ ])}
  [
    ={retsum: retsum+argument[indsum] }
    ={indsum: indsum+1 }
  ] }
  return{retsum}
] }
```

Beside evaluation of condition expressions in branching and looping interpreter commands, the expression evaluator can be used for evaluation of numerical arguments of file interpreter commands. Any numerical argument can be given, stated either in the direct form by specifying its value, or by an expression evaluator variable in the form

```
$ varname
```

or by a mathematical expression of the form

```
$ { expression }
```

The interpreter function that corresponds to the command called by such arguments, use the expression evaluators to evaluate the appropriate values that replace variables or expressions before arguments are used.

4.2.3 User Defined Variables

The shell uses a system of user defined variables (also referred to as shell variables) for data storage. Individual algorithms and other utilities usually have their own local data storage, but input and output data of built-in utilities should be transferable to or from the system of user defined variables. In this way results of any algorithm can be used in other algorithms. Since running of any algorithm or utility

as well as access to user variables is arranged through a common user interface (i.e. the command file interpreter), the necessary data transfer between different utilities is easily achieved.

The system of user-defined variables is considered as an individual module of the shell, which provides data storage and transfer services. These services are accessible through special interpreter and calculator commands for manipulating variables and through the possibility of using variables of various types as input arguments of interpreter commands. It must be noted that the calculator variables are a part of a separate system and are not treated as shell variables. Transfer between both systems is completely supported and in some cases calculator variables are used for the same purpose as the user-defined variables.

The shell variables hold objects (elements) of different types: options, counters, scalars, vectors, matrices, strings and files. Each variable can hold a multidimensional table of elements of a specific type (Figure 4.4). The number of dimensions of this table is referred to as the rank of the variable.

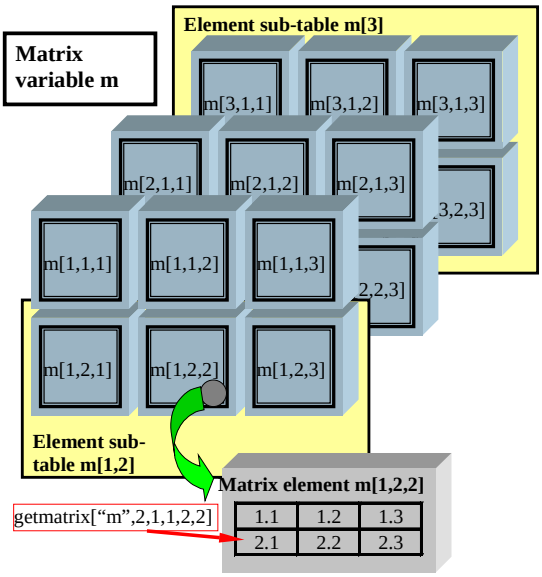

Figure 4.4: Example of a matrix variable that holds a $3\times2\times3$-dimensional table of 2 by 3 matrices.

For each type of variable there is a set of interpreter and expression evaluator functions for their manipulation, i.e. creation, initialisation, copying, moving, etc. The following example shows how to create a matrix variable as shown in Figure 4.4:

```
newmatrix { m [ 3 2 3 ] }
setmatrix { m [ 1 2 2 ]
  2 3 { { 1: 1.1 1.2 1.3 } { 2: 2.1 2.2 2.3 } }
}
```

The *newmatrix* command creates a matrix variable with a $3\times2\times3$ - dimensional table of elements, each of which is a matrix. Indices in square brackets specify dimensions of the variable element table (note that these dimensions are not related to dimensions of matrix elements of the variable). After creation, the matrix elements are not initialised and contain no data. Values of specific elements are set by the *setmatrix* command. Indices in square brackets in this case specify the matrix in the variable element table whose values are set. In the above case, the element with indices $[1,2,2]$ is set to the following 2×3 matrix:

$$\begin{bmatrix} 1.1 & 1.2 & 1.3 \\ 2.1 & 2.2 & 2.3 \end{bmatrix}. \tag{4.1}$$

If the rank of the matrix variable (which is three in this case) was zero, then a call to the *newmatrix* command would not be necessary since the *setmatrix* function creates a zero rank variable automatically if it does not yet exist.

The expression evaluator functions can be used for accessing data stored in variables. For each variable type there exist functions, which are evaluated as dimensions of variable element tables or as components of variable elements. For example, the *getmatrixdim* function evaluates a specific dimension of the variable element table. The expression

```
getmatrixdim[“m”,2]
```

evaluates to the second dimension of the element table of the matrix variable *m*, which is 2 in the case that *m* is defined as above (Figure 4.4).

The *getmatrix* function evaluates to the value of a specific component of a specific matrix element. The expression

```
getmatrix[“m”,2,1,1,2,2]
```

evaluates the component 2-1 of the element $[1,2,2]$ of the matrix variable *m*, which is 2.1 if the variable is defined as above. The first two indices specify the component (row and column number, respectively) and the last indices specify the matrix element of the variable element table.

There are several other functions for manipulation of matrix variables, and analogous functions are implemented for other types of variables. A complete list of these functions can be found in the corresponding manual[17].

Some interpreter commands can operate on whole sub-tables of variable elements. The notion of a sub-table is also illustrated in Figure 4.4. The matrix variable *m* shown in the figure contains an element table of rank 3 and of dimensions $3\times2\times3$. This table consists of two sub-tables of rank 2 and of dimensions 2×3, each of which further consists of two sub-tables of rank 1 and dimension 3, each of which contains 3 matrix elements.

Some shell variables are used for carrying specific data relevant for optimisation. Such variables are referred to as variables with pre-defined meaning or briefly pre-defined variables.

Of particular importance are those variables which are responsible for data links between user definition of the direct analysis in the *analysis* block of the command file and optimisation algorithms[18] (Figure 4.3 and the surrounding discussion). A list of these variables is shown in Table 4.1.

The pre-defined variables are a part of the user-defined variables, therefore all functions for manipulating variables are applicable to these variables. Some additional commands are designed especially for easier handling of these variables. Some general functions of the pre-defined variables operate in a slightly different way on pre-defined variables. This is especially true for creation and initialisation commands, which take into account known information regarding dimensions. Dimensions of the pre-defined variables are often related to the characteristics of the optimisation problem being solved, therefore the same dimensions can be shared with more than one variable (Table 4.2). These characteristic dimensions have a special storage space that is not a part of the variable system. Their values are however directly accessible through the interpreter and expression evaluator functions[17].

There are some other variables with pre-defined meaning[17], which support common tasks related to the solution of optimisation problems. For example, variables in Table 4.1 have equivalents with the suffix “opt” instead of “mom”, which store optimum values of the corresponding quantities so that they can be retained for further use. Vector variables *meas* and *sigma* are used for holding input data for inverse problems, namely the measurements and their estimated errors. Pre-defined file variables for holding commonly used files are also defined, i.e. *infile* for shell input file, *outfile* for shell output file, *aninfile* for simulation input file and *anoutfie* for simulation output file. There are groups of interpreter and calculator functions, which operate specifically on these variables. A set of output functions operate on *outpfile*, and a set of general interfacing functions operate on *infiile*[19].

Some functions of the interface module with the simulation programme operate on *aninfile* and *anoutfile*[22].

Table 4.1: Variables with pre-defined meaning, which are used for transfer of input and output arguments of direct analysis between user definition and the calling algorithm. The meaning of dimensions is shown in Table 4.2.

Variable name [element table dim.] (element dim.)	**Meaning**
Scalar variables	
objectivemom [] < [numobjectives] >	Value(s) of the objective function(s) at the current parameter values.
constraintmom [numconstraints]	Values of the constraint functions at the current parameter values.
Vector variables	
parammom [] (numparam)	Current values of parameters.
measmom [] (nummeas)	Current values of simulated measurements.
gradobjectivemom [] < [numobjectives] > (numparam)	Gradient of objective function(s) at the current parameter values
gradconstraintmom [numconstraints] (numparm)	Gradients of constraint functions at the current parameter values.
gradmeasmom [nummeas] (numparam)	Gradients of the simulated measurements at the current parameter values.
Matrix variables	
der2objectivemom [] < [numobjectives] > (numparam,numparam)	Second derivatives (Hessian) of the objective function(s) at the current parameter values.
der2constraintmom [numconstraints] (numparam,numparam)	Second derivatives (Hessian) of the constraint functions at the current parameter values.
der2measmom [nummeas] (numparam,numparam)	Second derivatives (Hessian) of the simulated measurements at the current parameter values.

Table 4.2: Characteristic dimensions of variables with a pre-defined meaning.

Dimension	Meaning
numparam	Number of optimization parameters
numconstraints	Number of constraint functions
Numobjectives	Number of objective functions (usually equals 1)
Nummeas	Number of measurements (applicable for inverse problems)

4.2.4 Argument Passing Conventions

The file interpreter itself has nothing to do with interpretation of command arguments. It only passes positions of command argument blocks to the corresponding functions. Each individual function (shown in the user interface area in Figure 4.3) is responsible for interpretation and treatment of its arguments. The shell provides some general rules about argument passing, which represent a non-obligatory recommendation and may be overridden by individual functions. This freedom allows implementation of interpreter commands with arguments of types and format adapted to specific tasks and not common for the shell. The shell user interface can therefore be customised to a great extent.

The shell provides a set of functions for interpretation of specific supported types of arguments. Within functions installed in the file interpreter system these functions can be used for interpretation of arguments. Shell functions for interpretation of arguments can be used as library functions and provide an implementation interface, which enables new utilities to be built into the shell in accordance with standard rules that apply for the shell. Such an implementation interface plays an important role in ensuring openness and flexibility discussed in section 4.1.2. Argument passing rules supported by the shell are briefly described below, while a complete description can be found in [15].

Multiple arguments may be separated either by spaces or by commas. Because some arguments (respectively strings that represent them) themselves contain commas and spaces, it must be unambiguous for each argument to which position it extends, which is a basic requirement for formatting conventions.

Objects of all types defined by the shell can be passed as arguments. Each type has its own formatting conventions. For example, a matrix object given by (4.1) can be given in one of the following forms:

```
1.2 3 { { 1: 1.1 1.2 1.3 } { 2: 2.1 2.2 2.3 } }

2.2 3 { { 1 1: 1.1 } { 1 2: 1.2 } { 1 3: 1.3 } { 2 1: 2.1 } { 2 2:
   2.2 } { 2 3: 2.3 }  }


3.2 3 { { 1.1 1.2 1.3 2.1 2.2 2.3 } }
```

If we just want to specify a matrix with a given number of rows and columns without specifying components, only dimensions need to be given followed by empty curly brackets, e.g.

```
    2 3 {  }
```

In commands that assign a matrix to an existing object, we can specify an arbitrary number of its components without dimensions, e.g.

```
{ { 1 1: 1.1 } { 2 2: 2.2 } { 2 3: 2.3 }  }
```

The *setmatrix* command mentioned in the previous section is an example of an interpreter command that takes a matrix argument. For this function, the last format can be used if the matrix element, which is being initialised by the *setmatrix* function, already exists and is of correct dimensions. Since in all possible formats specification of a matrix object is concluded by curly brackets (usually containing components), it is unambiguous where the argument ends and where to expect the next argument.

Other types of objects have similarly logical format conventions. String objects must be specified in double quotes if they contain spaces. Special characters that can not be represented in text files or can not be specified directly because of formatting rules, can be specified by two character sequences consisting of a backslash and specification character. For example, the newline character can be represented by the sequence “\n”.

Variable arguments are specified by variable names not included in quotes. Variables can contain more than one object of a specific type. Commands that operate on individual objects therefore do not take variable arguments. Variable arguments are typical for commands which create, copy or move a whole variable, like for example the *newmatrix* command for creation of matrix variables, mentioned in section 4.2.3.

Arguments that refer to variable elements are specified by a variable name followed by element indices in square brackets. Indices specify element position in the variable element table. Elements of zero-rank variables can be in some cases specified only by variable name, but the name followed by empty square brackets is always acceptable. Sub-tables of variable elements are specified in the same way as individual elements. For example, the $[2,3]$-th element of a matrix variable *mat1* is specified by

```
mat1 [2 3]
```

Numerical arguments can be specified by numbers, expression evaluator variables (variable name following the dollar sign) or mathematical expressions (stated in curly brackets following the dollar sign), as has been described in section 4.2.2. Indices in element specifications are also regarded as numerical arguments, therefore this rule applies. If an expression evaluator variable *a* is defined and has a value 2, the above specification of a matrix element can be written equivalently as

```
mat1 [$a ${a+1}]
```

Objects of any type can also be specified by a reference to an existing object of the same type in the shell variable system. A copy of that object is created in this case and passed as an argument. Object specification must be included in curly brackets following the hash sign, e.g.

```
# { mat1 [4 1] }
```

specifies a matrix that is a copy of the element with indices $[4, 1]$ of the matrix variable named *mat1*.

Finally, variable names in specification of variables, elements or sub-tables of elements can be replaced by a reference to an existing string element. For example, if a zero rank string variable *str* is defined and its only element is the string *"mat1"*, then the following specification of the $[2,3]$-th element of a matrix variable *mat1* is adequate:

```
# { str [ ] } mat1 [2 3]
```

4.2.5 Summary of Modules and Utilities

A brief survey of modules that provide basic functionality needed for solution of inverse and optimisation problems is given in this section. Figure 4.3 and the surrounding discussion provides a basic explanation of the importance of individual modules. A basic functionality of the shell is listed in Table 4.1.

The core of the shell is optimisation algorithms. Other utilities provide the functionality needed for the definition of the problems to which optimisation algorithms are applied. In this respect utilities that enable the definition of the direct analysis are the most important, which especially refers to interfacing with the simulation environment. Open structure of the shell enables interfacing with any simulation environment. An interface module[22] for a finite element system Elfen[30],[31] has already been implemented.

A general file interface[19] enables interfacing with any programme for which a special interface module is not implemented, through its input and output files. These modules provide a set of basic utilities for manipulating text files, such as searching for data, reading and updating data, copying data, etc.

Additional support for the definition of the direct analysis is offered by the expression evaluator. It can be used in combination with the file interpreter capabilities to specify how the quantities required by an optimisation algorithm are

derived from basic results obtained by a numerical simulation. It can be regarded in this respect that additional post-processing of results, which is not provided by the simulation environment but is needed for formation of information required by algorithms, is taken over by the shell.

Table 4.3: Principal modules of the optimisation shell inverse.

Optimisation module[18] includes optimisation algorithms and other tools (e.g. tabulating utilities, support for Monte Carlo simulations, etc.). It also includes utilities for definition of direct analysis, including organisation of data transfer between analysis definition and optimisation algorithms.
File interpreter[15] represents the shell user interface.
Flow control module includes implementation of branches and loops, a function definition utility, and some other flow control utilities.
Syntax checker[20] enables checking command file syntax before running it. Some troublesome errors such as parenthesis mismatches can be detected by this tool. Arguments are also checked for some basic interpreter commands (e.g. for flow control commands).
Debugger[20] allows step-by-step execution of commands, execution of arbitrary portions of code, checking and changing values of variables between execution, etc.
Expression evaluator (**calculator**)[16] evaluates mathematical expressions which appear in argument blocks of file interpreter commands.
Variable handling module[17] includes basic operations on variables such as creation and deleting, copying, initialisation, etc.
General file interface[19] provides a set of functions for interfacing simulation and other programmes.
Interfacing modules provide tools for interfacing specific simulation programmes, which includes execution control and data exchange functions.
Miscellaneous utilities module[21] include various auxiliary utilities, for example utilities for interaction with the operating system.

Various auxiliary utilities[21] can be used to control the solution process or provide additional support to the interfacing module. The most important are output commands, which enable the user to output any information of interest to the terminal or shell output file. Other utilities enable control of execution and CPU time and interaction with the file system (changing directories, deleting files, etc.).

The file interpreter[15] represents a user interface, which provides access to the shell utilities. Accessibility of the shell functionality through the file interpreter is the basis of flexibility, which enables the shell to be applied to a large variety of problems. The ability of installing new file interpreter commands is a basis of openness of the shell as regards the possibility of implementing new tools that interact with the existing functionality. An open library provides an implementation interface for building in new tools in accordance with the shell concepts. A part of this library consists for example of functions for interpretation of arguments of file interpreter commands.

Instruments for checking the shell execution and correctness of user definition of the problem are a significant part of the shell. A user interface implemented as a file interpreter imposes a high level of flexibility on one hand, but on the other hand definition of problems with such an interface is prone to errors. This is especially true when a lack of high level commands is experienced and must be overcome by using programming capabilities of the shell user interface to a large extent.

Table 4.4: A list of debugger commands with brief descriptions.

? prints a short help. **q** finishes the debugging process. **s** executes the next file interpreter's command. **S** executes the next file interpreter's command; commands that execute code blocks are executed as single commands. **n** *num*. Executes the next *num* commands. **N** *num* executes the next *num* commands; functions that contain code blocks are executed as single commands. **x** *num* executes the code until *num* levels lower lever of execution is reached. Default value for *num* is 1. **c** executes the code until the next active **break** command is reached. **ab** *id* activates all breaks with the identification number *id* ("*" means all identification numbers). **sb** *id* suspends all breaks with the identification number *id* ("*" means all identification numbers). **pb** prints information about active breaks. **tb** *id* prints status of breaks with identification number *id*. **v** *shift* prints a segment of code around the current viewing position shifted for *shift* lines. **vr** *shift* prints a segment of code around the line of interpretation shifted for *shift* lines. **va** *linenum* prints a segment of code in the interpreted file around the line *linenum*. **nv** *num1 num2* sets the number of printed lines before and after the centerline when the code is viewed.	**e** *expr* evaluates the expression *expr* by the expression evaluator. If *expr* is not specified the user can input expression in several lines, ending with an empty line. **w** *expr* adds expression e*xpr* to the watch table. Without the argument, values of all expressions in the watch table are printed. **dw** *num* removes the expression with serial number *num* from the watch table. **aw** *switch* with *switch* equal to zero turns automatic watching off; otherwise it turns it on. **pw** prints all expressions in the watch table. **r** *comblock* interprets *comblock* by the file interpreter. If *comblock* is not specified the user can input commands in several lines, ending with an empty line. **rd** *comblock* does the same as ***r***, except that the code is also debugged. **rf** *filename* sets the name of the file into which the user's commands will be written, to *filename*. **Breaks** are set in the command file by function **break**, whose argument (optional) is break identification number, e.g. `break { 3 }`

The shell contains two tools, which facilitate location of errors[20]. The syntax checker detects some common and obvious syntax errors such as misspelling of commands and mismatched brackets. It can be applied to check the command file before the shell is run. The debugger (Table 4.4) enables tracing an execution of the shell. It enables step by step interpretation of the command file between which the state of the calculator and shell variables can be inspected or changed. The debugger is a useful tool not only for detection of logical errors in the command file, but also

for detection of unexpected results of various built-in tools or stand-alone programs which are employed in problem solution.

4.2.6 A Simple Example

A simple example[13] is shown in order to highlight the shell function discussed in previous sections. The example shows how the following problem can be solved by the shell:

$$\begin{aligned} &\underset{x,y}{\textit{minimise}} && x^2 + y^4 \\ &\textit{subject to} && y \geq (x-3)^6 \wedge y \geq 17 - x^2 \end{aligned} \tag{4.2}$$

The objective and the two inequality constraint functions are

$$\begin{aligned} f(x,y) &= x^2 + y^4 \\ c_1(x,y) &= y - (x-3)^6 \\ c_2(x,y) &= x^2 + y - 17 \end{aligned} \tag{4.3}$$

The command file which makes the shell solve this problem is the following:

```
1. setfile{outfile quick.ct}

2. *{ Objective and constraint functions: }
3. ${f[x,y]: x^2+y^4 }
4. ${g1[x,y]: -((x-3)^6-y) }
5. ${g2[x,y]: -(17-x^2-y) }
6. *{ Objective function derivatives: }
7. ${dfdx[x,y]: 2*x }
8. ${dfdy[x,y]: 4*y^3 }
9. *{ First constraint function derivatives: }
10.  ${dg1dx[x,y]: -(6*(x-3)^5) }
11.  ${dg1dy[x,y]: 1 }
12.  *{ Second constraint function derivatives: }
13.  ${dg2dx[x,y]: 2*x }
14.  ${dg2dy[x,y]: 1 }

15.  setvector{parammom 2 {} }
16.  newscalar{objectivemom}
17.  newscalar{constraintmom[2]}

18.  analysis
19.  {
```

```
20.     ={x:getvector["parammom",1]}
21.     ={y:getvector["parammom",2]}
22.     setscalar{objectivemom ${f[x,y]} }
23.     setvector{ gradobjectivemom
24.       {  ${dfdx[x,y]}  ${dfdy[x,y]} }
25.     }
26.     setscalar{constraintmom[1] ${g1[x,y]} }
27.     setvector{ gradconstraintmom[1]
28.       {  ${dg1dx[x,y]} ${dg1dy[x,y]}  }
29.     }
30.     setscalar{constraintmom[2] ${g2[x,y]} }
31.     setvector{ gradconstraintmom[2]
32.       {  ${dg2dx[x,y]} ${dg2dy[x,y]}  }
33.     }
34.   }

35.   setvector{parammom { 0 0 } }
36.   analyse{}

37.   optfsqp0{ 1 2 0 0 0 0.00001 0.00001 300 1
38.     { 2 { 15 -3 } }
39.     { 2 {} }
40.     { 2 {} }
41.   }
```

Analysis block

The *setfile* command in line 1 creates the shell output file *outfile* where functions will write their reports and error reports, and connects this file with the physical file named “quick.ct”.

Lines 3 to 14 contain some preliminary definitions of new expression evaluator functions, which will be used later in the analysis block. These are the objective (line 3) and both constraint functions (lines 4 and 5), derivatives of the objective function with respect to the first (line 7) and the second (line 8) parameter, and derivatives of the first (lines 10 and 11) and the second (lines 13 and 14) constraint function with respect to both parameters.

In lines 15 to 16 we create variables with pre-defined meaning *parammom, objectivemom* and *constraintmom*. The aim of this is merely to specify the relevant characteristic dimensions of the problem. These are stored in internal variables of the shell and are used when creating pre-defined variables whose dimensions are by definition equal to these characteristic dimensions. By creating vector *parammom*, the number of parameters *numparam* is defined, by creating scalar *objectivemom* the number of objective functions *numobjectives* is defined and by creating scalar variable *constraintmom* the number of constraints *numconstraints* is defined. No values are assigned to these variables. The same effect as creating *parammom* would have been obtained for example by creating *paramopt*, and creating vector *gradconstraintmom* could replace both creating vector *parammom* and scalar

constraintmom, since both *numconstraints* and *numparam* are relevant for this variable.

Lines 20 to 33 form the analysis block, which represents the definition of the direct analysis and is interpreted at every analysis run. This block specifies how relevant quantities such as the objective and constraint functions and their derivatives are evaluated at a specific set of optimization parameters.

In lines 20 and 21 the current values of parameters are stored in expression evaluator variables x and y. These values are obtained during optimisation from vector *parammom* where they are put by the general analysis function, called by the algorithm that requests execution of a direct analysis.

In lines 22 to 33 the relevant quantities are evaluated and stored into the appropriate pre-defined variables where the calling algorithm can obtain them. The value of the objective function is stored into scalar *objectivemom* (line 22), its gradient is stored into vector *gradobjectivemom* (lines 23 to 25), values of the constraint functions are stored into scalar variable *constraintmom* (lines 26 and 30), and their gradients to vector variable *gradconstraintmom* (lines 27 to 29 and 31 to 33) Auxiliary functions, which were defined in lines 3 to 14 of the initialisation part are used, called with the current parameters stored in calculator variables x and y (lines 20 and 21). In more realistic cases this part would include running some numerical simulation at the current parameters, the necessary interfacing with the simulation *programme* (for updating simulation input and reading results) and possibly some housekeeping for deriving final values from the simulation results.

A test analysis at parameters $[0,0]^T$ is run in line 36 by the *analyse* command. This command takes parameter values from the pre-defined vector *parammom*; which is set in line 35.

Finally, the problem is solved using the command *fsqp0*, which runs the feasible sequential quadratic programming optimization algorithm (lines 37 to 41). This function requires nine numerical arguments, namely the number of objective functions, the number of non-linear inequality constraints, the number of linear inequality constraints, the number of non-linear equality constraints, the number of linear equality constraints, the final norm requirement for the Newton direction, the maximum allowed violation of nonlinear equality constraints at an optimal point, the the maximum number of iterations, information on whether gradients are provided or not, and three vector arguments, namely the initial guess and lower and upper parameter bounds.

Suppose that the above command file has been saved as “quick.cm” and that the shell programme is named “inverse”. We can run the shell by

```
inverse quick.cm
```

which solves the problem (4.2). The report including final results can then be checked in the file “quick.ct”.

4.3 Selected Implementation Issues

4.3.1 Programming Language and Style

The present section discusses some basic implementation issues, which influence the shell function, effectiveness and economy of its development.

As regards computer programming, the same operations can usually be implemented in a number of different ways. It is the function of a programme that matters the most, however there can be big differences in programme efficiency and final development cost between different implementations of the same system. Awkward implementation usually results in unexpected bugs and unnecessary problems with inefficiency, which can never be completely disclosed at the testing stage.

Careless programming frequently results in a rigid system, which serves its purpose well, but it is hard to introduce changes and add new functionality. For complex systems it is impossible to predict all requirements that can possibly arise from application needs. System design can therefore not be completely planned in advance and it is particularly important that it is easy to introduce changes in the system and expand its functionality.

Since the implementation style has a strong influence on the overall quality of the system and economy of its development, it deserves special attention. The desired system properties often impose conflicting implementation demands, therefore it is difficult to set generally applicable implementation rules. What is appropriate depends on what should be achieved.

A common conflicting situation in programming is induced when the need for maximum efficiency arises simultaneously with the requirement to make the system open and accessible to different developers. The last demand is achieved if the system is logically structured and its function implemented through small closed units. Such implementation leads to a certain overhead of function calls and

allocation of data, which could otherwise be reused, which to some extent affects the programme efficiency. The loss of efficiency is in some cases negligible and can be sacrificed without hesitation. In other cases a compromise must be accepted after consideration of solutions that are acceptable in view of openness and still have a lesser impact on efficiency.

Knowing programming rules means being aware of the effects of different programming approaches. Implementation of complex systems is to a large extent the art of making good compromises. This requires a significant amount of planning, sometimes in a very abstract sense because situations in which the system might be exposed can be foreseen only to a limited extent.

The optimisation shell Inverse is programmed in ANSI C[24]-[26]. This is an extension of the traditional Ritchie’s C[23]. It is implemented in more or less an invariant way on all modern platforms and is currently one of the most portable programming languages[1]. C is a terse and logical language with a small set of keywords and powerful set of operators which support low level access to computer capabilities. It still provides all facilities typical for high level languages, such as structured data types.

Unlike many other programming languages C does not impose unnecessary restrictions which do not arise from the computer architecture. Many of these restrictions imposed by other languages are in some cases extremely difficult to overcome. C completely supports dynamic memory allocation. Arrays can have variable length. This is for example not the case in Pascal, which makes it difficult to handle matrix operations in a modular way. Function addresses are treated in a natural way in C and can be assigned to variables. This enables the fully dynamic treatment of function calls, which can in some languages be achieved only by passing function addresses through function arguments (as in FORTRAN) or not at all (as in Pascal). A favourable feature for programmers who wish to have a complete understanding of code function is that function arguments are always passed by value. Effects equivalent to passing by reference in some other languages are achieved by passing a pointer to a variable instead of the variable itself. In many other languages this is done implicitly, so that the language rules hide to a great extent what is actually happening. Having complete insight in the code function therefore requires a deeper knowledge of the programming language, which is not true for C. A similar example is pointer arithmetic, which closely follows native computer logic.

An often heard argument against the use of C in numerical applications is that programmes written in C are slower than for example programmes written in FORTRAN. The author of this text is not aware of any theoretical arguments or

[1] Experience show that in practice there are minor differences between various implementations of ANSI C. However, the number of particularities is small and they can be kept under control.

comparable tests which would support such statements[1]. It is however possible to use C for specific tasks in an inefficient way, which is less usual in other programming languages because of their restrictions. This simply means that it is less likely in some languages that an unskilled programmer would implement specific tasks in an inefficient way. A typical example is unnecessary overuse of dereference inside iterative parts of the code, for example in matrix operations.

C does not directly support object oriented programming in a way as C++ does[28]. Object oriented programming was introduced to support more human-like formulation of ideas in programming languages and a more open structure of programmed modules. An especially strong feature of object oriented programming is that commonality between different ideas can be made explicit by using inheritance. Therefore it is possible to relate similar ideas in a natural way, which makes the code clearer. There are some examples in the shell where advantages of object oriented programming could be used. Such situations are however not typical and use of plain C does probably not represent a great loss in terms of development efficiency.

The shell consists of a number of hierarchically arranged modules. Modules represent closed units, which provide a given kind of functionality and make it available to other parts of the programme. The design of a module includes definition of related data types and a complete set of operations that can be performed on these types. This leads to a given functionality, which represents realisation of some a given idea in the programming sense.

Programming in a modular way allows concentration on one type of problems at a time. Solutions are provided independently of other problems. An important gain of such an approach is that functionality can be tested independently for small and well defined units, which significantly reduces testing complexity. Modular programming in C significantly reduces the possibility of memory handling errors, which are among the most problematic and common errors in C. Such errors can be avoided if all memory allocation and deallocation is performed by functions provided by the appropriate modules, which are designed so that they exclude the possibility of common errors such as accessing memory through bad pointers, releasing the same pointer several times, etc.

4.3.1.1 Example: the Stack Module

A typical example is the stack module. This module introduces stacks of objects and provides a complete set of operations on such data structures. Objects on

[1] It is possible that this common opinion was influenced by inefficiency of the C compilers at the early stage of the language development.

stacks are represented by their pointers, which are of type *void **, so that any type of objects can be stored on such stacks.

The module does actually not provide only the push and pop utility, which are typical for stacks in a common sense, but also insertion of an object to a given position, deletion of an object on a given position, sorting of objects according to a specified criterion, etc.

The stack type is defined in the following way:

```
typedef struct{
  int n,r,ex;
   void **s;
} _stack;

typedef _stack *stack;
```

Type *stack* is defined as a pointer to a structure of the type *_stack*. It is common in the shell that objects are presented by pointers of a given type. Another commonly accepted rule is that all pointers are initialised to NULL. This pre-defined value (which is essentially 0 on all modern systems) is used as an unambiguous indication that the specific object is not allocated.

Structure member (or field) *s* is the array of pointers, which holds objects that are on the stack. The structure also contains three integers. *n* is the number of objects that are on the stack, *r* is a number of allocated pointers in the array, and *ex* is an auxiliary field which defines the excess of memory allocated for the table *s* when it runs short of space to hold objects that are added to the stack. The possibility of allocating more memory that is currently needed allows that the array *s* is not reallocated every time a new objects is added to the stack.

All possible operations on stacks are provided by the functions which are defined in the module. These functions take care that the *stack* type is always used in a prescribed way, which excludes the possibility of errors. Internal rules that ensure proper function are hidden to the user of the module, which will be shown on some specific functions provided by the module. The user must only be aware of module functionality and some general rules for using the module.

The basic operations on any type of complex objects are creation and deletion. Stack objects are created by function *newstack*, which is declared as

```
stack newstack (int excess);
```

The function creates a new stack object and returns a pointer to it. It takes an integer argument, which specifies the value of the field *ex* of the created stack. The

function also allocates the table of pointers *s* so that it can hold *ex* pointers. *r* is set to *excess* and *n* to 0.

Deletion of a stack object is achieved by the function *deletestack*, which is declared as

```
void dispstack(stack *st);
```

The function requires a pointer to stack, which must be the address of the stack to be deleted. This enables the function to set the value of the deleted stack to NULL after performing other necessary operations on it (note that arguments in C are passed by value). In this way other functions that would operate on the same stack can detect that the stack no longer exists. The function *dispstack* first releases the memory used by the field *s* (if it is allocated). Then it releases the memory used by the structure itself. This memory was dynamically allocated by the *newstack* function and can be used by other objects after it is released. The *dispstack* function checks if the memory to be released is allocated (this is detected through pre-defined value NULL, which is generally used for pointers that are not initialised). This excludes the possibility of trying to release the same pointer twice, which is an error that on most systems causes abnormal programme termination. The user of the stack module does not need to take care of the possibility of such errors, because all necessary mechanisms are built into functions provided by the stack module. The only concern is that all pointers to objects are initialised to NULL before they are used. This is a general rule of safe C programming and is strictly obeyed in shell development.

The *dispstack* functions does not affect objects that are on the stack. If pointers to these objects reside only on the stack that is being deleted, these objects must be deleted prior to deletion of the stack, otherwise the memory occupied by them is not accessible any more and is a permanent waste untill the end of programme execution. Deletion of objects on the stack can be performed explicitly by deleting objects one by one. The module also provides a function for deletion of all objects at a time, which is declared as

```
void dispstackvalspec(stack st,void (*disp) (void **));
```

This function requires two arguments. The first argument is the stack (a pointer) whose elements will be deleted. The second argument (*disp*) is the function which is used for deletion of each individual object. The function deletes all objects on the stack by calling function *disp* with their addresses as arguments. *disp* must be a function that is equivalent to *dispstack* for the type of objects that reside on the stack. It is also supposed that the function sets the pointer value to NULL after the object pointed to by that pointer is deleted.

There also exists the function *dispstackallspec* which does deletion of objects contained in a stack and the stack itself. Its declaration differs from declaration of

dispstackvalspec in that the address of the stack must be passed as the first argument rather than the stack itself, because this is required for deletion of the stack. The declaration of the function is the following:

```
void dispstackallspec(stack *st,void (*disp) (void **));
```

It is appropriate to give an implementation note at this stage. The *dispstackallspec* function could be implemented in the following very simple way:

```
void dispstackallspec(stack *st,void (*disp) (void **))
{
  dispstackvalspec(*st,disp);
  dispstack(st);
}
```

Functions *dispstackvalspec* and *dispstack* perform all that is necessary for the function *dispstackallspec*. However, by implementing the function this way we have two additional function calls inside the function. From the point of view of efficiency it is better that bodies of both functions are explicitly repeated within the function, so that these two calls are avoided while other code that is executed remains the same. This will make the appearance of the function more complex, which is not important since the user of the stack module will typically not interfere with the function definition but only with its declaration. Also the compiled programme that uses the module will be of a slightly larger size, but this is not problematic because the code of the function body appears only in one place within the programme, while the function will be typically called many times. The effect on efficiency is in this case more important than the effect on the code size.

Basic operations on stacks are push and pop. Push adds an objects on the top of the stack, while pop takes an object from the top. Their declarations are

```
void pushstack(stack st, void *el);
```

and

```
void *popstack(stack st);
```

Both functions require the stack on which the operation is performed as the first argument. The second argument of the *push* function is the object (a pointer), which is pushed to the stack. The function adds this pointer to the table *s* of the stack after the last object on it and increments the field *n* which holds the number of occupied places. If the table of pointers *s* does not contain enough space to hold a new element, it is reallocated. Existing objects that are already on the stack are kept in the new table. The number of elements for which *s* is allocated is always written in the field *r*, which excludes the possibility of mistakes. Whenever the table is reallocated or deleted by any function of the module, the *r* field is updated.

Function *popstack* picks the last object on the stack and returns it as a pointer of undefined type (void *). This pointer can be assigned to any variable which is of the same type as the object obtained from the stack. It is the user’s responsibility to ensure that the types match. The type agreement could be more easily ensured in object oriented languages such as C++, in a sense that errors regarding type compatibility would be detected during compilation time. In practice this has not shown to be a serious problem, because stack objects are usually used in a limited scope where it is not hard to mix up pointer types.

The *popstack* function also decrements the value of the *n* field of the stack. It does not set the value of the last pointer in the array *s* to NULL, because since *n* is reduced, that pointer is out of the range and can never be accessed. When an object is picked from the stack, the array of pointers is larger than necessary. If the difference between the number of pointers which the array can hold (indicated by the field *r*) and the actual number of pointers on the stack (indicated by the field *n*) is larger than the value of the field *ex, s* is reallocated so that its physical size matches the number of objects which it holds.

Beside the above mentioned functions, the stack module contains many other functions, which allow the user of the module to perform the necessary operations. Among important functions provided are searching for an object with certain properties and sorting of objects on the stack according to a provided comparison operator.

The aim of this brief description of a module is to show the role of such closed modules in construction of the programme. A module consists of data types and a full set of operations on these types, which together represent implementation of some idea. In the case of the stack module the underlying idea is abstract and very general. It implies that multiple objects of a given type can be arranged in a stack, so that objects can be taken from the top or added to the top of the stack, sorted, searched for, etc.

The module hides all implementation details that are not relevant for use of the module. In other words, its user does not need to know much more about the module than what it is used for and how it is used. For a programmer who wants to use stack functionality it is not important what the structure of the stack object is. It is important for example that there exists a function for pushing a new object to a stack, that the objects on the stack can be sorted and that searching for objects on a sorted stack is much quicker than searching on unsorted stacks. The user must be aware of the underlying ideas, but on an abstract level which has nothing to do with implementation inside the module. Implementation of the module functions can be changed (e.g. in order to improve efficiency or eliminate bugs) without changing function declarations, which represent the implementation interface and the only interaction point of the user with the module. Introducing changes inside a well

designed module therefore does not affect any portion of code where this module is used. The same reasoning as for function applies to data types, which can be extended without affecting any portion of the code outside the module. The only consequence of such changes is that the programme must be recompiled.

The advantage of such closed modules is also that their function can be tested in a very limited scope, which makes it easier to find bugs before the module is actually used. Modules prevent unnecessary code replication. When similar ideas arise on different places, the same implementation is used everywhere. This implies also that bugs are likely to show on more than one place, which makes their detection easier. When for example a bug in the stack module is detected through errors in a part of the code where the idea of stacks is used, the bug is eliminated once (in the module) and this corrects the function of any portion of code where the same idea is used. This can be especially beneficial for bugs which show in rare and random occasions.

The idea of stacks as described above is very basic. The stack module has therefore a low rank in programme hierarchy and is used in many other modules. For example, stacks are used to hold shell variables, calculator variables, operators and functions, interpreter functions, etc. Stacks are often used for holding multiple input or output arguments of the same type, but variable number. For example, a function that finds all occurrences of a given string in a file returns string positions on a stack.

In many modules the idea of stacks arises within a broader context, which form the basis of the module. More complex derived data types therefore include stack objects as fields in their structure. All operations provided by the stack module can be performed on these fields, which means that an existing idea already implemented in a closed module is incorporated as a part of a broader idea. The new type inherits properties of stacks in a way, which resembles some concepts of object oriented programming.

The concept of inheritance can also be observed in a reverse way. We can have stacks of objects of different types, e.g. a stack of matrices or a stack of vectors. Various objects are implemented through completely different structures and sets of functions that can operate on them. Different objects are also deleted in different ways, and when we want to delete a whole stack of objects, this procedure must inherit specifics of deletion of the objects of a given data type. The mechanism of deletion of a stack of objects has already been described above in connection with the function *dispstackvalspec* and *dispstackallspec*.

Figure 4.5 shows the organisation of the main shell modules.

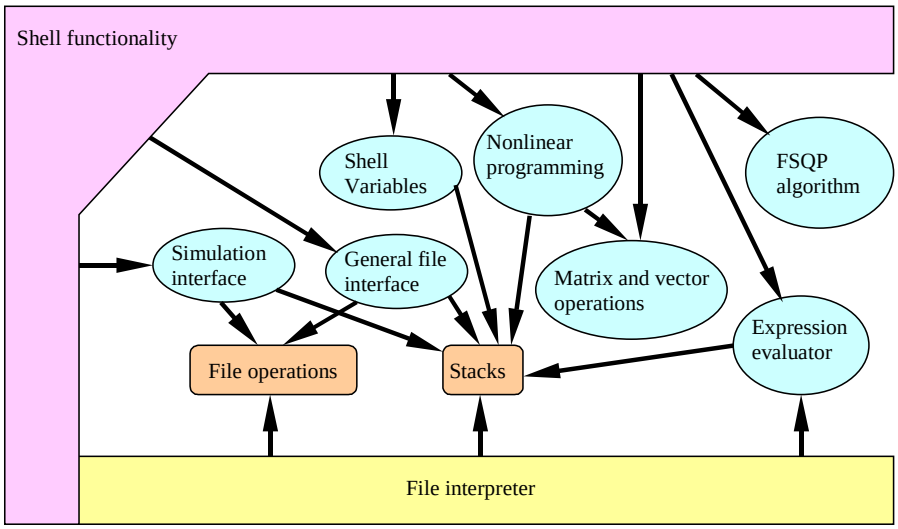

Figure 4.5: Organisation of shell modules. Arrows show dependencies. The scheme is not complete.

4.3.2 File Interpreter and Open Library

All shell functionality is accessed through file interpreter commands. The file interpreter therefore deserves special attention. Core data type of the file interpreter is the *_ficom* type, which is fundamentally defined as follows:

```
typedef struct{

  . . . /* syntax definition */

  char stopint;

  . . . /* support to flow control */

  stack functions;
  FILE *fp;
  char *filename;
  long from,to;

  . . . /* support to user defined functions */

  int which;
  fifunction func;
  long pos,begin,end,argpos;

  . . . /* temporary files */
```

```
    FILE *in;
    FILE *out;
    ssyst syst;

    . . . /* support to definition of calculator functins */

    . . . /* support to tracing of calling sequence */

    char debug, check;
    licom lint;

    . . . /* support to debugging */

} _ficom;
```

Only the most important fields are written. Comments indicate the missing groups of fields. File interpreter is an object of type *ficom*, which is a pointer to the above structure and is defined as

```
typedef _ficom *ficom;
```

Interpretation is performed by the function fileinterpret, which is declared as

```
void fileinterpret(ficom com);
```

Its only argument is an object of the type *ficom*, which actually represents the interpretation and is also referred to as the interpretation object. More than one interpretation can be independently performed within a programme, provided that there is more than one interpretation object. The corresponding object must be initialised before the interpretation begins. Memory for its structure must be allocated and the file name (field *filename*) and scope of interpretation (fields *from* and *to*) must be set.

An object of the type *ficom* holds all data relevant for interpretation. Beside the state of interpretation (e.g. current position of interpretation and information about the executed command) it also holds data which instructs the interpreter how to perform interpretation (e.g. in debugging mode or not). Function *fileinterpret*, which performs the interpretation, therefore does not need any local variables. This function is sometimes referred to simply as the interpreter, although this term is in its broadest meaning used for the whole interpreter module.

The field *stopint* tells the interpreter to exit interpretation if the field value is different from zero. This field is set by some functions or automatically if the interpreter hits the end of the interpreted code.

The field *fp* is the file pointer of the interpreted file. The interpreter and functions called by the interpreter accesses the interpreted file through this pointer.

The name of the interpreted file is stored in the field *filename*. It is used by the interpreter to open the file if it is not yet open. It is also used by some other functions, for example by those which report errors.

The interpreter can be used for interpretation of a part of a command file. Fields *from* and *to* hold the beginning and the end of the block which should be interpreted (zero values indicate that the whole file should be interpreted).

Interpreter commands are installed on the field *functions*, which is a stack. Objects on this stack are of the type *fifunction*. Such an object holds a command name and address of the function that corresponds to the command. When the interpretation object is initialised, its own commands such as looping and branching commands are installed on this stack. The shell installs additional commands through which its functionality can be accessed before it runs the interpreter. The *fifunction* type is defined in the following way:

```
typedef struct{
  char *name;
  void (*action) (ficom);
  void (*checkaction) (ficom);
} _fifunction;

typedef _fifunction *fifunction;
```

Field *name* is the name of the installed interpreter command, *action* is the function that corresponds to the command, and *checkaction* is the function that checks the syntax of command arguments. The last function can be executed only when the syntax checker is run.

Fields *which*, *func*, *pos*, *begin*, *end* and *argpos* are set by the interpreter and contain information about the command that is currently being interpreted. *which* is the position of the interpreted command on the stack *functions*, *func* is the corresponding object on this stack, and other fields are positions in the interpreted file. *pos* is the position of the command, *begin* and *end* define the position of the command argument block and *argpos* is an auxiliary field used by functions which correspond to interpreter commands. These functions use the field at interpretation of command arguments. *argpos* is set to the same value as *begin* by the interpreter.

While interpretation takes place, the interpreter searches for commands in the command file. When another command is found, the interpreter sets the field *pos* to its position and finds the corresponding object (i.e. the object with the command name) on the stack *functions*. Such an object represents the definition of the command and contains the address of the function that corresponds to the command (see the declaration of *fifunction* above). The interpreter sets the field *which* to the position of the object on the stack and the field *func* to the object itself, or reports an error if the corresponding object does not exist. It then finds the position of the

command argument block (which is enclosed in curly brackets) and sets the fields *begin*, *end* and *argpos* to the appropriate values. Finally it calls the function, which corresponds to the command and whose address can be accessed through the field *func* (i.e. *func->action*). This function takes the interpretation object as argument. This gives the function access to relevant information concerning the interpretation, especially information about command argument block, which is needed for interpretation of arguments.

The interpreter module provides some basic command such as commands for controlling execution flow and some input and output commands. Fields *in* and *out* hold input and output files of the interpreter. It is not necessary that these files are defined during interpretation. The shell installs its output file to the field *out* when that file becomes defined (which is on user command in the command file). This way the output commands which are defined in the interpreter module use the same file as the commands which are additionally installed by the shell.

The expression evaluator offers a similar example. This module is similar to the interpreter module in that several independent expression evaluators (with their own private set of user defined variables and functions) can exist in a programme at the same time. This is however not the case in the optimisation shell because there is no need to have more evaluators. Some basic file interpreter commands need the expression evaluator for evaluation of branching and looping conditions. It is represented by the field *syst*. The commands that need the expression evaluator access its function through this field. It is of the type *ssyst*, which has a similar meaning for an expression evaluator as the type *ficom* has for an interpreter. The optimisation shell initialises the expression evaluator and installs it in the file interpreter before it starts interpretation of the command file. Other functions of the shell that use the expression evaluator therefore use the same object as the file interpreter.

Initialisation of the expression evaluator prior to file interpretation is not compulsory. If none of the commands that need the expression evaluator is ever used, then the interpreter can run without it. Such a situation can actually occur if an interpreter is used for some specific purpose where the expression evaluator is not needed. This illustrates the flexibility of modules such as the file interpreter. The expression evaluator is a complex system that needs substantial memory in order to function, therefore it is beneficial if the interpreter may not use it when this is not necessary.

Fields *check* and *debug* hold instructions for the file interpreter. If *check* is nonzero, then the command file is just checked for syntax errors rather than interpreted. The interpreter can only find errors which concern its function and rules. This excludes errors in command arguments since the interpreter itself has nothing to do with interpretation of arguments. The interpreter allows functions to be installed that check argument syntax for commands that are installed on its system. These

functions are specified in the *checkaction* field of the object of the type *fifunction* (see its declaration above). Such objects are a representation of installed command on the stack field *functions* of the file interpreter object. If for a specific command the field *checkaction* is not NULL then the interpreter runs this function to check command arguments. The only argument of the function is the interpreter object itself, through which the checking function can find all necessary data, including the position of the argument block and the file pointer through which the command file is accessed.

If the *debug* field is nonzero, the interpretation is performed in the debug mode. A number of control parameters are used for telling the interpreter how to function, e.g. how many commands to interpret at a time or how many interpretation levels to exit. After interpretation of a specified portion of the code, control is passed to the user. A line interpreter is run in which the user inputs instructions for the interpreters through a command line. Basic debugger commands that can be used in this place are summarised in Table 4.4. Functions that carry out debugger commands are installed on the line interpreter system. Some of these functions give instructions to the interpreter through the appropriate fields on the interpreter objects. Such instructions are, for example, that only one command or a group of commands must be interpreted, or that a certain number of interpretation levels must be left. Other functions perform concrete actions, for example change the definition of the expression evaluator variables and functions or print variable values. In the debugger, the user can also run arbitrary interpreter commands. In debugger it is therefore possible to check directly the effect of changes in the command file on function of the shell.

Shell functions that correspond to interpreter commands usually just extract arguments and call other functions to perform algorithms and other tasks. Such a two stage arrangement is evident from Figure 4.3. Its advantage is that the shell side of the implementation is separated from the module which is a source of a specific kind of functionality. Different modules can therefore be developed independently. The functions that correspond to interpreter commands take care of proper incorporation of functionality provided in the shell system. The most basic task in this respect is data exchange between the shell and module functions. This is done through arguments of interpreter commands in accordance with argument passing conventions, which were described in section 4.2.4.

Argument passing mechanisms are facilitated by a set of shell functions for interpretation of command arguments and for interaction with the shell variable

system. These functions are a part of the shell open library[1], which represents an implementation interface for incorporation of any kind of functionality within the shell.

Functions of the shell open library are mostly used for implementation of interpreter functions, which correspond to interpreter commands and provide interface between the shell and the incorporated modules (Figure 4.6). Library functions hide unnecessary implementation details. They also provide a certain level of invariability with respect to changes in the shell structure and especially with respect to changes in incorporated modules. This implies that the shell itself can be treated as a module when new functionality is incorporated.

Figure 4.6 schematically shows the operation of an algorithm incorporated in the shell. A part of this scheme can be recognized in Figure 4.3. Relations between the incorporated algorithm, shell interpreter, interpreter function that provides an interface between the shell and the algorithm, and library functions which facilitate implementation of this interpreter function, are shown in more detail.

The implementation interface facilitates incorporation of new utilities in compliance with the shell conventions, but does not impose these rules a priori. Designers of new modules are given freedom to introduce individual rules for use of specific utilities. Interaction with the shell structure, function and philosophy is possible on different levels. For example, it is not necessary to use shell functions for interpretation of command arguments of specific types. Low level library functions enable more basic interaction with the file interpreter, such as direct access to the command file and position of the argument block. For specific commands individual rules can be set for interpretation of their arguments. Such low level interaction enables the introduction of additional concepts in the shell if they are necessary for a given functionality. For example, a new data type can be introduced together with complete support as offered for existing data types. Rules for passing objects of that type through command arguments can be imposed through functions that interpret arguments of that type and are added to the shell library.

[1] The term open library is used because functions in this library are designed for broader use. Many functions of modules which constitute the shell have global linkage, but only functions of the open library are designed for use outside the shell development team. These functions are designed with special emphasis on simplicity of use and invariability with respect to changes in the structure of the shell and its constitutive modules.

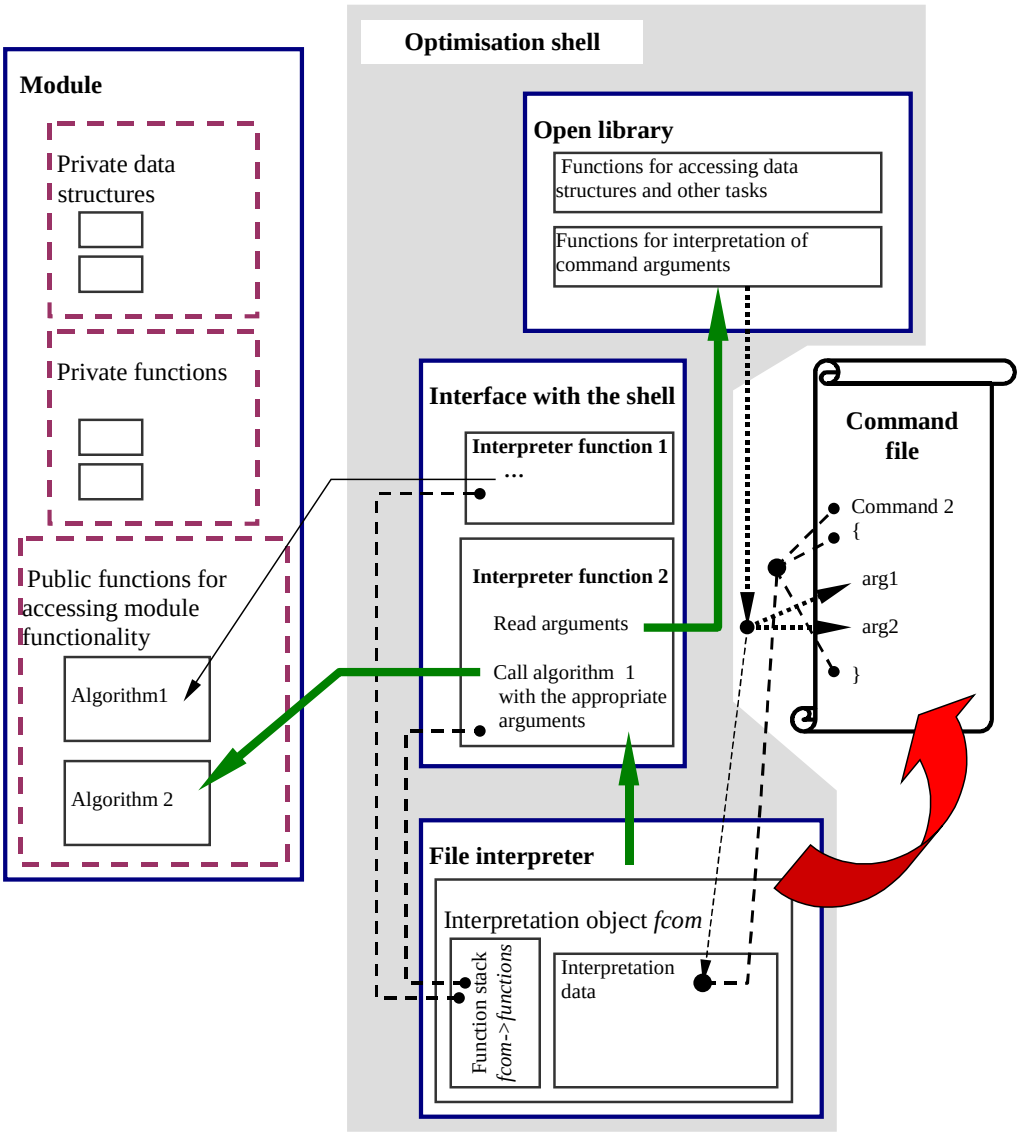

Figure 4.6: Incorporation of module functionality in the shell. Dashed lines show data connections. Continuous arrows indicate calling sequences.

The design of the implementation interface with the properties described requires some effort, especially careful planning, which sometimes includes formulation of ideas in a rather abstract way. The effort required for ensuring openness and flexibility can be justified if the shell is viewed as a general system designed for use and development by a broader community. Application to practical problems gives rise to many subproblems of a heterogeneous nature. Implementation of appropriate optimisation tools therefore by far exceeds merely the scope of implementation of effective optimisation algorithms. It requires an integrated

approach to the development of an optimisation system, which facilitates incorporation of expertise from different fields. Support to distributed development with sufficient implementation freedom and ability of introducing new concepts in the shell is essential from this point of view.

4.3.3 Incorporation of Optimisation Algorithms

Optimisation algorithms are the principal constituent of the shell functionality. Their incorporation in the shell follows similar rules as incorporation of other utilities, which was described in the final part of the previous section. There are some specifics, some of which are supported by functions in the shell open library, which are used specifically for incorporation of optimisation algorithms.

Each incorporated optimisation algorithm has a corresponding file interpreter function, which is installed on the function stack (field *functions*) of the interpretation object at the initialisation of the optimisation shell. Installation is performed by calling the open library function *instfintfunc* in a special portion of code, which is compiled and linked with the shell. Function *instfintfunc* takes the name of the installed file interpreter command and the address of the installed file interpreter function as arguments.

The interpreter function is an interface between the shell and the incorporated algorithm. This function is executed by the shell interpreter whenever it encounters the command installed together with the function (Figure 4.6). It reads command arguments and passes them to the optimisation algorithm which it calls. Reading of command arguments is performed by the appropriate open library functions.

The interpreter functions that correspond to optimisation algorithms normally provide a few additional things. It is a shell convention that the results of an optimisation algorithm are stored in specific pre-defined variables[1] (section 4.2.3). For example, optimal parameters are stored in the vector variable *paramopt* and optimal value of the objective function is stored in the vector variable *objectiveopt*. These values are normally output argument of the function which represents a given optimisation algorithm. It is the job of the appropriate file interpreter function to store these values in the appropriate places. This is performed by using the shell open library functions, which are designed for setting specific pre-defined variables.

Support to the above mentioned convention provides an evident example of what is needed for incorporation of a given functionality in the shell. The open

[1] This is a non-obligatory rule, which makes use of algorithm results by other utilities possible. A general rule is that if an algorithm does not follow this convention, this must be indicated in the appropriate manual.

library should in principle provide all necessary utilities, therefore a specific knowledge regarding the open library is a prerequisite. This requires some knowledge about the shell function, which however does not exceed the knowledge required for using the shell. In some particular cases additional knowledge is needed in order to incorporate the functionality in compliance with standard conventions. This knowledge does also not exceed the user level. The shell open library as an implementation interface complies with the requirements for interaction with the shell on a modular basis. The functions through which the shell functionality is accessed hide the implementation details and provide an interface which is invariant with respect to changes in the shell.

The above considerations are also instructive from the point of view of the implementation freedom. The shell and its implementation interface does not strictly imply the convention regarding storing algorithm results to specific pre-defined variables. Besides, implementation of the algorithm allows introduction of new rules that concern use of the incorporated functionality. For example, the optimisation algorithm might return the number of performed iterations in addition other results. The appropriate interpreter function could be implemented so that it would automatically store this result to some specific shell variable. This would introduce a new rule and would actually assign a meaning to that particular variable. The example is not characteristic because it would be much more elegant to assign the number of iterations to a variable specified by a command argument. However, it is possible that incorporation of some important sets of functionality (e.g. shape parametrisation) will be most conveniently implemented by introducing some additional general rules and new groups of pre-defined variables. This is fully supported by the shell implementation interface.

Optimisation algorithms iteratively require performance of the direct analysis, which include evaluation of the objective function and other quantities. Algorithms are usually implemented as functions, which take the function that performs a direct analysis as an argument. The file interpreter function that correspond to an optimisation algorithm must call such a function and provide the analysis function (i.e. its address) as an argument. Figure 4.3 and the surrounding discussion indicate how the optimisation algorithm in connection with the shell internal analysis function operates in practice, while section 4.3.2 and especially the discussion around Figure 4.6 explain how the algorithm is invoked through the shell interpreter command. An open question remains how different analysis functions with different sets of arguments are provided. It is clear that an open library can not contain all possible analysis functions, since the range of possible variants is practically unlimited.

Solution of this problem follows from the general arrangement regarding transfer of current optimisation parameters and results of the direct analysis between an optimisation algorithm and the direct analysis (section 4.1.3). There is a common function which performs a direct analysis for any kind of algorithm. It performs interpretation of the analysis block of the command file, which contains user

definition of the direct analysis. The common analysis function takes no arguments. A specific analysis function must be defined for each optimisation algorithm. This function is actually called by the algorithm and is specified as an argument at algorithm call in the corresponding file interpreter function. It calls the common analysis function and besides writes optimisation parameters and reads analysis output data from the appropriate pre-defined shell variables. The common analysis function is provided by the open shell library as well as functions for setting and reading the pre-defined variables.

The shell library includes some most common analysis functions with different argument lists. These functions can sometimes be used directly in a call to an algorithm or are called in another intermediate function which covers particularities of the algorithm. Particularities arise in various ways. Some algorithms require derivatives and others do not. Some algorithms solve unconstrained problems and therefore do not require values of constraint functions. Different algorithms require analysis data in different form. Some of them use derived structures for representation of matrices and vectors, while some of them use arrays of numbers with separate arguments for specification of array dimensions. Algorithms are also programmed in different languages with different calling and argument passing conventions, which must be accommodated by using intermediate functions.

The most difficult example is when an algorithm uses two or more separate direct analysis functions, e.g. one for evaluation of the objective function and its derivatives and the other for evaluation of constraint functions and their derivatives. Such example is the FSQP algorithm, which is currently the principal optimisation algorithm of the shell. The shell typically calls one common analysis function for a given set of optimisation parameters. The algorithm calls first the function for evaluation of the objective function and its derivatives and then the function for evaluation of constraint functions and their derivatives. The first function must execute the common analysis function, store the constraints related quantities to a temporary location and return the objective function and its derivatives to the algorithm. The second function must simply pick the values and derivatives of constraint functions from the temporary location and return these quantities to the algorithm.

For a general optimisation system it is essential that it includes various types of optimisation algorithms, since any algorithm can handle effectively only a given set of problems. Various mathematical fields relevant for the development of optimisation algorithms are still developing. It is therefore important that different algorithms developed in different environments can be easily incorporated in the system.

On the other hand such distributed development is not economic because it leads to unnecessary replication of work. In practice this also means replication of

code and therefore a large executable programme. Another consequence is rigidity of code where individual solutions can not be easily combined and applied in more complex algorithms because of obstacles related to different programming methodologies.

In the shell development several negative effects of disconnected development of algorithms have already appeared. By now several algorithms have been implemented such as the simplex method, Powell’s direction set method, the Fletcher-Reeves conjugate direction method, and some variants of the penalty methods (refer to chapter 3). These algorithms were programmed in a disconnected way and were many times incorporated in the shell in a nonmodular way. The disadvantage which soon showed was that changes in the shell affected interface between the shell and algorithms. It was difficult to follow permanent changes with a selection of algorithms, which showed the need for a more modular approach.

Unsystematic approach to development has usually a direct impact on code flexibility. In a rigid code it is difficult to change particular details and experiment with different variants of algorithms, which plays an important role in development of efficient algorithms. An evident example in this respect is offered by use of different line search algorithms, especially with respect to termination criteria. Exactness of line searches effects the overall efficiency of different methods in different ways, therefore it is important to allow different termination criteria. The basic idea of the line search algorithm is however independent of this and the algorithm can be programmed in such a way that its core is not affected much by application of different termination criteria. From the view of preventing code replication a good idea would be to implement line search algorithms in two levels. The first level function would contain actual implementation of a line search and would permit choosing between different termination criteria. Two or more higher level functions would be just interfaces for use of the first level function with one or another termination criterion. Algorithms would use these functions with respect to criteria which are better for a specific algorithm. Implementation of higher level functions contributes to clarity of code because their declarations are terse and do not show functionality which is not used in a given place. Implementing common operations only once on a lower level makes it easier to maintain the system and to introduce improvements. In the present example the interpolation can be improved only in the common low level line search function. The improvement affects all derived higher level functions without affecting the way how these functions are used in algorithms. Such a hierarchical system of functions must however be implemented with care. It causes some excess in function calls, therefore it must be checked if this has a significant effect on code efficiency.

Because of the reasons specified above, a more systematic approach to development of a modular optimisation library for the shell has been initiated. Stress is placed on the hierarchical structure of algorithms and utilities for the solution of various sub-problems, on invariant and easy to use implementation interfaces for

these utilities and on exploitation of commonality on the lower level as much as possible. This should enable easier derivation and testing of different variants of algorithms while keeping complexity under control and preventing unnecessary code replication.

Generality of such a library was also taken into consideration as an important design aspect. In this respect it is important to accept some compromises with taking into account the purpose for which the library will be used. For example, it seems unlikely that in problems to which the shell will be applied, any special structure of linear algebra sub-problems that arise could play a crucial role. When the outline of the module for matrix operations was set, dealing with matrices of a special structure was not taken into account. Much stress was placed on the design of a clear and easy to use implementation interface and on dealing with characteristics which often play an important role in optimisation algorithms (e.g. testing positive definiteness of matrices). The basic design of the module for matrix operations is outlined in [29].

For an optimisation library designed for incorporation in the shell, many aspects which are not directly related to algorithms are relevant. This includes some general aspects of synchronous function within a broader system, such as error handling and output control. The library will provide flexible access to these functions with possibility of their adjustment to the system in which the library functions will be used. Considering straightforward incorporation of developed algorithms in the shell is important also from the point of view that the shell can provide a good testing environment.

4.3.3.1 Example: Incorporation of an Optimisation Algorithm

A complete procedure of incorporating an optimisation algorithm in the shell is shown in an example. This should more evidently illustrate the properties of the shell implementation interface described above. The Nelder-Mead simplex algorithm was chosen to illustrate the procedure. It is implemented by a C function declared as

```
void minsimp(matrix simp, vector val, double tol, int *it, int
maxit, vector *paropt, double *valopt, double func(vector x),
FILE *fp)
```

Matrix argument *simp* is the matrix of simplex apices (input and output), vector argument *val* is the vector of function values in the simplex apices, *tol* is the tolerance in function value (convergence criterion – input), *it* is the number of iterations (output), *maxit* is the maximum allowed number of iterations (input), *paropt* is the vector of optimal parameters (output), *valopt* is the optimal value of the objective function (output), *func* is the address of the function that performs the direct analysis, and file argument *fp* is the file in which results are written. *matrix* and *vector* are derived types which represent matrices and vectors. Similarly as there

exists a module for manipulating *stack* objects described in section 4.3.1.1, corresponding modules exist for manipulation of objects of these two types.

First the analysis function, which will be passed as argument *func* of the algorithm function, must be defined. The function must take a vector argument (optimisation parameter) and return a floating point number (value of the objective function). It can be defined in the following way:

```
double ansimp(vector x)
{
setparammom(x);
analysegen( );
return getobjectivemom(0);
}
```

The function first sets the shell vector variable *parammom* (the current parameter values by convention) to its argument *x*, which is passed by the calling algorithm. This is done by the open library function *setparammom*. Then the common analysis function *analysegen* is called, which performs interpretation of the analysis block of the command file, which is the user definition of the direct analysis. The function then returns the value of the scalar variable *objectivemom* (value of the objective function by convention), which is obtained by the library function *getobjectivemom*. This value is set at interpretation of the analysis block.

The interpreter function which will run the algorithm can now be defined. It is assumed that the algorithm will be run by the interpreter command *optsimp* with the following argument list:

```
optsimp(simp val tol maxit)
```

where *simp* is a matrix argument (initial simplex), *val* is a vector argument (function values in the initial simplex), and *tol* (tolerance) and *maxit* (maximum number of iterations) are scalar arguments. It is assumed that the initial simplex and function values in its apices are provided by the user and passed as command arguments. The interpreter function can be defined as follows:

```
double fi_optsimp(ficom fcom)
{
matrix simp=NULL;
vector val=NULL, paropt=NULL;
double valopt, tol, maxit;
int it;
/* Extract interpreter command argumetns: */
readmatarg(fcom,&simp);
readvecarg(fcom,&val);
readscalarg(fcom,&tol);
readscalarg(fcom,&maxit);
/* Run the algorithm: */
```

```
minsimp(simp,val,tol,&it,maxit,&paropt,&valopt,ansimp,fcom-
>out)
/* Set pre-defined shell variables paramopt and objectiveopt,
which represent optimal parameters and optimal value of the
objective function, respectively: */
setparamopt(paropt);
setobjectiveopt(valopt);
/* release local variables: */
dispmatrix(&simp);
dispvector(&val);
dispvector(&paropt);
}
```

Comments in the above code explain its function to a large extent. The function takes a single argument, which is the interpretation object described in section 4.3.2 and through which all information regarding command file interpretation can be accessed. The function first extracts arguments which are passed through the argument block of the corresponding interpreter commands. Open library functions of extraction of different types of command arguments are used. These functions take the interpretation object as the first argument, since the command argument block is accessed through it. Each of these functions sets the field *fcom->argpos* to the position after the last extracted argument, so that the next function can begin argument extraction on the right place. The second argument of these functions is always the address of the variable in which the argument value is stored. All interpreter command arguments are in this case input arguments of the algorithm. In general there could also be other types of arguments, for example specifications of the shell variables where specific algorithm data should be stored.

The simplex algorithm is then called. The last argument of the call but one is the direct analysis function *ansimp*, which was defined above. The last argument is the file interpreter output file, which in the shell coincides with the shell output file.

The algorithm performs minimisation of the objective function and returns optimal parameters in the vector *paropt* and optimal value of the objective function in the number *valopt* (both are local variables). These values are then copied to the appropriate pre-defined shell variables, so that they are accessible for further use by other algorithms and utilities. This is done by the appropriate library functions. Finally, the dynamic storage which was allocated within the function is released.

After the file interpreter function is defined, the appropriate command can be installed in the file interpreter system. This is done by the following line of code:

```
instfintfunc ( “optsimp”, fi_ optsimp );
```

instfintfunc is another function of the shell open library, which installs a file interpreter command on its function definition stack. Its first argument is the name of the command, which will be used in the shell command file. The second argument is

the address of the function which is run by the file interpreter when the corresponding command is encountered. The above code must be added in the specific source file, which is compiled and linked with the optimisation shell. This code is executed at shell initialisation.

4.3.4 Parallelisation

Parallel execution of code represents an attempt at exceeding practical limits imposed by the capability of available computers, which are often very restrictive for demanding numerical applications. Parallelisation of the optimisation shell[8],[9] is the final implementation issue discussed in this text. It is instructive because parallelisation usually requires considerable rearrangement in the code structure. In the shell a different philosophy of parallel execution is transferred to the algorithmic level, which enables continuity of the described shell concepts and coexistence of the parallel and sequential schemes. The shell provides support for straightforward incorporation of parallel algorithms in a similar way as sequential algorithms.

The parallel interface has been built using the MPI (Message Passing Interface)[32] library and the LAM (Local Area Multicomputer)[33] environment. This enables a system to be implemented on a wide range of architectures, including shared and distributed memory multiprocessors as well as arbitrarily heterogeneous clusters of computers connected in a LAN (Local Area Network).

Direct analyses are treated as basic operations that can be executed simultaneously. The parallel scheme is implemented as a master-slave architecture (Figure 4.7). The master process controls execution of optimisation algorithms while slave processes perform simulations. Several slave processes run simultaneously on different processing units performing simulations for different sets of optimisation parameters. Both master and slave processes are actually optimisation shells with complete functionality, but their execution differs due to different functions.

The master process is responsible for the process management and keeps track of the slave processes status. All actions of the slave processes are triggered by the master, so every change of the slave status is conditional on the master request. The master process also registers the execution times of direct analyses. This information is used for the so-called load balancing in the case when the master can choose between several slaves that are available for execution of a new task.

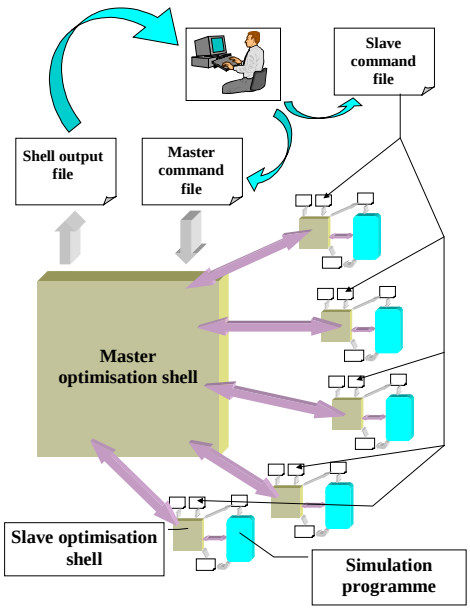

Figure 4.7: Parallel optimisation scheme[9]. The main optimisation shell controls the optimisation process while the slave shells interact with simulation programs and execute direct analyses.

The direct analysis is performed in three parts. The first part is executed by the master process, which sends the analysis request and appropriate data (e.g. parameter values) to a slave. The second part is executed by a slave process, which runs the simulation, collects results and evaluates appropriate quantities, which are then sent back to the master. The third part is executed by the master process. The quantities sent by the slave process are received and transferred to the calling optimisation algorithm.

Each part of the function evaluation is performed by its own function, which interprets a specific pre-defined block in the master or slave command file. This enables not only automatic exchange of parameter values and analysis results, but also arbitrary data exchange between the master and slave processes. The parallel interface provides file interpreter commands for this task.

Optimisation algorithms can use the first and third functions, which transfer data between algorithms and function definition in the appropriate command files. These two functions are an equivalent to the common analysis function in the sequential scheme. In order to enable proper task distribution, the first function must not only accept the parameter values, but also return to the algorithm identification of the slave process to which the analysis request has been sent. Similarly, the third

function must return the identification of the process that performed the task. The first function must check whether there is any slave process ready to accept the task. The third one must check if slave processes have finished any task and, if necessary, wait for the next available results.

The master process interprets a command file in the same way as a sequential scheme. Every action is a consequence of a function call in the command file. The behaviour of slave processes is different since these processes only respond to master requests. When a slave process is spawned, it interprets its command file without exiting and then waits for the master process requests. The interpretation of the command file is a part of initialisation, while later on every action is triggered by a master process request. The communication between the master and slave processes is synchronised[8],[9]. To make the described functionality more clear, the course of a direct analysis is desctibed below.

Figure 4.8 shows how a direct analysis is executed on the master (the left-hand side of the figure) and a slave (the right-hand side) process. Times of characteristic events are marked by t_0 through t_{26} in the order in which these events follow each other. The time scale is not proportional.

When the algorithm requires execution of a direct analysis and a slave process which is ready to accept a task exists, the master process sends a task request to that process (t_1 to t_9). The slave is in the waiting state at that time, which is known to the master because it keeps track of slave processes status. The slave reestablishes such a state every time after it completes an action requested by the master (t_0 and t_{25} in the figure).

The master notifies the slave of the request by sending it the “BEGIN_DATA” message. After the receipt of this message the slave accepts the data sent by the master and stores it to the appropriate location. The master is sending the contents of its variables to the slave. The slave stores these contents in the variables of the same names (note that both master and slave are actually complete optimisation shells). Data packages carry complete information regarding the position of the data in the system of shell variables. Interpretation of the “analysis” block of the master command file (t_3 to t_4) is a part of sending a task request to the slave. In this block the user can send arbitrary data from the master system of user defined variables by using appropriate file interpreter commands. After that, standard data, i.e. the vector of parameter values *parammom*, is sent automatically (t_5 to t_6). The “END_DATA” message is then sent to the slave. After its receipt (t_7) the slave stops accepting data from the master and expects the “START_AN” message. Its receipt invokes the slave process analysis function (t_8 to t_{14}), which includes interpretation of the “analysis” block of the slave command file (t_{10} to t_{12}). Normally this part corresponds to the actual performance of the direct analysis, while other parts simply take care of the proper data transfer between the master and the slave.

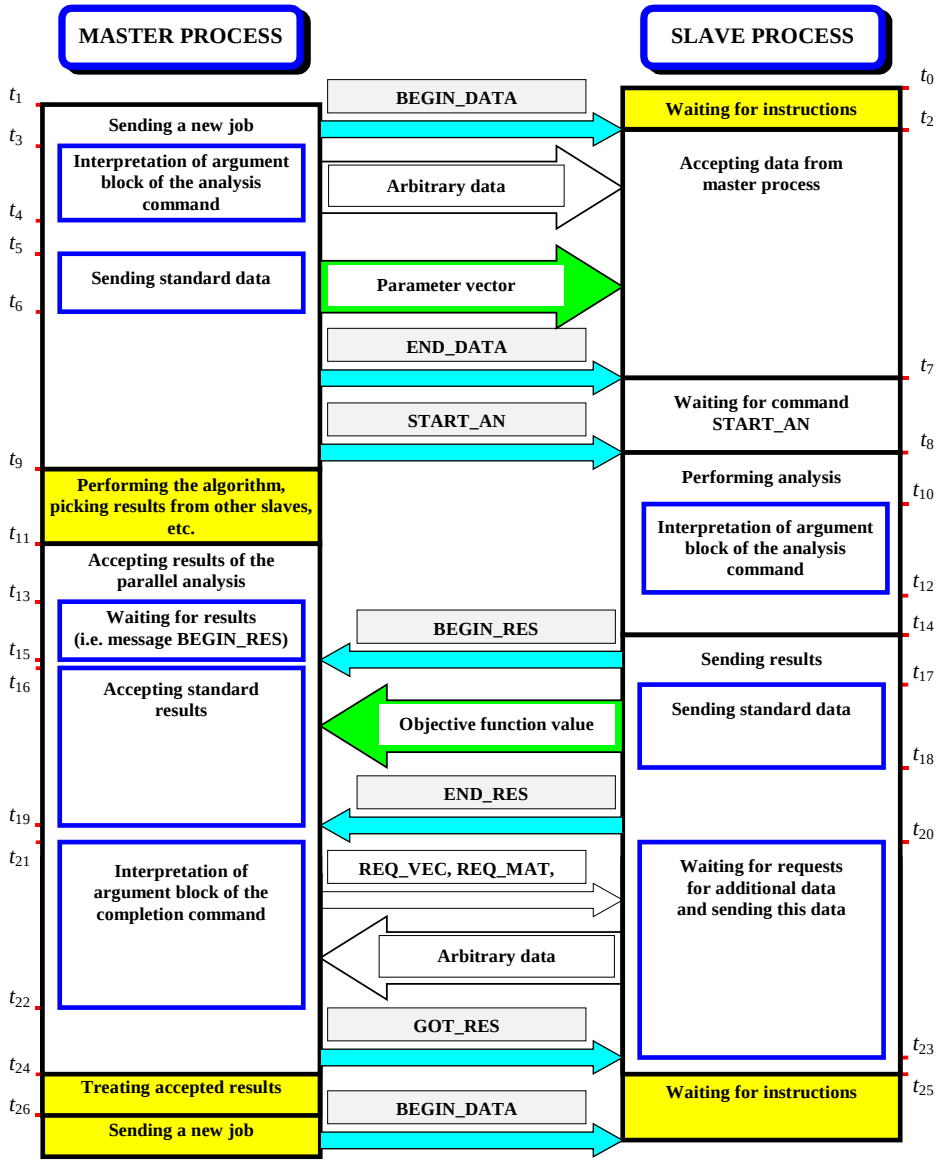

Figure 4.8: Course of a direct analysis in the parallel scheme[9].

After sending the task, the master can continue to perform the algorithm with available data, send parallel tasks to other slaves and accept results from them. The slave sends the “BEGIN_RES” message to the master after it finishes the analysis (t_{14}). The message is buffered until the master is ready to accept results (t_{11}). Depending on the state of the algorithm this can happen in two ways. The master can just check if such message has been sent by any of the slaves. If this is the case, it accepts the results (t_{16} to t_{24}); otherwise it continues to do other operations and

repeats the check later. The other situation occurs when the master can not do anything until it obtains results of at least one direct analysis. This typically happens when all available slaves are busy and the master has already treated all results which have arrived. In this case the master blocks its execution until the message “BEGIN_RES” is received from any of the slaves.

Receipt of results is somehow a reversed process to that of sending a task. After a receipt of the “BEGIN_RES” message the master accepts data sent by the slave until the receipt of the “END_RES” message. The slave sends this message after sending the standard data (t_{17} to t_{18}), e.g. the value of the objective and constraint functions and their gradients. After that, the master interprets the “completion” block of its command file (t_{21} to t_{22}) where the user can send requests for additional data to the slave. The slave waits for such requests and sends back the requested data (t_{20} to t_{23}) until it receives the “GOT_RES” message. The master sends this message after interpretation of the “completion” block. For the slave its receipt means that it has finishes the task. It reestablishes the waiting state (t_{25}) and is able to accept further task requests.

The shell provides various mechanisms for process management and load balancing. Processes can be controlled, enabled and disabled by the user during the runtime[34]. This enables better performance to be achieved for a given kind of algorithm and controlled use of computational resources.

The parallel scheme does not significantly affect the concepts of the shell, neither from the user point of view nor from the view of the implementation interface for incorporation of new tools.

Parallel algorithms are incorporated in the shell in a similar way as sequential algorithms. There are two fundamental differences. The direct analysis now consists of two parts, executed at different times while other analyses can be run in between. A special function executes each part. These functions are called by the algorithm and must be provided as arguments at the call to the algorithm in the appropriate file interpreter function. They are referred to as the calling analysis function and the returning analysis function. They are specific for each algorithm, while each of them calls the appropriate common analysis function (calling or returning, respectively). The second difference is that calling and returning analysis functions return the identification number of the process which performs the appropriate analysis. This is necessary for the algorithm in order to connect specific analysis results (which are received in unpredictable order) with the corresponding optimisation parameters. Both differences are fundamentally conditioned by the nature of parallel execution and therefore do not represent an unnecessary excess in complexity.

References:

[1] I. Grešovnik, J. Korelc, T. Rodič, T. Sustar, *An approach to more efficient development of computational systems for solution of inverse and optimisation problems in material forming*, In: SciTools'98 : book of abstracts, International Workshop on Modern Software Tools for Scientific Computing, Oslo, Norway, September 14-16, 1998, Oslo, SINTEF Applied Mathematics, 1998, pp. 31, Accessible on the Internet (URL): http://www.oslo.sintef.no/SciTool s98.

[2] T. Rodič, J. Korelc, M. Dutko, D.R.J. Owen, *Automatic optimisation of perform and tool design in forging*, ESAFORM bulletin, vol. 1, 1999.

[3] T. Rodič, I. Grešovnik, D.R.J. Owen, *Symbolic computations in inverse identification of constitutive constants*, In: WCCM III, Extended abstracts, Third World Congress on Computational Mechanics, Chiba, Japan, August 1-5, 1994, [S. l.], International Association for Computational Mechanics (IACM), 1994, vol. II, pp. 978-979.

[4] I. Grešovnik, T. Rodič, *Optimization system utilizing a general purpose finite element system*, In: Second World Congress of Structural and Multidisciplinary Optimization, Zakopane, Poland, May 26-30, 1997, Extended abstracts, [S. l.], ISSMO, 1997, pp. 368-369.

[5] I. Grešovnik, T. Rodič, *Optimization system utilizing a general purpose finite element system*, In: WCSMO-2 : proceedings of the Second World Congress of Structural and Multidisciplinary Optimization, Zakopane, Poland, May 26-30, 1997. Vol. 1, Witold Gutkowski, Zenon Mroz (editors), 1st ed., Lublin, Poland, Wydawnictwo ekoincynieria (WE), 1997, pp. 61-66.

[6] T. Rodič, I. Grešovnik, *A computer system for solving inverse and optimization problems*, Eng. comput., vol. 15, no. 7, pp. 893-907, 1998.

[7] I. Grešovnik, T. Rodič: *A general-purpose shell for solving inverse and optimisation problems in material forming*. In: COVAS, José Antonio (editor). Proceedings of the 2nd ESAFORM Conference on Material Forming, Guimaraes, Portugal, [13-17 April 1999]. Guimaraes, Portugal: Universidade do Minho, Departamento de Engenharia de Polímeros, 1999, pp. 497-500.

[8] I. Grešovnik, T. Šuštar, T. Rodič, *Paralelizacija v lupini za resevanje optimizacijskih in inverznih problemov* (in Slovene), In: Zbornik del, Kuhljevi dnevi '98, Logarska dolina, Slovenija, 1.-2. oktober 1998, Boris Štok (editor), Ljubljana, Slovensko društvo za mehaniko, 1998, pp. 169-176.

[9] I. Grešovnik., T. Šuštar, T. Rodič: *Parallelization of an optimization shell*. In: 3rd World congress of structural and multidisciplinary optimization : Buffalo, NY, May 17-21, 1999. Buffalo: WCSMO, 1999, pp. 221-223.

[10] *Optimization Shell Inverse*, electronic document at http://www.c3m.si/inverse/ , maintained by the Centre for Computational Continuum Mechanics, Ljubljana.

[11] I. Grešovnik et al., *What Inverse Is*, electronic document at http://www.c3m.si/inverse/basic/what/ , maintained by the Centre for Computational Continuum Mechanics, Ljubljana.

[12] *Inverse Manuals*, electronic document at http://www.c3m.si/inverse/man/ , maintained by the Centre for Computational Continuum Mechanics, Ljubljana.

[13] I. Grešovnik, *Quick Introduction to Optimization Shell "Inverse"*, electronic document at http://www.c3m.si/inverse/doc/other/quick/ , maintained by the Centre for Computational Continuum Mechanics, Ljubljana, 1999.

[14] I. Grešovnik, *A Short Guide to the Optimisation Shell Inverse – for version 3.5*, electronic document at http://www.c3m.si/inverse/doc/man/3.6/short/index.html , Ljubljana, 1999.

[15] I. Grešovnik, *Programme Flow Control in the Optimisation Shell Inverse*, electronic document at http://www.c3m.si/inverse/doc/man/3.6/flow/index.html , Ljubljana, 2000.

[16] I. Grešovnik, *The Expression Evaluator of the Optimisation Shell Inverse*, electronic document at http://www.c3m.si/inverse/doc/man/3.6/calc/index.html , Ljubljana, 2000.

[17] I. Grešovnik, *User-Defined Variables in the Optimisation Shell INVERSE*, electronic document at http://www.c3m.si/inverse/doc/man/3.6/var/index.html , Ljubljana, 2000.

[18] I. Grešovnik, *Solving Optimization Problems by the Optimization Shell Inverse*, electronic document at

http://www.c3m.si/inverse/doc/man/3.6/opt/index.html , Ljubljana, 2000.

[19] I. Grešovnik, *A General File Interface for the Optimisation Shell Inverse*, electronic document at http://www.c3m.si/inverse/doc/man/3.6/int/index.html , Ljubljana, 2000.

[20] I. Grešovnik, *Syntax Checker and Debugger for Programme INVERSE*, electronic document at http://www.c3m.si/inverse/doc/man/3.6/debug/index.html , Ljubljana, 2000.

[21] I. Grešovnik, *Miscellaneous Utilities of the Optimisation Shell INVERSE*, electronic document at http://www.c3m.si/inverse/doc/man/3.6/misc/index.html , Ljubljana, 2000.

[22] I. Grešovnik, D. Jelovšek, *Interfaces Between the Optimisation Shell INVERSE and Simulation Programmes*, electronic document at http://www.c3m.si/inverse/doc/man/3.6/sim/index.html , Ljubljana, 2000.

[23] B. W. Kernigham, D. M. Ritchie, *The C Programming Language (second edition)*, Prentice – Hall, 1988.

[24] A. Kelley, I. Pohl, *A Book on C - an Introduction to Programming in C*, Menlo Park, 1984.

[25] A. Kelley, I. Pohl, *A Book on C: Programming in C (fourth edition)*, Addison – Wessley Publishing, New York, 1998.

[26] M. Banahan, D. Brady, M. Doran, *The C Book – Featuring the ANSI C Standard (second edition)*, Addison – Wesley Publications, Wokingham, 1991.

[27] J. Valley, *C Programming for UNIX*, SAMS Publishing, Carmel, 1992.

[28] B. Stroustrup, *The C++ Language (second edition)*, Addison – Wesley Publishing, New York, 1992.

[29] I. Grešovnik, *Library of Matrix and Vector Operations*, C3M internal report, Ljubljana, 1999.

[30] *Elfen – Product Information*, electronic document at http://rsazure.swan.ac.uk/products/elfen/elfen.html , maintained by Rockfield Software Ltd., Swansea.

[31] *Elfen Implicit User Manual – version 2.7*, Rockfield Software Limited, Swansea, 1997.

[32] *MPI, A Message-Passing Interface Standard*, electronic document at http://www.mcs.anl.gov/mpi/mpi-report-1.1/mpi-report.html, University of Tennessee, Knoxville, Tennessee, 1995.

[33] *MPI Primer / Developing With LAM*, electronic document in PostScript at ftp://ftp.osc.edu/pub/lam/lam61.doc.ps.Z , Ohio Supercomputing Center, the Ohio State University, 1996

[34] Igor Grešovnik, *Paralelizacija v lupini Inverse* (in Slovene), C3M internal report, Ljubljana, 1997.