

## Chapter 2

# Automation of Research in Computational Modeling

*It is obvious that if we could find characters or signs suited for expressing all our thoughts as clearly and as exactly as arithmetic expresses numbers or geometry expresses lines, we could do in all matters insofar as they are subject to reasoning all that we can do in arithmetic and geometry.*

Gottfried Wilhelm Leibniz  
Preface to the General Science, 1677

The advances in reliability, generality and interdisciplinary nature of new computational methods derived in recent years are primary result of a holistic approach to computational modeling where advanced software tools and techniques are combined with advanced numerical methods. The holistic approach is playing nowadays a central role in the process that leads to the ultimate goal, i.e. a complete automation of computational modeling. The problem of an automation of the derivation of computational models has been explored by researches from the fields of mathematics, computer science and computational mechanics, resulting in a variety of approaches (e.g. object-oriented approaches, domain specific languages and a hybrid symbolic-numeric methods) and available software tools (e.g. symbolic and algebraic systems, automatic differentiation tools, problem solving environments and numerical libraries). Automation can address all steps of the finite element solution procedure from the strong form of boundary-value problem to the visualization of results, or it can be applied only to the automation of the selected steps of the whole procedure. Consequently, the automation procedures nowadays influence directly how the mechanical problem and corresponding numerical model are postulated and solved.

Alternative approaches to automation of computational modeling are discussed in the first part of the chapter, while an emphasis is given to automatic differentiation. The hybrid symbolic-numeric approach to automation of the finite element method and the system for automatic code generation *AceGen* are described in detail in the second part of the chapter.

## 2.1 Introduction

**Finite element technology.** As demonstrated throughout this book, there are almost countless ways of how a particular problem can be solved by the finite element method. The complete finite element simulation can be, from the aspect of automation, decomposed into the following steps:

1. formulation of the strong form of an initial boundary-value problem;
2. transformation of the strong form into a weak form or a variational functional;
3. definition of the discretization of the domain and approximation of the field variables and their virtual counterparts (test functions);
4. derivation and solution of additional algebraic equations or differential equations defined at the element level (e.g. constraints or plastic evolution equations);
5. derivation of algebraic equations that describe the contribution of one element to the global residual vector and to the global tangential stiffness matrix;
6. coding of the derived equations in a required computer language;
7. generation of a finite element mesh and its boundary conditions;
8. solution of the global problem;
9. visualization and analysis of the results.

Alternatively, one can also start from the free Helmholtz energy of the problem and derive element equations directly as a gradient of the free energy. This approach is especially appealing for the automation due to the numerical efficiency of the solution when the gradient is obtained by the backward mode of automatic differentiation (see Sect. 2.4). The presented list follows from the engineering approach related to finite element technology where requirements for highly efficient elements (e.g. mixed, under-integrated, etc.) and the variety of elements (etc. solid, structural, coupled) lead to a rather strict separation of the individual finite element level and the global level of the problem.

With respect to the software organization the procedure of solving a problem by the finite element method can be divided into two parts:

1. part independent of the actual physical problem or the *finite element environment*,
2. and the part dependent on the physical problem or the *finite elements*.<sup>1</sup>

Finite elements are provided in open source finite element environments through user defined finite element subroutines or *user subroutines*. A particular finite element subroutine is meant to calculate the contribution of the particular finite element to the global solution of the problem. In the present book only the automation of the generation of user subroutines is presented.

**Automation.** If the automation of all steps of a finite element process from the strong form of an initial boundary-value problem to the visualization and analysis of results is chosen, then only a very specific subset of possible formulations can be covered.

---

<sup>1</sup>Here the particular finite element is meant to solve a particular physical problem. Large commercial finite element environments can include several hundreds of different finite elements for different physical applications.

Usually, only the standard spatial discretization (see Chap. 4) is considered as presented in Logg (2007). On the other hand, the standard discretization is of little use for problems involving coarse mesh accuracy, locking phenomena and distorted element shapes for which highly problem specific formulations have to be implemented as described in Sects. 6.4 and 6.5. As usual in science, the high uniqueness of a specific formulation renders the whole concept of automation questionable. Making templates or deriving objects for something that is used only once simply does not pay off. This may be the main reason why a complete automation of the finite element method is still not used within the commercial finite element environments. More often, the level of automation used involves only steps that are from the numerical aspect deterministic (e.g. various correctness preserving symbolic manipulations, differentiation and automatic code generation) while the true decisions are left to the researcher.

The automation of computational modeling can be approached having in mind two fundamentally different goals. The automation procedure can aim at the

1. automation of the solution of physical or mathematical problems,
2. or at the automation of scientific research in developing the optimal computational approach for the solution of physical or mathematical problems.

Usually, the difference between the standard description and the algorithmic implementation prevents full automation of scientific research in computational methods. When the automation of the solution is sought then the description and the algorithmic implementation are done within the same framework and the problem is less limiting. However it is questionable whether a complete automation is achievable at the present scientific level. New methods, different ways of discretization, optimal variational principles and optimal algorithmic implementation are still very much under research. While the automation of the solution results in software solutions that are in many ways under control of the software developer the automation of the research into solution requires much higher level of interaction between the user of the software solution and the actual process of automation. In particular, debugging possibilities have to be included in an automatic code generator that can provide a backward link between the automatically generated code and the basic equations of the mathematical model.

### ***2.1.1 Abstract Symbolic Description of a Computational Model***

A general requirement for a successful use of automation system is that the basic equations defining the problem have to be written down in an input form that is capable of manipulating equations symbolically. The advantage of automation in computational mechanics becomes apparent only when the description of the problem, which means the way how the basic equations are written down, is appropriate for the symbolic description. When this condition is not met then the automation

of the symbolic description of a computational model may require more effort than the classical, well establish approaches. Unfortunately, some of the traditional formulations used in computational mechanics are not appropriate for the symbolic description. The symbolic formulation of computational mechanics problems differs often from the classical formulations described in detail in other chapters of this book and thus brings up the need for rethinking and reformulating known and traditional ways. Despite that, there exist strong arguments why at the end symbolic formulations are indeed beneficial:

1. A symbolic formulation is more compressed and thus gives fewer possibilities for an error.
2. Algebraic operations, such as differentiation, are done automatically.
3. Automatically generated codes are highly efficient and portable.
4. The multi-language and multi-environment capabilities of symbolic systems enable generation of numerical codes for various numerical environments from the same symbolic description.
5. An available collection of prepared symbolic inputs for a broad range of finite elements can be easily adjusted for the user specific problem leading to the on-demand numerical code generation.
6. Multi-field and multi-physic problems can be easily implemented. For example, the symbolic inputs for mechanical analysis and thermal analysis can be combined into a new symbolic input that creates a finite element for a fully coupled and quadratically convergent thermo-mechanical analysis.

An example, underlining the arguments above, is the standard formulation of the tangential stiffness matrix, leading to the matrix form  $\mathbf{B}^T \mathbf{D} \mathbf{B}$ . It can be easily implemented using the symbolic tools. Having in mind that element tangential stiffness matrix is either the Jacobian of the resulting system of discrete algebraic equations or the Hessian of the variational functional, it means that automatic differentiation is sufficient for obtaining the tangent matrix. The work of implementing a  $\mathbf{B}^T \mathbf{D} \mathbf{B}$  matrix formulation and the efficiency of the resulting code is inferior to the approach when the tangent matrix is derived by backward automatic differentiation. The latter approach requires, regardless on the complexity of the topology and the material model, a single line of symbolic input. The standard  $\mathbf{B}^T \mathbf{D} \mathbf{B}$  formulation requires much more input for the same result.

Modern finite element simulations are often coupled with optimization procedures that require additionally to the solution of the primal problem also a solution of the sensitivity problem. The aim of sensitivity analysis is to calculate derivatives of arbitrary response functionals with respect to chosen parameters (see e.g. Kleiber et al. 1997; Keulen et al. 2005; Choi and Kim 2005a or 2005b). Sensitivity analysis has become an indispensable part of modern computational algorithms. The use of an analytically exact sensitivity analysis can significantly improve optimization procedures, see Choi and Kim (2005b), Kristanic and Korelc (2008), multi-scale algorithms, see Solinc and Korelc (2015), and the implementation of nonlinear material models, see Korelc and Stupkiewicz (2014), Hudobivnik and Korelc (2016). Thus, any proposed method of automation should address automation of primal as well

as sensitivity analysis. However, to the authors knowledge, no large commercial code currently provides a general sensitivity analysis tool. Automation of sensitivity analysis is presented in Chap. 8.

The automation of the finite element methods should not be restricted only to the repetition of the same procedures that are normally done manually on a sheet of paper. However, the ability to describe the problems in terms that are more general does not necessarily mean that the derivation of the specific computational model would be any easier. The true advantages of automation become apparent only if the description of the problem, the notation and the mathematical apparatus are changed as well. It is demonstrated in the book that this can be achieved using the automatic differentiation technique as presented in Sect. 2.4.2. Thus, the basis for the automation of computational modeling is an automatic differentiation based form or ADB form of the basic equations used to describe the problem (see Chap. 3). In the actual implementation of the described methodology a general-purpose automatic code generator *AceGen* Korelc (2011) is used to derive and code characteristic finite element quantities (e.g. residual vector and stiffness matrix) at the level of an individual finite element. This code produces user defined elements for different finite element environments. However there exists also the general-purpose finite element environment *AceFEM* for the solution of the global problem that is tailored for the use together with *AceGen*.

## 2.2 Advanced Software Tools and Techniques

The use of advanced software technologies plays a central role in the process that leads to the ultimate goal: a complete automation of computational modeling. The problem of automation of computational methods has been explored by researchers from the fields of mathematics, computer science and computational mechanics, resulting in a variety of approaches (e.g. the hybrid object-oriented approach by Eyheramendy and Zimmermann 2000 and the hybrid symbolic-numeric approach by Korelc 2002) and available software tools (e.g. computer algebra systems, AD tools by Griewank 2000, problem solving environments and numerical libraries).

The techniques presented in the next sections are the result of rapid development in computer science in the last decades. They are particularly relevant for the description of a nonlinear finite element model on a high abstract level, while preserving the numerical efficiency.

### 2.2.1 Symbolic and Algebraic Computational Systems

Symbolic and algebraic computational (SAC) systems are tools for the manipulation of mathematical expressions in symbolic form. Widely used SAC systems such as *Mathematica* ([www.wolfram.com](http://www.wolfram.com)) or *Maple* ([www.maplesoft.com](http://www.maplesoft.com)) have become an

integrated computing environment that covers all aspects of a computational process, including numerical analysis and graphical presentation of the results. The general SAC systems are also one of the most complex software systems ever developed and the SAC system *Mathematica* is often described as the “world’s single largest consumer of algorithms”. In the case of complex mechanical models, the direct use of SAC systems is not possible due to several reasons. For the numerical implementation, SAC systems cannot keep up with the run-time efficiency of programming languages such as FORTRAN and C and by no means with highly problem-oriented and efficient numerical environments used for finite element analysis. However, SAC systems can be used for the automatic derivation of appropriate formulas and generation of numerical codes (Korelc 2002; Amberg et al. 1999). The FE method is within the general SAC systems usually implemented as an additional package or toolbox such as *AceGen* Korelc (2011) for *Mathematica*.

The major limitation of the symbolic systems, when applied to complex engineering problems, as pointed out before by many authors (see e.g. Wang 1986; Fritzson and Fritzson 1984; Korelc 1997b and Korelc 2002), is an uncontrollable growth of expressions and consequently redundant operations and inefficient codes. This is especially problematic when a SAC system is used to derive formulas needed in numerical procedures such as the finite element method where the numerical efficiency of the derived formulas and the generated code are of utmost priority.

General SAC systems are very powerful when dealing with one isolated formula, but become very awkward when the code to be generated contains control structures such as If and Do constructs.

### 2.2.2 Automatic Differentiation Tools

Differentiation is a symbolic operation that plays a crucial role in the development of new numerical procedures. Automatic differentiation or AD is a method to evaluate the derivative of a function specified by a computer program and represents an alternative to more common and widely used numerical differentiation and symbolic differentiation. The AD technique is explained more in detail in Sect. 2.4 due to the central role of AD in automation of the finite element method.

### 2.2.3 Problem Solving Environments

Problem-solving environments (PSE) are automatic code generators with libraries containing routines for various numerical solution methods. These routines form templates for the generated program codes. The system libraries determine a variety of numerical solution methods available in such systems. They are meant to solve problems, in particular ordinary differential equations or partial differential equations, in an already established way. Several problem-solving environments for

a high level abstract description of PDE's based on finite difference method have been derived, such as *SciNapse* (Akers et al. 1998) and *Ctadel* (van Engelen et al. 1995). A comprehensive overview can be found in Gallopoulos et al. (1994).

**Numerical libraries.** Additionally to the general problem-solving environments, there are also tools that support only numerical operations such as: compiled numerical libraries (e.g. *NAG*, [www.nag.co.uk](http://www.nag.co.uk)), numerical matrix languages (e.g. *MATLAB*, [www.mathworks.com](http://www.mathworks.com)), high-level object oriented languages with object libraries, etc.

**Specialized systems for FEM.** General finite element environments, such as such as commercial codes like *ABAQUS* ([www.hks.com](http://www.hks.com)) and *ANSYS* ([www.ansys.com](http://www.ansys.com)) or research codes like *FEAP* ([www.ce.berkeley.edu/feap](http://www.ce.berkeley.edu/feap)), can also be viewed as a specialized PSE. The general finite element environments can handle, regardless of the type of finite elements, all the phases of a typical FE simulation: pre-processing of the input data, manipulation and organization of the data related to nodes, elements, material characteristics, displacements and stresses, construction of the global matrices by invoking different element subroutines, solving the system of equations, post-processing and analysis of the results.

### 2.2.4 Hybrid Approaches

The level of automation of FE method can be greatly increased by combining several approaches and tools. Some possible combinations are discussed below.

**Hybrid object-oriented approach.** The object-oriented approach has brought a new perspective for the development of complex software and in the past decade, numerous object-oriented FE environments have been developed. While the object-oriented approach deals primary with the high level of data abstraction and organization, its principles can be extended also to the complete automation of the finite element method. An overview of the object-oriented hybrid symbolic-numerical approach can be found in Eyheramendy and Zimmermann (2000), Beall and Shephard (1999) and Logg (2007). Modern hybrid object-oriented (HOO) systems, such as *FEniCS* (Logg 2007; Logg and Wells 2012) or *FreeFem++* (Pironneau et al. 2008), provide tools for automation of all FE simulation steps, from the strong form of a PDE to the presentation of the results. A typical HOO system introduces its own domain-specific language and uses built-in C++ libraries for symbolic manipulation. The HOO systems are in general restricted to a particular type of formulations where the general knowledge of the appropriate procedure, that leads from strong form to the element equations, has already been established. This also reduces the expression growth problem since the symbolic code derivation is used only for sub-problems.

**Hybrid symbolic-numerical approach.** The disadvantage of the hybrid object-oriented approach is the loss of generality and flexibility compared to a general

computed algebra systems. Only a small fraction of symbolic manipulation capabilities of the general SAC systems is presented in specialized finite element C++ libraries for symbolic manipulation. The real power of the symbolic approach for testing and applying new, unconventional ideas is provided by general-purpose SAC systems. However, their use is limited for problems that lead to large systems like finite element simulations. Furthermore, the use of large commercial finite element environments for analyzing a variety of problems is standard practice of engineers. The hybrid symbolic-numerical (HSN) approach is a way to combine both. While the hybrid object-oriented systems tends to offer complete FE solution, the idea behind the hybrid symbolic-numerical (HSN) approach is to use a general SAC system for the derivation of the characteristic element quantities and the automatic code generation of user subroutines at the level of one finite element. The automatically generated code is then incorporated into the chosen finite element environment (one or possibly more) and used within the global numerical solution procedures. The HSN approach is explained in more detail in Sect. 2.5.1. An advantage of using a general SAC system is also that it provides well known and defined description language for the derivation of FE equations, generation of FE code and possibly also for the complete FE analysis, as opposed to the hybrid object-oriented systems which introduce their own domain-specific language.

## 2.3 Automatic Generation of Numerical Codes

Most of the existing numerical methods for solving partial differential equations can be subdivided into two classes: finite difference and finite element methods. In the last years, various approaches to the automation of the two methods were extensively studied. In many ways, the present stage of the automation of the finite difference method is more elaborated and more general than the automation of the finite element method. Various transformations, differentiation, matrix operations, and a large number of degrees of freedom involved in the derivation of characteristic finite element quantities often lead to exponential growth of the expressions in space and time, see e.g. Fritzson and Fritzson (1984). This makes automation of the FE method more complex than automation of the finite difference method. Different approaches have been proposed for the solution of the problem related to expression growth. In the first approach additional low-level knowledge about the problem is introduced and the symbolic code generation is used only for sub-problems, where expression growth does not appear or is not severe, see e.g. Amberg et al. (1999). A pure symbolic approach is often combined with the object-oriented approach and specialized routines for symbolic manipulations of the general CA systems are used instead. An extensive overview of object-oriented hybrid symbolic/numerical approaches can be found in Eyheramendy and Zimmermann (2000) and in Beall and Shephard (1999). The disadvantage of the approach is the loss of generality and flexibility of general CA systems. These specialized systems are often restricted to a particular type of problems where general knowledge of the solution procedure



has already been established. The real power of the symbolic approach for testing and applying new, unconventional ideas lies in general purpose systems. Not many attempts have been undertaken to produce a general FE code generator where the key issue of the FE code generation, i.e. expression swell, is treated within the automatic procedure. Techniques such as the use of the symmetric properties of the formulae, the automatic introduction of intermediate variables and the pattern search were applied in Finger Wang (1986), Wang (1991) and Tan et al. (1991).

When the symbolic approach is used in a standard way to describe complex-engineering problems then the common experience of the users of symbolic and algebraic systems is an uncontrollable swell of expression, as pointed out before, leading to inefficient or even unusable codes. The general computer algebra systems come with the built in code optimization capabilities, see e.g. *Maple*, or additional packages for code optimization such as *AceGen* for *Mathematica*. The classical way of optimizing expressions in computer algebra systems is searching for common sub-expressions after all the formulae have been derived and before the generation of the numerical code. This has been proven to be insufficient when applied to general non-linear mechanical problems and only relatively simple finite elements can be derived within such approach.

An alternative approach for automatic code generation is employed in *AceGen* and called Simultaneous Stochastic Simplification of numerical code, see Korelc (1997b). This approach avoids the problem of expression swell by using symbolic and algebraic capabilities of the general computer algebra system *Mathematica* and combining them with the following techniques: automatic differentiation, simultaneous optimization of expressions with automatic selection and introduction of appropriate intermediate variables. Formulae are optimized, simplified and replaced by auxiliary variables simultaneously with the derivation of the problem. A stochastic evaluation of the formulae is applied for determining the equivalence of algebraic expressions, see e.g. Gonnet (1986). The simultaneous approach is appropriate also for problems where intermediate expressions can be subjected to uncontrolled swell.

Using the general finite element environment for analyzing a variety of problems and for incorporating new elements is now already an everyday practice. Although large FE environments often offer a possibility to incorporate user defined elements and material modes, it is time consuming to develop and test these user defined new pieces of software. Practice shows that at the derivation stage of a new numerical model, different languages and different platforms are the best means for the assessment of specific performances and failures of the numerical model. The basic tests, which are performed on a single finite element or on a small patch of elements, can be done most efficiently by using general computer algebra system. Many design flaws, such as element instabilities or poor convergence properties, can be easily identified, if the element quantities are investigated on a symbolic level. Unfortunately a stand-alone CA system becomes very inefficient once there is a larger number of nonlinear finite elements to process or if iterative numerical procedures have to be executed. In order to assess element performances under real conditions, the easiest way is to run the necessary test simulations on sequential machines with good debugging capabilities and with the open source FE environment designed for research purposes,

e.g. *FEAP* ([www.ce.berkeley.edu/rlt/feap/](http://www.ce.berkeley.edu/rlt/feap/)), *AceFEM* ([www.fgg.uni-lj.si/symech/](http://www.fgg.uni-lj.si/symech/)) or *Diffpack* ([www.diffpack.com](http://www.diffpack.com)). At the end, for real industrial simulations involving complex geometries a large commercial FE environment has to be used. In order to meet all these demands in an optimal way, an approach is needed that would offer multi-language and multi-environment generation of numerical codes. The automatically generated code is then incorporated into the FE environment that is most suitable for the specific step of the research process. Using the classical code development procedures, re-coding of the element in different languages would be time consuming and is rarely done. With the general SAC systems, re-coding comes practically for free, since the code can be automatically generated for several languages and for several platforms using the same basic symbolic description.

## 2.4 Automatic Differentiation

The classical way of performing differentiation is to symbolically differentiate a function either manually or by a computer algebra system and to evaluate it at a chosen point. In the case of complex computational models, the symbolic derivatives are difficult to obtain, which is why the numerical differentiation by the finite difference method is often used instead. Automatic differentiation (see e.g. Griewank and Walther 2008; Bartholomew-Biggs et al. 2000; Bischof et al. 2002) is a method to compute the derivative of a function specified by a computer program. It represents an alternative solution to the numerical differentiation as well as to the symbolic differentiation. With the AD technique, one can avoid the problem of expression growth that is associated with the symbolic differentiation performed by a SAC system.

### 2.4.1 Principles of Automatic Differentiation

Automatic differentiation techniques are based on the fact that every computer program executes a sequence of elementary operations with known derivatives, thus allowing evaluation of exact derivatives via the chain rule for an arbitrary complex formulation. If one has a computer code which allows to evaluate a function  $f$  and needs to compute the gradient  $\nabla f$  of  $f$  with respect to arbitrary variables, then the automatic differentiation tools, see e.g. Griewank (2000), Bartholomew-Biggs et al. (2000), Bischof et al. (2002) can be applied to generate the appropriate program code.

There are two approaches for the automatic differentiation of a computer program, often characterized as the forward and the backward mode of automatic differentiation. The procedure is illustrated by a simple example of a function  $f$  defined by

$$f = b c \quad \text{with} \quad b = \sum_{i=1}^n a_i^2 \quad \text{and} \quad c = \sin(b) \quad (2.1)$$

where  $a_1, a_2, \dots, a_n$  are  $n$  independent variables. The forward mode accumulates the derivatives of intermediate variables with respect to the independent variables as follows

$$\begin{aligned}\nabla b &= \left\{ \frac{db}{da_i} \right\} = \{2 x_i\} & i = 1, 2, \dots, n \\ \nabla c &= \left\{ \frac{dc}{dx_i} \right\} = \{Cos(b) \nabla b_i\} & i = 1, 2, \dots, n \\ \nabla f &= \left\{ \frac{df}{da_i} \right\} = \{ \nabla b_i c + b \nabla c_i \} & i = 1, 2, \dots, n\end{aligned}\quad (2.2)$$

Contrary to the forward mode, the backward mode propagates adjoints  $\bar{x} = \frac{\partial f}{\partial x}$ , which are the derivatives of the final values, with respect to intermediate variables:

$$\begin{aligned}\bar{f} &= \frac{df}{df} = 1 & 1 \\ \bar{c} &= \frac{df}{dc} = \frac{\partial f}{\partial c} \bar{f} = b \bar{f} & 1 \\ \bar{b} &= \frac{df}{db} = \frac{\partial f}{\partial b} \bar{f} + \frac{\partial c}{\partial b} \bar{c} = c \bar{f} + Cos(b) \bar{c} & 1 \\ \nabla f &= \{\bar{a}_i\} = \left\{ \frac{\partial b}{\partial a_i} \bar{b} \right\} = \{2 a_i \bar{b}\} & i = 1, 2, \dots, n.\end{aligned}\quad (2.3)$$

The numerical efficiency of the differentiation of  $N$  scalar-valued functions  $\mathbf{F} = \{f_i : i \in [1, N]\}$  with respect to  $M$  independent variables  $\mathbf{a} = \{a_j : j \in [1, M]\}$  can be measured by the numerical work ratio defined as

$$wratio(\mathbf{F}(\mathbf{a})) = \frac{cost(\mathbf{F}(\mathbf{a}), \frac{\partial \mathbf{F}}{\partial \mathbf{a}})}{cost(\mathbf{F}(\mathbf{a}))} \quad (2.4)$$

The ratio is in general proportional to the number of independent variables ( $wratio(\mathbf{F}(\mathbf{a})) \propto M$ ) in the case of forward mode and proportional to the number of functions ( $wratio(\mathbf{F}(\mathbf{a})) \propto N$ ) in the case of backward mode. The upper bound for the ratio in the case of backward mode differentiation of one function with respect to arbitrary number of independent variables is 5 and is usually around 1.5 if care is taken in handling quantities that are common to the function and gradient, see e.g. Griewank (2000). Although numerically superior when the number of functions is small, the backward mode requires potential storage of a large amount of intermediate data during the evaluation of the function that can be as high as the number of numerical operations performed. Additionally, a complete reversal of the program flow is required. This is because the intermediate variables are used in reverse order when related to their computation. For the efficient automation of the FE method, it is desirable that both approaches are available and that the software tool used for automation can automatically select the most efficient approach according to the estimated work ratio.

It should be pointed out that the symbolic differentiation is one of the algebraic operations prone to severe expression growth and it can result even for relatively simple nonlinear elements in hundreds of pages of code. Thus, the use of hybrid system that combines the symbolic tool with the automatic differentiation technique

is essential for the high abstract symbolic formulation of FE models. To increase the numerical efficiency of the generated code and to limit the physical size of the generated code, it is essential to minimize the number of calls to the automatic differentiation procedure.

There exist many strategies how the AD procedure can be implemented, see e.g. Bischof et al. (2002). The simplest approach is to use operator overloading and during the evaluation of function  $f$  create a trace of all numerical operations and their arguments, later used to evaluate gradients in forward or backward mode. The operator overloading strategy is computationally too inefficient to be used within the finite element procedures. More efficient is a source-to-source transformation strategy that transforms the source code for computing a function into the source code for computing the derivatives of the function. The AD tools based on a source-to-source transformation have been developed for most of the programming languages e.g. *ADIFOR* ([www-unix.mcs.anl.gov/autodiff/ADIFOR/](http://www-unix.mcs.anl.gov/autodiff/ADIFOR/)) for FORTRAN, *ADOL-C* ([www.math.tu-dresden.de/~adol-c/](http://www.math.tu-dresden.de/~adol-c/)) for C, *MAD* ([www.amorg.co.uk/AD/MAD/](http://www.amorg.co.uk/AD/MAD/)) for Matlab and *AceGen* ([www.fgg.uni-lj.si/symech](http://www.fgg.uni-lj.si/symech)) for Mathematica.

### 2.4.2 Generalized Notation of Automatic Differentiation

Recent advances in the development of automatic differentiation technique, especially the backward mode implementation of the code-to-code approach to automatic differentiation, have rejected the traditional assumption that automatic differentiation is impracticable and that the numerical codes based on automatic differentiation are intrinsically too slow for large-scale numerical computations. However, as powerful as automatic differentiation technique is, the results of the automatic differentiation procedure might not automatically correspond to the specific mathematical formalism used to describe the mechanical problem. One of the primal goals of the present book is to introduce an appropriate notation that builds a bridge between the classical mathematical notation of computational models and the actual computer implementation. It is essential for the generalized notation that it can be directly translated into the program code and that the derived program code is numerically efficient. The idea presented in the book is to achieve the unification by means of extended automatic differentiation technique combined with the automatic code generation. The introduced notation does not only simplify the derivation of the corresponding equations, but also reflects much more closely the actual algorithmic implementation. In this way, the mathematical formulation and computer implementation become indistinguishable.

The introduced generalized notation is comprised of three components that:

- define the mathematical operation,
- indicate the algorithm used to obtain the quantity and
- specify the flow of information within the algorithm.

Let  $\mathbf{a}$  be a set of mutually independent variables and  $f$  an arbitrary function of  $\mathbf{a}$ . The “computational derivative” is then defined by the following formalism

$$\frac{\hat{\delta} f(\mathbf{a})}{\hat{\delta} \mathbf{a}} \quad (2.5)$$

The above formalism has to be viewed in an algorithmic way. The operator  $\frac{\hat{\delta} f(\mathbf{a})}{\hat{\delta} \mathbf{a}}$  represents a differentiation of a function  $f$  with respect to variables  $\mathbf{a}$  performed by the automatic differentiation algorithm. Thus, in the context of the book the operator has a dual purpose. It indicates the mathematical operation of differentiation as well as it indicates the algorithm used to obtain the required quantity. It also denotes that, in the process of derivation, the AD procedure has been called. The mathematical formalisms that are part of the traditional FE formulation such as partial derivatives  $\frac{\partial(\bullet)}{\partial(\bullet)}$ , total derivatives  $\frac{D(\bullet)}{D(\bullet)}$ , directional derivatives, consistent derivatives etc., can all be represented by the AD procedure. However, the result of an AD procedure might not automatically correspond to any of the above mathematical formalisms. Thus, there exists a need for an extended functionality of the traditional AD procedure as well as a need for an extended notation that would correspond to an extended functionality. This extended functionality can be provided by introducing additional information used within the process of automatic differentiation. It thus defines exceptions within the AD procedure.

Let  $\mathbf{b}$  be a set of mutually independent intermediate variables that are part of the evaluation of a function  $f$ .  $\mathbf{f}_b$  is a set of arbitrary functions of  $\mathbf{a}$  such that  $\mathbf{b} := \mathbf{f}_b(\mathbf{a})$ , and  $\mathbf{M}$  is an arbitrary matrix. AD exceptions are then introduced by the following formalism

$$\left. \frac{\hat{\delta}(\bullet)}{\hat{\delta}(\bullet)} \right| \frac{D\mathbf{b}}{D\mathbf{a}} = \mathbf{M} \quad (2.6)$$

where  $\frac{\hat{\delta}(\bullet)}{\hat{\delta}(\bullet)}$  stands for the various forms of AD operators that will be introduced later. Notation (2.6) indicates that during the AD procedure, the total derivatives of an arbitrary set of mutually independent intermediate variables  $\mathbf{b}$  with respect to independent variables  $\mathbf{a}$  are set to be equal to the matrix  $\mathbf{M}$ . The AD exceptions can be viewed as a bridge inside the chain-rule that goes directly from  $\mathbf{b}$  to  $\mathbf{a}$ .

In a special case,  $\mathbf{M}$  represents the actual total derivatives of  $\mathbf{b}$  with respect to  $\mathbf{a}$ . Thus  $\mathbf{M} = \frac{D\mathbf{b}}{D\mathbf{a}}$ , however  $\mathbf{M}$  can not be derived from the actual program code considered by the AD procedure. The matrix  $\mathbf{M}$  has to be in this special case calculated by an alternative computational procedure or imported as an input data of the problem. The situation is indicated by the following formalism

$$\left. \frac{\hat{\delta}(\bullet)}{\hat{\delta}(\bullet)} \right| \frac{D\mathbf{b}}{D\mathbf{a}} = \widehat{D_{\mathbf{a}}\mathbf{b}} \quad (2.7)$$

The formalism  $\frac{D\mathbf{b}}{D\mathbf{a}} = \widehat{D_{\mathbf{a}}\mathbf{b}}$  in (2.7) denotes that the actual total derivatives of  $\mathbf{b}$  with respect to  $\mathbf{a}$  were obtained by an alternative procedure, stored in a matrix  $\widehat{D_{\mathbf{a}}\mathbf{b}}$  and used during the AD procedure.

When no AD exceptions are defined for any of the intermediate variables that appear as part of evaluation of function  $f$ , then the computational derivative (2.5) yields the same result as the partial derivative, thus

$$\frac{\hat{\delta}f(\mathbf{a})}{\hat{\delta}\mathbf{a}} = \frac{\partial f(\mathbf{a})}{\partial \mathbf{a}}. \quad (2.8)$$

The use of the proposed AD formalism instead of the standard partial derivative formalism also indicates that the AD procedure is being used to obtain the result. Thus, the  $\frac{\hat{\delta}(\bullet)}{\hat{\delta}(\bullet)}$  formalism is used instead of the  $\frac{\partial(\bullet)}{\partial(\bullet)}$  formalism wherever the automation of the specific computational procedure is proposed.

If the computer program that defines the function is relatively simple, the AD can be applied at the global level of the program. More often, the computer program is complex (e.g. finite element environment) with some parts available only as compiled libraries and thus unavailable for automatic differentiation. In that case the AD can still be applied at the level of particular subroutines (e.g. finite element subroutines). When the AD procedure is used within the subroutine then the AD procedure recognizes only dependencies that appear explicitly within the derived subroutine. Thus, all the input parameters of the subroutine are automatically considered as independent variables with all derivatives set to zero. Any dependency of the input parameters of the subroutines has to be explicitly introduced as an AD exception.

For further derivations it is important to note that in the backward mode of automatic differentiation the expression  $\frac{\hat{\delta}f_1(\mathbf{a})}{\hat{\delta}\mathbf{a}} + \frac{\hat{\delta}f_2(\mathbf{a})}{\hat{\delta}\mathbf{a}}$  can result in a code that is twice as large as and twice times slower than the code produced by the equivalent expression  $\frac{\hat{\delta}(f_1(\mathbf{a})+f_2(\mathbf{a}))}{\hat{\delta}\mathbf{a}}$ . In general, the proposed automation procedure would be optimal if the number of AD calls is kept to a minimum.

The additional information introduced by the AD exception (2.6) is associated with the intermediate variables  $\mathbf{b}$ . The point in the algorithm where this additional information is introduced is important as well. Two typical situations are considered. The simplest case is when additional information is introduced together with a particular call of AD procedure. This situation is referred to as “local definition of AD exception”. In the case of complex computational models that involve a large number of differentiations and several types of variables for which AD exceptions have to be specified, local definition of AD exception becomes cumbersome. The information about all AD exceptions relevant for the intermediate variables  $\mathbf{b}$  can also be introduced at the point in algorithm where  $\mathbf{b}$  is introduced. The situation is referred to as “global definition of AD exceptions”.

### 2.4.3 Local Definition of AD Exceptions

The local definition of an AD exception is provided by

$$\nabla f_A = \frac{\hat{\delta} f(\mathbf{a}, \mathbf{b}(\mathbf{a}))}{\hat{\delta} \mathbf{a}} \bigg|_{\frac{D\mathbf{b}}{D\mathbf{a}}=\mathbf{M}}. \quad (2.9)$$

Notation (2.9) indicates that during the AD procedure, the total derivatives of intermediate variables  $\mathbf{b}$  with respect to independent variables  $\mathbf{a}$  are set to be equal to matrix  $\mathbf{M}$ . The use of AD exception (2.9) can be beneficial also when  $\mathbf{M} = \frac{D\mathbf{b}}{D\mathbf{a}}$  and explicit algorithmic dependency of  $\mathbf{b}$  with respect to  $\mathbf{a}$  exists. Thus, in principle the required total derivatives can be obtained automatically by the AD procedure using the chain-rule. This is often the case when there exists a profound mathematical relationship that enables the evaluation of total derivatives  $\frac{D\mathbf{b}}{D\mathbf{a}}$  in a more efficient way (e.g. when the evaluation of  $\mathbf{b}$  involves iterative loops, inverse matrices, etc.).

Several special cases and generalizations of (2.9) can also be introduced. The first special case is when the intermediate variables  $\mathbf{b}$  algorithmically depend on  $\mathbf{a}$  but have to be considered constant during the AD procedure. In this case, the direct use of AD would not give the correct result without intervention of the user. The correct results can be obtained by setting the matrix  $\mathbf{M}$  in (2.9) to zero as follows

$$\nabla f_B = \frac{\hat{\delta} f(\mathbf{a}, \mathbf{b}(\mathbf{a}))}{\hat{\delta} \mathbf{a}} \bigg|_{\frac{D\mathbf{b}}{D\mathbf{a}}=0} = \frac{\hat{\delta} f(\mathbf{a}, \mathbf{b}(\mathbf{a}))}{\hat{\delta} \mathbf{a}} \bigg|_{\mathbf{b}=\text{const.}}. \quad (2.10)$$

Situation (2.10) frequently appears in the formulation of mechanical problems where, instead of the total variation, some arbitrary variation of a given quantity has to be evaluated.

The second special case of (2.9) appears when intermediate variables  $\mathbf{b}$  algorithmically do not depend on  $\mathbf{a}$ , however from the mathematical formulation of the problem it is apparent that some additional (usually implicit) dependencies have to be considered for the differentiation leading to

$$\nabla f_C = \frac{\hat{\delta} f(\mathbf{a}, \mathbf{b})}{\hat{\delta} \mathbf{a}} \bigg|_{\frac{D\mathbf{b}}{D\mathbf{a}}=\mathbf{M}}. \quad (2.11)$$

Exception (2.9) can also be generalized. Let  $\mathbf{c}$  be an additional set of intermediate variables that appear during the evaluation of function  $f$  such that  $\mathbf{b} := \mathbf{f}_b(\mathbf{c})$  and let  $\mathbf{f}_c$  be a set of arbitrary functions of  $\mathbf{a}$  such that  $\mathbf{c} := \mathbf{f}_c(\mathbf{a})$ . A bridge in the chain-rule can now be formed that goes directly from  $\mathbf{b}$  to  $\mathbf{c}$ . The chain-rule from  $\mathbf{c}$  to  $\mathbf{a}$  is then propagated automatically by the AD procedure to complete the evaluation of computational derivatives. The situation is described by the following formalism

$$\nabla f_D = \frac{\hat{\delta} f(\mathbf{a}, \mathbf{b}(\mathbf{c}(\mathbf{a})))}{\hat{\delta} \mathbf{a}} \bigg|_{\frac{D\mathbf{b}}{D\mathbf{c}}=\mathbf{M}}. \quad (2.12)$$

It is important for the uniqueness of AD procedure that  $\frac{\partial \mathbf{b}}{\partial \mathbf{a}}$  is zero in this case, thus  $\mathbf{b}$  depends explicitly only on  $\mathbf{c}$  and it does not depend directly on  $\mathbf{a}$ .

#### 2.4.4 Global Definition of AD Exceptions

AD exceptions (2.9)–(2.12) are imposed during the execution of the particular call of the AD procedure to which they are attached. The global definition of AD exceptions relevant for the intermediate variables  $\mathbf{b}$  appears at the point in the algorithm where  $\mathbf{b}$  is introduced and is indicated by the following formalism

$$\begin{aligned} \mathbf{b} &= \mathbf{f}_b(\mathbf{a}) \big|_{\frac{D\mathbf{b}}{D\mathbf{a}}=\mathbf{M}(\mathbf{a})} \\ \nabla f_A &= \frac{\hat{\delta} f(\mathbf{a}, \mathbf{b}(\mathbf{a}))}{\hat{\delta} \mathbf{a}} \end{aligned} \quad (2.13)$$

The global definition of the AD exception (2.13) is equivalent to the local definition of the AD exception (2.9) except that the AD exception (2.13) is defined globally, thus valid for every subsequent call of the AD procedure while (2.9) applies only for the specific call of the AD procedure. The formalism (2.13) does not change the value of the variables  $\mathbf{b}$ . However, the definition (2.13) forces the AD procedure to ignore all dependencies of  $\mathbf{b}$  as defined by function  $\mathbf{f}_b$  and specifies that during all subsequent calls of the AD procedure the total derivatives of variables  $\mathbf{b}$  with respect to  $\mathbf{a}$  are set to be equal to matrix  $\mathbf{M}$ .

As for the local definition of AD exceptions, there also exist special cases and generalizations for the global definition of AD exceptions. In Table 2.1 the various types of local definitions of AD exceptions introduced in previous sections are summarized and accompanied by the corresponding global definitions. The letters A to D that identify the specific types that are introduced for later use.

Obviously, multiple definitions of AD exceptions for the same variables can clash. For example, it is possible to define first a global AD exception (2.13) and later, at the actual call of the AD procedure, additionally a local AD exception (2.9). When local and global exceptions clash, the local definition of an AD exception takes precedence over the global definition of an AD exception for the same variable.

#### 2.4.5 Differentiation with Respect to Variables with an Index

If  $f(\mathbf{a})$  is an explicit function of all components of  $\mathbf{a}$  then the physical size of the generated code for a derivative  $\frac{\hat{\delta} f(\mathbf{a})}{\hat{\delta} \mathbf{a}}$  depends on the number of independent



**Table 2.1** Typical automatic differentiation exceptions

Type	Local definition of AD exception	Global definition of AD exception
A	$\nabla f_A = \frac{\hat{\delta} f(\mathbf{a}, \mathbf{b}(\mathbf{a}))}{\hat{\delta} \mathbf{a}} \Big _{\frac{D\mathbf{b}}{D\mathbf{a}}=\mathbf{M}}$	$\mathbf{b} = \mathbf{f}_b(\mathbf{a}) \Big _{\frac{D\mathbf{b}}{D\mathbf{a}}=\mathbf{M}(\mathbf{a})}$ $\nabla f_A = \frac{\hat{\delta} f(\mathbf{a}, \mathbf{b}(\mathbf{a}))}{\hat{\delta} \mathbf{a}}$
B	$\nabla f_B = \frac{\hat{\delta} f(\mathbf{a}, \mathbf{b}(\mathbf{a}))}{\hat{\delta} \mathbf{a}} \Big _{\mathbf{b}=\text{const.}}$	$\mathbf{b} = \mathbf{f}_b(\mathbf{a}) \Big _{\frac{D\mathbf{b}}{D\mathbf{a}}=\mathbf{0}}$ $\nabla f_B = \frac{\hat{\delta} f(\mathbf{a}, \mathbf{b}(\mathbf{a}))}{\hat{\delta} \mathbf{a}}$
C	$\nabla f_C = \frac{\hat{\delta} f(\mathbf{a}, \mathbf{b})}{\hat{\delta} \mathbf{a}} \Big _{\frac{D\mathbf{b}}{D\mathbf{a}}=\mathbf{M}}$	$\mathbf{b} = \mathbf{f}_b \Big _{\frac{D\mathbf{b}}{D\mathbf{a}}=\mathbf{M}}$ $\nabla f_C = \frac{\hat{\delta} f(\mathbf{a}, \mathbf{b})}{\hat{\delta} \mathbf{a}}$
D	$\nabla f_D = \frac{\hat{\delta} f(\mathbf{a}, \mathbf{b}(\mathbf{c}(\mathbf{a})))}{\hat{\delta} \mathbf{a}} \Big _{\frac{D\mathbf{b}}{D\mathbf{c}}=\mathbf{M}}$	$\mathbf{c} = \mathbf{f}_c(\mathbf{a})$ $\mathbf{b} = \mathbf{f}_b(\mathbf{c}) \Big _{\frac{D\mathbf{b}}{D\mathbf{c}}=\mathbf{M}}$ $\nabla f_D = \frac{\hat{\delta} f(\mathbf{a}, \mathbf{b}(\mathbf{c}(\mathbf{a})))}{\hat{\delta} \mathbf{a}}$

variables. This can again lead to swell of expressions and is not acceptable for larger numbers of independent variables. Thus, explicit code for all terms of the gradient can be generated only if the number of independent variables is small. The size of the generated code can be reduced by an introduction of *representative formulas*. A representative formula is one general formula, that can be used for the evaluation of several other formulas. It is a formula with an index. If the index represents a loop counter then the representative formula can be evaluated within the loop several times in succession. The concept is of course well known and often used in mechanics to reduce the complexity of derivations. It is here extended to the automatic differentiation procedure. The generation of a representative formula is indicated by the following formalism

$$\frac{\hat{\delta} f(\mathbf{a})}{\hat{\delta} a_m} \quad (2.14)$$

The operator  $\frac{\hat{\delta} f(\mathbf{a})}{\hat{\delta} a_m}$  represents differentiation of the function  $f(\mathbf{a})$  with respect to the  $m$ th element of the vector of the independent variables  $\mathbf{a}$  performed by an automatic differentiation algorithm. The result is an algorithm that, when evaluated in a loop with the loop counter  $m$ , yields the elements of the gradient vector  $\nabla_{\mathbf{a}} f$ .

## 2.5 Automatic Code Generation with AceGen

### 2.5.1 Hybrid Symbolic-Numerical System AceGen

The idea implemented in *AceGen* is not to try to combine different tools, but to combine different techniques inside one system in order to avoid the above mentioned

problems. Thus, the main objective is to combine techniques in such a way that will lead to an optimal environment for the design and coding of numerical subroutines. Among the presented systems the most versatile are indeed the SAC systems. They normally contain, besides algebraic manipulation, graphics and numerical capabilities, also powerful programming languages. It is therefore quite easy to simulate other techniques inside the SAC system. The approach of automatic code generation used in *AceGen* is called *Simultaneous Stochastic Simplification of numerical code* (Korelc 1997a). This approach combines the general computer algebra system *Mathematica* with an automatic differentiation technique and an automatic theorem proving by examples. To alleviate the problem of the growth of expressions and redundant calculations, simultaneous simplification of symbolic expressions is used. Stochastic evaluation of the formulae is applied for determining the equivalence of algebraic expressions, instead of the conventional pattern matching technique. *AceGen* was designed to approach especially to hard problems, where the general strategy for an efficient formulation of numerical procedures, such as analytical sensitivity analysis of complex multi-field problems, has not yet been established.

The structure of *AceGen* is presented in Fig. 2.1. General characteristics of the *AceGen* code generator are:

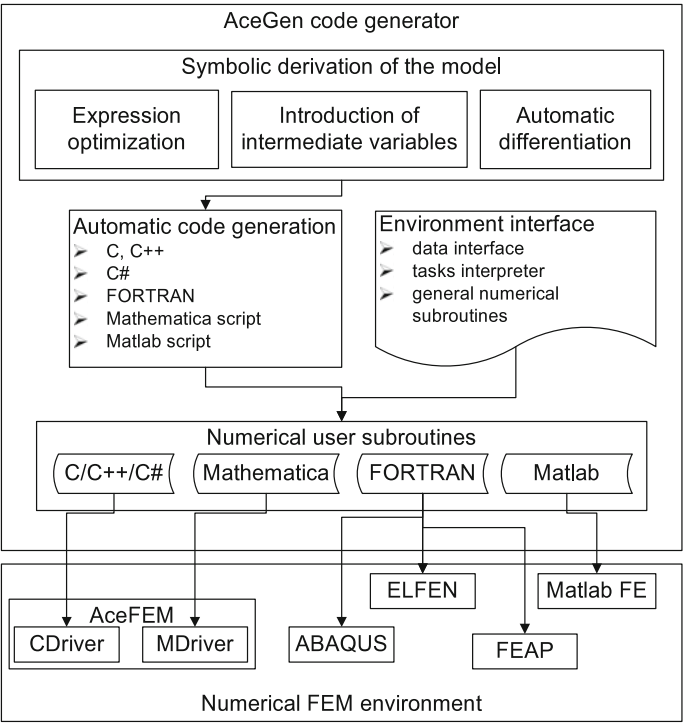


Fig. 2.1 Hybrid symbolic-numeric approach to automation of the finite element method

1. simultaneous optimization of expressions immediately after they have been derived,
2. automatic differentiation technique,
3. automatic selection of the appropriate intermediate variables,
4. generation of the whole program structure,
5. appropriate for large problems where also intermediate expressions can be subjected to the uncontrolled swell,
6. global expression optimization with stochastic evaluation of expressions,
7. differentiation with respect to indexed variables,
8. automatic interface to other numerical environments (by using the `Splice` command of *Mathematica*),
9. multi-language code generation (FORTRAN/FORTRAN90, C/C++/C#, *Mathematica* language, *Matlab* language),
10. advanced methods for exploring and debugging of generated formulas,
11. system is written in the symbolic language of *Mathematica*.

### 2.5.2 Typical AceGen Automatic Code Generation Procedure

A simple example is considered here to illustrate the standard *AceGen* procedure. It is a subroutine that returns the determinant of the Jacobi matrix of a nonlinear transformation from the reference to initial configuration for quadrilateral element topology (for details see Sect. 4.2.1). The syntax of the *AceGen* script language is the same as the syntax of the *Mathematica* script language with some additional functions. A short description of the *AceGen* syntax is given in Appendix A while the detailed description can be found in Korelc (2011). The input for *AceGen* that produces the required subroutine is as follows

```
<<AceGen` ;
SMSInitialize["DetJSub", "Language" -> "C"] ;
SMSModule["DetJ", Real[X$$[2, 4], k$$, e$$, Jg$$]] ;
{ξ, η} = SMSReal[{k$$, e$$}] ;
{Xc, Yc} = Table[SMSReal[X$$[i, j]], {i, 2}, {j, 4}] ;
Nh = {(1 - ξ) (1 - η), (1 + ξ) (1 - η), (1 + ξ) (1 + η), (1 - ξ) (1 + η)} / 4 ;
Jg = SMSD[{Nh.Xc, Nh.Yc}, {ξ, η}] ;
SMSEXP[Det[Jg], J$$] ;
SMSWrite["LocalAuxiliaryVariables" -> True] ;
```

(2.15)

This input can be divided into six characteristic steps:

#### Step 1: Initialization

```
<<AceGen` ;
SMSInitialize["DetJSub", "Language" -> "C"] ;
```

At the beginning of the session the `SMSInitialize` function initializes the system and specifies the name of the generated source code file. The additional option "Language"  $\rightarrow$  "C" specifies that the subroutine is generated in C language.

**Step 2:** Definition of input and output parameters

```
SMSModule["DetJ", Real[X$$[2, 4], k$$, e$$, Jg$$]];
```

The `SMSModule` function defines the name (`DetJ`) and input/output parameters of the subroutine. Input parameter `X$$` is a  $2 \times 4$  matrix of nodal coordinates, `k$$` and `e$$` are  $\xi$  and  $\eta$  coordinates of the material point in reference frame and `Jg$$` is an output parameter of the subroutine. Double  $\$$  character always indicates that the symbol is an input or output parameter of the generated subroutine.

**Step 3:** Definition of numeric-symbolic interface variables

```
{ξ, η} ← SMSReal[{k$$, e$$}];  
{Xc, Yc} ← Table[SMSReal[X$$[i, j]], {i, 2}, {j, 4}];
```

The `SMSReal` function assigns random signature to input parameters `k$$` and `e$$`. The  $\leftarrow$  operator then creates new auxiliary variables  $\xi$  and  $\eta$ . Similarly, vectors of auxiliary variables `Xc` and `Yc` hold  $X$  and  $Y$  coordinates of element nodes.

**Step 4:** Derivation of the problem

```
Nh = {(1-ξ) (1-η), (1+ξ) (1-η), (1+ξ) (1+η), (1-ξ) (1+η)}/4;  
Jg = SMSD[{Nh.Xc, Nh.Yc}, {ξ, η}];
```

During the description of the problem a special operator  $\models$  is used that performs global optimization of expressions and then creates new auxiliary variables if *AceGen* finds out that the introduction of a new auxiliary variable is necessary. The `SMSD` function performs the automatic differentiation of one or several expressions with respect to one or a vector of auxiliary variables by simultaneously enhancing the already derived code.

**Step 5:** Exporting results to output parameters

```
SMSExport[Det[Jg], Jg$$];
```

The results of the derivation are assigned to the output parameter `Jg$$` of the subroutine by `SMSExport` function.

**Step 6:** Code generation

```
SMSWrite["LocalAuxiliaryVariables"  $\rightarrow$  True];
```

At the end of the session the `SMSWrite` function writes the contents of the vector of the generated formulas to file `DetJSub.c` in a prescribed language format.

Due to the advantage of the simultaneous optimization procedure we can execute each step separately and examine intermediate results. This is also the basic way how to trace errors that might occur during the *AceGen* session. The generated source code in C language is as follows.

```
/***** S U B R O U T I N E *****/
void DetJ(double X[2][4],double (*k),double (*e),double (*Jg))
{
double v[36];
v[31]=(-1e0+(*e))/4e0;
v[30]=(1e0+(*e))/4e0;
v[29]=(-1e0-(*k))/4e0;
v[28]=(-1e0+(*k))/4e0;
(*J)=(v[31]*(X[0][0]-X[0][1])+v[30]*(X[0][2]-X[0][3]))*(v[29]*(X[1][1]-
X[1][2])+v[28]*(X[1][0]-
-X[1][3]))-(v[29]*(X[0][1]-X[0][2])+v[28]*(X[0][0]-X[0][3]))*(v[31]*(X[1][0]-
X[1][1])+v[30]*
(X[1][2]-X[1][3]));
};
```

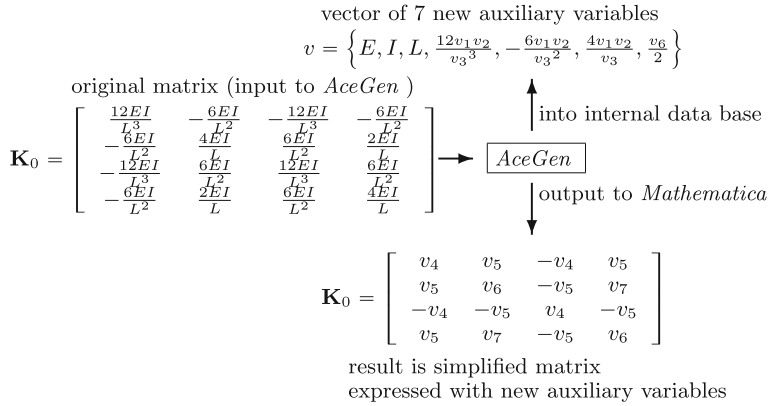
With the change of language option to "Language" -> "Fortran" the following source code in FORTRAN language is generated.

```
!***** S U B R O U T I N E *****/
      SUBROUTINE DetJ(X,k,e,Jg)
      IMPLICIT NONE
      include 'sms.h'
      DOUBLE PRECISION v(36),X(2,4),k,e,Jg
      v(31)=((-1d0)+e)/4d0
      v(30)=(1d0+e)/4d0
      v(29)=((-1d0)-k)/4d0
      v(28)=((-1d0)+k)/4d0
      J=(v(31)*(X(1,1)-X(1,2))+v(30)*(X(1,3)-X(1,4)))*(v(29)*(X(2,2)
&-X(2,3))+v(28)*(X(2,1)-X(2,4)))-(v(29)*(X(1,2)-X(1,3))+v(28)*(X
&(1,1)-X(1,4)))*(v(31)*(X(2,1)-X(2,2))+v(30)*(X(2,3)-X(2,4)))
      END
```

### 2.5.3 Simultaneous Simplification Procedure

During the description of the problem the special operator  $\models$  was used to perform the simultaneous optimization of expressions and the creation of new intermediate variables.

A typical *AceGen* function takes the expression provided by the user, either interactively or from a file, and returns an optimized version of the expression. Optimized version of the expression can result in a newly created auxiliary symbol ( $v_i$ ), or in an original expression in parts replaced by previously created auxiliary symbols. In the first case, *AceGen* stores the new expression in an internal database. The database contains a global vector of all expressions, information about dependencies of the symbols, labels and names of the symbols, partial derivatives, etc. The database is a global object that maintains information during the *AceGen* session.



**Fig. 2.2** Simultaneous simplification procedure

The classical way of optimizing expressions in a computer algebra systems is searching for common sub-expressions at the end of the derivation, before the generation of the numerical code. In the numerical code common sub-expressions appear as auxiliary variables. An alternative approach is implemented in *AceGen* where formulas are optimized, simplified and replaced by the auxiliary variables simultaneously with the derivation of the problem. The optimized version is then used in further operations (Fig. 2.2). When the optimization is performed simultaneously, the explicit form of an expression is obviously lost, since some parts are replaced by intermediate variables.

In real problems it is almost impossible to recognize the identity of two expressions (for example the symmetry of the tangent stiffness matrix in nonlinear mechanical problems) automatically using only the pattern matching mechanisms. Normally our goal is to recognize the identity automatically without introducing additional knowledge into the derivation such as tensor algebra, matrix transformations, etc. Commands in *Mathematica* such as *Simplify*, *Together*, and *Expand*, are useless in the case of large expressions. Additionally, these commands are efficient only when the whole expression is considered. When optimization is performed simultaneously, the explicit form of the expression is lost. The only possible way at this stage of computer technology seems to be an algorithm which finds equivalence of expressions numerically. This relatively old idea (see for example Martin 1971 or Gonnet 1986) is rarely used, although it is essential for dealing with especially hard problems. However, to show identity numerically is not a mathematically rigorous proof for the identity of two expressions. Thus the correctness of the simplification can be determined only within a certain degree of probability. With regard to our experience the proof of numerical identity of expressions is sufficient in mechanical analysis when dealing with more or less “smooth” functions.

### 2.5.4 Efficiency and Limitations of Automation of Computational Modeling

An important question arises: how to understand and analyze automatically generated formulas? The automatically generated codes should not act like a “black box”. For example, after using the automatic differentiation tools we have no insight in the actual structure of the derivatives. One might argue that this is not needed, however if the finite element codes are to be derived with an efficiency comparable with the manually written codes then an efficient debugging procedure becomes a necessity. The standard way in which *Mathematica* works is to evaluate a single expression completely, which then can be analyzed and printed in a required format. If the problem is complex with potentially hundreds of expressions and it includes loops and branching, then more sophisticated procedures for the analysis and interactive run time debugging are required. Additionally, when optimization of the derived numerical code is performed simultaneously with the derivation of formulas, the explicit form of the expressions is lost.

The automatically generated finite element subroutines are difficult to analyze and to understand directly. Thus, it is in general difficult to debug the generated subroutine after it is included into the commercial finite element environment. For an efficient interactive debugging it is important where the debugging actually takes place. The *Mathematica* based high-level symbolic interface standard for inter-program communication called *MathLink* allows to run external programs under whatever debugger is provided for a particular compiler. However, this approach is not applicable for automatically generated codes, since the structure of the automatically generated codes is rather “unreadable”, hence difficult to be debugged directly. The approach where the generated code is not debugged directly but where the programmer can look at the analytical expressions and their current numerical values when using the symbolic environment is essential for the debugging of automatically generated codes.

## 2.6 Automatic Differentiation and Finite Element Method

The AD tools were primarily developed for the evaluation of the gradient of an objective function used within gradient-based optimization procedures. In general, the objective function can be defined by a program composed of many subroutines including a complete FE environment. Thus, one can apply the AD tools directly to the complete FE environment to obtain the required derivatives when the evaluation of the objective function involves FE simulation. The AD tools have been successfully applied to FE environments with several 100,000 lines of code; see e.g. Bischof et al. (2003).

A large finite element environment usually employs a variety of finite elements, solution procedures. Commonly commercial numerical libraries are used within such

system for which the source codes are not readily available. For these finite element environments it is difficult to directly apply the AD tools to get e.g. the global stiffness matrix of a large-scale problem. However, the AD technology can still be used for the evaluation of specific quantities that appear as a part of the FE simulation. For example, one can use AD at the individual element level to evaluate element specific quantities such as:

- strain and stress tensors,
- nonlinear coordinate transformations,
- residual vectors,
- consistent tangent stiffness matrices and
- sensitivity pseudo-load vectors.

A direct use of automatic differentiation tools for the development of nonlinear finite elements turns out to be complex and not straightforward. Furthermore the numerical efficiency of the resulting codes is poor. One solution, followed mostly in hybrid object-oriented systems, is to use problem specific solutions to evaluate local tangent matrix in a optimal way, see. e.g. Kirby et al. (2005). Another solution, pursued in hybrid symbolic-numeric systems, is to combine a general computer algebra system and the AD technology, see e.g. Korelc (2002).

An implementation of the AD procedure has to fulfill specific requirements in order to obtain element source code that is as efficient as manually written codes. Some basic requirements are:

- The AD procedure can be initiated at any time and at any point of the derivation of the formulas and as many times as required (e.g. in the example at the end the AD is used 13 times during the generation of an element subroutine). The recursive use of standard AD tools on the same code, if allowed at all, leads to numerically inefficient source code. This requirement limits the use of standard AD tools. An alternative approach is implemented in Korelc (2002) where the source-to-source transformation strategy is replaced by a method that consistently enhances the existing code rather than producing a new one.
- The storage of the intermediate variables is not a limitation when the backward differentiation method is employed at single element level. Finite element formulations at the single element level involve a relatively small set of independent and intermediate variables.
- For reasons of efficiency, the results of all previous uses of AD have to be accounted for when automatic differentiation is applied several times inside the same subroutine.
- The user has to be able to use all capabilities of the symbolic system on the final and the intermediate results of the AD procedure.
- The AD procedure must offer a mechanism for the descriptions of various mathematical formalisms applied within a finite element formulation.

In implicit methods, the bulk of computational time is spent on performing differentiations, thus the efficiency of the automation of differentiation is most important. Two effects that have to be considered:



- the number of recursive uses of differentiation within the derivation of the same subroutine and
- the approach employed to perform differentiation.

As presented in Sect. 2.4 there are two approaches for the implementation of the automatic differentiation of a computer program, the forward and the backward mode of automatic differentiation. The numerical efficiency of the differentiation of  $N$  scalar-valued functions with respect to  $M$  independent variables can be measured by the numerical work ratio (2.4). The ratio is in general proportional to the number of independent variables in the case of forward mode and proportional to the number of functions in the case of backward mode (see also Sect. 2.4). For the efficient automation of the FE method, it is desirable that both approaches are available and that the software tool used for automation can automatically select the most efficient approach according to the estimated work ratio, though the backward mode would be preferable in most of the cases.

Contrary to the classical implementation of automatic differentiation where the AD works as a code-to-code translator, the implementation in *AceGen* enhances the currently derived code. However, the result of repeated use of AD within the same subroutine is that the simultaneous code optimization procedures become less and less efficient. To summarize, the automatically generated code would be numerically efficient if the number of functions to differentiate and the number of calls to the AD procedure is kept at minimum. One consequence of the rule is that in general formulations where the element residual vector is derived as the gradient of a scalar function, e.g. variational potential, leads to a more efficient numerical code than formulations based on the weak form of the equilibrium equations where the variation of the kinematic quantities e.g. strains tensor or tangential gap vector requires differentiation of several scalar functions. Thus, the possibility of transforming the weak form into the “pseudo-potential” scalar function is always worth to explore. The pseudo-potential has to be formed in a way that the automatic differentiation procedure of the pseudo-potential accompanied with the proper automatic differentiation exceptions leads to the correct equations of the problem.

The book demonstrates by means of several examples direct consequences of using automatic differentiation exceptions. It will show how the mechanical problem and its corresponding numerical model are formulated and solved. The ADB form of the problem description is rather straightforward for e.g. total-Lagrangian displacement-based finite element formulation of hyperelastic problems. However, it can be nontrivial for e.g. spatial formulation of finite strain elasto-plastic finite elements.

### 2.6.1 ADB Form of General Potential Form

Assume that the solution of a problem is defined as the minimum of the potential  $\Pi(\mathbf{p}) = \int_{\Omega} W(\mathbf{p}) d\Omega$  where  $\mathbf{p} = \{p_1, p_2, \dots, p_{n_p}\}^T$  is a set of unknown parameters of the problem. The variation of  $\Pi(\mathbf{p})$  is computed as

$$\delta \Pi(\mathbf{p}) = \frac{\partial \Pi(\mathbf{p})}{\partial \mathbf{p}} \delta \mathbf{p} = \int_{\Omega} \frac{\partial W(\mathbf{p})}{\partial \mathbf{p}} d\Omega \delta \mathbf{p} = 0 \quad (2.16)$$

where  $\delta \mathbf{p} = \{\delta p_1, \delta p_2, \dots, \delta p_{n_p}\}^T$  is a variation of unknown parameters. Equation (2.16) leads to a set of nonlinear algebraic equations of the form

$$\mathbf{R} = \int_{\Omega} \frac{\partial W(\mathbf{p})}{\partial \mathbf{p}} d\Omega = \mathbf{0}. \quad (2.17)$$

The ADB form of the general problem with potential is obtained from (2.17) where the partial derivative is directly replaced by the computational derivative

$$\mathbf{R} = \int_{\Omega} \frac{\hat{\delta} W(\mathbf{p})}{\hat{\delta} \mathbf{p}} d\Omega = \mathbf{0}. \quad (2.18)$$

Furthermore, the individual element contribution  $\mathbf{R}_e$  to the global residual vector  $\mathbf{R}$  is in standard finite element formulations obtained by a numerical integration rule—typically Gauss integration

$$\mathbf{R}_e \approx \sum_{g=1}^{n_g} w_{gp} \mathbf{R}_g \quad (2.19)$$

where  $w_{gp}$  stands for the Gauss point weights,  $n_g$  is the number of Gauss points and  $\mathbf{R}_g$  is the Gauss point contribution to the residual vector or **Gauss** point residual given by

$$\mathbf{R}_g = \frac{\hat{\delta} W(\mathbf{p})}{\hat{\delta} \mathbf{p}} \quad (2.20)$$

The corresponding ADB form of the Gauss point contribution to the global tangent matrix  $\mathbf{K}$  is

$$\mathbf{K}_g = \frac{\hat{\delta} \mathbf{R}_g}{\hat{\delta} \mathbf{p}_e}. \quad (2.21)$$

A schematic *AceGen* input for the general potential form (2.20) and (2.21) is provided by the following *AceGen* input segment

```

pe=SMSReal[Table[p[[i]],{i,1,np}]];
W=fW[pe];
Rg=SMSD[W,pe];
Kg=SMSD[Rg,pe];

```

(2.22)

### 2.6.2 ADB Form of General Weak Form

Assume the weak form  $\int_{\Omega} \mathbf{a} \cdot \delta \mathbf{b} d\Omega + \dots = 0$  where  $\mathbf{a}$  and  $\mathbf{b}$  are tensors of an arbitrary order and  $\delta \mathbf{b}$  is a directional derivative or variation of  $\mathbf{b}$ . The symbolic formulation of the weak form is not straightforward, since  $\delta \mathbf{b}$  is not a real but rather fictitious quantity and AD cannot be applied directly.<sup>2</sup> But AD can be applied directly after the weak form is discretized. The variation of  $\mathbf{b}$  is computed as

$$\delta \mathbf{b}(\mathbf{p}) = D \mathbf{b}(\mathbf{p}) \delta \mathbf{p} = \frac{\partial \mathbf{b}(\mathbf{p})}{\partial \mathbf{p}} \delta \mathbf{p} \quad (2.23)$$

and the scalar product  $\mathbf{a}(\mathbf{p}) \cdot \delta \mathbf{b}(\mathbf{p})$  as

$$\mathbf{a}(\mathbf{p}) \cdot \delta \mathbf{b}(\mathbf{p}) = \mathbf{a}(\mathbf{p}) \cdot \frac{\partial \mathbf{b}(\mathbf{p})}{\partial \mathbf{p}} \delta \mathbf{p} = \left( \mathbf{a}(\mathbf{p}) \cdot \frac{\partial \mathbf{b}(\mathbf{p})}{\partial \mathbf{p}} \right) \delta \mathbf{p}. \quad (2.24)$$

The product  $\mathbf{a}(\mathbf{p}) \cdot \frac{\partial \mathbf{b}(\mathbf{p})}{\partial \mathbf{p}}$  can also be written component wise

$$\left( \mathbf{a}(\mathbf{p}) \cdot \frac{\partial \mathbf{b}(\mathbf{p})}{\partial \mathbf{p}} \right)_m = \mathbf{a}(\mathbf{p}) \cdot \frac{\partial \mathbf{b}(\mathbf{p})}{\partial p_m} = tr \left( \mathbf{a}(\mathbf{p})^T \frac{\partial \mathbf{b}(\mathbf{p})}{\partial p_m} \right) \quad (2.25)$$

by using the trace operator, see Appendix B.1.3 and B.1.5. The discretized weak form is then given by

$$\int_{\Omega} \mathbf{a}(\mathbf{p}) \cdot \delta \mathbf{b}(\mathbf{p}) d\Omega + \dots = \sum_{m=1}^{n_{ip}} \left( \int_{\Omega} \mathbf{a}(\mathbf{p}) \cdot \frac{\partial \mathbf{b}(\mathbf{p})}{\partial p_m} d\Omega \right) \delta p_m + \dots = 0 \quad (2.26)$$

and leads to a set of  $n_{ip}$  algebraic equations of the form

$$\mathbf{R} = \int_{\Omega} \mathbf{a}(\mathbf{p}) \cdot \frac{\partial \mathbf{b}(\mathbf{p})}{\partial \mathbf{p}} d\Omega + \dots = \mathbf{0}. \quad (2.27)$$

The ADB form of the discretized weak form is obtained from (2.27) where the partial derivative is directly replaced by the computational derivative

$$\mathbf{R} = \int_{\Omega} \mathbf{a}(\mathbf{p}) \cdot \frac{\hat{\delta} \mathbf{b}(\mathbf{p})}{\hat{\delta} \mathbf{p}} d\Omega + \dots = \mathbf{0}. \quad (2.28)$$

---

<sup>2</sup>Of course, a general computer algebra system can be used for building the necessary apparatus to deal with the virtual quantities in a traditional way and then automatic code generation can be applied on the results. However, the elegance of using automatic differentiation would then be lost.

The vector  $\frac{\hat{\delta}\mathbf{b}(\mathbf{p})}{\hat{\delta}\mathbf{p}}$  in Eq.(2.28) has to be derived explicitly (all  $n_{tp}$  components in closed form) for the evaluation of the scalar product. This operation can lead to an expression swell problem. Additionally, small differences in the actual form of the derived expressions can disrupt the code optimization procedure. The form (2.28) is also not optimal for the use of backward mode of AD since the cost of backward AD depends linearly on the number of scalar functions to be differentiated. For the present case, the numerical cost depends linearly on the number of components of  $\mathbf{b}$ . Thus, as mentioned before, the possibility of transforming the weak form into the pseudo-potential scalar function is worth exploring. The pseudo-potential has to be formed in a way that automatic differentiation of the pseudo-potential accompanied by proper automatic differentiation exceptions leads to the correct set of discretized equations of the problem. For the case (2.26) this can be achieved by introducing the AD exception that hides the dependency  $\mathbf{a}(\mathbf{p})$  from the AD procedure as follows

$$\mathbf{R} = \int_{\Omega} \mathbf{a}(\mathbf{p}) \cdot \frac{\hat{\delta}\mathbf{b}(\mathbf{p})}{\hat{\delta}\mathbf{p}} d\Omega + \dots = \int_{\Omega} \frac{\hat{\delta}(\mathbf{a}(\mathbf{p}) \cdot \mathbf{b}(\mathbf{p}))}{\hat{\delta}\mathbf{p}} \bigg|_{\frac{D\mathbf{a}}{D\mathbf{p}}=\mathbf{0}} d\Omega + \dots = \mathbf{0}. \quad (2.29)$$

With the introduction of the pseudo-potential

$$W^P = \mathbf{a}(\mathbf{p}) \cdot \mathbf{b}(\mathbf{p}) = tr(\mathbf{a}(\mathbf{p})^T \mathbf{b}(\mathbf{p})) \quad (2.30)$$

the final ADB form of the discretized weak form (2.26) is obtained as

$$\mathbf{R} = \int_{\Omega} \frac{\hat{\delta}W^P(\mathbf{p})}{\hat{\delta}\mathbf{p}} \bigg|_{\mathbf{a}=\text{const.}} d\Omega + \dots = \mathbf{0}. \quad (2.31)$$

Assuming that the integral is evaluated by a numerical integration rule as in the previous section, the Gauss point contribution to the residual vector or **Gauss** point residual leads to

$$\mathbf{R}_g = \mathbf{a}(\mathbf{p}_e) \cdot \frac{\hat{\delta}\mathbf{b}(\mathbf{p}_e)}{\hat{\delta}\mathbf{p}_e} \quad (2.32)$$

or when using the pseudo-potential form to

$$\mathbf{R}_g = \frac{\hat{\delta}W^P(\mathbf{p}_e)}{\hat{\delta}\mathbf{p}_e} \bigg|_{\mathbf{a}=\text{const.}}. \quad (2.33)$$

The corresponding ADB form of Gauss point contribution to the global tangent matrix  $\mathbf{K}$  is for both cases given by

$$\mathbf{K}_g = \frac{\hat{\delta}\mathbf{R}_g}{\hat{\delta}\mathbf{p}_e}. \quad (2.34)$$

The above procedure is valid for tensors of arbitrary rank. A schematic *AceGen* input for the standard weak form (2.32) for the case when **a** and **b** are scalars is

```

pe=SMSReal[Table[p$$[i],{i,1,np}]];
a=fa[pe];
b=fb[pe];
δb=SMSD[b,pe];
Rg=a δb;
Kg=SMSD[Rg,pe];

```

(2.35)

in the case of rank-one tensors it can be written as

```

pe=SMSReal[Table[p$$[i],{i,1,np}]];
a=fa[pe];
lb=fb[pe];
δlb=SMSD[lb,pe];
Rg=a.δlb;
Kg=SMSD[Rg,pe];

```

(2.36)

and in the case of rank-two tensors (2.32) leads to

```

pe=SMSReal[Table[p$$[i],{i,1,np}]];
a=fa[pe];
lb=fb[pe];
δlb=Transpose[SMSD[lb,pe],{2,3,1}];
Rg=Table[Tr[aT δlb[[i]]],{i,1,Length[pe]}];
Kg=SMSD[Rg,pe];

```

(2.37)

Note that in the case of rank-two tensors, the presented formulation requires only one execution of the AD procedure.

Similarly, a schematic *AceGen* input for the pseudo-potential ADB form (2.33) of the generalized weak form for the scalar product leads to<sup>3</sup>

```

pe=SMSReal[Table[p$$[i],{i,1,np}]];
a=SMSFreeze[fa[pe]];
b=fb[pe];
WP=a b;
Rg=SMSD[WP,pe,"Constant"→a];
Kg=SMSD[Rg,pe];

```

(2.38)

---

<sup>3</sup>By using the `SMSFreeze` operator a new auxiliary variable is created that can be safely used as independent variable within differentiation, see also Appendix A.2.2.

for the rank-one tensors to

```
pe=SMSReal[Table[p$$[i],{i,1,np}]];
a=SMSFreeze[fa[pe]];
b=fb[pe];
WP=a.b;
Rg=SMSD[WP,pe,"Constant"→a];
Kg=SMSD[Rg,pe];
```

(2.39)

and in the case of rank-two tensors to

```
pe=SMSReal[Table[p$$[i],{i,1,np}]];
a=SMSFreeze[fa[pe]];
b=fb[pe];
WP=Tr[aT.b];
Rg=SMSD[WP,pe,"Constant"→a];
Kg=SMSD[Rg,pe];
```

(2.40)

Note that in the case when **b** and **a** are scalars, the efficiency of the pseudo-potential ADB form (2.33) is the same as efficiency of the standard ADB weak form (2.32).

### 2.6.3 *Representative Formulas for Residual and Tangent Matrix*

If  $n_p$  is the number element global degrees of freedom (DOF) then  $\mathbf{R}_g$  has  $n_p$  components and  $\mathbf{K}_g$  has  $n_p^2$  components. The physical size of the automatically generated code is consequently proportional to the square of the number of element DOF's. This can lead to a swell of expressions and is not acceptable for higher order elements. Thus, an explicit code for all terms of the residual and tangent matrix, as presented in Sect. 2.6.1 for a general potential based formulations and in Sect. 2.6.2 for a general weak form formulations, can be generated only if the number of unknowns is small. The size of the generated code can be reduced by the generation of representative formulas as described in Sect. 2.4.5.

The characteristic  $m$ th term of the Gauss point residual is given by

$$R_{gm} = \frac{\hat{\delta} W(\mathbf{p}_e)}{\hat{\delta} p_{em}}. \quad (2.41)$$

for the potential based form (2.20) and for the weak form (2.32) by

$$R_{gm} = \mathbf{a}(\mathbf{p}) \cdot \frac{\hat{\delta} \mathbf{b}(\mathbf{p})}{\hat{\delta} p_{em}} = \text{tr} \left( \mathbf{a}(\mathbf{p})^T \frac{\hat{\delta} \mathbf{b}(\mathbf{p})}{\hat{\delta} p_{em}} \right) \quad (2.42)$$

and for the pseudo-potential form (2.33) by

$$R_{gm} = \left. \frac{\hat{\delta} W^P(\mathbf{p}_e)}{\hat{\delta} p_{em}} \right|_{\mathbf{a}=\text{const.}}. \quad (2.43)$$

The corresponding representative formula for the  $(m, n)^{th}$  term of the tangent matrix at a Gauss point is for all cases given by

$$K_{gmn} = \frac{\hat{\delta} R_{gm}}{\hat{\delta} p_{en}}. \quad (2.44)$$

A schematic *AceGen* input where one representative formula is generated for an arbitrary element of the residual and one representative formula for an arbitrary element of the tangent matrix for the standard weak form (2.42) is presented in Box 2.1.

```

pe=SMSReal [Table[p$$[i], {i, 1, np}]] ;
W=fW[pe] ;
SMSDo[m, 1, np]
  Rgm=SMSD[W, pe, m] ;
  SMSExport[wgp Rgm, R$$[m], "AddIn"→True] ;
  SMSDo[n, 1, np]
    Kgm=SMSD[Rgm, pe, n] ;
    SMSExport[wgp Kgm, K$$[m, n], "AddIn"→True] ;
  SMSEndDo[] ;
SMSEndDo[] ;

```

**Box 2.1.** *AceGen* input for generation of representative formulas for a general potential based *ADB* formulation

The *AceGen* command `SMSD[W, pe, m]` in Box 2.1 performs automatic differentiation of the potential  $W$  with respect to the  $m$ th element of the set of unknowns  $\mathbf{p}_e$ . The resulting representative formula is evaluated within the `SMSDo[ ..., {m, 1, np} ]` loop  $n_p$  times. Since the successive evaluations override the results of the previous evaluations, the results have to be simultaneously stored or exported by the `SMSExport` command into externally defined or allocated array (`$$R`). The same is true for the inner loop that evaluates the elements of the tangent matrix.

A similar schematic *AceGen* input is presented for the general weak *ADB* formulation (2.42) in Box 2.2 and for the general pseudo-potential *ADB* formulation (2.43) in Box 2.3.

```

pe=SMSReal[Table[p$$[i],{i,1,np}]];
a=fa[pe];b=fb[pe];
SMSDo[
  δbm=SMSD[lb,pe,m];
  Switch[order
    , "scalars", Rgm=a δbm ;
    , "rank-one tensors", Rgm=a.δbm;
    , "rank-two tensors", Rgm=Tr[aT.δbm];
  ];
SMSEExport[wgp Rgm,R$$[m], "AddIn"→True];
SMSDo[
  Kgm=SMSD[Rgm,pe,m];
  SMSEExport[wgp Kgm,K$$[m,n], "AddIn"→True];
  , {n,1,np}]
  , {m,1,np}];

```

**Box 2.2.** *AceGen* input for generation of representative formulas for a general weak form *ADB* formulation

```

pe=SMSReal[Table[p$$[i],{i,1,np}]];
a=SMSFreeze[fa[pe]];b=fb[pe];
Switch[order
  , "scalars", WP=a b;
  , "rank-one tensors", WP=a.b;
  , "rank-two tensors", WP=Tr[aT.b];
];
SMSDo[
  Rgm=SMSD[WP,pe,m, "Constant"→a];
  SMSEExport[wgp Rgm,R$$[m], "AddIn"→True];
  SMSDo[
    Kgm=SMSD[Rgm,pe,n];
    SMSEExport[wgp Kgm,K$$[m,n], "AddIn"→True];
    , {n,1,np}]
    , {m,1,np}];

```

**Box 2.3.** *AceGen* input for generation of representative formulas for a general pseudo-potential *ADB* formulation

**Representative formulas for multi-field problems.** The set of unknowns  $\mathbf{p}_e$  of a finite element is commonly used to discretize more than one scalar field (e.g. three displacements  $u$ ,  $v$  and  $w$ ). In that case the generation of one very general



representative formula for an arbitrary element of the residual and tangent matrix can lead to redundant numerical operations and consequently slower codes. The solution of the problem is to split a set of independent variables  $\mathbf{p}_e$  into a union of disjoint subsets in a way that each subset discretizes only one scalar field. The representative formulas are then generated for an arbitrary element of a particular subset.

The  $k$ th subset of unknowns, denoted by  $\mathbf{p}_{ek}$ , is defined by

$$\mathbf{p}_{ek} \subset \mathbf{p}_e, \quad \bigcup_{k=1}^{n_s} \mathbf{p}_{ek} = \mathbf{p}_e, \quad \mathbf{p}_{ek} \cap \mathbf{p}_{el} = \mathbf{0} : \forall k \neq l \quad (2.45)$$

where  $n_s$  is the number of subsets. Let  $n_{pk}$  be the length of the  $k$ th subset of unknowns  $\mathbf{p}_{ek}$ ,  $p_{ekm_k}$  the  $m_k$ th element of  $\mathbf{p}_{ek}$  and  $\bar{R}_{gm_k}$  a representative formula for an arbitrary element of a subset of the residual  $\mathbf{R}_g$  that belongs to the subset of unknowns  $\mathbf{p}_{ek}$ . A set of  $n_s$  representative formulas for the evaluation of the residual is then given by

$$\bar{\mathbf{R}}_g = \left\{ \bar{R}_{gm_k} = \frac{\hat{\delta} W(\mathbf{p}_e)}{\hat{\delta}(p_{ekm_k})} : k = 1, \dots, n_s \right\}^T. \quad (2.46)$$

In the same way a set of  $n_s \times n_s$  representative formulas is obtained for the evaluation of the tangent matrix

$$\bar{\mathbf{K}}_g = \left[ \bar{K}_{gm_k n_l} = \frac{\hat{\delta} \bar{R}_{gm_k}}{\hat{\delta}(p_{eln_l})} : k = 1, \dots, n_s, l = 1, \dots, n_s \right]. \quad (2.47)$$

Consider an example where a set of unknowns is split into two subsets with the lengths  $n_{p1}$  and  $n_{p2}$  as follows

$$\begin{aligned} \mathbf{p}_{e1} &= \{p_{e1i}, i = 1, \dots, n_{p1}\}^T. \\ \mathbf{p}_{e2} &= \{p_{e2i}, i = 1, \dots, n_{p2}\}^T. \end{aligned} \quad (2.48)$$

The set  $\bar{\mathbf{R}}_g$  contains in this case two representative formulas and a matrix of representative formulas for the evaluation of the tangent matrix  $\bar{\mathbf{K}}_g$  has dimension  $2 \times 2$ . A schematic *AceGen* input for the evaluation of the residual and tangent matrix for the general two subset case is given in Box 2.4.

```

pe1=SMSReal[Table[p$$[i],{i,np1}]];
pe2=SMSReal[Table[p$$[np1+i],{i,np2}]];
W=fW[pe1,pe2];
SMSDo[m1,1,np1];
  Rgm1=SMSD[W,pe1,m1];
  SMSEExport[wgp Rgm1,p$$[m1],"AddIn"→True];
  SMSDo[n1,1,np1];
    Kgm1n1=SMSD[Rgm1,pe1,n1];
    SMSEExport[wgp Kgm1n1,s$$[m1,n1],"AddIn"→True];
  SMSEndDo[];
SMSDo[n2,1,np2];
  Kgm1n2=SMSD[wgp Rgm1,pe2,n2];
  SMSEExport[Kgm1n2,s$$[m1,np1+n2],"AddIn"→True];
SMSEndDo[];
SMSEndDo[];
SMSDo[m2,1,np2];
  Rgm2=SMSD[W,pe2,m2];
  SMSEExport[wgp Rgm2,p$$[np1+m2],"AddIn"→True];
  SMSDo[n1,1,np1];
    Kgm2n1=SMSD[Rgm2,pe1,n1];
    SMSEExport[wgp Kgm2n1,s$$[np1+m2,n1],"AddIn"→True];
  SMSEndDo[];
SMSDo[n2,1,np2];
  Kgm2n2=SMSD[Rgm2,pe2,n2];
  SMSEExport[wgp Kgm2n2,s$$[np1+m2,np1+n2],"AddIn"→True];
SMSEndDo[];
SMSEndDo[];

```

**Box 2.4.** *AceGen* input for generation of representative formulas for a general two-subsets case

The algorithm presented in in Box 2.4 contains 6 loops and is rather complex. The algorithm can be significantly simplified in a special case when a set of unknowns is divided into subsets of equal lengths, thus  $n_{p1} = n_{p2} = \frac{n_p}{2}$ . A schematic *AceGen* input for the evaluation of the residual and tangent matrix for the a two-subsets case with the same length is given in Box 2.5. An example for the first case are continuum elements based on mixed variational principles were unknowns can be divided into two subsets: a subset of displacement degrees of freedom and a subset of mixed modes. An example for the second case are isoparametric continuum elements where subsets can be formed based on nodal degrees of freedom. An extensive study how different implementations of the same element effect efficiency of the generated code is presented in Chap. 4.

```

nps=np/2;
pe1=SMSReal [Table[p$$[i],{i,nps}]];
pe2=SMSReal [Table[p$$[nps+i],{i,nps}]];
W=fW[pe1,pe2];
SMSDo[m,1,np/2];
  Rgm={SMSD[W,pe1,m],SMSD[W,pe2,m]};
  SMSExport[wgp Rgm,{p$$[m],p$$[nps+m]}, "AddIn"→True];
  SMSDo[n,1,np/2];
    Kgm={SMSD[Rgm,pe1,n],SMSD[Rgm,pe1,n]}T;
    SMSExport[wgp Kgm,{s$$[m,n],s$$[m,nps+n]},
      {s$$[nps+m,n],s$$[nps+m,nps+n]}}, "AddIn"→True];
  SMSEndDo[];
SMSEndDo[];

```

**Box 2.5.** *AceGen* input for generation of representative formulas for a two-subsets case with the equal lengths

## 2.7 Automatic Generation of FE User Subroutines

The procedure described above and the code generated is not intended to be included into a specific FE environment. FE environments that enable the use of user defined finite elements usually require a strict form of a list of input and output parameters. *AceGen* can automatically generate the list of input and output parameters.

The standard procedure to generate finite element source code using *AceGen* is comprised of four major phases:

1. *AceGen* initialization,
2. template initialization,
3. definition of standard user subroutines,

The definition of each user subroutine consists of several steps:

- (a) declaration of a standard user subroutine,
- (b) definition of numeric-symbolic interface variables,
- (c) derivation of the problem,
- (d) exporting results to output parameters.

4. code generation.

The specific *AceGen* input can be divided into six characteristic steps. As an example, the *AceGen* input that generates three-dimensional, 8 noded, isoparametric, hyperelastic solid element is presented here. Only the structure of the *AceGen* input is explained, while the theoretical background will be given later in Chap. 4. Here only the most common and basic features and data structures are depicted. More extensive information about the interactions between the *AceGen* code generator and the target FE environment is given in Appendix A.2.4.

**Step 1:** *AceGen* initialization

```
<<AceGen`;  
SMSInitialize["H1element", "Environment" -> "AceFEM"];
```

At the beginning of the session the `SMSInitialize` function initializes the system and specifies the name of the generated source code file. The additional option `"Environment" -> "AceFEM"` specifies the finite element environment for which the element source code will be generated. The *AceFEM* finite element environment is chosen in this case. Additionally to the *AceFEM* environment one can also specify e.g. the research code *FEAP* or the commercial code *ABAQUS*.

**Step 2:** Template initialization

```
SMSTemplate[  
  "SMSTopology" -> "H1"  
  , "SMSDomainDataNames" ->  
    { "E -elastic modulus", "ν -poisson ratio", ... }  
  , "SMSDefaultData" -> { 21000, 0.3, ... }  
  , "SMSSymmetricTangent" -> True  
];  
nen=SMSNoNodes; ndim=SMSNoDimensions;  
np=SMSNoDOFGlobal;
```

The `SMSTemplate` function initializes constants that are needed to create a proper interface between the generated user subroutines and the finite element environment. Typically, a minimum of four constants has to be specified:

`"SMSTopology"` constant specifies a keyword that defines the element topology. For example the `"H1"` keyword defines a three-dimensional, 8 noded, hexahedral element with 3 degrees of freedom per node. Setting the topology constant also sets other constants that directly depend on topology, e.g. the number of nodes (`"SMSNoNodes"`), the spatial dimension (`"SMSNoDimensions"`), etc. Their values are available after the `SMSTemplate` command and they can be used to make the *AceGen* input more general and problem independent.

`"SMSDomainDataNames"` constant specifies a list of keywords that characterize the material constants. Material constants appear as an input data for finite element simulation.

`"SMSDefaultData"` constant specifies default (most common) values of the material constants.

`"SMSSymmetricTangent"` constant specifies whether the tangent matrix is symmetric or not. Only the upper triangular matrix has to be explicitly formed when the tangent matrix is symmetric.<sup>4</sup>

---

<sup>4</sup>This option also effects the selection of the algorithm applied to solve the resulting system of linear equations when *AceFEM* is used.

**Step 3.a:** Declaration of standard user subroutine

```
SMSStandardModule["Tangent and residual"];
```

While the `SMSModule` command used in Sect. 2.5.2 starts the definition of a user subroutine with a user supplied set of input/output parameters, the `SMSStandardModule` command starts the definition of the subroutine with a predefined set of input/output parameters that is adjusted for the needs of the specified finite element environment. A standard set of user subroutines supported by *AceGen* is:

**“Tangent and residual”** standard user subroutine returns the tangent matrix (stored in variable `s$$` with dimensions  $(n_p + n_{eh}) \times (n_p + n_{eh})$ ) and the residual or internal load vector (stored in variable `p$$` with dimension  $n_p + n_{eh}$ ) for the current state of element related data.

**“Postprocessing”** user subroutine returns two arrays with an arbitrary number of post-processing quantities as follows: `gpost$$` is an array of post-processing quantities at an integration point with dimensions *number of integration points*  $\times$  *number of integration point quantities*; `npost$$` is an array of the nodal point quantities with dimensions *number of nodes*  $\times$  *number of nodal point quantities*.

**“Sensitivity pseudo-load”** user subroutine returns a matrix of pseudo-load vectors (stored in the variable `s$$` with dimensions  $n_\phi \times (n_p + n_{eh})$ ) that is used in sensitivity analysis to calculate the sensitivity of the global unknowns with respect to an arbitrary parameter.

**“Dependent sensitivity”** is a standard user subroutine that is employed to calculate the sensitivity of locally coupled unknowns at element level. Details about the sensitivity analysis are presented in Chap. 8.

**Step 3.b:** Definition of numeric-symbolic interface variables

```
{Em, v} = SMSReal[Table[es$$["Data", i], {i, 2}]];
{λ, μ} = SMSHookeToLame[Em, v];
XIIO = Table[SMSReal[nd$$[i, "X", j]], {i, nen}, {j, ndim}];
uIO = SMSReal[Table[nd$$[i, "at", j], {i, nen}, {j, ndim}]];
pe = Flatten[uIO];
SMSDo[Ig, 1, SMSInteger[es$$["id", "NoIntPoints"]]];
E = {ξ, η, ζ} = Table[SMSReal[es$$["IntPoints", i, Ig]], {i, 3}];
wgp = SMSReal[es$$["IntPoints", 4, Ig]];
```

The input parameters of the standard user subroutines allow access to all data stored in the environment data base that relates to a specific element. The most frequently used input data is:

`nd$$[i, "X", j]` or  $X_{Ij}$  is  $j$ th component of the initial coordinates of the  $I$ th node,  
`nd$$[i, "at", j]` or  $p_{Ij}$  is the  $j$ th unknown at the  $I$ th element node,

$es\$\$[ "Data", i ]$  is the  $i$ th material constant (as previously defined by "SMSDomainDataNames" constant),  
 $es\$\$[ "IntPoints", i, j ]$  is the  $i$ th coordinate of the  $j$ th Gauss point,  
 $es\$\$[ "IntPoints", 4, j ]$  is the Gauss point weight at the  $j$ th Gauss point,  
 $es\$\$[ "id", "NoIntPoints" ]$  is the number of Gauss points.

### Step 3.c: Derivation of the problem

```

En={ {-1,-1,-1},{1,-1,-1},{1,1,-1},{-1,1,-1},{-1,-1,1},
      {1,-1,1},{1,1,1},{-1,1,1}};
Nh=Table[1/8 (1+ξ En[[i,1]]) (1+η En[[i,2]]) (1+ζ En[[i,3]]),
          {i,1,nen}];
X=SMSFreeze[Nh.XIO];u=Nh.uIO;Je=SMSD[X,En];Jed=Det[Je];
H=SMSD[u,X,"Dependency"→{En,X,SMSInverse[Je]}];
F=IdentityMatrix[3]+H;Ce=FT.F;JF=Det[F];
W=1/2 λ (JF-1)2+μ (1/2 (Tr[Ce]-3)-Log[JF]);
Rg=Jed SMSD[W,pe];
Kg=SMSD[Rg,pe];
  
```

In this part, the shape functions are defined, the steps (computation of displacement gradient, deformation gradient and right Cauchy–Green strain tensor) are followed to formulate the potential form  $W$ . Furthermore the residual vector and the tangent matrix are computed. Details and theoretical background will be described later in Chap. 4.

### Step 3.d: Exporting results to output parameters

```

SMSEExport[wgp Rg,p$\$, "AddIn"→True];
SMSEExport[wgp Table[Kg[[i,j]],{i,1,np},{j,1,np}],
            Table[s$\$[i,j],{i,1,np},{j,1,np}], "AddIn"→True];
SMSEndDo[];
  
```

The results of the derivation are assigned to the output parameters  $s\$\$$  and  $p\$\$$  of the “*Tangent and residual*” standard user subroutine by the `SMSEExport` function. Here the symmetry of the tangent matrix is accounted for by exporting only the upper triangle matrix.

**Steps 3.a–3.d:** Definition of second user subroutines Steps 3.a–3.d can be repeated several times for the definition of all required user subroutines.

### Step 4: Code generation

```

SMSWrite[];
  
```

At the end of the session the `SMSWrite` function writes the generated formulas together with the code that is related to the interface of the chosen finite element environment. This output is written to a file in the programming language of the target finite element environment.

## References

- Akers, R., P. Baffes, E. Kant, C. Randall, and R. Y. Steinberg. 1998. Automatic synthesis of numerical codes for solving partial differential equations. *Mathematics and Computers in Simulation* 45: 3–22.
- Amberg, G., R. Tonhardt, and C. Winkler. 1999. Finite element simulations using symbolic computing. *Mathematics and Computers in Simulation* 49: 257–274.
- Bartholomew-Biggs, M., S. Brown, B. Christianson, and L. Dixon. 2000. Automatic differentiation of algorithms. *Journal of Computational and Applied Mathematics* 124: 171–190.
- Beall, M., and M. Shephard. 1999. Object-oriented framework for reliable numerical simulations. *Engineering with Computers* 15: 61–72.
- Bischof, C., P. Hovland, and B. Norris. 2002. Implementation of automatic differentiation tools. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, ed. C. Norris, and J.J.B. Fenwick. New York: ACM Press.
- Bischof, C., H.M. Buecker, B. Lang, A. Rasch, and J.W. Risch. 2003. Extending the functionality of the general-purpose finite element package sepran by automatic differentiation. *International Journal for Numerical Methods in Engineering* 58: 2225–2238.
- Choi, K.K., and N.H. Kim. 2005a. *Structural sensitivity analysis and optimization 1, Linear systems*. New York: Springer Science+Business Media.
- Choi, K.K., and N.H. Kim. 2005b. *Structural sensitivity analysis and optimization 2, Nonlinear systems and applications*. New York: Springer Science+Business Media.
- Eyheramendy, D., and T. Zimmermann. 2000. Object-oriented symbolic derivation and automatic programming of finite elements in mechanics. *Engineering with Computers* 15(1): 12–36.
- Fritzson, P., and D. Fritzson. 1984. The need for high-level programming support in scientific computing applied to mechanical analysis. *Computers and Structures* 45: 387–395.
- Gallopoloulos, E., E. Houstis, and J. Rice. 1994. Problem-solving environments for computational science. *IEEE Computational Science in Engineering* 1: 11–23.
- Gonnet, G. 1986. New results for random determination of equivalence of expression. In *Proceedings of 1986 ACM Symposium on Symbolic and Algebraic Computation*, ed. B.W. Char, 127–131. Waterloo, ON: ACM.
- Griewank, A. 2000. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. Philadelphia: SIAM.
- Griewank, A., and A. Walther. 2008. *Evaluating derivatives: principles and techniques of algorithmic differentiation*, 2nd ed. Philadelphia: SIAM.
- Hudobivnik, B., and J. Korelc. 2016. Closed-form representation of matrix functions in the formulation of nonlinear material models. *Finite Elements in Analysis and Design* 111: 19–32.
- Keulen, F., R. Haftka, and N. Kim. 2005. Review of options for structural design sensitivity analysis. part 1: Linear systems. *Computer Methods in Applied Mechanics and Engineering* 194: 3213–3243.
- Kirby, R.C., M. Knepley, A. Logg, and L.R. Scott. 2005. Optimizing the evaluation of finite element matrices. *SIAM Journal on Scientific Computing* 27: 741–758.
- Kleiber, M., H. Antúnez, T. Hien, and P. Kowalczyk. 1997. *Parameter sensitivity in nonlinear mechanics*. Chichester: Wiley.
- Korelc J. 1997a. Application of computer algebra systems in structural analysis. In *Proceedings of 7th Conference Computer in Structural Engineering*, pages 21–28. University of Ljubljana.
- Korelc, J. 1997b. Automatic generation of finite-element code by simultaneous optimization of expressions. *Theoretical Computer Science* 187: 231–248.
- Korelc, J. 2002. Multi-language and multi-environment generation of nonlinear finite element codes. *Engineering with Computers* 18: 312–327.
- Korelc, J. 2011. *AceGen and AceFEM user manual*. Technical report, University of Ljubljana [www.fgg.uni-lj.si/symech/](http://www.fgg.uni-lj.si/symech/).
- Korelc, J., and S. Stupkiewicz. 2014. Closed-form matrix exponential and its application in finite-strain plasticity. *International Journal for Numerical Methods in Engineering* 98: 960–987.

- Kristanic, N., and J. Korelc. 2008. Optimization method for the determination of the most unfavorable imperfection of structures. *Computational Mechanics* 42: 859–872.
- Logg, A. 2007. Automating the finite element method. *Archives of Computational Methods in Engineering* 14: 93–138.
- Logg, A.M.K.A., and G. Wells. 2012. *Automated solution of differential equations by the finite element method*. Berlin: Springer.
- Martin, W. A. 1971. Determining the equivalence of algebraic expressions by hash coding, *Proceedings of the 2nd Symposium on Symbolic and Algebraic Manipulation*, ed. S.R. Petrick, 305–310. Los Angeles, CA: ACM.
- Pironneau O., F. Hecht, and A. Hyaric. 2008. Freefem++. Technical report [www.freefem.org](http://www.freefem.org).
- Solinc, U., and J. Korelc. 2015. A simple way to improved formulation of fe2 analysis. *Computational Mechanics* 56: 905–915.
- Tan, H., T. Chang, and D. Zheng. 1991. On symbolic manipulation and code generation of a hybrid 3-dimensional solid element. *Engineering with Computers* 7: 47–59.
- van Engelen, R.A., L. Wolters, and G. Cats. 1995. *Ctadel: A Generator of Efficient Code for PDE-based Scientific Applications*. Technical report, Leiden Institute of Advanced Computer Science. <http://www.liacs.nl/TechRep/1995/tr95-26.ps.gz>.
- Wang, P.S. 1986. Finger: A symbolic system for automatic generation of numerical programs in finite element analysis. *Journal of Symbolic Computation* 2: 305–316.
- Wang P.S. 1991. Symbolic computation and parallel software. Technical report, Department of Mathematics and Computer Science, Kent State University, USA.





<http://www.springer.com/978-3-319-39003-1>

Automation of Finite Element Methods

Korelc, J.; Wriggers, P.

2016, XXVIII, 346 p. 46 illus., 10 illus. in color.,

Hardcover

ISBN: 978-3-319-39003-1