A Computer System for Solving Inverse and Optimization Problems

T. RODIÈ

Faculty of Natural Sciences and Engineering,
Aškerèeva 12, 1000 Ljubljana, Slovenia

I. GREŠOVNIK

C3M, Vandotova 55, Ljubljana, Slovenia

ABSTRACT

A system for solving wide variety of inverse and optimization problems in solid mechanics is introduced. The system consists of a general purpose finite element method (FEM) analysis system "Elfen" and a shell which controls this system. The shell functions as a stand-alone programme, so the system is physically divided into two separated parts. The "optimization part", which corresponds to the shell, possesses optimization and inverse problem solution algorithms. The "analysis part", which corresponds to a FEM system, serves for the definition of the objective function to which these algorithms are applied. The shell has a user interface implemented in the form of file interpreter which imposes a great flexibility at the definition of various optimization and inverse problems, including parameter identification in constitutive modelling, frictional contact problems and heat transfer. Concepts of the shell are discussed in detail.

KEYWORDS
optimization system, software, engineering

INTRODUCTION

A system described in this paper combines a FEM analysis (Owen, 1980, Zienkiewicz, 1991) system "Elfen" with a programme shell "Inverse" which utilizes the system for solving inverse (Grešovnik, 1996) and optimization problems.

Elfen is a general purpose FEM programme for solution of thermo-mechanical problems in solid mechanics. Since it is capable of solving nonlinear problems involving large strains and deformations, different material models, thermo-mechanical coupling and contact phenomena, it is convenient for simulation of a large range of forming processes or product behaviour in operating conditions.

Primary job of the shell is to perform the optimization or inverse problems solution algorithms and to control the FEM analyses execution through these algorithms. Beside the algorithms, the shell possesses a flexible user interface which enables the definition of a large set of problems, and an interface to a FEM system which enables total control over its execution.

The objective of this paper is to describe the concepts of the shell which make the system suitable for industrial use as well as for research purposes. A flexible user interface of the shell, implemented as an interpreter, enables user to define a large range of different problems and to combine built-in facilities in arbitrary way to construct the most suitable solution strategy for every individual problem. This interface and the way how different parts of the shell interact with each other leave the user a possibility to strongly interfere with the solution algorithms which are normally supposed to be a static part of the system. On the other hand, user is still free to use the system as easy-to-use, taking the advantage of the high-level pre-defined algorithms for more typical problems.

In the first part of the paper, some basic concepts of the shell are considered regarding a typical optimization or inverse problem solution scheme. After that, structure of the shell is described and connections between the shell's structure and functionality are showed. These features are then discussed from user's point of view by pointing out the relationship between structure and functionality of the shell. References to examples are also provided. Finally, directions for further development are outlined.

BASIC CONCEPTS OF THE SHELL

Inverse and optimization problems are often formulated as minimization problems where an objective function is minimized with respect to investigated parameters (Fletcher, 1987). The dependence of the objective function on the design parameters is implicit because it is defined through the system response. For each set of design parameters system response must be obtained using a FEM simulation in order to evaluate the objective function at a given set of design parameters. This implies the solution scheme of optimization problems to be, in general, similar to the scheme shown in Figure 1.

Since a finite element analysis is a part of the optimization scheme, it seems natural to include optimization algorithms directly into the existing FEM code and use this code as a subroutine which is successively executed in the optimization loop. An advantage of such approach is that optimization routines can access the FEM analysis' database directly, which means quick transfer of data between optimization and FEM part of the scheme. This includes transfer of design parameters, which are iteratively updated in optimization algorithms, to FEM analysis, and transfer of analysis results back to optimization algorithms.

On the other hand, time used for updating input and reading output of FEM analysis is usually negligible in comparison with the time needed for analysis itself. Besides, the optimization and the FEM part of an optimization system do not have much in common from the programming point of view.

FEM programmes typically have a user interface where user defines the problem in a descriptive way. A complex branching part takes care about proper translation of user's description of a problem to a sequence of algorithms resulting in solution of the problem. The branching part must ensure fulfillment of the demand that a FEM programme should be capable of solving a large range of different problems according to their physical nature and should also be able to use different solution strategies.

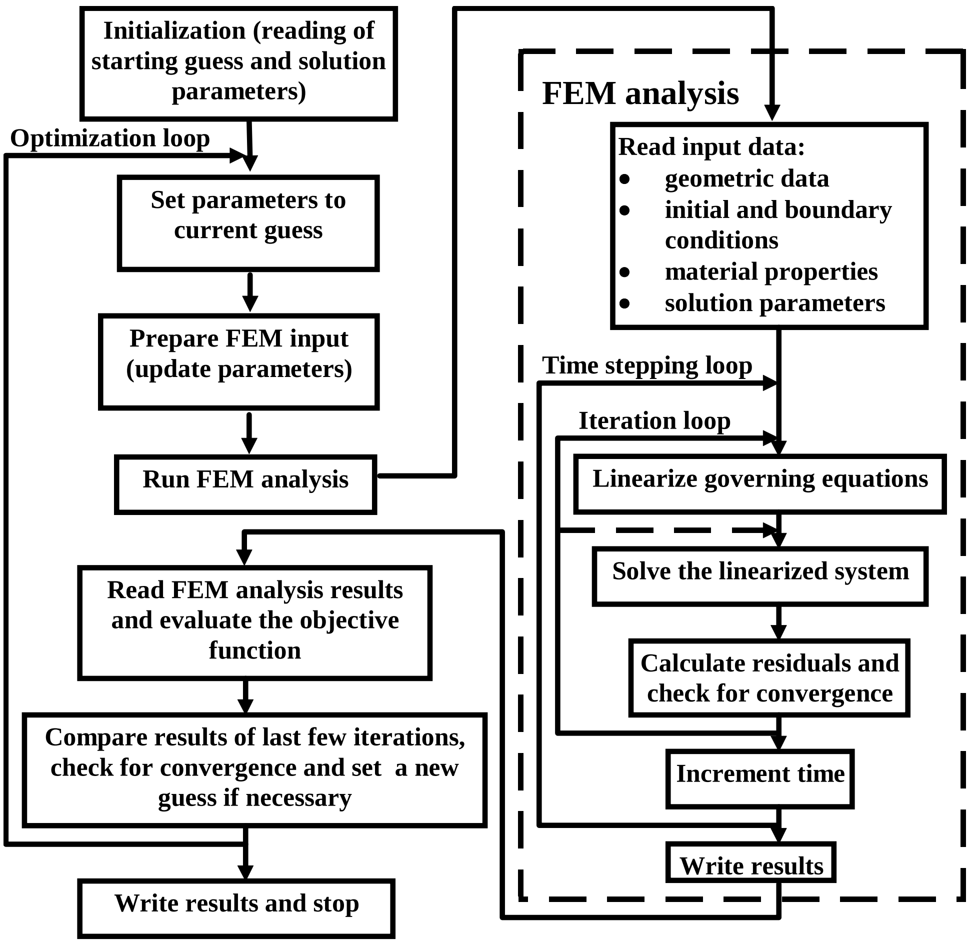

Initialization (reading of starting guess and solution parameters)
Optimization loop
Set parameters to current guess
Prepare FEM input (update parameters)
Run FEM analysis
Read FEM analysis results and evaluate the objective function
Compare results of last few iterations, check for convergence and set a new guess if necessary
Write results and stop
FEM analysis
Read input data:
• geometric data
• initial and boundary conditions
• material properties
• solution parameters
Time stepping loop
Iteration loop
Linearize governing equations
Solve the linearized system
Calculate residuals and check for convergence
Increment time
Write results

Figure 1. A simplified scheme of optimization of a physical system where a FEM analysis programme is used for evaluation of system response at different sets of design parameters.

For a limited range of optimization problems such descriptive problem definition would be sufficient and in that case a modified FEM user's interface could be used for a definition of a whole optimization problem. For a general case such approach would be to rigid and difficult to implement. It is therefore better to make a definition of a direct problem independent and separated from the optimization problem definition, which follows and potentially includes the instructions for redefining the direct problem according to changes in design of the system inside iterations of the optimization loop. Such concept is used in the presented optimization shell. Because the shell has totally different structure than a typical FEM programme, it is implemented as a stand-alone programme which uses to run a FEM analysis programme as it's subroutine for performing numerical simulations of the physical system at different designs. The solution scheme coincides with the one in Figure 1.

A FLEXIBLE SYSTEM FOR SOLVING OPTIMIZATION PROBLEMS

As mentioned above, one of the basic demands for a good optimization system is flexibility of user interface through which optimization problems are defined. A physical system, which behaviour can be simulated numerically, associates a whole range of possible optimization problems, each of which can be approached different ways. An optimization shell should not feature only a possibility of running optimization algorithms, but also provide other tools like tabulating various quantities derived from system response at different sets of design parameters. Such additional tools are often used to verify validity of results obtained by optimization algorithms or for pre-justification of optimization potentials.

These demands have affected the way the optimization shell was designed. It was built around a file interpreter which reads and executes functions or commands from a command file created by user. Each function launches a specific action, behaviour of which is dependent on function arguments. Apart from functions which perform optimization algorithms and other jobs, there are also functions which control the program flow. With such scheme, user of the shell is not strictly limited with its pre-built capabilities but can himself strongly interfere with solution algorithms. User can even program his own solution algorithms and use only some supporting capabilities of the shell.

Figure 2 shows the structure and functionality of the optimization system. The core module of the shell contains built-in routines for different jobs connected with optimization problems, such as the objective function minimization algorithms or tabulating utilities. Global shell's variables such as current vector of parameters and value of the objective function are also a part of this module. The core routines are invoked via user commands read by the file interpreter.

Some core routines iteratively invoke the FEM analysis for calculation of the objective function. Instead of being static, this invocation is defined in a special block in file interpreter. This block is an argument block of the "analysis" command of the file interpreter. It defines the objective function evaluation and is executed anew each time the evaluation of the objective function is requested by one of the core routines or by an explicit command in the command file. So the circle is closed, user of the system can define the objective function by first defining a direct problem for FEM analysis and then finishing the definition by writing the argument block of the "analysis" function in the command file. Then he can cause execution of shell's routines for the objective function minimization by an appropriate command in the command file. This coincides with implementation of the scheme in Figure 1.

Some further explanations of Figure 2 are necessary at this stage. The shell's file interpreter includes a system for evaluation of mathematical expressions (referred to as expression evaluator). Its primary job is to enable the program flow control. The flow control functions of the interpreter such as *if*, *while*, *goto*, etc. use the evaluator for evaluating the branching conditions. In addition, the interpreter possesses commands for defining new variables and functions within the expression evaluator. So user can program the shell in a way similar to writing programmes in the C or other high level programming languages.

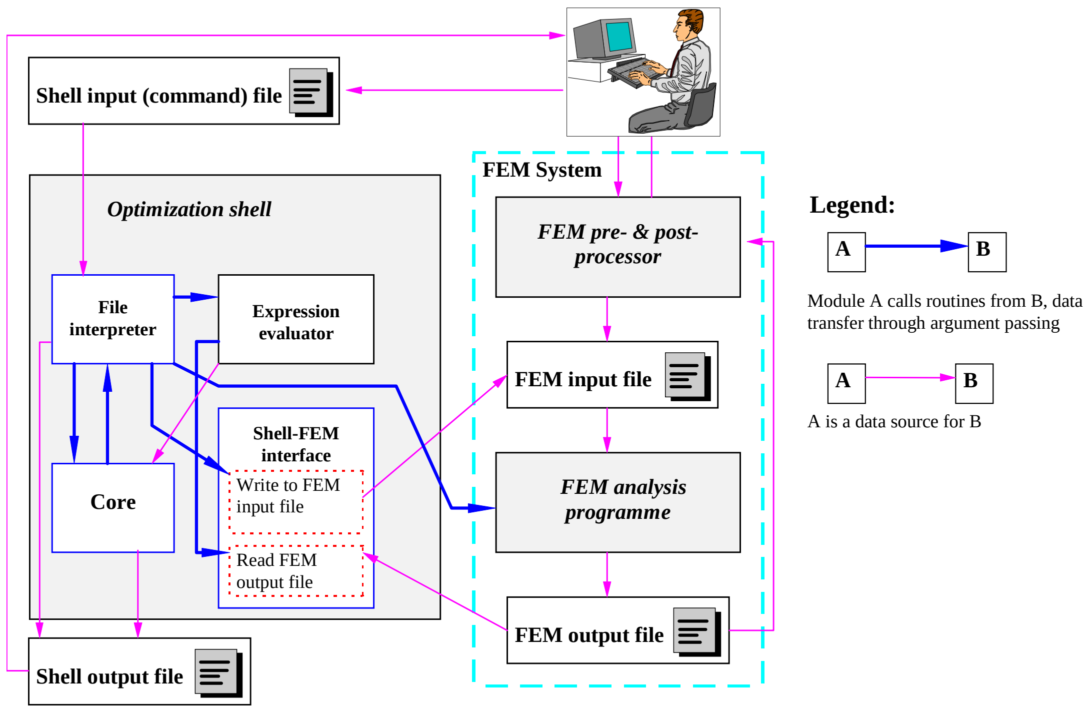


Figure 2. Structure of the optimization system. Data sources for individual modules and calling protocols are shown.

The file interpreter's functions can use expressions instead of values for their numerical arguments. At interpretation, these expressions are evaluated by the expression evaluator and replaced by obtained numerical values.

An important feature of expression evaluator are its functions for accessing the global variables defined in the shell's core. By these functions, all the global variables are accessible wherever the evaluator can be used, for example in conditions controlling the file interpretation flow or in arguments passed to the file interpreter's functions. Since every action of the shell is invoked with these functions, unlimited data transfer between different shell's modules is achieved and every result of built-in or programmed algorithms can be used for further analysis with other algorithms. This feature ensures the necessary flexibility at defining optimization problems and their solution strategies.

1 Definition of the objective function

As can be seen from Figure 1, the definition of the objective function is an important step at solving optimization problems. It consists of two major steps. In the first step user defines the direct problem which will be solved by a FEM analysis system. In the second step, the definition is completed by writing argument block of the shell's file interpreter's function "*analysis*". This block is executed every time the objective function is calculated within the shell and must therefore contain a mechanism for transferring data between the FEM system and the shell's core. This mechanism works through global variables of the shell's core.

Shell's core functions which require evaluation of the objective function call another core function to perform the task. This function requires vector of the design parameters as argument and returns the value of the objective function. When called, it copies its argument to a pre-defined global variable *parammom* representing the design parameters at a specific moment, then it invokes interpretation of the "analysis" command's argument block and at the end it returns the value of a pre-defined global variable *objectivemom*, representing the value of the objective function at a specific moment, to a calling function.

User must ensure that the design parameters are passed to the FEM system before the FEM analysis is invoked, that results are collected after the analysis completion, and that the objective function is evaluated using these results and its value is stored to global variable *objectivemom*. The shell-FEM interface module provides the necessary instrumentation to do that within the argument block of the file interpreter's command "*analysis*".

The design parameters are transferred to a FEM system by changing the FEM input file accordingly to these parameters. If the design parameters coincide with specific parameters in the shell input file, this can be done using built-in interface functions invoked by the file interpreter's commands. For more complicated cases, where a whole series of the FEM input parameters depend on a single design parameter, user can write his own interface programme for updating the FEM input file according to the design parameters, since the shell's file interpreter provides commands for invoking arbitrary programmes on the computer on which the shell is running. These commands are also used to run the FEM analysis after the input file is updated.

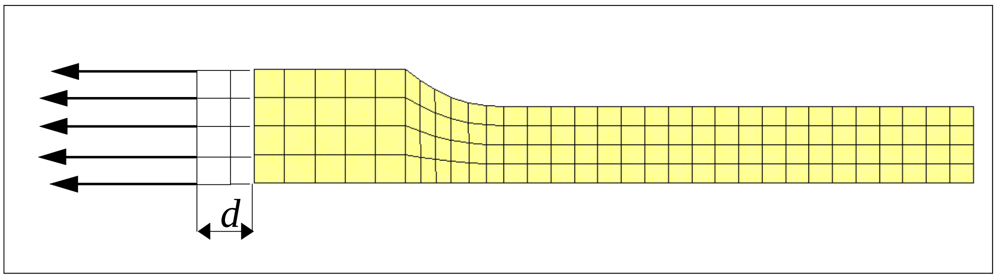
Results are collected using interface' functions which read the FEM output file. The expression evaluator contains pre-defined function which call these interface's function and can return any elementary result of the FEM system. So the results can be directly combined into more complicated expressions defining the recipe for calculating the objective function.

COMMAND FILE

The command file syntacs is simple. Interpreter searches through the file for known command names and triggers appropriate actions. Each command (also referred to as interpreter's function) can have arguments in brackets. It depends on the associated action how the arguments are treated.

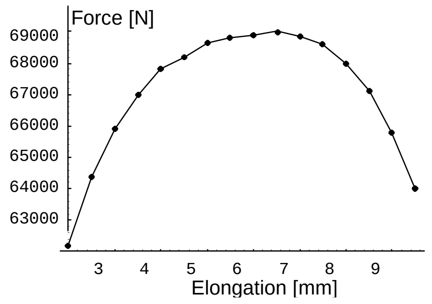
A simple example of a command file for solution of an inverse problem presented in Figure 3 is shown in Box 1. Command *"setfile"* in the second line specifies the name of the file where the shell's results and other output will be written. When it is executed, the file is opened for writing and after that all output from other actions is written to that file.

Tension Test



Material law: $\overline{\sigma} = C\overline{\varepsilon}^{n}$

Measurements:



Elongation [mm]	Force [N]	
2	**62200**	$= F_1^{(m)}$
4	**67800**	
6	**68900**	
8	**68000**	

Inverse identification: $$\chi^2(C,n) = \sum_{i=1}^{N} \frac{\left(F_i^{(m)} - F_i(C,n)\right)^2}{\sigma_i^2}$$

Figure 3. Example of inverse identification: Identification of hardening parameters in tension test.

```
1.  comment{ BEGINNING OF THE COMMAND FILE }
2.  setfile{outfile “test.inv”} *{output file of the shell}
3.  setvector{ meas 4 {1:62200} {2:67800} {3:68900} {4:68000} }  *{ vector of measurements}
4.  setvector{ sigma 4 {1 1 1 1} }  *{ vector of measurement erors }
5.  *{ Definition of a new function in expression evaluator: }
6.  ${force[inc]: nodreac[inc,4,1] +nodreac[inc,5,1] +nodreac[inc,6,1] +nodreac[inc,7,1] +nodreac[inc,8,1] }
7.  analysis
8.  {
9.      *{ beginning of  the “analysis” block }
10.     setfile{aninfile “test.dat”}  *{ FEM system’s input file }
11.     initinput{}  *{ initialization of interface }
12.     setparam{} *{ updating parameters in FEM system’s input data }
13.     system{“elfen16 test”}  *{ running a FEM programme }
14.     setfile{anoutfile “test.res”}  *{  FEM system’s output file  }
15.     initoutput{}  *{ initialization of interface }
16.     meas{1 “force[1]”}  *{ setting components of simulated measuremets }
17.     meas{2 “force[2]”}
18.     meas{3 “force[3]”}
19.     meas{4 “force[4]”}
20. }
21. setvector{ parammom 2 {1276  0.1124} }  *{ setting vector of parameters }
22. analyse{}  *{ running analysis at given prameters }
23. inverse  *{ running inverse analysis }
24. {
25.     nd LevMarqconst  0.001  300
26.     2 { { 1 : 1000 } { 2 : 0.1 } }
27.     2 {{ 1 : 0.1 } { 2 : 0.0001 }}
28. }
29. comment{ END OF THE COMMAND FILE }
```

Box 1. Example of command file for inverse problem presented in Figure 3.

The following *“setvector”* commands in lines 3 and 4 set the pre-defined vectors of measurements and measurement errors which are input data for inverse analysis.

Command *“$”* defines mathematical expressions and functions in the expression evaluator. In line 6, function “force” is defined as a sum of the first components of nodal forces in nodes 5, 6, 7 and 8. Argument of the function tells in which increment to take the forces. Pre-defined function *“nodreac”*, which is used in the definition, is a part of the FEM-shell interface (Figure 2). It returns the specified nodal reaction as can be found in the result file of the FEM analysis system and its three arguments specify the increment after which the force is took, the number of node in which the force acts, and which component of the force to take. To work properly, at the time of evaluation of this function, the result file must be open and the specified result must exist in that file. This means that the appropriate actions which take care of that must be executed before the point of function evaluation (this is done in lines 14 and 15 of the example). The interface module contains functions which can return any fundamental result written in the result file of the analysis. These functions are installed in the expression evaluator module and can be used everywhere the evaluator can be used.

Command *"analysis"* defines the objective function. Its argument block is executed every time the objective function evaluation is required. In this case, it begins with specification of the analysis input file in line 10. The file is opened and ready for reading after the command *"setfile"* is executed. Information about the file are stored in local variables of the FEM-shell interface module (Figure 2), so that they can be accessed by functions which update the design parameters in the input file for direct analysis. Command *"initinput"* updates the data structures of the interface module which keep information about how to update the parameters in the analysis input file. This information is obtained from the input file in which the data which represents the investigated parameters are marked by user. Command *"setparam"* then updates the parameter values in the analysis input file using information obtained at execution of previous two commands. Parameter values are copied from the pre-defined vector variable *"parammom"* defined in the core module. This way, the transfer of parameter values between functions performing optimization algorithms and interface routines which transfer these values to FEM analysis is ensured.

Direct analysis is then run by command *"system"* which runs arbitrary program available on the system where the shell is running. Control is returned to the shell when the program run by the *"system"* command exits.

After the direct analysis is finished, results are collected and the objective function is evaluated. The analysis output file is specified and opened with the *"setfile"* command in the line 14 of the example. The information about the file is stored in local variables of the FEM-shell interface to be accessible for the interface functions which use this information. Before actual reading of analysis results, interface auxiliary data structures are reset by the *"initoutput"* command. These structures serve for speeding up reading of the analysis output data. The interface functions which read the data simultaneously store all already obtained information about the analysis output file to these structures, so that the information are available for consequent reads which could potentially need it. After the execution of the *"setfile"* and *"initoutput"* commands, the functions which read the analysis output data are ready to be used. In the lines 16 to 19, components of the pre-defined vector of simulated measurements *"measmom"* is set by the commands *"meas"*. The first argument of this command is the component which is set and the second argument is a mathematical expression, which is evaluated by the expression evaluator and its value assigned to the specified component of *"measmom"*.

The pre-defined global variables of the shell with specific meaning are used for data exchange between routines defined in different modules of the shell. Optimization algorithms sequentially evaluate the objective function at different value of parameters. This job is always done by running a specific function defined in the core module. The function requires the parameters vector as an argument and returns the value of the objective function. At its execution, the argument vector is copied to global variable *"parammom"* and then interpretation of argument block of the *"analysis"* command is performed by file interpreter. Commands written by users (in lines 10 to 12 of the example) ensure that input to direct analysis is updated according to these values before the analysis is run. At the end of the block, user programmes collecting of results. Both operations are carried out by functions of the FEM-shell interface and since these functions use the same pre-defined variables for parameters, measurements and the objective function value, the correct data transfer is ensured.

In the above example, it was not explicitly specified by user how the objective function should be calculated. In such case, the function is automatically evaluated using vectors *"sigma"*, *"meas"* and *"measmom"* according to the standard least-squares formula and the value is assigned to the pre-defined global variable *"objectivemom"* immediately after the interpretation of the *"analysis"* command's argument block. This value is returned by the general core function called for the objective function evaluation in the solution algorithms. In general case, user specifies the formula for objective function evaluation with the *"objective"* command with mathematical expression according to which the value is evaluated as an argument. In this expression, the functions *"getmeasmom"* of the expression evaluator is typically used. It returns components of the *"measmom"* vector which are evaluated in lines 16 to 19 of the above expression. Of course, different core function for objective function evaluation is used when sensitivities (Michaleris, 1994, Mroz, 1994) or Hessian matrix are evaluated beside the function value.

After the *"analysis"* command, commands which tell the system which calculations to perform, follow. In line 22, the *"analyse"* commands runs the general core function for the objective function evaluation with parameters which reside in global variable *"parammom"* at the moment of execution, and writes a report about execution in the shell's output file. Vector *"parammom"* was set by the *"setvector"* command in line 21. The *"analyse"* command is often used to ensure that the objective function was defined correctly.

At the end of the command file, inverse analysis is run by the *"inverse"* command. Arguments of this command tell the shell which minimization algorithm should be used, the initial guess, tolerance for function minimum, maximum number of iterations, etc. At execution of the *"inverse"* command, its arguments are read and the appropriate solution algorithm defined in the core module is run with these arguments. This coincides with the scheme in Figure 1. The steps which are within the optimization loop (except the first and the last, which belong to the core function which performs the algorithm), are executed when argument block of the *"analysis"* function is interpreted (this is triggered in the general core function for the objective function evaluation which is iteratively called at different parameter values within the solution algorithm). Step "prepare FEM input" coincides with lines 10 to 12 of the above example, step "run FEM analysis", which includes whole right half of the scheme, coincides with line 13, and step "read FEM analysis results and evaluate the objective functions" coincides with lines 14 to 19 of the example.

1 Flow Control

The example given above illustrated the minimum set of shell's commands necessary to run a simple inverse analysis. Only two available utilities of the core module were used (direct analysis at given input parameters and objective function evaluation). Solution strategy was straight-forward, using a standard sequence of algorithms foreseen by constructors of the shell. For a predicted set of different kind of problems, which can be defined in such simple way, a descriptive user interface of the shell would be sufficient and preferable for industrial use. On the other hand, this would impose serious limitations in the range of problems and solution strategies which can be defined by the shell. To impose enough flexibility at problem definition and choice of solution strategies, user must be as much as possible allowed to interfere with built-in utilities of the shell.

Some additional flexibility is imposed by allowing user to control the flow of the command file interpretation, i.e. to change sequence of execution with regard to the current state at a given point. This is achieved by additional set of functions which examine a condition, given as a mathematical expression, and decide what to do with regard to expression's current value evaluated with the expression evaluator. Conditional branching and loops can be made using such functions.

Basic branching command is *"if"* which has the following syntacs:

if { (*condition*) [*trueblock*] else [*falseblock*] }

At execution, the *condition* expression is evaluated with the expression evaluator. If its value is non-zero, the *trueblock* block is interpreted, otherwise the *falseblock* is interpreted.

Command *"while"* is used as a basic loop command. It's syntacs is

while { (*condition*) [*loopblock*] } .

At its execution, the *loopblock* block is interpreted as long as the *condition*'s value calculated each time is non-zero.

With the *"function"* command it is possible to define new commands which can be executed by file interpreter. Its syntacs is

function { *functionname* (*arg1*, *arg2*, ...) [*definitionblock*] }

functionname is a name of newly defined function which will be recognized by file interpreter from the definition point. *arg1*, *arg2*, etc. are names of arguments used in the *definitionblock* block. Where a command defined this way is called in the command file, the *definitionblock* of the function definition is interpreted. If an argument name (one of *arg1*, *arg2* etc.) preceded by the "$" character appears in this block, it is replaced by an actual argument of the command. For example, the for loop, which is not pre-defined, can be implemented using the function definition facility:

```
function { for (start,cond,whattodo,forblock)
[
  $start
  while { ($cond)
  [
    $forblock
    $whattodo
  ]}
]}
```

After this definition, we can for example print numbers from 1 to 5 to the standard output:

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

2 Integration of the Expression Evaluator in the System

As has been seen, the expression evaluator plays an important role in the shell. Functions of the FEM-analysis interface which read the analysis results are implemented as pre-defined functions of the expression evaluator. These functions can be directly combined in expressions which, for example, define the formulas for evaluation of simulated measurements as in the above example. Other important use is in the conditions for looping and branching commands.

Beside the standard arithmetic functions and the FEM-analysis interface' functions, the expression evaluator possesses functions which can access practically all data existing in the shell at a given point. Because of the global variables concept shown at the explanation of the above example, this allows user to strongly interfere through the command file with the solution strategies and built-in algorithms

Apart from the commands which always take expressions as arguments, expressions can be used instead of any numerical argument of any file interpreter's command. The following example shows how the components of pre-defined vector *"sigma"* can be set to 1 percent of the components of vector *"meas"*:

```
1.  setvector{ sigma  ${ getvector["meas",0]} }  *{ Sets the dimensin of vector sigma }
2.  while { ( i<= getvector["meas",0] )
3.     setvector{ sigma { ${i} : ${0.01*abs[getvector["meas",i]] } }
4.     ={ i : i+1 }
5.  }
```

In the first line, the dimension of vector *"sigma"* is set to the dimension of vector *"meas"* (command *"setvector"* called with single argument only sets vector's dimension). Expression in the curled brackets which follow the "$" sign is always replaced by its current value evaluated by the expression evaluator. In this case, the pre-defined function *"getvector"*, which returns a specific component of a specified vector (or its dimension if the component number is 0), is used. In the *"while"* loop, each component of vector "sigma" is set to 1 percent of absolute value of the appropriate component of vector *"meas"*.

The file interpreter possesses functions which affect the expression evaluator itself. Function *"="* assigns a value to an evaluator's variable. The first argument identifies the variable to which the value of the expression defined by the second argument is assigned. Both arguments are separated by colon. If the variable does not exist at the time when the command is called, it is created.

Function *"$"* is similar to *"="*, only that it assigns an expression instead of its value. The expression is not evaluated when this command is executed. It is even allowed to use variables which are not yet defined in the expression. For example, this code is legal:

```
        $ { a : sin[3*Pi/8] }
        = { Pi : 3.14 }
        write { ${a} \n }
```

Function *"write"* evaluates the expression in curled brackets following the "$" sign which is legal, because all sub-expressions can be evaluated at the point of execution. The *"$"* commands can also be used for the definition of new functions of the evaluator. In this case, names of function arguments separated by commas must be specified in squared brackets following the function name. The second argument of the command is in this case the expression which defines the function. If it includes names of function arguments, these are replaced by actual arguments (by expressions, not values) whenever the newly defined function is called within some other mathematical expression.

There is an additional mechanism of definition of the expression evaluator's functions which allows incorporation of interpretation of the command file's portions when the function is evaluated. function *"definefunction"* defines a new expression evaluator's functions which causes interpretation of a command file's block every time it is evaluated. The syntacs is

definefunction { (*functionname*) [*definitionblock*] }

The first argument of the command is function name in round brackets. The second argument is the block in a square bracket which is interpreted each time the function is evaluated. Function defined this way returns a value of the expression which is the argument of the last *"return"* command which is executed at the definition block interpretation. If no *"return"* command is executed during the interpretation of the definition block, function returns an unpredictable value. In the definition block functions *"argument"* and *"numargs"* of the expression evaluator can be used. Function *"argument"* returns the value of specific argument with which the defined function was called. *"numargs"* returns the number of arguments with which function was called. Such functions can be called with arbitrary number of arguments, but if the function *"argument"* with its argument greater then the actual number of arguments is evaluated within the definition block, it will return an unpredictable value. The example below shows the definition of function *"fsum"*, which returns the sum of all its arguments:

```
definefunction { (fsum)
[
  ={i:1} ={ret:0}
  while
  { ( i<=numargs() )
  [
    ={ret:ret+argument(i)}
    ={i:i+1}
  ]}
  return {ret}
]}
```

For example, the expression *"fsum[3,1,6]"* will evaluate to 10 below the definition of the function.

CONCLUSION

A shell for solving optimization and inverse problems is described. The shell's structure and user interface, which is implemented as a command file interpreter, ensure efficient flexibility for defining various problems and choosing different solution strategies. The presented concepts proved successful in many practical cases including optimization of prestressing of cold forging dies (Rodiè, 1996), inverse identification of hardening parameters in plasticity (Rodiè, 1995) as well as optimization pre-form shapes in forging and evaluation of friction factors and heat transfer coefficients from experimental tests.

There are several possibilities for further developments of the shell. One of important issues is incorporation of new solution algorithms, which is relatively straight-forward and does not require changes of conceptual scheme of the shell.

Parallelization is another important issue. This means simultaneous execution of several direct analyses at different input parameters on different processors. Such parallelization would be independent from the parallelization of the finite element code and would additionally improve the effectiveness. It would ease the fall of effectiveness with increasing number of processors, which is a well known problem in parallelization of FEM codes. Currently, the preparation of conceptual background including protocols is in progress. The next step will be parallelization of the solution algorithms.

Development of general concepts for sensitivity analysis will be necessary in the future. A part of this work will be done in the FEM analysis code, but generalization of sensitivity analysis will also greatly affect the concepts of future FEM-shell interfaces. Recently, some work has been initiated on integration of symbolic systems such as SMS (Korelc, 1997) with FEM code.

Establishing a connection between the shell and a symbolic system will probably prove beneficial in the future. It would impose additional flexibility to the user interface of the shell by allowing symbolic differentiation and other operations on expressions defined in the expression evaluator. These operations could be implemented in the expression evaluator itself, but use of an existing symbolic system would ease the job significantly.

Suitability of the system for industrial use will have to be carefully considered in the future. One of the most important conditions for widely spread use of the system in industry is that it is easy-to-use. This concerns system's user interface at most. At the current stage, user defines an inverse or optimization problem in two separated steps. At the first step, direct problem is defined using the FEM system's preprocessor. This definition is actually a sceleton of the problem definition because real direct problem changes through the solution procedure according to the parameters. The second step is writing of the shell's command file.

It would be preferable to define shell's behaviour at the same place as the direct problem, but this would at the same time mean that concepts of both definitions should be similar. Descriptive way of problem definition as is used in typical FEM system's preprocessor would conflict with the demand for flexibility at the problem definition. One solution of this problem would be to make pre-defined forms with problem definitions for a few specific sets of problems which frequently appear in some areas. In these forms, only data should be replaced for the definition of different problems and this could be done using a possibly graphic preprocessor with traditional look.

ACKNOWLEDGMENT

Financial assistance for this work, provided by Ministry of Science and Technology, Republic of Slovenia under the contract number P23-1533/0112-005/15827/95, and INCO Copernicus under the contract number IC15-CT96-0709, is gratefully acknowledged.

REFERENCES

1. FLETCHER, R. (1987), *Practical Methods of Optimization (second edition),* John Wiley & Sons, New York.
2. GREŠOVNIK, I. (1996), "A Computer System for Solving Inverse Problems", in Štok, B. (Ed.), *Kuhljevi Dnevi 96,* Slovensko društvo za mehaniko, Ljubljana, p.p. 97-104, in Slovene.
3. KORELC, J. (1997), "Automatic Generation of Finite-element Code by Simultaneous Optimization of Expressions", *Theoretical Computer Science,* Vol. 187.
4. MICHALERIS, P., TORTORELLI, D. A., VIDAL, C. A. (1994), "Tangent Operators and Design Sensitivity Formulations for Transient Non-linear Coupled Problems with Applications to Elastoplasticity", *International Journal for Numerical Methods in Engineering,* Vol. 37, pp. 2471-2499.
5. MROZ, Z. (1994), "Variational methods in sensitivity analysis and optimal design", *Europ. J. Mech. A/Solids,* Vol. 13, pp. 115-149.
6. OWEN, D., HINTON, E. (1980), *Finite Elements in Plasticity,* Pineridge Press, Swansea.
7. RODIÈ, T., GREŠOVNIK, I. (1996), "Optimization of the Prestressing of a Cold Forging Tooling system", in Delaunay, D. (Ed.), *Preliminary Proceedings of the second International Conference on Inverse Problems in Engineering,* Engineering Foundation, New York, Vol. 2.
8. RODIÈ, T., GREŠOVNIK. I. and OWEN, D.R.J. (1995), "Application of error minimisation concept to estimation of hardening parameters in the tension test", in Owen, D.R.J. and Onate, E. (Eds.), *Proceedings of the Fourth International Conference: Computational Plasticity - Fundamentals and applications,* Pineridge Press, Swansea, Vol. 1, pp. 779-787.
9. ZIENKIEWICZ, O. C., TAYLOR, R. (1991), *The Finite Element Method, vol. 2 (fourth edition),* McGraw-Hill, London.