Contents

Introduction ........................................................................................................................................................................ 7

Introduction to AceGen ...................................................................................................................................................... 8
Preface .................................................................................................................................................................................. 8
General .................................................................................................................................................................................. 8
AceGen .................................................................................................................................................................................. 9
Mathematica and AceGen .................................................................................................................................................... 10
AceGen Palettes .................................................................................................................................................................. 11

AceGen Overview .............................................................................................................................................................. 13
• General AceGen Session • Assignments and expression manipulations • Symbolic-numeric interface • Program Flow Control • Special functions • Manipulating notebooks • Debugging • MathLink environment • AceGen Examples • Finite element environments

AceGen Basics .................................................................................................................................................................. 15

Standard AceGen Procedure ............................................................................................................................................. 16
Contents ................................................................................................................................................................................. 16
Introductory Example ......................................................................................................................................................... 16
• Description of Introductory Example • Description of AceGen Characteristic Steps • Step 1: Initialization • Step
2: Definition of Input and Output Parameters • Step 3: Definition of Numeric-Symbolic Interface Variables • Step 4: Description of the Problem • Step 5: Definition of Symbolic - Numeric Interface Variables • Step 6: Code Generation

Mathematica Syntax - AceGen Syntax .................................................................................................................................18

• Example 1: Assignments • Example 2: Conditional statements (If construct) • Example 3: Loops (Do construct) • Example 4: Conditional statements (Which construct)

AceGen Session Functions .........................................................................................................................................................25

• SMSInitialize • SMSModule • SMSWrite • SMSVerbatim

Examples of Multi-language Code Generation .............................................................................................................................34

• Generation of C code • Generation of MathLink code • Generation of Matlab code

User Interface ..................................................................................................................................................................................39

Contents ..........................................................................................................................................................................................39

General Description ...........................................................................................................................................................................39

Presentation of Formulas ...............................................................................................................................................................40

• Exploring the structure of the formula - Browser mode submenu • Output representations of the expressions • Polymorphism of the generated formulas - Variable tags submenu

Analyzing the structure of the program .............................................................................................................................................42

Run time debugging .........................................................................................................................................................................42

Auxiliary Variables ............................................................................................................................................................................43

Contents ..........................................................................................................................................................................................43

Expression Optimization ...................................................................................................................................................................43

• Probability of incorrect simplifications • Functions with unique signatures • Expression optimization and automatic differentiation

Types of Auxiliary Variables ...........................................................................................................................................................47

• Example: real, integer and logical variables • Example: multi-valued variables

Auxiliary Variables Functions ..........................................................................................................................................................50

• SMS • SMSV • SMSM • SMSS • SMSSimplify • SMSVariables • SMSFreeze, SMSUnFreeze • SMSFictive

Local and Non-local Operations ......................................................................................................................................................61

• General Description • SMSSmartReduce • SMSSmartRestore • SMSRestore • SMSSetAll

Program Flow Control .......................................................................................................................................................................67

Contents ..........................................................................................................................................................................................67

General Description ...........................................................................................................................................................................67

Conditionals .....................................................................................................................................................................................67

• SMSIF, SMSELSE, SMSENDIF, SMSWITCH • SMSWHICH

Loops ..............................................................................................................................................................................................78

• SMSDO, SMSENDDO • SMSRETURN, SMSBREAK, SMSCONTINUE

Symbolic-Numeric Interface ............................................................................................................................................................86

Contents ..........................................................................................................................................................................................86

General Description ...........................................................................................................................................................................86

Input Parameters ...............................................................................................................................................................................87

• SMSREAL • SMSINTEGER • SMSLOGICAL • SMSREALLIST

Output Parameters ..............................................................................................................................................................................90

• SMSFREEZE

Automatic Differentiation ..............................................................................................................................................................93

Contents ..........................................................................................................................................................................................93

Theory of Automatic Differentiation ............................................................................................................................................93

SMSD Function ...............................................................................................................................................................................95

Differentiation: Mathematica syntax versus AceGen syntax .......................................................................................................96

• Mathematica • AceGen

Automatic Differentiation Examples ..................................................................................................................................................98

• Example 1: Simple C subroutine • Example 2: Differentiation of complex program structure • Example 3: Differentiation with respect to symmetric matrix • Example 4: Differentiation with respect to sparse matrix • Example 5: Differentiation with respect to intermediate variables

Characteristic Formulæ .....................................................................................................................................................................102

• Example 1: characteristic formulæ - one subset • Example 2: characteristic formulæ - two subsets

Exceptions in Differentiation ..........................................................................................................................................................106
Specialized Functions

User Defined Functions
Numerical Root Finding
Matrix Functions
Tensor Algebra
Linear Algebra
General Utility Functions
System Utility Functions
Verification and Debugging
General Description
Contents
Example
Compressing Structures
Limitations: Incorrect structure of the program

Verification and Debugging

Contents
General Verification Procedures
Printing to Output Devices

Specialized Functions

Linear Algebra

Tensor Algebra

Numerical Root Finding

Matrix Functions

Exception Type A: Generic Example
Exception Type C: Implicit dependencies
Exception Type D: Alternative definition of partial derivatives

Limitations: Incorrect structure of the program

Verification and Debugging

Contents

Example

General Utility Functions

System Utility Functions

Compressing Structures

Limitations: Incorrect structure of the program

Verification and Debugging

Contents

Example

Arrays

Manipulating Notebooks

SMSEvaluateCellsWithTag

• Example:

SMSRecreateNotebook

• Example:

Manipulate the contents of the generated notebook
• Example SMSRecreateNotebook • Example: create a function with SMSTagInclude

Signatures of Expressions .................................................................................................................. 184
• Example 1 • Example 2

AceGen Examples ............................................................................................................................... 186
Summary of AceGen Examples .......................................................................................................... 187
Solution to the System of Nonlinear Equations .................................................................................. 188
Minimization of Free Energy .............................................................................................................. 190
Contents ............................................................................................................................................... 190

Numerical Environments .................................................................................................................. 203
Finite Element Environments ............................................................................................................ 204
Standard FE Procedure ...................................................................................................................... 207
Description of FE Characteristic Steps .......................................................................................... 207
Description of Introductory Example ............................................................................................... 207
• Data interface
AceGen input using high level I/O data management ........................................................................ 208
• Step 1: Initialization • Step 2: Element subroutine for the evaluation of tangent matrix and residual • Step 7: Post-processing subroutine • Step 8: Code Generation
AceGen input using low level I/O data management ....................................................................... 210
• Step 1: Initialization • Step 2: Element subroutine for the evaluation of tangent matrix and residual • Step 7: Post-processing subroutine • Step 8: Code Generation

Template Constants .......................................................................................................................... 213
Contents ............................................................................................................................................... 213
• SMSTemplate
Template Constants ............................................................................................................................ 214
• Geometry • Degrees of Freedom, K and R • Data Management • Graphics and Postprocessing • Sensitivity
Analysis • AceFEM Solution Procedure Specific • Description of the Element for AceShare • Environment Specific (FEAP, ELFEN, user defined environments, ...)

Data Structures ............................................................................................................................... 219
Contents ............................................................................................................................................... 219
IO Data Management ......................................................................................................................... 220
• General Input/Output data • Sensitivity analysis Input/Output data • Tasks Input/Output data • Example: IO data defined by SMSTemplate constants • Dynamically created IO data variables • Example: dynamically created IO data variables • Example: dynamically created multi-field element IO • Example: low-level IO data definitions
Utility IO data management commands
Integer Type Environment Data .................................................................................................... 241
• General data • Mesh input related data • Iterative procedure related data • Debugging and errors related data • Linear solver related data • Sensitivity related data • Contact related data
Real Type Environment Data ........................................................................................................... 247
Node Specification Data .................................................................................................................... 249
Node Data ......................................................................................................................................... 250
Domain Specification Data .............................................................................................................. 251
• General Data • Run Mathematica from code • Memory allocation • Mesh generation • Numerical integration • Graphics post-processing • Sensitivity analysis
Element Data ....................................................................................................................................... 260

Element Topology ........................................................................................................................... 281
Contents ............................................................................................................................................... 261
• SMCFEMTopologyData
Undefined ............................................................................................................................................... 261
Zero dimensional ............................................................................................................................... 262
One dimensional ................................................................................................................................ 262
Two dimensional .................................................................................................................................. 262
Three dimensional ............................................................................................................................ 265

Node Identification .......................................................................................................................... 268
Contents ............................................................................................................................................... 268
Node Types and Switches .................................................................................................................... 268
Appendix

MathLink, Matlab Environments ................................................................. 319

Symbol is defined in other branch of "If" construct ........................................ 318

General ........................................................................................................... 317

Message: Variables out of scope ................................................................. 317

Symbol appears outside the "If" or "Do" construct ........................................ 317

Symbol is defined in other branch of "If" construct ........................................ 318

Generated code does not compile correctly ................................................ 318

MathLink ........................................................................................................ 319
Introduction
Introduction to AceGen

Preface

© Prof. Dr.-Ing. Jože Korelc, 2006-2018
Ravnikova 4, SI-1000, Ljubljana, Slovenia
E-mail : AceProducts@fgg.uni-lj.si
http://symech.fgg.uni-lj.si

The Mathematica package AceGen is used for the automatic derivation of formulae needed in numerical procedures. Symbolic derivation of the characteristic quantities (e.g. gradients, tangent operators, sensitivity vectors, ...) leads to exponential behavior of derived expressions, both in time and space. A new approach, implemented in AceGen, avoids this problem by combining several techniques: symbolic and algebraic capabilities of Mathematica, automatic differentiation technique, automatic code generation, simultaneous optimization of expressions and theorem proving by a stochastic evaluation of the expressions. The multi-language capabilities of AceGen can be used for a rapid prototyping of numerical procedures in script languages of general problem solving environments like Mathematica or Matlab © as well as to generate highly optimized and efficient compiled language codes in FORTRAN or C. Through a unique user interface the derived formulae can be explored and analyzed.

The AceGen package also provides a collection of prearranged modules for the automatic creation of the interface between the automatically generated code and the numerical environment where the code would be executed. The AceGen package directly supports several numerical environments such as: MathLink connection to Mathematica, AceFEM is a research finite element environment based on Mathematica, FEAP © is a research finite element environment written in FORTRAN, ELFEN © and ABAQUS © are the commercial finite element environments written in FORTRAN etc.. The multi-language and multi-environment capabilities of AceGen package enable generation of numerical codes for various numerical environments from the same symbolic description. In combination with the finite element environment AceFEM the AceGen package represents ideal tool for a rapid development of new numerical models.

General

Symbolic and algebraic computer systems such as Mathematica are general and very powerful tools for the manipulation of formulae and for performing various mathematical operations by computer. However, in the case of complex numerical models, direct use of these systems is not possible. Two reasons are responsible for this fact: a) during the development stage the symbolic derivation of formulae leads to uncontrollable growth of expressions and consequently redundant operations and inefficient programs, b) for numerical implementation SAC systems can not keep up with the run-time efficiency of programming languages like FORTRAN and C and by no means with highly problem oriented and efficient numerical environments used for finite element analysis.

The following techniques which are results of rapid development in computer science in the last decades are particularly relevant when we want to describe a numerical method on a high abstract level, while preserving the numerical efficiency:

- symbolic and algebraic computations (SAC) systems,
- automatic differentiation (AD) tools,
- problem Solving Environments (PSE),
theorem proving systems (TP),
numerical libraries,
specialized systems for FEM.

AceGen

The idea implemented in AceGen is not to try to combine different systems, 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, beside the algebraic manipulation, graphics and numeric capabilities, also powerful programming languages. It is therefore quite easy to simulate other techniques inside the SAC system. An approach to 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 used for determining the equivalence of algebraic expressions, instead of the conventional pattern matching technique. AceGen was designed to approach especially hard problems, where the general strategy to efficient formulation of numerical procedures, such as analytical sensitivity analysis of complex multi-field problems, has not yet been established.

General characteristics of AceGen code generator:

- simultaneous optimization of expressions immediately after they have been derived (Expression Optimization),
- automatic differentiation technique (Automatic Differentiation, Exceptions in Differentiation),
- automatic selection of the appropriate intermediate variables,
- the whole program structure can be generated (Mathematica Syntax – AceGen Syntax),
- appropriate for large problems where also intermediate expressions can be subjected to the uncontrolled swell,
- improved optimization procedures with stochastic evaluation of expressions,
- differentiation with respect to indexed variables,
- automatic interface to other numerical environments (by using Splice command of Mathematica),
- multi-language code generation (Fortran/Fortran90, C/C++, Mathematica language, Matlab language),
- advanced user interface,
- advanced methods for exploring and debugging of generated formulae,
- special procedures are needed for Local and Non-local Operations.

The AceGen system is written in the symbolic language of Mathematica. It consists of about 300 functions and 20000 lines of Mathematica’s source code. Typical AceGen function takes the expression provided by the user, either interactively or in file, and returns an optimized version of the expression. Optimized version of the expression can result in a newly created auxiliary symbol (vi), 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 data base. The data base contains a global vector of all expressions, information about dependencies of the symbols, labels and names of the symbols, partial derivatives, etc. The data base is a global object which maintains information during the Mathematica session.

The classical way of optimizing expressions in 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 formulae 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. If the optimization is performed simultaneously, the explicit form of the 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 only by 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, numerical identity is not a mathematically rigorous proof for the identity of two expressions. Thus the correctness of the simplification can be determined only with a certain degree of probability. With regard to our experience this can be neglected in mechanical analysis when dealing with more or less ‘smooth’ functions.

Practice shows that at the research stage of the derivation of a new numerical software, different languages and different platforms are the best means for assessment of the specific performances and, of course, failures of the numerical model. Using the classical approach, re-coding of the source code in different languages would be extremely time consuming and is never done. With the symbolic concepts re-coding comes practically for free, since the code is automatically generated for several languages and for several platforms from the same basic symbolic description. The basic tests which are performed on a small numerical examples can be done most efficiently by using the general symbolic-numeric environments such as Mathematica, Maple, etc. It is well known that many design flaws such as instabilities or poor convergence characteristics of the numerical procedures can be easily identified if we are able to investigate the characteristic quantities (residual, tangent matrix, ...) on a symbolic level. Unfortunately, symbolic-numeric environments become very inefficient if we have a larger examples or if we have to perform iterative numerical procedures. In order to assess performances of the numerical procedure under real conditions the easiest way is to perform tests on sequential machines with good debugging capabilities (typically personal computers and programs written in Fortran or C language). At the end, for real industrial simulations, large parallel machines have to be used. With the symbolic concepts implemented in AceGen, the code is automatically generated for several languages and for several platforms from the same basic symbolic description.


**Mathematica and AceGen**

Since AceGen runs in parallel with Mathematica we can use all the capabilities of Mathematica. The major algebraic computations which play crucial role in the development of any numerical code are:
analytical differentiation,
- symbolic evaluation,
- symbolic solution to the system of linear equations,
- symbolic integration,
- symbolic solution to the system of algebraic equations.

Each of these operations can be directly implemented also with the built-in Mathematica functions and the result optimized by AceGen. However, by using equivalent functions in AceGen with simultaneous optimization of expressions, much larger problems can be efficiently treated. Unfortunately, the equivalent AceGen functions exist only for the ‘local’ operations (see Local and Non-local Operations).

**AceGen Palettes**

- Main AceGen palette.

- Display generated program.
Debugger palette and display.

Debugger palette

Debugger display
AceGen Overview

General AceGen Session

Standard AceGen Procedure

User Interface

SMSInitialize — start AceGen session

SMSModule — start new user subroutine

SMSWrite — end AceGen session and create source file

Assignments and expression manipulations

Auxiliary Variables

⊨ , ⊢ , ⫤ , ⊣ — assignment operators

SMSInt, SMSFreeze, SMSFictive — special assignments

SMSSimplify, SMSReplaceAll, SMSSmartReduce, SMSSmartRestore, SMSRestore, SMSVariables — auxiliary variables manipulations

Arrays

SMSArray, SMSPart, SMSReplacePart, SMSDot, SMSSum — operations with arrays

Automatic Differentiation

SMD, SMSDefineDerivative — automatic differentiation

Symbolic-numeric interface

Symbolic – Numeric Interface

SMSReal, SMSInteger, SMSLogical, SMSRealList — import from input parameters

SMSExport — export to output parameters

SMSCall — call external subroutines

Program Flow Control

Program Flow Control

SMSIf, SMSElse, SMSEndIf, SMSSwitch, SMSSWhich — conditionals

SMSDo, SMSEndDo — loop construct

SMSReturn, SMSBreak, SMSContinue.

Special functions

User Defined Functions

SMSVerbatim — include part of the code verbatim

SMSPrint, SMSPrintMessage — print to output devices from the generated code

SMSAbs, SMSSign, SMSKroneckerDelta, SMSSqrt, SMSMin, SMSMax, SMSRandom, SMSNumberQ, SMSPower, SMSTime, SMSUnFreeze — functions with random signature
SMSLinearSolve . SMSLUFactor . SMSLU Solve . SMSFactorSim . SMSInverse . SMSDet . SMSKrammer — linear algebra functions

SMSCovariantBase . SMSCovariantMetric . SMSContravariantMetric . SMGChristoffell1 . SMGChristoffell2 .
SMSTensorTransformation . SMSDCovariant — tensor algebra functions

SMSLameToHooke . SMSHookeToLame . SMSHookeToBulk . SMSBulkToHooke . SMSPlaneStressMatrix .
SMSPlaneStrainMatrix . SMSEigenvalues . SMSMatrixExp . SMSInvariantsI . SMSInvariantsJ — mechanics of solids functions

**Manipulating notebooks**

SMSEvaluateCellsWithTag — evaluate all notebook cells

SMSRecreateNotebook — create new notebook that includes only evaluated cells

SMSTagIf . SMSTagSwitch . SMSTagReplace — manipulate break points

**Debugging**

Verification and Debugging

SMSSetBreak — insert break point

SMSLoadSession — reload the data and definitions for debugging session

SMSClearBreak . SMSActivateBreak — create new notebook that includes only evaluated parts

**MathLink environment**

SMSInstallMathLink . SMSLinkNoEvaluations . SMSSetLinkOptions . Solution to the System of Nonlinear Equations — create installable MathLink Program from generated C code

**AceGen Examples**


**Finite element environments**

Template Constants

Element Topology

Numerical Integration

Data Structures

Standard User Subroutines
AceGen Basics
Standard AceGen Procedure

Contents

- Introductory Example
  - Description of Introductory Example
  - Description of AceGen Characteristic Steps
- Step 1: Initialization
- Step 2: Definition of Input and Output Parameters
- Step 3: Definition of Numeric - Symbolic Interface Variables
- Step 4: Description of the Problem
- Step 5: Definition of Symbolic - Numeric Interface Variables
- Step 6: Code Generation

Mathematica Syntax – AceGen Syntax

- Example 1: Assignments
- Example 2: Conditional statements (If construct)
- Example 3: Loops (Do construct)
- Example 4: Conditional statements (Which construct)

AceGen Session Functions

- SMSInitialize
- SMSModule
- SMSWrite
- SMSVerbatim

Examples of Multi-language Code Generation

- Generation of C code
- Generation of MathLink code
- Generation of Matlab code

Introductory Example

Description of Introductory Example

Let us consider a simple example to illustrate the standard AceGen procedure for the generation of a typical numerical sub-program that returns gradient of a given function $f$ with respect to the set of parameters. Let unknown function $u$ be approximated by a linear combination of unknown parameters $u_1$, $u_2$, $u_3$ and shape functions $N_1$, $N_2$, $N_3$.

$$u = \sum_{i=1}^{3} N_i u_i$$

$$N_1 = \frac{x}{l}$$
$$N_2 = 1 - \frac{x}{l}$$
$$N_3 = \frac{x}{l} \left(1 - \frac{x}{l}\right)$$

Let us suppose that our solution procedure needs gradient of function $f = u^2$ with respect to the unknown parameters. AceGen can generate complete subprogram that returns the required quantity.
Description of AceGen Characteristic Steps

The syntax of the AceGen script language is the same as the syntax of the Mathematica script language with some additional functions. The input for AceGen can be divided into six characteristic steps.

<table>
<thead>
<tr>
<th>step</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Initialization</td>
<td>[\text{SMSInitialize[&quot;test&quot;, &quot;Language&quot; -&gt; &quot;C&quot;]}]</td>
</tr>
<tr>
<td>2 Definition of input and output parameters</td>
<td>[\text{SMSModule[&quot;Test&quot;, Real[u$$[3], x$$, L$$, g$$[3]];}]</td>
</tr>
<tr>
<td>3 Definition of numeric-symbolic interface variables</td>
<td>[{x, L} \rightarrow {\text{SMSReal}[x$$], \text{SMSReal}[L$$]}; \quad \text{ui} \rightarrow \text{SMSReal}[\text{Table}[u$$[i], {i, 3}]]; \quad \text{N} \rightarrow \frac{\text{L} - 1}{\text{L}} \cdot \frac{x}{\text{L}} \cdot (1 - \frac{x}{\text{L}}); \quad \text{u} \rightarrow \text{Ni}. \text{ui}; \quad f \rightarrow \text{ui}; \quad g \rightarrow \text{SMSD}[f, \text{ui}];]</td>
</tr>
<tr>
<td>4 Derivation of the problem</td>
<td>[\text{SMSExport}[g,g$$];]</td>
</tr>
<tr>
<td>5 Definition of symbolic-numeric interface variables</td>
<td>[\text{SMSWrite[;}]</td>
</tr>
</tbody>
</table>

Characteristic steps of the AceGen session

Due to the advantage of simultaneous optimization procedure we can execute each step separately and examine intermediate results. This is also the basic way how to trace the errors that might occur during the AceGen session.

**Step 1: Initialization**

- This loads the AceGen package.

\[\text{In[1]:=} \langle \langle \text{AceGen}\rangle \]

- This initializes the AceGen session. FORTRAN is chosen as the final code language. See also SMSInitialize.

\[\text{In[2]:=} \text{SMSInitialize["test", "Language" -> "Fortran"]};\]

**Step 2: Definition of Input and Output Parameters**

- This starts a new subroutine with the name "Test" and four real type parameters. The input parameters of the subroutine are \(u, x,\) and \(L,\) and parameter \(g\) is an output parameter of the subroutine. The input and output parameters of the subroutine are characterized by the double $ sign at the end of the name. See also Symbolic-Numeric Interface.

\[\text{In[3]:=} \text{SMSModule["Test", Real[u$$[3], x$$, L$$, g$$[3]\]];}\]

**Step 3: Definition of Numeric-Symbolic Interface Variables**

- Here the input parameters of the subroutine are assigned to the usual Mathematica variables. The standard Mathematica assignment operator = has been replaced by the special AceGen operator \(\simeq\). Operator \(\simeq\) performs stochastic simultaneous optimization of expressions. See also Auxiliary Variables, SMSReal.

\[\text{In[4]:=} \text{x} \simeq \text{SMSReal}[x$$]\]

\[\text{Out[4]=} \{\}\]

\[\text{In[5]:=} \text{L} \simeq \text{SMSReal}[L$$]\]

\[\text{Out[5]=} \{\}\]

- Here the variables \(u[1], u[2], u[3]\) are introduced.

\[\text{In[6]:=} \text{ui} \simeq \text{SMSReal}[\text{Table}[u$$[i], \{i, 3\}\]];\]

\[\text{Out[6]=} \{\text{ui}, \text{ui}[1], \text{ui}[2]\}\]
Step 4: Description of the Problem

- Here is the body of the subroutine.

\[
N_i \equiv \{ \frac{x}{L}, 1 - \frac{x}{L}, \frac{x}{L} \ast (1 - \frac{x}{L}) \}
\]

\[
\text{Out} = \{ N_i, N_i, N_i \}
\]

\[
u = N_i \cdot u_i
\]

\[
\text{Out} = \%
\]

\[
f = u^2
\]

\[
\text{Out} = \%
\]

\[
g = \text{SMSD}[f, u_i]
\]

\[
\text{Out} = \{ g_1, g_2, g_3 \}
\]

Step 5: Definition of Symbolic - Numeric Interface Variables

- This assigns the results to the output parameters of the subroutine. See also SMSExport.

\[
\text{SMSExport}[g, g];
\]

Step 6: Code Generation

- During the session AceGen generates pseudo-code which is stored into the AceGen database. At the end of the session AceGen translates the code from pseudo-code to the required script or compiled program language and prints out the code to the output file. See also SMSWrite.

\[
\text{SMSWrite[]};
\]

<table>
<thead>
<tr>
<th>File:</th>
<th>test.f</th>
<th>Size: 988</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods</td>
<td>No.Formulae</td>
<td>No.Leafs</td>
</tr>
<tr>
<td>Test</td>
<td>6</td>
<td>81</td>
</tr>
</tbody>
</table>

- This displays the contents of the generated file.
In principle we can get AceGen input simply by replacing the = operators in standard Mathematica input by an appropriate AceGen assignment operator ( ⊨, ⊢, ⫤, ⊣), the standard Mathematica conditional statements If, Which and Switch by the AceGen SMSIf, SMSSwitch, SMSWhich statements and the standard Mathematica loop statement Do by the AceGen SMSDo statement. All other conditional and loop structures have to be manually replaced by the equivalent forms consisting only of SMSIf and SMSDo statements. It is important to notice that only the replaced conditionals and loops produce corresponding conditionals and loops in the generated code and are evaluated when the generated program is executed. The conditional and loops that are left unchanged are evaluated directly in Mathematica during the AceGen session.
lhs\equiv \text{rhs} \quad \text{Evaluates and optimizes rhs and assigns the result to be the value of lhs (see also: Auxiliary Variables).}

lhs \equiv rhs_1 \quad \text{Evaluates and optimizes rhs}_1 \text{ and assigns the result to be the value of lhs. The lhs variable can appear after the initialization more than once on a left-hand side of equation (see also: Auxiliary Variables).}

lhs \rightarrow rhs_2 \quad \text{A new value rhs}_2 \text{ is assigned to the previously created variable lhs (see also: Auxiliary Variables).}

\text{lhs} \equiv \text{SMSIf}\left[\text{condition, } t, f\right] \quad \text{Creates code that evaluates } t \text{ if condition evaluates to True, and } f \text{ if it evaluates to False. The value assigned to lhs during the AceGen session represents both options (see also: Program Flow Control, CMSIf).}

\text{lhs} \equiv \text{SMSWhich}\left[\text{test}_1, \text{value}_1, \text{test}_2, \text{value}_2, \ldots\right] \quad \text{Creates code that evaluates each of the test}_i \text{ in turn, returning the value of the value}_i \text{ corresponding to the first one that yields True. The value assigned to lhs during the AceGen session represents all options (see also: Program Flow Control, CMSWhich).}

\text{lhs} \equiv \text{SMSSwitch}\left[\text{expr, form}_1, \text{value}_1, \text{form}_2, \text{value}_2, \ldots\right] \quad \text{Creates code that evaluates expr, then compares it with each of the form}_i \text{ in turn, evaluating and returning the value}_i \text{ corresponding to the first match found. The value assigned to lhs during the AceGen session represents all options (see also: Program Flow Control, CMSSwitch).}

\text{SMSDo}\left[\text{expr}, \{i, i_{\text{min}}, i_{\text{max}}, \Delta i\}\right] \quad \text{Creates code that evaluates expr with the variable } i \text{ successively taking on the values } i_{\text{min}} \text{ through } i_{\text{max}} \text{ in steps of } \Delta i \text{ (see also: Program Flow Control, CMSDo).}

\text{ lhs} \equiv \text{lhs}_0 \quad \text{An initial value lhs}_0 \text{ is first assigned to the variable. lhs is in a loop continuously changed. After the loop the variable lhs (during the AceGen session) represents all possible values.}

\text{Syntax of the basic AceGen commands.}

The control structures in \textit{Mathematica} have to be completely located inside one notebook cell (e.g. loop cannot start in once cell and end in another cell). \textit{AceGen} extends the functionality of Mathematica with the cross-cell form of If and Do control structures as presented in Program Flow Control chapter.

\textbf{Example 1: Assignments}

- \textit{Mathematica input}

  \begin{verbatim}
  In[66]:= x =.; n =.;
  y = Sin[x]^n
  Out[66]= Sin[x]^n
  \end{verbatim}

- \textit{AceGen input}

  \begin{verbatim}
  n[10] = << AceGen;  
  SMSInitialize["test", "Language" -> "C"];  
  SMSModule["Test", \text{Real}\left[x$$, y$$\right], \text{Integer}\left[n$$\right]];  
  x \equiv \text{SMSReal}[x$$];  
  n \equiv \text{SMSInteger}[n$$];  
  y \equiv \text{Sin}[x]^n;  
  \text{AceGen code profile}
  \end{verbatim}
Example 2: Conditional statements (If construct)

\[
y = \begin{cases} 
7 & x \geq 7 \\
 x & x < 7 \\
 x^2 & x < 0 
\end{cases}
\]

\[z = \sin(y) + 1\]

Mathematica input:

\[
In[315]:= y = \text{If}[x \geq 0, \text{If}[x \geq 7, 7, x], x^2];
\]

\[
Out[315]= 1 + \sin[\text{If}[V[1, 1] \geq 0, \text{If}[x \geq 7, 7, x], x^2]]
\]
AceGen input

```
In[317]:= << AceGen`;
SMSInitialize["test", "Language" -> "C"];
SMSModule["Test", Real[x$$, z$$]];
x ⊢ SMSReal[x$$];
y ⊨ SMSIf[x ≥ 0
   , SMSIf[x ≥ 7
     , 7
     , x
   ]
   , x^2
];
z ⊨ Sin[y] + 1;
```

AceGen code profile

```
x ≡ x$$
If x ≥ 0
   If |x| ≥ 7
     ≡ 7
   Else
     ≡ x
   EndIf
Else
   ≡ x^2
EndIf
z ≡ Sin[y] + 1;
```

AceGen code profile

```
In[323]:= SMSExport[z, z$$];
SMSWrite[];
FilePrint["test.c"]
```
Example 3: Loops (Do construct)

\[ z(i) = \sin(x^i) \]
\[ y = \sum_{i=1}^{n} \left( z(i) + \frac{1}{2} \right) \]

- Mathematica input

\text{Clear}[x]; n = 5;
\text{y} = \theta;
\text{Do[}
   \text{z} = \sin[x^i];
   \text{y} = \text{y} + \left( \text{z} + \frac{1}{2} \right);
   \{i, 1, n, 1\}
\text{],}
\text{y}

\text{Out[345]= Csc}[x] + \sin[x] + (2 \text{Csc}[x^2] + \sin[x^2])^2 +
(3 \text{Csc}[x^3] + \sin[x^3])^3 + (4 \text{Csc}[x^4] + \sin[x^4])^4 +
(5 \text{Csc}[x^5] + \sin[x^5])^5

- AceGen input

\text{NOTE: Upper limit} n \text{ in Do can only have specific integer value!}

\text{NOTE: Original list of arguments of Do construct \{i, 1, n, 1\} is in SMSDo extended by an additional argument \{i, 1, n, 1, y\} that provides}
information about variables that are imported into the loop and have values changed inside the loop and all variables that are defined inside the loop and used outside the loop.

```
<<AceGen;
SMSInitialize["test", "Language" -> "C#"];
SMSModule["Test", Real[x$$, y$$], Integer[n$$]];
x : SMSReal[x$$];
n : SMSInteger[n$$];
y : 0;
SMSDo[
  z : Sin[x$$];
  y : y + i + i + 1;
  (i, 1, n, 1, y);
]
```

- AceGen code profile

```
private double Power(double a, double b) { return Math.Pow(a, b); }
```

Example 4: Conditional statements (Which construct)
\[ y = \begin{cases} 
  7 & x \geq 0 \\
  x & x < 7 \\
  x^2 & x < 0 
\end{cases} \]

\[ z = \sin(y) + 1 \]

- **Mathematica input**

```mathematica
In[342]:= Clear[x];
y = Which[x \geq 0 && x \geq 7, 7, x \geq 0 && x < 7, x, x < 0, x^2]
z = \sin(y) + 1

Out[342]= Which[x \geq 0 && x \geq 7, 7, x \geq 0 && x < 7, x, x < 0, x^2]

Out[343]= 1 + \sin(Wich[x \geq 0 && x \geq 7, 7, x \geq 0 && x < 7, x, x < 0, x^2])
```

- **AceGen input**

```mathematica
In[358]:= << AceGen`
SMSInitialize["test", "Language" -> "Fortran"];
SMSModule["Test", Real[x$$, z$$]]; x = SMSReal[x$$];
y = SMSWhich[x \geq 0 && x \geq 7, 7, x \geq 0 && x < 7, x, x < 0, x^2];
z = \sin(y) + 1;
```

- **AceGen code profile**

```plaintext
Test
\begin{align*}
&= x$$ \text{True} \\
&= 7 \text{False} = \text{False} \\
&= 7 \text{False} = \text{False} \\
&= 7 \text{False} = \text{False} \\
&= 1 + \sin(y)
\end{align*}
```

```mathematica
In[364]:= SMSExport[z, z$$]; SMSWrite[]; FilePrint["test.f"]
```
SUBROUTINE Test(v,x,z)
IMPLICIT NONE
include 'sms.h'
LOGICAL b2,b3,b5,b6
DOUBLE PRECISION v(5001),x,z
b2=.true.
IF (x.ge.0d0.and.x.ge.7d0) THEN
  b2=.false.
v(4)=7d0
ELSE
ENDIF
IF (b2.and.x.ge.0d0.and.x.lt.7d0) THEN
  b2=.false.
v(4)=x
ELSE
ENDIF
IF (b2.and.x.lt.0d0) THEN
  b2=.false.
  v(4)=x**2
ELSE
ENDIF
z=1d0+dsin(v(4))
END

AceGen Session Functions

SMSInitialize

SMSInitialize[name]
start a new AceGen session with the session name name

SMSInitialize[name,opt]
start a new AceGen session with the session name name and options opt

Initialization of the AceGen system.
### Option Table

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Language&quot;</td>
<td>&quot;Mathematica&quot;</td>
<td>source code language</td>
</tr>
<tr>
<td>&quot;Environment&quot;</td>
<td>&quot;None&quot;</td>
<td>is a character constant that identifies the numerical environment for which the code is generated</td>
</tr>
<tr>
<td>&quot;VectorLength&quot;</td>
<td>500</td>
<td>length of the system vectors (very large system vectors can considerably slow down execution)</td>
</tr>
<tr>
<td>&quot;Mode&quot;</td>
<td>&quot;Optimal&quot;</td>
<td>define initial settings for the options of the AceGen functions</td>
</tr>
<tr>
<td>&quot;GlobalNames&quot;</td>
<td>{&quot;v&quot;,&quot;i&quot;,&quot;b&quot;}</td>
<td>first letter of the automatically generated auxiliary real, integer, and logical type variables</td>
</tr>
<tr>
<td>&quot;SubroutineName&quot;</td>
<td>$$&amp;$$</td>
<td>pure function applied on the names of all generated subroutines</td>
</tr>
<tr>
<td>&quot;Debug&quot;</td>
<td>for &quot;Mode&quot;: &quot;Debug&quot;\Rightarrow True &quot;Prototype&quot;\Rightarrow False &quot;Optimal&quot;\Rightarrow False</td>
<td>if True extra (time consuming) tests of code correctness are performed during derivation of formulas and also included into generated source code</td>
</tr>
<tr>
<td>&quot;Precision&quot;</td>
<td>100</td>
<td>default precision of the Signatures of Expressions</td>
</tr>
</tbody>
</table>

Options of the SMSInitialize function.

### Language Description

<table>
<thead>
<tr>
<th>Language</th>
<th>Description</th>
<th>Generic name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Fortran&quot;</td>
<td>fixed form</td>
<td>&quot;Fortran&quot;</td>
</tr>
<tr>
<td>&quot;Fortran90&quot;</td>
<td>free form</td>
<td>&quot;Fortran&quot;</td>
</tr>
<tr>
<td>&quot;Mathematica&quot;</td>
<td>code written in Mathematica programming language</td>
<td>&quot;Mathematica&quot;</td>
</tr>
<tr>
<td>&quot;C&quot;</td>
<td>ANSI C code</td>
<td>&quot;C&quot;</td>
</tr>
<tr>
<td>&quot;C++&quot;</td>
<td>ANSI C++ code</td>
<td>&quot;C&quot;</td>
</tr>
<tr>
<td>&quot;Matlab&quot;</td>
<td>standard Matlab &quot;M&quot; file</td>
<td>&quot;Matlab&quot;</td>
</tr>
</tbody>
</table>

Supported languages.

### Mode

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Plain&quot;</td>
<td>all Expression Optimization procedures are excluded</td>
</tr>
<tr>
<td>&quot;Debug&quot;</td>
<td>options are set for the fastest derivation of the code, all the expressions are included into the final code and preceded by the explanatory comments</td>
</tr>
<tr>
<td>&quot;Prototype&quot;</td>
<td>options are set for the fastest derivation of the code, with moderate level of code optimization</td>
</tr>
<tr>
<td>&quot;Optimal&quot;</td>
<td>options are set for the generation of the fastest and the shortest generated code (it is used to make a release version of the code)</td>
</tr>
</tbody>
</table>

Supported optimization modes.
<table>
<thead>
<tr>
<th>environment</th>
<th>Language</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;None&quot;</td>
<td>defined by &quot;Language&quot; option</td>
<td>plain code</td>
</tr>
<tr>
<td>&quot;MathLink&quot;</td>
<td>&quot;C&quot;</td>
<td>the program is build from the generated source code and installed (see MathLink, Matlab Environments) so that functions defined in the source code can be called directly from Mathematica (see Standard AceGen Procedure, SMSInstallMathLink)</td>
</tr>
<tr>
<td>&quot;User&quot;</td>
<td>defined by &quot;Language&quot; option</td>
<td>arbitrary user defined finite element environment (see Standard FE Procedure, User Defined Environment Interface)</td>
</tr>
<tr>
<td>&quot;AceFEM&quot;</td>
<td>&quot;C&quot;</td>
<td>Mathematica based finite element environment with compiled numerical module (element codes and computationally intensive parts are written in C and linked with Mathematica via the MathLink protocol) (see Standard FE Procedure)</td>
</tr>
<tr>
<td>&quot;AceFEM-MDriver&quot;</td>
<td>&quot;Mathematica&quot;</td>
<td>AceFEM finite element environment with symbolic numerical module (elements and all procedures written entirely in Mathematica’s programming language) (see Standard FE Procedure)</td>
</tr>
<tr>
<td>&quot;FEAP&quot;</td>
<td>&quot;Fortran&quot;</td>
<td>research finite element environment written in FORTRAN (see FEAP – ELFEN – ABAQUS – ANSYS)</td>
</tr>
<tr>
<td>&quot;ELFEN&quot;</td>
<td>&quot;Fortran&quot;</td>
<td>commercial finite element environment written in FORTRAN (see FEAP – ELFEN – ABAQUS – ANSYS)</td>
</tr>
<tr>
<td>&quot;ABAQUS&quot;</td>
<td>&quot;Fortran&quot;</td>
<td>commercial finite element environment written in FORTRAN (see FEAP – ELFEN – ABAQUS – ANSYS)</td>
</tr>
</tbody>
</table>

Currently supported numerical environments.

In a "Debug" mode all the expressions are included into the final code and preceded by the explanatory comments. Derivation of the code in a "Optimal" mode usually takes 2-3 times longer than the derivation of the code in a "Prototype" mode.

- This initializes the AceGen system and starts a new AceGen session with the name "test". At the end of the session, the FORTRAN code is generated.

\[\text{In[2]}=\text{SMSInitialize["test", "Language" -> "Fortran"]};\]

**SMSModule**

- **SMSModule[name]**
  start a new module with the name name without input/output parameters

- **SMSModule[name, type1[p_{11}, p_{12}, ...], type2[p_{21}, p_{22}, ...], ...]**
  start a new module with the name name and a list of input/output parameters $p_{11}, p_{12}, ..., p_{21}, p_{22}$ of specified types (see Symbolic-Numeric Interface)

Syntax of SMSModule function.
parameter types  

Real\([p_1, p_2, \ldots]\)  

Integer\([p_1, p_2, \ldots]\)  

Logical\([p_1, p_2, \ldots]\)  

typename\([p_1, p_2, \ldots]\)  

Automatic\([p_1, p_2, \ldots]\)  

Types of input/output parameters.

The name of the module (method, subroutine, function, ...) name can be arbitrary string or Automatic. In the last case AceGen generates an unique name for the module composed of the session name and an unique number. All the parameters should follow special AceGen rules for the declaration of external variables as described in chapter Symbolic-Numeric Interface. An arbitrary number of modules can be defined within a single AceGen session. An exception is Matlab language where the generation of only one module per AceGen session is allowed.

---

<table>
<thead>
<tr>
<th>option</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Verbatim&quot;-&gt;Start</td>
<td>None</td>
<td>string or a list of strings Start is included at the end of the declaration block of the source code verbatim.</td>
</tr>
<tr>
<td>&quot;VerbatimEnd&quot;-&gt;End</td>
<td>None</td>
<td>string or a list of strings End is included at the end of the subroutine code verbatim.</td>
</tr>
<tr>
<td>&quot;Input&quot;</td>
<td>All</td>
<td>list of input parameters.</td>
</tr>
<tr>
<td>&quot;Output&quot;</td>
<td>All</td>
<td>list of output parameters.</td>
</tr>
</tbody>
</table>

Options of the SMSModule function.

---

<table>
<thead>
<tr>
<th>substitution</th>
<th>character</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>&quot;</td>
</tr>
<tr>
<td>/</td>
<td>\</td>
</tr>
<tr>
<td>/'</td>
<td>'</td>
</tr>
<tr>
<td>/&quot;</td>
<td>&quot;</td>
</tr>
<tr>
<td>/\n</td>
<td>\n</td>
</tr>
</tbody>
</table>

Character substitution table for "Verbatim" and "VerbatimEnd" strings.

---

By default all the parameters are labeled as input/output parameters. The "Input" and the "Output" options are used in MathLink (see Standard AceGen Procedure) and Matlab to specify the input and the output parameters.

The SMSModule command starts an arbitrary module. However, numerical environments usually require a standardized set of modules (traditionally called "user defined subroutines") that are used to perform specific task (e.g. to calculate tangent matrix) and with a strict set of I/O parameters. The SMSStandardModule command (see Standard User Subroutines) can be used instead of SMSModule for the definition of the standard user subroutines for supported finite element numerical environments.

**Example 1**

- This creates a Fortran subroutine named "sub1" with real parameters \(x, z\), real type array \(y(5)\), integer parameter \(i\), and parameter \(m\) of the user defined type 'mytype'.

---
Example 2

Create a C subroutine that calculates series $r_{ij} = \sum_{k=1}^{n} x_{ik}^j$, $i = 1, \ldots, n$; $j = 1, \ldots, m$.

- Solution 1: elements of the series are calculated and added to globally defined $r_{ij}$ directly. $r_{ij}$ is allocated on heap.
\[\text{In[481]} = << \text{AceGen';}\]
\begin{verbatim}
SMSInitialize["test", "Environment" -> "MathLink"];
SMSModule["test", Real[n$$[n$$, m$$]], Integer[n$$, m$$, k$$],
   Real[r$$[n$$, m$$]], "Input" -> \{x$$, m$$, k$$\}, "Output" -> r$$];
\end{verbatim}
\( (n, m, k) = \text{SMSInteger[\{n$$, m$$, k$$\}];} \)
\begin{verbatim}
SMSDo[
   SMSDo[
      SMSExport[SMSPower[SMSSerial[x$$[i, j], s], r$$[i, j], "AddIn" -> True]
         , \{(s, 1, k)\};
      , \{(i, 1, n)\};
      , \{(j, 1, m)\};
   ];
   SMSWrite[];
]
\end{verbatim}
\begin{table}[h]
  \begin{tabular}{|c|c|}
    \hline
    Method & test \\
    \hline
    No.\ Formulae & 2 \\
    No.\ Leafs & 38 \\
    \hline
  \end{tabular}
\end{table}
\begin{verbatim}
File: test.c  Size: 2538  Time: 0
\end{verbatim}

\[\text{Out[487]} = \text{SMSInstallMathLink[]}\]

\[\text{In[488]} = n = 100; m = 100; k = 1000;
   \text{Total[\{x$$[\text{Table}1. / (i + j), \{i, n\}, \{j, m\}], n, m, k\}, 2]]}
\]
\[\text{Out[489]} = 138.131\]

\[\text{In[490]} = \text{SMSUninstallMathLink[]}\]

- Solution 2: elements of the series are calculated and added to locally defined \( r_{ij} \) allocated on stack. The result is then exported to global \( r_{ij} \) allocated on heap. This can be numerically more efficient due to the local memory access, however stack memory is limited!

\[\text{In[491]} = << \text{AceGen'};\]
\begin{verbatim}
SMSInitialize["test", "Environment" -> "MathLink"];
SMSModule["test", Real[n$$[n$$, m$$]], Integer[n$$, m$$, k$$],
   Real[r$$[n$$, m$$]], "Input" -> \{x$$, m$$, k$$\}, "Output" -> r$$];
   "Verbatim" -> "double locr[\{n\}[\{m\}];
\end{verbatim}
\begin{verbatim}
int ii,jj;
for (ii = 0;ii < (*n);ii++)
   for (jj = 0;jj < (*m);jj++)
      locr[ii][jj]=0e0;"
   , "VerbatimEnd" -> "for (ii=0;ii<(*n);ii++)
for (jj=0;jj<(*m);jj++)
   r[ii][jj]=locr[ii][jj];"
];
\end{verbatim}
\( (n, m, k) = \text{SMSInteger[\{n$$, m$$, k$$\}];} \)
\begin{verbatim}
SMSDo[
   SMSDo[
      SMSPower[SMSSerial[x$$[i, j], s], locr$$[i, j], "AddIn" -> True]
         , \{(s, 1, k)\};
      , \{(i, 1, n)\};
      , \{(j, 1, m)\};
   ];
   SMSWrite[];
]
\end{verbatim}
\begin{table}[h]
  \begin{tabular}{|c|c|}
    \hline
    Method & test \\
    \hline
    No.\ Formulae & 2 \\
    No.\ Leafs & 38 \\
    \hline
  \end{tabular}
\end{table}
\begin{verbatim}
File: test.c  Size: 2711  Time: 0
\end{verbatim}
\begin{verbatim}
In[477]:= SMSInstallMathLink[];

In[478]:= n = 100; m = 100; k = 1000;
    Total[test[Table[1./(i + j), {i, n}, {j, m}], n, m, k], 2]
Out[478]= 138.131

In[480]:= SMSUninstallMathLink[];

SMSWrite

SMSWrite[]
write source code in the file "session_name.ext"

SMSWrite[filename, opt]
write source code in the file filename
Create automatically generated source code file.

language  file extension
"Fortran"  name.f
"Fortran90" name.f90
"Mathematica" name.m
"C"  name.c
"C++" name.cpp
"Matlab" name.m

File extensions.
\end{verbatim}
<table>
<thead>
<tr>
<th>option</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Splice&quot;</td>
<td>{}</td>
<td>list of files interpreted (see FileTemplate) and prepended to the generated source code file (in the case of standard numerical environment a special interlace file is added to the list automatically)</td>
</tr>
<tr>
<td>&quot;Substitutions&quot;</td>
<td>{}</td>
<td>list of rules applied on all expressions before the code is generated (see also User Defined Functions)</td>
</tr>
<tr>
<td>&quot;IncludeNames&quot;</td>
<td>False</td>
<td>the name of the auxiliary variable is printed as a comment before definition</td>
</tr>
<tr>
<td>&quot;IncludeAllFormulas&quot;</td>
<td>False</td>
<td>also the formulae that have no effect on the output parameters of the generated subroutines are printed</td>
</tr>
<tr>
<td>&quot;OptimizingLoops&quot;</td>
<td>1</td>
<td>number of additional optimization loops over the whole code</td>
</tr>
<tr>
<td>&quot;IncludeHeaders&quot;</td>
<td>{}</td>
<td>additional header files to be included in the declaration block of all generated subroutines (INCLUDE in Fortran and USE in Fortran90) or in the head of the C file. Default headers are always included as follows: &quot;Fortran&quot; =&gt; {&quot;sms.h&quot;}, &quot;Fortran90&quot; =&gt; {&quot;SMS&quot;}, &quot;Mathematica&quot; =&gt; {}, &quot;C&quot; =&gt; {&quot;sms.h&quot;}, &quot;C++&quot; =&gt; {&quot;sms.h&quot;}</td>
</tr>
<tr>
<td>&quot;MaxLeafCount&quot;</td>
<td>3000</td>
<td>large Fortran expressions are split into subexpressions of the size less than &quot;MaxLeafCount&quot; due to the limitations of Fortran compilers (size is measured by the LeafCount function)</td>
</tr>
<tr>
<td>&quot;LocalAuxiliaryVariables&quot;</td>
<td>False</td>
<td>True =&gt; The vector of auxiliary variables is defined locally for each module. False =&gt; The vector of auxiliary variables is to be an argument of the subroutines, e.g. double *v. Automatic =&gt; For standard numerical environments the vector of auxiliary variables is defined locally for each module if length of working vector is less than 25000. Integer =&gt; For standard numerical environments the vector of auxiliary variables is defined locally for each module if length of working vector is less than given value.</td>
</tr>
<tr>
<td>&quot;WorkingVectorSize&quot;</td>
<td>Automatic</td>
<td>The length of vector where AceGen generated auxiliary variables are stored in v vector (CDriver, ABAQUS, etc.). In some cases (e.g. for backward differentiation of complex program structures, call to external subroutines) significant storage might be required. Default value is set by FE environment. E.g. CDriver sets it to 100000. (see also General Data). Automatic =&gt; only when there are no variable length arrays / Integer =&gt; set by user / Function =&gt; AceGen make an estimate how large the vector should be for all generated subroutines. Function f[list of estimates] is applied on that list to get the actual WorkingVectorSize. E.g. if an input data has constants &quot;m&quot;, &quot;n&quot; and we know that they cannot be larger than 10 then &quot;WorkingVectorSize&quot;-&gt;Function[ lengths],Max[lengths/ (.&quot;m&quot;-&gt;10,&quot;n&quot;-&gt;10)].</td>
</tr>
</tbody>
</table>

Options of the SMSWrite function.

The "splice-file" is arbitrary text file that is first interpreted by the Mathematica's FileTemplate command and then prepended to the automatically generated source code file. Options "IncludeNames" and "IncludeAllFormulas" are useful during the 'debugging' period. They have effect only in the case that AceGen session was initiated in the "Debug" or "Prototype" mode. Option "OptimizingLoops" has effect only in the case that AceGen session was initiated in the "Optimal" or a higher mode.

The default header files are located in $BaseDirectory/Applications/AceGen/Include/ directory together with the collection of utility routines (SMSUtility.c and SMSUtility.f). The header files and the utility subroutines should be available during the compilation of the generated source code.

See also: Standard AceGen Procedure
This writes the generated code on the file "source.c" and prepends contents of the file "test.mc" interpreted by the Splice command.

```plaintext
In[8]= <<AceGen;

strm = OpenWrite["test.mc"]; 
WriteString[strm,"/*This is a \"splice\" file <<100+1>> */"]; 
Close[strm]; 

In[9]= FilePrint["test.mc"]
    /*This is a "splice" file <<100+1>> */

In[10]= SMSInitialize["test", "Language" -> "C"]; 
SMSModule["sub1", Real[x$, y$[2]]]; 
SMSExport[BesselJ[SMSReal[y$[1]], SMSReal[y$[2]]]], x$]; 
SMSWrite["source","Splice" -> "test.mc", "Substitutions"->{BesselJ[1,1]:"mybessel"[i,j]}];

Method : Sub1 1 formulae, 13 sub-expressions

File created : source.c Size : 742

In[15]= FilePrint["source.c"]

/*******************
S U B R O U T I N E
*******************/
void sub1(double v[5001],double (*x),double y[2])
{
    (*x)=mybessel(y[0],y[1]);
}
```

---

**SMSVerbatim**

SMSVerbatim[source]  
write textual form of the source into the automatically generated code verbatim

SMSVerbatim[language1->source1,language2->source2,...]  
write textual form of the source which corresponds to the currently used program language into the automatically generated file verbatim
The SMSVerbatim command automatically adds a separator character at the end of the code (e.g. ;" in the case of C++). With the option "Close"->False, no character is added.

List of symbols used within the code that have to have declared data type [e.g. "DeclareSymbol"->\{Real[x$$,y$$],Integer[n$$]\}].

Since the effect of the SMSVerbatim statement can not be always predicted, the verbatim code is always placed at the position in code from where the SMSVerbatim command is called. With "AddStructure"->Automatic, the optimal position of the verbatim code is determined automatically accordingly to the dependency of the code on other variables.

Input parameters source, source_2, source_3,... have special form. They can be a single string, or a list of arbitrary expressions. Expressions can contain auxiliary variables as well. Since some of the characters (e.g. ") are not allowed in the string we have to use substitution instead accordingly to the table below.

<table>
<thead>
<tr>
<th>substitution</th>
<th>character</th>
</tr>
</thead>
<tbody>
<tr>
<td>'</td>
<td>&quot;</td>
</tr>
<tr>
<td>/</td>
<td>\</td>
</tr>
<tr>
<td>/'</td>
<td>'</td>
</tr>
<tr>
<td>[/n</td>
<td>\n</td>
</tr>
</tbody>
</table>

Character substitution table.

The parameter "language" can be any of the languages supported by AceGen ("Mathematica", "Fortran", "Fortran90", "C", "C++", ...). It is sufficient to give a rule for the generic form of the language ("Mathematica", "Fortran", "C") (e.g instead of the form for language "Fortran90" we can give the form for language "Fortran").

The source can contain arbitrary program sequences that are syntactically correct for the chosen program language, however the source is taken verbatim and is neglected during the automatic differentiation procedure unless additional declarations are included as presented in User Defined Functions.

```
In[18]= SMSInitialize["test", "Language"->"C"];
SMSModule["test"];
SMSVerbatim[
    "Fortran"->{"write(*,*) 'Hello', \nstop"},
    "Mathematica"->{"Print['Hello'];", "\nAbort[];"},
    "C"->{"printf('Hello');", "\nexit(0);"}]
];
SMSWrite["test"];
```

Method: test 1 formulae, 2 sub-expressions

[0] File created: test.c Size : 683
In[20]:= FilePrint["test.c"]

/** **********************************************************
* AceGen 2.103 Windows (18 Jul 08)  *
* Co. J. Korelc 2007 18 Jul 08 15:41:07 *
**********************************************************/

User: USER
Evaluation time: 0 s Mode: Optimal
Number of formulae: 1 Method: Automatic
Subroutine: test size: 2
Total size of Mathematica code: 2 subexpressions
Total size of C code: 122 bytes

#include "sms.h"

/***********************
SUBROUTINE *******************/

t

/**
 * FilePrint["test.c"]
 */

For more examples see User Defined Functions.

Examples of Multi-language Code Generation

Generation of C code

- Instead of the step by step evaluation, we can run all the session at once. This time the C version of the code is generated.

In[14]:= <<AceGen
SMSInitialize["test", "Language" -> "C"]
SMSModule["Test", Real[u$$[3], x$$, L$$, g$$[3]]];
(x, L) -> SMSReal[x$$, SMSReal[L$$]];
u1 -> SMSReal[Table[u$$[i], {i, 3}]]
Ni -> \{x / L, 1 - x / L, \frac{1 - x}{L}\};
u = Ni.ui;
f = u^2;
g = MSD[f, ui];
SMSExport[g, g$$];
SMSWrite[];

Method -> Test 6 formulae, 81 sub-expressions

[0] File created: test.c Size: 863
Generation of MathLink code

- Here the MathLink (MathLink, Matlab Environments) version of the source code is generated. The generated code is automatically enhanced by an additional modules necessary for the proper MathLink connection.
Here the MathLink program `Test.exe` is build from the generated source code and installed so that functions defined in the source code can be called directly from Mathematica (see also SMSInstallMathLink).
Here the generated executable is used to calculate gradient for the numerical test example.

In[39]:= Test[0., 1., 7.], π // N, 10.

Out[39]= {1.37858, 3.00958, 0.945489}

Generation of Matlab code

- The AceGen generated M-file functions can be directly imported into Matlab. Here the Matlab version of the source code is generated.
function g=Test(u,x,L);
persistent v;
if size(v)<5001
    v=zeros(5001,'double');
end;
v(6)=x/L;
v(7)=1e0-v(6);
v(8)=v(6)*v(7);
v(9)=u(1)+v(6)+u(2)+v(7)+u(3)*v(8);
v(15)=2e0*v(9);
g(1)=v(15)+v(6);
g(2)=v(15)+v(7);
g(3)=v(15)+v(8);
end  

function [x]=SMKDelta(i,j)
if (i==j) , x=1; else x=0; end;
end  

function [x]=SMDeltaPart(a,i,j,k)
l=round(i/j);
if (mod(i,j) ~= 0 | l>k) , x=0; else x=a(l); end;
end  

function [x]=Power(a,b)
x=a^b;
end  

end
User Interface

Contents

- General Description
- Presentation of Formulas
  - Exploring the structure of the formula - Browser mode submenu
  - Output representations of the expressions
  - Polymorphism of the generated formulae - Variable tags submenu
- Analyzing the structure of the program
- Run time debugging

General Description

An important question arises: how to understand the automatically generated formulae? The automatically generated code 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. While formulae are derived automatically with AceGen, AceGen tries to find the actual meaning of the auxiliary variables and assigns appropriate names. By asking Mathematica in an interactive dialog about certain symbols, we can retain this information and explore the structure of the generated expressions. In the following AceGen sessions various possibilities how to explore the structure of the program are presented.

Test Example

Let start with the subprogram that returns solution to the system of the following nonlinear equations

$$
\Phi = \begin{cases} 
axy + x^3 = 0 \\
ax - xy^2 = 0 
\end{cases}
$$

where $x$ and $y$ are unknowns and $a$ is the parameter using the standard Newton-Raphson iterative procedure. The SMSSetBreak function inserts the breaks points with the identifications "X" and "A" into the generated code.
In[104]:= <<AceGen`; SMSInitialize["test", "Language" -> "Mathematica", "Mode" -> "Debug"]; SMSModule["test", Real[x$$, y$$, a$$, to1$$], Integer[nmax$$]]; 
{x0, y0, a, e} + SMSReal[[x$$, y$$, a$$, to1$$]]; nmax + SMSInteger[nmax$$];
{x, y} = {x0, y0}; SMSDo[
  \[ \begin{align*}
  \varepsilon &= \{a \cdot x \cdot y + x^3, a - x \cdot y^2\}; \\
  \text{Kt} &= \text{SMSD}[[\varepsilon, \{x, y\}]]; \\
  (\Delta x, \Delta y) &= \text{SMSSolve}[[\text{Kt}, -\varepsilon]]; \\
  \{x, y\} &= \{x, y\} + (\Delta x, \Delta y); \\
  \text{SMSIf}[\text{SMSSqrt}[[\Delta x, \Delta y]].(\Delta x, \Delta y)] < \varepsilon, \\
   \text{SMEExport}[[\{x, y\}, \{x$$, y$$\}]]; \\
   \text{SMSBreak}[];
\end{align*} \]
]; SMSIf[i == nmax, SMSPrintMessage["no convergion"]; SMSReturn[]]; 
{i, 1, nmax, 1, \{x, y\}}]; SMSWrite[];

\begin{align*}
\text{time} &= 0 \\
\text{variable} &= 0 \\
\end{align*}

\begin{tabular}{|c|c|c|}
\hline
\text{File:} & \text{test.m} & \text{Size:} & 2401 \\
\hline
\text{Methods} & \text{No.Formulae} & \text{No.Leafs} \\
\hline
\text{test} & 31 & 194 \\
\hline
\end{tabular}

\section*{Presentation of Formulas}

Exploring the structure of the formula - Browser mode submenu

\textit{AceGen} palette offers buttons that control how expressions are represented on a screen.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{palettes.png}
\caption{Palette for entering \textit{AceGen} commands that control user-AceGen interactions.}
\end{figure}

Auxiliary variables are represented as active areas (buttons) of the output form of the expressions in blue color. When we point with the mouse on one of the active areas, a new cell in the notebook is generated and the definition of the pointed variable will be
displayed. Auxiliary variables are again represented as active areas and can be further explored. Definitions of the external variables are displayed in red color. The ";' character is used to indicate derivatives (e.g. $\Phi_1\xi = \frac{\partial \Phi_1}{\partial \xi}$).

```
In[112]:= Kt[[1, 1]]
Out[112]=
```

There are two possibilities how the new cell is generated. The first possibility is that the new cell contains only the definition of the pointed variable.

```
Button: Expand selected variable

In[113]:= Kt
Out[113]=
```

The new cell can also contain the whole expression from the original cell and only pointed variable replaced by its definition.

```
Button: Expand selected cell

In[114]:= Kt
Out[114]=
```

The new cell can also contain detailed information about selected variables.

```
Button: Analyze selected variable

In[115]:= Kt
Out[115]=
```
Output representations of the expressions

Expressions can be displayed in several ways. The way how the expression is displayed does not affect the internal representation of the expression.

**StandardForm**

The most common is the representation of the expression where the automatically generated name represents particular auxiliary variable.

```
In[116]:= Kt
```

**FullForm**

The "true" or FullForm representation is when j-th instance of the i-th auxiliary variable is represented in a form $SV[i,j]$. In an automatically generated source code the i-th term of the global vector of auxiliary variables ($v(i)$) directly corresponds to the $SV[i,j]$ auxiliary variable.

```
In[117]:= Kt
```

**CondensedForm**

If variables are in a FullForm they can not be further explored. Alternative representation where j-th instance of the i-th auxiliary variable is represented in a form ""j"¥"i" enables us to explore FullForm of the automatically generated expressions.

```
In[118]:= Kt
```

**NumberForm**

Auxiliary variables can also be represented by their signatures (assigned random numbers) during the AceGen session or by their current values during the execution of the automatically generated code. This type of representation is used for debugging.

```
In[119]:= Kt
Out[119]= { { 0.75654875, 0.17561250 } , { -$0.55569743$, -$0.54195924$ } }
```

**Polymorphism of the generated formulae - Variable tags submenu**

Sometimes AceGen finds more that one meaning (tag) for the same auxiliary variable. By default it displays the shortest tag (Show shortest tag).

```
In[121]:= Kt
```

By pressing button Show first tag the last found meaning (name) of the auxiliary variables will be displayed.

```
In[122]:= Kt
```
All meanings (names) of the auxiliary variables can also be explored.

\[ K_t \]

\[ \text{Out[124]} = \{ \{ K_{t1}, \Phi_{1;x} \}, \{ -K_{t2}, K_{t2}, \Phi_{1;y} \} \} \]

**Analyzing the structure of the program**

The **Open code profile** button can be used in order to produce separate window where the structure of the program is displayed together with the links to all generated formulae.

**Run time debugging**

The **code profile window** is also used for the run-time debugging. See **Run Time Debugging** section for details.
Auxiliary Variables

Contents

- Expression Optimization
  - Probability of incorrect simplifications
  - Functions with unique signatures
  - Expression optimization and automatic differentiation
- Types of Auxiliary Variables
  - Example: real integer and logical variables
  - Example: multi-valued variables
- Auxiliary Variables Functions
  - SMSR
  - SMSV
  - SMSM
  - SMSS
  - SMSSimplify
  - SMSVariables
  - SMSFreeze
  - SMSUnFreeze
  - SMSFictive
- Local and Non-local Operations
  - SMSSmartReduce
  - SMSSmartRestore
  - SMSRestore
  - SMSReplaceAll

Expression Optimization

The basic approach to optimization of the automatically generated code is to search for the parts of the code that when evaluated yield the same result and substitute them with the new auxiliary variable. In the case of the pattern matching approach only sub-expressions that are syntactically equal are recognized as "common sub-expressions".


Signatures of the expressions are basis for the heuristic algorithm that can search also for complex relations among the expressions. The relations between expressions which are automatically recognized by the AceGen system are:
(a) two expressions or sub-expressions are the same
\[ e_1 \equiv e_2 \implies v_1 := e_1 \]
\[ e_2 \implies v_1 \]

(b) result is an integer value
\[ e_1 \equiv Z \implies e_1 \equiv Z \]

(c) opposite value
\[ e_1 \equiv -e_2 \implies v_1 := e_1 \]
\[ e_2 \implies -v_1 \]

(d) intersection of common parts for multiplication and addition
\[ a_{1 \ldots i} \ast b_{1 \ldots j} \implies v_1 := b_{1 \ldots j} \]
\[ c_{1 \ldots k} \ast d_{1 \ldots j} \implies a_{1 \ldots i} \ast b_{1 \ldots j} \Rightarrow a_{1 \ldots i} \ast v_1 \]
\[ b_n \equiv d_n \]
\[ c_{1 \ldots k} \ast d_{1 \ldots j} \Rightarrow c_{1 \ldots k} \ast v_1 \]

(e) inverse value
\[ e_1 \equiv \frac{1}{e_2} \implies v_1 := e_2 \]
\[ e_2 \Rightarrow \frac{1}{v_1} \]

In the formulae above, \( e_i, a_i, b_i, c_i, d_i \) are arbitrary expressions or sub-expressions, and \( v_i \) are auxiliary variables. Formula \( e_i \equiv e_j \) means that the signature of the expression \( e_i \) is identical to the signature of the expression \( e_j \). Expressions do not need to be syntactically identical. Formula \( v_i := e_j \) means that a new auxiliary variable \( v_i \) with value \( e_j \) is generated, and formula \( e_i \Rightarrow v_j \) means that expression \( e_i \) is substituted by auxiliary variable \( v_j \).

Sub-expressions in the above cases do not need to be syntactically identical, which means that higher relations are recognized also in cases where term rewriting and pattern matching algorithms in Mathematica fail. The disadvantage of the procedure is that the code is generated correctly only with certain probability.

**Probability of incorrect simplifications**

Let us first consider the two functions \( f_1 = x^3 - x^2 + 1 \) and \( f_2 = \text{Abs}[x] + x^3 \).

```
In[17]:= Plot[[x^3 - x^2 + 1, Abs[x] + x^3], {x, -2, 2}]
```

![](image1.png)

The value of \( f_1 \) is equal to the value of \( f_2 \) only for three discrete values of \( x \). If we take random value for \( x \in [-4, 4] \), then the probability of wrong simplification is for this case is negligible, although the event itself is not impossible. The second example are functions \( f_1 = x \) and \( f_2 = \text{Abs}[x] \).

```
In[18]:= Plot[{x, Abs[x]}, {x, -2, 2}]
```

![](image2.png)

We can see that, for a random \( x \) from interval \([-4,4]\), there is 50% probability to make incorrect simplification and consequently 50% probability that the resulting automatically generated numerical code will not be correct. The possibility of wrong simplifications can
be eliminated by replacing the Abs function with a new function (e.g. SMSAbs[x]) that has unique high precision randomly generated number as a signature. Thus at the code derivation phase the SMSAbs function results in random number and at the code generation phase is translated into the correct form (Abs) accordingly to the chosen language. Some useful simplifications might be overlooked by this approach, but the incorrect simplifications are prevented.

A typical situation when probability to make incorrect simplification is always high is when a non-monotonic function and its inverse counterpart together appear as a part of the problem description.

- Example: There exist a high probability that trigonometric (Sin, Cos, Tang) or hyperbolic functions (Cosh) combined with their inverse counterpart will be incorrectly simplified. However this is not true for monotonic functions such as Sinh, and Tanh.

```
In[44]:= GraphicsGrid[{
    Plot[{r, ArcSin[Sin[r]]}, {r, -5, 5}, PlotLabel -> ArcSin[Sin[r]]],
    Plot[{r, ArcTan[Tan[r]]}, {r, -5, 5}, PlotLabel -> ArcTan[Tan[r]]],
    Plot[{r, ArcCos[Cos[r]]}, {r, -5, 5}, PlotLabel -> ArcCos[Cos[r]]],
    {Plot[{r, ArcCosh[Cosh[r]]}, {r, -5, 5}, PlotLabel -> ArcCosh[Cosh[r]]],
     Plot[{r, ArcSinh[Sinh[r]]}, {r, -5, 5}, PlotLabel -> ArcSinh[Sinh[r]]],
     Plot[{r, ArcTanh[Tanh[r]]}, {r, -5, 5}, PlotLabel -> ArcTanh[Tanh[r]]]}
}]
```

![Graphs of inverse trigonometric and hyperbolic functions](image)

### Functions with unique signatures

When the result of the evaluation of the function is a randomly generated number then by definition the function has an **unique signature**. The AceGen package provides a set of "unique signature functions" that can be used as replacements for the most critical functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMSAbs[exp]</td>
<td>absolute value of <code>exp</code></td>
</tr>
<tr>
<td>SMSSign[exp]</td>
<td>-1, 0 or 1 depending on whether <code>exp</code> is negative, zero, or positive</td>
</tr>
<tr>
<td>SMSSKroneckerDelta[i,j]</td>
<td>1 or 0 depending on whether <code>i</code> is equal to <code>j</code> or not</td>
</tr>
<tr>
<td>SMSSqrt[exp]</td>
<td>square root of <code>exp</code></td>
</tr>
<tr>
<td>SMSMin[exp1,exp2]</td>
<td>Min[exp1,exp2]</td>
</tr>
<tr>
<td>SMSMax[exp1,exp2]</td>
<td>Max[exp1,exp2]</td>
</tr>
</tbody>
</table>

E.g. the SMSSqrt should be used instead of the Mathematica’s Sqrt function in order to reduce the possibility of incorrect simplification and to insure proper automatic differentiation. For all other cases we can wrap critical function with the general unique signature function SMSFreeze[exp,"Code"->True]. Option "Code"->True is necessary in order to prevent incorrect simplifications throughout the session.

- Example

```
In[210]:= << AceGen;
SMSInitialize["test", "Language" -> "C"];
SMSModule["test", Real[r$$]];
```
Here is a unique signature from interval \([0,1]\) assigned to \(r\).

\[
\text{In[213]}= \quad r \mapsto \text{SMSReal}[r\$$];
\]

\[
r // \text{SMSEvaluate}
\]

\[
\text{Out[213]= } 0.62196351
\]

- This input leads to incorrect simplification of \(\text{Sqrt}[x^2]\) expression.

\[
\text{In[215]}= \quad \phi = \text{Sqrt}[r^2];
\]

\[
\text{SMSSoSqrt}[x]
\]

- This input prevents incorrect simplification.

\[
\text{In[217]}= \quad \phi = \text{SMSSoSqrt}[r^2];
\]

\[
\text{SMSSoSqrt}[\phi]
\]

\[
\text{Out[217]= } \text{SMSSoSqrt}[\phi^2]
\]

- This input leads to incorrect simplification of \(\text{ArcCos}[\text{Cos}[r]]\) expression.

\[
\text{In[219]}= \quad \phi = \text{ArcCos}[\text{Cos}[r]];\]

\[
\text{SMSSoSqrt}[\phi]
\]

- This is incorrect since the simplification of \(\text{ArcCos}[\text{Cos}[r]]\) is not prevented yet.

\[
\text{In[221]}= \quad \phi = \text{SMSFreeze}[\text{ArcCos}[\text{Cos}[r]], \text{"Code"} \rightarrow \text{True}];\]

\[
\text{SMSSoSqrt}[\phi]
\]

- This input represents correct formulation.

\[
\text{In[223]}= \quad \phi = \text{ArcCos}[\text{SMSFreeze}[\text{Cos}[r]], \text{"Code"} \rightarrow \text{True}];\]

\[
\text{SMSSoSqrt}[\phi]
\]

- Expression optimization and automatic differentiation

Differentiation (Automatic Differentiation) is an example where the problems involved in simultaneous simplification are obvious. The table below considers the simple example of the two expressions \(x, y\) and the differentiation of \(y\) with respect to \(x\). \(L(a)\) is an arbitrary large expression and \(v_1\) is an auxiliary variable. From the computational point of view, simplification A is the most efficient and it gives correct results for both values \(x\) and \(y\). However, when used in a further operations, such as differentiation, it obviously leads to wrong results. On the other hand, simplification B has one more assignment and gives correct results also for the differentiation. To achieve maximal efficiency both types of simplification are used in the AceGen system. During the derivation of the formulae type B simplification is performed.
At the end of the session, before the code is generated, the formulae that are stored in global data base are reconsidered to achieve the maximum computational efficiency. At this stage type A simplification is used. All the independent variables (true independent or intermediate auxiliary) have to have an unique signature in order to prevent simplification A (e.g. one can define basic variables with the SMSFreeze function x=SMSFreeze[L(a)]).

### Types of Auxiliary Variables

AceGen system can generate three types of auxiliary variables: real type, integer type, and logical type auxiliary variables. The way of how the auxiliary variables are labeled is crucial for the interaction between the AceGen and Mathematica. New auxiliary variables are labeled consecutively in the same order as they are created, and these labels remain fixed during the Mathematica session. This enables free manipulation with the expressions returned by the AceGen system. With Mathematica user can perform various algebraic transformations on the optimized expressions independently on AceGen. Although auxiliary variables are named consecutively, they are not always stored in the data base in the same order. Indeed, when two expressions contain a common sub-expression, AceGen immediately replaces the sub-expression with a new auxiliary variable which is stored in the data base in front of the considered expressions. The internal representation of the expressions in the data base can be continuously changed and optimized.

Auxiliary variables have standardized form $V[i, j]$, where $i$ is an index of auxiliary variable and $j$ is an instance of the $i$-th auxiliary variable. The new instance of the auxiliary variable is generated whenever specific variable appears on the left hand side of equation. Variables with more that one instance are "multi-valued variables".

The input for Mathematica that generates new auxiliary variable is as follows:

#### lhs operator rhs

The structure 'lhs operator rhs' first evaluates right-hand side expression rhs, creates new auxiliary variable, and assigns the new auxiliary variable to be the value of of the left-hand side symbol lhs. From then on, lhs is replaced by a new auxiliary variable whenever it appears. The rhs expression is then stored into the AceGen database.

In AceGen there are four operators $\models$, $\triangleright$, $\triangleleft$, and $\triangleright$. Operators $\models$ and $\triangleright$ are used for variables that will appear only once on the left-hand side of equation. For variables that will appear more that once on the left-hand side the operators $\triangleleft$ and $\triangleright$ have to be used. These operators are replacement for the simple assignment command in Mathematica (see Mathematica Syntax – AceGen Syntax).

#### Syntax of AceGen assignment operators.

- $v \models exp$ A new auxiliary variable is created if AceGen finds out that the introduction of the new variable is necessary, otherwise $v=exp$. This is the basic form for defining new formulæ. Ordinary Mathematica input can be converted to the AceGen input by replacing the Set operator (=) with the $\models$ operator ($\models$).

- $v \triangleright exp$ A new auxiliary variable is created, regardless on the contents of exp. The primal functionality of this form is to force creation of the new auxiliary variable.

- $v \triangleleft exp$ A new auxiliary variable is created, regardless on the contents of exp. The primal functionality of this form is to create variable which will appear more than once on a left-hand side of equation (multi-valued variables).

- $v \triangleright\triangleright exp$ A new value (exp) is assigned to the previously created auxiliary variable v. At the input v has to be auxiliary variable created as the result of $v \models exp$ command. At the output there is the same variable v, but with the new signature (new instance of v).
If \( x \) is a symbol with the value \( SV[i,j] \), then after the execution of the expression \( x \leftarrow \text{exp} \), \( x \) has a new value \( SV[i,j+1] \). The value \( SV[i,j+1] \) is a new instance of the \( i \)-th auxiliary variable.

Additionally to the basic operators there are functions that perform reduction in a special way. The \texttt{SMSFreeze} function imposes various restrictions in how expression is evaluated, simplified and differentiated. The \texttt{SMSSmartReduce} function does the optimization in a 'smart' way. 'Smart' optimization means that only those parts of the expression that are not important for the implementation of non-local operation are replaced by a new auxiliary variables.

The "signature" of the expression is a high precision real number assigned to the auxiliary variable that represents the expression. The signature is obtained by replacing all auxiliary variables in expression by corresponding signatures and then using the standard \texttt{N} function on the result (\texttt{N[expr, SMSEvaluatePrecision]}). The expression that does not yield a real number as the result of \texttt{N[expr, SMSEvaluatePrecision]} will abort the execution. Thus, any function that yields a real number as the result of numerical evaluation can appear as a part of \texttt{AceGen} expression. However, there is no assurance that the generated code is compiled without errors if there exist no equivalent build in function in compiled language.

Two instances of the same auxiliary variable can appear in the separate branches of 'If' construct. At the code generation phase the active branch of the 'If' construct remains unknown. Consequently, the signature of the variable defined inside the "If" construct should not be used outside the "If" construct. Similar is valid also for "Do" construct, since we do not know how many times the "Do" loop will be actually executed. The scope of auxiliary variable is a part of the code where the signature associated with the particular instance of the auxiliary variable can be uniquely identified. The problem of how to use variables outside the "If"/"Do" constructs is solved by the introduction of fictive instances. Fictive instance is an instance of the existing auxiliary variable that has no effect on a generated source code. It has unique signature so that incorrect simplifications are prevented. Several examples are given in (\texttt{SMSIf}, \texttt{SMSDo}).

An unique signature is also required for all the basic independent variables for differentiation (see Automatic Differentiation) and is also automatically generated for parts of the expressions that when evaluated yield very high or very low signatures (e.g 10^100, 10^-100, see also \texttt{ExpressionOptimization}, \texttt{Signatures of Expressions}). The expression optimization procedure can recognize various relations between expressions, however that is no assurance that relations will be always recognized. Thus users input must not rely on expression optimization as such and it must produce the same result with or without expression optimization (e.g. in "Plain" mode).

**Example: real, integer and logical variables**
- This generates three auxiliary variables: real variable \( x \) with value \( \pi \), integer variable \( i \) with value 1, and logical variable \( l \) with value \texttt{True}.

```plaintext
In[324]:= <<AceGen`
SMSInitialize["test", "Language" -> "Fortran", "Mode" -> "Debug"]
SMSModule["Test"];
x \leftarrow \texttt{SMSReal}[\pi];
i \leftarrow \texttt{SMSSInteger}[1];
l \leftarrow \texttt{SMSLogical}[\texttt{True}];
SMSWrite[];
```

```plaintext
\texttt{time=0 \ variable= 0 = \{x\}  \\
[0] Consistency check - global  \\
[0] Consistency check - expressions  \\
[0] Generate source code :  \\
\textbf{Method} = \texttt{Test} 3 formulae, 13 sub-expressions  \\
Events: 0  \\
[0] Final formatting  \\
Export source code.  \\
[0] File created: \texttt{test.f} Size : 860
```
Intermediate variables are labeled consecutively regardless of the type of variable. This displays how internal variables really look like.

\begin{verbatim}
In[331]:= {x, i, l} // ToString
Out[331]= {V[1, 1], V[2, 1], V[3, 1]}
\end{verbatim}

This displays the generated FORTRAN code. AceGen translates internal representation of auxiliary variables accordingly to the type of variable as follows:

\begin{align*}
x & : V[1, 1] \\
i & : V[2, 1] \\
l & : V[3, 1]
\end{align*}

\begin{verbatim}
In[332]:= FilePrint["test.f"]

SUBROUTINE Test
IMPLICIT NONE
include 'sms.h'
INTEGER i2
LOGICAL b3
DOUBLE PRECISION v(5001)

1 = x
v(1) = 0.3141592653589793d1
2 = i
i2 = int(1)
3 = l
b3 = .true.
END
\end{verbatim}

Example: multi-valued variables

This generates two instances of the same variable \(x\). The first instance has value \(\pi\) and the second instance has value \(\pi^2\).

\begin{verbatim}
In[333]:= << AceGen;
SMSInitialize["test", "Language" -> "Fortran", "Mode" -> "Debug"];
SMSModule["Test"];
x = SMSReal[\pi];
x = \pi^2;
SMSWrite[];
\end{verbatim}
time=0  variable= 0 ≡ {x}

0] Consistency check - global
[0] Consistency check - expressions
[0] Generate source code :

Method: Test 2 formulae, 7 sub-expressions

Events: 0

[0] Final formatting

Export source code.

[0] File created :: test.f Size : 812

This displays how the second instance of x looks like inside the expressions.

\(x \equiv \{x\}\)

This displays the generated FORTRAN code. AceGen translates two instances of the first auxiliary variable into the same FORTRAN variable.

\(x := \{V[1, 1] \Rightarrow v(1)\}\)

\(x := \{V[1, 2] \Rightarrow v(1)\}\)

Auxiliary Variables Functions

**SMSR**

\[\text{SMSR}[symbol,exp]\]

create a new auxiliary variable if introduction of a new variable is necessary, otherwise \(symbol=exp\)

\(symbol \equiv exp\)

infix form of the SMSR function is equivalent to the standard form \(\text{SMSR}[symbol,exp]\)

The SMSR function first evaluates \(exp\). If the result of the evaluation is an elementary expression, then no auxiliary variables are created and the \(exp\) is assigned to be the value of \(symbol\). If the result is not elementary expression, then AceGen creates a new
auxiliary variable, and assigns the new auxiliary variable to be the value of symbol. From then on, symbol is replaced by the new auxiliary variable whenever it appears. Evaluated expression exp is then stored into the AceGen data base.

Precedence level of ⊨ operator is specified in precedence table below. It has higher precedence than arithmetical operators like +, *, /, but lower than postfix operators like // and /., /,... In these cases the parentheses or the standard form of functions have to be used.

For example, x ⊨ a+b/.a->3 statement will cause an error. There are several alternative ways how to enter this expression correctly. Some of them are:

x = (a+b/.a->3),

x = ReplaceAll[a+b,a->3],

SMSR[x,a+b/.a->3],

x = SMSSimplify[a+b/.a->3].

---

<table>
<thead>
<tr>
<th>Extensions of symbol names</th>
<th>x_, #2, ::, etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function application variants</td>
<td>e[e], e@@e, etc.</td>
</tr>
<tr>
<td>Power-related operators</td>
<td>√e, e^e, etc.</td>
</tr>
<tr>
<td>Multiplication-related operators</td>
<td>e e, e / e, etc.</td>
</tr>
<tr>
<td>Addition-related operators</td>
<td>e ⊕ e, e + e, etc.</td>
</tr>
<tr>
<td>Relational operators</td>
<td>e == e, e-&gt; e, etc.</td>
</tr>
<tr>
<td>Arrow and vector operators</td>
<td>e -&gt; e, e &lt;-&gt; e, etc.</td>
</tr>
<tr>
<td>Logic operators</td>
<td>∀e, e &amp;&amp; e, e ⊢ e, etc.</td>
</tr>
<tr>
<td><strong>AceGen operators</strong></td>
<td>⊨, ⊢, ⊤, ⊣</td>
</tr>
<tr>
<td>Postfix and rule operators</td>
<td>e//e, e/, etc.</td>
</tr>
<tr>
<td>Pure function operator</td>
<td>e&amp;</td>
</tr>
<tr>
<td>Assignment operators</td>
<td>e = e, etc.</td>
</tr>
<tr>
<td>Compound expression</td>
<td>e; e</td>
</tr>
</tbody>
</table>

Precedence of AceGen operators.

Numbers are elementary expressions thus a new auxiliary is created only for expression Sin[5].

In[37]:= SMSInitialize["test", "Language" -> "Fortran"];
SNSModule["sub"];
x = 1
y = Sin[5]
1
j

SMSV

SMSV[symbol, exp]
create a new auxiliary variable regardless of the contents of exp

\[\text{symbol} \leftarrow \text{exp}\]

an infix form of the SMSR function is equivalent to the standard form SMSV[\text{symbol},\text{exp}]

The SMSV function first evaluates \text{exp}, then AceGen creates a new auxiliary variable, and assigns the new auxiliary variable to be the value of symbol. From then on, symbol is replaced by the new auxiliary variable whenever it appears. Evaluated expression \text{exp} is then stored into the AceGen database.

Precedence level of \leftarrow operator is specified in \textit{Mathematica} precedence table and described in SMSR.

- The new auxiliary variables are created for all expressions.

\begin{verbatim}
In[101]:= SMSInitialize["test", "Language" -> "Fortran"];
    SMSModule["sub"];
    x \leftarrow 1
    y \leftarrow Sin[5]
\end{verbatim}

\begin{verbatim}

\textbf{SMSM}

\textbf{SMSM}[\text{symbol},\text{exp}]

create a new multi-valued auxiliary variable

\[\text{symbol} \leftarrow \text{exp}\]

an infix form of the SMSM function is equivalent to the standard form SMSM[\text{symbol},\text{exp}]

The primal functionality of this form is to create a variable which will appear more than once on the left-hand side of equation (multi-valued variables). The SMSM function first evaluates \text{exp}, creates a new auxiliary variable, and assigns the new auxiliary variable to be the value of symbol. From then on, symbol is replaced by a new auxiliary variable whenever it appears. Evaluated expression \text{exp} is then stored into the AceGen database. The new auxiliary variable will not be created if \text{exp} matches one of the functions which create by default a new auxiliary variable. Those functions are SMSReal, SMSInteger, SMSLogical, SMSFreeze and SMSFictive. The result of those functions is assigned directly to the symbol.

Precedence level of \leftarrow operator is described in SMSR.

\begin{verbatim}

\textbf{SMSS}

\textbf{SMSS}[\text{symbol},\text{exp}]

a new instance of the previously created multi-valued auxiliary variable is created

\[\text{symbol} \leftarrow \text{exp}\]

this is an infix form of the SMSS function and is equivalent to the standard form SMSS[\text{symbol},\text{exp}]

At the input the value of the \text{symbol} has to be a valid multi-valued auxiliary variable (created as a result of functions like SMSS, SMSM, SMSEndIf, SMSEndDo, etc.). At the output there is a new instance of the \text{i}-th auxiliary variable with the unique signature. SMSS function can be used in connection with the same auxiliary variable as many times as we wish.

Precedence level of \leftarrow operator is described in SMSR.

- Successive use of the \leftarrow and \rightarrow operators will produce several instances of the same variable \text{x}.

\end{verbatim}

In[103]:= SMSInitialize["test", "Language" -> "Fortran", "Mode" -> "Prototype"];

SMSModule["sub", Real[x$$]];

x = 1
x + x + 2
x + 5 x

In[110]:= SMSExport[4, x$$];

SMSWrite[];

Method : sub 4 formulae, 16 sub-expressions

File created : test.f Size : 808

In[112]:= FilePrint["test.f"]

SUBROUTINE sub
IMPLICIT NONE
include 'sms.h'
DOUBLE PRECISION v(5001),x
v(1)=1d0
v(1)=2d0+v(1)
v(1)=5d0+v(1)
x=v(1)
END

SMSSimplify

SMSSimplify[exp]
create a new auxiliary variable if the introduction of new variable is necessary, otherwise the original exp is returned

The SMSSimplify function first evaluates exp. If the result of the evaluation is an elementary expression, then no auxiliary variables are created and the original exp is the result. If the result is not an elementary expression, then AceGen creates and returns a new auxiliary variable. SMSSimplify function can appear also as a part of an arbitrary expression.

SMSVariables

SMSVariables[exp]
gives a list of all auxiliary variables in expression in the order of appearance and with duplicate elements removed
Example

\$\langle 187 \rangle := << \text{AceGen} >>$

\text{SMSInitialize} ["test"];
\text{SMSModule} ["Test", Real [a $$]]; 
\[ a \rightarrow \text{SMSReal} [a $$] ]; 
\[ M \rightarrow \{ a, a^2 \} \] 
\begin{equation}
(\{ 1, 1 \}, \{ 1, 0 \})
\end{equation}

\$\langle 192 \rangle := \text{SMSVariables} [M]$

\{ 1, 1 \}

\text{SMSFreeze, SMSUnFreeze}

\text{SMSFreeze}[\text{symbol}, \{ \text{exp}_1, \text{exp}_2, \{ \text{exp}_3 \} \ldots \} \ldots \}, \text{allOptions} ]

create data objects that represent elements of arbitrarily structured list of expressions \{ \text{exp}_1, \text{exp}_2, \{ \text{exp}_3 \} \ldots \} \ldots \} accordingly to given options \text{allOptions}. New auxiliary variables with the values \{ \text{exp}_1, \text{exp}_2, \{ \text{exp}_3 \} \ldots \} \ldots \} and random signature are then generated and the resulting arbitrarily structured list is then assigned to symbol \text{symbol}. The process can be additionally altered by special options listed below that are valid only for input expressions that are arbitrarily structured lists of expressions.

\text{SMSFreeze}[\text{symbol}, \{ \text{exp}_1, \text{exp}_2, \ldots \}, \text{generalOptions} ]

\equiv \text{symbol} \rightarrow \text{SMSFreeze}[\{ \text{exp}_1, \text{exp}_2, \ldots \}, \text{generalOptions} ]

\text{(note that this is not true when the special options for lists are used)}

\text{SMSFreeze}[\text{exp}]

create data object (\text{SMSFreezeF}) that represents expression \text{exp}, however its numerical evaluation yields an unique signature obtained by the random perturbation of the original signature of \text{exp}

\text{(original signature is restored for the second code optimization cycle at the code generation phase!!)}

\text{SMSFreeze}[\text{exp}, \text{generalOptions} ]

create data object (\text{SMSFreezeF}) that represents expression \text{exp} accordingly to given general options \text{generalOptions}

\text{SMSFreeze}[\{ \text{exp}_1, \text{exp}_2, \ldots \}, \text{generalOptions} ]

create list of data objects (\text{SMSFreezeF}) that represent expressions \{ \text{exp}_1, \text{exp}_2, \ldots \} accordingly to given general options \text{generalOptions} (note that special options that apply on lists of expressions cannot be used in this form)

\text{SMSUnFreeze}[\text{exp}]

first search \text{exp} argument for all auxiliary variables that have been frozen by the \text{SMSFreeze} command and after then replace any appearance of those variables in expression \text{exp} by their definitions

\noindent \text{Imposing restrictions on an optimization procedure.}

\noindent \textbf{option} \quad \text{default} \quad \text{description}

| "Dependency" | False | see below |
| "Contents" | False | whether to prevent the search for common sub expressions inside the expression \text{exp} |
| "Code" | False | whether to keep all options valid also at the code generation phase |
| "Differentiation" | False | whether to use \text{SMSFreeze} also for the derivatives of given expression \text{exp} |
| "Verbatim" | False | \text{SMSFreeze}[\text{exp}, "Verbatim"->True] \equiv \text{SMSFreeze}[\text{exp}, "Contents"->True, "Code"->True, "Differentiation"->True] |
| "Subordinate"->_List | \{ \} | list of auxiliary variables that represent control structures (e.g. \text{SMSCall}, \text{SMSVerbatim}, \text{SMSEXport}) that have to be executed before the evaluation of the current expression |
Options of the SMSFreeze function.

<table>
<thead>
<tr>
<th>options for structured lists</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Ignore&quot;-&gt;crit</td>
<td>(False&amp;)</td>
<td>the SMSFreeze function is applied only on parts of the list for which crit[e] yields False (NumberQ[exp] yields True)</td>
</tr>
<tr>
<td>&quot;Symmetric&quot;</td>
<td>False</td>
<td>if an input is a matrix (symmetric or not) then the output is a symmetric matrix</td>
</tr>
<tr>
<td>&quot;IgnoreNumbers&quot;</td>
<td>False</td>
<td>True = &quot;Ignore&quot;-&gt;NumberQ whether to apply SMSFreeze function only on parts of the list that are not numbers</td>
</tr>
<tr>
<td>&quot;KeepStructure&quot;</td>
<td>False</td>
<td>new auxiliary variables with random signatures are generated for all parts of the input expression that have random signatures in a way that the number of newly introduced auxiliary variables is at minimum (note that the result of this option might be dependent on Mathematica or AceGen version)</td>
</tr>
<tr>
<td>&quot;Variables&quot;</td>
<td>False</td>
<td>apply SMSFreeze function on auxiliary variables that explicitly appear as a part of expression instead of expression as a whole</td>
</tr>
</tbody>
</table>

Special options valid for input expressions that are arbitrarily structured lists of expressions.

```
SMSFreeze[exp, 
"Dependency"->value]
```

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>assume that SMSFreezeF data object is independent variable (all partial derivatives of exp are 0)</td>
</tr>
<tr>
<td>False</td>
<td>assume that SMSFreezeF data object depends on the same auxiliary variables as original expression exp (partial derivatives of SMSFreezeF are the same as partial derivatives of exp)</td>
</tr>
<tr>
<td>{{\partial exp \partial p_1}, {\partial exp \partial p_2}, ...}</td>
<td>assume that SMSFreezeF data object depends on given auxiliary variables (p_1, p_2, \ldots) and define the partial derivatives of SMSFreezeF data object with respect to given auxiliary variables (p_1, p_2, \ldots)</td>
</tr>
</tbody>
</table>

Values for "Dependency" option when the input is a single expression.
SMSFreeze[exp₁, exp₂, ...],
"Dependency" -> value

<table>
<thead>
<tr>
<th>True</th>
<th>assume that all expressions are independent variables (all partial derivatives of exp₁ are 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>assume that after SMSFreeze expressions depend on the same auxiliary variables as original expressions</td>
</tr>
</tbody>
</table>

\{p₁ \{\frac{\partial exp₁}{\partial p₁}, \frac{\partial exp₁}{\partial p₂}, \ldots\}\}\,
\{p₂ \{\frac{\partial exp₁}{\partial p₁}, \frac{\partial exp₁}{\partial p₂}, \ldots\}\},
\{\frac{\partial exp₁}{\partial p₁}, \frac{\partial exp₁}{\partial p₂}, \ldots\}\,
\{\frac{\partial exp₂}{\partial p₁}, \frac{\partial exp₂}{\partial p₂}, \ldots\}\,
\{\frac{\partial exp₂}{\partial p₁}, \frac{\partial exp₂}{\partial p₂}, \ldots\}\,

Values for "Dependency" option when the input is a vector of expressions.

The SMSFreeze function creates SMSFreezeF data object that represents input expression. The numerical value of resulting SMSFreezeF data object (signature) is calculated by the random perturbation of the numerical value of input expression (unique signature). The SMSFreeze function can impose various additional restrictions on how expressions are evaluated, simplified and differentiated (see options).

An unique signature is assigned to exp, thus optimization of exp as a whole is prevented, however AceGen can still simplify some parts of the expression. The "Contents"->True option prevents the search for common sub expressions inside the expression.

Original expression is recovered at the end of the session, when the program code is generated and all restrictions are removed. With the "Code"->True option the restrictions remain valid also in the code generation phase. An exception is the option "Dependency" which is always removed and true dependencies are restored before the code generation phase. Similarly the effects of the SMSFreeze function are not inherited for the result of the differentiation. With the "Differentiation"->True option all restrictions remain valid for the result of the differentiation as well.

With SMSFreeze[exp, "Dependency" -> \{\{p₁ \frac{\partial exp}{\partial p₁}\}, \{p₂ \frac{\partial exp}{\partial p₂}\}, \ldots, \{p_n \frac{\partial exp}{\partial p_n}\}\}] the true dependencies of exp are ignored and it is assumed that exp depends on auxiliary variables p₁, ..., pₙ. Partial derivatives of exp with respect to auxiliary variables p₁, ..., pₙ are taken to be \frac{\partial exp}{\partial p₁}, ..., \frac{\partial exp}{\partial pₙ}. (see also SMSDefineDerivative where the definition of the total derivatives of the variables is described).

SMSFreeze[exp,"Verbatim"] stops all automatic simplification procedures.

SMSFreeze function is automatically threaded over the lists that appear as a part of exp.

Basic Examples

In[53]:= << AceGen`;
SMSInitialize["test"];
SMSModule["sub", Real[x$$]];
x = SMSReal[x$$];
SMSFreeze creates data object (SMSFreezeF) that contains original expression $\sin(x)$. New auxiliary variable are not yet introduced!

```
In[37]:= SMSFreeze[\sin(x)]]
```

- However, its numerical evaluation yields an unique signature obtained by the random perturbation of the signature of original expression.

```
In[38]:= (\sin(x), SMSFreeze[\sin(x)]) // SMSEvaluate
```

```
{0.67104233, 0.66222981}
```

- New auxiliary variable that represents original expression can be introduced by

```
In[39]:= xf \triangleq SMSFreeze[\sin(x)]; xf
```

- or by

```
In[40]:= SMSFreeze[xf, \sin(x)]; xf
```

Options

Options of the SMSFreeze functions are applied on matrix $M = \begin{pmatrix} x & 2x & \cos(x) \\ 2x & 2 & 2x \\ -\cos(x) & 0 & \frac{1}{2} \end{pmatrix}$.

```
In[41]:= <<AceGen';
SMSInitialize["test"];
SMSModule["sub", Real[x$$]]; x \times SMSReal[x$$];
M = \begin{pmatrix} x & 2x & \cos(x) \\ 2x & 2 & 2x \\ -\cos(x) & 0 & \frac{1}{2} \end{pmatrix};
```

- The random signatures of elements of the original matrix are obtained by perturbation of the random signatures of the elements of original matrix.

```
In[3499]:= M // SMSEvaluate // MatrixForm
```

```
0.68150910 1.3630182 0.77662292
1.3630182 2.0000000 1.3630182
-0.77662292 0 0.50000000
```

- The SMSFreeze function applied on the whole matrix will create a new auxiliary variable for each element of the matrix regardless on the structure of the matrix.

```
In[3500]:= MF \triangleq SMSFreeze[M]; MF // MatrixForm
```

```
MF
```

- The random signatures of elements of the original matrix are obtained by perturbation of the random signatures of the elements of original matrix.
Here the new auxiliary variables with random signatures are generated only for the elements of the matrix for which \texttt{NumberQ[x]} yields true. 6 new auxiliary variables are generated.

Here the new auxiliary variables with random signatures are generated for all parts of the matrix that have random signatures (numbers do not have random signatures!) in a way that the number of new auxiliary variables is minimum. Only 3 new auxiliary variables are generated in this case. The properties of the matrix such as symmetry, antisymmetry etc. are preserved when detected. Note that the symmetry of the matrix is detected accordingly to the signature of the elements of the matrix, thus the detection of the symmetry is not absolutely guaranteed. When the symmetry or any other property of the matrix is essential for the correctness of derivation the property has to be enforced explicitly as presented below.

Here all the auxiliary variables in original expression are replaced by new auxiliary variables with random signatures.
In[3515]:= SMSFreeze[Mf, M, "Variables" -> True];
Mf // MatrixForm
Mf // SMSEvaluate // MatrixForm

\[
\begin{bmatrix}
2 & 2 \cos \left[ \frac{1}{2} \right] \\
2 & 2 & 2 \\
-\cos \left[ \frac{1}{2} \right] & 0 & \frac{1}{2}
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.65690486 & 1.3138097 & 0.79188613 \\
1.3138097 & 2.0000000 & 1.3138097 \\
-0.79188613 & 0 & 0.5000000
\end{bmatrix}
\]

Option dependency

In[55]:= << AceGen`
SMSInitialize["test"];
SMSSubModule["sub", Real[p1$$, p2$$]];
{p1, p2} ⊢ SMSReal[{{p1$$, p2$$}}];
{e1, e2} ⊢ {p1 p2, \sin[p1] \cos[p2]};

◼

Here all partial derivatives of expression e1 are set to 0 except:
\[
\frac{\partial e_1}{\partial p_1} = 5.
\]
The derivatives \[
\left\{ \frac{\partial \log(e_1)}{\partial p_1}, \frac{\partial \log(e_1)}{\partial p_2} \right\}
\]
are then evaluated assuming explicitly defined partial derivatives.

In[60]:= f1 ⊢ SMSFreeze[e1, "Dependency" -> {p1, 5}];
SMSD[\log[f1], {p1, p2}]

\[
\begin{bmatrix}
5 \\
0
\end{bmatrix}
\]

◼

Here all partial derivatives of expression e1 are set to 0 except:
\[
\frac{\partial e_1}{\partial p_1} = 5, \frac{\partial e_1}{\partial p_2} = 10.
\]
The derivatives \[
\left\{ \frac{\partial \log(e_1)}{\partial p_1}, \frac{\partial \log(e_1)}{\partial p_2} \right\}
\]
are then evaluated assuming explicitly defined partial derivatives.

In[62]:= f1 ⊢ SMSFreeze[e1, "Dependency" -> {{p1, 5}, {p2, 10}}];
SMSD[\log[f1], {p1, p2}]

\[
\begin{bmatrix}
5 & 10 \\
0 & 0
\end{bmatrix}
\]

◼

Here all partial derivatives of expressions e1 and e2 are set to 0 except:
\[
\frac{\partial e_1}{\partial p_1} = 5, \frac{\partial e_1}{\partial p_2} = 10,
\frac{\partial e_2}{\partial p_1} = 15, \frac{\partial e_2}{\partial p_2} = 20.
\]
The derivatives \[
\left\{ \frac{\partial \log(e_1)}{\partial p_1}, \frac{\partial \log(e_1)}{\partial p_2}, \frac{\partial \log(e_2)}{\partial p_1}, \frac{\partial \log(e_2)}{\partial p_2} \right\}
\]
are then evaluated assuming explicitly defined gradients of expressions.

In[70]:= \{f1, f2\} ⊢ SMSFreeze[\{e1, e2\}, "Dependency" ->

\{
\{\{p1, 5\}, \{p2, 10\}\} (*\forall e_1*)
, \{\{p1, 15\}, \{p2, 20\}\} (*\forall e_2*)
\}];

SMSD[\{\log[f1], \log[f2]\}, {p1, p2}]

\[
\begin{bmatrix}
5 & 10 \\
0 & 0
\end{bmatrix}, \begin{bmatrix}
15 & 20 \\
0 & 0
\end{bmatrix}
\]
The above result can be also obtained by defining a set of unknowns \( p_i \) and the Jacobian matrix \( J_{ij} = \frac{\partial e_i}{\partial p_j} \).

```plaintext
In[72]:= \{f1, f2\} \[\mapsto\] SMSFreeze[\{e1, e2\}, "Dependency" ->
\{(p1, p2), (*pi*)
\{(5, 10), (15, 20)\} (*Jacobian \( \frac{\partial e_i}{\partial p_j} \)*)
];
SMSD[\{Log[f1], Log[f2]\}, \{p1, p2\}]
\{"5, 10\}, \{"15, 20\}\}
```

Troubleshooting

The use of `SMSFreeze[exp,options]` form of the `SMSFreeze` function with options `Ignore`, `IgnoreNumbers`, `Symmetric`, `Variables` and `KeepStructure` may lead to unexpected results! Please consider to use `SMSFreeze[symbol,exp,options]` form instead.

```plaintext
In[24]:= <<AceGen``;
SMSInitialize["test"];
SMSModule[\"sub\", Real \[x\$$]];
x \[\mapsto\] SMSReal \[x\$$];
```

Here the option "IgnoreNumbers" is not accounted for in the final result.

```plaintext
In[3555]:= vf \[\mapsto\] SMSFreeze[\{Sin[x], 0\}, "IgnoreNumbers" -> True];
vf
```

The use of `SMSFreeze[exp,options]` form of the `SMSFreeze` functions with options `Ignore`, `IgnoreNumbers`, `Symmetric`, `Variables` and `KeepStructure` may lead to unexpected results! Please consider to use `SMSFreeze[symbol,exp,options]` form instead. See also: `SMSFreeze`.

```plaintext
In[3559]:= SMSFreeze[vf, \{Sin[x], 0\}, "IgnoreNumbers" -> True];
vf
```

Correct result is obtained if the `SMSFreeze[symbol,exp,options]` form is used.

**SMSFictive**

```plaintext
SMSFictive["Type"->fictive_type]
create fictive variable of the type `fictive_type` (Real, Integer or Logical)
SMSFictive[] \[\[\equiv\]\] SMSFictive["Type"->Real]
```

Definition of a fictive variable.

A fictive variable is used as a temporary variable to perform various algebraic operations symbolically on `AceGen` generated expression (e.g. integration, finding limits, solving algebraic equations symbolically, ...). For example, the integration variable \( x \) in a symbolically evaluated definite integral \( \int_a^b f(x) \, dx \) can be introduced as a fictive variable since it will not appear as a part of the result of integration.

The fictive variable is an auxiliary variable but it does not have assigned value, thus it must not appear anywhere in a final source code. The fictive variable that appears as a part of the final code is replaced by random value and a warning message appears.

See also: Auxiliary Variables, Local and Non-local Operations.
Example

Here the pure fictive auxiliary variable is used for \( x \) in order to evaluate expression
\[ f(n) = \sum_{n=1}^{m} \frac{\partial g(x)}{\partial x} \bigg|_{x=0}, \]
where \( g(x) \) is arbitrary expression (can be large expression involving If and Do structures). Not that 0 cannot be assigned to \( x \) before the differentiation.

```
In[175]:= <<AceGen`; SMSInitialize["test", "Language" -> "C"]
SMSModule["sub", Real, f$$, a$$, b$$, m$$];
SMSDo[n, 1, SMSInteger, m$$];
x ⊢ SMSFictive[];
g ⊨ Sin[x^n] + Cos[x^n];
f ⊣ f + SMSReplaceAll[SMSD[g, x], x → 0];
SMSEndDo[f];
SMSExport[f, f$$];
```

```
In[185]:=
File: test.c Size: 835
Methods No.Formulae No.Leafs
sub 3 15
```

```
In[186]:=
FilePrint["test.c"]
```

Local and Non-local Operations

General Description

Many high level operations in computer algebra can only be implemented when the whole expression to which they are applied is given in an explicit form. Integration and factorization are examples for such ‘non-local’ operations. On the other hand, some operations such as differentiation can be performed ‘locally’ without considering the entire expression. In general, we can divide algebraic computations into two groups:

Non-local operations have the following characteristics:

- symbolic integration, factorization, nonlinear equations,
the entire expression has to be considered to get a solution,
all the relevant variables have to be explicitly "visible".

Local operations have the following characteristics:
- differentiation, evaluation, linear system of equations,
- operation can be performed on parts of the expression,
- relevant variables can be part of already optimized code.

Symbolic integration is rarely used in numerical analysis. It is possible only in limited cases. Additionally, the integration is an operation of 'non-local' type. Nevertheless, we can still use all the built-in capabilities of Mathematica and then optimize the results.

For 'non-local' operations, such as integration, the AceGen system provides a set of functions which perform optimization in a 'smart' way. 'Smart' optimization means that only those parts of the expression that are not important for the implementation of the 'non-local' operation are replaced by new auxiliary variables. Let us consider expression \( f \) which depends on variables \( x, y, \) and \( z \).

```mathematica
In[281]:= << AceGen;
SMSInitialize["test", "Language" -> "Mathematica"];
SMSModule["Test", Real, x$$, y$$, z$$];
{x, y, z} \[\mapsto\] SMSReal[x$$]; SMSReal[y$$]; SMSReal[z$$];

f = x^2 + 2 xy + y^2 + 2 y z + z^2
```

```mathematica
Out[281]= x^2 + 4 x y + y^2 + 2 y z + z^2
```

Since integration of \( f \) with respect to \( x \) is to be performed, we perform 'smart' optimization of \( f \) by keeping the integration variable \( x \) unchanged which leads to the optimized expression \( f_x \). Additionally Normal converts expr to a normal expression, from a variety of AceGen special forms.

```mathematica
In[286]:= fx = SMSSmartReduce[f, x, Collect[#, x] &] // Normal
```

```mathematica
Out[286]= x^2 + 4 x y + y^2 + 2 y z + z^2
```

The following vector of auxiliary variables is created.

```mathematica
In[287]:= SMSShowVector[]
```

```mathematica
Out[287]= $S[Method, Null, 1]
V[1,1] = x = x$$
V[2,1] = y = y$$
V[3,1] = z = z$$
V[4,1] = $S1 = (2 y)^2 + 2 (2 y) (3 z) + (3 z)^2
V[5,1] = $S2 = 4 (3 y)
$S[End, Null, {}]
```

```mathematica
In[288]:= fint = \[\int\] fx \, dx
```

```mathematica
Out[288]= \[\frac{x^2}{3} + \frac{y^2}{2} + \frac{z^2}{2}
```

After the integration, the resulting expression \( fint \) is used to obtain another expression \( fr \). \( fr \) is identical to \( fint \), however with an exposed variable \( y \). New format is obtained by 'smart' restoring the expression \( fint \) with respect to variable \( y \).

```mathematica
In[289]:= fr = SMSSmartRestore[fint, y, Collect[#, y] &] // Normal
```

```mathematica
Out[289]= x^2 + \[\frac{y^2}{2} + \frac{z^2}{2}
```
At the end of the Mathematica session, the global vector of formulae contains the following auxiliary variables:

```
In[290]:= SMSShowVector[];

    $S[Method, Null, 1]
V[1,1] ≡ x ≡ x$$
V[2,1] ≡ y ≡ y$$
V[3,1] ≡ z ≡ z$$
V[6,1] ≡ 6¥ ≡ (3z)²
V[4,1] ≡ §1 ≡ (3y)² + 2 (3y) (3z) + 6¥
V[5,1] ≡ §2 ≡ 4 (3y)
V[7,1] ≡ §3 ≡ (3x)³ + (3x) (6¥)
V[8,1] ≡ §4 ≡ 2 (3x)² + 2 (3x) (3z)
$S[End, Null, {}]
```

**SMSSmartReduce**

SMSSmartReduce[exp,v1|v2|...]  
replace those parts of the expression exp that do not depend on any of the auxiliary variables v1|v2|... by a new auxiliary variable

SMSSmartReduce[exp,v1|v2|...,func]  
apply pure function func to the sub-expressions before they are replaced by a new auxiliary variable

The default value for func is identity operator #&. Recommended value is Collect[#1|v2|...]&. The function func should perform only correctness preserving transformations, so that the value of expression exp remains the same.

**SMSSmartRestore**

SMSSmartRestore[exp,v1|v2|...]  
replace those parts of expression exp that depend on any of the auxiliary variables v1|v2|... by their definitions and simplify the result

SMSSmartRestore[exp,v1|v2|...,func]  
apply pure function func to the sub-expressions that do not depend on v1|v2|... before they are replaced by a new auxiliary variable

SMSSmartRestore[exp,v1|v2|...,evaluation_rules,func]  
restore expression exp and apply list of rules {evaluation_rules} to all sub-expressions that depend on any of auxiliary variables v1,v2,...

At the output, all variables v1|v2|... become fully visible. The result can be used to perform non-local operations. The default values for func is identity operator #&. Recommended value is Collect[#1|v2|...]&. The function func should perform only correctness preserving transformations, so that the values of expression remain the same.

The list of rules evaluation_rules can alter the value of exp. It can be used for a symbolic evaluation of expressions (see SMSReplaceAll).

The difference between the SMSSmartReduce function and the SMSSmartRestore function is that SMSSmartRestore function searches the entire database of formulae for the expressions which depend on the given list of auxiliary variables v1, v2, ..., while SMSSmartReduce looks only at parts of the current expression.

The result of the SMSSmartRestore function is a single symbolic expression. If any of auxiliary variable involved has several definitions (multi-valued auxiliary variables), then the result can not be uniquely defined and the SMSSmartRestore function can not be used.
SMSRestore

**SMSRestore[exp,v1,v2,...]**

replace those parts of expression `exp` that depend on any of the auxiliary variables `v1,v2,...` by their definitions

**SMSRestore[exp,v1,v2,...,evaluation_rules]**

restore expression `exp` and apply list of rules `evaluation_rules` to all sub-expressions that depend on any of auxiliary variables `v1,v2,...`

**SMSRestore[exp]**

replace all visible auxiliary variables in `exp` by their definition

**SMSRestore[exp,"Global"]**

repeatedly replace all auxiliary variables until only basic input variables remain (objects such as `SMSExternalF`, `SMSFreezeF` and `SMSFictiveF` are left intact)

At the output, all variables `v1,v2,...` become fully visible, the same as in the case of SMSSmartRestore function. However, while SMSSmartRestore simplifies the result by introducing new auxiliary variables, SMSRestore returns original expression.

If any of auxiliary variable involved has several definitions (multi-valued auxiliary variables), then the result can not be uniquely defined and the SMSRestore function can not be used.

**SMSReplaceAll**

**SMSReplaceAll[exp,v1->new1,v2->new2,...]**

replace any appearance of auxiliary variable `v_i` in expression `exp` by corresponding expression `new_i`

Symbolic evaluation means evaluation of expressions with the symbolic or numerical value for a particular parameter. The evaluation can be efficiently performed with the `AceGen` function `SMSReplaceAll`.

At the output the `SMSReplaceAll` function gives `exp | v1=new1, v2=new2, ...`. The `SMSReplaceAll` function searches entire database for the auxiliary variables that influence evaluation of the given expression `exp` and at the same time depend on any of the auxiliary variables `v_i`. The current program structure is then enhanced by the new auxiliary variables. Auxiliary variables involved can have several definitions (multi-valued auxiliary variables).

It is **users responsibility** that the new expressions are correct and consistent with the existing program structure. Each time the `AceGen` commands are used, the system tries to modified the entire subroutine in order to obtain optimal solution. As the result of this procedures some variables can be redefined or even deleted. Several situations when the use of `SMSReplaceAll` can lead to incorrect results are presented on examples.

However, even when all seems correctly the `SMSReplaceAll` can abort execution because it failed to make proper program structure. Please reconsider to perform replacements by evaluating expressions with the new values directly when `SMSReplaceAll` fails.

**Example 1: Taylor series expansion**

A typical example is a Taylor series expansion,

\[ F(x) = F(x) \big|_{x=x_0} + \frac{\partial F(x)}{\partial x} \big|_{x=x_0} (x-x_0), \]

where the derivatives of `F` have to be evaluated at the specific point with respect to variable `x`. Since the optimized derivatives depend on `x` implicitly, simple replacement rules that are built-in `Mathematica` can not be applied.

- This generates `FORTRAN` code that returns coefficients `F(x) |_{x=x_0}` and `\frac{\partial F(x)}{\partial x} |_{x=x_0}` of the Taylor expansion of the function

\[ 3x^2 + \sin[x^2] - \log[x^2 - 1]. \]
Example 2: the variable that should be replaced does not exist

- The ⊨ command creates variables accordingly to the set of rules. Here the expression \( y = -x \) did not create a new variable \( y \) resulting in wrong replacement.

```mathematica
<< AceGen`;
SMSInitialize["test", "Language" -> "Fortran"];
SMSModule["Test", Real[x0$$, f0$$, fx0$$]]; 
x0 ⊨ SMSReal[x0$$];
x ⊨ SMSFictive[];
f ⊨ 3x^2 + \sin[x^2] - \log[x^2 - 1];
f0 ⊨ SMSReplaceAll[f, x -> x0];
fx ⊨ SMSD[f, x];
fx0 ⊨ SMSReplaceAll[fx, x -> x0];
SMSExport[{f0, fx0}];
```

```fortran
SUBROUTINE Test(v,x0,f0,fx0)
IMPLICIT NONE
include 'sms.h'
DOUBLE PRECISION v(5001),x0,f0,fx0
v(11)=x0**2
v(12)=(-1d0)+v(11)
f0=3d0*v(11)-dlog(v(12))+dsin(v(11))
fx0=2d0*x0*(3d0-1d0/v(12)+dcos(v(11)))
END
```

The \( \triangleright \) command always creates new variable and leads to the correct results.
Example 3: repeated use of SMSReplaceAll

- Repeated use of SMSReplaceAll can produce large intermediate codes and should be avoided if possible.

```mathematica
In[152]:= << AceGen`; SMSInitialize["test"]; SMSModule["sub", Real[x$$]]; x = SMSReal[x$$]; y = Sin[x]; z = Cos[x]; y0 = SMSReplaceAll[y, x -> 0]; z0 = SMSReplaceAll[z, x -> 0]; {y0, z0} = SMSReplaceAll[{y, z}, x -> 0];
```

- Better formulation.
Program Flow Control

Contents

- General Description
- Conditionals
  - SMSIf
  - SMSElse
  - SMSEndIf
  - SMSSwitch
  - SMSWhich
- Loops
  - SMSDo
  - SMSEndDo
  - SMSReturn
  - SMSBreak
  - SMSContinue

General Description

AceGen can automatically generate conditionals (SMSIf, SMSSwitch and SMSWhich constructs) and loops (SMSDo construct). The program structure specified by the conditionals and loops is created simultaneously during the AceGen session and it will appear as a part of automatically generated code in a specified language. All other conditional and loop structures have to be manually replaced by the equivalent forms consisting only of If and Do statements. It is important to notice that only the replaced conditionals and loops produce corresponding conditionals and loops in the generated code and are evaluated when the generated program is executed. The conditionals and loops that are left unchanged are evaluated directly in Mathematica during the AceGen session. Additionally, we can include parts of the final source code verbatim (SMSVerbatim statement).

The control structures in Mathematica have to be completely located inside one notebook cell (e.g. loop cannot start in once cell and end in another cell). Thus, the following input is in Mathematica incorrect

\[
\text{Do}[
\text{Print}[i],
(i, 1, 5)]
\]

AceGen extends the functionality of Mathematica with the cross-cell form of If and Do control structures. Previous example can be written by using cross-cell form Do construct as follows

\[
\text{SMSDo}[i, 1, 5]
\]

\[
\text{SMSPrintMessage}[i];
\]

\[
\text{SMSEndDo}[]
\]

and using in-cell form as

\[
\text{SMSDo}[\text{Print}[i], (i, 1, 5)]
\]
Conditionals

**SMSIf, SMSElse, SMSEndIf**

- **SMSIf[condition, t, f]**
  creates code that evaluates *t* if *condition* evaluates to True, and *f* if it evaluates to False and returns the auxiliary variable that during the *AceGen* session represents both options. *t* and *f* can be compound expressions.

- **SMSIf[condition, t]**
  creates code that evaluates *t* if *condition* evaluates to True

- **SMSIf[condition, t, f, "Inline"->True]**
  creates an expression that evaluates *t* if *condition* evaluates to True, and evaluates *f* if it evaluates to False. The command would not produce a new auxiliary variable. It creates a SMSConditional construct that during the *AceGen* session represents both options and remains a part of an expression. The command would, in a target language, generate a code where the conditional statement is coded using inline if or ternary operator [https://en.wikipedia.org/wiki/%3F: Inline if](https://en.wikipedia.org/wiki/%3F):. Inline if is an appropriate choice for the cases when all expressions are a simple (not a compound expressions). Compound expressions would not be correctly interpreted! When correctly used the inline operator leads to faster and more optimized code.

- **SMSIf[condition, t, f, "InsertBefore"->pos]**
  the created code is inserted before the given position where *pos* is:
  - False ⇒ insert code at the current position (also the default value of the option)
  - Automatic ⇒ insert code after the position of the last auxiliary variable referenced by *t* or *f*
  - *counter* ⇒ insert code before the Do loop with the counter *counter*
  - *var* ⇒ insert code after the position of the given auxiliary variable *var*

  Syntax of the "in-cell" If construct.

The "in-cell" form of the SMSIf construct is a direct equivalent of the standard If statement. The *condition* of "If" construct is a logical statement. The in-cell form of the SMSIf command returns multi-valued auxiliary variable with random signature that represents both options. If *t* or *f* evaluates to Null then SMSIf returns Null. If *t* and *f* evaluate to vectors of the same length then SMSIf returns a corresponding vector of multi-valued auxiliary variables.

**Warning:** The "==" operator has to be used for comparing expressions. In this case the actual comparison will be performed at the run time of the generated code. The "===" operator checks exact syntactical correspondence between expressions and is executed in Mathematica at the code derivation time and not at the code run time.

- **SMSIf[condition]**
  starts the TRUE branch of the *if..else..endif* construct

- **SMSElse[]**
  starts the FALSE branch of the *if..else..endif* construct

- **SMSEndIf[]**
  ends the *if..else..endif* construct

- **SMSEndIf[out_var]**
  ends *if..else..endif* construct and creates fictive instances of the *out_var* auxiliary variables with the random values taken as perturbed average values of all already defined instances

- **SMSEndIf[True, out_var]**
  creates fictive instances of the *out_var* auxiliary variables with the random values taken as perturbed values of the instances defined in TRUE branch of the "If" construct

- **SMSEndIf[False, out_var]**
  creates fictive instances of the *out_var* auxiliary variables with the random values taken as perturbed values of the instances defined in FALSE branch of the "If" construct
Syntax of the "cross-cell" If construct.

Formulae entered in between SMSIf and SMSElse will be evaluated if the logical expression condition evaluates to True. Formulae entered in between SMSElse and SMSEndIf will be evaluated if the logical expression evaluates to False. The SMSElse statement is not obligatory. New instances and new signatures are assigned to the out_var auxiliary variables. The out_var parameter can be a symbol or a list of symbols. The values of the symbols have to be multi-valued auxiliary variables. The cross-cell form of the SMSIf command returns the logical auxiliary variable where the condition is stored. The SMSElse command also returns the logical auxiliary variable where the condition is stored. The SMSEndIf command returns new instances of the out_var auxiliary variables or empty list. New instances have to be created for all auxiliary variables defined inside the "If" construct that are used also outside the "If" construct.

Interaction between automatic differentiation and an arbitrary program structure can result in redundant code. The user is strongly encouraged to keep "If" constructs as simple as possible to avoid redundant dependencies.

Example 1: Generic example - in-cell form

Generation of the Fortran subroutine which evaluates the following function

\[
\begin{align*}
  f(x) &= \begin{cases} 
    x^2 & \text{if } x \leq 0 \\
    \sin(x) & \text{if } x > 0 
  \end{cases}
\end{align*}
\]

This initializes the AceGen system and starts the description of the "test" subroutine.

\[
\begin{align*}
  \text{Method : test} & \quad 3 \text{ formulae, 16 sub-expressions} \\
  \text{File created : test.f} & \quad \text{Size : 861 bytes}
\end{align*}
\]

| User : USER |
| Evaluation time : 0 s |
| Mode : Optimal |
| Number of formulae : 3 |
| Method: Automatic |
| Subroutine size : test size :16 |
| Total size of Mathematica code : 295 bytes |

SUBROUTINE test(v,x,f)
IMPLICIT NONE
include 'sms.h'
LOGICAL b2
DOUBLE PRECISION v(5001),x,f
IF(x.le.0d0) THEN
  v(3)=x**2
ELSE
  v(3)=dsin(x)
ENDIF
f=v(3)
END

Example 2: Generic example - cross-cell form

Generation of the Fortran subroutine which evaluates the following function
This initializes the AceGen system and starts the description of the "test" subroutine.

\[ f(x) = \begin{cases} 
  x^2 & \text{if } x \leq 0 \\
  \sin(x) & \text{if } x > 0 
\end{cases} \]

- This initializes the AceGen system and starts the description of the "test" subroutine.

```plaintext
In[207]:= << AceGen;
SMSInitialize["test", "Language" -> "Fortran"];
SMSModule["test", Real[x$$, f$$]];
x \mapsto SMSReal[x$$];
```

- Description of the cross-cell "If" construct.

```plaintext
In[211]:= SMSIf[x <= 0] 
   \downarrow  \leq 0
In[212]:= f = x^2;
In[213]:= SMSElse[] 
   \downarrow  \leq 0
In[214]:= f = \sin(x);
In[215]:= SMSEndIf[f] 
   \downarrow
In[216]:= SMSExport[f, f$$];
SMSWrite["test"];
```

- This displays the contents of the generated file.

```plaintext
In[218]:= FilePrint["test.f"]!
```

---

**Example 3: Generic example - using ternary operator**

Generation of the Fortran subroutine which evaluates the following function

\[ f(x) = \begin{cases} 
  x^2 & \text{if } x \leq 0 \\
  \sin(x) & \text{if } x > 0 
\end{cases} \]
\[ f(x) = \begin{cases} x \leq 0 & x^2 \\ x > 0 & \sin(x) \end{cases} \]

- This initializes the AceGen system and starts the description of the "test" subroutine.

```plaintext
<<AceGen`; SMSInitialize["test", "Language" -> "Fortran"];
SMSModule["test", Real[x$$, f$$]]; 
x = SMSReal[x$$];
f = SMSIf[x <= 0, x^2, Sin[x], "Inline" -> True];
SMSExport[f, f$$];
SMSWrite["test"]; 
```

### Example 4: Incorrect logical expression

The expression \( x \leq 0 \land \text{\texttt{i }}=="0" \) in this example is evaluated already in Mathematica because the \( \text{\texttt{==}} \) operator always yields True or False. The correct form of the logical condition would be \( x \leq 0 \land \text{\texttt{i}}==0 \).

```plaintext
<<AceGen`; SMSInitialize["test", "Language" -> "Fortran"];
SMSModule["test", Real[x$$, f$$], Integer[i$$]]; 
x = SMSReal[x$$];
f = SMSIf[x <= 0 \&\& \text{\texttt{i}}=="0", x^2, Sin[x]]; 
SMSExport[f, f$$];
SMSWrite[];
```

The expressions of the form \( a==b \) or \( a!=b \) in Hold[\( x \leq 0 \land \text{\texttt{i}}==0 \)] are evaluated in Mathematica and not later in the source code !!!

Consider using \( a==b \) or \( a!=b \) instead. See also: SMSIf
Example 5: Incorrect use of ternary operator

Generation of the C subroutine which evaluates the following function

\[
f(x) = \begin{cases} 
  x^2 & \text{if } x \leq 0 \\
  x^2 & \text{if } x > 0 
\end{cases}
\]

```
In[136]:= << AceGen
   SMSInitialize["test", "Language" -> "C"];
   SMSModule["test", Real[x$$, f$$]]; 
   x + SMSReal[x$$];
```

- Compound expression \( y = \cos(x) \); \( y + y^2 \) will be incorrectly interpreted. \( \cos(x) \) will be evaluated also when \( x \leq 0 \) as seen from the generated code.

```
In[138]:= f = SMSIf[x <= 0, x^2, y + Cos[x]]; y + y^2, "Inline" -> True];
   SMSExport[f, f$$]; 
   SMSWrite["test"];
   FilePrint["test.c"]
```
Example 6: Redundant code

Generation of the C subroutine which evaluates derivative of $f$ with respect to $x$.

$$f(x) = \begin{cases} x^2 & \text{for } x \leq 0 \\ \sin(x) & \text{for } x > 0 \end{cases}$$

The first input given below leads to the construction of redundant code. The second differentiation involves $f$ that is also defined in the first "if" construct, so the possibility that the first "if" was executed and that somehow effects the second one has to be considered.

This redundant dependency is avoided in the second input by the use of temporary variable $tmp$ and leading to much shorter code.

- Input that results in inefficient code.

```mathematica
<< AceGen`
SMSInitialize["test", "Language" -> "C"];
SMSModule["test", Real[x$$, f$$, d$$]];
x = SMSReal[x$$];
SMSIf[x <= 0]; f = x^2;
d = SMSD[f, x];
SMSEndIf[f, d];
SMSIf[x > 0]; f = Sin[x];
d = SMSD[f, x];
SMSEndIf[f, d];
SMSExport[{f, d}, {f$$, d$$}];
SMSWrite[]
```

Method : test 7 formulae, 39 sub-expressions
In[358]:= FilePrint["test.c"]

/**************************
S U B R O U T I N E
**************************/
void test(double v[],double (*x),double (*f),double (*d))
{
 int b2,b6,b7;
 b2=(+x)<@0e0;
 if(b2){
  v[3]=Power((+x),2);
  v[5]=2e0*(+x);
 } else {
  }
 if((+x)>@0e0){
  if(b2){
   v[8]=2e0*(+x);
  } else [{
  }
  v[8]=cos((+x));
  v[3]=sin((+x));
  v[5]=v[8];
 } else {
  }
  (*f)=v[3];
  (*d)=v[5];
};

/* Corrected input. */

In[358]:= SMSInitialize["test", "Language" -> "C", "Mode" -> "Optimal"];
SMSModule["test", Real[x$$, f$$, d$$]];
 x = SMSReal[x$$];
 SMSIf[x <= @0];
 f = x$$;
 d = SMSD[f, x];
 SMSEndIf[f, d];
 SMSIf[x > @0];
 tmp = Sin[x];
 f += tmp;
 d = SMSD[tmp, x];
 SMSEndIf[f, d];
 SMSExport[(f, d), {f$$, d$$}];
 SMSWrite[]

Method := test 5 formulae, 30 sub-expressions

[0] File created: test.c  Size : 863
  0.361
Example 7: Variables out of scope

Generation of the Fortran subroutine which evaluates the following function

\[ f(x) = \begin{cases} 
  x^2 & \text{if } x \leq 0 \\
  \sin(x) & \text{if } x > 0 
\end{cases} \]

This symbol \( f \) appears outside the "If" construct. Since \( f \) is not specified in the \texttt{SMSEndIf} statement, we get "variable out of scope" error message.
Example 8: Variables out of scope 2

This is an example where a variable defined in one branch of "if" construct is used in another branch of "if" construct resulting in "Out of scope" message.

- Input that results in "Out of scope" message.

Input:  ```mathematica
In[418]:= <<AceGen`; SMSInitialize["test", "Language" -> "Mathematica", "Mode" -> "Debug"];
SMSModule["test", Integer[NN$]];
NN = SMSInteger[nmax$];
SMSIf[NN < 10];
  NN := 20;

  time=0 variable= 0 = {}
```

- Last definition of nmax is here

Input:  ```mathematica
In[424]:= SMSElse[];
```

- nmax from the last definition can’t be used here because it is out of the scope (it is a part of another if branch)

Input:  ```mathematica
In[425]:= SMSIf[NN < 5];
  NN := 0;
  SMSEndIf[];
SMSEndIf[NN];
SMSExport[NN, r$];
SMSWrite[];
```

- An additional variable NNaux is introduced at the point that is within the scope of the position where variable NN is used.
In[431]:= <<AceGen`
SMSInitialize["test", "Language" -> "Mathematica", "Mode" -> "Debug"];
SMSModule["test", Integer[NN$$]];
NN = SMSInteger[nmax$$];
NNaux = NN;
SMSIf[NN < 10];
    NN = 20;
SMSElse[];
    SMSIf[NNaux < 5];
    NN = 0;
    SMSEndIf[];
SMSEndIf[NN];
SMSExport[NN, r$$];
SMSWrite[];

time = 0 variable = 0 = {};
[0] Consistency check - global
[0] Consistency check - expressions
[0] Generate source code :
Events: 0
[0] Final formatting

File: test.m  Size: 1343
Methods No.Formulae No.Leafs
  test  12  28

SMSSwitch

SMSSwitch[expr, form1, value1, form2, value2, ...]
Creates code that evaluates expr, then compares it with each of the formi in turn, evaluating and returning the valuei corresponding to the first match found. The value returned during the AceGen session represents all options (see also: Program Flow Control)

SMSSwitch[expr, form1, value1, form2, value2, ..., default_value]
definition of the default_value

SMSSwitch[expr, form1, value1, form2, value2, ..., "Inline" -> True]
creates an expression that evaluates expr, then compares it with each of the formi in turn, evaluating and returning the valuei corresponding to the first match found.

The command would not produce a new auxiliary variable. It creates a SMSConditional construct that during the AceGen session represents all options and remains a part of an expression. The command would, in a target language, generate a code where the conditional statement is coded using inline if or ternary operator https://en.wikipedia.org/wiki/%3F:

Inline switch is an appropriate choice for the cases when all expressions are a simple (not a compound expressions).
Compound expressions would not be correctly interpreted! When correctly used the inline operator leads to faster and more optimized code.

Syntax of the SMSSwitch construct.

The SMSSwitch construct is a direct equivalent of the standard Switch statement. The expr and the formi are integer type expressions. The SMSSwitch command returns multi-valued auxiliary variable with random signature that represents all options. If all valuei evaluate to Null then SMSSwitch returns Null. If all valuei evaluate to vectors of the same length then SMSSwitch returns a corresponding vector of multi-valued auxiliary variables.

Warning: If none of the formi match expr, the SMSSwitch returns arbitrary value.
SMSWhich

SMSWhich[test1, value1, test2, value2, ...]

Creates code that evaluates each of the testi in turn, returning the value of the valuei corresponding to the first one that yields True. The value returned during the AceGen session represents all options (see also: Program Flow Control).

SMSWhich[test1, value1, test2, value2, ..., True, default_value]

definition of the default_value

SMSWhich[test1, value1, test2, value2, ..., "Inline" -> True]

creates an expression evaluates each of the testi in turn, returning the value of the valuei corresponding to the first one that yields True.

The command would not produce a new auxiliary variable. It creates a SMSConditional construct that during the AceGen session represents all options and remains a part of an expression. The command would, in a target language, generate a code where the conditional statement is coded using inline if or ternary operator https://en.wikipedia.org/wiki/%3F:

Inline which is an appropriate choice for the cases when all expressions are simple (not a compound expressions). Compound expressions would not be correctly interpreted! When correctly used the inline operator leads to faster and more optimized code.

Syntax of the SMSWhich construct.

The SMSWhich construct is a direct equivalent of the standard Which statement. The testi are logical expressions. The SMSWhich command returns multi-valued auxiliary variable with random signature that represents all options. If all valuei evaluate to Null then SMSWhich returns Null. If all valuei evaluate to vectors of the same length then SMSWhich returns a corresponding vector of multi-valued auxiliary variables.

Warning: If none of the testi evaluates to True, the SMSWhich returns arbitrary value.

Warning: The "==" operator has to be used for comparing expressions. In this case the actual comparison will be performed at the run time of the generated code. The "===" operator checks exact syntactical correspondence between expressions and is executed in Mathematica at the code derivation time and not at the code run time.

Loops

SMSDo, SMSEndo

SMSDo[expr, \{i, i_{min}, i_{max}, di\}]

create code that evaluates expr with the variable i successively taking on the values i_{min} through i_{max} in steps of di. Possible forms:

SMSDo[expr, \{i, i_{max}\}] \equiv SMSDo[expr, \{i, i_{min}, i_{max}, 1\}]

SMSDo[expr, \{i, i_{min}, i_{max}\}] \equiv SMSDo[expr, \{i, 1, i_{max}\}]

SMSDo[expr, \{i, i_{min}, i_{max}, di, in\_out\_var\}]

create code that evaluates expr with the variable i successively taking on the values i_{min} through i_{max} in steps of di and define input/output in\_out\_var variables of the loop

SMSDo[expr, \{i, i_{min}, i_{max}, di, in\_var, out\_var\}]

create code that evaluates expr with the variable i successively taking on the values i_{min} through i_{max} in steps of di and define input in\_var and output out\_var variables of the loop

Syntax of the "in-cell" loop construct.

SMSDo[i, i_{min}, i_{max}]

start the "Do" loop with an auxiliary variable i successively taken on the values i_{min} through i_{max} (in steps of 1)

SMSDo[i, i_{min}, i_{max}, di]

start the "Do" loop with an auxiliary variable i successively taken on the values i_{min} through i_{max} in steps of di
start the "Do" loop with an auxiliary variable $i$ successively taken on the values $i_{\text{min}}$ through $i_{\text{max}}$ in steps of $di$ and create fictive instances of the $in\_var$ auxiliary variables

$\text{SMSEndo[]}$
end the loop

$\text{SMSEndo[out\_var]}$
end the loop and create fictive instances of the $out\_var$ auxiliary variables

Syntax of the "cross-cell" loop construct.

Optimization procedures (see Expression Optimization) require that a new instance with the random signature have to be created for:

⇒ all auxiliary variables that are imported into the loop and have values changed inside the loop ($in\_var$ and $in\_out\_var$),

⇒ all variables that are defined inside the loop and used outside the loop ($out\_var$ and $in\_out\_var$).

New instances with random signature are assigned to the $in\_var$ and $in\_out\_var$ variables at the start of the loop and to the $out\_var$ and $in\_out\_var$ auxiliary variables at the end of the loop. The "in-cell" form of SMSDo command returns new instances of the $in\_out\_var$ auxiliary variables or empty list. The "cross-cell" form of SMSDo command returns new instances of the $in\_var$ auxiliary variables or empty list. The SMSEndo command returns new instances of the $out\_var$ variables or empty list.

The $in\_var$, $out\_var$ and $in\_out\_var$ parameters can be a symbol or a list of symbols. The values of the symbols have to be multi-valued auxiliary variables. The iteration variable of the "Do" loop is an integer type auxiliary variable ($i$).

Example 1: Generic example (in-cell)

Generation of the Fortran subroutine which evaluates the following sum $f(x) = 1 + \sum_{i=1}^{n} x^i$.

```fortran
f[i] += f + xi;
{(i, 1, n, 1, f)};
SMSExport[f, f$]$
SMWrite["test"]$
FilePrint["test.f"]
```
Example 2: Generic example (cross-cell)

Generation of the Fortran subroutine which evaluates the following sum \( f(x) = 1 + \sum_{i=1}^{n} x^i \).

- This initializes the AceGen system and starts the description of the "test" subroutine.

```plaintext
In[133]:= << AceGen;
SMSInitialize["test", "Language" -> "Fortran"];
SMSSModule["test", Real[xx, ff], Integer[nn]];
x + SMSReal[xx];
n + SMSInteger[nn];
```

- Description of the loop.

```plaintext
In[133]= f = 1;
SMSDo[i, 1, n, 1, f];
  f + f + x;i;
SMSEndo[f];
```

- This assigns the result to the output parameter of the subroutine and generates file "test.for".

```plaintext
In[133]= SMSExport[f, ff];
SMSWrite["test"];
```

- This displays the contents of the generated file.
Example 3: Incorrect and correct use of "Do" construct

Generation of Fortran subroutine which evaluates the n-th term of the following series \( S_0 = 0, S_n = \cos S_{n-1} \).

- Incorrect formulation
  
  Since the signature of the \( S \) variable is not random at the beginning of the loop, AceGen makes wrong simplification and the resulting code is incorrect.

- Correct formulation
  
  In the expression of the form \( x = f(x) \), \( x \) appears to have a constant value! \( x = 0 \) See also: SMSS
**Correct formulation**

Assigning a random signature to the `S` auxiliary variable prevents wrong simplification and leads to the correct code.

```plaintext
In[152]:= SMSInitialize["test", "Language" -> "Fortran", "Mode" -> "Optimal"];
SMSModule["test", Real[S$$], Integer[n$$]];
   n ⊢ SMSInteger[n$$];
   S = 0;
SMSDo[i, 1, n, 1, S];
   S = Cos[S];
SMSEndDo[S];
SMSExport[S, S$$];
SMSWrite["test"];
FilePrint["test.f"]
```
The "in-cell" form by default assigns the random signature to $S$ at the beginning and at the end of the loop, thus gives correct result.
Example 4: How to use variables defined inside the loop outside the loop?

Only the multi-valued variables (introduced by the \( \uparrow \) or \( \downarrow \) command) can be used outside the loop. The use of the single-valued variables (introduced by the \( \leftarrow \) or \( \rightarrow \) command) that are defined within loop outside the loop will result in Variables out of scope error.

- Here the variable X is defined within the loop and used outside the loop.
- Incorrect formulation
In[182]:= << AceGen;
SMSInitialize["test", "Language" -> "Fortran"];
SMSModule["test", Real[$$], Integer[$$]];
n + SMSInteger[$$];
S = 0;
SMSDo[
    X = Cos[S];
    S = S + X;
    , {i, 1, n, 1, {S}}
];
Y = X^2;

Some of the auxiliary variables in expression are defined outside the scope of the current position.
Module: test Description: Error in user input parameters for function: ≠
Input parameter: ≠ Current scope: {}
Misplaced variables :
   1 = 5V[4, 1] Scope: Do[i, 1, j, 1]
Events: 0
Version: 5.001 Windows (26 Dec 12) (MMA 9.) Module: ≠
See also: Auxiliary Variables AceGen Troubleshooting

$Aborted

Correct formulation for "in-cell" form

In[189]:= << AceGen;
SMSInitialize["test", "Language" -> "Fortran"];
SMSModule["test", Real[$$], Integer[$$]];
n + SMSInteger[$$];
S = 0;
SMSDo[
    i, 1, n, 1, S
    X = Cos[S];
    S = S + X;
    , {i, 1, n, 1, {S}, {S, X}}
];
Y = X^2;

Correct formulation for "cross-cell" form

In[206]:= << AceGen;
SMSInitialize["test", "Language" -> "Fortran"];
SMSModule["test", Real[$$], Integer[$$]];
n + SMSInteger[$$];
S = 0;
SMSDo[1, 1, n, 1, S
    X = Cos[S];
    S = S + X;
SMSEndDo[S, X];
Y = X^2;

SMSReturn, SMSBreak, SMSContinue

SMSReturn[] ≡ SMSVerbatim["C"->"return;", "Fortran"->"return", "Mathematica"->"Return[Null,Module];"]
(see Mathematica command Return)

\[
\text{SMSBreak[]} \equiv \text{SMSVerbatim["C"->"break;", "Fortran"->"exit", "Mathematica"->"Break[];"]}
\]

(see Mathematica command Break)

\[
\text{SMSContinue[]} \equiv \text{SMSVerbatim["C"->"continue;", "Fortran"->"cycle", "Mathematica"->"Continue[];"]}
\]

(see Mathematica command Continue)
Symbolic-Numeric Interface

Contents
- General Description
- Input Parameters
  - SMSReal
  - SMSInteger
  - SMSLogical
  - SMSRealList
- Output Parameters
  - SMSExport

General Description
A general way of how to pass data from the main program into the automatically generated routine and how to get the results back to
the main program is through external variables. External variables are used to establish the interface between the numerical environ-
ment and the automatically generated code.

External variables appear in a list of input/output parameters of the declaration of the subroutine, as a part of expression, and when
the values are assigned to the output parameters of the subroutine.

<table>
<thead>
<tr>
<th>definition of the input/output parameters</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMSModule[&quot;name&quot;, Real[real variables], Integer[integer type variables], Logical[logical variables]]</td>
<td>SMSModule[&quot;test&quot;, Real[y$$[2,5]]]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>external variables as a part of expression</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMSReal[real external data]</td>
<td>y = 2 Sin[SMSReal[y$$[2,5]]]</td>
</tr>
<tr>
<td>SMSInteger[integer external data]</td>
<td>i = SMSInteger[i$$]</td>
</tr>
<tr>
<td>SMSLogical[logical data]</td>
<td>l = SMSLogical[bool$$] &amp; &amp; y &lt; 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>exporting values</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMSExport[value, real external]</td>
<td>SMSExport[x+5, y$$[2,5]]</td>
</tr>
<tr>
<td>SMSExport[value, integer external]</td>
<td>SMSExport[2 i+7, i$$]</td>
</tr>
<tr>
<td>SMSExport[value, logical external]</td>
<td>SMSExport[True, bool$$]</td>
</tr>
</tbody>
</table>

Use of external variables.

The form of the external variables is prescribed and is characterized by the $ signs at the end of its name. The standard AceGen form is
automatically transformed into the chosen language when the code is generated. The standard formats for external variables when
they appear as part of subroutine declaration and their transformation into FORTRAN and C language declarations are as follows:
Arrays can have arbitrary number of dimensions. The dimension can be an integer constant, an integer external variable or a "*" character constant. The "*" character stands for the unknown dimension.

The standard format for external variables when they appear as part of expression and their transformation into FORTRAN and C language formats is then:

<table>
<thead>
<tr>
<th>type</th>
<th>AceGen form</th>
<th>FORTRAN form</th>
<th>C form</th>
</tr>
</thead>
<tbody>
<tr>
<td>real variable</td>
<td>SMSReal[x$$]</td>
<td>x*8 x</td>
<td>double *x</td>
</tr>
<tr>
<td>real array</td>
<td>SMSReal[x$$[10]]</td>
<td>x (10)</td>
<td>x[10]</td>
</tr>
<tr>
<td>integer variable</td>
<td>SMSInteger[i$$]</td>
<td>i</td>
<td>int *i</td>
</tr>
<tr>
<td>integer array</td>
<td>SMSInteger[i$$[10]]</td>
<td>i (10)</td>
<td>i[10]</td>
</tr>
<tr>
<td>logical variable</td>
<td>SMSLogical[l$$]</td>
<td>l</td>
<td>int *l</td>
</tr>
</tbody>
</table>

A characteristic high precision real type number called "signature" is assigned to each external variable. This characteristic real number is then used throughout the AceGen session for the evaluation of the expressions. If the expression contains parts which cannot be evaluated with the given signatures of external variables, then AceGen reports an error and aborts the execution.

External variable is represented by the data object with the head SMSExternalF. This data object represents external expressions together with the information regarding signature and the type of variable.

**Input Parameters**

**SMSReal**

**SMSReal[exte]**
≡ create real type external data object (SMSExternalF) with the definition exte and an unique signature

**SMSReal[i_List]**
≡ Map[SMSReal,i]
Introduction of the real type external variables or input parameters of the subroutine.

<table>
<thead>
<tr>
<th>option</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Dependency&quot;</td>
<td>True</td>
<td>define partial derivatives of external data object (SMSExternalF) with respect to given auxiliary variables (for the detailed syntax see SMSFreeze, User Defined Functions)</td>
</tr>
<tr>
<td>&quot;Subordinate&quot;</td>
<td>{}</td>
<td>list of auxiliary variables that represent control structures (e.g. SMSCall, SMSVerbatim, SMSExport) that have to be executed before the evaluation of the current expression (see User Defined Functions)</td>
</tr>
</tbody>
</table>

Options of the SMSReal function.

The SMSReal function does not create a new auxiliary variable. If an auxiliary variable is required, then we have to use one of the functions that introduces a new auxiliary variable (e.g. n-SMSReal[i$$]). The exte is, for the algebraic operations like differentiation, taken as independent on any auxiliary variable that might appear as part of exte. The parts of the exte which have proper form for the external variables are at the end of the session translated into FORTRAN or C format.

By default an unique signature (random high precision real number) is assigned to the SMSExternalF object. If the numerical evaluation of exte (obtained by N[exte,SMSEvaluatePrecision]) leads to the real type number then the default signature is calculated by it’s perturbation, else the default signature is taken as a real type random number form interval [0,1]. In some cases user has to provide it’s own signature in order to prevent situations where wrong simplification of expressions might occur (for mode details see Signatures of Expressions).

**SMSInteger**

- **SMSInteger[exte]**
  - create integer type external data object (SMSExternalF) with the definition exte and an unique signature

- **SMSInteger[exte,Hash]**
  - create integer type external data object (SMSExternalF) with the definition exte and a hash value of exte as signature (signature is not unique, thus simplifications are allowed)

Introduction of integer type external variables.

<table>
<thead>
<tr>
<th>option</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Subordinate&quot;</td>
<td>{}</td>
<td>list of auxiliary variables that represent control structures (e.g. SMSCall, SMSVerbatim, SMSExport) that have to be executed before the evaluation of the current expression (User Defined Functions)</td>
</tr>
</tbody>
</table>

Options of the SMSInteger function.

The SMSInteger function does not create a new auxiliary variable. If an auxiliary variable is required, then we have to use one of the functions that introduces a new auxiliary variable (e.g. i-SMSInteger[i$$]). In order to avoid wrong simplifications an unique real type signature is assigned to the integer type variables.

**SMSLogical**

- **SMSLogical[exte]**
  - create logical type external data object with definition exte
Logical expressions are ignored during the simultaneous simplification procedure. The SMSLogical function does not create a new auxiliary variable. If an auxiliary variable is required, then we have to use one of the functions that introduces a new auxiliary variable (e.g. `b ⊢ SMSLogical[b]).

**SMSRealList**

`SMSRealList[{eID_1, eID_2, ...}, vector_Function]`

The primal purpose of the SMSRealList command is dynamic allocation of memory within the given array. Function returns a list of real type external data objects that corresponds to the list of array element identifications `{eID_1, eID_2, ...}` and represents consecutive elements of the array.

**SMSRealList[pattern]**

gives the real type external data objects that correspond to elements which array element identification `eID` match pattern

**SMSRealList[pattern, return_code]**

gives the data accordingly to the `code` that correspond to elements which array element identification `eID` match pattern

Introduction of the list of real type external variables.

### Options of the SMSRealList function.

<table>
<thead>
<tr>
<th>option</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Description&quot;-&gt; {...}</td>
<td><code>{eID_1, eID_2, ...}</code></td>
<td>a list of descriptions that corresponds to the list of array element identifications <code>{eID_1, eID_2, ...}</code></td>
</tr>
<tr>
<td>&quot;Length&quot;-&gt;<code>l</code></td>
<td>1</td>
<td>each array element identification <code>eID</code>, can also represent a part of <code>array</code> with the given length</td>
</tr>
<tr>
<td>&quot;Index&quot;-&gt;<code>i</code></td>
<td>1</td>
<td>index of the actual array element taken from the part of <code>array</code> associated with the array element identification <code>eID</code> (index starts with 1)</td>
</tr>
<tr>
<td>&quot;Signature&quot;</td>
<td><code>{1,1,...}</code></td>
<td>a list of characteristic real type values that corresponds to the list of array element identifications <code>{eID_1, eID_2,...}</code></td>
</tr>
</tbody>
</table>

Return codes for SMSRealList.

<table>
<thead>
<tr>
<th>return_code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Description&quot;</td>
<td>the values of the option &quot;Description&quot;</td>
</tr>
<tr>
<td>&quot;Signature&quot;</td>
<td>the values of the option &quot;Signature&quot;</td>
</tr>
<tr>
<td>&quot;Export&quot;</td>
<td>the patterns (e.g. ed$S[5]) suitable as parameter for SMSExport function</td>
</tr>
<tr>
<td>&quot;Length&quot;</td>
<td>the accumulated length of all elements which array element identification <code>eID</code> match pattern</td>
</tr>
<tr>
<td>&quot;ID&quot;</td>
<td>array element identifications</td>
</tr>
<tr>
<td>&quot;Plain&quot;</td>
<td>external data objects with all auxiliary variables replaced by their definition</td>
</tr>
<tr>
<td>&quot;Exist&quot;</td>
<td>True if the data with the given pattern exists</td>
</tr>
<tr>
<td>&quot;Start&quot;</td>
<td>if the external data objects is an array then the first element of the array (Index=1) with all auxiliary variables replaced by their definition</td>
</tr>
</tbody>
</table>


The SMSRealList command remembers the number of array elements allocated. When called second time for the same array the consecutive elements of the array are taken starting from the last element from the first call. The array element identifications $eID$ is a string that represents the specific element of the array and can be used later on (through all the AceGen session) to retrieve the element of the array that was originally assigned to $eID$.

The parameter array is a pure function that returns the $i$-th element of the array. For the same array it should be always identical. The definitions $x$$[#]& and $x$$[#+1]& are considered as different arrays.

Example

In[17]:= << AceGen`
SMSInitialize["test", "Language" -> "C"]
SMSModule["test", Real[a$$[10], b$$[10], c$$[100]], Integer[L$$, i$$]]

Out[17]=
SMSRealList["a1", "a2"], a$$[#] &

In[18]:= Out[18]=

In[19]:= SMSRealList["a3"]

Out[19]= a$$3

In[20]:= SMSRealList[{"b1", "b2"}, b$$[#] &] | "Length" -> 5, "Index" -> 2

Out[20]=

In[21]:= Out[21]=

In[22]:= SMSRealList[{"b3", "b4"}, b$$[#] &] | "Length" -> 20, "Index" -> 4

Out[22]=

In[23]:= SMSRealList[Array["β", 2], c$$[#] &] | "Length" -> Hold[2 L], "Index" -> i + 1

Out[23]=

In[24]:= TableForm[{
SMSRealList["β"[1], "ID"], SMSRealList["β"[2]], SMSRealList["β"[3], "Plain"],
SMSRealList["β"[4], "Export"]}, TableHeadings -> {("ID", "AceGen", "Plain", "Export"), None}]

Out[24]= TableForm[

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AceGen</td>
<td>c$$[1]</td>
<td>c$$[2]</td>
</tr>
</tbody>
</table>

In[25]:= SMSRealList["β"[1], "Length"]

Out[25]= 2 L$$
Output Parameters

SMSExport

SMSExport[exp, ext]
export the expression exp to the external variable ext

SMSExport[{exp_1, exp_2, ..., exp_N}, ext]
≡ SMSExport[{exp_1, exp_2, ..., exp_N}, Table[ext[i], {i, 1, N}]]
export the list of expressions \{exp_1, exp_2, ..., exp_N\} to the external array ext formed as Table[ext[i], {i, 1, N}]

SMSExport[exp, ext, "AddIn"->True]
add the value of exp to the current value of the external variable ext

SMSExport[exp, ext, "Atomic"->True]
the "#pragma omp atomic" statement is added to ensures that race conditions are avoided in the case of parallelization

SMSExport[exp, ext, "Position"->"Optimal"]
Regulates how the "export" object is treated during the code optimisation. "Optimal" ⇒ optimal position is determined based on the dependency of exp and ext on other variables.

Options for the SMSExport function.

The expressions that are exported can be any regular expressions. The external variables have to be regular AceGen external variables.

At the end of the session, the external variables are translated into the FORTRAN or C format.

```
SMSShow;  
SMSInitialize["test", "Language"->"Fortran"];  
SMSModule["test", Real[x$$, y$$, a$$[2], r$$[2, 2]]];  
x = SMSReal[x$$];  
SMSExport[x^2, y$$];  
(* three equivalent forms how to export list of two values*)  
SMSExport[{3, 4}, {a$$[1], a$$[2]}];  
SMSExport[{5, 6}, Table[a$$[i, (i, 1, 2)]];  
(* two equivalent forms how to export two-dimensional array*)  
SMSExport[Table[Sin[ij], {i, 2}, {j, 2}], r$$];  
SMSExport[Table[Sin[ij], {i, 2}, {j, 2}], Table[r$$[i, j], {i, 2}, {j, 2}]];  
SMSSend["test"];  
```
SUBROUTINE test(v,x,y,a,r)
  IMPLICIT NONE
  include 'sms.h'
  DOUBLE PRECISION v(5001),x,y,a(2),r(2,2)
  y=x**2
  a(1)=1d0
  a(2)=2d0
  a(1)=3d0
  a(2)=4d0
  a(1)=5d0
  a(2)=6d0
  r(1,1)=dsin(1d0)
r(1,2)=dsin(2d0)
r(2,1)=dsin(2d0)
r(2,2)=dsin(4d0)
r(1,1)=dsin(1d0)
r(1,2)=dsin(2d0)
r(2,1)=dsin(2d0)
r(2,2)=dsin(4d0)
END
Theory of Automatic Differentiation

Differentiation is an algebraic operation that plays crucial role in the development of new numerical procedures. We can easily recognize some areas of numerical analysis where the problem of analytical differentiation is emphasized:

- evaluation of consistent tangent matrices for non-standard physical models,
- sensitivity analysis according to arbitrary parameters,
- optimization problems,
- inverse analysis.

In all these cases, the general theoretical solution to obtain exact derivatives is still under consideration and numerical differentiation is often used instead. The automatic differentiation generates a program code for the derivative from a code for the basic function.

Throughout this section we consider function $y = f(v)$ that is defined by a given sequence of formulae of the following form

$$\text{For } i = n+1, n+2, \ldots, m$$

$$v_i = f_i(v_j) \mid_{v_j}$$
Here functions \( f_i \) depend on the already computed quantities \( v_j \). This is equivalent to the vector of formulae in AceGen where \( v_j \) are auxiliary variables. For functions composed from elementary operations, a gradient can be derived automatically by the use of symbolic derivation with Mathematica. Let \( v_i, i = 1 \ldots n \) be a set of independent variables and \( v_j, i = n+1,n+2,\ldots,m \) a set of auxiliary variables. The goal is to calculate the gradient of \( y \) with respect to the set of independent variables \( \nabla y = \{ \partial y / \partial v_1, \partial y / \partial v_2, \ldots, \partial y / \partial v_n \} \). To do this we must resolve dependencies due to the implicitly contained variables. Two approaches can be used for this, often recalled as forward and reverse mode of automatic differentiation.

The forward mode accumulates the derivatives of auxiliary variables with respect to the independent variables. Denoting by \( \nabla v_i \) the gradient of \( v_i \) with respect to the independent variables \( v_j, j = 1 \ldots n \), we derive from the original sequence of formulae by the chain rule:

\[
\nabla v_i = \{ \partial v_i / \partial v_1, \partial v_i / \partial v_2, \ldots, \partial v_i / \partial v_n \} \quad \text{for} \quad i = 1, 2, \ldots, m
\]

\[
\text{For } i = n+1,n+2,\ldots,m
\]

\[
\nabla v_i = \sum_{j=1}^{i-1} \frac{\partial v_i}{\partial v_j} \nabla v_j
\]

\[
\nabla y = \nabla v_m
\]

In practical cases gradients \( \nabla v_i \) are more or less sparse. This sparsity is considered automatically by the simultaneous simplification procedure.

In contrast to the forward mode, the reverse mode propagates adjoints, that is, the derivatives of the final values, with respect to auxiliary variables. First we associate the scalar derivative \( \nabla y \) with each auxiliary variable \( v_i \).

\[
\nabla y_i = \partial y / \partial v_i \quad \text{for } i = m,m-1,\ldots,1
\]

\[
\nabla y = \{ \partial y / \partial v_1, \partial y / \partial v_2, \ldots, \partial y / \partial v_n \} \quad \text{for } i = 1, 2, \ldots, n
\]

As a consequence of the chain rule it can be shown that these adjoint quantities satisfy the relation

\[
\nabla y = \sum_{j=i}^{m} \frac{\partial y}{\partial v_j} \nabla v_j
\]

To propagate adjoints, we have to reverse the flow of the program, starting with the last function first as follows

\[
\text{For } i = m,m-1,\ldots,1
\]

\[
\nabla y = \{ \nabla y, \nabla v_2, \ldots, \nabla v_m \}
\]

Again, simultaneous simplification improves the efficiency for the reverse mode by taking into account the actual dependency between variables.

The following simple example shows how the presented procedure actually works. Let us define three functions \( f_1, f_2, f_3 \), dependent on independent variables \( x_i \). The forward mode for the evaluation of gradient \( \nabla v_j = \{ \partial v_j / \partial x_i \} \) leads to

\[
\begin{align*}
\nabla v_1 &= f_1 (x_i) \\
\nabla v_2 &= f_2 (x_i, v_2) \\
\nabla v_3 &= f_3 (x_i, v_2, v_3)
\end{align*}
\]

\[
\text{for } i = 1, 2, \ldots, n
\]

The reverse mode is implemented as follows
\[ v_3 = f_3 (x_i, v_2, v_3) \quad \tilde{v}_3 = \frac{\partial v_3}{\partial x_i} = 1 \]
\[ v_2 = f_2 (x_i, v_1) \quad \tilde{v}_2 = \frac{\partial v_2}{\partial x_i} = \frac{\partial f_2}{\partial v_1} \tilde{v}_1 \]
\[ v_1 = f_1 (x_i) \quad \tilde{v}_1 = \frac{\partial v_1}{\partial x_i} = \frac{\partial f_1}{\partial v_1} \tilde{v}_1 + \frac{\partial f_1}{\partial v_2} \tilde{v}_2 \]
\[ \tilde{x}_i = \frac{\partial v_n}{\partial x_i} = \frac{\partial v_n}{\partial x_i} \tilde{v}_3 + \frac{\partial v_n}{\partial x_i} \tilde{v}_2 + \frac{\partial v_n}{\partial x_i} \tilde{v}_1 \quad i = 1, 2, \ldots, n. \]

By comparing both techniques, it is obvious that the reverse mode leads to a more efficient solution.

AceGen does automatic differentiation by using forward or backward mode of automatic differentiation. The procedure implemented in the AceGen system represents a special version of automatic differentiation technique. The vector of the new auxiliary variables, generated during the simultaneous simplification of the expressions, is a kind of pseudo code, which makes the automatic differentiation with AceGen possible. There are several situations when the formulae and the program structure alone are not sufficient to make proper derivative code. These exceptions are described in chapter Exceptions in Differentiation.

AceGen uses Mathematica's symbolic differentiation functions for the differentiation of explicit parts of the expression. The version of reverse or forward mode of 'automatic differentiation' technique is then employed on the global level for the collection and expression of derivatives of the variables which are implicitly contained in the auxiliary variables. At both steps, additional optimization of expressions is performed simultaneously.

Higher order derivatives are difficult to be implemented by standard automatic differentiation tools. Most of the automatic differentiation tools offer only the first derivatives. When derivatives are derived by AceGen, the results and all the auxiliary formulae are stored on a global vector of formulae where they act as any other formula entered by the user. Thus, there is no limitation in AceGen concerning the number of derivatives which are to be derived.


**SMSD Function**

\[ \text{SMSD}[\text{exp}, v] \quad \text{partial derivative} \quad \frac{\partial \text{exp}}{\partial v} \]

\[ \text{SMSD}[\text{exp}, \{v_1, v_2, \ldots\}] \quad \text{gradient of exp} \quad \left[ \frac{\partial \text{exp}}{\partial v_1}, \frac{\partial \text{exp}}{\partial v_2}, \ldots \right] \]

\[ \text{SMSD}[\{\text{exp}_1, \text{exp}_2, \ldots\}, \{v_1, v_2, \ldots\}] \quad \text{the Jacobian matrix} \quad J = \left[ \frac{\partial \text{exp}_i}{\partial v_j} \right] \]

\[ \text{SMSD}[\text{exp}, \{\{v_{11}, v_{12}, \ldots\}, \{v_{21}, v_{22}, \ldots\}, \ldots\}] \quad \text{differentiation of scalar with respect to matrix} \quad \left[ \frac{\partial \text{exp}}{\partial \text{acegenarray}} \right] \]

\[ \text{SMSD}[\text{exp}, \{v_1, v_2, \ldots\}, \text{index}] \quad \text{create a characteristic expression for an arbitrary element of the gradient} \quad \left[ \frac{\partial \text{exp}}{\partial v_1}, \frac{\partial \text{exp}}{\partial v_2}, \ldots \right] \quad \text{and return an index data object that represents characteristic element of the gradient with the index index} \]

\[ \text{SMSD}[\text{exp}, \text{acegenarray}, \text{index}] \quad \text{create a characteristic expression for an arbitrary element of the gradient} \quad \left[ \frac{\partial \text{exp}}{\partial \text{acegenarray}} \right] \quad \text{and return an index data object that represents characteristic element of the gradient with the index index} \]

\[ \text{SMSD}[\text{exp}_{\text{structure}}, \text{var}_{\text{structure}}] \]
differentiation of an arbitrary exp_structure with respect to an arbitrary var_structure.

The result has the same global structure as the exp_structure with each scalar exp replaced by the substructure

e.g. derivatives of second order tensors can be generated $D_{ijkl} = \frac{\partial f_{ij}}{\partial \alpha_k}$

Automatic differentiation procedures.

<table>
<thead>
<tr>
<th>option</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Constant&quot;-&gt;</td>
<td>{}</td>
<td>perform differentiation under assumption that formulas involved do not depend on given variables</td>
</tr>
<tr>
<td>{v1,v2,...}</td>
<td></td>
<td>{v1,v2,...} (see also Exceptions in Differentiation)</td>
</tr>
<tr>
<td>&quot;Constant&quot;-&gt;v</td>
<td></td>
<td>= &quot;Constant&quot;-&gt;{v}</td>
</tr>
<tr>
<td>&quot;Mode&quot;-&gt;AD_mode</td>
<td>&quot;Automatic&quot;</td>
<td>Method used to perform differentiation:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;Forward&quot; ⇒ forward mode of automatic differentiation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;Backward&quot; ⇒ backward mode of automatic differentiation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;Automatic&quot; ⇒ appropriate AD mode is selected automatically</td>
</tr>
<tr>
<td>&quot;Dependency&quot;-&gt;</td>
<td>{}</td>
<td>during differentiation assume that derivative of auxiliary variable v with respect to auxiliary variable z is $\frac{dv}{dz}$ (for the detailed syntax see SMSFreeze command and Exceptions in Differentiation chapter, note also that, contrary to the SMSFreeze command, in the case of SMSD command the independent variables have to specified explicitly!)</td>
</tr>
<tr>
<td>{,...}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;Symmetric&quot;-&gt;</td>
<td>False</td>
<td>see example below</td>
</tr>
<tr>
<td>truefalse</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;Ignore&quot;-&gt;crit</td>
<td>(False&amp;)</td>
<td>If differentiation is performed with respect to matrix then the elements of the matrix for which crit[e] yields False are ignored (NumberQ[exp] yields True). (see example &quot;Differentiation with respect to matrix&quot;)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;SaveLast&quot;-&gt;</td>
<td>True</td>
<td>True = the final results is optimized</td>
</tr>
<tr>
<td>truefalse</td>
<td></td>
<td>False = the final formula is left unoptimized</td>
</tr>
<tr>
<td>&quot;SaveAll&quot;-&gt;</td>
<td>True</td>
<td>True = all partial derivatives are optimized</td>
</tr>
<tr>
<td>truefalse</td>
<td></td>
<td>False = partial derivatives are left unoptimized</td>
</tr>
</tbody>
</table>

Options of the SMSD function.

The argument index is an integer type auxiliary variable, array is an auxiliary variable that represents an array data object (the SMSArray function returns an array data object, not an auxiliary variable), and arrayindex is an auxiliary variable that represents index data object (see Arrays).

Sometimes differentiation with respect to intermediate auxiliary variables can lead to incorrect results due to the interaction of automatic differentiation and Expression Optimization. In order to prevent this, all the basic independent variables have to have an unique signature. Functions such as SMSFreeze, SMSReal, and SMSFictive return an auxiliary variable with the unique signature.

**Differentiation: Mathematica syntax versus AceGen syntax**

The standard Mathematica syntax is compared here with the equivalent AceGen Syntax.

**Mathematica**

```mathematica
Clear [x, y, z, k];
f = x + 2 y + 3 z + 4 k
4 k + x + 2 y + 3 z
```

- Partial derivative: $\frac{\partial f}{\partial x}$
\[ D[f, x] \]
\[ 1 \]

- Gradient: \( \nabla_x = \frac{\partial f}{\partial x} \)

\[ D[f, \{\{x, y, z, k\}\}] \]
\[ \{1, 2, 3, 4\} \]

- Jacobian: \( J_{ij} = \frac{\partial f_i}{\partial x_j} \)

\[ D[(f, f^2), \{\{x, y\}\}] // \text{MatrixForm} \]
\[
\begin{pmatrix}
1 & 2 \\
2 & (4k + x + 2y + 3z) & 4 & (4k + x + 2y + 3z)
\end{pmatrix}
\]

- Derivatives of second order tensors: \( D_{ij,kl} = \frac{\partial f_{ij}}{\partial x_{kl}} \)

\[ D[\{(f, f^2), (f^3, f^4)\}, \{\{x, y\}, \{z, k\}\}] // \text{MatrixForm} \]
\[
\begin{pmatrix}
\begin{pmatrix} 1 & 2 \\ 2 & (4k + x + 2y + 3z) & 4 & (4k + x + 2y + 3z) \\
(4k + x + 2y + 3z) & 0 & 0 & 0
\end{pmatrix} & \begin{pmatrix} 2 & 4 & 4 \ \\
4 & 4 & 4 \\
4 & 4 & 4
\end{pmatrix} \\
9 & 12 & 16 & 20
\end{pmatrix}
\]

AceGen

<<AceGen``;
SMSInitialize["test", "Language" -> "C"];
SMSModule["test", Real[x$$, y$$, z$$, k$$]]; 
\{x, y, z, k\} \rightarrow \text{SMSReal}[\{x$$, y$$, z$$, k$$]\]; 
\( f = x + 2y + 3z + 4k; \)

- Partial derivative: \( \frac{\partial f}{\partial x} \)

\[ dx = \text{SMSD}[f, x] \]
\[ 1 \]

- Gradient: \( \nabla_x = \frac{\partial f}{\partial x} \)

Note that in \texttt{Mathematica} the vector of independent variables has an extra bracket. This is due to the legacy problems with the \texttt{Mathematica} syntax.

\[ \nabla_x = \text{SMSD}[f, \{x, y, z\}] \]
\[ \{1, 2, 3\} \]

- Jacobian: \( J_{ij} = \frac{\partial f_i}{\partial x_j} \)

\[ Jx = \text{SMSD}[\{(f, f^2), \{x, y\}\}] \]
\[ \{(1, 2), \{x$$, y$$\}\} \]

\[ \text{SMSRestore}[Jx, "Global"] // \text{MatrixForm} \]
\[
\begin{pmatrix}
1 & 2 \\
2 & (4x + 4y + 4z + 4k) & 4 & (4x + 4y + 4z + 4k)
\end{pmatrix}
\]

- Derivatives of second order tensors: \( D_{ij,kl} = \frac{\partial f_{ij}}{\partial x_{kl}} \)
\[ \text{Dx} = \text{SMSD}([([f, f^2]), ([f^3, f^4]), ([x, y], [z, k])]) \]

\[ \text{SMSRestore}[\text{Dx}, \text{"Global"}] \] // \text{MatrixForm} 

\[
\begin{pmatrix}
1 & 2 \\
3 & 4 \\
5 & 6 \\
9 & 12
\end{pmatrix}
= \begin{pmatrix}
2 & 4 \\
6 & 8 \\
4 & 8 \\
12 & 16
\end{pmatrix}
\]

See also \text{SMSD} for additional examples.

**Automatic Differentiation Examples**

**Example 1: Simple C subroutine**

Generation of the C subroutine which evaluates derivative of function \( z(x) \) with respect to \( x \).

\[ z(x) = 3x^2 + 2y + \log(y). \]

\[ y(x) = \sin(x^2). \]

\[
\begin{verbatim}
<< AceGen;
SMSInitialize["test", \text{"Language" -> } \text{"C"}];
SMSModule["test", \text{Real[x$$, r$$]}];
x = \text{SMSReal}[x$$];
y = \sin[x$$];
z = 3 \cdot x^2 + 2 \cdot y + \log[y];
\end{verbatim}
\]

Here the derivative of \( z \) with respect to \( x \) is calculated.

\[ \text{zx} = \text{SMSD}[z, x]; \]

\[ \text{SMSExport}[\text{zx}, r$$]; \]

\[ \text{SMSWrite}[]; \]

| File: test.c | Size: 819 |
| Methods No. | Formulae No. | Leafs |
| test         | 4            | 38    |
FilePrint["test.c"]

/******************************************************************************/
* AceGen    5.001 Windows (3 Jan 13)                               *
* Co. J. Korelc  2007   3 Jan 13 13:53:05                          *
/******************************************************************************/

User : USER
Notebook : AceGenTutorials.nb
Evaluation time : 0 s     Mode : Optimal
Number of formulae : 4       Method: Automatic
Subroutine : test size :38
Total size of Mathematica code : 38 subexpressions
Total size of C code : 219 bytes/

#include "sms.h"

/*******************
SUBROUTINE ***********/
void test(
    double v[5001], double (*x), double (*r))
{
    v[10]=Power((*x),2);
    v[9]=2e0+(*x);
    v[6]=v[9]*cos(v[10]);
    (*r)=3e0+v[9]+v[6]*(2e0+1e0/sin(v[10]));
}

Example 2: Differentiation of complex program structure

Generation of the C function file which evaluates derivative of function

\[ f(x) = 3z^2 \text{ with respect to } x, \text{ where } z \text{ is } \]

\[ z(x) = \begin{cases} 
\cos(x^2) & \text{if } x \geq 0 \\
\cos(x^2) & \text{if } x \leq 0
\end{cases} \]

and \( y = \sin(x^2) \).

<<AceGen``;
SMSInitialize["test", "Language" -> "C"];
SMSModule["test", Real[x$$, r$$]];
x = SMSReal[x$$];
z = SMSIf[x > 0
    , y = Sin[x$$];
    3 x$$ + 2 y + Log[y]
    , Cos[x$$]
    ];
f = SMSD[3 z$$, x$$];
SMSExport[f, r$$];
SMSWrite[];
FilePrint["test.c"]
Example 3: Differentiation with respect to symmetric matrix

The differentiation of a scalar value with respect to the matrix of differentiation variables can be nontrivial if the matrix has a special structure.

If the scalar value \( \exp(M) \) depends on a symmetric matrix of independent variables

\[
M = \begin{pmatrix}
M_{11} & M_{12} \\
M_{12} & M_{22}
\end{pmatrix}
\]

then we have two possibilities to make proper differentiation:

A) the original matrix \( M \) can be replaced by the new matrix of unique variables

\[
M' = \text{SMSFreeze}[M];
\]

\( \delta \exp(\text{SMSD}[\exp(M'), M']) \);

B) if the scalar value \( \exp \) is an isotropic function of \( M \) then the "Symmetric" -> True option also leads to proper derivative as follows

\[
\delta \exp = \text{SMSD}[\exp(M, M, "Symmetric" -> True)]
\]

Example:

Let's have matrix \( M = \begin{pmatrix} a & b \\ b & c \end{pmatrix} \) and functions \( f_{\text{iso}} = \det(M) \) and \( f_{\text{general}} = M_{11}^2 + 5M_{12} - \sin(M_{21}) - 3M_{12} \), thus
$$\frac{\partial f_{\text{iso}}}{\partial M} = \begin{bmatrix} a & -b \\ -b & c \end{bmatrix} \quad \text{and} \quad \frac{\partial f_{\text{general}}}{\partial M} = \begin{bmatrix} 2a & 5 \\ -\cos(b) & -3 \end{bmatrix}$$

The result of differentiation is incorrect under the assumption that $M$ is a symmetric matrix of independent variables.

\[
\text{SMSD}[f_{\text{iso}}, M] \quad \text{// MatrixForm}
\]

Some of the independent variables appear several times in a list of independent variables: \{\{a, b\}, \{b, c\}\}. The multiplicity of variables will be ignored. See Symmetric->True option.

See also: SMSD

With the "Symmetric"→True we obtain correct result for isotropic argument.

\[
\text{SMSD}[f_{\text{iso}}, M, "\text{Symmetric}\rightarrow\text{True}"] \quad \text{// MatrixForm}
\]

The argument can be also an arbitrary structure composed of isotropic functions.

\[
\text{SMSD}\{f_{\text{iso}}, \sin[f_{\text{iso}}]\}, M, "\text{Symmetric}\rightarrow\text{True}"] \quad \text{// MatrixForm}
\]

With the "Symmetric"→True option wrong result is obtained for a general argument.

\[
\text{SMSD}[f_{\text{general}}, M, "\text{Symmetric}\rightarrow\text{True}"] \quad \text{// MatrixForm}
\]

SMSFreeze creates unique variables for all components of matrix. Note that the result is less optimized that the one with "Symmetric"→True option, however it creates correct results regardless on the type of the argument.
Example 4: Differentiation with respect to sparse matrix

By default all differentiation variables have to be defined as auxiliary variables with unique random value. With the option "Ignore"->NumberQ the numbers are ignored and derivatives with respect to numbers are assumed to be 0.

\[
\text{SMSD}[\exp, \begin{pmatrix} v_{11} & v_{12} & \cdots \\ v_{21} & 0 & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}, \text{"Ignore"->NumberQ}] = \begin{pmatrix} \frac{\partial \exp}{\partial v_{11}} & \frac{\partial \exp}{\partial v_{12}} & \cdots \\ \frac{\partial \exp}{\partial v_{21}} & 0 & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}
\]

Without the "Ignore"->NumberQ option the AceGen reports an error.

Syntax error in differentiation. Independent variables have to be true variables.

Example 5: Differentiation with respect to intermediate variables

Generation of the C subroutine which evaluates derivative of function Sin(w) with respect to w where w is intermediate auxiliary variable defined as \( w = x^2 + 1 \).

\[
\frac{\partial \sin(w)}{\partial w} = 2x
\]

The intermediate auxiliary variable is not truly independent variable and as such does not possess unique signature. Differentiation is in this case not possible.
Characteristic Formulae

If the result would lead to large number of formulae, we can produce a characteristic formula. Characteristic formula is one general formula, that can be used for the evaluation of all other formulae. Characteristic formula can be produced by the use of AceGen functions that can work with the arrays and indices on a specific element of the array.

If \( N_{\text{df}} \) unknown parameters are used in our numerical procedure, then an explicit form of the gradient and the Hessian will have at least \( N_{\text{df}} + (N_{\text{df}})^2 \) terms. Thus, explicit code for all terms can be generated only if the number of unknowns is small. If the number of parameters of the problem is large, then characteristic expressions for arbitrary term of gradient or Hessian have to be derived. The first step is to present a set of parameters as a union of disjoint subsets. The subset of unknown parameters, denoted by \( a_i \), is defined by

\[
\begin{align*}
\cap_{i=1}^L a_i &= a \\
\forall a_i \cap a_j &= \emptyset, i \neq j.
\end{align*}
\]

Let \( f(a) \) be an arbitrary function, \( L \) the number of subsets of \( a \), and \( \frac{\partial f}{\partial a} \) the gradient of \( f \) with respect to \( a \).

\[
\frac{\partial f}{\partial a} = \begin{bmatrix}
\frac{\partial f}{\partial a_1} \\
\frac{\partial f}{\partial a_2} \\
\vdots \\
\frac{\partial f}{\partial a_L}
\end{bmatrix}
\]

Let \( a_i \) be an arbitrary element of the \( i \)-th subset. At the evaluation time of the program, the actual index of an arbitrary element \( a_i \) becomes known. Thus, \( a_{i,j} \) represents an element of the \( i \)-th subset with the index \( j \). Then we can calculate a characteristic formula for the gradient of \( f \) with respect to an arbitrary element of subset \( i \) as follows
\[ \frac{\partial f}{\partial a_{ij}} = \text{SMSD}[f, a_{ij}] \]

Let \( a_{kl} \) represents an element of the \( k \)-th subset with the index \( l \). Characteristic formula for the Hessian of \( f \) with respect to arbitrary element of subset \( k \) is then

\[ \frac{\partial^2 f}{\partial a_{ij} \partial a_{kl}} = \text{SMSD}\left[ \frac{\partial f}{\partial a_{ij}}, a_{kl} \right] \]

**Example 1: characteristic formulae - one subset**

Let us again consider the example presented at the beginning of the tutorial. A function which calculates gradient of function \( f = u^2 \), with respect to unknown parameters \( u_i \) is required.

\[ u = \sum_{i=1}^{3} u_i \]
\[ N_1 = \frac{x}{L}, \ N_2 = 1 - \frac{x}{L}, \ N_3 = \frac{x}{L} \left( 1 - \frac{x}{L} \right) \]

The code presented here is generated without the generation of characteristic formulae. This time all unknown parameters are grouped together in one vector. *AceGen* can then generate a characteristic formula for the arbitrary element of the gradient.

```acegen
<< AceGen;
SMSInitialize["test", "Language" -> "Fortran"]
SMSModule["Test", Real[u$$[3], x$$, L$$, g$$[3]]];
{x, L} \rightarrow \{SMSReal[x$$], SMSReal[L$$]\};
u = SMSReal[Table[u$$[i], (i, 3)]];
Ni = \left\{ \frac{x}{L}, 1 - \frac{x}{L}, \frac{x}{L} \left( 1 - \frac{x}{L} \right) \right\};
u = Ni.ui;
f = u^2;
SMSDo[i, 1, 3];
```

Here the derivative of \( f \) with respect to \( i \)-th element of the set of unknown parameters \( u_i \) is calculated.

\[ f_{ui} = \text{SMSD}[f, ui, i] \]
\[ \text{SMSExport}[f_{ui}, g$$[i]]; \]
\[ \text{SMSEndDo[]}; \]
\[ \text{SMSWrite[]}; \]

**Method**: Test 6 formulae, 95 sub-expressions

Example 2: characteristic formulae - two subsets

Write function which calculates gradient $\frac{\partial f}{\partial a_i}$ and the Hessian $\frac{\partial^2 f}{\partial a_i \partial a_j}$ of the function,

\[ f = f(u_1, v_1, u_2, v_2, u_3, v_3, u_4, v_4) = u^2 + v^2 + uv, \]

with respect to unknown parameters \( u_i \) and \( v_i \), where

\[ u = \sum_{i=1}^{4} N_i u_i \]
\[ v = \sum_{i=1}^{4} N_i v_i \]

and

\[ N = \{(1-X)(1-Y), (1+X)(1-Y), (1+X)(1+Y), (1-X)(1+Y)\}. \]

We make two subsets \( u_i \) and \( v_i \) of the set of independent variables \( a_i \).

\[ a_i = (u_1, v_1, u_2, v_2, u_3, v_3, u_4, v_4) \]
\[ u_i = \{u_1, u_2, u_3, u_4\}, \quad v_i = \{v_1, v_2, v_3, v_4\} \]
<< AceGen;
SMSInitialize["test", "Language" -> "C"]
SMSModule["Test", Real[u1$$[4], v1$$[4], X$$, Y$$, g$$[8], H$$[8, 8]]];
(X, Y) = {SMSReal[X$$], SMSReal[Y$$]};
u1 = SMSReal[Table[u1$$[i], {i, 4}]];  
v1 = SMSReal[Table[v1$$[i], {i, 4}]];  
Ni = {(1 - X) (1 - Y), (1 + X) (1 - Y), (1 + X) (1 + Y), (1 - X) (1 + Y)};
u = Ni.ui; v = Ni.vi;
f = u^2 + v^2 + uv;
SMSDo[
(*Here the characteristic formulae for the sub-vector of the gradient vector are created.*)
{g1i, g2i} = {SMSD[f, ui, i], SMSD[f, vi, i]};
(*Characteristic formulae have to be exported to
the correct places in a gradient vector.*)
SMSExport[{g1i, g2i}, {g$$(2 i - 1), g$$[2 i]}];
SMSDo[
(*Here the 2x2 characteristic sub-matrix of the Hessian is created.*)
H = {{SMSD[g1i, ui, j], SMSD[g1i, vi, j]},
     {SMSD[g2i, ui, j], SMSD[g2i, vi, j]}};
SMSExport[H, {H$$[2 i - 1, 2 j - 1], H$$[2 i - 1, 2 j]},
      {H$$[2 i, 2 j - 1], H$$[2 i, 2 j]}];
    , {j, 1, 4}
  ];
  , {i, 1, 4}
];
SMSWrite[];
FilePrint["test.c"]
There are basic situations that have to be considered:

1. There are situations when the formulae and the program structure alone are not sufficient to make proper derivative code. The "Dependency" option of the SMSFreeze, SMSGreal and SMSD function should be used instead whenever possible.

2. Exceptions in Differentiation

The SMSDefineDerivative function should be used cautiously since derivatives are defined permanently and globally. The "Dependency" option of the SMSFreeze, SMSGreal and SMSD function should be used instead whenever possible.

There are several situations when the formulae and the program structure alone are not sufficient to make proper derivative code. The basic situations that have to be considered are:

---

#include "sms.h"

double v[5025], double ul[4], double vl[4], double (*X), double (*Y),
        double g[8], double H[8][8]

int i22, i31;

v[16] = 1e0 - (*X);

v[14] = 1e0 - (*X);

v[12] = 1e0 - (*Y);


v[18] = v[16] + v[17];

v[5012] = v[11];

v[5013] = v[13];

v[5014] = v[15];

v[5015] = v[18];


v[26] = v[19] + 2e0 + v[20];

v[24] = 2e0 * v[19] - v[20];

for (i22 = 1; i22 <= 4; i22++) {
    v[28] = v[5011 + i22];
    g[1 - 2 + i22] = v[24] + v[28];
    g[1 - 2 + i22] = v[26] + v[28];

    for (i31 = 1; i31 <= 4; i31++) {
        v[38] = v[5011 + i31];
        v[37] = 2e0 * v[28] * v[38];
        v[39] = v[37] / 2e0;
        H[1 - 2 + i22] [1 - 2 + i31] = v[37];
        H[1 - 2 + i22] [1 - 2 + i31] = v[39];
        H[1 - 2 + i22] [1 - 2 + i31] = v[39];
        H[1 - 2 + i22] [1 - 2 + i31] = v[37];
    } /* end for */
} /* end for */

```
- **Type A**
  Basic case: The total derivatives of intermediate variables $b(a)$ with respect to independent variables $a$ are set to be equal to matrix $M$.

- **Type A**
  Basic case: The total derivatives of intermediate variables $b(a)$ with respect to independent variables $a$ are set to be equal to matrix $M$.

- **Type B**
  Special case of A: There exists explicit dependency between variables that has to be neglected for the differentiation.

- **Type C**
  Special case of A: There exists implicit dependency between variables (the dependency does not follow from the algorithm itself) that has to be considered for the differentiation.

- **Type D**
  Generalization of A: The total derivatives of intermediate variables $b(c)$ with respect to intermediate variables $c(a)$ are set to be equal to matrix $M$.

<table>
<thead>
<tr>
<th>Type</th>
<th>Local AD exception</th>
<th>Schematic AceGen input</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$\nabla f_A := \frac{\delta f(a,b(a))}{\delta a} \bigg</td>
<td>_{\frac{Db}{D_a} = -M}$</td>
</tr>
<tr>
<td>B</td>
<td>$\nabla f_B := \frac{\delta f(a,b(a))}{\delta a} \bigg</td>
<td>_{\frac{Db}{D_a} = 0}$</td>
</tr>
<tr>
<td>C</td>
<td>$\nabla f_C := \frac{\delta f(a,b)}{\delta a} \bigg</td>
<td>_{\frac{Db}{D_a} = -M}$</td>
</tr>
<tr>
<td>D</td>
<td>$\nabla f_D := \frac{\delta f(a,b(c(a)))}{\delta a} \bigg</td>
<td>_{\frac{Db}{D_c} = -M}$</td>
</tr>
</tbody>
</table>
It was shown in the section Theory of Automatic Differentiation that with a simple chain rule we obtain derivatives with respect to the arbitrary variables by following the structure of the program (forward or backward). However this is no longer true when variables depend implicitly on each other. This is the case for nonlinear coordinate mapping, collocation variables at the collocation points etc. These implicit dependencies cannot be detected without introducing additional knowledge into the system.

The local definition of AD exception is defined as an option given to the SMSD command while the global definition of AD exception is introduced by SMSFreeze command. With the SMSFreeze[exp, "Dependency"] the true dependencies of exp with respect to auxiliary variables are neglected and all partial derivatives are taken to be 0. With the SMSFreeze\[\exp, \text{Dependency} \rightarrow \{p_1, \frac{\partial \exp}{\partial p_1}, \ldots, \{p_n, \frac{\partial \exp}{\partial p_n}\}\}\] the true dependencies of the exp are ignored and it is assumed that exp depends on auxiliary variables p_1, ..., p_n. Partial derivatives of exp with respect to auxiliary variables p_1, ..., p_n are then taken to be \(\frac{\partial \exp}{\partial p_1}, \ldots, \frac{\partial \exp}{\partial p_n}\).

**SMSDefineDerivative**

**SMSDefineDerivative[v,z,exp]**

define the derivative of auxiliary variable v with respect to auxiliary variable z to be exp

\(\frac{\partial v}{\partial z} := \text{exp}\)

**SMSDefineDerivative[v,\{z_1,z_2,...,z_N\},D]**

define gradient of auxiliary variable v with respect to variables \{z_1,z_2,...,z_N\} to be vector \(D:=\{\frac{\partial v}{\partial z_1}, \ldots, \frac{\partial v}{\partial z_N}\}\) and set \(\frac{\partial z_j}{\partial z_i} = \delta_{ji}\)

**SMSDefineDerivative[\{v_1,v_2,...,v_M\},z,\{d_1,d_2,...,d_M\}]**

define the derivatives of the auxiliary variables \{v_1,v_2,...,v_M\} with respect to variable z to be \(\frac{\partial v_i}{\partial z} = d_i\)

**SMSDefineDerivative[\{v_1,v_2,...,v_M\},\{z_1,z_2,...,z_N\},J]**

define a Jacobian matrix of the transformation from \{v_1,v_2,...,v_M\} to \{z_1,z_2,...,z_N\} to be matrix \(J:=\{\frac{\partial v_i}{\partial z_1}, \ldots, \frac{\partial v_i}{\partial z_N}\}\) and set \(\frac{\partial z_j}{\partial z_i} = \delta_{ji}\)

**Exception Type A: Generic Example**

Let’s take a function \(f=\sin(w)\) where \(w\) is intermediate auxiliary variable defined as \(w = ax^2 + 1\) and evaluate the following derivative.
\[ \frac{\partial f}{\partial x} \bigg|_{x=5} = 5 \cos(w) \]

- Local definition of AD exception

\[
\langle \langle \text{AceGen} \rangle \rangle; \\
\text{SMSInitialize}[\text{"test"}]; \\
\text{SMSModule}[\text{"test"}, \text{Real}[x$$, a$$]]; \\
x \mapsto \text{SMSReal}[x$$]; a \mapsto \text{SMSReal}[a$$]; \\
w \mapsto \text{SMSFreeze}[a x^2 + 1]; \\
f \mapsto \sin[w]; \\
\text{SMSRestore}[\text{SMSD}[f, x, \text{"Dependency"} \rightarrow \{w, x, 5\}], \text{"Global"}]
\]

\[ 5 \cos[w] \]

- Global definition of AD exception with SMSFreeze

\[
\langle \langle \text{AceGen} \rangle \rangle; \\
\text{SMSInitialize}[\text{"test"}]; \\
\text{SMSModule}[\text{"test"}, \text{Real}[x$$, a$$]]; \\
x \mapsto \text{SMSReal}[x$$]; a \mapsto \text{SMSReal}[a$$]; \\
w \mapsto \text{SMSFreeze}[a x^2 + 1, \text{"Dependency"} \rightarrow \{x, 5\}]; \\
f \mapsto \sin[w]; \\
\text{SMSRestore}[\text{SMSD}[f, x], \text{"Global"}]
\]

\[ 5 \cos[w] \]

Note that the "Dependency" option of the SMSFreeze command hides all other dependencies thus the derivative of \( f \) with respect to \( a \) is 0.

\[ \text{SMSD}[f, a] \]

0

**Exception Type C: Implicit dependencies**

The generation of the subroutine that calculates displacement gradient \( Dg \) defined by

\[
\begin{align*}
\xi = \{\xi, \eta, \zeta\} & \quad \text{reference coordinates} \\
X (\xi) = \sum_k N_k(\xi_k) X_k & \quad \text{actual coordinates} \\
u (\xi) = \sum_k N_k(\xi_k) u_k & \quad \text{displacements} \\
D & \quad \text{displacement gradient}
\end{align*}
\]

where \( N_k = 1/8 \left( 1 + \xi \xi_k \right) \left( 1 + \eta \eta_k \right) \left( 1 + \zeta \zeta_k \right) \) is the shape function for \( k \)-th node where \( (\xi_k, \eta_k, \zeta_k) \) are the coordinates of the \( k \)-th node.
<<AceGen';
SMSInitialize["test", "Language" -> "Fortran"];
SMSModule["Test", Real[X$$[8, 3], u$$[8, 3], ksi$$, eta$$, ceta$$];
Ξ = {ξ, η, ζ} ⊢ Table[SMSSReal[es$$["IntPoints", i, Ig]], (i, 3)];
XI ⊢ Table[SMSSReal[nd$$[i, "X", j]], (i, 8), (j, 3)];
Ξn = {{-1, -1, -1}, {1, -1, -1}, {1, 1, 1}, {-1, 1, -1},
     (-1, -1, 1), (1, 1, 1), (-1, 1, 1), (1, -1, 1)};
NI ⊢ Table[1/8 (1 + η Ξn[i, 1]) (1 + η Ξn[i, 2]) (1 + η Ξn[i, 3]), (i, 1, 8)];
-
Coordinates X = {Xg, Yg, Zg} are the basic independent variables. To prevent wrong simplifications, we have to define unique signatures for the definition of X.
X ⊢ SMSFreeze[NI.XI];
-
Here the Jacobian matrix of nonlinear coordinate transformation is calculated.
Jg ⊢ SMSD[X, Ξ]{{Jg[1, 1], Jg[1, 2], Jg[1, 3]}, {Jg[2, 1], Jg[2, 2], Jg[2, 3]}, {Jg[3, 1], Jg[3, 2], Jg[3, 3]}];
-
Interpolation of displacements.
uI ⊢ SMSReal[Table[nd$$[i, "at", j]], (i, 8), (j, 3)];
u ⊢ NI.uI;
-
Simple use of SMSD leads to wrong results.
SMSD[u, X]
{(0, 0, 0), (0, 0, 0), (0, 0, 0)}
-
The implicit dependency of Ξ on X is now taken into account when the derivation of u is made with respect to X.
Local definition of type C AD exception.

\[
\nabla f_C := \frac{\partial f(a,b)}{\partial a} \quad \text{if } \frac{\partial f(a,b)}{\partial a} = M
\]
\[
\nabla f_C := \frac{\partial f(a,b)}{\partial a} \quad \text{if } a = \text{SMSSReal}[a\$$]
\]
\[
\nabla f_C := \frac{\partial f(a,b)}{\partial a} \quad \text{if } b = \text{SMSSReal}[b\$$]
\]
\[
\nabla f_C := \frac{\partial f(a,b)}{\partial a} \quad \text{if } \text{SMSInverse}[Jg]]]
\]

Exception Type D: Alternative definition of partial derivatives

The generation of the FORTRAN subroutine calculates the derivative of function \( f = \frac{\sin(2a^2)}{a} \) where \( a = \cos(x) \) with respect to \( x \). Due to the numerical problems arising when \( \alpha \to 0 \) we have to consider exceptions in the evaluation of the function as well as in the evaluation of its derivatives as follows:

\[
\begin{align*}
 f := \begin{cases} 
 \frac{\sin(2a^2)}{a} & \text{if } \alpha \neq 0 \\
 \lim_{a \to 0} \frac{\sin(2a^2)}{a} & \text{if } \alpha = 0
\end{cases} \\
 \frac{\partial f}{\partial \alpha} := \begin{cases} 
 \frac{\partial}{\partial \alpha} \left( \frac{\sin(2a^2)}{a} \right) & \text{if } \alpha \neq 0 \\
 \lim_{a \to 0} \frac{\partial}{\partial \alpha} \left( \frac{\sin(2a^2)}{a} \right) & \text{if } \alpha = 0
\end{cases}
\end{align*}
\]
<<AceGen;
SMSInitialize["test", "Language" -> "Fortran"];
SMSModule["Test", Real[x$$, f$$, dfdx$$]];
x = SMSReal[x$$];
α ⊢ SMSFreeze[Cos[x]];
f = SMSIf[SMSAbs[α] > 10^-10
  , Sin[2 a^2]/α
  , SMSFreeze[Limit[Sin[2 a^2]/α, α -> 0],
    "Dependency" -> {{α, Limit[D[Sin[2 a^2]/α, α] // Evaluate, α -> 0]]}]]
];
dfdx = SMSD[f, x];
SMSExport[dfdx, dfdx$$];
SMSWrite[];

File:  test.f  Size:  1055
Methods  No. Formulae  No. Leafs
Test  7  54

FilePrint["test.f"]

!***********************************************************************
!                     S U B R O U T I N E                             *
!***********************************************************************

SUBROUTINE Test(v,x,f,dfdx)
IMPLICIT NONE
include 'sms.h'
LOGICAL b3
DOUBLE PRECISION v(5001),x,f,dfdx
v(11)=dcos(x)
v(10)=-dsin(x)
v(2)=v(11)
IF (dabs(v(2)).gt.0.1d-9) THEN
  v(6)=2d0*(v(11)+v(11))
v(8)=v(10)*(4d0+dcos(v(6))-dsin(v(6))/v(11)**2)
ELSE
  v(8)=2d0*v(10)
ENDIF
dfdx=v(8)
END

Limitations: Incorrect structure of the program

Differentiation cannot start inside the "If" construct if the variables involved have multiple instances defined on a separate branches of the same "if" construct. The limitation is due to the interaction of the simultaneous simplification procedure and the automatic differentiation procedure.
The first instance of variable \( f \) can not be evaluated at the same time as the second instance of variable \( f \). Thus, only the derivative code of the second expression have to be constructed. However, if the construct appears inside the loop, then some indirect dependencies can appear and both branches have to be considered for differentiation. The problem is that AceGen can not detect this possibility at the point of construction of the derivative code. There are several possibilities how to resolve this problem.

With the introduction of an additional auxiliary variable we force the construction of the derivative code only for the second instance of \( f \).

If the differentiation is placed outside the "If" construct, both instances of \( f \) are considered for the differentiation.

If \( f \) does not appear outside the "If" construct, then \( f \) should be defined as a single-valued variable (\( f \ni \)) and not as multi-valued variable (\( f \ni \)). In this case, there are no dependencies between the first and the second appearance of \( f \). However in this case \( f \) can not be used outside the "If" construct. First definition of \( f \) is overwritten by the second definition of \( f \).
Verification and Debugging

Contents

- General Verification Procedures
- Printing to Output Devices
  - SMSPrint
  - Example 1: printing from Mathematica code
  - Example 2: printing out to all output devices - C language
  - Example 3: printing out to all output devices - Fortran language
  - Example 4: printing from MathLink code
  - Example 5: printing out from numerical environment - AceFEM - MDriver
- SMSPrintMessage
- Run Time Debugging
  - Example of run time debugging
  - SMSSetBreak
  - SMSLoadSession
  - SMSClearBreak
  - SMSActivateBreak

General Verification Procedures

We can verify the correctness of the generated code directly in Mathematica. To do this, we need to rerun the problem and to generate the code in a script language of Mathematica. The SMSSetBreak function inserts a break point into the generated code where the program stops and enters interactive debugger (see also User Interface).

```
<< AceGen`
SMSInitialize["test", "Language" -> "Mathematica", "Mode" -> "Debug"];
SMSModule["Test", Real[u$$[3], x$$, L$$, g$$[3]]];
{x, L} ∈ {SMSReal[x$$], SMSReal[L$$]};
u i ∈ Table[SMSReal[u$$[i]], {i, 3}];
Ni = {x, 1 - x, x/L - (1 - x/L)};
u = Ni.ui;
f = u^2;
g = SMSD[f, u];
SMSEXport[g, g$$];
SMSWrite[];
```
We have several possibilities how to explore the derived formulae and generated code and how to verify the correctness of the model and of the generated code (see also User Interface).

The first possibility is to explore the generated formulae interactively with Mathematica in order to see whether their structure is logical.

```mathematica
In[136]:= u

Out[1]=
```

In the case of more complex code, the code profile can be explored where the structure of the program is displayed together with the links to all generated formulae (see also User Interface).

The second possibility is to make some numerical tests and see whether the numerical results are logical.

- This reads definition of the automatically generated “Test” function from the test.m file.

```mathematica
(Dialog) In[409]:=<"test.m"
```

- Here the numerical values of the input parameters are defined. The context of the symbols used in the definition of the subroutine is global as well as the context of the input parameters. Consequently, the new definition would override the old ones. Thus the names of the arguments cannot be the same as the symbols used in the definition of the subroutine.

```mathematica
(Dialog) In[410]:= xv = \[Pi]; Lv = 10.; uv = \{0., 1., 7.\}; gv = \{Null, Null, Null\};
```

- Here the generated code is used to calculate gradient for the numerical test example.

```mathematica
(Dialog) In[411]:= Test[uv, xv, Lv, gv]
```

- Here the numerical results are displayed.

```mathematica
(Dialog) In[412]:= gv

(Dialog) Out[412]= \{1.37858, 3.00958, 0.945489\}
```

Partial evaluation, where part of expressions is numerically evaluated and part is left in a symbolic form, can also provide useful information.

- Here the numerical values of u and x input parameters are defined, while L is left in a symbolic form.

```mathematica
(Dialog) In[413]:= xv = \[Pi] \[LeftDivision] N; Lv = .; uv = \{0., 1., 7.\}; gv = \{Null, Null, Null\};
```

- Here the generated code is used to calculate gradient for the given values of input parameters.

```mathematica
(Dialog) In[414]:= Test[uv, xv, Lv, gv]
```

- Here the partially evaluated gradient is displayed.
The third possibility is to compare the numerical results obtained by AceGen with the results obtained directly by Mathematica.

- Here the gradient is calculated directly by Mathematica with essentially the same procedure as before. AceGen functions are removed and replaced with the equivalent functions in Mathematica.

- Here the numerical results are calculated and displayed for the same numerical example as before. We can see that we get the same results.

- The last possibility is to look at the generated code directly.

Due to the option "Mode" -> "Debug" AceGen automatically generates comments that describe the actual meaning of the generated formulae. The code is also less optimized and it can be a bit more easily understood and explored.
Several modifications of the above procedures are possible.

**Printing to Output Devices**

**SMSPrint**

```
SMSPrint[expr1, expr2, ..., options]
```

create a source code sequence that prints out all the expressions expr, accordingly to the given options

```
SMSPrintMessage[expr1, expr2, ...]
```
The SMSPrint function is active only in "Debug" and "Prototype" mode while the SMSPrintMessage function always creates source code and prints out to standard output device (see SMSPrintMessage).

Expression expr, can be a string constant or an arbitrary AceGen expression. If the chosen language is Mathematica language or Fortran, then the expression can be of integer, real or string type.

**Printing from finite elements might be problematic if the program is parallelized!! Please turn off parallelization with SMTInputData["Threads" -> 1].**

The following restrictions exist for the C language:

⇒ the integer type expression is allowed, but it will be cased into the real type expression;

⇒ the string type constant is allowed and should be of the form "text";

⇒ the string type expression is not allowed and will result in compiler error.

<table>
<thead>
<tr>
<th>Language or Environment</th>
<th>default output device (&quot;Console&quot;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Mathematica&quot;</td>
<td>current notebook</td>
</tr>
<tr>
<td>&quot;C&quot;</td>
<td>console window (printf (… ))</td>
</tr>
<tr>
<td>&quot;Fortran&quot;</td>
<td>console window (write(<em>,</em>) … )</td>
</tr>
<tr>
<td>&quot;Matlab&quot;</td>
<td>matlab window (disp (… ) )</td>
</tr>
<tr>
<td>&quot;AceFEM&quot;</td>
<td>MDriver ⇒ current notebook</td>
</tr>
<tr>
<td></td>
<td>CDriver ⇒ current notebook</td>
</tr>
<tr>
<td></td>
<td>CDriver console ⇒ console window (SMTInputData[&quot;Console&quot; -&gt; True])</td>
</tr>
</tbody>
</table>

Additional AceFEM specific restrictions on printout
In order to prevent excessive printing to notebook or file AceFEM applies two additional conditions that limits the number of printouts.

Printing is executed accordingly to the value of the SMTIData["DebugElement"] environment variable:

- \texttt{SMTIData["DebugElement",-1]} ⇒ printouts are active for all elements (default value)
- \texttt{SMTIData["DebugElement",0]} ⇒ printouts are disabled
- \texttt{SMTIData["DebugElement",i]} ⇒ printout is active for \textit{i}-th element

Printing is suspended if the actual number of printouts exceeds the maximum number allowed as defined by SMTIData["MaxMessages"] environment variable. Warning message is produced when the number of printouts reaches SMTIData["MaxMessages"].

Printing messages to console or terminal window

- **Windows**
  On Windows application can be started in a separate window with the option "Console"=>True of the \texttt{SMSInstallMathLink} command (e.g. \texttt{SMSInstallMathLink["Console"=>True]}). Additional messages are then printed to command window as shown below.

- **Mac OS**
  In order to direct printouts to terminal window on Mac \textit{Mathematica} has to be started from the terminal as follows:
  - open Terminal (look under Applications-Utilities for Terminal.app),
  - start \textit{Mathematica} form Terminal (e.g. /Applications/Mathematica.app/Contents/MacOS/Mathematica),
  - after \texttt{SMTAnalysis[]} set SMTIData["ConsoleWindow",1]
  - messages will now be printed to terminal window.

- **Linux**
  In order to direct printouts to terminal window on Linux \textit{Mathematica} has to be started from the terminal as follows:
  - open Terminal (e.g. search for Terminal),
  - start \textit{Mathematica} form Terminal (e.g. Mathematica &),
  - after \texttt{SMTAnalysis[]} set SMTIData["ConsoleWindow",1]
  - messages will now be printed to terminal window.

\textbf{Example 1: printing from Mathematica code}

\begin{verbatim}
In[1]:= << AceGen
SMSInitialize["test", "Language" -> "Mathematica", "Mode" -> "Prototype"];
SMSModule["test", Real[x$$, r$$], "Input" -> {x$$}, "Output" -> {r$$}];
SMSPrint["\pi =", \texttt{\pi}];
SMSExport[\pi, r$$];
SMSWrite[];
\end{verbatim}
Consistency check - expressions

File: test.m  Size: 694  Time: 1

<table>
<thead>
<tr>
<th>Method</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>No.Formulae</td>
<td>2</td>
</tr>
<tr>
<td>No.Leafs</td>
<td>9</td>
</tr>
</tbody>
</table>

In[7]= Get["test.m"]

In[8]= FilePrint["test.m"]

(*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-*
  + AceGen   6.922 Windows (21 Feb 19)  +
  + Co. J. Korelc  2013  21 Feb 19 21:25:09 +
  ******************************************************
User : USER
Notebook : AceGenTutorials
Evaluation time : 1 s  Mode : Prototype
Number of formulae : 2  Method: Automatic
Module : test size: 9
Total size of Mathematica code : 9 subexpressions *)

(*-----------------------------------------------------------------------
M O D U L E
-----------------------------------------------------------------------*)

SetAttributes[test,HoldAll];
test[x$$_,r$$_] := Module[{},
Print["pi="," \",Pi];
r$$=Pi;
];

In[8]= r = 0.;
test[1., r]

Pi = π

Messages are printed to "console window" and to "test.out" file. Printing to notebook currently not supported!

Time = 1.3681e+009
E = 2.71828
Example 2: printing out to all output devices - C language

In[1]:= << AceGen;
SMSInitialize["test", "Language" -> "C", "Mode" -> "Prototype"]
SMSModule["test", Real[x$]];

(* print to standard console *)
SMSPrint["pi", π];

(* print to file test.out *)
SMSPrint["time", SMSTime[], "Output" -> {"File", "test.out"}];

(* print to file test.out only when x>0 *)
SMSPrint["e", E, "Output" -> {"File", "test.out"}, "Condition" -> SMSReal[x$] > 0];

SMSWrite[];

Out[1]= True

[1] Consistency check - expressions

<table>
<thead>
<tr>
<th>Method</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>No.Formulae</td>
<td>4</td>
</tr>
<tr>
<td>No.Leafs</td>
<td>11</td>
</tr>
</tbody>
</table>

In[8]:= FilePrint["test.c"]

/***** include "sms.h" ******/

/* AceGen  6.922 Windows (21 Feb 19)  *
  + Co. J. Korelc  2013           21 Feb 19 21:47:57 +
**********************************************************
User     : USER
Notebook : AceGenTutorials
Evaluation time                 : 1 s     Mode  : Prototype
Number of formulae              : 4       Method: Automatic
Subroutine                      : test size: 11
Total size of Mathematica code : 11 subexpressions
Total size of C code            : 455 bytes */

#include "sms.h"

void test(double v[114], double (*x))
{
  FILE *SMSFile;
  printf("%
  %s %g ", "pi-", (double)0.3141592653589793e1); v[2]=Time();
  SMSFile=fopen("test.out","a");if(SMSFile!=NULL){
    fprintf(SMSFile,"%
  %s %g ", "time-", (double)v[2]);
    fclose(SMSFile);};
  if((*x)>0e0){
    SMSFile=fopen("test.out","a");if(SMSFile!=NULL){
      fprintf(SMSFile,"%
  %s %g ", "e-", (double)0.2718281828459045e1);
      fclose(SMSFile);}
  };
};
Example 3: printing out to all output devices - Fortran language

In[9]:= << AceGen;
SMSInitialize["test", "Language" -> "Fortran", "Mode" -> "Prototype"];
SMSModule["test", Real[x$$]]; (*print to standard console*)
SMSPrint["pi", π]; (*print to file test.out*)
SMSPrint["time", SMSTime[], "Output" -> {"File", "test.out"}]; (*print to Fortran I/O unit number 4 only when x>0*)
SMSPrint["e", E, "Output" -> {"File", 4}, "Condition" -> SMSReal[x$$] > 0];
SMSWrite[];

[0] Consistency check - expressions

File: test.f  Size: 1053  Time: 0

<table>
<thead>
<tr>
<th>Method</th>
<th>No.Formulae</th>
<th>No.Leafs</th>
</tr>
</thead>
<tbody>
<tr>
<td>test</td>
<td>6</td>
<td>15</td>
</tr>
</tbody>
</table>

In[16]:= FilePrint["test.f"]

!************************************************************
! User     : USER
! Notebook : AceGenTutorials
! Evaluation time : 0 s     Mode  : Prototype
! Number of formulae : 6 Method: Automatic
! Subroutine : test size: 15
! Total size of Mathematica code : 15 subexpressions
! Total size of Fortran code : 450 bytes

SUBROUTINE test(v,x)
IMPLICIT NONE
include 'sms.h'
DOUBLE PRECISION v(116),x
write(*,'(a,x,g11.5)')"pi",0.3141592653589793d1
v(2)=Time()
OPEN(UNIT=10,FILE="test.out",STATUS="UNKNOWN")
write(10,'(a,x,g11.5)')"time",v(2)
CLOSE(10)
IF(x.gt.0d0) THEN
write(4,'(a,x,g11.5)')"e",0.2718281828459045d1
ENDIF
END
Example 4: printing from MathLink code

In[17]:= << AceGen; SMSInitialize["test", "Environment" -> "MathLink", "Mode" -> "Prototype"]; SMSModule["test", Real[x$$, r$$], "Input" -> {x$$}, "Output" -> {r$$}]; (*print to standard console*) SMSPrint["pi = ", \[Pi]]; (*print to file test.out*) SMSPrint["time = ", SMSTime[], "Output" -> {"File", "test.out"}]; (*print to file test.out only when x > 0 *) SMSExport["e = ", E, "Output" -> {"File", "test.out"}, "Condition" -> SMSReal[x$$] > 0]; SMSWrite[];

[0] Consistency check - expressions

<table>
<thead>
<tr>
<th>File: test.c</th>
<th>Size: 1797</th>
<th>Time: 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>test</td>
<td></td>
</tr>
<tr>
<td>No.Formulae</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>No.Leafs</td>
<td>18</td>
<td></td>
</tr>
</tbody>
</table>

In[25]:= SMSInstallMathLink["Console" -> True]

Out[25]= (SMSSetLinkOption["test", {i _ Integer, j _ Integer}], SMSLinkNoEvaluations["test"], test[x_/?(Head[#1] == Real || Head[#1] == Integer &)]}


- Messages are printed to "console window" and to "test.out" file. Printing to notebook is currently not supported!
Example 5: printing out from numerical environment - AceFEM-MDriver

In[27]:= << AceGen;
SMSInitialize["test", "Environment" -> "AceFEM-MDriver", "Mode" -> "Prototype"];
SMSTemplate["SMSTopology" -> "T1"];
SMSStandardModule["Tangent and residual"];
SMSPrint["pi = ", \[Pi]]; SMSPrint["load = ", rdata$$["Multiplier"]];
SMSWrite[];

[0] Consistency check - expressions

File: test.m  Size: 2244  Time: 0

Method
No.Formulae  SMT`SKR
No.Leafs  2

In[34]:= FilePrint["test.m"]

(* *******************************************************
* AceGen 6.922 Windows (21 Feb 19)               *
* Co. J. Korelc 2013                            *
* *******************************************************
User: USER
Notebook: AceGenTutorials
Evaluation time: 0 s
Mode: Prototype
Number of formulae: 2
Method: Automatic
Module: SMT`SKR size: 4
Total size of Mathematica code: 4 subexpressions
*)

SMT`SetElSpec["test",idata$$[ic_,gd_]] := Block[{q1,q2,q3,q4},
q4 = If[ic == -1, 12, ic];
q3 = SMCMultiIntegration[q4];
q1 = ("test",
"SKR" -> SMT`SKR, _ -> Null),
{"SpecIndex",2,6,0,3,0,8,
q4,"NoTimeStorage","NoElementData",q3[[1]],0,0,1,
q3[[3]],q3[[4]],q3[[5]],0,0,0,
0,12,0,"NoAdditionalData",
0,0,0,Null,0,
1,0,0,0,112}
,"T1",
SMCToMMAString[]];

 AceGenPrint.nb  127
\[ \Phi = \begin{cases} \alpha xy + x^2 = 0 \\ a - xy^2 = 0 \end{cases} \]

where \( x \) and \( y \) are unknowns and \( \alpha \) is the parameter using the standard Newton-Raphson iterative procedure. The SMSSetBreak function inserts the break points with the identifications "X" and "A" into the generated code.
In[35]:= <<AceGen`; 
SMSInitialize["test", "Language" -> "Mathematica", "Mode" -> "Debug"]; 
SMSModule["test", Real[x $$, y $$, a $$, to1 $$], Integer[nmax $$]]; 
(x0, y0, a, e) = SMSReal[\{x $$, y $$, a $$, to1 $$\}]; 
nmax = SMSInteger[nmax $$]; 
(x, y) = \{x0, y0\}; 
SMSDo[
Φ ⊨ \{a x y + x^3, a - xy^2\}; 
Kt = SMSDo[\{x, y\}]; 
(Δx, Δy) = SMSLinearSolve[Kt, -Φ]; 
(x, y) = \{(x, y) + {Δx, Δy}\}; 
SMSSetBreak["A", "Active" -> False]; 
SMSIf[SMSqrt[{Δx, Δy}].{Δx, Δy} < e
, SMSExport[{x, y}, \{x $$, y $$\}]; 
SMSBreak[];
]; 
SMSIf[i = nmax
, SMSPrintMessage["no convergion"];
SMSReturn[];
]; 
SMSSetBreak["X"]; 
, {i, 1, nmax, 1, \{x, y\}}
]; 
SMSWrite[];

time=0 variable= 0 \{\}
[0] Consistency check - global
[0] Consistency check - expressions
Events: 0
[0] Generate source code :
[0] Final formatting

<table>
<thead>
<tr>
<th>File: test.m</th>
<th>Size: 2560</th>
<th>Time: 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>No.Formulae</td>
<td>35</td>
</tr>
<tr>
<td>No.Leafs</td>
<td>246</td>
<td></td>
</tr>
</tbody>
</table>

Here the program is loaded and the generated subroutine is called.

In[48]:= <<AceGen`; 
<<"test.m"; 
SMSLoadSession["test"]; 
x = 1.9; y = -1.2; 
test[x, y, 3., 0.0001, 10]

At the break point the structure of the program is displayed together with the links to all generated formulae and the actual values of the auxiliary variables.
The program stops and enters interactive debugger whenever selective `SMSEvaluateBreakPoint` function is executed. The function also initiates special dialog mode supported by the Mathematica (see also Dialog). The "dialog" is terminated by button. Break points can be switched on () and off () by pressing the button at the position of the break point. The break points are automatically generated at the end of if.. else.. endif and do.. enddo structures additional to the user defined break points. The current break point is displayed with the sign.

Menu legend:

- Refresh ⇒ refresh the contents of the debug window
- ○ ⇒ disable all breaks points
- ● ⇒ enable break point at the beginning of the subroutine
- ❯❯ ⇒ continue to the next break point

- Here the break point "X" is inactivated and the break point "A" is activated. The break point "A" is given a pure function that is executed whenever the break point is called. Note that the `SMSLoadSession` also restores all definitions of the symbols that have been assigned value during the AceGen session (e.g. the definition of the \( K_t \) variable in the current example).

```
In[48]:= << AceGen`; << "test.m";
SMSLoadSession["test"]; SMSClearBreak["X"]; SMSActivateBreak["A", Print["K = ", Kt // MatrixForm] &];
x = 1.9; y = -1.2;
test[x, y, 3., 0.0001, 10]
K =  
7.23  5.7 
-1.44  4.56

K =  
7.48513  5.80332
-1.55506  4.82457

K =  
7.4744  5.79955
-1.55185  4.81646
```
**SMSSetBreak**

The code profile window is also used for the run-time debugging. The break points can be inserted into the source code by the SMSSetBreak command.

```plaintext
SMSSetBreak[breakID]
insert break point call into the source code with the string breakID as the break identification
```

<table>
<thead>
<tr>
<th>option</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Active&quot;</td>
<td>True</td>
<td>break point is by default active</td>
</tr>
<tr>
<td>&quot;Optimal&quot;</td>
<td>False</td>
<td>by default the break point is included into source code only in &quot;Debug&quot; mode. With the option &quot;Optimal&quot; the break point is always generated.</td>
</tr>
</tbody>
</table>

Options of the SMSSetBreak function.

Break points are inserted only if the code is generated with the "Mode"\text{→}"Debug" option. In "Debug" mode the system also automatically generates file with the name "sessionname.dbg" where all the information necessary for the run-time debugging is stored. The number of break points is not limited. All the user defined break points are by default active. With the option "Active"\text{→}False the break point becomes initially inactive. The break points are also automatically generated at the end of If.. else..endif and Do..endo statements additionally to the user defined break points. All automatically defined break points are by default inactive. Using the break points is also one of the ways how the automatically generated code can be debugged.

The data has to be restored from the "sessionname.dbg" file by the SMSLoadSession command before the generated functions are executed.

**SMSLoadSession**

```plaintext
SMSLoadSession[name]
reload the data and definitions associated with the AceGen session with the session name name and open profile window
```

With an additional commands SMSClearBreak and SMSActivateBreak the breaks points can be activated and deactivated at the run time.

**SMSClearBreak**

```plaintext
SMSClearBreak[breakID]
disable break point with the break identification breakID
SMSClearBreak["Default"]
set all options to default values
SMSClearBreak[
disable all break points
```

**SMSActivateBreak**

```plaintext
SMSActivateBreak[breakID_String, opt]
activate break point with the break identification breakID and options opt
SMSActivateBreak[im_Integer, opt]
activate the automatically generated break point at the beginning of the im-th module(subroutine)
```
SMSActivateBreak[]≡ SMSActivateBreak[1]

<table>
<thead>
<tr>
<th>option</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Interactive&quot;</td>
<td>True</td>
<td>initiates dialog (see also Dialog)</td>
</tr>
<tr>
<td>&quot;Window&quot;</td>
<td>True</td>
<td>open new window for debugging</td>
</tr>
<tr>
<td>&quot;Function&quot;</td>
<td>None</td>
<td>execute pure user defined function at the break point</td>
</tr>
</tbody>
</table>

Options of the SMSActivateBreak function.

The program can be stopped also when there are no user defined break points by activating the automatically generated break point at the beginning of the chosen module with the SMSActivateBreak[module_index] command.

If the debugging is used in combination with the finite element environment AceFEM, the element for which the break point is activated has to be specified first (SMTIData["DebugElement",elementnumber]).
General Utility Functions

Contents
- Standard Functions With Unique Signature
- Standard Functions Without Unique Signature
- System Utility Functions
- Utility Functions for Numerical Environments
- Compressing Structures

Standard Functions With Unique Signature

\[
\begin{align*}
\text{SMSAbs}[\text{exp}] & \quad \text{absolute value of } \text{exp} \text{ (real type function)} \\
\text{SMSSign}[\text{exp}] & \quad -1, 0 \text{ or } 1 \text{ depending on whether } \text{exp} \text{ is negative, zero, or positive (real type function)} \\
\text{SMSKroneckerDelta}[i, j] & \quad 1 \text{ or } 0 \text{ depending on whether } i \text{ is equal to } j \text{ or not (real type function)} \\
\text{SMSSum}[\text{exp1}, \text{exp2}] & \quad = \text{Min}[\text{exp1}, \text{exp2}] \text{ (the type is the same as the type of arguments)} \\
\text{SMSMax}[\text{exp1}, \text{exp2}] & \quad = \text{Max}[\text{exp1}, \text{exp2}] \text{ (the type is the same as the type of arguments)} \\
\text{SMSPower}[x, y] & \quad x^{y} \text{ (see details below)} \\
\text{SMSSqrt}[x] & \quad \sqrt{x} \text{ (see details below)} \\
\text{SMSMod}[i, j] & \quad \text{remainder of } i/j \text{ (the type is the same as the type of arguments)} \\
\text{SMSFloor}[x] & \quad \text{returns the nearest integer not greater than the given value (real type function)} \\
\text{SMSCeiling}[x] & \quad \text{returns the smallest integer value greater than or equal to } x \text{ (real type function)}
\end{align*}
\]

Functions with unique signature and support for derivatives.

\[
\text{SMSSqrt}(\sqrt{x})
\]

\[
\begin{align*}
\text{SMSSqrt}[x] & \quad \sqrt{x} \text{ with an unique signature (see Expression Optimization)} \\
(\text{SMSSqrt}[x]=\text{SMSSqrt}[x,0,\infty])
\end{align*}
\]

\[
\begin{align*}
\text{SMSSqrt}[x, \text{type}, dc] & \quad dc=0 \quad \Rightarrow \text{evaluate function only} \\
& \quad dc=1 \quad \Rightarrow \text{evaluate function and its first derivative} \\
& \quad dc=2 \quad \Rightarrow \text{evaluate function and its first and second derivative} \\
\end{align*}
\]

\[
\begin{align*}
type=0 & \quad \Rightarrow \sqrt{x} \\
type=1 & \quad \Rightarrow \text{function } \sqrt{x} \text{ is approximated with } 5\text{-th order polynomial approximation on interval } [-\infty,0] \\
type=2 & \quad \Rightarrow \text{function } \sqrt{x} \text{ is approximated with Pade approximation on interval } [-\infty,0]
\end{align*}
\]
Options of the SMSSqrt function.

The square root function exhibits the singularity in derivatives around the origin. The SMSSqrt modifies the square root function on interval \([0,x_0]\) in a way that singularity in derivatives is avoided.

\[
\text{SMSPower}(x^y)
\]

\[
\text{SMSPower}[x,y]
\]

\(x^y\)

this form is appropriate when \(x>0\) or \(y\) is positive integer

\[
\text{SMSPower}[x,y, \text{type}, dc]
\]

dc=0 ⇒ evaluate function only

dc=1 ⇒ evaluate function and its first derivative

dc=2 ⇒ evaluate function and its first and second derivative

type=0 ⇒ \(x^y\) (≡ SMSPower\((x,y)\))

type=1 ⇒ for \(y=1\) is function \(x^1\) is approximated with 5-th order polynomial approximation on interval \([-\infty,x_0]\)

type=2 ⇒ function \(x^y\) is approximated with Pade approximation for \(0<y<1\) and 5-th order polynomial approximation for \(y<0\) on interval \([-\infty,x_0]\)
The $x^y$ function exhibits the singularity in function or its derivatives around the origin for $y<1$. The SMSPower modifies the $x^y$ function on interval $[0,x_0]$ in a way that singularities are avoided.
Standard Functions Without Unique Signature

- \( \tan(x) \)
- \( \cot(x) \)
- \( \cos(x) \)
- \( \sec(x) \)
- \( \sin(x) \)
- \( \text{Scs}(x) \)
- \( \arctan(x) \)
- \( \arccos(x) \)
- \( \arcsin(x) \)
- \( \log(x) \)
- \( \exp(x) \)
- \( \cosh(x) \)
- \( \sinh(x) \)
- \( \text{SmsCbrt}(x) \) \( x^3 \) (also works for negative real \( x \))

Functions without unique signature and support for derivatives provided by Mathematica.

System Utility Functions

- \( \text{SMSNumberQ}(\exp) \) gives True if \( \exp \) is a real number and False if the results of the evaluation is \( \text{N/A} \) (not a number). \( \text{SMSNumberQ} \) produces a code segment that test results at run-time.
- \( \text{SMSTime}[] \) returns number of seconds elapsed since midnight (00:00:00), January 1, 1970, coordinated universal time (UTC)

System functions with unique signature and no support for derivatives.

Compressing Structures

\( \text{structCompressed} = \text{SMSCompress}([\text{struct}]) \)

- Analyzes structure and finds pattern inside the structure that can be used to compress the structure into vector. Function returns \( \text{structCompressed} \) data structure.
- \( \text{structCompressed}["\text{Data}"] \) returns compressed \( \text{struct} \)
- \( \text{structCompressed}["\text{Length}"] \) returns the length of vector into which the structure is compressed.

\( \text{SMSUncompress}([\text{structCompressed}]) \)

- Returns uncompressed structure \( \text{struct} \)
- \( \text{SMSUncompress}([\text{structCompressed}, \text{compressedVector}] \)

- Returns uncompressed \( \text{compressedVector} \)
- \( \text{SMSUncompress}([\text{structCompressed}, \text{otherStruct}] \)

- Returns \( \text{otherStruct} \) modified in such a way that \( \text{otherStruct} \) has the same pattern as \( \text{struct} \).
- \( \text{structCompressed} = \text{SMSCompress}([\text{struct}, "\text{Export}"->\text{Function}[[i], f(i)]] \)

- \( \text{struct} \) is compressed into vector which is then exported (stored) into used defined memory space
- \( \text{SMSUncompress}([\text{structCompressed}, "\text{Import}"->\text{importKey}] \)

- \( \text{compressedVector} \) is imported from user defined memory space accordingly to the \( \text{importKey} \).
<table>
<thead>
<tr>
<th>option</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Export&quot;</td>
<td>False</td>
<td>False =&gt; nothing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Function[i, f(i)] ⇒ structure is compressed into vector which is then exported (stored) into used defined memory space. User supplied memory space must be sufficiently large. (e.g SMSCompress[struct, &quot;Export&quot;-&gt;Function[i, cVector$[i]]]) will export compressed struct into array cVector[maxLength]).</td>
</tr>
<tr>
<td>&quot;AddIn&quot;</td>
<td>False</td>
<td>compressed data is added to the already stored compressed data</td>
</tr>
<tr>
<td>&quot;Import&quot;</td>
<td>False</td>
<td>True =&gt; compressed vector is taken from the same memory space it was used before to store compressed data. Function[i, f(i)] ⇒ compressed vector is taken from the memory space Table[f(i), {i, 1, structCompressed[&quot;Length&quot;]].</td>
</tr>
</tbody>
</table>

Options of the SMSPower function.

Example

```
In[193]:= << AceGen;
SMSInitialize["tmptest", "Environment"->"MathLink", "VectorLength"->50, "Mode"->"Optimal"];
SMSModule["Test", Real[u$$[3], x$$, L$$, k21$$[4], c$$[9]],
  "Input"->{u$$, x$$, L$$}, "Output"->{k21$$, c$$}];
{x, L} = {SMSReal[x$$], SMSReal[L$$]};
u i = SMSReal[Array[u$$[Hi 1 & , 3]]];
Ni = {x, 1 - x, x/L, L (1 - x/L)};
u = Ni.ui;
f = u;
g = SMSD[f, ui];
struct = SMSD[g, ui];

In[203]:= struct
Out[203]= {struct, struct, struct}, {struct, struct, struct}, {struct, struct, struct}

- analyze structure and find pattern inside the structure that can be used to compress the structure into vector. Function returns data that can be used to compress or uncompress structures.

In[204]:= structCompressed = SMSCompress[struct];
- returns the length of vector into which the structure is compressed

In[205]:= structCompressed["Length"]
Out[205]= 6
- returns uncompressed structure accordingly to the given signature

In[206]:= structUncompressed = SMSUncompress[structCompressed]
Out[206]= {struct, struct, struct}, {struct, struct, struct}, {struct, struct, struct}
- returns uncompressed compressedVector

In[207]:= compressedVector = Table[i, {i, 6}]
Out[207]= {1, 2, 3, 4, 5, 6}

In[208]:= SMSUncompress[structCompressed, compressedVector]
Out[208]= {{1, 2, 3}, {2, 4, 5}, {3, 5, 6}}
```
struct2 is original structure with symmetric element 2,1 set to zero.

```
In[209]:= otherStruct = ReplacePart[struct, (2, 1) -> 0]
```

```
Out[209]=
```

otherStruct is modified in such a way that otherStruct has the same pattern as struct, thus element 2,1 is set back to the original value

```
In[210]:= otherStructModified = SMSUncompress[structCompressed, otherStruct]
```

```
Out[210]=
```

compress struct and export compressed vector into working vector c$$

```
In[211]:= structCompressed2 = SMSCompress[struct, "Export" -> Function[i], c$$[i]]
```

```
Out[211]=
```

import compressed data from working vector c$$ and uncompress it

```
In[212]:= Table[Function[i], c$$[i], {i, 6}]
```

```
Out[212]=
```

```
In[213]:= structUncompressed2 = SMSUncompress[structCompressed2, "Import" -> True]
```

```
Out[213]=
```

```
In[214]:= SMSExport[{struct[[2, 1]], structUncompressed[[2, 1]], otherStructModified[[2, 1]], structUncompressed2[[2, 1]]}, k21$$];
```

Utility Functions for Numerical Environments

- SMSIsDOFConstrained
- SMSNoDummyNodes
- SMSLastTrueNode
- SMSLastTrueDOF
- SMSIsDummyNode
- SMSIsRealNode
CHAPTER 3

Specialized Functions
Linear Algebra

Enormous growth of expressions typically appears when the SAC systems such as Mathematica are used directly for solving e.g. a system of linear algebraic equations analytically. It is caused mainly due to the redundant expressions, repeated several times. Although the operation is "local" by its nature, only systems with a small number of unknowns (up to 10) can be solved analytically. When linear algebra operations are executed on numerical matrices, no expression growth appears, thus they can be applied on the matrices of an arbitrary size.

With AceGen we can generate a code that evaluates linear algebra operations numerically or symbolically (closed form solution). Numerical solution is performed by calling the LAPACK linear algebra routines. If the code is generated for the AceFEM finite element environment, the necessary link is done automatically. For other cases the uses must provide the name of the routine and link the executable with an appropriate numerical library.

In all closed form linear algebra routines it is assumed that the solution exist (det(A)≠0) and that no pivoting is needed. Pivoting is automatically done for the numerical solution by LAPACK package.

The following routines are provided:

<table>
<thead>
<tr>
<th>operation</th>
<th>solver available</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMSLinearSolve</td>
<td>&quot;Closed form&quot;/</td>
<td>solution to the system of linear equations</td>
</tr>
<tr>
<td></td>
<td>&quot;LAPACK&quot;</td>
<td></td>
</tr>
<tr>
<td>SMSEigensystem</td>
<td>&quot;LAPACK&quot;</td>
<td>eigenvalues and eigenvectors (only supported for AceFEM finite elements)</td>
</tr>
<tr>
<td>SMSLUFactor</td>
<td>&quot;Closed form&quot;</td>
<td>the LU decomposition</td>
</tr>
<tr>
<td>SMSFactorSim</td>
<td>&quot;Closed form&quot;</td>
<td>the LU decomposition along with the pivot list of symmetric matrix M</td>
</tr>
<tr>
<td>SMSLUSSolve</td>
<td>&quot;Closed form&quot;</td>
<td>solution of the linear system represented</td>
</tr>
<tr>
<td></td>
<td></td>
<td>by decomposed matrix LU and right-hand side B</td>
</tr>
<tr>
<td>SIMInverse</td>
<td>&quot;Closed form&quot;</td>
<td>the inverse of square matrix M</td>
</tr>
<tr>
<td>SIMDet</td>
<td>&quot;Closed form&quot;</td>
<td>the determinant of square matrix M</td>
</tr>
<tr>
<td>SMSKrammer[M,B]</td>
<td>&quot;Closed form&quot;</td>
<td>generate a code sequence that solves the system of linear equations A x=</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B analytically and return the solution vector</td>
</tr>
<tr>
<td>SMSEigenvalues</td>
<td>&quot;Closed form&quot;</td>
<td>create code sequence that calculates the eigenvalues of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>third order (3x3) matrix and return the vector of 3 eigenvalues</td>
</tr>
<tr>
<td>SIMInvartantsI</td>
<td>&quot;Closed form&quot;</td>
<td>I₁, I₂, I₃ invariants of the third order matrix</td>
</tr>
<tr>
<td>SIMInvartantsJ</td>
<td>&quot;Closed form&quot;</td>
<td>J₁, J₂, J₃ invariants of the deviator of the third order matrix</td>
</tr>
</tbody>
</table>

SMSLinearSolve

```
SMSLinearSolve[A,B]
```

generate the code sequence that solves the system of linear equations $A x= B$ analytically and return the solution vector.

Parameter A is a square matrix. Parameter B can be a vector (one right-hand side) or a matrix (multiple right-hand sides). The Gauss elimination procedure is used without pivoting.
### Options of SMSLinearSolve function.

#### Example

This generates the FORTRAN code that returns the solution to the general linear system of equations:

\[
\begin{bmatrix}
  a_{11} & a_{12} & \cdots & a_{1N} \\
  a_{21} & a_{22} & \cdots & a_{2N} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{N1} & a_{N2} & \cdots & a_{NN}
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  x_2 \\
  \vdots \\
  x_N
\end{bmatrix}
= 
\begin{bmatrix}
  b_1 \\
  b_2 \\
  \vdots \\
  b_N
\end{bmatrix}
\]

```plaintext
In[22]:= << AceGen
SMSInitialize["test", "Language" -> "C"];
SMSModule["Test", Real[a$$[4, 4], b$$[4], x$$[4]]];
a + SMSReal[Table[a$$[i, j], {i, 4}, {j, 4}]];
b + SMSReal[Table[b$$[i], {i, 4}]];
x + SMSLinearSolve[a, b];
SMSExport[x, x$$];
SMSWrite[];
```

File:  test.c  Size:  1425

<table>
<thead>
<tr>
<th>Methods</th>
<th>No. Formulae</th>
<th>No. Leafs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
<td>18</td>
<td>429</td>
</tr>
</tbody>
</table>
SMSEigensystem

SMSEigensystem[B]

calculates eigenvalues and eigenvectors of matrix $A$ using LAPACK numerical library

Function returns:
- (eigenvalues, list of eigenvectors) for symmetric matrix
- (real part of eigenvalues, complex part of eigenvalues, list of eigenvectors) for nonsymmetric matrix

Eigenvalues are given in descending order.

Only available for AceFEM finite elements!
### Options of SMSEigensystem function.

Example

\[
\{\sigma, \sigma_{\text{Main}}, \sigma_{\text{MainDirection}}\} \ni \text{SMSEigensystem}[\sigma, \text{"Solver" \to \"LAPACK\", \"Symmetric" \to \text{True}}];
\]

#### SMSLUFactor

**SMSLUFactor\[A\]**

the LU decomposition along with the pivot list of \( A \)

The Gauss elimination procedure is used and simultaneous simplification is performed during the process. The SMSLUFactor performs the factorization of matrix \( A \) and returns a new matrix. The matrix generated by the SMSLUFactor is a compact way of storing the information contained in the upper and lower triangular matrices of the factorization.

### Options of SMSLUFactor function.

#### SMSSLUSolve

**SMSSLUSolve\[LU,B\]**

solution of the linear system represented by decomposed matrix \( LU \) and right-hand side \( B \)

\(\text{SMSLinearSolve}[A,B]\) = SMSSSLUSolve[SMSSLUFactor[A],B]

The Gauss elimination procedure is used and simultaneous simplification is performed during the process. Parameter \( B \) can be a vector (one right-hand side) or a matrix (multiple right-hand sides).

#### SMSFactorSim

**SMSFactorSim\[M\]**

the LU decomposition along with the pivot list of symmetric matrix \( M \)

The Gauss elimination procedure is used and simultaneous simplification is performed during the process. The SMSFactorSim...
performs factorization of the matrix $A$ and returns a new matrix. The matrix generated by the SMSFactorSim is a compact way of storing the information contained in the upper and lower triangular matrices of the factorization.

**SMSInverse**

$$\text{SMSInverse}[M]$$

the inverse of square matrix $M$

Simultaneous simplification is performed during the process. The Krammer’s rule is used and simultaneous simplification is performed during the process. For more than 6 equations is more efficient to use SMSLinearSolve[$M$,IdentityMatrix[$M/\text{Length}$]] instead.

**SMSDet**

$$\text{SMSDet}[M]$$

the determinant of square matrix $M$

Simultaneous simplification is performed during the process.

**SMSKrammer**

$$\text{SMSKrammer}[M,B]$$

generate a code sequence that solves the system of linear equations $A \times B$ analytically and return the solution vector

The Krammer’s rule is used and simultaneous simplification is performed during the process.

**SMSEigenvalues**

$$\text{SMSEigenvalues}[M]$$

create code sequence that calculates the eigenvalues of the third order (3×3) matrix and return the vector of 3 eigenvalues

All eigenvalues have to be real numbers. Solution is obtained by solving a general characteristic polynomial. Ill-conditioning around multiple zeros as well as wrong derivatives can occur!! In general, it is advisable to avoid the formulations that involve eigenvalues and use formulations based on matrix functions as described in Matrix Functions.

**SMSInvariantsI**

$$\text{SMSInvariantsI}[M]$$

$I_1, I_2, I_3$ invariants of the third order matrix

**SMSInvariantsJ**

$$\text{SMSInvariantsJ}[M]$$

$J_1, J_2, J_3$ invariants of the deviator of the third order matrix
Tensor Algebra

\[ \text{SMSCovariantBase}[\{\phi_1, \phi_2, \phi_3\}, \{\eta_1, \eta_2, \eta_3\}] \]

the covariant base vectors of transformation from the coordinates \(\{\eta_1, \eta_2, \eta_3\}\) to coordinates \(\{\phi_1, \phi_2, \phi_3\}\)

Transformations \(\phi_1, \phi_2, \phi_3\) are arbitrary functions of independent variables \(\eta_1, \eta_2, \eta_3\). Independent variables \(\eta_1, \eta_2, \eta_3\) have to be proper auxiliary variables with unique signature (see also SMSD).

Example: Cylindrical coordinates

\[ \begin{bmatrix} \text{Cos}[\phi] & \text{Sin}[\phi] & 0 \\ -\text{Sin}[\phi] & \text{Cos}[\phi] & 0 \\ 0 & 0 & 1 \end{bmatrix} \]

\text{SMSCovariantMetric}[\{\phi_1, \phi_2, \phi_3\}, \{\eta_1, \eta_2, \eta_3\}] 

the covariant metric tensor of transformation from coordinates \(\{\eta_1, \eta_2, \eta_3\}\) to coordinates \(\{\phi_1, \phi_2, \phi_3\}\)

Transformations \(\phi_1, \phi_2, \phi_3\) are arbitrary functions of independent variables \(\eta_1, \eta_2, \eta_3\). Independent variables \(\eta_1, \eta_2, \eta_3\) have to be proper auxiliary variables with unique signature (see also SMSD).

Example: Cylindrical coordinates

\[ \begin{bmatrix} 1 & 0 & 0 \\ 0 & i^2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \]

\text{SMSContravariantMetric}[\{\phi_1, \phi_2, \phi_3\}, \{\eta_1, \eta_2, \eta_3\}] 

the contravariant metric tensor of transformation from coordinates \(\{\eta_1, \eta_2, \eta_3\}\) to coordinates \(\{\phi_1, \phi_2, \phi_3\}\)

Transformations \(\phi_1, \phi_2, \phi_3\) are arbitrary functions of independent variables \(\eta_1, \eta_2, \eta_3\). Independent variables \(\eta_1, \eta_2, \eta_3\) have to be proper auxiliary variables with unique signature (see also SMSD).

Example: Cylindrical coordinates
In[378]:= << AceGen``
SMSInitialize["test", "Language" -> "Mathematica"];
SMSModule["test"]; {r, ϕ, z} = Array[SMSFictive[] &, {3}];
SMSContravariantMetric[{r Cos[ϕ], r Sin[ϕ], z}, {r, ϕ, z}] // MatrixForm
Out[380]//MatrixForm =
\[
\begin{pmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{pmatrix}
\]

SMSChristoffell1[{ϕ₁, ϕ₂, ϕ₃}, {η₁, η₂, η₃}]
the first Christoffell symbol \(\Gamma_{ij}^k\) of transformation from coordinates \(\{\eta_1, \eta_2, \eta_3\}\) to coordinates \(\{\phi_1, \phi_2, \phi_3\}\)

Transformations \(\phi_1, \phi_2, \phi_3\) are arbitrary functions of independent variables \(\eta_1, \eta_2, \eta_3\). Independent variables \(\eta_1, \eta_2, \eta_3\) have to be proper auxiliary variables with unique signature (see also SMSD).

Example: Cylindrical coordinates

In[383]:= << AceGen``
SMSInitialize["test", "Language" -> "Mathematica"];
SMSModule["test"]; {r, ϕ, z} = Array[SMSFictive[] &, {3}];
SMSChristoffell2[{ϕ₁, ϕ₂, ϕ₃}, {η₁, η₂, η₃}]
Out[386]//MatrixForm =
\[
\begin{pmatrix}
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0
\end{pmatrix}
\]

SMSChristoffell2[{ϕ₁, ϕ₂, ϕ₃}, {η₁, η₂, η₃}]
the second Christoffell symbol \(\Gamma_{ij}^k\) of transformation from coordinates \(\{\eta_1, \eta_2, \eta_3\}\) to coordinates \(\{\phi_1, \phi_2, \phi_3\}\)

Transformations \(\phi_1, \phi_2, \phi_3\) are arbitrary functions of independent variables \(\eta_1, \eta_2, \eta_3\). Independent variables \(\eta_1, \eta_2, \eta_3\) have to be proper auxiliary variables with unique signature (see also SMSD).

Example: Cylindrical coordinates


\begin{verbatim}
\(Out[388]//MatrixForm= \)
\begin{align*}
\begin{pmatrix}
0 & 0 & 0 \\
0 & 1 & 0 \\
\frac{1}{r} & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0
\end{pmatrix}
\end{align*}
\end{verbatim}

\textbf{SMSTensorTransformation\([\text{tensor,transf,coord,index\_types}]\)}

tensor transformation of arbitrary tensor field \(\text{tensor}\) with indices \(\text{index\_types}\) defined in curvilinear coordinates \(\text{coord}\) under transformation \(\text{transf}\).

Transformations \(\text{transf}\) are arbitrary functions while coordinates \(\text{coord}\) have to be proper auxiliary variables with the unique signature (see also \(\text{SMSD}\)). The type of tensor indices is specified by the array \(\text{index\_types}\) where \(\text{True}\) means covariant index and \(\text{False}\) contravariant index.

Example: Cylindrical coordinates

Transform contravariant tensor \(u^i = \{r^2, r \sin(\phi), rz\}\) defined in cylindrical coordinates \(\{r, \phi, z\}\) into Cartesian coordinates.

\(Out[393]//\) MatrixForm=
\begin{align*}
\begin{pmatrix}
\cos(\phi) r^2 + \sin(\phi)^2, \\
\cos(\phi) \sin(\phi) - \sin(\phi), \frac{1}{r} \sin(\phi), \frac{1}{r}
\end{pmatrix}
\end{align*}

\textbf{SMSDCovariant\([\text{tensor,transf,coord,index\_types}]\)}

covariant derivative of arbitrary tensor field \(\text{tensor}\) with indices \(\text{index\_types}\) defined in curvilinear coordinates \(\text{coord}\) under transformation \(\text{transf}\).

Transformations \(\text{transf}\) are arbitrary functions while coordinates \(\text{coord}\) have to be proper auxiliary variables with unique signature (see also \(\text{SMSD}\)). The type of tensor indices is specified by the array \(\text{index\_types}\) where \(\text{True}\) means covariant index and \(\text{False}\) contravariant index.

The \(\text{SMSDCovariant}\) function accepts the same options as \(\text{SMSD}\) function.

Example: Cylindrical coordinates

Derive covariant derivatives \(u^i_{\ j}\) of contravariant tensor \(u^i = \{r^2, r \sin(\phi), rz\}\) defined in cylindrical coordinates \(\{r, \phi, z\}\).
In[398]:= << AceGen`
   SMSInitialize["test", "Language" -> "Mathematica"];
   SMSModule["test"]; {r, \(\phi\), z} \[Function] Array[SMSFictive[] &, {3}]; 
   SMSDCovariant[\{r^2, r \ Sin[\(\phi\)], r z\}, \{r \ Cos[\(\phi\)], r \ Sin[\(\phi\)], z\}, \{r, \(\phi\), z\}, \{False\}] // MatrixForm

Out[398]//MatrixForm =

\[
\begin{pmatrix}
2 & -r^2 \ Sin[\(\phi\)] & 0 \\
2 \ Sin[\(\phi\)] & r \ Cos[\(\phi\)] & 0 \\
0 & 0 & 1
\end{pmatrix}
\]
Matrix Functions

**SMSMatrixExp**[$\mathbf{M}, dc$] returns exponential of 3×3 matrix with real eigenvalues, where $dc$ is a code that defines the number of derivatives calculated together with the function evaluation as follows:

- $dc=0$ \Rightarrow function only $f(\mathbf{M})$
- $dc=1$ \Rightarrow function and its first derivative $\left(f(\mathbf{M}), \frac{\partial f(\mathbf{M})}{\partial \mathbf{M}}\right)$
- $dc=2$ \Rightarrow function and its first and second derivative $\left(f(\mathbf{M}), \frac{\partial f(\mathbf{M})}{\partial \mathbf{M}}, \frac{\partial^2 f(\mathbf{M})}{\partial \mathbf{M}^2}\right)$

**SMSMatrixLog**[$\mathbf{M}, dc$] returns logarithm of positive definite 3×3 matrix,

**SMSMatrixSqrt**[$\mathbf{M}, dc$] returns square root of positive definite 3×3 matrix,

**SMSMatrixPower**[$\mathbf{M}, p, dc$] returns $\mathbf{M}^p$ where $\mathbf{M}$ is positive definite 3×3 matrix and $p$ is an arbitrary real number,

**SMSMatrixPowerSeries**[$\mathbf{M}, cp, n, dc$],

\[ \sum_{i=1}^{n} c_i \mathbf{M}^{p_i} \] where $\mathbf{M}$ is positive definite 3×3 matrix and $cp=[c_1, p_1, \ldots]$ is a vector or pointer to vector of coefficients and powers and $n$ is integer defining the number of terms taken from $cp$.

The form of $cp$ has to be appropriate for the **SMSCall** function argument.

<table>
<thead>
<tr>
<th>option name</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Report&quot;</td>
<td>False</td>
</tr>
</tbody>
</table>
|              | False \Rightarrow function returns calculated function value: $f(\mathbf{M})$  
|              | True \Rightarrow function returns calculated function value and report value $r$: $[f(\mathbf{M}), r]$ |
| "Order"      | Infinity      |
|              | Infinity \Rightarrow closed form representation  
|              | r_Integer \Rightarrow Taylor series expansion of order $r$  
|              | $\epsilon_{\text{Real}}$ \Rightarrow Taylor series expansion truncated when the norm of term is less than $\epsilon$ |

Options for matrix functions.

AceGen provides closed form representation of some important matrix functions of 3×3 matrix together with its first and second derivatives. Matrix functions are provided as compiled libraries for MathLink, CDriver and MDriver environments. For a commercial and noncommercial use in other numerical environments please refer to AceProducts@fgg.uni-lj.si. More about matrix function can be found in HUDOBIšNIK, Blaž, KORELC, Jože. Closed-form representation of matrix functions in the formulation of nonlinear material models. Finite Elements in Analysis and Design, 2016, 111:19-32


**WARNING:** Library with the compiled matrix functions is a part of AceFEM installation, thus full AceFEM package is required in order to run examples with matrix functions.

Function returns calculated functions value. Derivatives are not returned directly. An appropriate AD exceptions are defined for each component of matrix functions that defined first and second order derivatives.

In the case of "Report"->True option, function additional returns integer $r$ with the following value:

- $r = 0$ \Rightarrow matrix function cannot be evaluated due to the lack of positive definiteness of matrix (or real eigenvalues in the case of matrix exponential),
- $r = 1$ \Rightarrow regular point,
- $r = 2$ \Rightarrow singular point A (multiple eigenvalues),
- $r = 3$ \Rightarrow singular point B (complex eigenvalues),
- $r = 4$ \Rightarrow singular point C (complex eigenvalues).

**Examples**

A) Matrix exponential

- Create MathLink program that calculates matrix exponential.
In[82]:= <<AceGen`;
SMSInitialize["ExpM", "Environment" -> "MathLink"];
SMSModule["ExpM", Real[MM[[3, 3]], EMM[[3, 3]]], "Input" -> M$, "Output" -> EM$];
M0 = SMSReal[Array[M$, {3, 3}]];
EM = SMSMatrixExp[M0, 0];
SMSExport[EM, EMM$];
SMSWrite[];
SMSInstallMathLink["ExpM", "Console" -> True];

Closed form matrix function module. Derivatives are not supported.
See also: MatrixFunctions

<table>
<thead>
<tr>
<th>File</th>
<th>Size: 2201</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods</td>
<td>No.Formulae No.Leafs</td>
</tr>
<tr>
<td>ExpM</td>
<td>3 214</td>
</tr>
</tbody>
</table>

B) Matrix power and its first derivative

Test: calculate $e^I$.

In[78]= M = {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}};
ExpM[M] // MatrixForm
MatrixExp[M] // MatrixForm

Out[78]/MatrixForm=
\[
\begin{pmatrix}
2.71828 & 0. & 0. \\
0. & 2.71828 & 0. \\
0. & 0. & 2.71828 \\
\end{pmatrix}
\]

Out[78]/MatrixForm=
\[
\begin{pmatrix}
e & 0 & 0 \\
0 & e & 0 \\
0 & 0 & e \\
\end{pmatrix}
\]

B) Matrix power and its first derivative

Create MathLink program that calculates $M^p$ together with its first derivative.

In[79]= <<AceGen`;
SMSInitialize["PowerM", "Environment" -> "MathLink"];
SMSModule["PowerM", Real[MM[[3, 3]], p$, EMM[[3, 3]], DEMM[[3, 3, 3, 3]]],
"Input" -> {M$, p$}, "Output" -> {EM$, DEM$;};
M0 = SMSReal[Array[M$, {3, 3}]];
p = SMSReal[p$];
EM = SMSMatrixPower[M0, 0, p];
SMSExport[EM, EMM$];
DEM = SMSD[EM, M0];
SMSExport[DEM, DEM$];
SMSWrite[];
SMSInstallMathLink["PowerM", "Console" -> True];

Closed form matrix Power module for
full matrix. Number of terms in $M$ is 9 and in $f(M)$ is 9.
First derivatives $df(M)/dM$ are supported with 45 terms. See also: MatrixFunctions

<table>
<thead>
<tr>
<th>File</th>
<th>Size: 5709 Time: 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>No.Formulae No.Leafs</td>
</tr>
<tr>
<td>PowerM</td>
<td>41 1203</td>
</tr>
</tbody>
</table>
Test: calculate \((2I)^{3.5}\) and its first derivatives.

```math
M = \{\{1, 0, 0\}, \{0, 1, 0\}, \{0, 0, 1\}\};
(powerM, dpowerM) = PowerM[M, 3.5];
```

```math
powerM // MatrixForm
```

```
\[
\begin{bmatrix}
11.3137 & 0. & 0. \\
0. & 11.3137 & 0. \\
0. & 0. & 11.3137 \\
\end{bmatrix}
\]
```

```math
MatrixPower[M, 3.5] // MatrixForm
```

```
\[
\begin{bmatrix}
11.3137 & 0. & 0. \\
0. & 11.3137 & 0. \\
0. & 0. & 11.3137 \\
\end{bmatrix}
\]
```

```math
dpowerM // MatrixForm
```

```
\[
\begin{bmatrix}
19.799 & 0. & 0. \\
0. & 1.77636 \times 10^{-15} & 0. \\
0. & 0. & 1.77636 \times 10^{-15} \\
\end{bmatrix}
\]
```

```
\[
\begin{bmatrix}
0. & 19.799 & 0. \\
0. & 0. & 0. \\
0. & 0. & 0. \\
\end{bmatrix}
\]
```

```
\[
\begin{bmatrix}
0. & 1.77636 \times 10^{-15} & 0. \\
0. & 19.799 & 0. \\
0. & 0. & 1.77636 \times 10^{-15} \\
\end{bmatrix}
\]
```

```
\[
\begin{bmatrix}
0. & 0. & 0. \\
0. & 0. & 0. \\
0. & 0. & 0. \\
\end{bmatrix}
\]
```

```
\[
\begin{bmatrix}
0. & 0. & 0. \\
0. & 0. & 0. \\
0. & 0. & 0. \\
\end{bmatrix}
\]
```

```
\[
\begin{bmatrix}
19.799 & 0. & 0. \\
0. & 19.799 & 0. \\
0. & 0. & 19.799 \\
\end{bmatrix}
\]
```

```
\[
\begin{bmatrix}
19.799 & 0. & 0. \\
0. & 19.799 & 0. \\
0. & 0. & 19.799 \\
\end{bmatrix}
\]
```

```math
1.776:
```

```
\[
\begin{bmatrix}
1.77636 \times 10^{-15} & 0. & 0. \\
0. & 19.799 & 0. \\
0. & 0. & 19.799 \\
\end{bmatrix}
\]
```

```
\[
\begin{bmatrix}
0. & 0. & 0. \\
0. & 0. & 0. \\
0. & 0. & 0. \\
\end{bmatrix}
\]
Numerical Root Finding

Contents

- Numerical root finding of scalar function
- Numerical root finding of a system of nonlinear equations
- Numerical root finding of a system of nonlinear, path dependent, parameterized equations

Numerical root finding of scalar function

\[
\text{SMSFindRoot}\left[f, x_0\right]
\]
inserts a code that searches for a numerical root of explicitly defined scalar function \( f(x) \), starting from the point \( x = x_0 \). Function \( f(x) \) must be an explicit expression with the unknown \( x \) explicitly visible.

\[
\text{SMSFindRoot}\left[f\_\text{Function}, x_0\right]
\]
inserts a code that searches for a numerical root of scalar function defined by a pure function (e.g., \( \text{Function}\left[\{x\}, x^2\right] \)), starting from the point \( x_0 \). Function can create an algorithm.

Function \( \text{SMSFindRoot} \) returns a vector:

\[
\{x - \text{calculated root}, \text{True if convergence was obtained and False if not}, \text{number of iteration for scalar function or number of sub-steps for parameterized nonlinear system}\}
\]

The returned root \( x \) is an auxiliary variable. When implicit derivatives are required (option "Derivatives") an automatic differentiation exceptions are attached to the \( x \) that define first and optionally second order derivatives of \( x \) with respect to given implicit variables (option "ImplicitVariables").

<table>
<thead>
<tr>
<th>option name</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;MaxIterations&quot;</td>
<td>30</td>
</tr>
<tr>
<td>&quot;Tolerance&quot;</td>
<td>( 10^{-10} )</td>
</tr>
</tbody>
</table>
| "Derivatives" | 0 | 0 \( \Rightarrow \) find root only 
| &nbsp; | 1 \( \Rightarrow \) find root together with the first derivatives of the root with respect to implicit variables |
| &nbsp; | 2 \( \Rightarrow \) find root together with the first and second derivatives of the root with respect to implicit variables |
| "ImplicitVariables" | Automatic | variables for which derivatives of the root have to be calculated and defined |

General options.

Example

- calculate root of equation \( ax^3 + bx + c \) together with its first and second derivative with respect to \( a, b, \) and \( c \)

\( \text{a} = << \text{AceGen}; \)

\( \text{a} = \text{SMSInitialize}["tmptest", "Environment" \rightarrow "MathLink"]; \)

\( \text{SMSModule}["test", \text{Real}[a$$, b$$, c$$, x0$$, tol$$, x$$, dxdfi$$[3], d2xdfi$$[3, 3]], \text{Integer}[maxNR$$, NR$$], "Input" \rightarrow \{a$$, b$$, c$$, x0$$, tol$$, maxNR$$\}, \text{"Output" \rightarrow \{x$$, dxdfi$$, d2xdfi$$, NR$$\}; \}

\( \{a, b, c, tol\} \ast \text{SMSReal}[\{a$$, b$$, c$$, tol$$\}]; \)

\( x \ast \text{SMSReal}[x0$$]; \)
\( \phi = \{ a, b, c \} \);
maxNR \rightarrow \text{SMSInteger}[\text{maxNR}];
\( (x, \text{conv}, \text{NR}) \rightarrow \text{SMSFindRoot}[a x^3 + b x + c, (x, x),
"\text{Derivatives}" \rightarrow 2, "\text{MaxIterations}" \rightarrow \text{maxNR}, "\text{Tolerance}" \rightarrow \text{tol}] \);
\text{SMSExport}[x, x];
maxNR \vdash \text{SMSInteger}[\text{maxNR}];
\{x, \text{conv}, \text{NR}\} \models \text{SMSFindRoot}[a x^3 + b x + c, (x, x),
"\text{Derivatives}" \rightarrow 2, "\text{MaxIterations}" \rightarrow \text{maxNR}, "\text{Tolerance}" \rightarrow \text{tol}];
\text{SMSExport}[x, x];
\Delta x \models \text{SMSD}[x, \phi];
\text{SMSExport}[\Delta x, \text{dxdfi}];
\Delta \Delta x \models \text{SMSD}[\Delta x, \phi];
\text{SMSExport}[\Delta \Delta x, \text{d2xdfi2}];
\text{SMSExport}[\text{NR}, \text{NR}];
\text{SMSWrite}[\text{NR}, \text{NR}];
\text{File: tmptest.c} \text{ Size: } 3625 \text{ Time: } 1
\text{Method} \text{ test} \text{ No.Formulae} 35 \text{ No.Leafs} 387
In[29]:=
\phi = \{ a, b, c \};
maxNR \rightarrow \text{SMSInteger}[\text{maxNR}];
(x, \text{conv}, \text{NR}) \rightarrow \text{SMSFindRoot}[a x^3 + b x + c, (x, x),
"\text{Derivatives}" \rightarrow 2, "\text{MaxIterations}" \rightarrow \text{maxNR}, "\text{Tolerance}" \rightarrow \text{tol}];
\text{SMSExport}[x, x];
\Delta x \models \text{SMSD}[x, \phi];
\text{SMSExport}[\Delta x, \text{dxdfi}];
\Delta \Delta x \models \text{SMSD}[\Delta x, \phi];
\text{SMSExport}[\Delta \Delta x, \text{d2xdfi2}];
\text{SMSExport}[\text{NR}, \text{NR}];
\text{SMSWrite}[\text{NR}, \text{NR}];
\text{File: tmptest.c} \text{ Size: } 3625 \text{ Time: } 1
\text{Method} \text{ test} \text{ No.Formulae} 35 \text{ No.Leafs} 387
\text{Out[35]=} \text{test}[1, 1, 1, -0.68, 10. ^{-12}, 10]
\text{Out[36]=} (-0.682328, \{0.132545, 0.284693, -0.417238\}, \{-0.124475, -0.156754, 0.148685\},
\{-0.156754, -0.099123, -0.0288157\}, \{0.148685, -0.0288157, 0.297369\}), 5
\text{\textbf{ verification}}
In[97]:= \texttt{x = ;}
\text{rr }= \texttt{Solve}[a a x^3 + b b x + c c = 0, x][[1, 2]]
\text{Out[97]=} \left(\frac{2}{3}\right)^{1/3} b b
+ \left(-9 a a^2 c c + \sqrt{3} \left(4 a a^3 b b^3 + 27 a a^4 c c^2\right)\right)^{1/3}
2^{1/3} \times 3^{2/3} a a
\text{Out[98]=} \dot{\texttt{d d}} = \texttt{D[rr, \{(a a, b b, c c)\}];}
\text{Out[99]=} \dot{\texttt{d d d}} = \texttt{D[dd, \{(a a, b b, c c)\}];}
\text{Out[100]=} \{\texttt{rr, dd, d d d} \}/.\{\texttt{a a} \rightarrow 1., \texttt{b b} \rightarrow 1, \texttt{c c} \rightarrow 1\}
\text{Out[101]=} (-0.682328, 0.132545, 0.284693, -0.417238), \{-0.124475, -0.156754, 0.148685\},
\{-0.156754, -0.099123, -0.0288157\}, \{0.148685, -0.0288157, 0.297369\})
inserts a code that searches for a numerical root of a system of nonlinear, equations $Q(h)$. The iterative procedure starts from the point $h_0$ (predictor) and it uses the standard Newton–Raphson corrector

Given pure function must return for the given input parameters $\{h, h_n, \lambda\}$ the following vector:

```
{ Q, data }
```

, $data$ - arbitrary data

Function SMSFindRoot returns a vector:

```
{h, Null, True, data}
```

for the given input parameters $\{h, h_n, \lambda\}$ the following vector:

```
{Q, data}
```

, $data$ - arbitrary data

Option name default value

<table>
<thead>
<tr>
<th>option name</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;MaxIterations&quot;</td>
<td>30</td>
</tr>
<tr>
<td>&quot;Tolerance&quot;</td>
<td>$10^{-10}$</td>
</tr>
</tbody>
</table>
| "ImplicitVaria-
| "bility"         | \{\}              |
| "Derivatives"    | 0                 |
| "SkipIterations"| False             |

Options for the solution of system of equations.
Example

Generate and verify the MathLink program that returns solution to the system of nonlinear equations:

\[\begin{align*}
\Phi &= axy + x^3 = 0 \\
-a - xy^2 &= 0
\end{align*}\]

where \(x\) and \(y\) are unknowns and \(a\) is parameter.

```mathematica
In[22] := << AceGen`;

smsInitialize["test", "Environment" -> "MathLink"];

smsModule["test", Real \[x$$, y$$, a$$, tol$$, dx$$, dy$$], Integer \[n$$] ,

"Input" \rightarrow \{x$$, y$$, a$$, tol$$, n$$\},

"Output" \rightarrow \{x$$, y$$, dx$$, dy$$\}];

(x0, y0, a, e) \rightarrow SMSReal[\{x$$, y$$, a$$, tol$$\}];

nmax \rightarrow SMSInteger[\[n$$] ;

\{(xc, yc), dx, conv, niter, data\} \rightarrow SMSFindRoot[

Function[{h},

\{\{axy + x^3, a - x y^2\}, Null\}

\], \{x0, y0\}, "MaxIterations" \rightarrow nmax

"Tolerance" \rightarrow e,

"Derivatives" \rightarrow 1

"ImplicitVariables" \rightarrow {a}
]
;

SMIF[Not[conv], SMSPrintMessage["no convergence"]];

SMSExport[\{xc, yc\}, \{x$$, y$$\}];

SMSExport[dx // Flatten, \{dx$$, dy$$\}];

SMSWrite[];

File: test.c  Size: 2858  Time: 0

<table>
<thead>
<tr>
<th>Method</th>
<th>No. Formulae</th>
<th>No. Leafs</th>
</tr>
</thead>
<tbody>
<tr>
<td>test</td>
<td>26</td>
<td>243</td>
</tr>
</tbody>
</table>
```

Here the MathLink program test.exe is build from the generated source code and installed so that functions defined in the source code can be called directly from Mathematica. (see also SMSInstallMathLink)

```mathematica
In[32] := SMSInstallMathLink[]

Out[32]= {SMSSetLinkOption[test, \{i\_Integer, j\_Integer\}],

SMSLinkNoEvaluations[test], test[x\_, y\_, a\_, tol\_, dx\_, dy\_] \rightarrow

\{\{axy + x^3, a - x y^2\}, Null\}

\], \{x0, y0\}, "MaxIterations" \rightarrow nmax

"Tolerance" \rightarrow e,

"Derivatives" \rightarrow 1

"ImplicitVariables" \rightarrow {a}
]

In[33] := test[1.9, -1.2, 3., 0.0001, 10]

Out[33]= \{1.93318, -1.24573, 0.386636, -0.0830487\}
```

For the verification of the generated code the solution calculated by the build in function is compared with the solution calculated by the generated code.
```math
In[34]:= x =.; y =.; a =.;
    sol = Solve[{a x y + x^3 == 0, a - x y^2 == 0}, {x, y}]
Out[34]= 
    {{x -> a^(3/5), y -> -a^(1/5)},
     {x -> -(1)^(3/5) a^(3/5), y -> -(1)^(1/5) a^(1/5)},
     {x -> -(1)^(4/5) a^(3/5), y -> -(1)^(3/5) a^(1/5)},
     {x -> -(1)^(2/5) a^(3/5), y -> -(1)^(4/5) a^(1/5)}}
In[36]:= (sol[[1]], D[sol[[1]], a]) /. a -> 3.
Out[36]= 
    {{x -> 1.93318, y -> -1.24573},
     {0 -> 0.386636, 0 -> -0.0830487}}
```

### Numerical root finding of a system of nonlinear, path dependent, parameterized equations

```math
\text{SMSFindRoot}[
    \text{Function}[
      \{h, h_n, \lambda\}, Q(h, h_n, \lambda)
    ],
    h_0, h_s
]
```

Inserts a code that searches for a numerical root of a system of nonlinear, path dependent, parameterized equations \( Q(h, h_n, \lambda) \) where parameter \( \lambda \in [0,1] \), \( h_s = h_{\lambda=0} \) is known solution at \( \lambda = 0 \). The iterative procedure starts from the point \( h_0 \) (predictor) and it uses the standard Newton–Raphson corrector. If convergence is not achieved in one step, then the algorithm splits the parameter interval \( \lambda \in [0,1] \) into several sub-steps adaptively. The procedure is terminated if the number of sub-steps exceeds “MaxSubsteps”.

Given pure function must return for the given input parameters \( \{h, h_n, \lambda\} \) the following vector:

- \( Q \ldots \) vector of equations
- \( \text{True} \) if for the given sub-step the convergence was obtained
- \( \text{True} \) if the solution is acceptable from the physical point of view (if False is given then the step cutting is performed)
- arbitrary data

```math
\text{Function SMSFindRoot returns a vector:}

\begin{align*}
    \text{h} - & \text{calculated root } (Q(h)=0), \\
    \frac{\partial h}{\partial r} - & \text{derivatives of the root with respect to given set of implicit variables,} \\
    \text{True} \text{ if convergence was obtained and False if not,} \\
    \text{actual number of sub-steps performed,} \\
    & \text{the last arbitrary data returned by given pure function}
\end{align*}
```
<table>
<thead>
<tr>
<th>option name</th>
<th>default value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;MaxIterations&quot;</td>
<td>30</td>
<td>maximum number of iterations</td>
</tr>
<tr>
<td>&quot;Tolerance&quot;</td>
<td>$10^{-10}$</td>
<td>accuracy of the solution</td>
</tr>
<tr>
<td>&quot;ImplicitVariables&quot;</td>
<td>{}</td>
<td>a set of variables $r$ on which the equations depend,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thus $Q(h(r), h_n, \lambda, r)$ for which derivatives of the root $\frac{dh}{dr}$ have to be calculated</td>
</tr>
<tr>
<td>&quot;Derivatives&quot;</td>
<td>0</td>
<td>$0 \Rightarrow$ find root only, <strong>CURRENTLY NOT SUPPORTED!</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$1 \Rightarrow$ find root together with the first</td>
</tr>
<tr>
<td></td>
<td></td>
<td>derivatives of the root with respect to implicit variables $\frac{dh}{dr}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2 \Rightarrow$ <strong>CURRENTLY NOT SUPPORTED!</strong></td>
</tr>
<tr>
<td>&quot;MinSubsteps&quot;</td>
<td>1</td>
<td>minimum number of sub-steps</td>
</tr>
<tr>
<td>&quot;MaxSubsteps&quot;</td>
<td>10</td>
<td>maximum number of sub-steps</td>
</tr>
<tr>
<td>&quot;SkipIterations&quot;</td>
<td>False</td>
<td>assume that predictor $h_0$ is the correct solution and calculate the implicit derivatives $\frac{dh}{dr}$</td>
</tr>
</tbody>
</table>

Options for the solution of parameterized system of equations.
Mechanics of Solids

- **SMSLameToHooke[λ, G]** transform Lame’s constants $\lambda$, $G$ to Hooke’s constants $E$, $\nu$
- **SMSHookeToLame[E, $\nu$]** transform Hooke’s constants $E$, $\nu$ to Lame’s constants $\lambda$, $G$
- **SMSHookeToBulk[E, $\nu$]** transform Hooke’s constants $E$, $\nu$ to shear modulus $G$ and bulk modulus $K$
- **SMSBulkToHooke[G, K]** transform shear modulus $G$ and bulk modulus $K$ to Hooke’s constants $E$, $\nu$

Transformations of mechanical constants in mechanics of solids.

- This transforms Lame’s constants $\lambda$, $G$ to Hooke’s constants $E$, $\nu$. **No simplification is preformed!**

```
In[16]:= SMSLameToHooke[λ, G] // Simplify
Out[16]= \[
\begin{array}{c}
\frac{G (2 G + 3 \lambda)}{G + \lambda}, \\
\frac{\lambda}{2 (G + \lambda)}
\end{array}
\]
```

- **SMSPlaneStressMatrix[E, $\nu$]** linear elastic plane strain constitutive matrix for the Hooke’s constants $E$, $\nu$
- **SMSPlaneStrainMatrix[E, $\nu$]** linear elastic plane stress constitutive matrix for the Hooke’s constants $E$, $\nu$

Find constitutive matrices for the linear elastic formulations in mechanics of solids.

- This returns the plane stress constitutive matrix. **No simplification is preformed!**

```
In[228]:= SMSPlaneStressMatrix[e, $\nu$] // MatrixForm
Out[228]= \[
\begin{pmatrix}
e & e & 0 \\
e & e & 0 \\
0 & 0 & 2 (1 + \nu)
\end{pmatrix}
\]
Advanced Features
User Defined Functions

Contents

◼ General Description
◼ SMSCall
◼ Examples
◼ Intrinsic user function 1: Scalar function exists but has different syntax in source code language
◼ Intrinsic user function 2: Scalar function with closed form definition of the function and its derivatives
◼ User AceGen module 1: Definition of the user subroutine and first derivatives
◼ User AceGen module 2: Definition of the user subroutine and first and second derivatives
◼ User external subroutines 1: Source code file added to the generated source code
◼ User external subroutines 2: Header file added to the generated source code file
◼ Embedded sequence with SMSVerbatim command: User defined source code is embedded into generated source code
◼ MathLink program with modules

General Description

The user can define additional output formats for standard Mathematica functions or new functions. The advantage of properly defined function is that allows optimization of expressions and automatic differentiation. In general there are several types of user defined functions supported in AceGen:

◼ **Intrinsic user function** is scalar function of scalar input parameters with closed form definition of the function and its derivatives that can be expressed with the existing Mathematica functions. The definition of the intrinsic user function becomes an integral part of Mathematica and AceGen. Thus, a full optimization of the derived expressions and unlimited number of derivatives is provided.

◼ **User defined AceGen module** is arbitrary subroutine with several input/output parameters of various types generated with AceGen within the same AceGen session as the main module. All AceGen modules generated within the same AceGen session are automatically written into the same source file and the proper definitions and declarations of input/output parameters are also included automatically. The user AceGen module can be called from the main module using the SMSCall command. Optimization of expressions is performed only within the module. Differentiation is not supported unless derivatives are also derived and exported to main module.

◼ **User defined external subroutines**: External subroutines are arbitrary subroutines with several input/output parameters of various types written in source code language and provided by the user. The user defined external subroutines can be called from the main module using the SMSCall command in a same way as user AceGen module. The "System"->False option has to be included in order to signify that the subroutine has not been generated by AceGen. For the generation of the final executable we have two options:

  - The source code file can be incorporated into the generated source code file using the "Splice" option of the SMSWrite command. The original source code file of the user subroutine is not needed for the compilation.
  - Alternatively one can include only the header file containing the declaration of the function accordingly to the chosen source code language using the "IncludeHeaders" option of the SMSWrite command. The original source code of the external subroutine has to be compiled separately and linked together with the AceGen generated file.

◼ **Embedded sequences**: User can embed the sequence of the code in chosen language with the use of SMSVerbatim command.

See also: Elements that Call User External Subroutines.
SMSCall

\[ \text{sc} = \text{SMSCall}["\text{sub"}, p_1, p_2, \ldots] \]

returns auxiliary variable sc that represents the call of external subroutine sub with the given set of input and output parameters

<table>
<thead>
<tr>
<th>option</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ArgumentsByValue&quot;</td>
<td>False</td>
<td>By default all arguments are passed to subroutine by reference. This can be changed with &quot;ArgumentsByValue&quot;-&gt;True option. ONLY FOR ADVANCED USERS!!</td>
</tr>
<tr>
<td>&quot;Position&quot;</td>
<td>&quot;Current&quot;</td>
<td>Regulates how the &quot;export&quot; object is treated during the code optimisation. &quot;Current&quot; (\Rightarrow) the object remains at the position in the program where it was called &quot;Optimal&quot; (\Rightarrow) optimal position is determined based on the dependency of exp and ext on other variables &quot;Automatic&quot; (\Rightarrow) &quot;Optimal&quot;</td>
</tr>
<tr>
<td>&quot;Subordinate&quot;-&gt;</td>
<td>{}</td>
<td>list of auxiliary variables that represent control structures (e.g. SMSVerbatim, SMSExport) that have to be executed before the evaluation of the current expression</td>
</tr>
<tr>
<td>&quot;System&quot;-&gt;</td>
<td>True</td>
<td>the subroutine that is called has been generated by AceGen</td>
</tr>
</tbody>
</table>

Options of the SMSCall function.

The name of the subroutine can be arbitrary string. The SMSCall command inserts into the generated source code the call to the external subroutine "sub" with the given set of input and output parameters. The input parameters can be arbitrary expressions. The input and output arguments are always passed to functions by reference (pointers not values!). The input and output parameters are defined as local variables of the master subroutine unless they are I/O arguments of the master subroutine. No dynamic allocation is performed inside the subroutine, thus arrays must have known constant length. The only exceptions are I/O parameters that are at the same time also I/O parameters of the master subroutine.

| 12.34 | real value |
| 1234  | integer value is considered to be real input parameter |
| SMSInteger[1234] | SMSInteger force creation of integer input parameter (also SMSInteger[n$]) |
| expr  | the type (real or integer) of arbitrary scalar expression is determined automatically |
| {1,2,3,4} | real array with constant length input parameter |
| Real[a$[3]] | real array with constant length input parameter |
| (the values should be set before the call by SMSExport (e.g. SMSExport[1,2,3,a$])) | |
| Real[a$["*"] | real array with variable length input parameter |
| (available only when a$ is also input parameter of the master subroutine!) | |
| "type"[a$] | arbitrary type |

Some examples of declarations of input parameters.

| Integer[n$] | integer output parameter |
| Real[n$] | real output parameter |
| Real[a$[3]] | real array with constant length output parameter |
| Real[a$["*"]] | real array with variable length output parameter |
| (available only when a$ is also input parameter of the master subroutine!) | |

Some examples of declarations of output parameters.
Some examples of declarations of arbitrary type and form I/O parameters.

The use of output parameters later in the program and their declaration should follow AceGen rules for the declaration and use of external variables as described in chapter Symbolic-Numeric Interface (e.g. \texttt{x}=\texttt{SMSReal}[x$$[5]]\texttt{,"Subordinate"}\to\texttt{sc}, \texttt{ni=SMSInteger}[n$$[5]]\texttt{,"Subordinate"}\to\texttt{sc}, \texttt{b=SMSLogical}[b$$[5]]\texttt{,"Subordinate"}\to\texttt{sc}]. The proper order of evaluation of expressions is assured by the "Subordinate"\to\texttt{sc} option where the parameter \texttt{sc} is an auxiliary variable that represents the call of external subroutine. Additionally, the partial derivatives of output parameters with respect to input parameters can be defined by the option "Dependency" as described in \texttt{SMSFreeze}.

**Examples**

**Intrinsic user function 1:** Scalar function exists but has different syntax in source code language

```plaintext
In[52]:<AceGen`
SMSInitialize["test", "Language" \to "Fortran"];

\textbf{This is an additional definition of output format for function tangent.}

In[54]:= SMSAddFormat[
    Tan[i_] \to Switch[SMSLanguage, "Mathematica", "Tan"[i], "Fortran", "dtan"[i], "C", "tan"[i]]
];

In[55]:= SMSModule["sub1", Real[x$$, y$$[5]]];
   x = SMSReal[x$$];
   SMSExport[Tan[x], y$$[1]];
   SMSWrite[];
```

```
\begin{verbatim}
File: test.f Size: 786
Methods No.Formulae No.Leafs
sub1 1 7
\end{verbatim}
```

The final code can also be formatted by the "Substitutions" option of the \texttt{SMSWrite} command.

```plaintext
In[59]:= FilePrint["test.f"]
```

```plaintext
!******************************************************************************!
!* User: USER
!* Evaluation time : 0 s Mode : Optimal
!* Number of formulae : 1 Method: Automatic
!* Subroutine : sub1 size :7
!* Total size of Mathematica code : 7 subexpressions
!* Total size of Fortran code : 203 bytes
!******************************************************************************!
SUBROUTINE sub1(v,x,y)
IMPLICIT NONE
include 'sms.h'
DOUBLE PRECISION v(5001),x,y(5)
y(1)=dtan(x)
END
```

IMPORTANT: Differentiation is not supported for the User AceGen module and User external subroutines unless derivatives are derived within the user subroutine explicitly and exported into the main module through the output parameters of the module (see \texttt{AceGenPrint.nb}).
Intrinsic user function 2: Scalar function with closed form definition of the function and its derivatives

- This adds alternative definition of Power function MyPower[x, y] ≡ x^y that assumes that x>0 and
  \[ D[MyPower[x,y],x] = y \frac{MyPower[x,y]}{x}, \]
  \[ D[MyPower[x,y],y] = MyPower[x,y] \log[x]. \]

- This is an additional definition of output format for function MyPower.
- Here the derivatives of MyPower with respect to all parameters are defined.
- Here is defined the numerical evaluation of MyPower with the \( p \)-digit precision.

In\[95\]:= 
N[MyPower[i\_\_, j\_\_], p\_] := i^j;

In\[97\]:= 
SMSModule[\"sub1\", Real\[x\$$, y\$$, z\$$\]];
\[x\] \[\[\to\]\] SMSReal[x\$$];
\[y\] \[\[\to\]\] SMSReal[y\$$];
SMSExport[SMSE[MyPower[x, y], x], z\$$];

In\[100\]:= 
SMSWrite[];

In\[101\]:= 
File:\ test.c  Size: 729

Methods No.Formulae No.Leafs
sub1 1 22

In\[102\]:= 
FilePrint[\"test.c\"]

/* **********************************************************************
 * AceGen 2.502 Windows (5 Nov 10) *
 * Co. J. Korelc 2007 5 Nov 10 11:14:36 *
 * **********************************************************************

User : USER
Evaluation time : 0 s Mode : Optimal
Number of formulae : 1 Method: Automatic
Subroutine : sub1 size :22
Total size of Mathematica code : 22 subexpressions
Total size of C code : 167 bytes
*/

#include "sms.h"

void sub1(double *v[5001], double *x), double *y), double *z))
{
  (*z) = (\(+y\) *Power((\(+x\),(\(+y\))))/(\(+x\));
};
**User AceGen module 1: Definition of the user subroutine and first derivatives**

In[324]:= \<<
    AceGen;
    SMSInitialize["test", "Language" -> "C"];

This generates **user AceGen module** \( f = \sin(a_1 x + a_2 x^2 + a_3 x^3) \) with an input parameter \( x \) and constants \( a[3] \) and the output parameters \( y = f(x) \) and first \( \frac{dy}{dx} \) derivatives.

In[326]:= SMSModule["f", Real[x$$, a$$[3]], y$$, dy$$];

This generates subroutine \( m/f \) that calls subroutine \( f \).

In[333]:= SMSModule["mf", Real[w$$, r$$]]; w$$ ⊢ SMSReal[w$$];

The SMSReal is used here to import the output parameters of the subroutine to AceGen. The option "Subordinate" is necessary to ensure that the call to \( f \) is executed before the output parameters are imported.

In[337]:= dfdz ⊢ SMSReal[dy$$, "Subordinate" -> fo];

The "Dependency" -> \( \{\sin(x, dy)\} \) option defines that output parameter \( y \) depends on input parameter \( x \) and defines partial derivative of \( y \) with respect to input parameter \( x \). By default all first partial derivatives of output parameters with respect to input parameters are set to 0.

In[339]:= f ⊢ SMSReal[y$$, "Subordinate" -> fo, "Dependency" -> \( \{z, dfdz\}\)];

First derivatives are derived and displayed here.

In[340]:= dw = SMSD[f, w]; SMSRestore[dw, "Global"]

Second derivatives are derived and displayed here. It is obvious that the second derivatives are **incorrect**, due to the lack of proper definition of the second derivative of \( f \) with respect to \( z \).

In[341]:= ddw = SMSD[ddw, w]; SMSRestore[ddw, "Global"]

Out[341]= 2 ddw
User AceGen module 2: Definition of the user subroutine and first and second derivatives

- This generates user AceGen module \( f = \sin(a_1 x + a_2 x^2 + a_3 x^3) \) with an input parameter \( x \) and constants \( a[3] \) and the output parameters \( y = f(x) \) and first \( dy = \frac{df}{dx} \) and second \( ddy = \frac{d^2f}{dx^2} \) derivatives.
<<AceGen`;
SMSInitialize["test", "Language" -> "C"];
SMSModule["f", Real[x$$, a$$[3], y$$, dy$$, ddy$$]]; x = SMSReal[x$$];
{a1, a2, a3} = SMSReal[Table[a$$[i], {i, 3}]]; y = Sin[a1 x + a2 x^2 + a3 x^3];
dy = SMSD[y, x]; ddy = SMSD[y, x];
SMSExport[{y, dy, ddy}, {y$$, dy$$, ddy$$}];
SMSModule["main", Real[w$$, r$$]]; w = SMSReal[w$$]; z = w^2;
fo = SMSCall["f", z, {1/2, 1/3, 1/4}, Real[y$$], Real[dy$$], Real[ddy$$]];

dfdz2 = SMSReal[ddy$$, "Subordinate" -> fo];
dfdz = SMSReal[dy$$, "Subordinate" -> fo, "Dependency" -> {z, dfdz2}];
f = SMSReal[y$$, "Subordinate" -> fo, "Dependency" -> {z, dfdz}];
dw = SMSD[f, w];
ddw = SMSD[{dw, w}];

Both first and second derivatives are correct.

In[364]:= SMSRestore[{dw, ddw}, "Global"]

Out[364]= {2 dfdz w, 2 dfdz + 4 dfdz^2}

In[365]:= SMSExport[{dw, ddw}, {dy$$, ddy$$}];

In[366]:= SMSWrite[];

File: test.c  Size: 1397

<table>
<thead>
<tr>
<th>Methods</th>
<th>No. Formulae</th>
<th>No. Leafs</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>7</td>
<td>91</td>
</tr>
<tr>
<td>main</td>
<td>6</td>
<td>81</td>
</tr>
</tbody>
</table>
FilePrint["test.c"]

```c
#include "sms.h"

/*******************
SUBROUTINE*******************/

void f(double v[5001],double (*x),double a[3],double (*y),double (*dy),double (*ddy))
{
    v[28]=2e0*a[1]+(*x);
    v[27]=a[2]*Power((*x),3);
    v[26]=a[0]+(*x);
    v[25]=Power((*x),2);
    (*y)=sin(v[6]);
    (*dy)=v[8];
    (*ddy)=v[8];
}

/*******************
SUBROUTINE*******************/

void main(double v[5001],double (*w),double (*r))
{
    double ddy;double dy;double v01;double y;double v02[3];
    v[29]=2e0+(*w);
    v01=Power((*w),2);
    v02[0]=0.5e0;
    v02[1]=0.3333333333333333e0;
    v02[2]=0.25e0;
    f(&v[5009],&v01,v02,&y,&dy,&ddy);
    v[18]=dy;
    (*dy)=v[18]+v[29];
    (*ddy)=2e0+v[18]+ddy+(v[29]+v[29]);
}
```

User external subroutines 1: Source code file added to the generated source code

Lets create the C source file "Energy.c" with the following contents
This generates subroutine Stress with an input parameter right Cauchy-Green tensor and first and second derivative of the strain energy with respect to the input parameters $I_1$ and $I_2$. Subroutine Energy calculates the strain energy and the $C$ header file "Energy.h" with the following contents:

```c
void Energy (double *I1p, double *I3p, double *C1p, double *C2p, double *C3p, double *pi, double dp[2], double ddp[2][2])
{
    double I1, I3, C1, C2, C3;
    I1 = *I1p; I3 = *I3p; C1 = *C1p; C2 = *C2p; C3 = *C3p;
    *pi = (C2*(-3 + I1))/2. + (C1*(-1 + I3 - log (I3)))/4. - (C1*log (I3))/2.;
    dp[0] = C2/2.;
    dp[1] = (C1*(1 - 1/I3))/4. - C2/(2.*I3);
    ddp[0][0] = 0;
    ddp[0][1] = 0;
    ddp[1][0] = 0;
    ddp[1][1] = C1/(4.*I3*I3) + C2/(2.*I3*I3);
}
```

and the C header file "Energy.h" with the following contents:

```c
void Energy (double *I1p, double *I3p, double *C1p, double *C2p, double *C3p, double *pi, double dp[2], double ddp[2][2])
```

Subroutine Energy calculates the strain energy $\Pi(I_1,I_3)$ where $I_1$ and $I_3$ are first and third invariant of the right Cauchy-Green tensor and first and second derivative of the strain energy with respect to the input parameters $I_1$ and $I_2$.

This generates subroutine Stress with an input parameter right Cauchy-Green tensor $C$ that returns Second Piola-Kirchoff stress tensor $S$. Stress tensor corresponds to the arbitrary strain energy function given by source code file Energy.c. The user supplied source code is added to the generated source code file.

```c
<<AceGen
;
SMSInitialize["test", "Language"->"C"];
SMSModule["Stress", Real[C$$[3, 3], S$$[3, 3], C$$, C$$, C$$]];
{C1, C2, C3} = SMSReal[{{C$$, C$$, C$$}}];
C = SMSReal[Table[C$$[i, j], (i, 3), (j, 3)]];  
C[2, I] = C[1, 2]; C[3, 1] = C[1, 3]; C[3, 2] = C[2, 3];
{I1, I3} = {Tr[C], Det[C]};
p=call = SMSCall["Energy", I1, I3, C1, C2, C3,
                    Real[pi$$], Real[dp$$[2]], Real[ddp$$[2, 2]], "System"->False];
              ddp = SMSReal[Table[ddp$$[i, j], (i, 2), (j, 2)], "Subordinate"->p=call];
              dp = SMSReal[Table[dp$$[i, 1], (i, 2)], "Subordinate"->p=call, "Dependency"->({I1, I3}, ddp)];
Pi = SMSReal[pi$$, "Subordinate"->p=call, "Dependency"->({I1, dp[1][1]}, {I3, dp[2][2]}));
S = 2 SMSD[Pi, ci, "Symmetric"->True];
SMSExport[S, S$$];
SMSWrite["Splice"->{"Energy.c"}];
```

File: test.c  Size: 2117
Methods No.Formulae No.Leafs
Stress 21 361
double I1, I3, C1, C2, C3;
I1 = \texttt{+I1p}; I3 = \texttt{+I3p}; C1 = \texttt{+C1p}; C2 = \texttt{+C2p}; C3 = \texttt{+C3p};
\texttt{+pi} = \frac{(C2+(-3 + I1))/2. + (C1-(-1 + I3 - \log (I3))/4. - (C2+\log (I3))/2.}{C2/2.};
dp[0] = C2/2.;
dp[1] = (C1/(1 - I13))/4. - C2/(2.*I3);
ddp[0][0] = 0;
ddp[1][0] = 0;
ddp[1][1] = C1/(4.*I3*I3) + C2/(2.*I3*I3);

void Energy(double *I1p, double *I3p, double *C1p, double *C2p, double *pi, double *dp, double double ddp[2][2])
{
    double I1, I3, C1, C2, C3;
    I1 = \texttt{+I1p}; I3 = \texttt{+I3p}; C1 = \texttt{+C1p}; C2 = \texttt{+C2p}; C3 = \texttt{+C3p};
    \texttt{+pi} = \frac{(C2+(-3 + I1))/2. + (C1-(-1 + I3 - \log (I3))/4. - (C2+\log (I3))/2.}{C2/2.};
    dp[0] = C2/2.;
    dp[1] = (C1/(1 - I13))/4. - C2/(2.*I3);
    ddp[0][0] = 0;
    ddp[1][0] = 0;
    ddp[1][1] = C1/(4.*I3*I3) + C2/(2.*I3*I3);
}

 chemin "sms.h"

#include "sms.h"

void Stress(double v[5001], double C[3][3], double S[3][3], double (*C1), double (*C2), double (*C3))
{
    double pi;double v01;double v02;double v03;double v04;double v05;
    double ddp[2][2];double *dp;
    v[46]=2e0+C[0][0]+C[1][2];
    v[45]=2e0+C[0][1];
    v[44]=2e0+C[0][2];
    v[43]=(C[2][2]+v[45]);
    v[42]=Power(C[1][2],2);
    v[41]=C[1][2]+v[44];
    v[40]=Power(C[0][2],2);
    v[39]=Power(C[0][1],2);
    v[47]=-C[0][0]-C[1][1]-v[39];
    v[01]=C[0][0]-C[0][1]-C[2][2];
    v[02]=-(C[1][1]+v[40])-C[0][1]-v[41]-C[0][0]+v[42]-C[2][2]+v[47];
    v[03]=(-C1);
    v[04]=(-C2);
    v[05]=(-C3);

    Energy(&v01,&v02,&v03,&v04,&v05,\texttt{+pi},\texttt{+dp},\texttt{+ddp});
    v[25]=dp[0];
    v[26]=dp[1];
    v[31]=v[26]+v[41]+v[43];
    S[0][0]=2e0+(v[25]+v[26]+(C[0][0]+C[2][2]-v[40]));
    S[0][1]=v[31];
    S[0][2]=v[32];
    S[1][0]=v[31];
    S[1][1]=2e0+(v[25]+v[26]+(C[0][0]+C[2][2]-v[40]));
    S[1][2]=v[35];
    S[2][0]=v[32];
    S[2][1]=v[35];
}
User external subroutines 2: Header file added to the generated source code file

Previous example is here modified in a way that only the header file "Energy.h" is added to the generated source code file.

```plaintext
<<AceGen`;
SMSInitialize["test", "Language" -> "C"];
SMSModule["main", Real[C$$[3, 3], S$$[3, 3], C1$$, C2$$, C3$$]];
(C1, C2, C3) -> SMSReal[[C1$$, C2$$, C3$$]];
C = SMSReal[Table[C$$[i, j], {i, 3}, {j, 3}]];
C[[2, 1]] = C[[1, 2]]; C[[3, 1]] = C[[1, 3]]; C[[3, 2]] = C[[2, 3]];
{I1, I3} -> {Tr[C], Det[C]};
pcall = SMSCall["Energy", I1, I3, C1, C2, C3,
  Real[pi$$], Real[dp$$[2]], Real[ddp$$[2, 2]], "System" -> False];
dp = SMSReal[Table[dp$$[i, j], {i, 2}, {j, 2}], "Subordinate" -> pcall];
π = SMSReal[π$$, "Subordinate" -> pcall, "Dependency" -> {{I1, I3}, ddp}];
Π ⊢ SMSReal[π$$, "Subordinate" -> pcall, "Dependency" -> {{I1, dp[[1]]}, {I3, dp[[2]]}}];
S ⊢ 2 SMSD[Π, π, "Symmetric" -> True];
SMSExport[S, S$$];
SMSWrite["IncludeHeaders" -> {"Energy.h"}];
```

### File: test.c  Size: 1662

<table>
<thead>
<tr>
<th>Method</th>
<th>No. Formulae</th>
<th>No. Leafs</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>21</td>
<td>361</td>
</tr>
</tbody>
</table>
Invariants $I_1$ and $I_2$. Within the code, the embedded part calculates the strain energy. Here is the part of the previously defined subroutine Energy.c embedded directly into the generated source code. The Verbatim Embeded sequence with

```c
#include "Energy.h"

double ddp
```

Here is the part of the previously defined subroutine Energy.c embedded directly into the generated source code. The Verbatim command is used for that purpose where an additional option "Declaresymbol" adds declarations of the local variables that are used within the code. The embedded part calculates the strain energy $\Pi(I_1, I_3)$ and first derivative of the strain energy with respect to the invariants $I_1$ and $I_2$. 

```c
void main(double v[5001], double C[3][3], double S[3][3], double (*C1), double (*C2), double (*C3))
{
    double pi; double v01; double v02; double v03; double v04; double v05;
    double ddp[2][2]; double dp[2];
    v[46] = 2e0 * C[0][0] * C[1][2];
    v[45] = 2e0 * C[0][1];
    v[44] = 2e0 * C[0][2];
    v[43] = -(C[2][2] + v[45]);
    v[42] = Power(C[1][2], 2);
    v[41] = C[1][2] * v[44];
    v[40] = Power(C[0][2], 2);
    v[39] = Power(C[0][1], 2);
    v[47] = C[0][0] * C[1][1] - v[39];
    v01 = C[0][0] * C[1][1] + C[2][2];
    v02 = -(C[1][1] + v[40]) + C[0][1] + v[41] + C[0][0] + v[42] + C[2][2] + v[47];
    v03 = (*C1);
    v04 = (*C2);
    v05 = (*C3);
    Energy(&v01, &v02, &v03, &v04, &v05, &pi, dp, ddp);
    v[25] = ddp[0][0];
    v[26] = ddp[1][1];
    v[31] = v[26] * (v[41] + v[43]);
    S[0][0] = 2e0 * (v[25] + v[26] + (C[1][1] + C[2][2] - v[42]));
    S[0][1] = v[31];
    S[0][2] = v[32];
    S[1][0] = v[31];
    S[1][1] = 2e0 * (v[25] + v[26] + (C[0][0] + C[2][2] - v[40]));
    S[1][2] = v[35];
    S[2][0] = v[32];
    S[2][1] = v[35];
}
```

Embedded sequence with SMSVerbatim command: User defined source code is embedded into generated source code
<<AceGen`

SMSInitialize["test", "Language" -> "C"]

SMSModule["Stress", Real[C$$[3, 3], S$$[3, 3], C1$$, C2$$, C3$$]];

C, C2, C3] -> SMSReal[{C1$$, C2$$, C3$$}]

c-> SMSReal[Table[C$$[i, j], (i, 3), (j, 3)]];

c[2, 1] = c[1, 2]; c[3, 1] = c[1, 3]; c[3, 2] = c[2, 3];

{I1, I3} + {Tr[C], Det[C]};

code = SMSVerbatim[
 "I1c" = I1, "I3c" = I3, ";nC1c=" C1, "C2c=" C2, ";C3c=" C3, ";",
 "np1 = (C2c*(-3 + I1))/2. + (C1c*(-1 + I3c - log(I3c))) /4. - (C2c*log(I3c))/2.;

dp[0] = C2c/2.;

dp[1] = (C1c*(1 - 1/I3c))/4. - C2c/(2.*I3c)"
 , "DeclareSymbol" -> {Real[C1$$, C2$$, C3$$, I1$$, I3$$, pi$$, dp$$[2]]]};

dp -> SMSReal[Table[dp$$[i], (i, 2)], "Subordinate" -> code];

Pi -> SMSReal[pi$$, "Subordinate" -> code, "Dependency" -> {{I1, dp[[1]]}, {I3, dp[[2]]}}];

S = 2 SMSD[Pi, C, "Symmetric" -> True];

SMSExport[S, S$$];

SMSWrite[];

<table>
<thead>
<tr>
<th>File:</th>
<th>test.c</th>
<th>Size: 1703</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods No.</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>Formulae No.</td>
<td>160</td>
<td>350</td>
</tr>
<tr>
<td>Stress</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
This generates the user AceGen module "f" that defines function
\[ f(x) = \sum_{i=1}^{n} a_i x^i \]
with an input parameters \( x, n \), and a vector of
coefficients \( a \) and output parameters \( y = f(x) \), first derivative \( dy = \frac{\partial f}{\partial x} \) and second derivative \( d^2 y = \frac{\partial^2 f}{\partial x^2} \).
Definition of the MathLink module "mlinkA" that evaluates function $y = f(e^w)$ and returns second derivative $ddw = \frac{\partial^2 f(e^w)}{\partial w^2}$ for a constant set of coefficients $\alpha = \{1/2, 1/3, 1/4\}$.

Definition of the MathLink module "mlinkB" that evaluates function $y = f(e^w)$ and returns second derivative $ddw = \frac{\partial^2 f(e^w)}{\partial w^2}$ for arbitrary set of coefficients given as an argument of "mlinkB" function.
Test with MMA for \( w = 2.5 \)

\begin{verbatim}
In[136]:= w = 2.5; x = Exp[w]; D[{1/2, 1/3, 1/4}.{1, x, x^2}, w, w] /. w -> 2.5
Out[136]= 152.474
\end{verbatim}

"mlinkA" and "mlinkB" give correct results.

\begin{verbatim}
In[137]:= mlinkA[2.5]
Out[137]= 152.474

In[138]:= mlinkB[2.5, {1/2, 1/3, 1/4}] // N, 3
Out[138]= 152.474
\end{verbatim}

Results is correct even when the length of \( a \) is more than 3, provided that only 3 coefficients are used.

\begin{verbatim}
In[139]:= mlinkB[2.5, {1/2, 1/3, 1/4, 1/5, 1/6}] // N, 3
Out[139]= 152.474
\end{verbatim}
Arrays

Contents

- General Description
  - SMSArray
- Manipulating Arrays
  - SMSPart
  - SMSReplacePart
  - SMSArrayLength
  - SMSSum

General Description

One can use all Mathematica built-in functions or any of the external Mathematica packages to perform those operations. Mathematica only supports integer indices (e.g. in expression \( M\{\text{index}\} \)) index has to have integer value assigned at the time of evaluation) and the array \( M \) has to have explicit value assigned before the evaluation (e.g. \( M=\{1,2,3,4\} \)). If either \( M \) has no value assigned yet or the index is not an integer number the matrix operations cannot be performed in Mathematica directly. The result of matrix operation in Mathematica is always a closed form solution for each component of the array. After the matrix operation is performed, one can optimize the result by using AceGen optimization capabilities. For each component of the array a new auxiliary variable is created (if necessary) that stores a closed form solution of the specific component (see e.g. Introduction). For example:

\[
\begin{align*}
\text{In[44]} &= << \text{AceGen}'; \\
& \quad \text{SMSInitialize}['test', 'Language' -> "C"]; \\
& \quad \text{SMSModule}['test', \text{Real}[x$$[5]], \text{Integer}[i$$]]; \\
& \quad x \mapsto \text{SMSReal}[x$$]; \\
& \quad i \mapsto \text{SMSInteger}[i$$];
\end{align*}
\]

Here a standard Mathematica vector is defined and a vector of new auxiliary variables is created to store the result.

\[
\begin{align*}
\text{In[45]} &= X \mapsto \text{Table}[\text{SMSReal}[x$$[i]], \{i, 5\}]
\end{align*}
\]

\[
\text{Out[45]} = \{X_1, X_2, X_3, X_4, X_5\}
\]

Here the third component of the vector \( X \) is displayed.

\[
\begin{align*}
\text{In[46]} &= X[3]
\end{align*}
\]

\[
\text{Out[46]} = X
\]

\[
\begin{align*}
\text{In[47]} &= X[1]
\end{align*}
\]

... Part: The expression \( X[2, 1] \) cannot be used as a part specification.

... Part: The expression \( X[2, 1] \) cannot be used as a part specification.

... Part: The expression \( i \) cannot be used as a part specification.

... General: Further output of Part::pkspec1 will be suppressed during this calculation.

\[
\text{Out[47]} = \{X_1, X_2, X_3, X_4, X_5\}[[1]]
\]

The Mathematica arrays can be fully optimized and they result in a numerically efficient code. However, if the arrays are large then the resulting code might become to large to be compiled. In this case one can use AceGen defined arrays. With the AceGen arrays one can
express an array of expressions with a single auxiliary variable and to make a reference to an arbitrary or representative element of the array of expressions (see also Characteristic Formulae). Using the representative elements of the arrays instead of a full arrays will result in a much smaller code, however some optimization of the code might be prevented. Thus, one should use AceGen arrays only if they are absolutely necessary. Only one dimensional arrays (vectors) are currently supported in AceGen and only a following small set of operations is provided:

- **SMSArray** - create a new AceGen array
- **SMSPart** - take an arbitrary element of the array
- **SMSReplacePart** - replace an arbitrary element of the array. (WARNING: The SMSReplacePart command should be used only if it is absolutely necessary.)
- **SMSDot** - dot product of two vectors
- **SMSSum** - sum of all elements of the vector

AceGen supports two types of arrays:

- **Constant arrays**: a constant array is an array of arbitrary expressions (e.g. X\(_\text{SMSArray}[[1,2,3,400+x]]\)). All elements of the array are set to have given values.
- **General arrays**: The elements of the general array have no default values. Only a necessary memory space is allocated on the global vector of formulas at the time of introduction of a general array (e.g. M\(_\text{SMSArray}[10]\) allocates memory for the real array M with length 10 and SMSArray[M,10,n\&] would allocates memory and initialize all elements of M to \(\pi\)). General arrays are always introduced as a multi-valued auxiliary variables.

```mathematica
In[238]:= << AceGen;
SMSInitialize["test", Language" -> "C";
SMSModule["test", Real[x$$[5]], Integer[i$$, n$$];
x \[Right属] SMSReal[x$$];
1 \[Right属] SMSInteger[i$$];
```

Here a constant AceGen array is created. Result is a single auxiliary variable.

```mathematica
In[243]:= X \[Left属] SMSArray[Table[SMSReal[x$$[k]], {k, 5}]]
```

In this case, the third component of \(\sin(X)\) cannot be accessed by Mathematica since X is a symbol not an array.

```mathematica
In[244]:= X[3]
```

---

```
Part: Part 3 of 5 does not exist.
Part: Part 3 of 5 does not exist.
Part: Part 3 of 5 does not exist.
General: Further output of Part::partw will be suppressed during this calculation.
```

---

The third component of \(\sin(X)\) can be accessed by AceGen using SMSPart command.

```mathematica
In[245]:= SMSPart[X, 3]
```

However, with AceGen one can access the \(i\)-th component of X as well. During the AceGen sessions the actual value of the index \(i\) is not known, only later, at the evaluation time of the program, the value of the index \(i\) becomes known.

```mathematica
In[246]:= SMSPart[X, i]
```

Here a general array M of length \(n\) is created with the values set to \(\sin[i \pi/n], i=1,...,n\).
In[247] := n \[\triangleright\] SMSInteger[n$];
  SMSArray[M, n, Sin[\(\pi \# / n\)] &]

Out[247]=

- The third component of M can be accessed by AceGen using SMSPart command.

In[249] := SMSPart[M, 3]

Out[249]=

- Here, the 3-th element is not actually the 3-th element but an SMSIndexF object that points to the 3-th element.


Out[250]=

- General i-th component of M.

In[251] := SMSPart[M, i]

Out[251]=

- Here, the 3-th element is changed to Cos[2 \(\pi / n\)].

In[252] := M \[\triangleright\] SMSReplacePart[M, Cos[3 \(\pi / n\)], 3]

Out[252]=

Arrays are physically stored at the end of the global vector of formulae. The dimension of the global vector (specified in SMSInitialize) is automatically extended in order to accommodate additional arrays. In the case of array with the length that cannot be calculated at the code generation time, the user have to specify the length of global vector with the "WorkingVectorSize" option of the SMSWrite command.

**SMSArray**

- **SMSArray[\{exp\_1, exp\_2, \ldots\}]**
  create an array data object that represents a constant array of expressions \{exp\_1, exp\_2, \ldots\}

- **SMSArray[length]**
  create an array data object that represents a general real type array of length \(length\) and allocate space on the global vector of formulas

- **SMSArray[symbol, length, func]**
  create a multi-valued auxiliary variable that represents a general array data object of length \(length\), with elements \(func[i]\), \(i=1, \ldots, length\). Function \(func\) is called in a loop with symbolic counter \(i\). The result is assigned to \(symbol\).

- **SMSArray[symbol, \{n, length\}, func]**
  create \(n\) multi-valued auxiliary variables that represents \(n\) general array data objects of length \(length\), with elements \(func[i][1]\), \(func[i][2]\), \ldots, \(func[i][n]\), \(i=1, \ldots, length\). Function \(func\) is called in a loop with symbolic counter \(i\). Length \(length\) can be a positive integer or an arbitrary integer type expression. \(n\) is a positive integer. The resulting vector of arrays is assigned to \(symbol\).

<table>
<thead>
<tr>
<th>option</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Type&quot;</td>
<td>Automatic</td>
<td>Real (\Rightarrow) real type array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Integer (\Rightarrow) integer type array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Automatic (\Rightarrow) by default type is Real. Type is Integer if all elements are defined by SMSInteger[expi] (note that integer constants are considered to be real numbers unless introduced explicitly with SMSInteger command)</td>
</tr>
</tbody>
</table>
Options of the SMSArray function.

In[1]:= <<AceGen`;
SMSInitialize["test", "Language" -> "C"];
SMSModule["test", Real[x$$, r$$, s$$, t$$], Integer[n$$, m$$]];
x = SMSReal[x$$];
n = SMSInteger[n$$];

Constant arrays: A constant array is represented in the final source code as a sequence of auxiliary variables and formulae. The SMSArray[{exp1,exp2,...}] function returns the array data object (head SMSGroupF). All elements of the array are set to have given values. If an array is required as auxiliary variable then we have to use one of the functions that introduces a new auxiliary variable (e.g. n=SMSArray[{1,2,3,4}]).

- This creates the AceGen array object with constant contents. If we look at the result of the SMSArray function we can see that a single object has been created (Array[...]) representing the whole array of 4 real values.

In[6]:= SMSArray[{x, x^2, 0, π}]
Out[6]= Array[{i, i^2, 0, π}, Real]

General arrays: Definition of the general array only allocates space on the global vector. An array data object (head SMSArrayF) represents an array together with the information regarding random evaluation. Reference to the particular or an arbitrary element of the array is represented by the data object with the array index object (head SMSIndexF).

The SMSArray[length] function returns the SMSArrayF data object. The elements of the array have no default values. The SMSArrayF object HAS TO BE introduced as a new multi-valued auxiliary variable (e.g. r=SMSArray[10]). The value of the i-th element of the array can be set or changed by the SMSReplacePart[array, new value, i] command.

- This allocates space on the global vector of formulae and creates a general AceGen array object of length n. The values of the vector are NOT INITIALIZED.

In[7]:= SMSArray[n]
Out[7]= Array[i, Real]

The SMSArray[M, length, func] function assigns a multi-valued auxiliary variable that represent array of length length to symbol M. The elements of the array are set to the values returned by the function func. Function func has to return a representative formula valid for the arbitrary element of the array.

- Here a general array M of length n is created with the values initialized to Sin[i π/n],i=1,...,n. Note that command returns an auxiliary variable assigned to symbol M.

In[8]:= SMSArray[M, n, Sin[π #/n]] &
Out[8]=

- The above command is equivalent to:

In[9]:= M = SMSArray[n];
SMSDo[
    M = SMSReplacePart[M, i -> Sin[π i/n]];
    {i, 1, n, 1, M}];
M
Out[9]=

- The length of the array is obtained by SMSArrayLength

In[12]:= SMSArrayLength[M]
Out[12]=

The SMSArray[M, {n,Length},func] function assigns n multi-valued auxiliary variables that points at the nth SMSArrayF data objects to symbol M. The elements of the array are set to the values returned by the function func. Function func has to return n representative
formulae valid for the arbitrary elements of the arrays.

**Manipulating Arrays**

**SMSPart, SMSReplacePart, SMSArrayLength**

<table>
<thead>
<tr>
<th>Formula</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMSPart[{exp_1,exp_2,...},index]</td>
<td>create an index data object that represents the (n)-th element of the array of expressions {exp_1,exp_2,...} (only real type arrays!)</td>
</tr>
<tr>
<td>SMSPart[array,(i)]</td>
<td>create an index data object that represents the (i)-th element of the array of expressions represented by the array data object array</td>
</tr>
<tr>
<td>SMSArrayLength[array]</td>
<td>returns the length of the array array</td>
</tr>
<tr>
<td>SMSReplacePart[array, (i)-&gt;new]</td>
<td>set (i)-th element of the array to be equal new (array has to be an auxiliary variable that represents a general array data object)</td>
</tr>
</tbody>
</table>

The argument array is an array data object defined by SMSArray function or an auxiliary variable that represents an array data object. The argument index is an arbitrary integer type expression. During the AceGen sessions the actual value of the index is not known, only later, at the evaluation time of the program, the actual value of the index becomes known. Consequently, AceGen assigns a new signature to the index data object in order to prevent false simplifications. The values are calculated as perturbed mean values of the expressions that form the array.

The SMSPart function does not create a new auxiliary variable. If an arbitrary element of the array is required as an auxiliary variable, then we have to use one of the functions that introduces a new auxiliary variable (e.g. n-SMSPart[\{1,2,3,4\},\(i\)]).

**Example: SMSPart**

```plaintext
SMSInitialize["test"];
SMSModule["test", Real[\$$x$$, \$$r$$], Integer[\$$i$$]];  
x ⊨ SMSReal[\$$x$$];
i ⊨ SMSInteger[\$$i$$];
g ⊨ SMSArray[\{x, x^2, 0, \pi\}];
gi ⊨ SMSPart[g, i];
SMSExport[gi, \$$r$$];
SMSWrite["test"];  
Method: test 2 formulae, 29 sub-expressions  
[0] File created: test.m Size : 726
```
FilePrint["test.m"]

(*-------------------------------------------------------------------------
  * AceGen  2.103 Windows (18 Jul 08)  *
  * Co. J. Korelc  2007  18 Jul 08 15:41:16  *
  *-------------------------------------------------------------------------)

User : USER
Evaluation time : 0 s  Mode : Optimal
Number of formulae : 2  Method: Automatic
Module : test size : 29
Total size of Mathematica code : 29 subexpressions  (*)

(*-----------------------------------------------------------------------
  M O D U L E  
  *-----------------------------------------------------------------------)

SetAttributes[test,HoldAll];
test[x_\_\_,r_\_\_] := Module[{},
$VV[5000]=x$;
$VV[5001]=x^2$;
$VV[5002]=0$;
$VV[5003]=\pi$;
r$=$=$VV[Round[4999+i$]];]

Example: SMSReplacePart

In[278]:=
SMSInitialize["test", "Language"->"C"];
SMSModule["test", Real[x_,r_], Integer[n_]]; x := SMSReal[x$_$];
n := SMSInteger[n$_$];
SMSArray[M, n, Function[{i}, i^2]]; M := SMSReplacePart[M, 0, 2];
SMSExport[SMSPart[M, 1], r$_$];
SMSWrite["test", "WorkingVectorSize"->200];

[0] Consistency check - expressions

Method: test  Working length must be more than: 124 + n$_$

File: test.c  Size: 872  Time: 0
In[286]:= FilePrint["test.c"]

/***********************************************************/
User     : USER
Notebook : AceGenTutorials
Evaluation time                 : 0 s     Mode  : Optimal
Number of formulae              : 5       Method: Automatic
Subroutine                      : test size: 58
Total size of Mathematica  code : 58 subexpressions
Total size of C code            : 269 bytes */

#include "sms.h"

/*******************
S U B R O U T I N E
*********************/

void test
(double v[200],
double (*x),
double (*r),
int (*n))
{
int i3,i5,i0;
i0 = 4;
i3 = i0;
i0 = i0 + (*n);
for(i5=1;i5<=(*n);i5++){
v[20+i3+i5]=(double)((i5*i5));
}/* end for */
v[22+i3]=0e0;
(*r)=v[21+i3];
}

SMSDot

SMSDot[arrayo1, arrayo2]
dot product of the two arrays of expressions represented by the array data objects arrayo1 and arrayo2

Example

The task is to create a function that returns a dot product of the two vectors of expressions and the i-th element of the second vector.

- This initializes the AceGen system and starts the description of the "test" subroutine.

In[227]:= << AceGen'';
SMSInitialize["test", "Language" -> "Fortran"];
SMSModule["test", Real[x$$, r$$, s$$], Integer[n$$, m$$]];
x += SMSSReal[x$$];
n += SMSSInteger[n$$];
m += SMSSInteger[m$$];

- If an array is required as auxiliary variable then we have to use one of the functions that introduces a new auxiliary variable. Note that a single auxiliary variable has been created representing arbitrary element of the array. The signature of the array is calculated as perturbed average signature of all array elements.

In[232]= A += SMSArray[{x, x^2, θ, π}]

Out[232]= 

- This creates the second AceGen array object with constant contents.

In[233]= B += SMSArray[{3 x, 1 + x^2, Sin[x], Cos[x π]}]

Out[233]= 

- This calculates a dot product of vectors A and B.
In[234]:= dot = SMSSDot[A, B]

Out[234]= A • B

This creates an index to the n-th element of the second vector.

In[235]:= Bn = SMSPart[B, n]

Out[235]= Bn

The same can be obtained with command

In[236]:= SMSEXport[(dot, Bn), {r$$, s$$}];  

SMSEXport["test"];  

[0] Consistency check - expressions

File: test.f Size: 1053 Time: 0

Method test No.Formulae 3 No.Leafs 72

In[238]:= FilePrint["test.f"]

SUBROUTINE test(v,x,r,s,n,m)
IMPLICIT NONE
include 'sms.h'
INTEGER n,m
DOUBLE PRECISION v(125),x,r,s
v(5)=x**2
v(22)=3d0*x
v(23)=1d0+v(5)
v(24)=dsin(x)
v(25)=dcos(0.3141592653589793d1*x)
v(18)=x
v(19)=v(5)
v(20)=8d0
v(21)=0.3141592653589793d1
r=SMSSDot(v(18),v(22),4)
s=v(21+n)
END

SMSSum

SMSSum[array]
sum of all elements of the array represented by an array data object array

The argument is an array data object that represents an array of expressions (see Arrays). The signature of the result is sum of the signatures of the array components.
Manipulating Notebooks

Cell tags are used in Mathematica to find notebook cells or classes of cells in notebook. The Add/Remove Cell Tags menu item opens a dialog box that allows you to add or remove cell tags associated with the selected cell(s). Mathematica attaches the specified cell tag to each of the selected notebook cells. The cell tags are not visible unless Show Cell Tags in the Cell Tags menu is checked. To search for cells according to their cell tags, you can use the Cell Tags submenu command.

The Mathematica mechanism of tagging cells is in AceGen used to find parts of the current notebook that contain the actual AceGen input. Selected parts are then evaluated. An unevaluated form of the selected parts is also stored into internal AceGen database for later creation of a new notebook that is composed of the parts of the notebook that were actually used to generate numerical subroutines. Selected part can also be modified before they are included into data base.

SMSEvaluateCellsWithTag

The SMSEvaluateCellsWithTag command finds and evaluates all cells with the specified tag.

<table>
<thead>
<tr>
<th>option</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;CollectInputStart&quot;</td>
<td>False</td>
<td>starts the process of collecting the unevaluated contents of all the notebook cells evaluated by the SMSEvaluateCellsWithTag command during the session</td>
</tr>
<tr>
<td>&quot;RemoveTag&quot;</td>
<td>False</td>
<td>remove the tag tag from the cells included into recreated notebook</td>
</tr>
<tr>
<td>&quot;CollectInput&quot;</td>
<td>True</td>
<td>on False temporarily suspends the process of collecting cells for the current SMSEvaluateCellsWithTag call</td>
</tr>
<tr>
<td>&quot;BlockStyle&quot;</td>
<td>False</td>
<td>do not change cell style</td>
</tr>
<tr>
<td>&quot;Position&quot;</td>
<td>-1</td>
<td>main level position of the collecting input cells relative to the current position (each main level call of SMSEvaluateCellsWithTag creates a new position, structure is stored in variable $SMSSession[[10]]$)</td>
</tr>
<tr>
<td>&quot;CreateFunction&quot;</td>
<td>False</td>
<td>create a function with the given name from the collected input (NOTE that a maximum of one nested call to SMSEvaluateCellsWithTag is allowed in that case)</td>
</tr>
<tr>
<td>&quot;Active&quot;</td>
<td>True</td>
<td>collected cells are evaluatable</td>
</tr>
<tr>
<td>&quot;Echo&quot;</td>
<td>True</td>
<td>print the tag to the current notebook</td>
</tr>
<tr>
<td>&quot;Evaluate&quot;</td>
<td>True</td>
<td>True = evaluates cells and includes the contents of the cells into recreated notebook (see SMSRecreateNotebook) False = includes the contents of the cells into recreated notebook, but it does not evaluate the cells</td>
</tr>
</tbody>
</table>

Example:

```mathematica
CELLTAG
In[23]= Print["this is cell with the tag CELLTAG"]; In[24]= <<AceGen;
    SMSInitialize["test", "Language" -> "C"];
    SMSModule["sub1"];
    SMSEvaluateCellsWithTag["CELLTAG"];```
The `SMSRecreateNotebook` command creates a new notebook that includes an un(evaluated) contents of all the notebook cells that were evaluated by the `SMSEvaluateCellsWithTag` commands during the AceGen session. The first `SMSEvaluateCellsWithTag` call has to have option "CollectInputStart"->True in order to start the process of collecting the cells.

Option default description

<table>
<thead>
<tr>
<th>option</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;File&quot;</td>
<td>current session name</td>
<td>notebook file name</td>
</tr>
<tr>
<td>&quot;Head&quot;</td>
<td>{}</td>
<td>list of additional Cells included at the head of the notebook</td>
</tr>
<tr>
<td>&quot;Close&quot;</td>
<td>False</td>
<td>close notebook after creation</td>
</tr>
</tbody>
</table>

Example:

```
In[131]:= Print["This is cell with the tag CELLTAG-1."];
In[131]:= Print["This is cell with the tag CELLTAG-2."];<
AceGen`; SMSInitialize["test", "Language" -> "C"];
SMSEvaluateCellsWithTag["CELLTAG-1","CollectInputStart"->True];
SMSEvaluateCellsWithTag["CELLTAG-2","RemoveTag"->True];
```

The `SMSRecreateNotebook` command creates a new notebook with the name "test.nb" that contains cells with the cell tag "CELLTAG-1" and the cell tag "CELLTAG-2". Note that the second cell has its cell tag removed due to the `SMSEvaluateCellsWithTag` option "RemoveTag"->True.

```
 SMSRecreateNotebook[];
```

**Manipulate the contents of the generated notebook**

A series of commands can be used to change the appearance of the cells. First, based on the user input cell, two additional cells are generated:

a) **evaluation cell**, 

b) **input cell**, 

c) **output cell**,
Evaluation cell is directly evaluated immediately after it is created. **Recreated cell** is included into data base for the generation of a special notebook that hold all the cells evaluated during the AceGen session. **NOTE THAT ALL SMSTag...** commands are evaluated before the evaluation cell is generated, thus arguments of the SMSTag... commands can not depend on anything that will be evaluated inside the cell.

**SMSTagIf[condition, texp, fexp]**
texp is evaluated and included unevaluated into both, evaluation cell and recreated cell, if condition yields True and fexp if condition yields False. **Expression condition is evaluated before the cell is evaluated!**

**SMSTagSwitch[switchexp, form1, value1, form2, value2, ...]**
evaluates switchexp, then compares it with each of formi in turn and chooses the one that corresponds to the first match found. The chosen value is then included in an unevaluated form into evaluation cell and recreated cell. **Expression switchexp is evaluated before the cell is evaluated!**

**SMSTagWhich[test1, value1, test2, value2, ...]**
evaluates each of the testi in turn, choosing the value of the valuei corresponding to the first one that yields True. The chosen value is then included in an unevaluated form into evaluation cell and recreated cell. **Expression testi is evaluated before the cell is evaluated!**

**SMSTagReplace[evaluated, included]**
cell is modified in a such a way that evaluated is included into evaluation cell and included is included into recreated cell

**SMSTagReplace[evaluated]**
cell is modified in a such a way that evaluated is included into evaluation cell and nothing is included into recreated cell

**SMSTagEvaluate[exp]**
evaluated exp is included into evaluation cell and recreated cell. **Expression exp is evaluated before the cell is evaluated! Expression exp can NOT contain or implicitly call any SMSTag... command.**

**SMSTagExclude[exp]**
evaluates exp but nothing is included into evaluation cell or recreated cell. **Expression exp is evaluated before the cell is evaluated! Expression exp can NOT contain or implicitly call any SMSTag... command.**

**SMSTagInclude[tag, condition]**
SMSTagInclude[tag]≡SMSTagInclude[tag, True]
includes the contents of the Cell with the tag tag at the position of the SMSTagInclude call if the value of condition is True

This commands are executed after the cell has been read by the SMSEvaluateCellsWithTag command and before the cell is evaluated.
Example SMSRecreateNotebook:

```mathematica
In[49]:= SMSTagIf[condition,
  Print["This is True branch of the SMSTagIf structure."];,
  Print["This is False branch of the SMSTagIf structure."]];

SMSTagSwitch[expr,
  "A", Print["This is A branch of the SMSTagSwitch structure."];
  "B", Print["This is B branch of the SMSTagSwitch structure."];
  "C", Print["This is C branch of the SMSTagSwitch structure."]];

SMSTagReplace[
  Print["This branch of the SMSTagReplace structure is evaluated."];,
  Print["This branch of the SMSTagReplace structure is included in new notebook."]];
```

\(x = 1 + 1\); (*this is included in new notebook in an unevaluated form*)
\(x = \text{SMSTagEvaluate}[1 + 1]\); (*1+1 is evaluated before it is included in new notebook*)

\(\text{In[75]}\):= <<AceGen`; SMSInitialize["test", "Language" -> "C"]; SMSModule["sub1"]; condition=True; expr="C"; SMSEvaluateCellsWithTag["CELLTAG-3","CollectInputStart"->True];

\(0-0\) Include Tag : CELLTAG-3 (1 cells found, 1 evaluated)
This is True branch of the SMSTagIf structure.
This is C branch of the SMSTagSwitch structure.
This branch of the SMSTagReplace structure is evaluated.

- The SMSRecreateNotebook command creates a new notebook with the name “test.nb” that contains cell with the cell tag “CELLTAG-3”.

\(\text{In[81]}\):= SMSRecreateNotebook[];
- The contents of the "CELLTAG-3" cell is modified due to the use of SMSTagIf, SMSTagSwitch and SMSTagEvaluate commands.

Example: create a function with SMSTagInclude

\(\text{In[136]}\):= <<AceGen`
\(\text{In[137]}\):= Clear[func]
func[r_, t_] := Module[{x, y},
   SMSTagInclude["functionpart1C", part1];
   SMSTagInclude["functionpart2C", part2];
   x + y
]

x = 5 r;
y = 6 t;
y + y + Sin[t];

In[138] := SMSInitialize["tmptest", "Language" -> "C", "CollectInput" -> True];

- Here function func is created from the given set of cells. Function is defined, but not evaluated!

In[139] := part1 = True;
   part2 = True;
   SMSEvaluateCellsWithTag["functionheadC"];
   [0s e0 Include Tag : functionheadC (1 cells found, 1 evaluated]

In[142] := SMSModule["Include", Real[r$$, t$$, e$$]];

- Here function func is symbolically evaluated for the given set of parameters.

In[143] := e = func[r, t];
In[145] := SMSExport[e, e$$];
   SMSWrite[];

   File: tmptest.c  Size: 790  Time: 0

   Method
   No.Formulae  Include 3
   No.Leafs 31

In[148] := FilePrint["tmptest.c", -8]

/******************** SUBROUTINE ********************/
void Include(double v[117], double (*r), double (*t), double (*e))
{
   v[4] = 6e0 * (*t);
   (*e) = 5e0 * (*r) + v[4];
}

- The SMSRecreateNotebook command creates a new notebook with the name "tmptest.nb" that contains definition of the function.

In[149] := nb = SMSRecreateNotebook[];
```mathematica
func[r_, t_] := Module[{x, y},
    x = 5 r;
    y = 6 t;
    y + y + Sin[t];
    x + y]
```

```mathematica
In[150]:= NotebookClose[nb];
```
Signatures of Expressions

The input parameters of the subroutine (independent variables) have assigned a randomly generated high precision real number or an **unique signature**. The signature of the dependent auxiliary variables is obtained by replacing all auxiliary variables in the definition of variable with corresponding signatures and is thus deterministic. The randomly generated high precision real numbers assigned to the input parameters of the subroutine can have in some cases effects on code optimization procedure or even results in wrong code. One reason for the incorrect optimization of the expressions is presented in section **Expression Optimization**. Two additional reasons for wrong simplification are round-off errors and hidden patterns inside the sets of random numbers. In AceGen we can use randomly generated numbers of arbitrary precision, so that we can exclude the possibility of wrong simplifications due to the round-off errors. AceGen also combines several different random number generators in order to minimize the risk of hidden patterns inside the sets of random numbers.


The precision of the randomly generated real numbers assigned to the input parameters is specified by the "Precision" option of the SMSInitialize function. Higher precision would slow down execution.

In rare cases user has to provide its own signature or increase default precision in order to prevent situations where wrong simplification of expressions might occur. This can be done by providing an additional argument to the symbolic-numeric interface functions SMSReal and SMSInteger, by the use of function that yields an unique signature (SMSFreeze, SMSFictive, Expression Optimization) or by increasing the general precision (SMSInitialize).

---

**SMSReal[exte,code]**
create real type external data object with the signature accordingly to the code

**SMSInteger[exte,code]**
create integer type external data object with the definition exte and signature accordingly to the code

**SMSReal[i_List,code]** $\equiv$ Map[SMSReal[#&],i]
User defined signature of input parameters.

code | the signature is:
---|---
$v_{Real}$ | real type random number form interval $[0.95 v, 1.05 v]$ 
{$v_{min}_{Real},v_{max}_{Real}$} | real type random number form interval $[v_{min},v_{max}]$
False | if exte yields regular signature then perturbed signature of the expression else random signature (signature is always random!)
(True,signature_{Real}) | given signature (it has to have precision SMSEvaluatePrecision)
True | if exte yields regular signature then signature of the expression else random signature (signature does not need to be random, thus simplifications are possible)

Evaluation codes for the generation of the signature.

**SMSRandom[]**
random number on interval $[0,1]$ with the precision SMSEvaluatePrecision

**SMSRandom[i,j]**
random number on interval $[i,j]$ with the precision SMSEvaluatePrecision

**SMSRandom[i]**
gives random number from the interval $[0.9^i,1.1^i]$

**SMSRandom[i_List]** $\equiv$ Map[SMSRandom[#&],i]
Generation of random signatures with the precision required by AceGen.

Example 1

The numerical constants with the Infinity precision (11, π, Sqrt[2], 2/3, etc.) can be used in AceGen input without changes. The fixed precision constants have to have at least SMSEvaluatePrecision precision in order to avoid wrong simplifications. If the precision of the numerical constant is less than default precision (SMSInitialize) then AceGen automatically increases precision with the SetPrecision[exp,SMSEvaluatePrecision] command.

In[346]:= << AceGen`;
SMSInitialize["test", "Language" -> "Mathematica", "Mode" -> "Debug"];
SMSModule["test"];
time=0 variable= 0 = {} 

In[349]:= x π;

In[350]:= y 3.1415;

Precision of the user input real number (3.1415)
has been automatically increased. See also: Signatures of the Expressions

Example 2

This initializes the AceGen system, starts the description of the "test" subroutine and sets default precision of the signatures to 40.

In[351]:= << AceGen`;
SMSInitialize["test", "Language" -> "Fortran", "Precision" -> 40];
SMSModule["test", Real[x$$, y$$], Integer[n$$]]; 

Here variable x gets automatically generated real random value from interval [0,1], for variable y three interval is explicitly prescribed, and an integer external variable n also gets real random value.

In[354]:= x = SMSReal[x$$];  
y = SMSReal[y$$, {-100, 100}];  
n = SMSInteger[n$$];

This displays the signatures of external variables x, y, and n .

In[357]:= {x, y, n} // SMSEvaluate // Print 
{0.5126296357476784249473803865168894899875, 
47.74123398661873123794181249417015192, 4.641185606823421179019188449964375656913}
CHAPTER 5

AceGen Examples
Summary of AceGen Examples
The presented examples are meant to illustrate the general symbolic approach to automatic code generation and the use of AceGen in the process. They are NOT meant to represent the state of the art solution or formulation of particular numerical or physical problem.

More examples are available at www.fgg.uni-lj.si/symech/examples.htm.

Basic AceGen Examples
Standard AceGen Procedure
Solution to the System of Nonlinear Equations

Advanced AceGen Examples
User Defined Functions
Minimization of Free Energy

Implementation of Finite Elements in AceFEM
Examples related to the automation of the Finite Element Method using AceFEM are part of AceFEM documentation (see Summary of Examples).

Standard FE Procedure

Implementation of Finite Elements in Alternative Numerical Environments
FEAP – ELFEN – ABAQUS – ANSYS

User Defined Environment Interface
Solution to the System of Nonlinear Equations

- **Description**

Generate and verify the MathLink program that returns solution to the system of nonlinear equations:

\[
\begin{aligned}
\phi_1 &= axy + x^3 = 0 \\
\phi_2 &= a - xy^2 = 0
\end{aligned}
\]

where \(x\) and \(y\) are unknowns and \(a\) is parameter.

- **Solution**

Here the appropriate MathLink module is created.

```mathematica
<< AceGen`
SMSInitialize["test", "Environment" -> "MathLink"];
SMSModule["test", Real[x$$, y$$, a$$, tol$$], Integer[n$$],
  "Input" -> \{x$$, y$$, a$$, tol$$, n$$\},
  "Output" -> \{x$$, y$$\}];
{x0, y0, a, e} = SMSReal[\{x$$, y$$, a$$, tol$$\}];
nmax = SMSInteger[n$$];
{x, y} = \{x0, y0\};
SMSDo[
  \[\theta = \{a \times y + x^3, a - x \times y^2\};
  Kt = SMSDo[\theta, \{x, y\}];
  (Δx, Δy) = SMSLinearSolve[Kt, -θ];
  \{x, y\} = \{x + Δx, y + Δy\};
  SMSIf[SMSQrt[(Δx, Δy).{Δx, Δy}] < e
    , SMSExport[\{x, y\}, \{x$$, y$$\}];
    SMSBreak[]];
  SMSIf[i = nmax
    , SMSPrintMessage["no convergence"];
    SMSReturn[]];
  , {i, 1, nmax, 1, \{x, y\}}]
];
SMSWrite[];
```

Here the MathLink program test.exe is build from the generated source code and installed so that functions defined in the source code can be called directly from Mathematica. (see also SMSInstallMathLink)

- **Verification**

For the verification of the generated code the solution calculated by the build in function is compared with the solution calculated by the generated code.
\[\text{In[46]} = \text{test}[1.9, -1.2, 3., 0.0001, 10]\]
\[\text{Out[46]} = \{1.93318, -1.24573\}\]
\[\text{In[47]} = \text{x =.; y =.; a = 3.};\]
\[\text{Solve\{a x y + x^3 = 0, a - x y^2 = 0\}, (x, y)\}\]
\[\text{Out[47]} = \{(y \to -1.24573, x \to 1.93318),
(y \to -0.384952 + 1.18476 i, x \to -1.56398 + 1.1363 i),
(y \to -0.384952 - 1.18476 i, x \to -1.56398 - 1.1363 i),
(y \to 1.00782 + 0.732222 i, x \to 0.597386 - 1.83857 i),
(y \to 1.00782 - 0.732222 i, x \to 0.597386 + 1.83857 i)\}\]
Minimization of Free Energy

Contents

- Problem Description
- A) Trial Lagrange polynomial interpolation
- B) Finite difference interpolation
- C) Finite element method

Problem Description

In the section Standard FE Procedure the description of the steady-state heat conduction on a three-dimensional domain was given. The solution of the same physical problem can be obtained also as a minimum of the free energy of the problem. Free energy of the heat conduction problem can be formulated as

$$\Pi = \int_{\Omega} \left( \frac{1}{2} k \Delta \phi - \phi \frac{\partial}{\partial \Omega} \right) d\Omega$$

where $\phi$ indicates temperature, $k$ is the conductivity and $Q$ is the heat generation per unit volume and $\Omega$ is the domain of the problem.

The domain of the example is a cube filled with water ([-0.5m,0.5m] × [-0.5m,0.5m] × [0,1m]). On all sides, apart from the upper surface, the constant temperature $\phi=0$ is maintained. The upper surface is isolated so that there is no heat flow over the boundary. There exists a constant heat source $Q=500 \text{ W/m}^3$ inside the cube. The thermal conductivity of water is 0.58 W/m K. The task is to calculate the temperature distribution inside the cube.

The problem is formulated using various approaches:

A. Trial polynomial interpolation
   - M.G Gradient method of optimization + Mathematica directly
   - M.N Newton method of optimization + Mathematica directly
   - A.G Gradient method of optimization + AceGen+MathLink
   - A.N Newton method of optimization + AceGen+MathLink

B. Finite difference interpolation
   - M.G Gradient method of optimization + Mathematica directly
   - M.N Newton method of optimization + Mathematica directly
   - A.G Gradient method of optimization + AceGen+MathLink
   - A.N Newton method of optimization + AceGen+MathLink

C. AceFEM Finite element method

The following quantities are compared:

- temperature at the central point of the cube ($\phi(0,0,0.5)$)
- time for derivation of the equations
- time for solution of the optimization problem
- number of unknown parameters used to discretize the problem
- peak memory allocated during the analysis
The case A with the trial polynomial interpolation represents the situation where the merit function is complicated and the number of parameters is small. The case B with the finite difference interpolation represents the situation where the merit function is simple and the number of parameters is large.

REMARK: The presented example is meant to illustrate the general symbolic approach to minimization of complicated merit functions and is not the state of the art solution of thermal conduction problem.

- **A)** Trial Lagrange polynomial interpolation

**Definitions**

- A trial function for temperature $\phi$ is constructed as a fifth order Lagrange polynomial in $x$, $y$, and $z$ direction. The chosen trial function is constructed in a way that satisfies boundary conditions.

```plaintext
In[1] := << AceGen``
Clear[{x, y, z, \alpha} ];
kcond = 0.58; Q = 500;
order = 5;
nterm = (order - 1) (order - 1) (order)
```

**Output**: 80

- Here the fifth order Lagrange polynomials are constructed in three dimensions.
Here the Gauss points and weights are calculated for \( ngp \times ngp \times ngp \) Gauss numerical integration of the free energy over the domain \([-0.5m, 0.5m] \times [-0.5m, 0.5m] \times [0, 1m] \).

```mathematica
ngp = 6;
<< NumericalDifferentialEquationAnalysis`
g1 = GaussianQuadratureWeights[ngp, -0.5, 0.5];
g2 = GaussianQuadratureWeights[ngp, -0.5, 0.5];
g3 = GaussianQuadratureWeights[ngp, 0, 1];
gp = {g1[[#1[[1]], 1]], g2[[#1[[2]], 1]],
g3[[#1[[3]], 1]], g1[[#1[[1]], 2]] + g2[[#1[[2]], 2]] * g3[[#1[[3]], 2]]} & /@ Flatten[Array[{{n3, n2, n1} &, (ngp, ngp, ngp)}, 2];
```
\begin{align*}
\Delta \phi &= \{ D[\phi, x], D[\phi, y], D[\phi, z] \}; \\
\Pi &= 1/2 k_{\text{cond}} \Delta \phi \cdot \Delta \phi - \phi Q; \\
\end{align*}

\begin{align*}
\Pi_i &= \text{Total} \left[ \frac{\Pi}{\text{Map}[#[{4}], \Pi]}, \text{Map}[#[{1}], \text{Map}[#[{2}], \Pi]], \text{Map}[#[{3}]] \right] \& \text{gp}; \\
\text{derivation} &= \text{SessionTime[]} - \text{start}
\end{align*}

\textbf{G. Gradient based optimization}

\begin{align*}
\text{sol} &= \text{FindMinimum}[\Pi_i, \text{Array}[\alpha[i], 0.] \& \text{nterm}], \\
\text{Method} &\rightarrow \text{"Gradient"}, \text{EvaluationMonitor} \rightarrow \text{(ii++)}}]; \\
\text{(ii, } \phi[i] / . \text{sol}[2] / . \{x \rightarrow 0, y \rightarrow 0, z \rightarrow 0.5\})
\end{align*}

\textbf{N. Newton method based optimization}

\begin{align*}
\text{sol} &= \text{FindMinimum}[\Pi_i, \text{Array}[\alpha[i], 0.] \& \text{nterm}], \\
\text{Method} &\rightarrow \text{"Newton"}, \text{EvaluationMonitor} \rightarrow \text{(ii++)}}]; \\
\text{(ii, } \phi[i] / . \text{sol}[2] / . \{x \rightarrow 0, y \rightarrow 0, z \rightarrow 0.5\})
\end{align*}

\section*{AceGen code generation}

- The subsection Definitions has to be executed before the current subsection.
In[75]:= SMSModule["FThermal", Real[nterm, f$], "Input" -> a$, "Output" -> f$]; SMSExport[\Pi, f$]; SMSDo[i, 1, gp // Length]; \Pi = \Pi f[i]; SMSExport[\Pi, f$, "AddIn" -> True]; SMSEndDo[];

In[81]:= SMSModule["GThermal", Real[nterm, g$], "Input" -> a$, "Output" -> g$]; SMSExport[Table[0, {nterm}], g$]; SMSDo[i, 1, gp // Length]; \Pi = \Pi f[i]; SMSDo[j, 1, nterm]; \delta\Pi = SMSD[\Pi, ag, j, "Mode" -> "Forward"]; SMSExport[\delta\Pi, g$[j], "AddIn" -> True]; SMSEndDo[];

In[85]:= derivation = SessionTime[] - start

Out[85]= 5.2324732

In[87]:= SMSModule["HThermal", Real[nterm, nterm, nterm], "Input" -> a$, "Output" -> h$]; SMSDo[i, 1, nterm]; SMSDo[j, 1, nterm]; SMSExport[\Pi, h$[i, j]]; SMSEndDo[];
SMSDo[i, 1, gp // Length]; \Pi = \Pi f[i]; SMSDo[j, 1, nterm]; \delta\Pi = SMSD[\Pi, ag, j, "Mode" -> "Forward"]; SMSDo[k, 1, nterm]; \hij = SMSD[\delta\Pi, ag, k, "Mode" -> "Forward"]; SMSExport[\hij, h$[j, k], "AddIn" -> True]; SMSEndDo[];
SMSEndDo[];
SMSEndDo[];

In[90]:= SMSWrite[];

[16] Consistency check - expressions

<table>
<thead>
<tr>
<th>File: Thermal.c</th>
<th>Size: 125709</th>
<th>Time: 17</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>FThermal</td>
<td>GThermal</td>
</tr>
<tr>
<td>No. Formulae</td>
<td>150</td>
<td>162</td>
</tr>
<tr>
<td>No. Leafs</td>
<td>4816</td>
<td>5377</td>
</tr>
</tbody>
</table>
AceGen Solution

G. Gradient based optimization

\begin{verbatim}
In[110]:= start = SessionTime[]; ii = 0;
sol = FindMinimum[FThermal[ϕi], {ϕi, Table[0, {nterm}]}],
    Method -> "Gradient", Gradient -> GThermal[ϕi], EvaluationMonitor -> (ii++);]
Out[110]= {915, 55.8724, 0.4513012}
\end{verbatim}

N. Newton method based optimization

\begin{verbatim}
In[113]:= start = SessionTime[]; ii = 0;
sol = FindMinimum[FThermal[ϕi], {ϕi, Table[0, {nterm}]}],
    Method -> "Newton", Hessian -> HThermal[ϕi], Gradient -> GThermal[ϕi], EvaluationMonitor -> (ii++);]
Out[113]= {4, 55.8724, 0.0580390}
\end{verbatim}

B) Finite difference interpolation

Definitions

- The central difference approximation of derivatives is used for the points inside the cube and backward or forward difference for the points on the boundary.

\begin{verbatim}
In[116]:= << AceGen``; Clear[α, i, j, k]; nx = ny = nz = 11;
dlx = 1./ (nx - 1); dly = 1./ (ny - 1); dlz = 1./ (nz - 1); bound = {0}; nboun = 1; kcond = 0.58; Q = 500;
\end{verbatim}
In[129]= nterm = 0; dofs = {};
index = Table[Which[
  i ≤ 2 || i ≥ nx + 1 || j ≤ 2 || j ≥ ny + 1 || k ≤ 2, b[1],
  k = nz + 2,
  If[FreeQ[dofs, α[i, j, k - 1]], ++nterm; AppendTo[dofs, α[i, j, k - 1] → nterm]; nterm,
  α[i, j, k - 1] /. dofs
], True,
  If[FreeQ[dofs, α[i, j, k]], ++nterm; AppendTo[dofs, α[i, j, k] → nterm]; nterm,
  α[i, j, k] /. dofs
],
  ],
{i, 1, nx + 2}, {j, 1, ny + 2}, {k, 1, nz + 2}) /. b[_] → nterm + i;
ϕ = Array[α, nterm];
nterm

Out[129]= 810

Direct use of Mathematica

- The subsection Definitions have to be executed before the current subsection.

In[130]= start = SessionTime[];
Π = Sum[dlxt = If[i == 2 || i == nx + 1, dlxt = dlx/2, dlx];
  dlyt = If[j == 2 || j == ny + 1, dlyt = dly/2, dly];
  dlzt = If[k == 2 || k == nz + 1, dlzt = dlz/2, dlz];
  vol = dlxt*dlyt*dlzt;
  aijk = Map[If[# > nterm, bound[[# - nterm]], α[#]] &,
    Extract[index, {i, j, k}, {i - 1, j, k}, {i + 1, j, k}, {i, j - 1, k},
    {i, j + 1, k}, {i, j, k - 1}, {i, j, k + 1}]];
  grad = {aijk[[3]] - aijk[[2]], aijk[[5]] - aijk[[4]], aijk[[7]] - aijk[[6]]}/
    {2 dlxt, 2 dlyt, 2 dlzt};
  vol (1/2 kcond grad . grad - Q aijk[[1]])
  , {i, 2, nx + 1}, {j, 2, ny + 1}, {k, 2, nz + 1}
];
derivation = SessionTime[] - start

Out[130]= 0.1040714

- G. Gradient based optimization

In[132]= start = SessionTime[]; ii = 0;
sol = FindMinimum[Π, Array[{α[#], 0.} &, nterm],
  Method -> "Gradient", EvaluationMonitor :> (ii++);]
  {ii, α[index][[(nx + 3)/2, (ny + 3)/2, (nz + 3)/2]]} /. sol[[2]], SessionTime[] - start]

... FindMinimum: Failed to converge to the requested accuracy or precision within 100 iterations.

Out[132]= {1604, 57.5034, 2.5997275}

- N. Newton method based optimization
In[135]:= 
start = SessionTime[]; ii = 0;
sol = 
FindMinimum[Πi, Array[α[#], 0., nterm], Method -> "Newton", EvaluationMonitor -> (ii++;)]; 
{ii, α[index[index[(nx + 3)/2, (ny + 3)/2, (nz + 3)/2]] /. sol[[2]], SessionTime[] - start}

Out[135]= {4, 57.5034, 0.6234138}

AceGen code generation
The subsection Definitions have to be executed before the current subsection.

In[138]:= 
start = SessionTime[];
SMSInitialize["Thermal", "Environment" -> "MathLink"]

Πf[i_, j_, k_] := 
indexp ⊢ SMSInteger[Map[ 
index$[(#1) - 1] * (nyp + 2) (nzp + 2) + (#2) - 1] * (nyp + 2) + #3] &, 
{(i, j, k), (i - 1, j, k), (i + 1, j, k), (i, j - 1, k), 
(i, j + 1, k), (i, j - k - 1), (i, j + k + 1)]}; 
aijk ⊢ SMSReal[Map[a[#] &, indexp]]; 
{dx, dy, dz, kc, Qt} ⊢ SMSReal[Array[mc$, 5]]; 
SMSIf[i == 2 || i == nxp + 1]; 
dlxt = dx/2; 
SMSElse[]; 

dlxt - dx; 
SMSEndIf[dlxt]; 

SMSIf[j == 2 || j == nyp + 1]; 
dlyt = dy/2; 
SMSElse[]; 

dlyt - dy; 
SMSEndIf[dlyt]; 

SMSIf[k == 2 || k == nzp + 1]; 
dlzt = dz/2; 
SMSElse[]; 

dlzt - dz; 
SMSEndIf[dlzt]; 
vol = dlxt dlyt dlzt; 
vol (1/2 kc grad.grad - Qt aijk[1])

Out[138]= True
In[141]:=

```
SMSModule["FThermal", Integer[ndof$$, nt$$[3], index$$["*"], Real[a$$["*"], mc$$["*"], f$$], "Input"→{ndof$$, nt$$, index$$, a$$, mc$$}, "Output"→f$$];

SMSExport[0, f$$];

(nxp, nyp, nzp)→SMSInteger[Array[nt$$, 3]]; SMSDo[i, 2, nxp + 1]; SMSDo[j, 2, nyp + 1]; SMSDo[k, 2, nzp + 1]; Π ⊨ Π[f[i, j, k]; SMSExport[Π, f$$, "AddIn"→True]; SMSEndDo[]; SMSEndDo[]; SMSEndDo[];
```
The tangent matrix is in the case of finite difference approximation extremely sparse.

N. Newton method based optimization

G. Gradient based optimization

```
In[202]:= SMSInstallMathLink[]
derivation = SessionTime[] - start

Out[202]= {SMSSetLinkOption[{Thermal, {i_Integer, j_Integer}}], SMSLinkNoEvaluations[{Thermal},
FThermal[ndof_? (Head[#1] == Real || Head[#1] == Integer &), nt_?
(ArrayQ[#1, 1, Head[#1] == Real || Head[#1] == Integer & && Dimensions[#1] === {3} &),
index_? (ArrayQ[#1, 1, Head[#1] == Real || Head[#1] == Integer & &),
a_? (ArrayQ[#1, 1, Head[#1] == Real || Head[#1] == Integer & &),
mc_? (ArrayQ[#1, 1, Head[#1] == Real || Head[#1] == Integer & & ]},
GThermal[ndof_? (Head[#1] == Real || Head[#1] == Integer &), nt_?
(ArrayQ[#1, 1, Head[#1] == Real || Head[#1] == Integer & && Dimensions[#1] === {3} &),
index_? (ArrayQ[#1, 1, Head[#1] == Real || Head[#1] == Integer & &),
a_? (ArrayQ[#1, 1, Head[#1] == Real || Head[#1] == Integer & &),
mc_? (ArrayQ[#1, 1, Head[#1] == Real || Head[#1] == Integer & & ])}

Out[204]= 7.2057857

AceGen Solution

- G. Gradient based optimization
```

```
In[204]= start = SessionTime[]; ii = 0;
indexb = Flatten[indexb];
sol = FindMinimum[
FThermal[nterm, {nx, ny, nz}, indexb, Join[ϕi, bound], {dlx, dly, dlz, kcond, Q}]
, (ϕi, Table[0, {nterm}]),
Method -> "Gradient",
Gradient -> DThermal[nterm, {nx, ny, nz}, indexb, Join[ϕi, bound], {dlx, dly, dlz, kcond, Q}]
, EvaluationMonitor :> (ii++)]
(ii, α[index[[{(nx + 3) / 2, (ny + 3) / 2, (nz + 3) / 2}]]] /. MapThread[Rule, List@@sol[[2, 1]], SessionTime[] - start]

Out[204]= (1596, 57.5034, 10.2127803)

- N. Newton method based optimization
```

```
In[207]= start = SessionTime[]; ii = 0;
indexb = Flatten[index /. b[i_] :> nterm + 1];
sol = FindMinimum[
FThermal[nterm, {nx, ny, nz}, indexb, Join[ϕi, bound], {dlx, dly, dlz, kcond, Q}]
, (ϕi, Table[0, {nterm}]),
Method -> {"Newton", Hessian ->
HThermal[nterm, {nx, ny, nz}, indexb, Join[ϕi, bound], {dlx, dly, dlz, kcond, Q}]
}, Gradient -> DThermal[nterm, {nx, ny, nz}, indexb, Join[ϕi, bound], {dlx, dly, dlz, kcond, Q}]
, EvaluationMonitor :> (ii++)]
(ii, α[index[[{(nx + 3) / 2, (ny + 3) / 2, (nz + 3) / 2}]]] /. MapThread[Rule, List@@sol[[2, 1]], SessionTime[] - start]

Out[207]= (4, 57.5034, 0.2761834)
```

- The tangent matrix is in the case of finite difference approximation extremely sparse.
C) Finite element method

First the finite element mesh 30×30×30 is used to obtain convergence solution at the central point of the cube. The procedure to generate heat-conduction element that is used in this example is explained in AceGen manual section Standard FE Procedure.
In[224]:= SMTShowMesh["Mesh" -> True, "FillElements" -> True, "Field" -> "T*"]

Out[224]=

AceFEM
Numerical Environments
Finite Element Environments

Numerical simulations are well established in several engineering fields such as in automotive, aerospace, civil engineering, and material forming industries and are becoming more frequently applied in biophysics, food production, pharmaceutical and other sectors. Considerable improvements in these fields have already been achieved by using standard features of the currently available finite element (FE) packages. The mathematical models for these problems are described by a system of partial differential equations. Most of the existing numerical methods for solving partial differential equations can be classified into two classes: Finite Difference Method (FDM) and Finite Element Method (FEM). Unfortunately, the applicability of the present numerical methods is often limited and the search for methods which can provide a general tool for arbitrary problems in mechanics of solids has a long history. In order to develop a new finite element model quite a lot of time is spent in deriving characteristic quantities such as gradients, Jacobian, Hessian and coding of the program in an efficient compiled language. These quantities are required within the numerical solution procedure. A natural way to reduce this effort is to describe the mechanical problem on a high abstract level using only the basic formulas and leave the rest of the work to the computer.

The symbolic-numeric approach to FEM and FDM has been extensively studied in the last few years. Based on the studies various systems for automatic code generation have been developed. In many ways the present stage of the generation of finite difference code is more elaborated and more general than the generation of FEM code. Various transformations, differentiation, matrix operations, and a large number of degrees of freedom involved in the derivation of characteristic FEM quantities often lead to exponential growth of expressions in space and time. Therefore, additional structural knowledge about the problem is needed, which is not the case for FDM.

Using the general finite element environment, such as FEAP (Taylor, 1990), ABAQUS, etc., for analyzing a variety of problems and for incorporating new elements is now already an everyday practice. The general finite element environments can handle, regardless of the type of elements, most of the required operations such as: pre-processing of the input data, manipulating and organizing of the data related to nodes and elements, material characteristics, displacements and stresses, construction of the global matrices by invoking different elements subroutines, solving the system of equations, post-processing and analysis of results. However, large FE systems can be for the development and testing of new numerical procedures awkward. 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 the general symbolic-numeric environments such as Mathematica, Maple, etc. It is well known that many design flaws such as element instabilities or poor convergence properties can be easily identified if we are able to investigate element quantities on a symbolic level. Unfortunately, symbolic-numeric environments become very inefficient if there is a larger number of elements or if we have to perform iterative numerical procedures. In order to assess element performances under real conditions the easiest way is to perform tests on sequential machines with good debugging capabilities (typically personal computers and programs written in Fortran or C/C++ language). In the end, for real industrial simulations, large parallel machines have to be used. By the classical approach, re-coding of the element in different languages would be extremely time consuming and is never done. With the symbolic concepts re-coding comes practically for free, since the code is automatically generated for several languages and for several platforms from the same basic symbolic description.

The AceGen package provides a collection of prearranged modules for the automatic creation of the interface between the finite element code and the finite element environment. AceGen enables multi-language and multi-environment generation of nonlinear finite element codes from the same symbolic description. The AceGen system currently supports the following FE environments:

- **AceFem** is a model FE environment written in a Mathematica’s symbolic language and C (see AceFEM),
- **FEAP** is the research environment written in FORTRAN (see FEAP – ELFEN – ABAQUS – ANSYS),
- **ELFEN** is the commercial environment written in FORTRAN (see FEAP – ELFEN – ABAQUS – ANSYS),
- **ABAQUS** is the commercial environment written in FORTRAN (see FEAP – ELFEN – ABAQUS – ANSYS).

The AceGen package is often used to generate user subroutines for various other environments. It is advisable for the user to use standardized interface as described in User Defined Environment Interface.

There are several benefits of using the standardized interface:

- automatic translation to other FE packages,
other researchers are able to repeat the results,
commercialization of the research is easier,
eventually, the users interface can be added to the list of standard interfaces.

The number of numerical environments supported by AceGen system is a growing daily. Please visit the www.fgg.uni-lj.si/symech/extensions/ page to see if the numerical environment you are using is already supported or www.fgg.uni-lj.si/consulting/ to order creation of the interface for your specific environment.

All FE environments are essentially treated in the same way. Additional interface code ensures proper data passing to and from automatically generated code for those systems. Interfacing the automatically generated code and FE environment is a two stage process. The purpose of the process is to generate element codes for various languages and environments from the same symbolic input. At the first stage user subroutine codes are generated. Each user subroutine performs specific task (see StandardUserSubroutines). The input/output arguments of the generated subroutines are environment and language dependent, however they should contain the same information. Due to the fundamental differences among FE environments, the required information is not readily available. Thus, at the second stage the contents of the "splice-file" (see SMSWrite) that contains additional environment dependent interface and supplementary routines is added to the user subroutines codes. The "splice-file" code ensures proper data transfer from the environment to the user subroutine and back.

Automatic interface is already available for a collection of basic tasks required in the finite element analysis (see StandardUserSubroutines). There are several possibilities in the case of need for an additional functionality. Standard User Subroutines can be used as templates by giving them a new name and, if necessary, additional arguments. The additional subroutines can be called directly from the environment or from the enhanced "splice-file". Source code of the "splice-files" for all supported environments are available at directory $BaseDirectory/Applications/AceGen/Splice/. The additional subroutines can be generated independently just by using the AceGen code generator and called directly from the environment or from the enhanced "splice-file".

Since the complexity of the problem description mostly appears in a symbolic input, we can keep the number of data structures (see DataStructures) that appear as arguments of the user subroutines at minimum. The structure of the data is depicted below. If the "default form" of the arguments as external AceGen variables (see Symbolic-Numeric Interface) is used, then they are automatically transformed into the form that is correct for the selected FE environment. The basic data structures are as follows:

- environment data defines a general information common to all nodes and elements (see Integer Type Environment Data, Real Type Environment Data),
nodal data structure contains all the data that is associated with the node (see Node Data),

- element specification data structure contains information common for all elements of particular type (see Domain Specification Data),

- node specification data structure contains information common for all nodes of particular type (see Node Specification Data),

- element data structure contains all the data that is associated with the specific element (see Element Data).

Standard FE Procedure

Description of FE Characteristic Steps

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

A) **AceGen initialization**
- see SMSInitialize

B) **Template initialization**
- see SMSTemplate
  - general characteristics of the element
  - rules for symbolic-numeric interface

C) **Definition of user subroutines**
- see Standard User Subroutines
  - tangent matrix, residual, post-processing, ...

D) **Code generation**
- see SMSWrite
  - additional environment subroutines
  - compilation, dll, ...

Due to the advantage of simultaneous optimization procedure we can execute each step separately and examine intermediate results. This is also the basic way how to trace the errors that might occur during the AceGen session.

Description of Introductory Example

Let us consider a simple example to illustrate the standard AceGen procedure for the generation and testing of a typical finite element. The problem considered is steady-state heat conduction on a three-dimensional domain, defined by:

\[
\frac{\partial}{\partial x} \left( k \frac{\partial \phi}{\partial x} \right) + \frac{\partial}{\partial y} \left( k \frac{\partial \phi}{\partial y} \right) + \frac{\partial}{\partial z} \left( k \frac{\partial \phi}{\partial z} \right) + Q = 0 \quad \text{on domain } \Omega,
\]

\[
\phi - \bar{\phi} = 0 \quad \text{essential boundary condition on } \Gamma_{\phi},
\]

\[
k \frac{\partial \phi}{\partial n} - \bar{q} = 0 \quad \text{natural boundary condition on } \Gamma_{q},
\]

where \( \phi \) indicates temperature, \( k \) is conductivity, \( Q \) heat generation per unit volume, and \( \bar{\phi} \) and \( \bar{q} \) are the prescribed values of temperature and heat flux on the boundaries. Thermal conductivity here is assumed to be a quadratic function of temperature:

\[
k = k_0 + k_1 \phi + k_2 \phi^2.
\]

Corresponding weak form is obtained directly by the standard Galerkin approach as

\[
\int_\Omega \left[ v^T \delta \phi \right] k \left[ \nabla \phi - \delta \phi Q \right] d\Omega - \int_{\Gamma_q} \delta \phi \bar{q} d\Gamma = 0.
\]

Only the generation of the element subroutine that is required for the direct, implicit analysis of the problem is presented here. Additional user subroutines may be required for other tasks such as sensitivity analysis, post-processing etc.. The problem considered is non-linear and it has unsymmetric Jacobian matrix.
Data interface

Interface to the input data of the element user subroutines can be created by using:

- high level IO data management routines specifically designed for FEM interface (see IO Data Management)
- or low level direct specification of real and integer type variables with SMSReal and SMSInteger AceGen commands.

High level IO Data Management routines are used first to build the interface. The input is repeated later by using low level interface.

AceGen input using high level I/O data management

Step 1: Initialization

- This loads the AceGen code generator.

```
In[172]:= << AceGen;
```

- This initializes the AceGen session. The AceFEM is chosen as the target numerical environment. See also SMSInitialize.

```
In[173]:= SMSInitialize["ExamplesHeatConduction", "Environment" -> "AceFEM"];
```

- This initializes constants that are needed for proper symbolic-numeric interface (See Template Constants). Three-dimensional, eight node, hexahedron element with one degree of freedom per node is initialized.

```
In[174]:= SMSTemplate["SMSTopology" -> "H1", "SMSDOFGlobal" -> 1, 
    "SMSymmetricTangent" -> False, 
    "SMSDomainDataNames" -> {"k0 - conductivity parameter", "k1 - conductivity parameter", 
    "k2 - conductivity parameter", "Q - heat source"}, 
    "SMSDefaultData" -> {1, 0, 0, 0}];
```

Step 2: Element subroutine for the evaluation of tangent matrix and residual

- Start of the definition of the user subroutine for the calculation of tangent matrix and residual vector and set up input/output parameters (see Standard User Subroutines).

```
In[175]:= SMSStandardModule["Tangent and residual"];
```

Step 3: Interface to the input data of the element subroutine

- Here the coordinates of the element nodes and current values of the nodal temperatures are taken from the supplied arguments of the subroutine. The default values for "H1" topology are used to set the number of nodes and the type of nodes and SMSDOF-Global is used to get the number of DOFs per node.

```
In[176]:= {XIO, ϕIO} ⊨ SMSIO["All coordinates and DOFs"]; 
ϕI = ϕIO[[All, 1]];
```

- The conductivity parameters k0, k1, k2 and the internal heat source Q are assumed to be common for all elements in a particular domain (material or group data). Thus they are placed into the element specification data field "Data" (see Element Data). In the case that material characteristic vary substantially over the domain it is better to use element data field "Data" instead of element specification data.

```
In[177]:= {k0, k1, k2, Q} ⊨ SMSIO["Domain data"];
```

- Element is numerically integrated by one of the built-in standard numerical integration rules (see Numerical Integration). This starts the loop over the integration points, where ξ, η, ζ are coordinates of the current integration point and wGauss is integration point weight.

```
In[178]:= SMSDo[IG, 1, SMSIO["No. integration points"]]; 
Ξ = (ξ, η, ζ) ⊨ SMSIO["Integration point"[IG]];
```

Step 4: Definition of the trial functions

- This defines the trilinear shape functions Ni, i=1,2,…,8 and interpolation of the physical coordinates within the element. Jm is Jacobian matrix of the isoparametric mapping from actual coordinate system X, Y, Z to reference coordinates ξ, η, ζ.
The trial function for the temperature distribution within the element is given as linear combination of the shape functions and the nodal temperatures \( \phi = N_n \). The \( \phi_n \) are unknown parameters of the variational problem.

The implicit dependencies between the actual and the reference coordinates are given by \( \frac{\partial \xi_j}{\partial x_i} = J_{m-1} \frac{\partial x_i}{\partial \xi_j} \), where \( J_m \) is the Jacobean matrix of the nonlinear coordinate mapping.

Here is the definition of the weak form of the steady state heat conduction equations. The strength of the heat source is multiplied by the global variable \( rdata"Multiplier" \) (see \( RealTypeEnvironmentData \)).

This evaluates the explicit form of the Jacobian (tangent) matrix and exports result into the \( sSS \) output parameter of the "Tangent and residual" subroutine (see \( StandardUserSubroutines \)).

This is the end of the integration loop.
Step 7: Post-processing subroutine

- Start of the definition of the user subroutine for the definition and evaluation of post-processing quantities. The subroutine is not obligatory, however it makes the post-processing much easier.

```plaintext
In[197]:= SMSStandardModule["Postprocessing"];
```

- Here the nodal point post-processing quantity "Temperature" is introduced and exported to array of the nodal point quantities \( n_{\text{post}} \).

```plaintext
In[198]:= {XIO, \( \phi \)IO} \[\subseteq\] SMSIO[{"All coordinates and DOFs"];
\( \phi \)I = \( \phi \)IO[{{-1,-1,-1}, {1,1,-1}, {-1,1,1}, {-1,-1,1}}];
```

- Here the integration point post-processing quantity "Conductivity" is introduced and exported to array of the integration point quantities \( g_{\text{post}} \).

```plaintext
In[202]:= {k0, k1, k2, Q} \[\subseteq\] SMSIO[{"Domain data"];
```

Step 8: Code Generation

- At the end of the session AceGen translates the code from pseudo-code to the required script or compiled program language and prepends the contest of the interface file to the generated code. See also SMSWrite. The result is ExamplesHeatConduction.c file with the element source code written in a C language.

```plaintext
In[204]:= SMSWrite[];
```

<table>
<thead>
<tr>
<th>Method</th>
<th>SKR</th>
<th>SPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>No.Formulae</td>
<td>243</td>
<td>27</td>
</tr>
<tr>
<td>No.Leafs</td>
<td>4862</td>
<td>366</td>
</tr>
</tbody>
</table>

AceGen input using low level I/O data management

Most of the AceGen code is the same as in the previous example, thus it will not be commented again.

Step 1: Initialization

```plaintext
<< AceGen;
SMSInitialize["tmpExamplesHeatConduction", "Environment" -> "AceFEM"];
SMSTemplate["SMSTopology" -> "H1", "SMSDOFGlobal" -> 1, 
"SMSSymmetricTangent" -> False, 
"SMSDomainDataNames" -> {"k0 -conductorivity parameter", "k1 -conductorivity parameter", 
"k2 -conductorivity parameter", "Q -heat source"}, 
"SMSDefaultData" -> {1, 0, 0, 0}];
```

Step 2: Element subroutine for the evaluation of tangent matrix and residual

```plaintext
In[203]:= SMSStandardModule["Tangent and residual"];
```
Step 3: Interface to the input data of the element subroutine

- Here the coordinates of the element nodes and current values of the nodal temperatures are taken from the supplied arguments of the subroutine \texttt{nd[i,"X",j]} and \texttt{nd[i,"at",1]} (see Node Data).

- The conductivity parameters \(k_0, k_1, k_2\) and the internal heat source \(Q\) are assumed to be common for all elements in a particular domain (material or group data). Thus they are placed into the element specification data field "Data" (see Element Data).

- Element is numerically integrated by one of the built-in standard numerical integration rules (see Numerical Integration). This starts the loop over the integration points, where \(\xi, \eta, \zeta\) are coordinates of the current integration point and \(w_{\text{Gauss}}\) is integration point weight.

Step 4: Definition of the trial functions

- The strength of the heat source is multiplied by the global variable \texttt{rdata["Multiplier"]} (see Real Type Environment Data).

Step 5: Definition of the governing equations

Step 6: Definition of the Jacobian matrix
Step 7: Post-processing subroutine

In[229]:= SMSStandardModule["Postprocessing"];
ϕI[ salarié $i$, "at", 1], {i, 8}];
SMSPostNames = {"Temperature"};
SMSExport[ϕI, Table[nsalari[$i$, 1], {i, 8}]];
{k0, k1, k2, Q} ⊢ SMSReal[Table[es[$"Data", i], {i, Length[SMSDomainDataNames]]}]];
SMDO[
Ξ = {ξ, η, ζ} ⊢ SMSReal[es["IntPoints", i, Ig]]

Step 8: Code Generation

File: tmpExamplesHeatConduction.c  Size: 16977  Time: 6

<table>
<thead>
<tr>
<th>Method</th>
<th>SKR</th>
<th>SPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. Formulae</td>
<td>242</td>
<td>26</td>
</tr>
<tr>
<td>No. Leafs</td>
<td>4856</td>
<td>360</td>
</tr>
</tbody>
</table>
Template Constants

Contents

- General Description
  - SMSTemplate
- Template Constants
  - Geometry
  - Degrees of Freedom; K and R
  - Data Management
  - Graphics and Postprocessing
  - Sensitivity Analysis
  - AceFEM Solution Procedure Specific
  - Description of the Element for AceShare
  - Environment Specific

General Description

The AceGen uses a set of global constants that at the code generation phase define the major characteristics of the finite element (called finite element template constants). In most cases the element topology and the number of nodal degrees of freedom are sufficient to generate a proper interface code. Some of the FE environments do not support all the possibilities given here. The AceGen tries to accommodate the differences and always generates the code. However, if the proper interface can not be done automatically, then it is left to the user. For some environments additional constants have to be declared (see e.g. ELFEN).

The template constants are initialized with the SMSTemplate function. Values of the constants can be also set or changed directly after SMSTemplate command.

SMSTemplate

```
SMSTemplate[options]
```

initializes constants that are needed for proper symbolic-numeric interface for the chosen numerical environment

The general characteristics of the element are specified by the set of options options. Options are of the form "Element_constant"->value (see also Template Constants for list of all constants). The SMSTemplate command must follow the SMSInitialize command.

See also Template Constants section for a list of all constants and the Data Structures section to see how template constants relate to the external variables in AceGen and the data manipulation routines in AceFEM.

- This defines the 2D, quadrilateral element with 4 nodes and 5 degrees of freedom per node.

```
SMSTemplate["SMSTopology" -> "Q1", "SMSDOFGlobal" -> 5];
```
### Template Constants

#### Geometry

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMSSTopology</td>
<td>default</td>
<td>element topology (see Element Topology)</td>
</tr>
<tr>
<td>SMSNoNodes</td>
<td>Automatic</td>
<td>number of nodes (only if different than the one defined by element topology, see Node Identification)</td>
</tr>
</tbody>
</table>
| SMSAdditionalNodes  | Function[{},{}] | The argument can be:  
  a) a pure function (see Function) that defines the transformation of the mesh (mesh transformation function). Function must return additional nodes or defines transformation of the existing nodes as defined in Self-transforming meshes.  
  b) a constant Null in the case of an arbitrary number of auxiliary nodes. If the Null is given instead of the coordinates then the node will have no association with specific spatial position. Examples are given in Self-transforming meshes. Additional nodes can be topological (see Cubic triangle, Additional nodes) or auxiliary nodes (see Mixed 3 D Solid FE, Auxiliary Nodes). |
| SMSNodeID           | Table["D", SMSNoNodes] | for all nodes a keyword that is used for identification of the nodes in the case of multi-field problems (see Node Identification) |
| SMSNoDimensions     | Automatic | number of spatial dimensions (only in the case of "XX" topology)                               |
| SMSNodeOrder        | Automatic | relative ordering of the nodes with respect to the standard ordering defined in Element Topology |

#### Degrees of Freedom, K and R

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMSDOFGlobal</td>
<td>Table[ SMSNoDimensions, SMSNoNodes ]</td>
<td>number of d.o.f per node for all nodes</td>
</tr>
</tbody>
</table>
| SMSSymmetricTangent   | True    | True ⇒ tangent matrix is symmetric  
  False ⇒ tangent matrix is unsymmetrical                                                            |
| SMSNoDOFCondense      | 0       | number of d.o.f that have to be condensed before the element quantities are assembled (see Elimination of Local Unknowns, Mixed 3 D Solid FE, Elimination of Local Unknowns) |
| SMSCondensationData   |         | storage scheme for local condensation (see Elimination of Local Unknowns)                      |
| SMResidualSign        | Automatic | 1 ⇒ equations are formed in the form $K a + \Phi = 0$  
  -1 ⇒ equations are formed in the form $K a = \Phi$ (used to ensure compatibility between the numerical environments) |
<p>| SMSDefaultIntegrationCode | Automatic | default numerical integration code (see Numerical Integration)                                |
| SMSNoDOFGlobal        | calculated value | total number of global d.o.f.                                                                 |
| SMSNoAllDOF           | calculated value | total number of all d.o.f.                                                                     |
| SMSMaxNoDOFNode       | calculated value | maximum number of d.o.f. per node                                                               |</p>
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMSDomainDataNames</td>
<td>{}</td>
<td>description of the input data values that are common for all elements with the same element specification (e.g. material characteristics) (defines the dimension of the es[&quot;Data&quot;,j] data field)</td>
</tr>
<tr>
<td>SMSDefaultData</td>
<td>Table[0., SMSDomainDataNames//Length]</td>
<td>default values for input data values</td>
</tr>
<tr>
<td>SMSDataCheck</td>
<td>True</td>
<td>logical expression that checks the correctness of the user supplied constants stored in es[&quot;Data&quot;,i]. It should return True if the data is correct.</td>
</tr>
<tr>
<td>SMSNoTimeStorage</td>
<td>0</td>
<td>total number of history dependent real type values per element that have to be stored in the memory for transient type of problems (defines the dimension of the ed[&quot;ht&quot;,j] and ed[&quot;hp&quot;,j] data fields)</td>
</tr>
<tr>
<td>SMSNoElementData</td>
<td>0</td>
<td>total number of arbitrary real values per element (defines the dimension of the ed[&quot;Data&quot;,j] data field)</td>
</tr>
<tr>
<td>SMSNoNodeStorage</td>
<td>Table[0, SMSNoNodes]</td>
<td>total number of history dependent real type values per node that have to be stored in the memory for transient type of problems (can be different for each node) (defines the dimension of the nd[i,&quot;ht&quot;,j] and nd[i,&quot;hp&quot;,j] data fields)</td>
</tr>
<tr>
<td>SMSNoNodeData</td>
<td>0</td>
<td>total number of arbitrary real values per node (can be different for each node) (defines the dimension of the nd[i,&quot;Data&quot;,j] data field)</td>
</tr>
<tr>
<td>SMSIDataNames</td>
<td>{}</td>
<td>list of the keywords of additional integer type environment data variables (global), see Integer Type Environment Data</td>
</tr>
<tr>
<td>SMSRDataNames</td>
<td>{}</td>
<td>list of the keywords of additional real type environment data variables (global), see Real Type Environment Data</td>
</tr>
<tr>
<td>SMSNoAdditionalData</td>
<td>0</td>
<td>number of additional input data values that are common for all elements with the same element specification (the value can be expression) (defines the dimension of the es[&quot;AdditionalData&quot;,i] data field)</td>
</tr>
<tr>
<td>SMSCharSwitch</td>
<td>{}</td>
<td>list of character type user defined constants (local)</td>
</tr>
<tr>
<td>SMSIntSwitch</td>
<td>{}</td>
<td>list of integer type user defined constants (local)</td>
</tr>
<tr>
<td>SMSDoubleSwitch</td>
<td>{}</td>
<td>list of double type user defined constants (local)</td>
</tr>
</tbody>
</table>
Graphics and Postprocessing

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMSGPostNames</td>
<td>{}</td>
<td>description of the post-processing quantities defined per material point (see Subroutine: Postprocessing)</td>
</tr>
<tr>
<td>SMSNPostNames</td>
<td>{}</td>
<td>description of the post-processing quantities defined per node (see Subroutine: Postprocessing)</td>
</tr>
<tr>
<td>SMSSegments</td>
<td>Automatic</td>
<td>for all segments on the surface of the element the sequence of the element node indices that define the edge of the segment (if possible the numbering of the nodes should be done in a way that the normal on a surface of the segment represents the outer normal of the element)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SMSSegments={1,2,3,4} ... no post-processing (see example Cubic triangle, Additional nodes)</td>
</tr>
<tr>
<td>SMSSegmentsTriangulation</td>
<td>Automatic</td>
<td>for all segments define a rule that splits the segments specified by SMSSegments into triangular or quadrilateral subsegments (the data is used to color the interior of the segments and post-processing of field variables)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SMSSegments={{1,2,3},{1,3,4}} ... no field post-processing (see example Cubic triangle, Additional nodes)</td>
</tr>
<tr>
<td>SMSReferenceNodes</td>
<td>Automatic</td>
<td>coordinates of the nodes in the reference coordinate system in the case of elements with variable number of nodes (used in post processing, Subroutine: Postprocessing)</td>
</tr>
<tr>
<td>SMSPostNodeWeights</td>
<td>Automatic</td>
<td>additional weights associated with element nodes and used for post-processing of the results (see SMTPostData, Subroutine: Postprocessing). In general, the weight of the nodes that form the segments is 1 and for the others is 0.</td>
</tr>
<tr>
<td>SMSAdditionalGraphics</td>
<td>{Null,Null,Null}</td>
<td>A set of three pure functions {addGraphicsPrimitives, trueSegments, trueTriangulation} (see Function) that are called for each element as described below or Null if additional graphics is not defined for specific element type. (see also SMTUpdatePostData)</td>
</tr>
</tbody>
</table>

Function[{},{}] = {Function[{},{}], Null, Null}

SMSAdditionalGraphics option defines a set of three pure functions {addGraphicsPrimitives, trueSegments, trueTriangulation}. Functions are called before the SMTShowMesh creates the post-processing mesh for each element. If the node coordinates or the element connectivity has been changed then the SMTUpdatePostData command has to be called first.

The results are used as follows:

- **addGraphicsPrimitives**
  
  addGraphicsPrimitives[
  (element index, domain index,list of node indices)
  ,is True if node marks are required
  ,is True if boundary conditions are required
  ,(node coordinates for all element nodes)
  ]

  must returns additional graphics primitives (Line, Point, etc.) that will be included into the final graphics verbatim (e.g. Function[{{1,1,1},{1,2,2}}]) would produce a line connecting first and second element node.

- **trueSegments**

  trueSegments[element index, domain index,list of node indices]

  must return all segments on the surface of the specific element the sequence of the element node indices that define the edge of the segment (e.g. {{1,2,3,4}})
trueTriangulation
trueTriangulation[element index, domain index, list of node indices]
for all segments define a rule that splits the segments specified by trueSegments into triangular or quadrilateral sub-segments (e.g. {{{1,2,3},{1,3,4}}})

Sensitivity Analysis

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMSSensitivityNames</td>
<td>{}</td>
<td>description of sensitivity dependent input parameters (e.g. spatial coordinates, material parameters) for which sensitivity pseudo-load code is derived</td>
</tr>
<tr>
<td>SMSShapeSensitivity</td>
<td>False</td>
<td>True ⇒ shape sensitivity pseudo-load code is supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>False ⇒ shape sensitivity is not supported (informative data)</td>
</tr>
<tr>
<td>SMSEBCSensitivity</td>
<td>False</td>
<td>True ⇒ essential boundary condition sensitivity pseudo-load code is supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>False ⇒ essential boundary condition sensitivity is not supported (informative data)</td>
</tr>
</tbody>
</table>
| SMSEExtraSensitivityData      | {0}     | {The length of additional data storage per element needed for the execution of backward mode sensitivity analysis}
                               | finances Backward mode for time dependent locally coupled problems |

See also AceFEM manual sections Sensitivity Analysis.

AceFEM Solution Procedure Specific

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
</table>
| SMSMMAInitialisation         | {Hold[Null], Hold[Null]} | {beforeMesh, afterMesh}
                               | beforeMesh is a list of arbitrary length Mathematica’s codes and additional function definitions executed before the mesh generation |
                               |         | afterMesh is a list of arbitrary length Mathematica’s codes and additional function definitions executed after the mesh generation (wrapping the code in Hold prevents evaluation) |
                               |         | afterMesh equivalent to {Hold[Null], afterMesh}                              |
| SMSMMANextStep               | Hold[Null] | short Mathematica’s code executed after SMTNextStep command (wrapping the code in Hold prevents evaluation) |
| SMSMMAStepBack               | Hold[Null] | short Mathematica’s code executed after SMTStepBack command (wrapping the code in Hold prevents evaluation) |
| SMSMMAPreIteration           | Hold[Null] | short Mathematica’s code executed before SMTNextStep command (wrapping the code in Hold prevents evaluation) |
| SMSPreIterationCall          | False   | force one additional call of the SKR user subroutines after the convergence of the global solution has been archived in order to improve accuracy of the solution of additional algebraic equations at the element level (see Three Dimensional, Elasto-Plastic Element) |
| SMDDOFScaling                | False   | True ⇒ scaling of DOF is allowed                                           |
|                               |         | False ⇒ scaling of DOF is not allowed                                     |
### Description of the Element for AceShare

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMSMainTitle</td>
<td>&quot;&quot;&quot;&quot;</td>
<td>description of the element (see SMSVerbatim how to insert special characters such as \n or &quot;)</td>
</tr>
<tr>
<td>SMSSubTitle</td>
<td>&quot;&quot;&quot;&quot;</td>
<td>description of the element</td>
</tr>
<tr>
<td>SMSSubSubTitle</td>
<td>&quot;&quot;&quot;&quot;</td>
<td>description of the element</td>
</tr>
<tr>
<td>SMSBibliography</td>
<td>&quot;&quot;&quot;&quot;</td>
<td>reference</td>
</tr>
</tbody>
</table>

### Environment Specific (FEAP, ELFEN, user defined environments, …)

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMSUserSubroutine</td>
<td>None</td>
<td>generation of environment specific user subroutines (e.g. ABAQUS or ANSYS UMAT, (see FEAP – ELFEN – ABAQUS – ANSYS ))</td>
</tr>
<tr>
<td>SMSTemplateFile</td>
<td>None</td>
<td>an alternative template file for the generation of AceGen - FE environment interface</td>
</tr>
<tr>
<td>SMSUserDataRules</td>
<td>{}</td>
<td>user defined replacement rules that transform standard input/output parameters to user defined input/output parameters (see also User Defined Environment Interface)</td>
</tr>
<tr>
<td>FEAP$*</td>
<td>&quot;&quot;&quot;&quot;</td>
<td>FEAP specific template constants described in chapter FEAP (see FEAP – ELFEN – ABAQUS – ANSYS)</td>
</tr>
<tr>
<td>ELFEN$*</td>
<td>&quot;&quot;&quot;&quot;</td>
<td>ELFEN specific template constants described in chapter ELFEN (see FEAP – ELFEN – ABAQUS – ANSYS)</td>
</tr>
<tr>
<td>SMSBibliography</td>
<td>&quot;&quot;&quot;&quot;</td>
<td>reference</td>
</tr>
</tbody>
</table>

Constants defining the general element characteristics.
Data Structures

Contents

- General Description
- IO Data Management
  - General Input / Output data
  - Sensitivity analysis Input / Output data
  - Tasks Input / Output data
  - Example: IO data defined by SMSTemplate constants
  - Dynamically created IO data variables
  - IO data types
    - Example: dynamically created IO data variables
    - Example: dynamically created multi-field element IO
    - Example: low-level IO data definitions
  - Utility IO data management commands
    - SMSIO
    - SMSIOInfo
    - SMSUndefineIO
- Integer Type Environment Data
  - General data
    - Mesh input related data
      - Iterative procedure related data
      - Debugging and errors related data
      - Linear solver related data
      - Sensitivity related data
      - Contact related data
  - Real Type Environment Data
- Node Specification Data
- Node Data
- Domain Specification Data
  - Memory allocation
  - General Data
    - Run Mathematica from code
    - Mesh generation
    - Domain input data
    - Numerical integration
    - Graphics postprocessing
Sensitivity analysis

Element Data

General Description

Environment data structure defines the general information common for all nodes and elements of the problem. If the “default form” of the data is used, then AceGen automatically transforms the input into the form that is correct for the selected FE environment. The environment data are stored into two vectors, one for the integer type values (Integer Type Environment Data) and the other for the real type values (Real Type Environment Data). All the environments do not provide all the data, thus automatic translation mechanism can sometimes fail. All data can be in general divided into 6 data structures:

- Integer Type Environment Data (in AceGen idata$$, in AceFEM SMTIData)
- Real Type Environment Data (in AceGen rdata$$, in AceFEM SMTRData)
- Domain Specification Data (in AceGen es$$, in AceFEM SMTDomainData)
- Element Data (in AceGen ed$$, in AceFEM SMTElementData)
- Node Specification Data (in AceGen ns$$, in AceFEM SMTNodeSpecData)
- Node Data (in AceGen nd$$, in AceFEM SMTNodeData)

Node Data Structures

Two types of the node specific data structures are defined. The structure (Node Specification Data, ns$$) defines the major characteristics of the nodes sharing the same node identification (NodeID, Node Identification). Nodal data structure (Node Data, nd$$) contains all the data that are associated with specific node. Nodal data structure can be set and accessed from the element code. For example, the command SMSReal[nd$$[i,"X",1]] returns x-coordinate of the i-th element node. At the analysis phase the data can be set and accessed interactively from the Mathematica by the user (see SMTNodeData, SMTElementData...). The data are always valid for the current element that has been processed by the FE environment. Index i is the index of the node accordingly to the definition of the particular element.

Element Data Structures

Two types of the element specific data structures are defined. The domain specification data structure (Domain Specification Data, es$$) defines the major characteristics of the element that is used to discretize particular sub-domain of the problem. It can also contain the data that are common for all elements of the domain (e.g. material constants). The element data structure (Element Data, ed$$) holds the data that are specific for each element in the mesh.

For a transient problems several sets of element dependent transient variables have to be stored. Typically there can be two sets: the current (ht) and the previous (hp) values of the transient variables. The hp and ht data are switched at the beginning of a new step (see SMTNextStep).

All element data structures can be set and accessed from the element code. For example, the command SMSInteger[ed$$["nodes",1]] returns the index of the first element node. The data is always valid for the current element that has been processed by the FE environment.

IO Data management

IO data management routines provide higher level IO data management based on textual description of data required.

IO Data Management

IO data management routines provide keyword based interface between the input output parameters of user subroutine and AceGen session. Whenever several user subroutines are required, the allocated data structures have to be kept synchronized throughout the AceGen session. AceGen enables three levels of IO data management depending on the complexity of the element.

- Level 1: IO data defined by SMSTemplate constants
For a simple elements all the information needed for the correct formulation of IO data variables is defined by \texttt{SMSTemplate} constants (\texttt{SMSTopology, SMSAdditionalNodes, SMSNodeID, SMSDOFGlobal, SMSDomainDataNames}, etc.). Specific data is identified by its name \texttt{(dataName)} defined in \texttt{(General Input / Output data, Sensitivity analysis Input / Output data, Tasks Input / Output data)} and it can be imported form or exported to IO parameters of corresponding user subroutine with the following commands:

<table>
<thead>
<tr>
<th>form</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{SMSIO[dataName]}</td>
<td>returns all data associated with the \texttt{dataName} (scalar, vector of matrix)</td>
</tr>
<tr>
<td>\texttt{SMSIO[dataName[index]]}</td>
<td>if \texttt{dataName} defines a vector commands returns \texttt{index}-th element of vector</td>
</tr>
<tr>
<td>\texttt{SMSIO[dataName[indexes_List]]}</td>
<td>if \texttt{dataName} defines a vector commands returns a subset of elements defined by a set of indexes</td>
</tr>
<tr>
<td>\texttt{SMSIO[value, &quot;Add to&quot;, dataName]}</td>
<td>adds the value (scalar, vector or matrix) to the output parameter of the subroutine defined by the \texttt{dataName}</td>
</tr>
<tr>
<td>\texttt{SMSIO[value, &quot;Add to&quot;, dataName[index]]}</td>
<td>adds the value to the \texttt{index}-th element of the vector defined by the \texttt{dataName}</td>
</tr>
<tr>
<td>\texttt{SMSIO[values, &quot;Add to&quot;, dataName[indexes_List]]}</td>
<td>adds a set of values to the a subset of elements of vectors defined by a set of indexes</td>
</tr>
<tr>
<td>\texttt{SMSIO[value, &quot;Add to&quot;, dataName[indexi,indexj]]}</td>
<td>adds the value to the \texttt{[indexi-th,indexj-th]} element of matrix</td>
</tr>
<tr>
<td>\texttt{SMSIO[value, &quot;Export to&quot;, dataName]}</td>
<td>exports the value (scalar, vector or matrix) to the output parameter of the subroutine defined by the \texttt{dataName}</td>
</tr>
<tr>
<td>\texttt{SMSIO[value, &quot;Export to&quot;, dataName[index]]}</td>
<td>exports the value to the \texttt{index}-th element of the vector defined by the \texttt{dataName}</td>
</tr>
<tr>
<td>\texttt{SMSIO[values, &quot;Export to&quot;, dataName[indexes_List]]}</td>
<td>exports the value (scalar, vector or matrix) to the output parameter of the subroutine defined by the \texttt{dataName}</td>
</tr>
<tr>
<td>\texttt{SMSIO[value, &quot;Export to&quot;, dataName[indexi,indexj]]}</td>
<td>exports the value (scalar, vector or matrix) to the output parameter of the subroutine defined by the \texttt{dataName}</td>
</tr>
</tbody>
</table>

Syntax of generic SMSIO commands for static exporting and importing of IO data.

\begin{itemize}
  \item Level 2: dynamically allocated IO data variables (Dynamically created IO data variables)
  
  In the case of complex multi-field numerical models the definition of IO variables becomes prone to errors. The primal purpose of the mechanism for the automatic \texttt{IO Data Management} is dynamic allocation of data of given type and association of the data with the keys. The allocated data is kept synchronized throughout the AceGen session based on the given keys. In this case the \texttt{SMSTemplate} constants (\texttt{SMSTopology, SMSAdditionalNodes, SMSNodeID, SMSDOFGlobal, SMSDomainDataNames, etc.}) are automatically updated and they don't have to be defined with the \texttt{SMSTemplate} command.

<table>
<thead>
<tr>
<th>form</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{SMSIO[dataName]}</td>
<td>returns all data associated with the \texttt{dataName} (scalar, vector of matrix)</td>
</tr>
<tr>
<td>\texttt{SMSIO[dataType, key-&gt;addData]}</td>
<td>creates a new auxiliary variable of the type associated with \texttt{dataName} and associates the value and the corresponding additional \texttt{addData} to the \texttt{key}</td>
</tr>
<tr>
<td>\texttt{SMSIO[value, &quot;Add to&quot;, dataType, key]}</td>
<td>adds the value (scalar or vector) to the output parameters of the subroutine defined by \texttt{dataType} and \texttt{key}</td>
</tr>
<tr>
<td>\texttt{SMSIO[value, &quot;Export to&quot;, dataType, key]}</td>
<td>exports the value (scalar or vector) to the output parameters of the subroutine defined by \texttt{dataType} and \texttt{key}</td>
</tr>
</tbody>
</table>

  Syntax of generic SMSIO commands for dynamic creation and exporting of IO data.

  \begin{itemize}
    \item Level 3: low-level definition of IO data definition (Example: low-level IO data definitions)
  \end{itemize}
Number of possible input/output data combinations is huge, thus only the most basic are supported by automatic IO Data management. IO data of complex finite elements have to be defined directly using SMSInteger, SMSReal and SMSExport functions.

**General Input/Output data**

A selected set of data can be access thought keywords (e.g. "Integration point")using the SMSIO command. IO data not covered by SMSIO command have to be defined directly using SMSInteger, SMSReal and SMSExport functions and actual appearance of the IO parameter as a part of user subroutine input/output parameters (e.g. esS"["IntPoints",coordinate,Ig]).

<table>
<thead>
<tr>
<th>form</th>
<th>I/O parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Type Environment Data</td>
<td>iData$$[idataCode]</td>
<td>returns an integer variable holding iData$$ with the given code iData$$ (see Integer Type Environment Data)</td>
</tr>
<tr>
<td>Real Type Environment Data</td>
<td>rData$$[rdataCode]</td>
<td>returns an real variable holding rData$$ with the given code rData$$ (see Real Type Environment Data)</td>
</tr>
</tbody>
</table>

selected set of Domain Specification Data

<table>
<thead>
<tr>
<th>form</th>
<th>I/O parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMSIO[&quot;Integration point&quot;]</td>
<td>es$$[&quot;IntPoints&quot;,coordinate,Ig]</td>
<td>returns coordinates (ξ,η,ζ) of Ig-th integration point (see Domain Specification Data)</td>
</tr>
<tr>
<td>SMSIO[&quot;Integration weight&quot;]</td>
<td>es$$[&quot;IntPoints&quot;,4,Ig]</td>
<td>returns weight in Ig-th integration point (see Domain Specification Data)</td>
</tr>
<tr>
<td>SMSIO[&quot;No. integration points&quot;]</td>
<td>es$$[&quot;id&quot;,&quot;NoIntPoints&quot;]</td>
<td>returns number of integration points (see Domain Specification Data)</td>
</tr>
<tr>
<td>SMSIO[&quot;Domain data&quot;]</td>
<td>es$$[&quot;Data&quot;,Index]</td>
<td>returns all domain data accordingly to the given SMSDomainDataNames and SMSDefaultData</td>
</tr>
</tbody>
</table>

SMSIO commands for static importing general named IO data.

<table>
<thead>
<tr>
<th>form</th>
<th>I/O parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>selected set of Node Data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMSIO[&quot;All coordinates and DOFs&quot;]</td>
<td>nd$$[node,&quot;X&quot;,coordinate]</td>
<td>returns {all nodes, all DOFs} accordingly to the given element topology, SMSDOFGlobal, SMSNodeID (see Node Data)</td>
</tr>
<tr>
<td>SMSIO[&quot;Nodal coordinates&quot;]</td>
<td>nd$$[node,&quot;X&quot;,coordinate]</td>
<td>returns coordinates of all nodes accordingly to the given element topology and SMSNodeID (each nodes gets associated keyword of the form NodeID[node_index], thus SMSIO[&quot;Nodal coordinates&quot;,NodeID[...]] would return all nodes with node identification NodeID)</td>
</tr>
<tr>
<td>SMSIO[&quot;Nodal coordinates&quot;]</td>
<td>nd$$[node,&quot;X&quot;,coordinate]</td>
<td>returns coordinates of node where node is an arbitrary expressions</td>
</tr>
<tr>
<td>SMSIO[&quot;Nodal coordinates&quot;]</td>
<td>nd$$[node,&quot;X&quot;,coordinate]</td>
<td>returns coordinate-th spatial coordinate in node node where node and coordinate are an arbitrary expressions</td>
</tr>
<tr>
<td>SMSIO[&quot;Nodal coordinates&quot;]</td>
<td>nd$$[node,&quot;X&quot;,coordinate]</td>
<td>returns coordinates of all nodes with node identification NodeID</td>
</tr>
</tbody>
</table>
SMSIO["Nodal DOFs"] nd$$[node, "at", dof] returns all DOFs accordingly to the given SMSDOFGlobal, SMSNodeID (each set of nodal DOFs gets associated keyword of the form NodeID[node_index], thus SMSIO["Nodal DOFs",NodeID_] would return all DOFs in all nodes with node identification NodeID).

SMSIO["Nodal DOFs"[node,dof]] nd$$[node, "at", dof] returns dof-th nodal DOF in node node where node and dof are an arbitrary expressions.

SMSIO["Nodal DOFs"[NodeID]] nd$$[node, "at", dof] returns all nodal DOF of all nodes with node identification NodeID.

SMSIO["Nodal DOFs n"] nd$$[node, "at", dof] returns all DOFs in all nodes with node identification NodeID, thus SMSIO["Nodal DOFs",NodeID_] would return all DOFs in all nodes with node identification NodeID.

SMSIO["Nodal data"] nd$$[node, "Data", index] returns all nodal data accordingly to the given SMSNoNodeData.

SMSIO["Nodal data"[node,index]] nd$$[node, "Data", index] returns index-th nodal data in node node where node and dof are an arbitrary expressions.

SMSIO["Nodal time dependent data"] nd$$[node, "ht", index] returns all nodal time dependent data accordingly to the given SMSNoNodeStorage.

SMSIO["Nodal time dependent data"[node,index]] nd$$[node, "ht", index] returns index-th nodal time dependent data in node node where node and dof are an arbitrary expressions.

SMSIO["Nodal time dependent data n"] nd$$[node, "hp", index] returns all nodal time dependent data accordingly to the given SMSNoNodeStorage at time n.

SMSIO["Nodal time dependent data n"[node,index]] nd$$[node, "hp", index] returns index-th nodal time dependent data at time n where node and dof are an arbitrary expressions.

SMSIO["Element data"] ed$$["Data", index] returns all element data accordingly to the given SMSNoElementData.

SMSIO["Element data"[index]] ed$$["Data", index] returns all element data accordingly to the given SMSNoElementData.

SMSIO["Element time dependent data"] ed$$["ht", index] returns index-th element time dependent data (data can also be dynamically allocated on ht vector, see Dynamically created IO data variables).

SMSIO["Element time dependent data"[index]] ed$$["ht", index] returns all element time dependent data accordingly to the given SMSNoTimeStorage.

SMSIO["Element time dependent data n"] ed$$["hp", index] returns index-th element time dependent data at time n (data can also be dynamically allocated on hp vector, see Dynamically created IO data variables).

SMSIO["Element time dependent data n"[index]] ed$$["hp", index] returns all element time dependent data accordingly to the given SMSNoTimeStorage at time n.

SMSIO["Element data"[index]] ed$$["Data", index] returns index-th element data (data can also be dynamically allocated on Data vector, see Dynamically created IO data variables).

SMSIO["Element data"] ed$$["Data", index] returns all element data accordingly to the given SMSNoElementData.
SMSIO[\text{value}, \text{"Add to"} or \text{"Export to"}, \text{"Residual"}[\text{\textit{index}}]] \quad p\{\text{index}\} \quad \text{adds or exports to residual vector where \textit{value} can be a scalar or a vector and \textit{index} is the position or a list of positions on the residual vector}

SMSIO[\text{value}, \text{"Add to"} or \text{"Export to"}, \text{"Tangent"}[\textit{\textit{indexi}}, \textit{\textit{indexj}}]] \quad s\{\textit{\textit{indexi}}, \textit{\textit{indexj}}\} \quad \text{adds or exports to tangent matrix where \textit{value} can be a scalar, vector or matrix and \textit{\textit{indexi}}, \textit{\textit{indexj}} are the positions or a list of positions on the tangent matrix}

SMSIO[\{	ext{key1} \rightarrow \text{val1}, \text{key2} \rightarrow \text{val2}, \ldots\}, \text{"Export to"}, \text{"Nodal point post"}[\text{\textit{node}}]] \quad \text{export post-processing quantities \{\text{val1}, \text{val2}, \ldots\} to node with the index \textit{\textit{node}} and associate them with the keywords SMSNPostNames={{\text{key1}}, \text{\textit{val2}}, \ldots}\} (see Subroutine: Postprocessing)

SMSIO[\{	ext{key1} \rightarrow \text{\textit{val11}}, \text{key2} \rightarrow \text{\textit{val12}}, \ldots, \text{\textit{\textit{\textit{val_NoNodes}}}}, \ldots\}, \text{"Export to"}, \text{"Nodal point post"}] \quad \text{export post-processing quantities \textit{val11} to \textit{j}-th node and associate them with the \textit{i}-th keywords SMSNPostNames={{\text{key1}}, \text{\textit{val2}}, \ldots}\} (see Subroutine: Postprocessing)

SMSIO[\{	ext{key1} \rightarrow \text{\textit{val1}}, \text{key2} \rightarrow \text{\textit{\textit{val2}}, \ldots, \text{\textit{\textit{\textit{\textit{val_NoNodes}}}}, \ldots\}, \text{"Export to"}, \text{"Integration point post"}[\text{\textit{Ig}}]] \quad \text{export post-processing quantities \textit{val1} to \textit{j}-th node and associate them with the i-th keywords SMSNPostNames={{\text{key1}}, \text{\textit{val2}}, \ldots}\} (see Subroutine: Postprocessing)

**Examples:**

```
SMSIO[\text{"Nodal DOFs"}];
SMSIO[\text{SMSInteger}[2], \text{"Export to"}, \text{"ErrorStatus"}];
SMSIO[Rgi, \text{"Add to"}, \text{"Residual"}[\text{i}]];
SMSIO[Kgij, \text{"Add to"}, \text{"Tangent"}[\text{i}, \text{j}]];
SMSIO[\{\text{\"DeformedMeshX\"} \rightarrow \text{\textit{u}}, \ldots\}, \text{"Export to"}, \text{"Nodal point post"}[\text{\textit{node}}]];
SMSIO[\{\text{\"Sxx\"} \rightarrow \text{\textit{\textit{\sigma}}, \ldots\}, \text{"Export to"}, \text{"Integration point post"}[\text{\textit{Ig}}]];
SMSIO[\{\text{\"DeformedMeshX\"} \rightarrow \{\text{\textit{u1}}, \text{\textit{u2}}, \ldots\}, \ldots\}, \text{"Export to"}, \text{"Nodal point post"}];
```
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMSIO[ &quot;Shape velocity field&quot;[is][node, coor] ]</td>
<td>returns value of first order shape velocity field for is-th sensitivity parameter in node-th element node for coor-th coordinate ($\frac{\partial X_{\text{node}}}{\partial \phi_{\text{is}}}$)</td>
</tr>
<tr>
<td>SMSIO[ &quot;Shape velocity field&quot;[is, js][node, coor] ]</td>
<td>returns value of second order shape velocity field for is, js-th sensitivity parameters in node-th element node for coor-th coordinate ($\frac{\partial^2 X_{\text{node}}}{\partial \phi_{\text{is}} \partial \phi_{\text{js}}}$)</td>
</tr>
<tr>
<td>SMSIO[ &quot;Shape velocity field&quot;[is][node]     ]</td>
<td>returns first or second order shape velocity field in node-th node for all coordinates</td>
</tr>
<tr>
<td>SMSIO[ &quot;Parameter velocity field&quot;[is][node, iparm] ]</td>
<td>returns the value of first order parameter velocity fields for is-th sensitivity parameter in node-th element node for iparm-th constant defined by SMSSensitivityNames ($\frac{\partial d_{\text{par}}}{\partial \phi_{\text{is}}}$)</td>
</tr>
<tr>
<td>SMSIO[ &quot;Parameter velocity field&quot;[is, js][node, iparm] ]</td>
<td>returns the value of second order parameter velocity field for is, js-th sensitivity parameters in node-th element node for iparm-th constant defined by SMSSensitivityNames ($\frac{\partial^2 d_{\text{par}}}{\partial \phi_{\text{is}} \partial \phi_{\text{js}}}$)</td>
</tr>
<tr>
<td>SMSIO[ &quot;Parameter velocity field&quot;[is][node]     ]</td>
<td>returns first or second order parameter velocity field in node-th node for all constant defined by SMSSensitivityNames</td>
</tr>
<tr>
<td>SMSIO[ &quot;Sensitivity DOFs&quot;[is][node, dof]       ]</td>
<td>returns first derivative of dof-th nodal DOF for is-th sensitivity parameter in node-th element node ($\frac{\partial p_{\text{e,dof}}}{\partial \phi_{\text{is}}}$)</td>
</tr>
<tr>
<td>SMSIO[ &quot;Sensitivity DOFs&quot;[is, js][node, dof]   ]</td>
<td>returns second derivative of dof-th nodal DOF for is, js-th sensitivity parameters in node-th element node ($\frac{\partial^2 p_{\text{e,dof}}}{\partial \phi_{\text{is}} \partial \phi_{\text{js}}}$)</td>
</tr>
<tr>
<td>SMSIO[ &quot;Sensitivity DOFs&quot;[is][node]            ]</td>
<td>returns first or second derivative of all nodal DOF in node-th element node</td>
</tr>
<tr>
<td>SMSIO[ &quot;Sensitivity DOFs&quot;[is, js][node]        ]</td>
<td>returns first or second derivative of all nodal DOF in all element nodes</td>
</tr>
<tr>
<td>SMSIO[ &quot;Sensitivity DOFs n&quot;[is][node, dof]     ]</td>
<td>returns first or second derivative of all nodal DOF in all element nodes</td>
</tr>
<tr>
<td>SMSIO[ &quot;Sensitivity DOFs n&quot;[is, js][node]      ]</td>
<td>returns first or second derivative of all nodal DOF in all element nodes</td>
</tr>
<tr>
<td>SMSIO[ &quot;Sensitivity DOFs n&quot;[is][node, coor]    ]</td>
<td>returns first or second derivative of all nodal DOF in all element nodes</td>
</tr>
<tr>
<td>SMSIO[ &quot;Sensitivity DOFs n&quot;[is, js][node]      ]</td>
<td>returns first or second derivative of all nodal DOF in all element nodes</td>
</tr>
<tr>
<td>SMSIO[ &quot;Parameter velocity field&quot;[is][node, iparm] ]</td>
<td>returns the value of first order parameter velocity fields for is-th sensitivity parameter in node-th element node for iparm-th constant defined by SMSSensitivityNames ($\frac{\partial d_{\text{par}}}{\partial \phi_{\text{is}}}$)</td>
</tr>
<tr>
<td>SMSIO[ &quot;Parameter velocity field&quot;[is, js][node, iparm] ]</td>
<td>returns the value of second order parameter velocity field for is, js-th sensitivity parameters in node-th element node for iparm-th constant defined by SMSSensitivityNames ($\frac{\partial^2 d_{\text{par}}}{\partial \phi_{\text{is}} \partial \phi_{\text{js}}}$)</td>
</tr>
<tr>
<td>SMSIO[ &quot;Sensitivity DOFs&quot;[is][node]            ]</td>
<td>returns first or second derivative of all nodal DOF in node-th element node</td>
</tr>
<tr>
<td>SMSIO[ &quot;Sensitivity DOFs&quot;[is, js][node]        ]</td>
<td>returns first or second derivative of all nodal DOF in all element nodes</td>
</tr>
<tr>
<td>SMSIO[ &quot;Sensitivity DOFs n&quot;[is][node, dof]     ]</td>
<td>returns first or second derivative of all nodal DOF in all element nodes</td>
</tr>
<tr>
<td>SMSIO[ &quot;Sensitivity DOFs n&quot;[is, js][node]      ]</td>
<td>returns first or second derivative of all nodal DOF in all element nodes</td>
</tr>
<tr>
<td>SMSIO[ &quot;Sensitivity DOFs n&quot;[is, js][node]      ]</td>
<td>returns first or second derivative of all nodal DOF in all element nodes</td>
</tr>
</tbody>
</table>
SMSIO["Adjoint vector DOFs"[ifunc][node, dof]]
returns the value of adjoint vector for $i_{th}$ functional in $node$-th node and $dof$-th nodal DOF (see Backward Mode Implementation Notes)

ed$$["BSETS", pos]
returns data stored for each element for the implementation of backward sensitivity at position $pos$ on the allocated "BSETS" data file (see Backward Mode Implementation Notes)

SMT$[$"->TasksStructure. IntegerInput", pos]
additional user input data accessible from backward mode element subroutines (see Backward sensitivity load, Backward sensitivity derivatives)

SMT$[$"->TasksStructure. IntegerInputLength "]

SMT$[$"->TasksStructure. RealInput", pos]
SMSIO["Real input vector length"[pos]]
SMSIO["Real input vector"[pos]]

SMT$[$"->TasksStructure. RealInputLength"]
SMSIO["Integer input vector length"[pos]]
SMSIO["Integer input vector"[pos]]

SMT$$[value, "Add to" or "Export to", "First order pseudo load"[is][dof]]
SMT$$[value, "Add to" or "Export to", "Second order pseudo load"[is,js][dof]]
SMT$$[value, "Add to" or "Export to", "Adjoint pseudo-load"[ifunc][dof]]
SMT$$[value, "Add to" or "Export to", "Backward sensitivity element storage"[pos][fog]]
adds or exports to first order sensitivity pseudo load value for $is$-th sensitivity parameter and $dof$-th nodal DOF
adds or exports to second order sensitivity pseudo load value for $is, js$-th sensitivity parameters and $dof$-th 1
adds or exports adjoint vector of $sof$-th nodal DOF for $ifunc$-th functional (see Backward sensitivity load, Backward sensitivity derivatives)
adds or exports value at position $pos$ on the allocated "BSETS" data file (see Backward Mode Implementation Notes)

SMT$$["->TasksStructure. IntegerInput", "BSAV", ifunc, dof]
nd$$[node, "BSAV", ifunc, dof]
SMSIO["Adjoint vector DOFs"[ifunc][node]]
SMSIO["Adjoint vector DOFs"[ifunc]]
returns the value of adjoint vector for $i_{th}$ functional in $node$-th node and $dof$-th nodal DOF (see Backward Mode Implementation Notes)

SMT$$[value, "Add to" or "Export to", "Second order pseudo load"[is,js][dof]]
SMT$$[value, "Add to" or "Export to", "Export to", "Second order pseudo load"[is,js]]
SMT$$[value, "Add to" or "Export to", "Export to", "Second order pseudo load"[is,js]]
adds or exports second order sensitivity pseudo load value for $is, js$-th sensitivity parameters and $dof$-th 1

SMT$$[value, "Add to" or "Export to", "Second order pseudo load"[is,js][dof]]
adds or exports second order sensitivity pseudo load value for $is, js$-th sensitivity parameters and $dof$-th 1

SMT$$[value, "Add to" or "Export to", "First order pseudo load"[is][dof]]
SMT$$[value, "Add to" or "Export to", "First order pseudo load"[is]]
SMT$$[value, "Add to" or "Export to", "First order pseudo load"[is]]
adds or exports to first order sensitivity pseudo load value for $is$-th sensitivity parameter and $dof$-th nodal DOF
adds or exports to first order sensitivity load values for $is$-th sensitivity parameter
adds or exports to first order sensitivity load values for $is$-th sensitivity parameter

SMT$$[value, "Add to" or "Export to", "Export to", "Second order pseudo load"[is,js]]
SMT$$[value, "Add to" or "Export to", "Export to", "Second order pseudo load"[is,js]]
adds or exports to second order sensitivity pseudo load value for $is, js$-th sensitivity parameters and $dof$-th 1
adds or exports to second order sensitivity pseudo load value for $is, js$-th sensitivity parameters and $dof$-th 1

SMT$$[value, "Add to" or "Export to", "Adjoint pseudo-load"[ifunc][dof]]
SMT$$[value, "Add to" or "Export to", "Adjoint pseudo-load"[ifunc]]
adds or exports adjoint vector of $sof$-th nodal DOF for $ifunc$-th functional (see Backward sensitivity load, Backward sensitivity derivatives)
adds or exports adjoint vector of $sof$-th nodal DOF for $ifunc$-th functional (see Backward sensitivity load, Backward sensitivity derivatives)

SMT$$[value, "Add to" or "Export to", "Backward sensitivity element storage"[pos][fog]]
SMT$$[value, "Add to" or "Export to", "Backward sensitivity element storage"[pos1,pos2,...]]
SMT$$[value, "Add to" or "Export to", "Backward sensitivity element storage"[pos]]
SMSIO["Backward sensitivity element storage"[pos1,pos2,...]]
SMSIO["Backward sensitivity element storage"[pos]]
SMSIO["Backward sensitivity element storage"[pos]]
eds$$["BSETS", pos]

For details see Forward Mode Implementation Notes, Backward Mode Implementation Notes.
### Tasks Input/Output data

<table>
<thead>
<tr>
<th>form</th>
<th>I/O parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMSIO[&quot;Task index&quot;]</td>
<td>Tasks$$</td>
<td>see User Defined Tasks</td>
</tr>
<tr>
<td>SMSIO[&quot;Task data&quot;[index]]</td>
<td>TasksData$$[index]</td>
<td>see User Defined Tasks</td>
</tr>
<tr>
<td>SMSIO[&quot;Task real input&quot;[index]]</td>
<td>RealInput$$[index]</td>
<td>see User Defined Tasks</td>
</tr>
<tr>
<td>SMSIO[&quot;Task integer input&quot;[index]]</td>
<td>IntegerInput$$[index]</td>
<td>see User Defined Tasks</td>
</tr>
<tr>
<td>SMSIO[value, &quot;Add to&quot; or &quot;Export to&quot;, &quot;Task data&quot;[index]]</td>
<td>TasksData$$[index]</td>
<td>see User Defined Tasks</td>
</tr>
<tr>
<td>SMSIO[value, &quot;Add to&quot; or &quot;Export to&quot;, &quot;Task real output&quot;[index]]</td>
<td>RealOutput$$[index]</td>
<td>see User Defined Tasks</td>
</tr>
<tr>
<td>SMSIO[value, &quot;Add to&quot; or &quot;Export to&quot;, &quot;Task integer output&quot;[index]]</td>
<td>IntegerOutput$$[index]</td>
<td>see User Defined Tasks</td>
</tr>
<tr>
<td>SMSIO[value, &quot;Add to&quot; or &quot;Export to&quot;, &quot;Task local vector&quot;[index]]</td>
<td>p$$[index]</td>
<td>see User Defined Tasks</td>
</tr>
<tr>
<td>SMSIO[value, &quot;Add to&quot; or &quot;Export to&quot;, &quot;Task local matrix&quot;[indexi,indexj]]</td>
<td>s$$[indexi,indexj]</td>
<td>see User Defined Tasks</td>
</tr>
</tbody>
</table>

SMSIO commands for I/O of data related to tasks.

#### Example: IO data defined by SMSTemplate constants

```plaintext
In[103]:= << "AceGen";
SMSInitialize["tmpTest", "Environment" -> "AceFEM"];
SMSTemplate["SMSTopology" -> "Q1", "SMSymmetricTangent" -> True,
  "SMSDomainDataNames" -> {"E - elastic modulus", "ν - poisson ratio", "t - thickness"},
  "SMSDefaultData" -> {21000, 0.3, 1}];
SMSStandardModule["Tangent and residual"];
SMSDo[Ig, 1, SMSIO["No. integration points"]];
SMSSaveAndEvaluate["Element definitions",
  Ξ = {ξ, η, ζ} ⊨ SMSIO["Integration point"[Ig]];
  {XIO, uIO} ⊨ SMSIO["All coordinates and DOFs"];
  Nh = 1/(1-ξ) (1-η), (1+ξ) (1-η), (1+ξ) (1+η), (1-ξ) (1+η));
  X ⊨ SMSFreeze[Append[Nh.XIO, ξ]];
  Je ⊨ SMSD[X, Ξ];
  Jed = Det[Je];
  u ⊨ Append[Nh.uIO, ξ];
  H ⊨ SMSD[u, X, "Dependency" -> {Ξ, X, SMSInverse[Je]}];
  JF = Det[F];
  Ct = Transpose[F].F;
  {Em, ν, tζ} ⊨ SMSIO["Domain data"];
  {λ, μ} ⊨ SMSHookeToLame[Em, ν];
  1
  W = λ (3F - 1)² + μ 1 - (Tr[Ct] - 3) - Log[JF];
  wgp ⊨ SMSIO["Integration weight"[Ig]];
];
pe = Flatten[ SMSIO["Nodal DOFs"] ];
SMSDo[
```

Rgi := Jed t CG wgp SMSD[W, pe, i];
SMSIO[Rgi, "Add to", "Residual"[i]];
SMSDo[
    Kgij = SMSD[Rgi, pe, j];
    SMSIO[Kgij, "Add to", "Tangent"[i, j]];
    {j, i, 8}];
    {i, 1, 8}];
SMSEndDo[];

SMSStandardModule["Postprocessing"];  
{u, v} = Transpose[ SMSIO["Nodal DOFs"] ];
SMSIO[{"DeformedMeshX" → u, "DeformedMeshY" → v, "u" → u, "v" → v},  
    "Export to", "Nodal point post"];  
SMSDo[Ig, 1, SMSIO["No. integration points"];  
    SMSEvaluateSaved["Element definitions"];  
    Eg = 1/2 (Ct - IdentityMatrix[3]);
    σ = (1/JF) * SMSD[W, F, "Ignore" → NumberQ].Transpose[F];
    SMSIO[  
        "Sxx" → σ[1, 1], "Sxy" → σ[1, 2], "Sxz" → σ[1, 3], "Syx" → σ[2, 1], "Syy" → σ[2, 2],  
        "Syz" → σ[2, 3], "Szz" → σ[3, 3], 
        "Exx" → Eg[1, 1], "Exy" → Eg[1, 2], "Exz" → Eg[1, 3], "Eyx" → Eg[2, 1], "Eyy" → Eg[2, 2],  
        "Eyz" → Eg[2, 3], "Ezx" → Eg[3, 1], "Ezy" → Eg[3, 2], "Ezz" → Eg[3, 3]]  
    , "Export to", "Integration point post"[Ig]];
SMSEndDo[];

SMSWrite[];

\begin{verbatim}
File: tmpTest.c Size: 11475 Time: 2
Method | SKR | SPP
No.Formulae | 94 | 76
No.Leafs | 1395 | 1286
\end{verbatim}
SMTInputData[];
SMTAddDomain["d", "tmpTest", {}];
SMTAddMesh[Polygon[{{0, 0}, (48, 44), (48, 44 + 16), (0, 44)}], "d", "Q1", (3, 3)];
SMTAddEssentialBoundary[Line[{{0, 0}, (0, 44)}], 1 -> 0, 2 -> 0];
SMTAddNaturalBoundary[Line[{{48, 44}, (48, 44 + 16)}], 2 -> Line[{-1000}]];
SMTAnalysis[];
nstep = 10; \[\Delta \lambda\] = 0.1; tolNR = 10^-8; maxNR = 15;
Do[
    SMTNextStep["\[\Delta \lambda\]" \rightarrow \[\Delta \lambda\]];  
    While[SMTConvergence[tolNR, maxNR], SMTNewtonIteration[]; 
    , {i, 1, nstep}
    ]
SMTShowMesh["DeformedMesh" \rightarrow True, "BoundaryConditions" \rightarrow True]

Dynamically created IO data variables

A simple IO interface presented in General Input / Output data uses template constants such as SMSTopology to create proper IO interface.

In the case that template constants are not known in advance there is an alternative mechanism available that allows dynamic allocation of data and creates template constants while creating the interface. The dynamic SMSIO associates values with the keys in order to simplify the data management. The primal purpose of the mechanism for the automatic IO Data Management is dynamic allocation of data of given type and association of the data with the keys. The SMSIO command remembers the number of elements of specific type allocated. When called the second time with the same key the already defined value is returned. The key represents the specific value and can be used later on (through all the AceGen session!) to retrieve the value that was originally assigned to the key.
form | description
--- | ---
SMSIO[**dataType**, **key->addData**] | creates a new auxiliary variable of the type associated with **dataType** and associates the value and the corresponding additional **addData** to the **key**. If the **key** already exists it returns the auxiliary variable associated with the **key**. The **dataTypes** and the necessary additional data (**addData**) is listed in a table below.

SMSIO[**dataType**, \{**key1->addData1**, **key2->addData2**, ...\}] | creates a set of new auxiliary variables of the type associated with **dataType** and associates the values and the corresponding additional data **addData**, to the given keys **key**.

SMSIO[**value**, "Add to", **dataType**, **key**] | adds the value (scalar or vector) to the output parameters of the subroutine defined by **dataType** and **key**.

SMSIO[**value**, "Export to", **dataType**, **key**] | exports the value (scalar or vector) to the output parameters of the subroutine defined by **dataType** and **key**.

SMSIO[values_List, "Add to", **dataType**, keys_List] | adds the values to the output parameters of the subroutine defined by **dataType** and **keys**.

SMSIO[values_List, "Export to", **dataType**, keys_List] | exports the values to the output parameters of the subroutine defined by **dataType** and **keys**.

SMSIO[**dataType**] | Updates all the data associated with the given **dataType**. New auxiliary variables are created for all the **keys** associated with the data **dataType** using the same **addData** when given. Command returns newly created auxiliary variables. New auxiliary variables are created only for the keys that do not yet have associated auxiliary variables within the current subroutine. With the **SMSUndefineIO[]** command all already defined auxiliary variables are cleared before updating.

---

<table>
<thead>
<tr>
<th>form</th>
<th>data type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMSIO[&quot;Nodal coordinates&quot;, <strong>key</strong>]</td>
<td>nd$$[node, &quot;X&quot;, coordinate]</td>
<td>adds/returns coordinates of the nodes associated with the given key (see Node Data)</td>
</tr>
<tr>
<td>SMSIO[&quot;Nodal DOFs&quot;, <strong>key</strong>]</td>
<td>nd$$[node, &quot;at&quot;, dof]</td>
<td>adds/returns a set of nodal DOFs at time n associated with the given key (see Node Data)</td>
</tr>
<tr>
<td>SMSIO[&quot;Nodal DOFs n&quot;, <strong>key</strong>]</td>
<td>nd$$[node, &quot;ap&quot;, dof]</td>
<td>adds/returns nodal DOFs at time n (<strong>key</strong> must be the same as for the time n+1) (see Node Data)</td>
</tr>
<tr>
<td>SMSIO[&quot;Element time dependent data&quot;, <strong>key</strong>]</td>
<td>ed$$[&quot;ht&quot;,/i]</td>
<td>adds/returns time dependent variable at time n+1 associated with the given key (see Element Data)</td>
</tr>
<tr>
<td>SMSIO[&quot;Element time dependent data n&quot;, <strong>key</strong>]</td>
<td>ed$$[&quot;hp&quot;,/i]</td>
<td>adds/returns time dependent variable at time n associated with the given key (<strong>key</strong> must be the same as for the time n+1) (see Element Data)</td>
</tr>
<tr>
<td>SMSIO[&quot;Domain data&quot;, <strong>key</strong>]</td>
<td>es$$[&quot;Data&quot;,/i]</td>
<td>data common for all the elements within a particular domain associated with the given key</td>
</tr>
</tbody>
</table>

**Generics SMSIO commands for importing and exporting arbitrary data.**

<table>
<thead>
<tr>
<th>form</th>
<th>data type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Returns dynamically created IO variable associated with the given key/keys.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>key</strong> can also be a patterns, e.g. SMSIO[&quot;Nodal coordinates&quot;, &quot;u&quot;[_]] would return a set of nodal coordinates of all nodes with associated key that matches pattern &quot;u&quot;[_.].</td>
</tr>
</tbody>
</table>

**dataType | addData | description**
### "Nodal coordinates"  
\[ \text{nd}[, \text{"X"}, \_] \]

Pattern defines a new node

\( \text{(see Node Data, Node Identification)} \) \( \text{nd}[,\text{"X"},\_] \) is \( j \)-th component of the spatial coordinate of \( i \)-th element node.

The value of the \text{nodeType} can be:

a) "Topological" in the case of topological node,

b) an integer number in the case of additional topological node. Additional node is placed at the same position as the \text{nodeType}th topological node.

c) pure function in the case of an additional topological node. Additional node is placed at position defined by the function (\( e.g. \text{Function}(\#1+\#2)/2 \) will place additional node in the middle between the first and the second topological node.

d) "Auxiliary" or "Dummy" in case of additional auxiliary or dummy node (see Node Identification)

### "Nodal DOFs"  
\[ \text{nd}[,\text{"at"},\_] \]

Pattern defines DOF-\( s \) of the node with node identification \text{nodeID} and with \text{nDOF} unknowns (see Node Data) at time \( n+1 \).

\( \text{nd}[,\text{"at"},\_] \) is \( j \)-th component of the vector of DOF-\( s \) of \( i \)-th element node at time \( n+1 \).

### "Nodal DOFs n"  
\[ \text{nd}[,\text{"ap"},\_] \]

Pattern defines DOFs associated with the node at time \( n \) (see Node Data)

\( \text{nd}[,\text{"ap"},\_] \) is \( j \)-th component of the vector of DOF-\( s \) of \( i \)-th element node at time \( n \). \text{addData} is taken from the corresponding "Nodal DOFs".

### "Element time dependent data"  
\[ \text{ed}[,\text{"ht"},\_] \]

Pattern defines a current value of history dependent real type variable automatically allocated on element data field "ht" (see Element Data). The SMSIO returns auxiliary variable that points to corresponding element on "ht".

### "Element time dependent data"  
\[ \text{ed}[,\text{"ht"},\_] \{\text{length},\text{index1}\} \]

Pattern defines a current value of history dependent real type vector of length \text{length} automatically allocated on element data field "ht" (see Element Data). The SMSIO returns auxiliary variable that points to the \text{index}th element of the allocated vector. If \text{index1}=Null then only vector is allocated.

### "Element time dependent data index2"  
\[ \text{ed}[,\text{"ht"},\_] \]

Real type vector allocated on element data field "ht" can have an arbitrary number of indices assigned. Only the first time the length has to be specified as well.

### "Element time dependent data index"  
\[ \text{ed}[,\text{"hp"},\_] \]

Pattern defines a value of history dependent real type variable at the end of previous time step allocated on element data field "hp" (see Element Data). Corresponding "Element time dependent data" data have to be defined before "Element time dependent data index" data! The SMSIO returns auxiliary variable that points to the \text{index}th element of the allocated vector.
### Domain data

Pattern defines domain data or material data (see Domain Specification Data) defined by identification `domainDataName` and with the default value `defaultData`. The data is associated to the specific type of elements (if `isSensitivityParameter` is True then the parameters).

### Char type switch

Pattern defines character type switch associated to the specific type of elements (see Domain Specification Data, `es$"CharSwitch",i` is `i`th character type constant). The `switchValue` is written into domain data structure and `switchDescription` is used to make documentation.

### Integer type switch

Pattern defines integer type switch associated to the specific type of elements (see Domain Specification Data, `es$"IntSwitch",i` is `i`th integer type constant). The `switchValue` is written into domain data structure and `switchDescription` is used to make documentation.

### Real type switch

Pattern defines real type switch associated to the specific type of elements (see Domain Specification Data, `es$"DoubleSwitch",i` is `i`th integer type constant). The `switchValue` is written into domain data structure and `switchDescription` is used to make documentation.

### Element data

Pattern defines real type variable(s) automatically allocated on element data field "Data" (see Element Data). Depending on `addData` command does:

- `Null` \( \Rightarrow \) allocates and returns scalar variable
- `index1` \( \Rightarrow \) allocates vector of length `length` and returns scalar variable that points to `index1`-th element of allocated vector
- `index2` \( \Rightarrow \) returns scalar variable that points to `index2`-th element of previously allocated vector

Data structures with an automatic data management support.
Examples:

```plaintext
SMSIO["Nodal coordinates", Table["X"[i], {i, SMSNoNodes}]];
SMSIO["Nodal DOFs", Table["u"[i] \rightarrow \{SMSNoDimensions, "D"\}, {i, SMSNoNodes}]];
SMSIO["Domain data", {"Em" \rightarrow \{"E - elastic modulus", 21000\}}];
SMSIO["Element time dependent data", Table["heIO"[i], {i, nhe}]];
SMSIO["Element time dependent data", Table["hgIO"[i] \rightarrow \{Hold[NoIp], Ig\}, {i, nhg}]];
(*allocate vector on ed$["Data",_]*)
SMSIO["Element data", "vec" \rightarrow \{length, Null\}];
SMSIO["Char type switch", "V" \rightarrow \{"Volume", "Calculate element volume"\}];
```
Example: dynamically created IO data variables

```plaintext
<< "AceGen";
SMSInitialize["tmpTest", "Environment" -> "AceFEM"];
SMSTemplate["SMSTopology" -> "Q1", "SMSSymmetricTangent" -> True];

SMSStandardModule["Tangent and residual"]; SMSDo[1, SMSIO["No. integration points"]; SMSSaveAndEvaluate["Element definitions", B = {ξ, η, ξ} = SMSIO["Integration point"[Ig]]; XIO = SMSIO["Nodal coordinates", Table["X"[i], {i, SMSNoNodes}]]; uIO = SMSIO["Nodal DOFs", Table["u"[i] -> {SMSNoDimensions, "D"}, {i, SMSNoNodes}]]; Nh = 1/4 ((1 - ξ) (1 - η), (1 + ξ) (1 - η), (1 + ξ) (1 + η), (1 - ξ) (1 + η)); X = SMSFreeze[Append[Nh.XIO, ξ]]; Je = SMSDO[X, ξ]; Jed = Det[Je]; u = Append[Nh.uIO, 0]; H = SMSD[u, X, "Dependency" -> {ξ, X, SMSInverse[Je]}]; SMSFreeze[F, IdentityMatrix[3] + H, "Ignore" -> PossibleZeroQ]; JF = Det[F]; Ct = Transpose[F].F; {Em, ν, tξ} = SMSIO["Domain data", {"Em" -> {-"E - elastic modulus", 21000}, "ν" -> {"ν - poisson ratio", 0.3}, "tξ" -> {"t - thickness", 1}}]; {λ, μ} = SMSHookeToLame[Em, ν]; W = 1/2 λ (JF - 1) ^2 + μ (1/2 (Tr[Ct] - 3) - Log[JF]); ]; wgp = SMSIO["Integration weight"[Ig]]; pe = Flatten[SMSIO["Nodal DOFs"]]; SMSDo[ Rgi = Jed tξ wgp SMSD[W, pe, 1]; SMSIO[Rgi, "Add to", "Residual"[i]]; SMSDo[ Kgij = SMSD[Rgi, pe, j]; SMSIO[Kgij, "Add to", "Tangent"[i, j]]; , {i, 1, SMSNoNodes}; , {j, 1, 8}]; SMSEndDo[];

SMSStandardModule["Postprocessing"]; (u, v) = Transpose[SMSIO["Nodal DOFs", "u"[...]]]; SMSIO["DeformedMeshX" -> u, "DeformedMeshY" -> v, "u" -> u, "v" -> v], "Export to", "Nodal point post"]; SMSDo[1, SMSIO["No. integration points"]; SMSEvaluateSaved["Element definitions"]; Eg = 1/2 (Ct - IdentityMatrix[3]); σ = (1/2F) * SMSD[W, F, "Ignore" -> NumberQ] . Transpose[F]; SMSIO[ "Sxx" -> σ[1, 1], "Sxy" -> σ[1, 2], "Sxz" -> σ[1, 3], "Syy" -> σ[2, 1], "Syz" -> σ[2, 2], "Szx" -> σ[2, 3], "Szz" -> σ[3, 1], "Szy" -> σ[3, 2], "Sz" -> σ[3, 3], "Exx" -> Eg[1, 1], "Exy" -> Eg[1, 2], "Exz" -> Eg[1, 3], "Eyy" -> Eg[2, 1], "Eyz" -> Eg[2, 2], "Ezx" -> Eg[2, 3], "Ezy" -> Eg[3, 1], "Ezy" -> Eg[3, 2], "Ezz" -> Eg[3, 3], "Exy" -> Eg[2, 3], "Ezx" -> Eg[3, 1], "Ezy" -> Eg[3, 2], "Ezz" -> Eg[3, 3]); , "Export to", "Integration point post"[Ig]]; SMSEndDo[];

SMSWrite[];
```
Example: dynamically created multi-field element IO

Create an element with the following nodes and nodal unknowns:

- 4 topological nodes with 2 unknowns for discretization of displacements
- 4 additional nodes with 1 unknown for discretization of temperature field
- 1 additional node with 1 unknown for local Lagrange type constrain equation

```mathematica

\text{SMTInputData[];}
\text{SMTAddDomain["d", "tmpTest", {}];}
\text{SMTAddMesh[Polygon[{{{0, 0}, (48, 44), (48, 44 + 16), {0, 44}}}, "d", "Q1", {3, 3}];}
\text{SMTAddEssentialBoundary[Line[{{{0, 0}, {0, 44}}}, 1 \rightarrow 0, 2 \rightarrow 0];}
\text{SMTAddNaturalBoundary[Line[{{{48, 44}, (48, 44 + 16)}}, 2 \rightarrow Line[{-1000}]};
\text{SMTAnalysis[];}
\text{nstep = 10; } \Delta \lambda = 0.1; \text{tolNR} = 10^{-8}; \text{maxNR} = 15;
\text{Do[}
\text{SMTNextStep["\Delta \lambda" } \rightarrow \Delta \lambda];
\text{While[SMTConvergence[\text{tolNR, maxNR}], SMTNewtonIteration[]];}
\text{, (i, 1, nstep)\text{]}}
\text{SMTShowMesh["DeformedMesh" } \rightarrow \text{True, "BoundaryConditions" } \rightarrow \text{True}]
In[89]:= X = SMSIO["Nodal coordinates", Table["X"[i], {i, SMSNoNodes}]]

Out[89]= {{x[1], x[2]}, {x[3], x[4]}, {x[5], x[6]}, {x[7], x[8]}}

In[90]:= p = SMSIO["Nodal DOFs", Table["u"[i] -> (SMSNoDimensions, "D"), {i, SMSNoNodes}]]

Out[90]= {{[x[1], x[2]], [x[3], x[4]], [x[5], x[6]], [x[7], x[8]]}}

In[91]:= (SMSNoNodes, SMSNodeID, SMSDOFGlobal, SMSAdditionalNodes)

Out[91]= {4, {D, D, D, D}, {2, 2, 2, 2}, {}} &

In[92]:= SMSIO["Nodal coordinates"]

Out[92]= {{[x[1], x[2]], [x[3], x[4]], [x[5], x[6]], [x[7], x[8]]}}

In[93]:= SMSIO["Nodal coordinates", "X"[3]]

Out[93]= [[x[3], x[4]]]

In[94]:= SMSIOInfo["Nodal coordinates", "X"[3], "Export"]

Out[94]= {nd$$[3, X, 1], nd$$[3, X, 2]}

In[95]:= SMSIOInfo["Nodal coordinates", "Key"]

Out[95]= {{X[1], X[2], X[3], X[4]}}

- definition of additional nodes and nodal unknowns for discretization of temperature field

In[96]:= XIo = SMSIO["Nodal coordinates", Table["XT"[i] -> i, {i, 4}]]

Out[96]= {{[x[1], x[2]], [x[3], x[4]], [x[5], x[6]], [x[7], x[8]]}}

Out[96]= {{[x[1], x[2]], [x[3], x[4]], [x[5], x[6]], [x[7], x[8]]}}

Out[96]= {8, {D, D, D, D, T -> M, T -> M, T -> M, T -> M}, {2, 2, 2, 2, 1, 1, 1, 1}, {1, 2, 3, 4} &}

- definition of additional node and nodal unknown for local Lagrange type constrain equation

In[97]:= SMSIO["Nodal coordinates", "LC" -> "Auxiliary"]

Out[97]= {{x[1], x[2]}}

In[98]:= LIO = SMSIO["Nodal DOFs", "L" -> {1, "LC - LP"}]

Out[98]= {{}}

In[99]:= (SMSNoNodes, SMSNodeID, SMSDOFGlobal, SMSAdditionalNodes)

Out[99]= {9, {D, D, D, D, T -> M, T -> M, T -> M, T -> M, LC -> LP}, {2, 2, 2, 2, 1, 1, 1, 1, 1}, {1, 2, 3, 4, Null} &}

- Definition of the second user subroutine.

In[100]:= SMSStandardModule["Postprocessing"];

- IO data structures are automatically updated based on previous definitions

In[101]:= SMSIO["Nodal coordinates", "X"[3]]

Out[101]= {{x[3], x[4]}}
Example: low-level IO data definitions

```
In[88]:= \nIn[100]:= \nOut[100]= \nOut[101]=

Out[101]= xio = SMSIO["Nodal coordinates"]
Out[102]= {{x1, x1}, {x2, x2}, {x1, x2}, {x1, x2},
{x1, x2}, {x1, x2}, {x1, x2}, {x1, x2}}

Out[103]= peio = SMSIO["Nodal DOFs"]
Out[104]= {{u1, u1}, {u1, u2}, {u2, u2}, {u1, u2}, {u2, u2}, {u1, u2}, {u2, u2}, {u1, u2}}
```

```
File: tmpTest.c  Size: 7494  Time: 2

<table>
<thead>
<tr>
<th>Method</th>
<th>No. Formulae</th>
<th>No. Leafs</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKR</td>
<td>93</td>
<td>1389</td>
</tr>
</tbody>
</table>
```
In[111]:= SMTInputData[];
SMTAddDomain["d", "tmpTest", {}];
SMTAddMesh[Polygon[{{0, 0}, {48, 44}, {48, 44 + 16}, {0, 44}}], "d", "Q1", {3, 3}];
SMTAddEssentialBoundary[Line[{{0, 0}, {0, 44}}], 1 \[RightArrow] 0, 2 \[RightArrow] 0];
SMTAddNaturalBoundary[Line[{{48, 44}, {48, 44 + 16}}], 2 \[RightArrow] Line[{-1000}]]; SMTAnalysis[];
nstep = 10; \[Delta]\[Lambda] = 0.1; tolNR = 10^-8; maxNR = 15;
Do[
SMTNextStep["\[Delta]\[Lambda]" \[RightArrow] \[Delta]\[Lambda]];
While[SMTConvergence[tolNR, maxNR], SMTNewtonIteration[]; , {i, 1, nstep}]
SMTShowMesh["DeformedMesh" \[RightArrow] True, "BoundaryConditions" \[RightArrow] True]

Utility IO data management commands

SMSUndefineIO[]
clear all definitions of IO data, thus they can be defined with SIO[dataType] again (the data is cleared automatically at the beginning of each subroutine)

SMSIOInfo[dataType, taskCode]
SMSIOInfo[dataType, keyPattern, taskCode]
returns data accordingly to the given taskCode

taskCode description
"Exist" function returns True if data with specified dataType and keyPattern exists and False if not
"Start" function returns a pointer to the first element of the vector allocated on ed$["ht",_] or ed$["hp",_] with all auxiliary variables replaced by their definition
"Length" function returns length of the vector allocated on ed$["ht",_] or ed$["hp",_]  
"Export" function returns the patterns (e.g. ed$[5]) suitable as parameter for SMSExport function
"Key" function returns all keys associated with specified dataType and keyPattern
"Data" function returns all addData associated with specified dataCode and keyPattern
taskCodes for the SMSIOInfo function.

**Integer Type Environment Data**

- General data
- Mesh input related data
- Iterative procedure related data
- Debugging and errors related data
- Linear solver related data
- Sensitivity related data
- Contact related data

The data is in AceGen input created with SMSIO(idataCode) (see General Input / Output data). See also SMTIData.

**General data**

<table>
<thead>
<tr>
<th>Default form</th>
<th>Default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idata$$[&quot;IDataLength&quot;]</td>
<td>200/R</td>
<td>actual length of idata vector</td>
</tr>
<tr>
<td>idata$$[&quot;RDataLength&quot;]</td>
<td>200/R</td>
<td>actual length of rdata vector</td>
</tr>
<tr>
<td>idata$$[&quot;IDataLast&quot;]</td>
<td>?/R</td>
<td>index of the last value reserved on idata vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(we can store additional user defined data after this point)</td>
</tr>
<tr>
<td>idata$$[&quot;RDataLast&quot;]</td>
<td>?/R</td>
<td>index of the last value reserved on rdata vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(we can store additional user defined data after this point)</td>
</tr>
<tr>
<td>idata$$[&quot;LastIntCode&quot;]</td>
<td>?/R</td>
<td>last integration code for which numerical integration points and weights were calculated</td>
</tr>
<tr>
<td>idata$$[&quot;OutputFile&quot;]</td>
<td>?/R</td>
<td>output file number or output channel number</td>
</tr>
<tr>
<td>idata$$[&quot;SymmetricTangent&quot;]</td>
<td>?/R</td>
<td>1 = global tangent matrix is symmetric</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = global tangent matrix is unsymmetrical</td>
</tr>
<tr>
<td>idata$$[&quot;MinNoTmpData&quot;]</td>
<td>3</td>
<td>minimum number of real type variables per node stored temporarily</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(actual number of additional temporary variables per node is calculated as Max[&quot;MinNoTmpData&quot;, number of nodal d.o.f])</td>
</tr>
<tr>
<td>idata$$[&quot;Task&quot;]</td>
<td>?/R</td>
<td>code of the current task performed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-1 - Quit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - undefined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000+n - User n</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - standard NR iteration</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - residual</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 - arc length iteration</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 - arc length estimate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 - postprocessing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 - global part of first order forward mode sensitivity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 - local part of first order forward mode sensitivity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 - global part of second order forward mode sensitivity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9 - local part of second order forward mode sensitivity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 - first order backward mode sensitivity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11 - second order backward mode sensitivity</td>
</tr>
<tr>
<td>idata$$[&quot;CurrentElement&quot;]</td>
<td>0/R</td>
<td>index of the current element processed</td>
</tr>
<tr>
<td>idata$$[&quot;TmpContents&quot;]</td>
<td>0</td>
<td>the meaning of the temporary real type variables stored during the execution of a single analysis into nd$$[i,&quot;tmp&quot;, j] data structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = not used</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = residual (reactions)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = used for post-processing</td>
</tr>
</tbody>
</table>
0 = residual vector is not formed separately
1 = during the execution of the SMTNewtonIteration
command the residual vector is formed separately and stored into
ndSS[i,"tmp", j] (at the end the ndSS[i,"tmp", j] contains the j-
th component of the nodal reaction in the i-th node) (see SMTResidual)

1 → prevent closing of the CDriver console on exit

memory used to store data (bytes)

Used for buckling analysis (K_0 + \lambda K_\sigma) \{\Psi\} = \{0\}:
0 = form full nonlinear matrix
1 = form K_0
2 = form K_\sigma
3 = form K_0 + K_\sigma

**ONLY VALID FOR ELEMENTS THAT SUPPORT THE OPTION.**

type of extrapolation of integration point values to nodes
0 = least square extrapolation ( )
1 = integration point value is multiplied by the user
defined weight factors (see Standard User Subroutines )

number of processors that are available for the parallel execution

The length of system vector where AceGen
generated variables are stored in CDriver. In some cases
(e.g. for backward differentiation of complex program structures)
significant additional storage might be required.

True of CDriver opens console window
and False when printing is only possible to notebook

### Mesh input related data

<table>
<thead>
<tr>
<th>Default form</th>
<th>Default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idata$$[&quot;NoAllNodes&quot;]</td>
<td>?/R</td>
<td>total number of nodes</td>
</tr>
<tr>
<td>idata$$[&quot;NoNodes&quot;]</td>
<td>?/R</td>
<td>number of mesh nodes</td>
</tr>
<tr>
<td>idata$$[&quot;NoSystemNodes&quot;]</td>
<td>?/R</td>
<td>number of system nodes (e.g. dummy nodes, global Lagrange nodes, etc..)</td>
</tr>
<tr>
<td>idata$$[&quot;NoElements&quot;]</td>
<td>?/R</td>
<td>total number of elements</td>
</tr>
<tr>
<td>idata$$[&quot;NoESpec&quot;]</td>
<td>?/R</td>
<td>total number of domains</td>
</tr>
<tr>
<td>idata$$[&quot;NoDimensions&quot;]</td>
<td>?/R</td>
<td>number of spatial dimensions of the problem (2 or 3)</td>
</tr>
<tr>
<td>idata$$[&quot;NoNSpec&quot;]</td>
<td>?/R</td>
<td>total number of node specifications</td>
</tr>
<tr>
<td>idata$$[&quot;NoEquations&quot;]</td>
<td>?/R</td>
<td>total number of global equations</td>
</tr>
<tr>
<td>idata$$[&quot;DummyNodes&quot;]</td>
<td>0</td>
<td>1 = dummy nodes are supported for the current analysis</td>
</tr>
<tr>
<td>idata$$[&quot;NoMultipliers&quot;]</td>
<td>1</td>
<td>number of boundary conditions multipliers</td>
</tr>
<tr>
<td>idata$$[&quot;MaxEquations&quot;]</td>
<td>?/R</td>
<td>maximum number of global equations (without constrains)</td>
</tr>
</tbody>
</table>

### Iterative procedure related data

See: Iterative Solution Procedures, SMTConvergence, SMTStatusReport, SMTErrorCheck
<table>
<thead>
<tr>
<th>Default form</th>
<th>Default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idata$$[&quot;Iteration&quot;]</td>
<td>?/R</td>
<td>index of the current iteration within the iterative loop</td>
</tr>
<tr>
<td>idata$$[&quot;TotalIteration&quot;]</td>
<td>?/R</td>
<td>total number of iterations in session</td>
</tr>
<tr>
<td>idata$$[&quot;Step&quot;]</td>
<td>0</td>
<td>total number of completed solution steps (set by Newton-Raphson iterative procedure)</td>
</tr>
<tr>
<td>idata$$[&quot;ErrorStatus&quot;]</td>
<td>0/RW</td>
<td>code for the type of the most important error event (see SMTErrorCheck)</td>
</tr>
<tr>
<td>idata$$[&quot;LinearEstimate&quot;]</td>
<td>0/RW</td>
<td>the value of the &quot;EBCUpdate&quot; option of SMTNewtonIteration command</td>
</tr>
<tr>
<td>idata$$[&quot;ResidualAndTangentTask&quot;]</td>
<td>?</td>
<td>SMTNewtonIteration is performing: 0 = corrector iteration (standard Newton-Raphson or related iteration) 1 = predictor iteration (usually the first iteration) 2 = recalculate element local equations and update history vectors (ht, for locally coupled problems) 3 = evaluate $R_e$ and $K_e$ for current element state (no other actions should be taken) The value is set by various procedures and should be properly interpreted by all elements.</td>
</tr>
<tr>
<td>idata$$[&quot;PostIteration&quot;]</td>
<td>0</td>
<td>is set by the SMTConvergence command to 1 if idata$$[&quot;PostIterationCall&quot;] = 1 or the SMTConvergence has been called with switch &quot;PostIteration&quot; -&gt;True</td>
</tr>
<tr>
<td>idata$$[&quot;PostIterationCall&quot;]</td>
<td>0</td>
<td>1 = additional call of the SKR user subroutines after the convergence of the global solution is enabled in at least one of the elements (&quot;SMTPostIterationCall&quot;-&gt;True)</td>
</tr>
<tr>
<td>idata$$[&quot;SkipTangent&quot;]</td>
<td>0</td>
<td>1 = the global tangent matrix is not assembled</td>
</tr>
<tr>
<td>idata$$[&quot;SkipResidual&quot;]</td>
<td>0</td>
<td>1 = the global residual vector is not assembled</td>
</tr>
<tr>
<td>idata$$[&quot;SkipSolver&quot;]</td>
<td>0</td>
<td>0 = full Newton-Raphson iteration 1 = the tangent matrix and the residual vector are assembled but the resulting system of equations is not solved</td>
</tr>
<tr>
<td>idata$$[&quot;NoSubIterations&quot;]</td>
<td>0/R</td>
<td>maximal number of local sub-iterative process iterations performed during the analysis</td>
</tr>
<tr>
<td>idata$$[&quot;SubIterationMode&quot;]</td>
<td>0</td>
<td>Switch used in the case that alternating solution has been detected by the SMTConvergence function. $0 = i+1, b_0 = b^p$ $1 = i, b_0 = b^i$</td>
</tr>
<tr>
<td>idata$$[&quot;GlobalIterationMode&quot;]</td>
<td>0</td>
<td>Switch used in the case that alternating solution has been detected by the SMTConvergence function. $0 = no restrictions on global equations$ $1 = freeze all &quot;If&quot; statements (e.g. nodes in contact, plastic-elastic regime)</td>
</tr>
<tr>
<td>idata$$[&quot;MaxPhysicalState&quot;]</td>
<td>0/RW</td>
<td>used for the indication of the physical state of the element (e.g. 0-elastic, 1-plastic, etc., user controlled option)</td>
</tr>
<tr>
<td>idata$$[&quot;LineSearchUpdate&quot;]</td>
<td>False</td>
<td>activate line search procedure (see also rdata$$[&quot;LineSearchStepLength&quot;])</td>
</tr>
<tr>
<td>idata$$[&quot;NoBackStep&quot;]</td>
<td>0</td>
<td>number of failed iterative solution steps</td>
</tr>
<tr>
<td>idata$$[&quot;DOFScaling&quot;]</td>
<td>0</td>
<td>1 = scaling of DOF is performed $0 = scaling of DOF is not performed</td>
</tr>
<tr>
<td>idata$$[&quot;NoDiscreteEvents&quot;]</td>
<td>0</td>
<td>number of discrete events recorded during the NR-iteration by the elements (e.g. new contact node, transformation from elastic to plastic regime)</td>
</tr>
</tbody>
</table>

### Debugging and errors related data

See: Iterative Solution Procedures, SMTConvergence, SMTStatusReport, SMTErrorCheck
Printing is suspended if the actual number of printouts exceeds the maximum number allowed as defined by SMTIData["MaxMessages"] environment variable. Warning message is produced when the number of printouts reaches SMTIData["MaxMessages"], see SMSPrintMessage.

idatad["ErrorStatus"] 0/RW code for the type of the most important error event (see SMTErrortCheck)

idatad["SubDivergence"] 0/RW number of the "Divergence of the local sub-iterative process" error events detected form the last error check

idatad["ErrorElement"] 0 last element where error event occurred

idatad["MaterialState"] 0/RW number of the "Non-physical material point state" error events detected form the last error check

idatad["ElementShape"] 0/RW number of the "Non-physical element shape" error events detected form the last error check

idatad["MissingSubroutine"] 0/RW number of the "Missing user defined subroutine" error events detected form the last error check

idatad["ElementState"] 0/RW number of the "Non-physical element state" error events detected form the last error check

idatad["DebugElement"] 0 -1 ⇒ break points (see Debugging of AceGen–AceFEM Codes) and control print outs (see SMSPrint) are active for all elements

0 ⇒ break points and control print outs are disabled

>0 ⇒ break points and control print outs are active only for the element with the index SMTIData["DebugElement"]

### Linear solver related data

- idatad["SkipSolver"] 0 0 ⇒ full Newton-Raphson iteration

1 ⇒ the tangent matrix and the residual vector are assembled but the resulting sistem of equations is not solved

- idatad["SetSolver"] 0 1 ⇒ recalculate solver dependent data structures if needed

- idatad["SolverMemory"] 0 memory used by solver (bytes)

- idatad["TangentMatrixMemory"] 0 memory used to store global tangent matrix (bytes)

- idatad["Solver"] 0 solver identification number

- idatad["Solver1"], idatad["Solver2"], idatad["Solver3"], idatad["Solver4"], idatad["Solver5"] solver specific parameters

- idatad["ZeroPivots"] 0 number of near-zero pivots (see also SMTSetSolver)

- idatad["NegativePivots"] 0 number of negative pivots (or -1 if data is not available), (see also SMTSetSolver)

- idatad["NoLinearConstraints"] 0 number of linear constraint equations
## Sensitivity related data

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idata$$[&quot;SensitivityScheme&quot;]</td>
<td>0</td>
<td>general sensitivity analysis scheme</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - no sensitivity analysis</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - first order forward mode sensitivity analysis</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - first order forward mode sensitivity, second order forward mode sensitivity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 - first order backward mode sensitivity analysis</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 - first order forward mode sensitivity, second order backward mode sensitivity</td>
</tr>
<tr>
<td>idata$$[&quot;NoSensParameters&quot;]</td>
<td>F/B</td>
<td>number of sensitivity parameters of the problem (Φ)</td>
</tr>
<tr>
<td>idata$$[&quot;NoSensDerivatives&quot;]</td>
<td>F</td>
<td>NoFirstOrderDerivatives+NoSecondOrderDerivatives+NoThirdOrderDerivatives</td>
</tr>
<tr>
<td>idata$$[&quot;SensIndexStart&quot;]</td>
<td>F</td>
<td>- an index of the first sensitivity parameter within the currently calculated subset of first order sensitivity parameters</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- in the case of second order sensitivity the index to derivative that corresponds to diagonal element ( \phi_{\text{SensRowStart}} \phi_{\text{SensRowStart}} ) at row idata$$[&quot;SensRowStart&quot;]</td>
</tr>
<tr>
<td>idata$$[&quot;SensIndexEnd&quot;]</td>
<td>F</td>
<td>- an index of the last sensitivity parameter within the currently calculated subset of first order sensitivity parameters</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- in the case of second order sensitivity the index to derivative that corresponds diagonal element at row idata$$[&quot;SensRowEnd&quot;]</td>
</tr>
<tr>
<td>idata$$[&quot;SensRowStart&quot;]</td>
<td>F</td>
<td>an index of the first row in the currently calculated band of matrix of second order derivatives</td>
</tr>
<tr>
<td>idata$$[&quot;SensRowEnd&quot;]</td>
<td>F</td>
<td>an index of the last row in the currently calculated band of matrix of second order derivatives</td>
</tr>
<tr>
<td>idata$$[&quot;SensMaxGroupLength&quot;]</td>
<td>F</td>
<td>maximum length of the subset of sensitivity derivatives</td>
</tr>
<tr>
<td>idata$$[&quot;NoFirstOrderDerivatives&quot;]</td>
<td>F</td>
<td>number of first order derivatives (( \equiv ) idata$$[&quot;NoSensParameters&quot;]))</td>
</tr>
<tr>
<td>idata$$[&quot;NoSecondOrderDerivatives&quot;]</td>
<td>F</td>
<td>number of second order derivatives (( \equiv n_\phi (n_\phi+1)/2, ) symmetric matrix!)</td>
</tr>
<tr>
<td>idata$$[&quot;NoThirdOrderDerivatives&quot;]</td>
<td>F</td>
<td>number of third order derivatives</td>
</tr>
<tr>
<td>idata$$[&quot;SensIndex&quot;]</td>
<td>F</td>
<td>index of the currently processed sensitivity derivative for post-processing and debugging</td>
</tr>
<tr>
<td>idata$$[&quot;NoDesignVelocityFields&quot;]</td>
<td>F</td>
<td>total number of design velocity fields (user defined fields + Identity + Zero)</td>
</tr>
<tr>
<td>idata$$[&quot;NoBCFields&quot;]</td>
<td>F</td>
<td>number of nonzero boundary conditions related design velocity fields (time dependent velocity fields)</td>
</tr>
<tr>
<td>idata$$[&quot;NoSensProblems&quot;]</td>
<td>F/B</td>
<td>total number of solved sensitivity problems</td>
</tr>
<tr>
<td>idata$$[&quot;NoResponseFunctionals&quot;]</td>
<td>B</td>
<td>number of response functionals (relevant for backward mode)</td>
</tr>
<tr>
<td>idata$$[&quot;NoBackwardSteps&quot;]</td>
<td>B</td>
<td>number of time steps for backward sensitivity</td>
</tr>
<tr>
<td>idata$$[&quot;BackwardStepIndex&quot;]</td>
<td>B</td>
<td>current time step for backward sensitivity</td>
</tr>
</tbody>
</table>

The relevance of the specific data depends on the type of sensitivity analysis (forward or backward). See also Sensitivity Analysis.
### Contact related data

<table>
<thead>
<tr>
<th>Default form</th>
<th>Default</th>
<th>description</th>
</tr>
</thead>
</table>
| idata$$["ContactProblem"] | 1/R | 1 = global contact search is enabled  
0 = global contact search is disabled |
| idata$$["Contact1"], idata$$["Contact2"], idata$$["Contact3"], idata$$["Contact4"], idata$$["Contact5"] | | contact problem specific parameters |

### Real Type Environment Data

<table>
<thead>
<tr>
<th>Default form</th>
<th>Default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdata$$[&quot;Multiplier&quot;]</td>
<td>0</td>
<td>current values of the natural and essential boundary conditions are obtained by multiplying initial values with the rdata$$[&quot;Multiplier&quot;] (the value is also known as load level or load factor)</td>
</tr>
<tr>
<td>rdata$$[&quot;ResidualError&quot;]</td>
<td>$10^{55}$</td>
<td>Modified Euclidean norm of the residual vector $\sqrt{R/R}$</td>
</tr>
<tr>
<td>rdata$$[&quot;IncrementError&quot;]</td>
<td>$10^{55}$</td>
<td>Modified Euclidean norm of the last increment of global d.o.f $\sqrt{\Delta a/\Delta a}$</td>
</tr>
<tr>
<td>rdata$$[&quot;MFlops&quot;]</td>
<td></td>
<td>estimate of the number of floating point operations per second</td>
</tr>
<tr>
<td>rdata$$[&quot;SubMFlops&quot;]</td>
<td></td>
<td>number of equivalent floating point operations for the last call of the user subroutine</td>
</tr>
<tr>
<td>rdata$$[&quot;Time&quot;]</td>
<td>0</td>
<td>real time</td>
</tr>
<tr>
<td>rdata$$[&quot;TimeIncrement&quot;]</td>
<td>0</td>
<td>value of the current real time increment</td>
</tr>
<tr>
<td>rdata$$[&quot;MultiplierIncrement&quot;]</td>
<td>0</td>
<td>value of the current boundary conditions multiplier increment</td>
</tr>
<tr>
<td>rdata$$[&quot;PreviousMultiplierIncrement&quot;]</td>
<td>0</td>
<td>value of the previous boundary conditions multiplier increment (used to calculate initial values of the unknowns at the beginning of the increment)</td>
</tr>
<tr>
<td>rdata$$[&quot;SubIterationTolerance&quot;]</td>
<td>$10^{-9}$</td>
<td>tolerance for the local sub-iterative process</td>
</tr>
<tr>
<td>rdata$$[&quot;LineSearchStepLength&quot;]</td>
<td>Automatic</td>
<td>step size control factor $\eta$ (see also idata$$[&quot;LineSearchUpdate&quot;]))</td>
</tr>
<tr>
<td>rdata$$[&quot;PostMaxValue&quot;]</td>
<td>0</td>
<td>the value is set by the post-processing SMTPostData function to the true maximum value of the required quantities (note that the values returned by the SMTPostData function are smoothed over the patch of elements)</td>
</tr>
<tr>
<td>rdata$$[&quot;PostMinValue&quot;]</td>
<td>0</td>
<td>the value is set by the post-processing SMTPostData function to the true minimum value of the required quantity</td>
</tr>
<tr>
<td>rdata$$[&quot;Solver1&quot;], rdata$$[&quot;Solver2&quot;], rdata$$[&quot;Solver3&quot;], rdata$$[&quot;Solver4&quot;], rdata$$[&quot;Solver5&quot;],</td>
<td></td>
<td>solver specific parameters</td>
</tr>
<tr>
<td>rdata$$[&quot;Contact1&quot;], rdata$$[&quot;Contact2&quot;], rdata$$[&quot;Contact3&quot;], rdata$$[&quot;Contact4&quot;], rdata$$[&quot;Contact5&quot;],</td>
<td></td>
<td>contact problem specific parameters</td>
</tr>
</tbody>
</table>
Node Specification Data

One of the input parameters of the user subroutines is a list of pointers to the node specification of all nodes. The ns$$[i, ...]$$ construct represents the node specification data structure that belongs to the $i$-th local element node.
### Default form

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>ns$[i,&quot;id&quot;,&quot;SpecIndex&quot;]</td>
<td>global index of the $i$-th node specification data structure</td>
<td>1</td>
</tr>
<tr>
<td>ns$[i,&quot;id&quot;,&quot;NoDOF&quot;]</td>
<td>number of nodal d.o.f ($= \text{nd}ns[i,&quot;id&quot;,&quot;NoDOF&quot;]$)</td>
<td>1</td>
</tr>
<tr>
<td>ns$[i,&quot;id&quot;,&quot;NoNodeStorage&quot;]</td>
<td>total number of history dependent real type values per node that have to be stored in the memory for transient type of problems</td>
<td>1</td>
</tr>
<tr>
<td>ns$[i,&quot;id&quot;,&quot;NoNodeData&quot;]</td>
<td>total number of arbitrary real values per node</td>
<td>1</td>
</tr>
<tr>
<td>ns$[i,&quot;id&quot;,&quot;NoData&quot;]</td>
<td>total number of arbitrary real values per node specification</td>
<td>1</td>
</tr>
<tr>
<td>ns$[i,&quot;id&quot;,&quot;NoTmpData&quot;]</td>
<td>number of temporary real type variables stored during the execution of a single analysis directive ($\max(SMTIData[&quot;MinNoTmpData&quot;],\text{NoDOF})$)</td>
<td>1</td>
</tr>
<tr>
<td>ns$[i,&quot;id&quot;,&quot;Constrained&quot;]</td>
<td>$1 \Rightarrow$ node has initially all d.o.f. constrained</td>
<td>1</td>
</tr>
<tr>
<td>ns$[i,&quot;id&quot;,&quot;Fictive&quot;]</td>
<td>$1 \Rightarrow$ The node does not represent a topological point. The switch is set automatically for the nodes with the -D and -P node identification switch.</td>
<td>1</td>
</tr>
<tr>
<td>ns$[i,&quot;id&quot;,&quot;Dummy&quot;]</td>
<td>$1 \Rightarrow$ node specification describes a dummy node</td>
<td>1</td>
</tr>
<tr>
<td>ns$[i,&quot;id&quot;,&quot;DummyNode&quot;]</td>
<td>index of the dummy node</td>
<td>1</td>
</tr>
<tr>
<td>ns$[i,&quot;Data&quot;, j]</td>
<td>arbitrary node specification specific data</td>
<td>NoData real numbers</td>
</tr>
<tr>
<td>ns$[i,&quot;NodeID&quot;]</td>
<td>node identification (see Node Identification)</td>
<td>string</td>
</tr>
<tr>
<td>ns$[i,&quot;id&quot;,&quot;Active&quot;]</td>
<td>$1 \Rightarrow$ nodes with specific node identification are active $0 \Rightarrow$ elements with specific node identification are ignored for all actions</td>
<td>integer</td>
</tr>
<tr>
<td>ns$[i,&quot;DOFScaling&quot;, j]</td>
<td>DOFScaling factors</td>
<td>NoDOF real numbers</td>
</tr>
<tr>
<td>ns$[i,&quot;SensBVFIndex&quot;, j,k]</td>
<td>an index of the boundary conditions velocity field that corresponds to the $j$-th nodal unknown and $k$-th sensitivity combination in $i$-th node specification. Index points to the &quot;ADVF&quot; node data field. It is used to construct boundary conditions related sensitivity data structures. $\text{AceFEM: index } (\partial p/\partial \phi_k) \equiv SMTNodeSpecData[\text{NodeID},\text{SensBVFIndex}]_j(k-1)\text{NoDOF}+j$ (see AceFEM implementation of Sensitivity Analysis)</td>
<td>NoSensDerivatives integer numbers</td>
</tr>
</tbody>
</table>

Node specification data structure.
Node Data

<table>
<thead>
<tr>
<th>Default form</th>
<th>Description</th>
<th>Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>ndSS[i,&quot;id&quot;, &quot;NodeIndex&quot;]</td>
<td>global index of the i-th node</td>
<td>1</td>
</tr>
<tr>
<td>ndSS[i,&quot;id&quot;,&quot;NoDOF&quot;]</td>
<td>number of nodal d.o.f</td>
<td>1</td>
</tr>
<tr>
<td>ndSS[i,&quot;id&quot;, &quot;SpecIndex&quot;]</td>
<td>index of the node specification data structure</td>
<td>1</td>
</tr>
<tr>
<td>ndSS[i,&quot;id&quot;, &quot;NoElements&quot;]</td>
<td>number of elements associated with i -th node for calculation of tangent and residual (inactive nodes are excluded)</td>
<td>1</td>
</tr>
<tr>
<td>ndSS[i,&quot;DOF&quot;, j]</td>
<td>global index of the j-th nodal d.o.f or &lt; 0 if there is an essential boundary condition assigned to the j-th d.o.f. -1 constrained by input data -2 constrained by user after SMTAnalysis (for e.g. staggered schemes) -3 constrained automatically due to the deactivation of NodeSpec.id.Active - set by SMTSetSolver -4 constrained automatically due to no elements connects with node - set by SMTSetSolver</td>
<td>NoDOF</td>
</tr>
<tr>
<td>ndSS[i, &quot;Elements&quot;, j]</td>
<td>index of j-th element associated with i -th node (field is divided into NoElements elements that are active for assembly + inactive elements and ends with 0)</td>
<td>NoElements</td>
</tr>
<tr>
<td>ndSS[i,&quot;X&quot;, j] =&gt; SMSIO[ &quot;Nodal coordinates&quot;[i,j]]</td>
<td>initial coordinates of the node</td>
<td>3 (1-X,2-Y,3-Z)</td>
</tr>
<tr>
<td>ndSS[i,&quot;at&quot;, j] =&gt; SMSIO[ &quot;Nodal DOFs&quot;[i,j]]</td>
<td>current value of the j-th nodal d.o.f (a_i^t)</td>
<td>NoDOF</td>
</tr>
<tr>
<td>ndSS[i,&quot;ap&quot;, j] =&gt; SMSIO[ &quot;Nodal DOFs n&quot;[i,j]]</td>
<td>value of the j-th nodal d.o.f at the end of previous step (a_i^p)</td>
<td>NoDOF</td>
</tr>
<tr>
<td>ndSS[i,&quot;da&quot;, j]</td>
<td>value of the increment of the j-th nodal d.o.f in last iteration (Δa_i)</td>
<td>NoDOF</td>
</tr>
<tr>
<td>ndSS[i,&quot;Bt&quot;, j]</td>
<td>ndSS[i,&quot;DOF&quot;, j] == -1 ⇒ current value of the j-th essential boundary condition ndSS[i,&quot;DOF&quot;, j] ≥ 0 ⇒ current value of the j-th natural boundary condition value of the j -th boundary condition (either essential or natural) at the end of previous step reference value of the j -th boundary condition in node i (current boundary value is defined as Bt= Bp+Δλ dB, where Δλ is the multiplier increment)</td>
<td>NoDOF</td>
</tr>
<tr>
<td>ndSS[i,&quot;Bp&quot;, j]</td>
<td></td>
<td>NoDOF</td>
</tr>
<tr>
<td>ndSS[i,&quot;dB&quot;, j]</td>
<td></td>
<td>NoDOF</td>
</tr>
<tr>
<td>ndSS[i,&quot;Data&quot;, j] =&gt; SMSIO[ &quot;Nodal data&quot;[i,/] ]</td>
<td>arbitrary node specific data + design sensitivity velocity fields+ adjoint vectors (user defined data length is internally extended to accommodate sensitivity related data)</td>
<td>NoNodeData+ NoDesignVelocityFields +NoDOF* NoResponseFunctionals real numbers</td>
</tr>
<tr>
<td>ndSS[i,&quot;ht&quot;, j] =&gt; SMSIO[ &quot;Nodal time dependent data&quot;[i,/] ]</td>
<td>current state of the j -th history dependent real type variable in the i-th node</td>
<td>NoNodeStorage real numbers</td>
</tr>
<tr>
<td>ndSS[i,&quot;hp&quot;, j] =&gt; SMSIO[ &quot;Nodal time dependent data n&quot;[i,/] ]</td>
<td>the state of the j -th history dependent real type variable in the i -th node at the end of the previous step</td>
<td>NoNodeStorage real numbers</td>
</tr>
<tr>
<td>ndSS[i,&quot;tmp&quot;, j]</td>
<td>temporary real type variables stored during the execution of a single analysis directive (restricted use)</td>
<td>Max[ndSS[i[ &quot;MinNoTmpData&quot;], NoDOF]]</td>
</tr>
</tbody>
</table>

Nodal data structures.
### Default form

<table>
<thead>
<tr>
<th>Default form</th>
<th>Description</th>
<th>Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>nd$$[i,&quot;st&quot;, j, k]$$</td>
<td>current $j$-th derivative of the $k$-th nodal d.o.f in $i$-th node</td>
<td>NoDOF* NoSensDerivatives</td>
</tr>
<tr>
<td></td>
<td><strong>AceFEM:</strong> $\frac{\partial \phi}{\partial n_i} = \text{SMTNodeData}[i,&quot;st&quot;][(j-1) \text{NoDOF}+k]$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\frac{\partial^2 \phi}{\partial n_i \partial n_k} = \text{SMTNodeData}[i,&quot;st&quot;][(\text{sensPos}(j,m)-1) \text{NoDOF}+k]$</td>
<td></td>
</tr>
<tr>
<td>nd$$[i,&quot;sp&quot;, j, k]$$</td>
<td>$j$-th derivative of the $k$-th nodal d.o.f in $i$-th node in time $n$</td>
<td>NoDOF* NoSensDerivatives</td>
</tr>
<tr>
<td>nd$$[i,&quot;ADVF&quot;, j]$$</td>
<td>the $j$-th nonzero velocity field in $i$-th node</td>
<td>NoDesignVelocityFields</td>
</tr>
</tbody>
</table>
| nd$$[i,"SDVF", j, k]$$ | the $j$-th derivative (first derivatives are followed by the second order derivatives) of $k$-th element general input data field in $i$-th node (components of the element general design velocity matrix $D\phi\Psi_e$)  
(see AceFEM implementation)  
(input structure is translated into an appropriate position at the "ADVF" data field (nd$$[i,"SDVF", j, k]=nd$$[i,"ADVF",\text{position}(j,k)]$) | interpreted data NoDesignVelocityFields |
| nd$$[i,"BSAV", j, k]$$ | value of backward sensitivity adjoint vector for the $j$-th functional in $i$-th node and $k$-th dof | interpreted data stored into Data field NoDOF* NoResponseFunctionals |
| nd$$[i,"BSFVF", j, k]$$ | Value of backward sensitivity velocity field for the $j$-th functional in $i$-th node and $k$-th dof  
(see AceFEM implementation)  
(input structure is translated into an appropriate position at the "ADVF" data field (nd$$[i,"BSFVF", j, k]=nd$$[i,"ADVF",\text{position}(j,k)]$)  
(and has length NoDOF*NoResponseFunctional and is stored for $n$-th node as folows  
$\{ \frac{\partial F_1}{\partial \phi_1}, \frac{\partial F_1}{\partial \phi_2}, \ldots, \frac{\partial F_n}{\partial \phi_1}, \frac{\partial F_n}{\partial \phi_2}, \ldots, \frac{\partial F_n}{\partial \phi_{\text{NoDOF}}} \}$) | interpreted data stored into Data field NoDOF* NoResponseFunctionals |

Nodal data related to sensitivity analysis.

For more details see AceFEM implementation of Sensitivity Analysis.

### Domain Specification Data

- General Data
- Run Mathematica from code
- Memory allocation
- Domain input data
- Mesh generation
- Numerical integration
- Graphics postprocessing
- Sensitivity analysis

### General Data
<table>
<thead>
<tr>
<th>Default form</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>es$[&quot;Code&quot;]</td>
<td>element code according to the general classification</td>
<td>string</td>
</tr>
<tr>
<td>es$[&quot;id&quot;,&quot;SpecIndex&quot;]</td>
<td>global index of the domain specification structure</td>
<td>integer</td>
</tr>
<tr>
<td>es$[&quot;id&quot;, &quot;NoDimensions&quot;]</td>
<td>number of spatial dimensions (1/2/3)</td>
<td>integer</td>
</tr>
<tr>
<td>es$[&quot;Topology&quot;]</td>
<td>element topology code (see Template Constants)</td>
<td>string</td>
</tr>
<tr>
<td>es$[&quot;MainTitle&quot;]</td>
<td>description of the element</td>
<td>string</td>
</tr>
<tr>
<td>es$[&quot;SubTitle&quot;]</td>
<td>description of the element</td>
<td>string</td>
</tr>
<tr>
<td>es$[&quot;SubSubTitle&quot;]</td>
<td>detailed description of the element</td>
<td>string</td>
</tr>
<tr>
<td>es$[&quot;Bibliography&quot;]</td>
<td>reference</td>
<td>string</td>
</tr>
<tr>
<td>es$[&quot;id&quot;, &quot;SymmetricTangent&quot;]</td>
<td>1 = element tangent matrix is symmetric</td>
<td>integer</td>
</tr>
<tr>
<td>0 = element tangent matrix is unsymmetrical</td>
<td></td>
<td></td>
</tr>
<tr>
<td>es$[&quot;id&quot;, &quot;PostIterationCall&quot;]</td>
<td>force an additional call of the SKR user subroutines after the convergence of the global solution is achieved</td>
<td>False</td>
</tr>
<tr>
<td>es$[&quot;id&quot;, &quot;NoDOFGlobal&quot;]</td>
<td>total number of global d.o.f per element</td>
<td>integer</td>
</tr>
<tr>
<td>es$[&quot;DOFGlobal&quot;, i]</td>
<td>number of d.o.f for the i-th node (each node can have different number of d.o.f)</td>
<td>NoNodes× integer</td>
</tr>
<tr>
<td>es$[&quot;id&quot;, &quot;NoDOFCondense&quot;]</td>
<td>number of d.o.f that have to be statically condensed before the element quantities are assembled to global quantities (see also Template Constants)</td>
<td>integer</td>
</tr>
<tr>
<td>es$[&quot;user&quot;, i]</td>
<td>the i-th user defined element subroutines (interpretation depends on the FE environment)</td>
<td>link</td>
</tr>
<tr>
<td>es$[&quot;id&quot;, &quot;Active&quot;]</td>
<td>1 = elements with specific domain are active</td>
<td>integer</td>
</tr>
<tr>
<td>0 = elements with specific domain are ignored for all actions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>es$[&quot;Version&quot;]</td>
<td>{AceGen version number, AceFEM version number, Mathematica version number} used to create the element</td>
<td>Real list</td>
</tr>
<tr>
<td>es$[&quot;id&quot;, &quot;WorkingVectorSize&quot;]</td>
<td>The length of system vector where AceGen generated variables are stored. In some cases (e.g. for backward differentiation of complex program structures) significant additional storage might be required.</td>
<td>Integer</td>
</tr>
<tr>
<td>Template Constant</td>
<td>AceGen external variable</td>
<td>AceFEM data</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>&quot;SMSTopology&quot;-&gt;&quot;Key&quot;</td>
<td>es$[&quot;Topology&quot;]</td>
<td>SMTDomainData[dID,&quot;Topology&quot;]</td>
</tr>
<tr>
<td>&quot;SMSNoDimensions&quot;-&gt;NoDimensions</td>
<td>es$[&quot;id&quot;,&quot;NoDimensions&quot;]</td>
<td>SMTDomainData[dID,&quot;NoDimensions&quot;]</td>
</tr>
<tr>
<td>&quot;SMSNoNodes&quot;-&gt;NoNodes</td>
<td>es$[&quot;id&quot;,&quot;NoNodes&quot;]</td>
<td>SMTDomainData[dID,&quot;NoNodes&quot;]</td>
</tr>
<tr>
<td>&quot;SMSDOFGlobal&quot;-&gt;[N,...]</td>
<td>es$[&quot;DOFGlobal&quot;,i]</td>
<td>SMTDomainData[dID,&quot;DOFGlobal&quot;]</td>
</tr>
<tr>
<td>&quot;SMSNoDOFCondense&quot;-&gt;NoDOFCondense</td>
<td>es$[&quot;id&quot;,&quot;NOFCondense&quot;]</td>
<td>SMTDomainData[dID,&quot;NOFCondense&quot;]</td>
</tr>
<tr>
<td>&quot;SMSNodeID&quot;-&gt;&quot;Key&quot; ...</td>
<td>es$[&quot;NodeID&quot;,i]</td>
<td>SMTDomainData[dID,&quot;NodeID&quot;]</td>
</tr>
<tr>
<td>&quot;SMSSMainTitle&quot;-&gt;&quot;ab&quot;</td>
<td>es$[&quot;MainTitle&quot;]</td>
<td>SMTDomainData[dID,&quot;MainTitle&quot;]</td>
</tr>
<tr>
<td>&quot;SMSSubTitle&quot;-&gt;&quot;ab&quot;</td>
<td>es$[&quot;SubTitle&quot;]</td>
<td>SMTDomainData[dID,&quot;SubTitle&quot;]</td>
</tr>
<tr>
<td>&quot;SMSSubSubTitle&quot;-&gt;&quot;ab&quot;</td>
<td>es$[&quot;SubSubTitle&quot;]</td>
<td>SMTDomainData[dID,&quot;SubSubTitle&quot;]</td>
</tr>
<tr>
<td>&quot;SMSSymmetricTangent&quot;-&gt;True-or-False</td>
<td>es$[&quot;id&quot;,&quot;SymmetricTangent&quot;]</td>
<td>SMTDomainData[dID,&quot;SymmetricTangent&quot;]</td>
</tr>
<tr>
<td>&quot;SMSPreIterationCall&quot;-&gt;True-or-False</td>
<td>es$[&quot;PreIterationCall&quot;]</td>
<td>SMTDomainData[dID,&quot;PreIterationCall&quot;]</td>
</tr>
</tbody>
</table>

**Run Mathematica from code**

<table>
<thead>
<tr>
<th>es$[ &quot;MMAInitialisation&quot;]</th>
<th>Mathematica’s code executed after SMTAnalysis command</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>es$[ &quot;MMANextStep&quot;]</td>
<td>Mathematica’s code executed after SMTNextStep command</td>
<td>string</td>
</tr>
<tr>
<td>es$[ &quot;MMAStepBack&quot;]</td>
<td>Mathematica’s code executed after SMTStepBack command</td>
<td>string</td>
</tr>
<tr>
<td>es$[ &quot;MMAPreIteration&quot;]</td>
<td>Mathematica’s code executed before SMTNextStep command</td>
<td>string</td>
</tr>
</tbody>
</table>

**Memory allocation**

256

AceGenPrint.nb
<table>
<thead>
<tr>
<th>Default form</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>es$[&quot;id&quot;,</td>
<td>total length of the vector of history</td>
<td>integer expression</td>
</tr>
<tr>
<td>&quot;NoTimeStorage&quot;]</td>
<td>dependent variables per element (element level)</td>
<td>integer</td>
</tr>
<tr>
<td>es$[&quot;id&quot;,</td>
<td>number of input data values that are common for all elements in domain</td>
<td>NoDomainData</td>
</tr>
<tr>
<td>&quot;NoDomainData&quot;]</td>
<td>(e.g. material constants) and are provided by the user in input data</td>
<td></td>
</tr>
<tr>
<td>es$[&quot;DomainDataNames&quot;, i]</td>
<td>description of the i-th input data value that is common for all elements with the same specification</td>
<td></td>
</tr>
<tr>
<td>es$[&quot;Data&quot;,j]=SMSIO[&quot;Domain data&quot;[j]]</td>
<td>data common for all the elements within a particular domain (fixed length)</td>
<td>NoDomainData</td>
</tr>
<tr>
<td>es$[&quot;id&quot;,</td>
<td>number of additional input data values that are common for all elements in domain (e.g. flow curve points) and are provided by the user in input data (variable length)</td>
<td></td>
</tr>
<tr>
<td>&quot;NoAdditionalData&quot;]</td>
<td>additional data common for all the elements within a particular domain (variable length)</td>
<td></td>
</tr>
<tr>
<td>es$[&quot;id&quot;,</td>
<td>total length of vector of history independent variables per element (element level)</td>
<td>integer expression</td>
</tr>
<tr>
<td>&quot;NoElementData&quot;]</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>es$[&quot;id&quot;,</td>
<td>number of additional integer type environment variables (global level)</td>
<td>NoIntData</td>
</tr>
<tr>
<td>&quot;NoIData&quot;]</td>
<td>name of the i-th additional integer type environment data variable (the corresponding value can be accessed by idata$[name])</td>
<td></td>
</tr>
<tr>
<td>es$[&quot;IDataNames&quot;,i]</td>
<td>position of the i-th additional integer type environment data variable on the idata$ vector</td>
<td>NoIntData</td>
</tr>
<tr>
<td>es$[&quot;IDataIndex&quot;,i]</td>
<td>number of additional real type environment variables (global level)</td>
<td>NoIntData</td>
</tr>
<tr>
<td>es$[&quot;RDataNames&quot;,i]</td>
<td>name of the i-th additional real type environment data variable (the corresponding value can be accessed by rdata$[name])</td>
<td></td>
</tr>
<tr>
<td>es$[&quot;RDataIndex&quot;,i]</td>
<td>position of the i-th additional real type environment data variable on the rdata$ vector</td>
<td>NoRData</td>
</tr>
<tr>
<td>es$[&quot;id&quot;,</td>
<td>number of character type user defined constants (domain level)</td>
<td>NoCharSwitch</td>
</tr>
<tr>
<td>&quot;NoCharSwitch&quot;]</td>
<td>i-th character type user defined constant</td>
<td>0</td>
</tr>
<tr>
<td>es$[&quot;CharSwitch&quot;,i]</td>
<td>number of integer type user defined constants (domain level)</td>
<td>NoIntegerSwitch</td>
</tr>
<tr>
<td>es$[&quot;IntSwitch&quot;]</td>
<td>i-th integer type user defined constant</td>
<td>0</td>
</tr>
<tr>
<td>es$[&quot;IntSwitch&quot;,i]</td>
<td>number of real type user defined constants (domain level)</td>
<td>NoDoubleSwitch</td>
</tr>
<tr>
<td>es$[&quot;DoubleSwitch&quot;]</td>
<td>i-th real type user defined constant</td>
<td>NoNodes</td>
</tr>
<tr>
<td>es$[&quot;NoNodeStorage&quot;,i]</td>
<td>total length of the vector of history dependent real variables for the i-th node (node level)</td>
<td>NoNodes</td>
</tr>
<tr>
<td>es$[&quot;NoNodeData&quot;,i]</td>
<td>total length of the vector of arbitrary real values for the i-th node (node level)</td>
<td>NoNodes</td>
</tr>
<tr>
<td>Template constant</td>
<td>AceGen external variables</td>
<td>AceFEM data</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>---------------------------------------------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>&quot;SMSNoTimeStorage&quot; -&gt; eN</td>
<td>edSS[&quot;id&quot;,&quot;NoTimeStorage&quot;]</td>
<td>SMTDomainData[dID,&quot;NoTimeStorage&quot;]</td>
</tr>
<tr>
<td></td>
<td>edSS[&quot;ht&quot;,&quot;i&quot;]</td>
<td>SMTElementData[e,&quot;ht&quot;][i]</td>
</tr>
<tr>
<td></td>
<td>edSS[&quot;hp&quot;,&quot;i&quot;]</td>
<td>SMTElementData[e,&quot;hp&quot;][i]</td>
</tr>
<tr>
<td>&quot;SMSDomainDataNames&quot; -&gt; {&quot;Key&quot; ...}</td>
<td>esSS[&quot;id&quot;,&quot;NoDomainData&quot;]</td>
<td>SMTDomainData[dID,&quot;NoDomainData&quot;]</td>
</tr>
<tr>
<td></td>
<td>esSS[&quot;DomainDataNames&quot;,i]</td>
<td>SMTDomainData[dID,&quot;DomainDataNames&quot;]</td>
</tr>
<tr>
<td></td>
<td>esSS[&quot;Data&quot;,i]=SMSIO[&quot;Domain data&quot;[i]]</td>
<td>SMTDomainData[dID,&quot;Data&quot;][i]</td>
</tr>
<tr>
<td>&quot;SMSNoAdditionalData&quot; -&gt; NoAdditionalData</td>
<td>esSS[&quot;id&quot;,&quot;NoAdditionalData&quot;]</td>
<td>SMTDomainData[dID,&quot;NoAdditionalData&quot;]</td>
</tr>
<tr>
<td></td>
<td>esSS[&quot;AdditionalData&quot;,i]</td>
<td>SMTDomainData[dID,&quot;AdditionalData&quot;][i]</td>
</tr>
<tr>
<td>&quot;SMSNoElementData&quot; -&gt; eN</td>
<td>edSS[&quot;id&quot;,&quot;NoElementData&quot;]</td>
<td>SMTDomainData[dID,&quot;NoElementData&quot;]</td>
</tr>
<tr>
<td></td>
<td>edSS[&quot;Data&quot;,i]</td>
<td>SMTElementData[e,&quot;Data&quot;][i]</td>
</tr>
<tr>
<td>&quot;SMSNoNodeStorage&quot; -&gt; eN</td>
<td>ndSS[en,&quot;ht&quot;,i]</td>
<td>SMTDomainData[dID,&quot;NoElementData&quot;]</td>
</tr>
<tr>
<td></td>
<td>ndSS[en,&quot;hp&quot;,i]</td>
<td>SMTNodeData[gn,&quot;ht&quot;][i]</td>
</tr>
<tr>
<td></td>
<td>SMTNodeData[gn,&quot;hp&quot;][i]</td>
<td>SMTNodeData[gn,&quot;hp&quot;][i]</td>
</tr>
<tr>
<td>&quot;SMSNoNodeData&quot; -&gt; eN</td>
<td>edSS[&quot;id&quot;,&quot;NoNodeData&quot;]</td>
<td>SMTDomainData[dID,&quot;NoNodeData&quot;]</td>
</tr>
<tr>
<td></td>
<td>edSS[&quot;Data&quot;,i]</td>
<td>SMTElementData[e,&quot;Data&quot;][i]</td>
</tr>
<tr>
<td>&quot;SMSIDataNames&quot; -&gt; {&quot;Key&quot; ...}</td>
<td>esSS[&quot;id&quot;,&quot;NoIData&quot;]</td>
<td>SMTDomainData[dID,&quot;NoIData&quot;]</td>
</tr>
<tr>
<td></td>
<td>esSS[&quot;IDataIndex&quot;,i]</td>
<td>SMTDomainData[dID,&quot;IDataNames&quot;]</td>
</tr>
<tr>
<td></td>
<td>idataSS[&quot;K&quot;]</td>
<td>SMTData[&quot;K&quot;]</td>
</tr>
<tr>
<td>&quot;SMRDDataNames&quot; -&gt; {&quot;Key&quot; ...}</td>
<td>esSS[&quot;id&quot;,&quot;NoRData&quot;]</td>
<td>SMTDomainData[dID,&quot;NoRData&quot;]</td>
</tr>
<tr>
<td></td>
<td>esSS[&quot;RDataIndex&quot;,i]</td>
<td>SMTDomainData[dID,&quot;RDataNames&quot;]</td>
</tr>
<tr>
<td></td>
<td>rdataSS[&quot;K&quot;]</td>
<td>SMTData[&quot;K&quot;]</td>
</tr>
<tr>
<td>&quot;SMSCharSwitch&quot; -&gt; {&quot;Key&quot; ...}</td>
<td>esSS[&quot;id&quot;,&quot;NoCharSwitch&quot;]</td>
<td>SMTDomainData[dID,&quot;NoCharSwitch&quot;]</td>
</tr>
<tr>
<td></td>
<td>esSS[&quot;CharSwitch&quot;,i]</td>
<td>SMTDomainData[dID,&quot;CharSwitch&quot;][i]</td>
</tr>
<tr>
<td>&quot;SMISIntSwitch&quot; -&gt; {v1,...}</td>
<td>esSS[&quot;id&quot;,&quot;NoIntSwitch&quot;]</td>
<td>SMTDomainData[dID,&quot;NoIntSwitch&quot;]</td>
</tr>
<tr>
<td></td>
<td>esSS[&quot;IntSwitch&quot;,i]</td>
<td>SMTDomainData[dID,&quot;IntSwitch&quot;][i]</td>
</tr>
<tr>
<td>&quot;SMSDoubleSwitch&quot; -&gt; {v1,...}</td>
<td>esSS[&quot;id&quot;,&quot;NoDoubleSwitch&quot;]</td>
<td>SMTDomainData[dID,&quot;NoDoubleSwitch&quot;]</td>
</tr>
<tr>
<td></td>
<td>esSS[&quot;DoubleSwitch&quot;,i]</td>
<td>SMTDomainData[dID,&quot;DoubleSwitch&quot;][i]</td>
</tr>
</tbody>
</table>
### Mesh generation

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoNodes</td>
<td>number of nodes per element</td>
<td>integer</td>
</tr>
<tr>
<td>NodeSpec,i</td>
<td>node specification index for the $i$-th node</td>
<td>NoNodes</td>
</tr>
<tr>
<td>NodeID,i</td>
<td>integer number that is used for identification of the nodes in the case of multi-field problems for all nodes</td>
<td>NoNodes*</td>
</tr>
<tr>
<td>AdditionalNodes</td>
<td>pure function (see Function) that returns coordinates of nodes additional to the user defined nodes that are nodes required by the element (if node is a dummy node than coordinates are replaced by the symbol Null)</td>
<td>pure function</td>
</tr>
</tbody>
</table>

### Numerical integration
<table>
<thead>
<tr>
<th>Default form</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>es$\text{SS}[^{\text{id}},[^{\text{IntCode}}]}$</td>
<td>integration code according to the general classification (see Numerical Integration)</td>
<td>integer</td>
</tr>
<tr>
<td>es$\text{SS}[^{\text{id}},[^{\text{DefaultIntegrationCode}}]}$</td>
<td>default numerical integration code (Numerical Integration). Value is initialized by template constant SMSDefaultIntegrationCode (see Template Constants).</td>
<td>integer</td>
</tr>
<tr>
<td>es$\text{SS}[^{\text{id}},[^{\text{NoIntPoints}}]}$</td>
<td>total number of integration points for numerical integration (see Numerical Integration)</td>
<td>integer</td>
</tr>
<tr>
<td>es$\text{SS}[^{\text{id}},[^{\text{NoIntPointsA}}]}$</td>
<td>number of integration points for first integration code (see Numerical Integration)</td>
<td>integer</td>
</tr>
<tr>
<td>es$\text{SS}[^{\text{id}},[^{\text{NoIntPointsB}}]}$</td>
<td>number of integration points for second integration code (see Numerical Integration)</td>
<td>integer</td>
</tr>
<tr>
<td>es$\text{SS}[^{\text{id}},[^{\text{NoIntPointsC}}]}$</td>
<td>number of integration points for third integration code (see Numerical Integration)</td>
<td>integer</td>
</tr>
<tr>
<td>es$\text{SS}[^{\text{IntPoints}},[^{\text{v,p}}]}$</td>
<td>coordinates and weights of the numerical integration points $\xi_v, \eta_p, \zeta_p$ and $w_p$</td>
<td>NoIntPoints*4 real numbers</td>
</tr>
</tbody>
</table>

### Template Constant

<table>
<thead>
<tr>
<th>&quot;SMSDefaultIntegrationCode&quot;-&gt;n</th>
<th>AceGen external variables</th>
<th>AceFEM data</th>
</tr>
</thead>
<tbody>
<tr>
<td>es$\text{SS}[^{\text{id}},[^{\text{DefaultIntegrationCode}}]}$</td>
<td>SMTDomainData[^{\text{dID}}, &quot;DefaultIntegrationCode&quot;]</td>
<td></td>
</tr>
<tr>
<td>es$\text{SS}[^{\text{id}},[^{\text{IntCode}}]}$</td>
<td>SMTDomainData[^{\text{dID}}, &quot;IntCode&quot;]</td>
<td></td>
</tr>
<tr>
<td>es$\text{SS}[^{\text{id}},[^{\text{NoIntPoints}}]}$</td>
<td>SMTDomainData[^{\text{dID}}, &quot;NoIntPoints&quot;]</td>
<td></td>
</tr>
<tr>
<td>SMSIO[^{\text{No. integration points}}]</td>
<td>SMTDomainData[^{\text{dID}}, &quot;NoIntPointsA&quot;]</td>
<td></td>
</tr>
<tr>
<td>es$\text{SS}[^{\text{id}},[^{\text{NoIntPointsA}}]}$</td>
<td>SMTDomainData[^{\text{dID}}, &quot;NoIntPointsB&quot;]</td>
<td></td>
</tr>
<tr>
<td>es$\text{SS}[^{\text{id}},[^{\text{NoIntPointsB}}]}$</td>
<td>SMTDomainData[^{\text{dID}}, &quot;NoIntPointsC&quot;]</td>
<td></td>
</tr>
<tr>
<td>es$\text{SS}[^{\text{id}},[^{\text{NoIntPointsC}}]}$</td>
<td>SMTDomainData[^{\text{dID}}, &quot;IntPoints&quot;, 4* ([p-1] + v)]</td>
<td></td>
</tr>
<tr>
<td>es$\text{SS}[^{\text{IntPoints}},[^{\text{v,p}}]}$</td>
<td>SMSIO[^{\text{Integration point}}[^{\text{p}}]], SMSIO[^{\text{Integration weight}}[^{\text{p}}]]</td>
<td></td>
</tr>
</tbody>
</table>

Relations between the template constants, AceGen and AceFEM
Graphics post-processing

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>es$[&quot;id&quot;,&quot;NoGPostData&quot;]</td>
<td>number of post-processing quantities per material point (see Standard User Subroutines)</td>
<td>integer</td>
</tr>
<tr>
<td>es$[&quot;id&quot;,&quot;NoNPostData&quot;]</td>
<td>number of post-processing quantities per node (see Standard User Subroutines)</td>
<td>integer</td>
</tr>
<tr>
<td>es$[&quot;GPostNames&quot;, i]</td>
<td>description of the <em>i</em>-th post-processing quantities evaluated at each material point (see Standard User Subroutines)</td>
<td>NoGPostData×string</td>
</tr>
<tr>
<td>es$[&quot;NPostNames&quot;, i]</td>
<td>description of the <em>i</em>-th post-processing quantities evaluated at each nodal point (see Standard User Subroutines)</td>
<td>NoNPostData×string</td>
</tr>
<tr>
<td>es$[&quot;Segments&quot;, i]</td>
<td>sequence of element node indices that defines the segments on the surface or outline of the element (e.g. for “Q1” topology {1,2,3,4,0})</td>
<td>NoSegmentPoints×integer</td>
</tr>
<tr>
<td>es$[&quot;id&quot;,&quot;NoSegmentPoints&quot;]</td>
<td>the length of the es$[&quot;Segments&quot;] field</td>
<td>integer</td>
</tr>
<tr>
<td>es$[&quot;ReferenceNodes&quot;,i]</td>
<td>coordinates of the nodes in a reference coordinate system (reference coordinate system is specified by the integration code)</td>
<td>NoNodes*3</td>
</tr>
<tr>
<td>es$[&quot;PostNodeWeights&quot;,i]</td>
<td>see SMTPostData</td>
<td>NoNodes real numbers</td>
</tr>
<tr>
<td>es$[&quot;AdditionalGraphics&quot;]</td>
<td>pure function (see Function) that is called for each element and returns additional graphics primitives per element (see Template Constants)</td>
<td>string</td>
</tr>
</tbody>
</table>

Relations between the template constants, AceGen and AceFEM

Sensitivity analysis
<table>
<thead>
<tr>
<th>Default form</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>es$[$&quot;id&quot;, &quot;NoSensNames&quot;]</td>
<td>number of element input data fields (except the nodal unknowns!)</td>
<td>integer</td>
</tr>
<tr>
<td>es$[$&quot;SensitivityNames&quot;, i]</td>
<td>set of element input data fields identifiers (e.g. &quot;X&quot;, &quot;Y&quot;, &quot;u&quot;.... except the nodal unknowns!)</td>
<td>NoSensNames*string</td>
</tr>
<tr>
<td>es$[$&quot;SensPVFIndex&quot;, i]</td>
<td>is a vector of indexes that point to the &quot;ADVF&quot; node data field where the element general design velocity fields are stored $\begin{bmatrix} \frac{\partial \Phi_1}{\partial \eta_1}, \ldots, \frac{\partial \Phi_1}{\partial \eta_\text{NoSensNames}}, \ldots, \frac{\partial \Phi_t}{\partial \eta_1}, \ldots, \frac{\partial \Phi_t}{\partial \eta_\text{NoSensNames}} \end{bmatrix}$ Data is used to interpret the &quot;SDVF&quot; keyword.</td>
<td>NoSensDerivatives*integer numbers</td>
</tr>
<tr>
<td>es$[$&quot;SensLowerOrderIndex&quot;, i]</td>
<td>for all higher order sensitivity derivatives indexes of the sensitivity parameters (or 0 if the lower order derivatives does not exist) $\ldots$, sensPos ($\phi^n_1$), sensPos ($\phi^n_2$), sensPos ($\phi^n_t$), sensPos ($\phi^n_{t+1}$), sensPos ($\phi^n_{t+2}$), ..., $n=1,\ldots,\text{NoSensDerivatives}$</td>
<td>6*NoSensDerivatives integer numbers</td>
</tr>
<tr>
<td>es$[$&quot;SensIndexi&quot;, n]</td>
<td>= sensPos ($\phi^n_1$) (taken from es$[$&quot;SensLowerOrderIndex&quot; vector)</td>
<td>integer</td>
</tr>
<tr>
<td>es$[$&quot;SensIndexj&quot;, n]</td>
<td>= sensPos ($\phi^n_2$) (taken from es$[$&quot;SensLowerOrderIndex&quot; vector)</td>
<td>integer</td>
</tr>
<tr>
<td>es$[$&quot;id&quot;, &quot;ShapeSensitivity&quot;]</td>
<td>1 = shape sensitivity is supported</td>
<td>integer</td>
</tr>
<tr>
<td>es$[$&quot;id&quot;, &quot;EBCSensitivity&quot;]</td>
<td>0 = shape sensitivity is not supported</td>
<td>integer</td>
</tr>
<tr>
<td>es$[$&quot;id&quot;, &quot;SensitivityOrder&quot;]</td>
<td>Order of sensitivity analysis that is supported by the element: 1 = sensitivity analysis is not supported 2 = first order sensitivity analysis is supported 3 = second order sensitivity analysis is supported</td>
<td>integer</td>
</tr>
<tr>
<td>es$[$&quot;ExtraSensitivityData&quot;, i]</td>
<td>1- length of backward sensitivity element time storage (BSETS data field attached to the ed$[$&quot;Data&quot;)</td>
<td>integer list</td>
</tr>
</tbody>
</table>

For more details see AceFEM implementation of Sensitivity Analysis.
## Element Data

<table>
<thead>
<tr>
<th>Default form</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>edSS[&quot;id&quot;,&quot;ElemIndex&quot;]</td>
<td>global index of the element</td>
<td>integer</td>
</tr>
<tr>
<td>edSS[&quot;id&quot;,&quot;SpecIndex&quot;]</td>
<td>index of the domain specification data structure</td>
<td>integer</td>
</tr>
<tr>
<td>edSS[&quot;id&quot;,&quot;Active&quot;]</td>
<td>&gt;0 active element for assembly</td>
<td>integer</td>
</tr>
<tr>
<td>100 - visualization, original</td>
<td>200 - no visualization, automatic hidden surface, original</td>
<td>integer</td>
</tr>
<tr>
<td>300 - no visualization by user, potential surface, original</td>
<td>400 - no visualization by user, inner element, original</td>
<td>integer</td>
</tr>
<tr>
<td>X+1 - added elements after SMTAnalysis</td>
<td>X+10 - elements inactive for SMTPost</td>
<td>integer</td>
</tr>
<tr>
<td>&lt;0 inactive element for assembly</td>
<td>-100 - visualization, original</td>
<td>integer</td>
</tr>
<tr>
<td>-200 - no visualization, automatic hidden surface, original</td>
<td>-300 - no visualization by user, potential surface, original</td>
<td>integer</td>
</tr>
<tr>
<td>-400 - no visualization by user, inner element, original</td>
<td>X-1 - added elements after SMTAnalysis</td>
<td>integer</td>
</tr>
<tr>
<td>X-10 - elements inactive for SMTPost</td>
<td>-1000 - deleted from memory</td>
<td>integer</td>
</tr>
<tr>
<td>edSS[&quot;Nodes&quot;,j]</td>
<td>index of the j-th element nodes</td>
<td>NoNodes</td>
</tr>
<tr>
<td>edSS[&quot;Data&quot;,j]=SMSIO[&quot;Element data&quot;[j]]</td>
<td>arbitrary element specific data</td>
<td>NoElementData</td>
</tr>
<tr>
<td>edSS[&quot;ht&quot;,j]=SMSIO[&quot;Element time dependent data&quot;[j]]</td>
<td>current state of the j-th history dependent real type variable</td>
<td>NoTimeStorage</td>
</tr>
<tr>
<td>edSS[&quot;hp&quot;,j]=SMSIO[&quot;Element time dependent data n&quot;[j]]</td>
<td>the state of the j-th history dependent real type variable at the end of the previous step</td>
<td>NoTimeStorage</td>
</tr>
<tr>
<td>edSS[&quot;BSETS&quot;,j]</td>
<td>backward sensitivity element time storage (attached at the end of edSS[&quot;Data&quot;,_])</td>
<td>SMSEextraSensitivityData[][1]]</td>
</tr>
</tbody>
</table>

Element data structure.
Element Topology

Contents
- Zero dimensional
- One dimensional
- Two dimensional
- Three dimensional
- SMCFEMTopologyData

Element topology type defines an outline of the element, dimensions, embedded spatial dimensions, number of nodes, default number of DOF per node, etc. The topology of the element can be defined in several basic ways:

- When the element has one of the standard topologies with fixed number of nodes, then the proper interface for all supported environments is automatically generated. E.g. the SMSTemplate["SMSTopology"-> "Q1"] command defines two dimensional, 4 node element.
- Standard topology with fixed number of nodes can be enhanced by an arbitrary number of additional nodes (see Node Identification).
- Element topology with arbitrary number of nodes and nonstandard numbering of nodes, but known general topology is defined with an “X” at the end. E.g. the SMSTemplate["SMSTopology"-> "TX", SMSNoNodes->5] command defines an element that has a triangular shape and 5 nodes, but the numbering of the nodes is arbitrary. All nodes have to be specified at the mesh generation phase.
- If the element topology is completely unknown (“SMSTopology"-> "XX"), then the number of dimensions and the number of nodes have to be specified explicitly and the proper interface is left to the user.

The coordinate systems in the figures below are only informative (e.g. X, Y can also stand for axisymmetric coordinate system X, Y, ϕ).

SMCFEMTopologyData

topology/.SMCFEMTopologyData
returns a list of an additional data related to specific topology

- 1 ... spatial dimensions
- 2 ... number of nodes
- 3 ... segments
- 4 ... element dimensions
- 5 ... description

In[37]:= "Q1" /. SMCFEMTopologyData
Out[37]= {2, 4, {{1, 2, 3, 4}}, 2, Quadrilateral}

Undefined

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Node numbering</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;XX&quot;</td>
<td>user defined topology</td>
<td>arbitrary</td>
</tr>
</tbody>
</table>
Zero dimensional

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Node numbering</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;V1&quot;</td>
<td>point in 1 D</td>
<td>1</td>
</tr>
<tr>
<td>&quot;V2&quot;</td>
<td>point in 2 D</td>
<td>1</td>
</tr>
<tr>
<td>&quot;V3&quot;</td>
<td>point in 3 D</td>
<td>1</td>
</tr>
</tbody>
</table>

One dimensional

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Node numbering</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;D1&quot;</td>
<td>element with 2 nodes in 1 D</td>
<td>1-2</td>
</tr>
<tr>
<td>&quot;D2&quot;</td>
<td>element with 3 nodes in 1 D</td>
<td>1-2-3</td>
</tr>
<tr>
<td>&quot;DX&quot;</td>
<td>element with arbitrary number of nodes in 1 D</td>
<td>arbitrary</td>
</tr>
<tr>
<td>&quot;L1&quot;</td>
<td>straight line segment in 2D bounded by two nodes</td>
<td></td>
</tr>
<tr>
<td>&quot;L2&quot;</td>
<td>curve in 2 D with three nodes</td>
<td></td>
</tr>
<tr>
<td>&quot;LX&quot;</td>
<td>curve with arbitrary number of nodes and arbitrary numbering in 2 D</td>
<td>arbitrary</td>
</tr>
<tr>
<td>&quot;C1&quot;</td>
<td>straight line segment in 3D bounded by two nodes</td>
<td></td>
</tr>
<tr>
<td>&quot;C2&quot;</td>
<td>curve with 3 nodes in 3 D</td>
<td></td>
</tr>
<tr>
<td>&quot;CX&quot;</td>
<td>curve with arbitrary number of nodes and arbitrary numbering in 3 D</td>
<td>arbitrary</td>
</tr>
</tbody>
</table>

Two dimensional

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Node numbering</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;T1&quot;</td>
<td>triangle with 3 nodes in 2 D (numerical integration rules and post-processing routines assume &quot;AREA COORDINATES&quot; of the reference element!)</td>
<td></td>
</tr>
<tr>
<td>&quot;T2&quot;</td>
<td>triangle with 6 nodes in 2 D (numerical integration rules and post-processing routines assume &quot;AREA COORDINATES&quot;</td>
<td></td>
</tr>
<tr>
<td>Shape</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>&quot;TX&quot;</td>
<td>Triangle with arbitrary number of nodes and arbitrary numbering in 2 D</td>
<td></td>
</tr>
<tr>
<td>&quot;Q1&quot;</td>
<td>Quadrilateral with 4 nodes in 2 D</td>
<td></td>
</tr>
<tr>
<td>&quot;Q2&quot;</td>
<td>Quadrilateral with 9 nodes in 2 D</td>
<td></td>
</tr>
<tr>
<td>&quot;Q2S&quot;</td>
<td>Quadrilateral with 8 nodes (serendipity family) in 2 D</td>
<td></td>
</tr>
<tr>
<td>&quot;QX&quot;</td>
<td>Quadrilateral with arbitrary number of nodes and arbitrary numbering in 2 D</td>
<td></td>
</tr>
<tr>
<td>&quot;P1&quot;</td>
<td>Triangle with 3 nodes in 3 D</td>
<td></td>
</tr>
<tr>
<td>&quot;P2&quot;</td>
<td>Triangle with 6 nodes in 3 D</td>
<td></td>
</tr>
<tr>
<td>&quot;PX&quot;</td>
<td>Triangle with arbitrary number of nodes and arbitrary numbering in 3 D</td>
<td></td>
</tr>
<tr>
<td>&quot;S1&quot;</td>
<td>Quadrilateral with 4 nodes in 3 D</td>
<td></td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>&quot;S2&quot;</td>
<td>quadrilateral with 9 nodes in 3 D</td>
<td></td>
</tr>
<tr>
<td>&quot;S2S&quot;</td>
<td>quadrilateral with 8 nodes (serendipity family) in 3 D</td>
<td></td>
</tr>
<tr>
<td>&quot;SX&quot;</td>
<td>quadrilateral with arbitrary number of nodes and arbitrary numbering in 3 D</td>
<td></td>
</tr>
</tbody>
</table>

**Three dimensional**
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Node numbering</th>
</tr>
</thead>
</table>
| "O1" | tetrahedron with 4 nodes  
(numerical integration rules and post-processing routines assume "AREA CCORDINATES" of the reference element!) | ![Diagram](image1.png) |
| "O2" | tetrahedron with 10 nodes  
(numerical integration rules and post-processing routines assume "AREA CCORDINATES" of the reference element!) | ![Diagram](image2.png) |
| "OX" | tetrahedron with arbitrary number of nodes  
and arbitrary numbering | arbitrary |
| "H1" | hexahedron with 8 nodes | ![Diagram](image3.png) |
| "H2" | hexahedron with 27 nodes | ![Diagram](image4.png) |
| "H2S" | hexahedron with 20 nodes (serendipity family) | ![Diagram](image5.png) |
"HX" hexahedron with arbitrary number of nodes and arbitrary numbering
Node Identification

Contents

- Node Types and Switches
- AceFEM Code Generation Utility Functions
  - SMSIsDOFConstrained
  - SMSNoDummyNodes
  - SMSLastTrueNode
  - SMSLastTrueDOF
  - SMSIsDummyNode
  - SMSIsRealNode
- Self - transforming meshes
- Node Identification Examples

Node Types and Switches

The node identification is a string that is used for identification of the nodes accordingly to the physical meaning of the nodal unknowns. Node identification is used by the SMTAnalysis command to construct the final FE mesh on a basis of the user defined topological mesh. Node identification can have additional switches (see table below). In order to have consistent set of elements one has to use the same names for the nodes with the same physical meaning. Standard node identification are: "D" - node with displacements for d.o.f., "DFi" - node with displacements and rotations for d.o.f., "T"-node with temperature d.o.f, "M"- node with magnetic potential d.o.f. etc..

Some node identification are reserved and can not be used. Reserved node identification are as follows:

- "Multiplier"
- "Time"
- "X" "Y" "Z" "ID"
  This strings cannot be used because they are used to select nodes (see Selecting Nodes).

Accordingly to the node identification switches a node can be one of three basic types:

- **Topological node**
  Topological node belongs to a specific position in space. It can have associated unknowns.

- **Auxiliary node**
  Auxiliary node does not belong to a specific position in space and is created automatically. Auxiliary node can have associated unknowns. Instead of the nodal coordinates a Null sign must be given. Actual coordinates in a data base are set to zero. An auxiliary node can be a **local auxiliary node**, thus created for each element separately or a **global auxiliary node**, thus created at the level of the structure.
  See also Self - transforming meshes, Mixed 3D Solid FE, Auxiliary Nodes

- **Dummy node**
  Dummy node does not belong to a specific position in space and have no associated unknowns. Instead of the nodal coordinates a Null sign must be given. The actual coordinates of the node in a data base are set to zero. Only one nodal data structure is generated for all dummy nodes with particular node identification. Dummy nodes can only appear as automatically generated additional nodes. The dummy nodes can not be selected directly by SMTFindNodes command. An index of the dummy node associated with nodes with node identification NodeID can be accessed by SMTNodeSpecData[NodeID,"DummyNode"] command.
### Switch description

<table>
<thead>
<tr>
<th>Switch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-LP</td>
<td>The node with the switch –LP is a <strong>local auxiliary node</strong>. The –LP switch implies switches –P –S –F –T.</td>
</tr>
<tr>
<td>-GP</td>
<td>The node with the switch –GP is a <strong>global auxiliary node</strong>. The –GP switch implies switches –P –S –F –E.</td>
</tr>
<tr>
<td>-D</td>
<td>The node with switch –D is a standard <strong>dummy</strong> node. The ”–D” switch implies switches –C –F –S. (&quot;name –D&quot; is identical to &quot;name$$&quot;).</td>
</tr>
</tbody>
</table>
| -M     | A node with the switch M is:  
  a) **real node (topological or auxiliary)** if there already exist a node with the same node specification and the same coordinates introduced by other element,  
  b) **dummy node** if the corresponding real node does not exist (the switch can be used in the case of multi-field problems for nodes representing secondary fields that are not actually calculated). |

### Basic node identifications switches.

<table>
<thead>
<tr>
<th>Switch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-P</td>
<td>The node with the switch –P is auxiliary node. The –P switch implies switches –S –F.</td>
</tr>
<tr>
<td>-C</td>
<td>The unknowns associated with the nodes with the switch –C are initially constrained (by default all the unknowns are initially unconstrained).</td>
</tr>
<tr>
<td>-T</td>
<td>A node with the switch –T is ignored by the ”Tie” command (see also SMTAnalysis.)</td>
</tr>
<tr>
<td>-S</td>
<td>Switch indicates nodes that can not be located and selected by coordinates alone (node identification has to be given explicitly as a part of criteria for selecting nodes to select nodes with –S switch, see also Selecting Nodes).</td>
</tr>
<tr>
<td>-E</td>
<td>An unknown variables associated with the node are placed at the end of the list of unknowns.</td>
</tr>
<tr>
<td>-L</td>
<td>The equations associated with the nodal unknowns always result in zeros on the main diagonal of the tangent matrix (e.g. for Lagrange type unknowns).</td>
</tr>
<tr>
<td>-AL</td>
<td>The equations associated with the nodal unknowns might result (or not) in zeros on the main diagonal of the tangent matrix (e.g. for Augmented Lagrange type unknowns).</td>
</tr>
<tr>
<td>-F</td>
<td>All nodes with the switch –F are ignored by the SMTShowMesh[”Marks”-&gt;”NodeNumber”] command, but they can be used to define the edge of the elements (see SMSSegments).</td>
</tr>
</tbody>
</table>

### Detailed node identifications switches.

- During the final mesh generation two or more nodes with the same coordinates and the same node identification are automatically joined (tied) together into a single node. Tieing of the nodes can be suppressed by the - T switch. All the nodes that should be unique for each element (internal nodes) should have - T switch in order to prevent accidental tieing. 
- The dummy node mechanism can be used to generate elements with variable number of real nodes. For example the contact element has only slave nodes when there is no contact and slave and master segment nodes in the case of contact. Thus, the master segments nodes are dummy nodes if there is no contact and real nodes in the case of contact. 
- The string type identification is transformed into the integer type identification at run time. Transformation rules are stored in a SMSNodeIDIndex variable.

### AceFEM Code Generation Utility Functions

Functions in this section are producing code sequences that are part of generated element code and executed at AceFEM run time.

**SMSIsDOFConstrained[inode, idof]**
returns True if idof-s DOF in inode-s element node is globally constrained (supported) and False otherwise

\[ \text{SMSIsDOFConstrained[inode, idof, trueValue, falseValue]} \]
returns trueValue if idof-s DOF in inode-s element node is globally constrained (supported) and falseValue otherwise

\[ \text{SMSNoDummyNodes[]} \]
returns the number of dummy element nodes (at AceFEM run time!)

\[ \text{SMSLastTrueNode[]} \]
returns an index of the last element node that is not dummy node (at AceFEM run time!)

\[ \text{SMSLastTrueDOF[]} \]
returns an index of the last unconstrained DOF (only dummy node constraints are used) in a set of all element DOF-s (at AceFEM run time!)

\[ \text{SMSIsDummyNode[inode]} \]
returns True if inode-s element node is a dummy node and False otherwise (at AceFEM run time!)

\[ \text{SMSIsRealNode[inode]} \]
returns True if inode-s element node is no a dummy node and False otherwise (at AceFEM run time!)

Functions that can be part of AceGen input for the generation of AceFEM elements.

**Self-transforming meshes**

The combination of various node types and SMSTemplate option SMSAdditionalNodes creates environment for self-transforming meshes. The basis for the self-transforming mesh is topological mesh given as user input with SMTAddMesh command. The transformation is performed by the SMTAnalysis function after the original mesh was given by the user.

Mesh transformation is defined by mesh transformation function given as an SMSAdditionalNodes option of the SMSTemplate command. Input argument of the function are the coordinates of the topological nodes given as mesh input data.

The output argument of the function must be a list nodes. The node in the list can be defined by:

- **list of coordinates**
  A new topological node will be generated, added to the global list of nodes and the index of the global node becomes an element of the list of element nodes.

- **Null**
  If the node is defined as an auxiliary node (-LP or -GP) then a new auxiliary node will be generated, added to the global list of nodes and the index of the global node becomes an element of the list of element nodes. If the node is defined as dummy node (-D) the dummy node index becomes an element of the list of element nodes.

- **local node index**
  Local node index refers to the nodes given as mesh input data. Index of the global node that correspond to the local index becomes an element of the list of element nodes. No new nodes are generated.

The length of the list returned can be:

- **less than the number of element nodes (SMTNoNodes)**
  In this case a new list of nodes is added to the user supplied list of element nodes, thus the combined length of the user defined nodes and newly generated nodes must be exactly the number of element nodes. In this case the SMSAdditionalNodes defines a list of nodes that are added to the user supplied nodes.

- **exactly the number of nodes of the element**
  If the length returned is exactly the number of the nodes of the element, then the new list replaces the user supplied list of element nodes. In this case the SMSAdditionalNodes defines an arbitrary mesh transformation function.

Examples of self-transforming meshes can be found in Mixed 3 D Solid FE, Auxiliary Nodes, Cubic triangle, Additional nodes, 2D slave node - line master segment element.
Node Identification Examples

Examples: Non-standard elements
- Element with single node. "simc -F -C -L" identifies the node with the identification "simc" that are not shown on a graphs, unknowns associated with the node are initially constrained and the resulting tangent matrix has zeros at the main diagonal.

\[
\text{SMSTemplate["SMSTopology"\rightarrow "V3", "SMSNodeID"\rightarrow "simc -F -C -L", ...]};
\]

Examples: Additional nodes generated from the element
- This defines the 2D, triangular element with an additional node automatically generated at the center of triangle. Command defines an element with the basic outline as three node triangle and with an additional node at the center of the element. In that case, various subsystems (e.g. mesh generation and post-processing) assume that the first three nodes form the standard triangle. At the mesh generation phase only the 3 nodes of the underlying "T1" topology have to be given and the forth node is generated automatically.

\[
\text{SMSTemplate["SMSTopology"\rightarrow "T1",}
\text{ "SMSNoNodes"\rightarrow 4,}
\text{ "SMSNodeID"\rightarrow \{"A", "A", "A", "A"\},}
\text{ "SMSAdditionalNodes"\rightarrow Function[\{(x1, x2, x3), ((x1 + x2 + x3) / 3)\}]};
\]

- This defines the 2D, triangular element with an additional local auxiliary node. An auxiliary node has no specific spatial position indicated by the Null constant. At the mesh generation phase only the 3 nodes of the underlying "T1" topology have to be given and the forth node is generated automatically.

\[
\text{SMSTemplate["SMSTopology"\rightarrow "T1",}
\text{ "SMSNoNodes"\rightarrow 4,}
\text{ "SMSNodeID"\rightarrow \{"A", "A", "A", "B -LP"\}, "SMSAdditionalNodes"\rightarrow Function[\{\}, \{Null\}], ...]};
\]

- The number of nodes provided as an input can be arbitrary. This will supplement missing nodes with auxiliary nodes. All nodes are by default dummy nodes, unless they are explicitly provided as an input.

\[
\text{maxNodes = 15;}
\text{SMSTemplate["SMSTopology"\rightarrow "T1",}
\text{ "SMSNoNodes"\rightarrow maxNodes,}
\text{ "SMSNodeID"\rightarrow Table["D -D", \{maxNodes\}], "SMSAdditionalNodes"\rightarrow Null]};
\]

- This changes constant triangle into quadratic one.
Examples: An arbitrary transformation of the list of nodes

- This changes the clockwise numeration of the nodes of triangle element into counter-clockwise numeration.

- This changes constant triangle into quadratic one with an alternative node ordering.

- This doubles the nodes of the triangle.
This doubles the nodes of an arbitrary polygon. The maximum number of nodes of the polygon has to specified in advance. All nodes are originally defined as dummy nodes and replaced later by the real nodes at the mesh generation phase. Note that the ## in the function definition represents the complete set of user defined nodes.

```plaintext
With[{maxNodes = 15},
SMSTemplate["SMSTopology" → "T1",
  "SMSNoNodes" → 2 maxNodes
  , "SMSNodeID" → Flatten@Table[{"D" - "D", "T" - "D"}, {maxNodes}]
  , "SMSAdditionalNodes" → Function[PadRight[Flatten[MapIndexed[#2[[1]], #] & (###), 1], 2 maxNodes, Null]]
];
];
```
Numerical Integration

Contents

- One dimensional
- Quadrilateral
- Triangle
- Tetrahedra
- Hexahedra
- Implementation of Numerical Integration
- Example 1
- Example 2

The coordinates and the weight factors for numerical integration for several standard element topologies are available. Specific numerical integration is defined by its code number.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>No. of points</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>numerical integration is not used</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>default integration code is taken accordingly to the topology of the element</td>
<td>topology dependent</td>
</tr>
<tr>
<td>&gt;0</td>
<td>integration code is taken accordingly to the given code</td>
<td></td>
</tr>
</tbody>
</table>

One dimensional

Cartesian coordinates of the reference element: \([\zeta, \eta, \zeta] \in [-1,1] \times [0,0] \times [0,0]\)

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>No. of points</th>
<th>Disposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1 point Gauss</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>2 point Gauss</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>3 point Gauss</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>4 point Gauss</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>5 point Gauss</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>6 point Gauss</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>7 point Gauss</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>8 point Gauss</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>9 point Gauss</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>10 point Gauss</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>2 point Lobatto</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>3 point Lobatto</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>4 point Lobatto</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>5 point Lobatto</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>6 point Lobatto</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>
Quadrilateral

Cartesian coordinates of the reference element: \((\zeta, \eta, \xi) \in [-1,1] \times [-1,1] \times [0,0]\)

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>No. of points</th>
<th>Disposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 point integration</td>
<td>1</td>
<td><img src="image1" alt="Dots" /></td>
</tr>
<tr>
<td>2</td>
<td>2x2 Gauss integration</td>
<td>4</td>
<td><img src="image2" alt="Dots" /></td>
</tr>
<tr>
<td>3</td>
<td>3x3 Gauss integration</td>
<td>9</td>
<td><img src="image3" alt="Dots" /></td>
</tr>
<tr>
<td>4</td>
<td>5 point special rule</td>
<td>5</td>
<td><img src="image4" alt="Dots" /></td>
</tr>
<tr>
<td>5</td>
<td>points in nodes</td>
<td>4</td>
<td><img src="image5" alt="Dots" /></td>
</tr>
<tr>
<td>{19+N,19+N}</td>
<td>NxN Gauss integration (N≤10)</td>
<td>N^2</td>
<td><img src="image6" alt="Dots" /></td>
</tr>
<tr>
<td>{29+N,29+N}</td>
<td>NxN Lobatto integration (2&lt;N≤6)</td>
<td>N^2</td>
<td><img src="image7" alt="Dots" /></td>
</tr>
</tbody>
</table>
Triangle

AREA coordinates of the reference element: \((\zeta, \eta, \xi) \in [0,1] \times [0,1] \times [0,0]\)

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>No. of points</th>
<th>Disposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>1 point integration</td>
<td>1</td>
<td><img src="image" alt="Triangle 1 point integration" /></td>
</tr>
<tr>
<td>13</td>
<td>3 point integration</td>
<td>3</td>
<td><img src="image" alt="Triangle 3 point integration" /></td>
</tr>
<tr>
<td>14</td>
<td>3 point integration</td>
<td>3</td>
<td><img src="image" alt="Triangle 3 point integration" /></td>
</tr>
<tr>
<td>16</td>
<td>4 point integration</td>
<td>4</td>
<td><img src="image" alt="Triangle 4 point integration" /></td>
</tr>
<tr>
<td></td>
<td>(negative weight at the centre!)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>7 point integration</td>
<td>7</td>
<td><img src="image" alt="Triangle 7 point integration" /></td>
</tr>
<tr>
<td>35</td>
<td>3 point integration</td>
<td>3</td>
<td><img src="image" alt="Triangle 3 point integration" /></td>
</tr>
<tr>
<td>39</td>
<td>12 point integration</td>
<td>12</td>
<td><img src="image" alt="Triangle 12 point integration" /></td>
</tr>
<tr>
<td>41</td>
<td>6 point integration</td>
<td>6</td>
<td><img src="image" alt="Triangle 6 point integration" /></td>
</tr>
<tr>
<td>42</td>
<td>7 point integration</td>
<td>7</td>
<td><img src="image" alt="Triangle 7 point integration" /></td>
</tr>
</tbody>
</table>
**Tetrahedra**

AREA coordinates of the reference element: \([\zeta, \eta, \zeta] \in [0,1] \times [0,1] \times [0,1]\)

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>No. of points</th>
<th>Disposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>1 point integration</td>
<td>1</td>
<td><img src="Image" alt="Tetrahedron" /></td>
</tr>
<tr>
<td>18</td>
<td>4 point integration</td>
<td>4</td>
<td><img src="Image" alt="Tetrahedron" /></td>
</tr>
<tr>
<td>19</td>
<td>5 point integration (negative weight at the centre!)</td>
<td>5</td>
<td><img src="Image" alt="Tetrahedron" /></td>
</tr>
<tr>
<td>40</td>
<td>14 point integration</td>
<td>14</td>
<td><img src="Image" alt="Tetrahedron" /></td>
</tr>
<tr>
<td>43</td>
<td>8 point integration</td>
<td>8</td>
<td><img src="Image" alt="Tetrahedron" /></td>
</tr>
</tbody>
</table>

**Hexahedra**

Cartesian coordinates of the reference element: \([\zeta, \eta, \zeta] \in [-1,1] \times [-1,1] \times [-1,1]\)

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>No. of points</th>
<th>Disposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1 point integration</td>
<td>1</td>
<td><img src="Image" alt="Hexahedron" /></td>
</tr>
<tr>
<td></td>
<td>Gauss Integration</td>
<td>Nodes</td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>------------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2x2x2 Gauss integration</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3x3x3 Gauss integration</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>4x4x4 Gauss integration</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>9 point special rule</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>points in nodes</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>4 point special rule</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>6 point special rule</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

\{19+N, 19+N, 19+N\} $N \times N \times N$ Gauss integration ($N \leq 10$)

\{29+N, 29+N, 29+N\} $N \times N \times N$ Lobatto integration ($2 < N \leq 6$)

Note: The values are approximations and may require further clarification or calculation.
Implementation of Numerical Integration

Numerical integration is available under all supported environments as a part of supplementary routines. The coordinates and the weights of integration points are set automatically before the user subroutines are called. They can be obtained inside the user subroutines for the $i$-th integration point in the following way:

\[
\begin{align*}
\xi_i & \gets \text{smsReal} \{\text{IntPoints},1,i\} \\
\eta_i & \gets \text{smsReal} \{\text{IntPoints},2,i\} \\
\zeta_i & \gets \text{smsReal} \{\text{IntPoints},3,i\} \\
w_i & \gets \text{smsReal} \{\text{IntPoints},4,i\}
\end{align*}
\]

where $\{\xi, \eta, \zeta\}$ are the coordinates and $w_i$ is the weight. The coordinates of the reference element are CARTESIAN for the one dimensional, quadrilateral and hexahedra topologies and AREA coordinates for the triangle and tetrahedra topologies. The integration points are constructed accordingly to the given integration code. Codes for the basic one two and three dimensional numerical integration rules are presented in tables below. Basic integration codes can be combined in order to get more complicated multidimensional integration rules. The combined code is given in the domain specification input data as a list of up to three basic codes as follows:

\[
\begin{align*}
\{\text{codeA}\} & \equiv \text{codeA} \\
\{\text{codeA,codeB}\} \\
\{\text{codeA,codeB,codeC}\}
\end{align*}
\]

where codeA, codeB and codeC are any of the basic integration codes. For example $2 \times 2 \times 5$ Gauss integration can be represented with the code $\{2, 24\}$ or equivalent code $\{21, 21, 24\}$. The integration code $7$ stands for three dimensional $8 \times 2 \times 2$ Gauss integration rule and integration code $21$ for one dimensional $2$ point Gauss integration. Thus the integration code $7$ and the code $\{21,21,21\}$ represent identical integration rule.

The numbering of the points is for the Cartesian coordinates depicted below.
Example 1

- This generates simple loop over all given integration points for 2D integration.
  
  ```math
  \text{SMSDo}[Ig, 1, \text{SMSInteger}[\text{es}[$"id", "NoIntPoints"$]]];
  (\xi, w) \rightarrow \text{Table}[\text{SMSReal}[\text{es}[$"IntPoints"$, i, Ig]] \&, \{i, \{1, 4\}\}];
  ...
  \text{SMSEndo}[];
  ```

- This generates simple loop over all given integration points for 2D integration.
  
  ```math
  \text{SMSDo}[Ig, 1, \text{SMSInteger}[\text{es}[$"id", "NoIntPoints"$]]];
  (\xi, \eta, w) \rightarrow \text{Table}[\text{SMSReal}[\text{es}[$"IntPoints"$, i, Ig]] \&, \{i, \{1, 2, 4\}\}];
  ...
  \text{SMSEndo}[];
  ```

- This generates simple loop over all given integration points for 3D integration.
  
  ```math
  \text{SMSDo}[Ig, 1, \text{SMSInteger}[\text{es}[$"id", "NoIntPoints"$]]];
  (\xi, \eta, \zeta, w) \rightarrow \text{Table}[\text{SMSReal}[\text{es}[$"IntPoints"$, i, Ig]] \&, \{i, \{1, 2, 3, 4\}\}];
  ...
  \text{SMSEndo}[];
  ```

Example 2

- In the case of the combined integration code, the integration can be also performed separately for each set of points.
(nA, nB, nC) \rightarrow \text{SMSInteger}
[\{es\$["id", "NoIntPointsA"], es\$["id", "NoIntPointsB"], es\$["id", "NoIntPointsC"]\}]

\text{SMSDo}[i\xi, 1, nA];
\xi \rightarrow \text{SMSReal}[es\$["IntPoints", 1, i\xi]];...

\text{SMSDo}[i\eta, 1, nB];
\eta \rightarrow \text{SMSReal}[es\$["IntPoints", 2, (i\eta - 1) nA + 1]];...

\text{SMSDo}[i\zeta, 1, nC];
\zeta \rightarrow \text{SMSReal}[es\$["IntPoints", 3, (i\zeta - 1) nA nB + 1]];...

w \rightarrow \text{SMSReal}[es\$["IntPoints", 4, i\xi + (i\eta - 1) nA + (i\zeta - 1) nA nB]];...

\text{SMSEndDo}[]; ...
\text{SMSEndDo}[];
\text{SMSEndDo}[];
Elimination of Local Unknowns

Some elements have additional internal degrees of freedom that do not appear as part of formulation in any other element. Those degrees of freedom can be eliminated before the assembly of the global matrix, resulting in a reduced number of equations. The structure of the tangent matrix and the residual before the elimination should be as follows:

\[
\begin{pmatrix}
K_{uu}^n & K_{uh}^n \\
K_{hu}^n & K_{hh}^n
\end{pmatrix}
\begin{pmatrix}
\Delta u^n \\
\Delta h^n
\end{pmatrix} =
\begin{pmatrix}
-R_u^n \\
-R_h^n
\end{pmatrix} \Rightarrow K_{\text{cond}} \Delta u^n = -R_{\text{cond}}
\]

where \( u \) is a global set of unknowns, 'n' is an iteration number and \( h \) is a set of unknowns that has to be eliminated. The built-in mechanism ensures automatic condensation of the local tangent matrix before the assembly of the global tangent matrix as follows:

\[
K_{\text{cond}} = K_{uu}^n - K_{uh}^n H_a^n
\]

\[
R_{\text{cond}} = R_u^n + K_{uh}^n H_b^n
\]

where \( H_a \) is a matrix and \( H_b \) a vector defined as

\[
H_a^n = K_{hh}^n - 1_K_{hu}^n
\]

\[
H_b^n = -K_{hh}^n - 1 R_h^n
\]

The actual values of the local unknowns are calculated first time when the element tangent and residual subroutine is called by:

\[
h_{n+1} = h^n + H_b^n - H_a^n \Delta u^n
\]

Three quantities have to be stored at the element level for the presented scheme: the values of the local unknowns \( h^n \), the \( H_a^n \) matrix and the \( H_b^n \) matrix. The default values are available for all constants, however, user should be careful that the default values do not interfere with his own data storage scheme. When default values are used, the system also increases the constants that specify the allocated memory per element (SMSNoTimeStorage and SMSNoElementData).

The total storage per element required for the elimination of the local unknowns is:

\[
\text{SMSNoDOFCondense} + \text{SMSNoDOFCondense} + \text{SMSNoDOFCondense} * \text{SMSNoDOFGlobal}
\]

It is assumed that the sensitivity of the local unknowns (\( \delta h^n \)) is stored as follows:

\[
\left\{ \frac{\partial h_1}{\partial p_1}, \frac{\partial h_1}{\partial p_2}, ..., \frac{\partial h_1}{\partial \text{NoSensDerivatives}}, \frac{\partial h_2}{\partial p_1}, \frac{\partial h_2}{\partial p_2}, ..., \frac{\partial h_\text{NoSensDerivatives}}{\partial \text{NoSensDerivatives}} \right\}
\]

The template constant SMSCondensationData stores pointers to the beginning of the corresponding data field as follows:

<table>
<thead>
<tr>
<th>Data</th>
<th>Position</th>
<th>Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h^n )</td>
<td>SMSCondensationData[1]</td>
<td>SMSNoDOFCondense</td>
</tr>
<tr>
<td>( H_a^n )</td>
<td>SMSCondensationData[2]</td>
<td>SMSNoDOFCondense</td>
</tr>
<tr>
<td>( H_b^n )</td>
<td>SMSCondensationData[3]</td>
<td>SMSNoDOFCondense+SMSNoDOFCondense+SMSNoDOFCondense+SMSNoDOFGlobal</td>
</tr>
<tr>
<td>( \delta h^n )</td>
<td>SMSCondensationData[4]</td>
<td>SMSNoDOFCondense+SMSNoDOFGlobal+NoSensDerivatives</td>
</tr>
</tbody>
</table>

Storage scheme for the elimination of the local unknowns.

The actual position is calculated accordingly to the data provided to the SMSTemplate command as follows:
option to SMSTemplate command | actual SMSCondensationData
---|---
"SMSNoDOFCondense"→nhe | \{edSS["ht",1], edSS["ht",nhe+1], edSS["ht",2 nhe+1], edSS["ht",2 nhe+nhe+ SMSNoDOFGlobal]\}
"SMSNoDOFCondense"→nhe | \{h_position, edSS["ht",SMSNoTimeStorage+1], edSS["ht",SMSNoTimeStorage+nhe+1], edSS["ht",SMSNoTimeStorage+nhe+nhe+SMSNoDOFGlobal+1]\}
"SMSCondensationData"→h_position | \{h_position, edSS["ht",SMSNoTimeStorage+1], edSS["ht",SMSNoTimeStorage+nhe+1], dh_position\}
"SMSCondensationData"→{h_position, dh_position} | \{h_position, Hb_position, Ha_position, dh_position\}
"SMSCondensationData"→{h_position, Hb_position, Ha_position, dh_position} | \{h_position, Hb_position, Ha_position, dh_position\}
SMSCondensationData data accordingly to the SMSTemplate command input data

All three inputs given below would yield the same default storage scheme if no time storage was previously prescribed. See also: Mixed 3D Solid FE, Elimination of Local Unknowns.

```math
\text{SMSTemplate}["SMSTopology"→"H1", "SMSNoDOFCondense"→9]
\text{SMSTemplate}["SMSTopology"→"H1", "SMSNoDOFCondense"→9, "SMSCondensationData"→edSS["ht",1], "SMSNoTimeStorage"→9]
\text{SMSTemplate}["SMSTopology"→"H1", "SMSNoDOFCondense"→9, "SMSCondensationData"→\{edSS["ht",1], edSS["ht",10], edSS["ht",19], edSS["ht",235]\}, "SMSNoTimeStorage"→234+9 idataSS["NoSensDerivatives"]]
```

Example

Let assume that SMSNoTimeStorage constant has value \(le\) before the SMSWrite command is executed and that the local unknowns were allocated by the "SMSNoDOFCondense"→nhe template constant. The true allocation of the storage is then done automatically by the SMSWrite command. The proper AceGen input and the position of the data within the "ht" history filed that corresponds to the input is as follows:

```math
\text{SMSInitialize["test", "Environment"→"AceFEM"]};
\text{SMSTemplate["SMSTopology"→"Q1", "SMSNoTimeStorage"→le, "SMSNoDOFCondense"→nhe]}
\text{hi}→\text{SMSReal[Array[edSS["ht", le + \#] & nhe]]};
\text{j}→\text{SMInteger[idataSS["SensIndex"]]};
\text{dh}→\text{SMSReal[Array[edSS["ht", le + 2 nhe + nhe+SMSNoDOFGlobal + (j - 1)*nhe + \#] & nhe]]};
```

.....

\text{SMSWrite[]}
<table>
<thead>
<tr>
<th>Data</th>
<th>Position of $i$-th element</th>
<th>position</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h^a$</td>
<td>$h^a[i]$</td>
<td>edSS[&quot;ht&quot;, $le+i$]</td>
</tr>
<tr>
<td>$H_b^a$</td>
<td>$H_b^a[i]$</td>
<td>edSS[&quot;ht&quot;, $le+nhe+i$]</td>
</tr>
<tr>
<td>$H_2^a$</td>
<td>$H_2^a[i]$</td>
<td>edSS[&quot;ht&quot;, $le+2\ nhe+i$]</td>
</tr>
<tr>
<td>$\delta h^a$</td>
<td>$\delta h^a[i]$ for $j$-th sensitivity parameter</td>
<td>edSS[&quot;ht&quot;, $le+2\ nhe+nhe^*$] SMSNoDOFGlobal+ ($j-1)*nhe+i$]</td>
</tr>
</tbody>
</table>

See also: Mixed 3D Solid FE, Elimination of Local Unknowns
Standard User Subroutines

Contents

- SMSStandardModule
- Standard User Subroutines
  - Subroutine: Tangent and residual
  - Subroutine: Postprocessing
  - Subroutine: sensitivity analysis related subroutines
  - Subroutine: Tasks

SMSStandardModule

SMSStandardModule[code]
start the definition of the user subroutine with the default names and arguments

Generation of standard user subroutines.
<table>
<thead>
<tr>
<th>codes for the user defined subroutines</th>
<th>default subroutine name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Tangent and residual&quot;</td>
<td>&quot;SKR&quot;</td>
<td>See: Subroutine: Tangent and residual Standard subroutine that returns the tangent matrix and residual for the current values of nodal and element data.</td>
</tr>
<tr>
<td>&quot;Postprocessing&quot;</td>
<td>&quot;SPP&quot;</td>
<td>See: Subroutine: Postprocessing Standard subroutine that returns post-processing quantities.</td>
</tr>
<tr>
<td>&quot;Element initialisation&quot;</td>
<td>&quot;ELI&quot;</td>
<td>User controlled additional initialisation of data (e.g. &quot;ht&quot;, &quot;hp&quot;, etc.) after the SMT/Analysis. Subroutine is called once for each element where it is defined.</td>
</tr>
<tr>
<td>&quot;Forward mode first order pseudo-load&quot;</td>
<td>&quot;SSE&quot;</td>
<td>See: Sensitivity pseudo-load user subroutine User subroutine that returns the forward mode first order pseudo-load vector for the current subset of sensitivity parameters.</td>
</tr>
<tr>
<td>&quot;Forward mode second order pseudo-load&quot;</td>
<td>&quot;SS2&quot;</td>
<td>See: Sensitivity 2nd order user subroutine Standard subroutine that returns the forward mode second order pseudo-load vector for the current subset of sensitivity parameters.</td>
</tr>
<tr>
<td>&quot;Backward mode pseudo-load&quot;</td>
<td>&quot;SBL&quot;</td>
<td>See: Backward sensitivity pseudo-load user subroutine Standard subroutine that returns the pseudo-load vector for backward mode sensitivity analysis.</td>
</tr>
<tr>
<td>&quot;Backward mode first order derivatives&quot;</td>
<td>&quot;SBG&quot;</td>
<td>See: Backward sensitivity derivatives user subroutine Standard subroutine that calculates element contribution to gradient of functional.</td>
</tr>
<tr>
<td>&quot;Backward mode second order derivatives&quot;</td>
<td>&quot;SBH&quot;</td>
<td>See: Backward sensitivity derivatives user subroutine Standard subroutine that calculates element contribution to Hessian of functional.</td>
</tr>
<tr>
<td>&quot;Residual&quot;</td>
<td>&quot;SRE&quot;</td>
<td>Standard subroutine that returns residual for the current values of the nodal and element data.</td>
</tr>
<tr>
<td>&quot;Nodal information&quot;</td>
<td>&quot;PAN&quot;</td>
<td>Standard subroutine that returns position of the nodes at current and previous time step and normal vectors if applicable (used for contact elements).</td>
</tr>
<tr>
<td>&quot;Tasks&quot;</td>
<td>&quot;Tasks&quot;</td>
<td>See: User Defined Tasks Perform various user defined tasks that require assembly of the results over the whole or part of the mesh. User subroutine &quot;Tasks&quot; is used for the communication between the AceFEM environment and the finite element and it should not be used for subroutines that are local to the element code. The ordinary subroutines local to the element code can be generated using SMSModule and SMSCall commands.</td>
</tr>
<tr>
<td>&quot;User n&quot;</td>
<td>&quot;Usern&quot;</td>
<td>n-th user defined system subroutine (low-level system feature intended to be used by advanced users)</td>
</tr>
</tbody>
</table>

Standard set of user subroutines.

<table>
<thead>
<tr>
<th>option</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Name&quot;-&gt;[&quot;name&quot;]</td>
<td>use a given name for the generated subroutine instead of default name (for the default names see table below)</td>
</tr>
<tr>
<td>&quot;AdditionalArguments&quot;-&gt;[arg1,arg2,...]</td>
<td>extends the default set of input/output arguments (see table below) by the given list of additional arguments (for the syntax of the additional arguments see SMSModule)</td>
</tr>
</tbody>
</table>

Options for SMSStandardModule.

There is a standard set of input/output arguments passed to all user subroutines as shown in the table below. The arguments are in all supported source code languages are passed "by address", so that they can be either input or output arguments. The element data
structures can be set and accessed from the element code as the AceGen external variables. For example, the command `SMSReal[nds$$["X",i]]` returns the first coordinate of the *i*-th element node. The data returned are always valid for the current element that has been processed by the FE environment.

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>es$$[...]</code></td>
<td>element specification data structure (see Element Data)</td>
</tr>
<tr>
<td><code>ed$$[...]</code></td>
<td>element data structure (see Element Data)</td>
</tr>
<tr>
<td><code>ns$$[1,...], ns$$[2,...], ...</code></td>
<td>node specification data structure for all element nodes (see Node Data)</td>
</tr>
<tr>
<td><code>nd$$[1,...], nd$$[2,...], ...</code></td>
<td>nodal data structure for all element nodes (see Node Data)</td>
</tr>
<tr>
<td><code>idata$$</code></td>
<td>integer type environment variables (see Integer Type Environment Data)</td>
</tr>
<tr>
<td><code>rdata$$</code></td>
<td>real type environment variables (see Real Type Environment Data)</td>
</tr>
</tbody>
</table>

The standard set of input/output arguments passed to all user subroutines.

Some additional I/O arguments are needed for specific tasks as follows:

<table>
<thead>
<tr>
<th>user subroutine</th>
<th>argument</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Tangent and residual&quot;</td>
<td><code>p$$[NoDOFGlobal]</code></td>
<td>element residual vector</td>
</tr>
<tr>
<td><code>s$$[NoDOFGlobal,NoDOFGlobal]</code></td>
<td>element tangent matrix</td>
<td></td>
</tr>
<tr>
<td>&quot;Postprocessing&quot;</td>
<td><code>gpost$$[NoIntPoints,NoGPostData]</code></td>
<td>integration point post-processing quantities</td>
</tr>
<tr>
<td><code>npost$$[NoNodes,NoNPostData]</code></td>
<td>nodal point post-processing quantities</td>
<td></td>
</tr>
<tr>
<td>&quot;Element initialisation&quot;</td>
<td><code>-</code></td>
<td>-</td>
</tr>
<tr>
<td>&quot;Tangent&quot;</td>
<td><code>s$$[NoDOFGlobal,NoDOFGlobal]</code></td>
<td>element tangent matrix</td>
</tr>
<tr>
<td>&quot;Residual&quot;</td>
<td><code>p$$[NoDOFGlobal]</code></td>
<td>element residual vector</td>
</tr>
<tr>
<td>&quot;Nodal information&quot;</td>
<td><code>d$$[problem dependent , 6]</code></td>
<td><code>{x_1_1,t_1_1,y_1_1,z_1_1,x_1_p,y_1_p,z_1_p},{x_2_1,t_2_1,...}-&gt;{x_1_2,...}</code></td>
</tr>
<tr>
<td>&quot;Tasks&quot;</td>
<td><code>Task$$</code></td>
<td>see User Defined Tasks, Standard User Subroutines, SMTTask</td>
</tr>
<tr>
<td><code>TasksData$$[5]</code></td>
<td><code>IntegerInput$$[TasksData$$[2]]</code></td>
<td></td>
</tr>
<tr>
<td><code>RealInput$$[TasksData$$[3]]</code></td>
<td><code>IntegerOutput$$[TasksData$$[4]]</code></td>
<td></td>
</tr>
<tr>
<td><code>RealOutput$$[TasksData$$[5]]</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Additional set of input/output arguments.

The user defined subroutines described here are connected with a particular element. For the specific tasks such as shape sensitivity analysis additional element independent user subroutines my be required (e.g. see Standard User Subroutines).

All the environments do not support all user subroutines. In the table below the accessibility of the user subroutine according to the environment is presented. The subroutine without the mark should be avoided when the code is generated for a certain environment.
This creates the element source with the environment dependent supplementary routines and the user defined subroutine "Tangent and residual". The code is created for the 2D, quadrilateral element with 4 nodes, 5 degrees of freedom per node and two material constants. Just to illustrate the procedure the $X$ coordinate of the first element node is exported as the first element of the element residual vector $p$. The element is generated for AceFEM and FEAP environments. The AceGen input and the generated codes are presented.

```
In[229]:= << AceGen`;
SMSInitialize["test", "Environment" -> "AceFEM"];
SMSTemplate["SMSTopology" -> "Q1", "SMSDOFGlobal" -> 5,
   "SMSDomainDataNames" -> {"Constant 1", "Constant 2"}];
SMSStandardModule["Tangent and residual"];
SMSExport[SMSReal[nd[1, "X", 1], p[1]]];
SMSWrite[];

Method -> SKR 1 formulae, 9 sub-expressions

File created : test.c Size : 3570
```

```
In[484]:= FilePrint["test.c"]
```

```
In[233]:= << AceGen`;
SMSInitialize["test", "Environment" -> "FEAP"];
SMSTemplate["SMSTopology" -> "Q1", "SMSDOFGlobal" -> 5,
   "SMSDomainDataNames" -> {"Constant 1", "Constant 2"}];
SMSStandardModule["Tangent and residual"];
SMSExport[SMSReal[nd[1, "X", 1], p[1]]];
SMSWrite[];

Method -> SKR10 1 formulae, 8 sub-expressions

File created : test.f Size : 7121
```
Standard User Subroutines

Subroutine: "Tangent and residual"

The "Tangent and residual" standard user subroutine returns the tangent matrix and residual for the current values of nodal and element data.

See Standard FE Procedure and

- Example: IO data defined by SMSTemplate constants
- Example: dynamically created IO data variables
- Example: low level of IO data definitions

Subroutine: "Postprocessing"

The "Postprocessing" user subroutine returns two arrays with arbitrary number of post-processing quantities as follows:

- $gpost$ array of the integration point quantities with the dimension "number of integration points" x "number of integration point quantities",
- $npost$ array of the nodal point quantities with the dimension "number of nodes" x "number of nodal point quantities".

The dimension and the contents of the arrays are defined by the two vectors of strings $MSGPostNames$ and $SMNPostNames$. They contain the keywords of post-processing quantities. Those names are also used in the analysis to identify specific quantity (see $SMTPostData$).

The keywords can be arbitrary. It is the responsibility of the user to keep the keywords of the post-processing quantities consistent for all used elements. Some keywords are reserved and have predefined meaning as follows:
See also:
- Example: IO data defined by SMSTemplate constants
- Example: dynamically created IO data variables
- Example: low-level of IO data definitions

This outlines the major parts of the "Postprocessing" user subroutine using IO data interface.

```plaintext
(* template constants related to the post-processing *)
SMSTemplate[
  "SMSSegments" → ..., "SMSReferenceNodes" → ..., 
  "SMSPostNodeWeights" → ..., "SMSAdditionalGraphics" → ...
]
SMSStandardModule["Postprocessing"]; 
(* nodal post-processing *)
{u, v} = Transpose[SMSTO["Nodal DOFs"]];
SMSTO[{"DeformedMeshX" → u, "DeformedMeshY" → v, "u" → u, "v" → v}, 
  "Export to", "Nodal points"]; 
(* integration point post-processing *)
SMSDo[1g, 1, SMSSTO["No. integration points"]]; 
...  
SMSTO[{"Sxx" → σ[1, 1], "Sxy" → σ[1, 2], ...}, "Export to", "Integration point"[Ig]]; 
SMSEndDo[];
```
This outlines the major parts of the "Postprocessing" user subroutine with low-level I/O interface.

(* template constants related to the post-processing*)

SMSTemplate[
  "SMSSegments"\to\ldots, "SMSReferenceNodes"\to\ldots,
  "SMSPostNodeWeights"\to\ldots, "SMSAdditionalGraphics"\to\ldots
]

SMSStandardModule["Postprocessing"];

(* integration point post-processing *)

SMSGPostNames = {"Sxx", "Syy", "Sxy", \ldots};

SMSDo[Ig, 1, SMSInteger[es\"["id", "NoIntPoints"]]];

(* export integration point post-processing values to Ig-th integration point*)

SMSExport[(Sxx, Syy, Sxy, \ldots), gpost\[Ig, \#1 \&];

SMSEndDo[];

(* nodal post-processing *)

SMSNPostNames = {"DeformedMeshX", "DeformedMeshY", "DeformedMeshZ", "a", "b", \ldots};

(* Example: export nodal point post-processing values for all nodes*)

SMSExport[Table[{ui[[1]], vi[[1]], w[[1]], a[[1]], b[[1]], \ldots}, (i, 1, SMSNoNodes}], npost\];

(* Example: export nodal point post-processing values to i-th nodes*)

SMSDo[node, 1, SMSNoNodes]

uNode = \ldots;

SMSExport[{uNode, vNode, wNode, aNode, bNode, \ldots}, npost\[node, \#1 \&];

SMSEndDo[];

Integration point quantities are mapped to nodes accordingly to the type of extrapolation as follows:

Type 0: Least square extrapolation from integration points to nodal points is used.

Type 1: The integration point value is multiplied by the weight factor. Weight factor can be e.g the value of the shape functions at the integration point and have to be supplied by the user. By default the last NoNodes integration point quantities are taken for the weight factors (see SMTPostData).

The type of extrapolation is defined by the value of idata\"["ExtrapolationType"] (Integer Type Environment Data). The nodal value is additionally multiplied by the user defined nodal wight factor that is stored in element specification data structure for each node (es\"["PostNodeWeights", nodenumber]). Default value of the nodal weight factor is 1 for all nodes. It can be changed by setting the SMSPostNodeWeights template constant.

Subroutine: sensitivity analysis related subroutines

See AceFEM manual sections Sensitivity Analysis.

Subroutine: "Tasks"

See AceFEM manual sections User Defined Tasks.
User Defined Environment Interface

Regenerate the heat conduction element from chapter Standard FE Procedure for arbitrary user defined C based finite element environment in a way that element description remains consistent for all environments.

- Here the SMSStandardModule["Tangent and residual"] user subroutine is redefined for user environment. Mathematica has to be restarted in order to get old definitions back!!

```mathematica
In[500]:= <<AceGen`;
SMSStandardModule["Tangent and residual"]=
SMSModule["RKt",Real[D$[2],X$[2,2],U$[2,2],load$,K$[4,4],S$[2]]];
```

- Here the replacement rules are defined that transform standard input/output parameters to user defined input/output parameters.

```mathematica
In[502]:= datarules = {nd$[i_, "X", j_] \[Rule] X$[i, j],
                       nd$[i_, "at", j_] \[Rule] U$[i, j],
                       es$["Data", i_] \[Rule] D$[i],
                       s$[i_, j_] \[Rule] K$[i, j],
                       p$[i_] \[Rule] S$[i],
                       rdata$["Multiplier"] \[Rule] load$};
```

- The element description remains essentially unchanged. Additional subroutines (for initialization, dispatching of messages, etc..) can be added to the source code using the "Splice" option of SMSWrite command. The "splice-file" is arbitrary text file that is first interpreted by the Mathematica’s Splice command and then prepended to the automatically generated source code file.
In[503]:= SMSInitialize["UserEnvironment", "Environment" -> "User", "Language" -> "C"];
SMSTemplate["SMSTopology" -> "H1", "SMSDOFGlobal" -> 1, "SMSSymmetricTangent" -> False,
"SMSDomainDataNames" -> {"Conductivity parameter k0", "Conductivity parameter k1",
"Conductivity parameter k2", "Heat source"}, "SMSS UserDataRules" -> datarules];
SMSStandardModule["Tangent and residual"]; SMSDo[ Ig, 1, SMSInteger];
Ξ = {ξ, η, ζ} ⊢ Table[SMSReal[es$["IntPoints", i, Ig]], {i, 3}];
X ⊢ Table[SMSReal[nd$[i, "X", j]], {i, SMSNoNodes}, {j, SMSNoDimensions}];
Ξ n = {{-1, -1, -1}, {1, -1, -1}, {1, 1, -1}, {-1, 1, -1}, {-1, -1, 1}, {1, -1, 1}, {1, 1, 1}, {-1, 1, 1}};
NI ⊨ Table[1/8 (1 + Ξ n〚i, 1〛) (1 + η Ξ n〚i, 2〛) (1 + ζ Ξ n〚i, 3〛), {i, 1, 8}];
X ⊢ SMSFreeze[NI.XI]; Jg ⊢ SMS[ξ, ξ]; Jgd ⊢ Det[Jg];
ϕ I ⊢ SMSReal[Table[nd$[i, "at", 1], {i, SMSNoNodes}]]; ϕ ⊨ NI.ϕ I;
{k0, k1, k2, Q} ⊢ SMSReal[Table[es$["Data", i], {i, Length[SM SDomainDataNames]]}];
k = k0 + k1 ϕ + k2 ϕ^2; SMSSetBreak["k"];
λ ⊢ SMSReal[rdata$["Multiplier"]]; wgp ⊢ SMSReal[es$["IntPoints", 4, Ig]];
SMSDo[ Dϕ = SMSD[ϕ, X, "Dependency" -> {Ξ, X, SMSInverse[Jg]]];
δϕ = SMSD[ϕ, ϕ, i];
Dδϕ = SMSD[δϕ, X, "Dependency" -> {Ξ, X, SMSInverse[Jg]]];
Rg = Jgd wgp (k Dϕ. Dϕ - δϕ λ Q); SMSExport[SMSResidualSign Rg, p$[i], "AddIn" -> True];
SMSDo[ Kg = SMSD[Rg, ϕ I]; SMSExport[Kg, s$[i, j], "AddIn" -> True];
, {j, 1, 8}];
, {i, 1, 8}];
SMSEndDo[];
SMSWrite[];
AceFEM

About AceFEM

The AceFEM package is a general finite element environment designed for solving multi-physics and multi-field problems. (see also AceFEM Overview)

Examples related to the automation of the Finite Element Method using AceFEM are part of AceFEM documentation (see Summary of Examples).
Running the generated code with the target numerical environment

AceGen can automatically compile and link the generated codes into the target numerical environment, if the proper script files are written by the user. Script files are located at the

$UserBaseDirectory/Applications/AceGen/Include/environment
directory where environment can be FEAP, ABAQUS, ANSYS or ELFEN.

User must adjust the script files accordingly to his specific operating system, FORTRAN compiler and environment version.

The following script template files are needed depending on the task executed:

<table>
<thead>
<tr>
<th>environment</th>
<th>script template files</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEAP</td>
<td>FEAPCompile.bat</td>
</tr>
<tr>
<td></td>
<td>FEAPRun.bat</td>
</tr>
<tr>
<td>ABAQUS</td>
<td>ABAQUSCompile.bat</td>
</tr>
<tr>
<td></td>
<td>ABAQUSRun.bat</td>
</tr>
<tr>
<td></td>
<td>abaqus_v6.env</td>
</tr>
<tr>
<td>ELFEN</td>
<td>ELFENCompile.bat</td>
</tr>
<tr>
<td></td>
<td>ELFENRun.bat</td>
</tr>
<tr>
<td>ANSYS</td>
<td>compileAUP.sh, SetAUP.sh</td>
</tr>
</tbody>
</table>

Script command files

 environmentCompile.bat file is a script template file used to compile the element source code into object file. The environmentRun.bat file is a script template file used to link the compiled source code into environment and run the simulation using the given simulation input data file. The script template files are using the Wolfram Language symbolic template framework (see Working with Templates). The script template file is first interpreted by the Wolfram Language Symbolic Templating framework and then written to the working directory. The following special form can be used in the template script file:

- `###slotname###` is literary replaced by the value of the slotname
- `<*expr*>` is replaced by the evaluated expr. expr is an arbitrary expression where template slots appear as #slotname (e.g. `<*#SourceCodeFileName#>*.for` would produce a name of the file together with its extension)

The following slot names are provided:
In order to compile and link a new element or user subroutine with environment we need:

- header file sms.h (available at $UserBaseDirectory/Applications/AceGen/Include/environment/sms.h)
- utility library libaceutilityf.lib (available at $UserBaseDirectory/Applications/AceGen/Include/Fortran/directory)
- user element or other user subroutines source code file.

### SMSFEEnvironment

SMSFEEnvironment[environment]

depending on given options, the command compiles the given user subroutines, links the resulting object file with chosen FE environment and runs the simulation using the given simulation input data file.

Compile user subroutines and run simulation.

<table>
<thead>
<tr>
<th>option</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;SourceCode&quot;-&gt;fname</td>
<td>None</td>
<td>compiles user subroutines specified by the file name or Unified Element Code using the environmentCompile.bat script template file.</td>
</tr>
<tr>
<td>&quot;SimulationInputFile&quot;-&gt;fname</td>
<td>None</td>
<td>runs FE environment with the fname as input data file using the environmentRun.bat script template file.</td>
</tr>
<tr>
<td>&quot;SimulationTemplate&quot;-&gt;fname</td>
<td>None</td>
<td>given template file is first processed and then written to the &quot;SimulationData&quot; file. The result is then used as input data file to perform simulation using the environmentRun.bat script template file.</td>
</tr>
<tr>
<td>&quot;SimulationOutputFile&quot;-&gt;fname</td>
<td>&quot;tmp.txt&quot;</td>
<td>the results of simulation are written to files defined by the given file name</td>
</tr>
<tr>
<td>&quot;CompileTemplate&quot;-&gt;fname</td>
<td>Automatic</td>
<td>environmentCompile.bat script template file</td>
</tr>
<tr>
<td>&quot;RunTemplate&quot;-&gt;fname</td>
<td>Automatic</td>
<td>environmentRun.bat script template file</td>
</tr>
<tr>
<td>&quot;Debug&quot;-&gt;truefalse</td>
<td>False</td>
<td>pause before exiting the script files</td>
</tr>
</tbody>
</table>

Options for SMSFEEnvironment.

### ABAQUS

ABAQUS® is a commercial FE environment developed by ABAQUS, Inc.

The generated code is linked with the ABAQUS® through the user element subroutines (UEL) and user material subroutines (UMAT). Currently the interface for ABAQUS® support direct, static implicit analysis. The interface does not support elements with the internal degrees of freedom.

**User material subroutine for ABAQUS**
AceGen provides an automatic interface for the generation of the user material subroutine for ABAQUS. Input/Output parameters of the UMAT subroutine are different from the I/O parameters of the UEL subroutine. In general UMAT receives the deformation tensors and returns the stress tensor together with the material constitutive matrix. The automatic interface ensures maximal compatibility between the standardized user element interface and user material interface, however only the constants and variables that have meaning at material point are actually available in UMAT subroutine.

The traditional FEM environments are more oriented towards mechanics of solids and require definition of an additional template constants for the construction of proper interface code. Additional numerical environment dependent constants are defined automatically and can be used within the code generation process. Default values are based on the number of spatial dimensions and are not correct for some special cases such as plane stress condition.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S[nTensorComponents]</td>
<td>components of Cauchy stress tensor ($\sigma$) (output)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F[n]</td>
<td>current deformation gradient ($F_{n+1}$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fn[n]</td>
<td>deformation gradient at time $n$ ($F_n$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>specific strain energy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>En[nTensorComponents]</td>
<td>deformation tensor at time $n$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ed[nTensorComponents]</td>
<td>increment of deformation tensor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSDE[nTensorComponents, nTensorComponents]</td>
<td>material constitutive matrix (output)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ed[&quot;ht&quot;, nTimeStorage]</td>
<td>history dependent real type values per material point</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ed[&quot;hp&quot;, nTimeStorage]</td>
<td>history dependent real type values per material point at time $n$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>es[&quot;Data&quot;, nDomainDataNames//Length]</td>
<td>vector of material constants (in input file USER MATERIAL, CONSTANTS=nDomainDataNames//Length)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

I/O parameters of the UMAT subroutine.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nTensorComponents</td>
<td>Switch[</td>
<td>the length of the vectorized stress/strain tensor</td>
</tr>
<tr>
<td>nDimensions, 1,1,2,4,3,6]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Additional ABAQUS UMAT template constants.

Stress tensor and constitutive matrix must be vectorized accordingly to ABAQUS manual. Stress tensor is vectorized as:

```latex
tensorComponents = Switch[ContinuumModel 
   , "PS", 
   
   , "PE" | "AX", 
   
   , "D3", 
   
   ]
```

Constitutive tensor is vectorized by:

```latex
Table[Extract[DoDE, Join[i1, i2]], {i1, TensorComponents}, {i2, TensorComponents}]
```
Example of 3D hyper-elastic user material subroutine for ABAQUS

```mathematica
In[1]:=
<< "AceGen`";
SMSInitialize["test", "Environment" -> "ABAQUS"];
SMSTemplate["SMSUserSubroutine" -> "UMAT"
  , "SMSTopology" -> "XX", "SMSNoDimensions" -> 3, "SMSNoNodes" -> 0, "SMSNoTensorComponents" -> 6
  , "SMSDomainDataNames" ->
    {"E - elastic modulus", "ν - poisson ratio", "β - volumetric strain exponent"}
  , "SMSDefaultData" -> {21000, 0.3, -2}];
SMSStandardModule["ABAQUS UMAT"];
{Em, ν, β} ⊢ SMSReal Table[es$$["Data", i], {i, Length[SMSDomainDataNames]}];
Postprocessing of the integration point quantities is suspended for the element.
See also: SMSReferenceNodes

◼
In[5]:=
deformation gradient: needs NLGEOM=YES set in ABAQUS input
In[6]:=
F ⊢ SMSReal Array[F$$, {3, 3}];
JF = Det[F];
SMSFreeze[be, F.F', "Symmetric" -> True];
Jbe = Det[be];
{μ, Κ} ⊨ SMSHookeToBulk[Em, ν];
W = \frac{\mu}{2} \left( Jbe^{-1/3} Tr[be] - 3 \right) + \frac{\Κ}{\β^2} \left( Jbe^{-3/2} - 1 + \frac{\β}{2} \Log[Jbe] \right);
τ = Simplify[2 be.SMSD[W, be, "Ignore" -> NumberQ, "Symmetric" -> True]];
σ = τ/JF;
DτF = SMSD[τ, F];
DoDE = Table[Sum[
  F[[1, m]] . DτF[[i, j, k, m]] + F[[k, m]] . DτF[[i, j, 1, m]], {m, 3}],
  {i, 3}, {k, 3}, {j, 3}, {i, 3}] / JF / 2;
SMSExport[W, W$$];

◼
In[17]:=
tensorComponents = {{1, 1}, {2, 2}, {3, 3}, {1, 2}, {1, 3}, {2, 3}};
SMSEXport[Extract[σ, tensorComponents], S$$];
SMSEXport[
  Table[Extract[DoDE, Join[i1, i2]], {i1, tensorComponents}, {i2, tensorComponents}], DSDE$$];

◼
In[20]:=
SMSEXport[];

File: test.for  Size: 16111  Time: 7

Method | ConstitutiveEquations | No.Formulae | No.Leafs
--- | --- | --- | ---
ConstitutiveEquations | 182 | 4287

◼ Here is the generated UMAT code compiled and linked into the ABAQUS. The SMSFEEnvironment compiles the source code file, links the resulting object file into ABAQUS and starts ABAQUS.

◼
In[21]:=
SMSEXport["ABAQUS", "SourceCode" -> "test",
  "SimulationInputFile" -> "test.dat", "SimulationOutputFile" -> "test.res"]

User element subroutine for ABAQUS

Regenerate the three-dimensional, eight node finite element described in AceFEM documentation (see Mixed 3D Solid FE, Elimination of Local Unknowns) for ABAQUS environment. The AceGen input presented in previous examples can be used again with the "Environment" -> "ABAQUS" option to produce ABAQUS’s source code file. The current ABAQUS interface does not support internal degrees of freedom. Consequently, the mixed deformation modes are skipped. The generated code is then incorporated into ABAQUS.
Example of 3D hyper-elastic user element subroutine for ABAQUS

```plaintext
Example of 3D hyper-elastic user element subroutine for ABAQUS

```

In the input file of ABAQUS we have to specify that material will be user defined:

```
*Material, NAME=user1
  Time:
  5296
```

In the ABAQUS input data file and simulation

```
$BaseDirectory/Applications/AceGen/Include/ABAQUS directory.
```

The ABAQUS input data file for the one element test example is available at the $BaseDirectory/Applications/AceGen/Include/ABAQUS directory.

```
SMSFEEnvironment["ABAQUS", "SourceCode" -> "test",
  "SimulationInputFile" -> "ABAQUSExample.dat",
  "SimulationOutputFile" -> "tmp.res"]
```

ABAQUS input data file and simulation

In the input file of ABAQUS we have to specify that material will be user defined:

```
**Material, NAME=user1
  USER MATERIAL, CONSTANTS=7
  1000, 0.2, -2.0, 100, 0, 0, 1"
```

- deformation gradient: needs NLGEOM=YES set in ABAQUS input
The compiled object file must be provided when calling the ABAQUS analysis (ifort must be called first):

- call `abaqus job = ` SimulationInputFile` user=" SourceCodeFileName.obj" interactive`

We can supply the code directly to ABAQUS and it will compile and link it itself (ifort must be called first):

- call `abaqus job = ` SimulationInputFile` user=" SourceCodeFileName.for" interactive`

If we use include or external object we have to specify the folders in which the files or libraries are located in the file "abaqus_v6.env" (we can add additional options at 'ifort' and 'LINK'). This applies to both cases, when we provide .obj file or .for file.

An examples of ABAQUS script templates

Version:

Intel Parallel Studio XE 2016 (intel fortran compiler)

ABAQUS version 6.14 - 1

```bash
# ABAQUSCompile.bat file

call "C:\Program Files (x86)\IntelSWTools\compilers_and_libraries_2016.3.207\windows\bin\ifortvars.bat" intel64 vs2012

call ifort /c /DABQ_WIN64 /extend-source /fpp /iface:cref /recursive /Qauto-scalar /QxSSE3 /QaxAVX /heap-arrays:1 /Qdiag:
file:SMSCompile.txt -o `SourceCodeFileName`.obj /I" "IncludeDirectory" " SourceCodeFileName".for

<*>This is a example of ABAQUS Compile script file

# ABAQUSRun.bat file

call "C:\Program Files (x86)\IntelSWTools\compilers_and_libraries_2016.3.207\windows\bin\ifortvars.bat" intel64 vs2012


call abaqus job=`SimulationInputFile` <"If[#SourceCodeFileName===""","","user=""#SourceCodeFileName="".obj"]>
interactive

<*>This is a example of ABAQUS Run script file
```
FEAP

FEAP is an FE environment developed by R. L. Taylor, Department of Civil Engineering, University of California at Berkeley, Berkeley, California 94720.

(refer to http://www.ce.berkeley.edu/~rlt/feap/).

FEAP is the research type FE environment with open architecture, but only basic pre/post-processing capabilities. The generated user subroutines are connected with the FEAP through its standard user subroutine interface (see Standard User Subroutines). By default, the element with the number 10 is generated.

Specific FEAP Interface Data

Additional template constants (see Template Constants) have to be specified in order to process the FEAP’s “splice-file” correctly.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEAP$ElementNumber</td>
<td>&quot;10&quot;</td>
<td>element user subroutine number (elmt ??)</td>
</tr>
</tbody>
</table>

Additional FEAP template constants.

Example: Mixed 3D Solid FE for FEAP

Regenerate the three-dimensional, eight node finite element described in AceFEM documentation (see Mixed 3D Solid FE, Elimination of Local Unknowns) for FEAP environment.
Here is the FEAP input data file for the test example from the chapter Mixed 3D Solid FE, Elimination of Local Unknowns. You need to install FEAP environment in order to run the example.

```
File: test.f  Size: 29000  Time: 7

Method  SKR10
No. Formulae  269
No. Leavs  7004

Elimination of local unknowns requires additional memory. Corresponding constants are set to:

SmsNoTimeStorage = 234 + 9 idata$ [NoSensDerivatives]  See also: Elimination of local unknowns

Test example: FEAP

- Here is the FEAP input data file for the test example from the chapter Mixed 3D Solid FE, Elimination of Local Unknowns. You need to install FEAP environment in order to run the example.

feap
0,0,0,3,3,8
Here is the generated element compiled and linked into the FEAP. The SMSFEEnvironment compiles the source code file, links the resulting object file into FEAP and starts FEAP with a feap.inp file as a input file and tmp.res as output file. The FEAP input data file for the one element test example is available at the $BaseDirectory/Applications/AceGen/Include/FEAP/directory.

In[348]:= SMSFEEnvironment["ELFEN", "SourceCode" -> "test", 
   "SimulationInputFile" -> "feap.inp", "SimulationOutputFile" -> "tmp.res"]
ELFEN

ELFEN® is commercial FE environment developed by Rockfield Software, The Innovation Centre, University of Wales College Swansea, Singleton Park, Swansea, SA2 8PP, U.K.

ELFEN is a general FE environment with the advanced pre and post-processing capabilities. The generated code is linked with the ELFEN® through the user defined subroutines. By default the element with the number 2999 is generated. Interface for ELFEN® does not support elements with the internal degrees of freedom.

Due to the non-standard way how the Newton-Raphson procedure is implemented in ELFEN, the ELFEN source codes of the elements presented in the examples section can not be obtained directly. Instead of one "Tangent and residual" user subroutine we have to generate two separate routines for the evaluation of the tangent matrix and the residual.

The traditional FEM environments are more oriented towards mechanics of solids and require definition of an additional template constants for the construction of proper interface code. Additional numerical environment dependent constants are defined automatically and can be used within the code generation process. Default values are based on the number of spatial dimensions and are not correct for some special cases such as plane stress condition.

Specific ELFEN Interface Data

Additional template constants (see Template Constants) have to be specified in order to process the ELFEN’s "splice-file" correctly. Default values for the constants are chosen accordingly to the element topology.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Default value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELFEN$ElementModel</td>
<td>&quot;L1&quot;, &quot;LX&quot;→&quot;B2&quot;</td>
<td>&quot;B2&quot;⇒two dimensional beam elements</td>
</tr>
<tr>
<td></td>
<td>&quot;C1&quot;, &quot;CX&quot;→&quot;B3&quot;</td>
<td>&quot;B3&quot;⇒three dimensional beam elements</td>
</tr>
<tr>
<td></td>
<td>&quot;T1&quot;, &quot;T2&quot;, &quot;TX&quot;,</td>
<td>&quot;PS&quot;⇒two dimensional plane stress elements</td>
</tr>
<tr>
<td></td>
<td>&quot;Q1&quot;, &quot;Q2&quot;,</td>
<td>&quot;PE&quot;⇒two dimensional plane strain elements</td>
</tr>
<tr>
<td></td>
<td>&quot;QX&quot;⇒&quot;PE&quot;</td>
<td>&quot;D3&quot;⇒three dimensional solid elements</td>
</tr>
<tr>
<td></td>
<td>&quot;P1&quot;, &quot;P2&quot;, &quot;PX&quot;, &quot;S1&quot;,</td>
<td>&quot;AX&quot;⇒axi-symmetric elements</td>
</tr>
<tr>
<td></td>
<td>&quot;S2&quot;, &quot;SX&quot;⇒&quot;SH&quot;</td>
<td>&quot;PL&quot;⇒plate elements</td>
</tr>
<tr>
<td></td>
<td>&quot;O1&quot;, &quot;O2&quot;,</td>
<td>&quot;ME&quot;⇒membrane elements</td>
</tr>
<tr>
<td></td>
<td>&quot;OX&quot;, &quot;O1&quot;, &quot;H2&quot;,</td>
<td>&quot;SH&quot;⇒shell elements</td>
</tr>
<tr>
<td></td>
<td>&quot;HX&quot;⇒&quot;D3&quot;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Default value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELFEN$NoState</td>
<td>0</td>
<td>number of state variables</td>
</tr>
<tr>
<td>SMSNoStressComponents</td>
<td>Switch[</td>
<td>the length of the vectorized stress tensor</td>
</tr>
<tr>
<td></td>
<td>SMSNoDimensions,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1,1,2,4,3,6]</td>
<td></td>
</tr>
<tr>
<td>SMSNoStrainComponents</td>
<td>Switch[</td>
<td>the length of the vectorized strain tensor</td>
</tr>
<tr>
<td></td>
<td>SMSNoDimensions,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1,1,2,4,3,6]</td>
<td></td>
</tr>
</tbody>
</table>

Additional ELFEN constants.

- Here the additional constants for the 2D, plane strain element are defined.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mswitch</td>
<td>integer mswitch</td>
<td>dimensions of the integer switch data array</td>
</tr>
<tr>
<td>switch</td>
<td>integer switch( mswitch)</td>
<td>integer type switches</td>
</tr>
<tr>
<td>meuvbl</td>
<td>integer meuvbl</td>
<td>dimensions of the element variables vlues array</td>
</tr>
<tr>
<td>lesvbl</td>
<td>integer lesvbl(meuvbl)</td>
<td>array of the element variables vlues</td>
</tr>
<tr>
<td>nehist</td>
<td>integer nehist</td>
<td>number of element dependent history variables</td>
</tr>
<tr>
<td>jfile</td>
<td>integer jfile</td>
<td>output file ( FORTRAN unit number)</td>
</tr>
<tr>
<td>morder</td>
<td>integer morder</td>
<td>dimension of the node ordering array</td>
</tr>
<tr>
<td>order</td>
<td>integer orde(morder)</td>
<td>node ordering</td>
</tr>
<tr>
<td>mgdata</td>
<td>integer mgdata</td>
<td>dimension of the element group data array</td>
</tr>
<tr>
<td>gdata</td>
<td>character*32 gdata(mgdata)</td>
<td>description of the element group specific input data values</td>
</tr>
<tr>
<td>ngdata</td>
<td>integer ngdata</td>
<td>number of the element group specific input data values</td>
</tr>
<tr>
<td>mstate</td>
<td>integer mstate</td>
<td>dimension of the state data array</td>
</tr>
<tr>
<td>state</td>
<td>character*32 state(mstate)</td>
<td>description of the element state data values</td>
</tr>
<tr>
<td>nstate</td>
<td>integer nstate</td>
<td>number of the element state data values</td>
</tr>
<tr>
<td>mgpost</td>
<td>integer mgpost</td>
<td>dimension of the integration point post-processing data array</td>
</tr>
<tr>
<td>gpost</td>
<td>character*32 gpost(mgpost)</td>
<td>description of the integration point post-processing values</td>
</tr>
<tr>
<td>ngpost</td>
<td>integer ngpost</td>
<td>total number of the integration point post-processing values</td>
</tr>
<tr>
<td>ngspost</td>
<td>integer ngspost</td>
<td>number of sensitivity parameter dependent integration point post-processing values</td>
</tr>
<tr>
<td>mnpost</td>
<td>integer mnpost</td>
<td>dimension of the integration point post-processing data array</td>
</tr>
<tr>
<td>npost</td>
<td>character*32 npost(mnpost)</td>
<td>description of the integration point post-processing values</td>
</tr>
<tr>
<td>nnpost</td>
<td>integer nnpost</td>
<td>total number of the integration point post-processing values</td>
</tr>
<tr>
<td>nnspost</td>
<td>integer nnspost</td>
<td>number of sensitivity parameter dependent integration point post-processing values</td>
</tr>
</tbody>
</table>

Example list for the SMSInnn ELFEN nnnn'th user element subroutine.

<table>
<thead>
<tr>
<th>Switch</th>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>output</td>
<td>number of gauss points</td>
</tr>
<tr>
<td>2</td>
<td>input</td>
<td>number of sensitivity parameters</td>
</tr>
</tbody>
</table>

Example: 3D Solid FE for ELFEN

Regenerate the three-dimensional, eight node finite element described in AceFEM documentation (see Mixed 3D Solid FE, Elimination of Local Unknowns) for ELFEN environment.

The AceGen input presented in previous example can be used again with the "Environment"→"ELFEN" option to produce Elfen's source code file. The procedure is controlled by the values of environment constants "SkipTangent", "SkipResidual" and "SubIterationMode".

When the tangent matrix is required the variables are set to

```plaintext
data$$["SkipTangent"] = 0,
data$$["SkipResidual"] = 1,
data$$["SubIterationMode"] = 1
```
Here is the generated element compiled and linked into the ELFEN. The SMSFEEnvironment compiles the source code file, links the resulting object file into ELFEN and starts ELFEN with a ELFENExample.dat file as an input file and tmp.res as output file. The ELFEN input data file for the one element test example is available at the $BaseDirectory/Applications/AceGen/Include/ELFEN/ directory.

![image]

Here is the generated element compiled and linked into the ELFEN. The SMSFEEnvironment compiles the source code file, links the resulting object file into ELFEN and starts ELFEN with a ELFENExample.dat file as an input file and tmp.res as output file. The ELFEN input data file for the one element test example is available at the $BaseDirectory/Applications/AceGen/Include/ELFEN/ directory.
ANSYS

ANSYS© is a commercial FE environment developed by ANSYS, Inc.

The generated code is linked with the ANSYS© through the user material subroutines (USERMAT). The interface does not support elements with the internal degrees of freedom. If we specify user material in ANSYS, it will call the function USERMAT and from USERMAT the AceGen functions.

ANSYS interface has been contributed by asist. dr. Blaž Hudobivnik, Institute of Continuum Mechanics, Leibniz University of Hannover (blaz.hudobivnik@fgg.uni-lj.si).

User material subroutine for ANSYS

AceGen provides an automatic interface for the generation of the user material subroutine for ANSYS. In general USERMAT receives the deformation tensors and returns the stress tensor together with the material constitutive matrix. The automatic interface ensures maximal compatibility between the standardized user element interface and user material interface, however only the constants and variables that have meaning at material point are actually available in USERMAT subroutine.

The traditional FEM environments are more oriented towards mechanics of solids and require definition of an additional template constants for the construction of proper interface code. Additional numerical environment dependent constants are defined automatically and can be used within the code generation process. Default values are based on the number of spatial dimensions and are not correct for some special cases such as plane stress condition.

### Abbreviation Description

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ω[309]</td>
<td>components of Cauchy stress tensor ($\sigma$) (output)</td>
</tr>
<tr>
<td>F[9]</td>
<td>current deformation gradient ($F_{n+1}$)</td>
</tr>
<tr>
<td>F[n]</td>
<td>deformation gradient at time $n$ ($F_n$)</td>
</tr>
<tr>
<td>W[9]</td>
<td>specific strain energy</td>
</tr>
<tr>
<td>E[n]</td>
<td>deformation tensor at time $n$</td>
</tr>
<tr>
<td>dE[n]</td>
<td>increment of deformation tensor</td>
</tr>
<tr>
<td>DSDE[n]</td>
<td>material constitutive matrix (output)</td>
</tr>
<tr>
<td>ed[&quot;ht&quot;,]</td>
<td>history dependent real type values per material point</td>
</tr>
<tr>
<td>ed[&quot;hp&quot;,]</td>
<td>history dependent real type values per material point at time $n$</td>
</tr>
<tr>
<td>es[&quot;Data&quot;,]</td>
<td>vector of material constants</td>
</tr>
<tr>
<td>SMSNoTensorComponents</td>
<td>Switch[Switch[SMSNoDimensions, 1,1,2,4,3,6]]</td>
</tr>
</tbody>
</table>

IO parameters of the USERMAT subroutine.

### Additional ANSYS USERMAT template constants.

Stress tensor and constitutive matrix must be vectorized accordingly to ANSYS manual. Stress tensor is vectorized as:
TensorComponents = Switch[ContinuumModel , 
  "PS", 
  ({{1, 1}, {2, 2}, {1, 2}} 
  , "PE" | "AX", 
  ({{1, 1}, {2, 2}, {3, 3}, {1, 2}} 
  , "D3", 
  ({{1, 1}, {2, 2}, {3, 3}, {1, 2}, {2, 3}, {1, 3}} 
  
  Extract[σ, TensorComponents]

Constitutive tensor is vectorized by:

Table[Extract[DoDE, Join[{i1, i2}], {i1, TensorComponents}, {i2, TensorComponents}]

Example Neo-Hooke user material for ANSYS

In[1]:=<<>AceGen`
In[2]:= SMSInitialize["ANSYS_USERMAT_NH", "Environment" -> "ANSYS"]
SMSTemplate["SMSUserSubroutine" -> "USERMAT" , "SMSTopology" -> "XX", "SMSNoDimensions" -> 3, "SMSNoNodes" -> 0, "SMSNoTensorComponents" -> 6 , "SMSDomainDataNames" -> 
  {"E - elastic modulus", "ν - poisson ratio", "β -volumetric strain exponent"} , "SMSSDefaultData" -> {1000, 0.2, -2}]

Postprocessing of the integration point quantities is suspended for the element.

See also: SMSReferenceNodes

In[4]:= SMSStandardModule["ANSYS USERMAT"];
In[5]:= {Em, ν, β} ⊢ SMSReal[Table[es$$["Data", i], {i, Length[SMSDomainDataNames]}], {i, 3}];
In[6]:= ⊢ SMSReal[Array[es$$, {3, 3}]]
In[7]:= JF = Det[F];
In[8]:= SMSFreeze[Jbe, F.F, "Symmetric" -> True];
Jbe = Det[Jbe];

In[9]:= {λ, μ} = SMSHookeToLame[Em, ν];
{μ, Κ} = SMSHookeToBulk[Em, ν];
W = \frac{μ}{2} \left(3μe^{-\frac{1}{3}} Tr[Jbe] - 3\right) + \frac{Κ}{8^2} \left(3μe^{-\frac{1}{3}} - 1 + \frac{β}{2} \log[Jbe]\right);
τ = Simplify[2 be.SMSD[W, Jbe, "Ignore" -> NumberQ, "Symmetric" -> True]]
σ = τ/JF;
DoDE = (Table[Sum[
  F[[i, m]]·DtDF[[{i, j, k, m}]] + F[[k, m]]·DtDF[[i, j, 1, m]], {m, 3}],
  {i, 3}, {j, 3}, {k, 3}, {j, 3}, {i, 3}})]/JF/2;
SMSExport[W, W$$];
In[10]:= TensorComponents = ({{1, 1}, {2, 2}, {3, 3}, {1, 2}, {2, 3}, {1, 3}};
SMSExport[Extract[σ, TensorComponents], S$$];
SMSExport[
  Table[Extract[DoDE, Join[{i1, i2}], {i1, TensorComponents}, {i2, TensorComponents}], DSDE$$];
AceGenPrint.nb

In[22]:=

311

SMSWrite[];
File: ANSYS_USERMAT_NH.for

Size: 21 518

Time: 7

ConstitutiveEquations
Method
No.Formulae 179
No.Leafs
4278
Compiling the ANSYS user materials

◼ ANSYS provides a file ANSUSERSHARED which links and compiles all the code in the chosen folder and creates a library file, which
can then be used by ANSYS.
◼ The file ANSUSERSHARED (in Windows ANSUSERSHARED.bat) is located at:
...\ANSYS Inc\v170\ansys\custom\user\winx64 or ...\ANSYS Inc\v170\ansys\customise\user\
The file needs to be copied in the folder where the Fortran code is present (workpath represents the absolute path to the file).

◼ In Windows or Linux we have change the active directory to the workpath (in CMD or Console) and then running the file
ANSUSERSHARED or ANSUSERSHARED.bat. This scripts searches for *.F files (Linux is case sensitive) and calls ifort compiler,
(in Windows ifort should be initialized first, same way as for ABAQUS). We can copy all the required libraries and include files
to the folder, and they will be compiled and linked in single file “userlib.a” (in Linux).

◼ To use the generated library (Dll or a), the system environment variable must be set globally: ANS_USER _PATH = workpath
OpenSuse:
We must save the .bashrc to our home directory, with the content:
“export ANS_USER_PATH=’workpath’”
In Windows we must set the path:
by set command:
“setx ANS_USER_PATH workpath #must be run as administrator”
or
“set ANS_USER_PATH=workpath ”
check with command:
“echo %ANS_USER_PATH%”
Using the ANSYS user materials

◼ ANSYS to use the user library file we must first make sure, that ANS_USER_PATH points to the correct folder. This can be
checked with simple Python script in workbench. (open File/Scripting/Open Command Window):
◼ print os.environ.get('ANS_USER_PATH')

◼ Path can also be set (is to be tested if it works properly) :
◼ os.environ['ANS_USER_PATH']= "workpath "

◼ ANSYS should have now user defined code. To use it we must call it first: We can define user elements or user gauss point
model. To use e.g. User material, we have to attach a script file to the model in benchmark with the following content (or
write it in input file):
◼ "tb,user,matid,2,4
tbtemp,1.0
tbdata,1,19e5, 0.3, 1e3,100
tbtemp,2.0
tbdata,1,21e5, 0.3, 2e3,100"

◼ If we want to add state variables (ANSYS does not care about the content, it just provides the interface. FOr physical meaning
we must export the variables to physical vector, e.g. if it stores plastic variables, we can put them into plastic strain tensor,
available for us):
◼ "TB,state,matid,,8"

◼ Note the “user” means we are calling the User material function. We can also specify other functions as described.
◼ ANSYS Mechanical APDL Programmer' s Reference :
2.4 .1. Subroutine UserMat (Creating Your Own Material Model)
2.4 .2. Subroutine UserMatTh (Creating Your Own Thermal Material Model)
2.4 .3. Subroutine UserHyper (Writing Your Own Hyperelasticity Laws)
2.4 .4. Subroutine UserCreep (Defining Creep Material Behavior)


2.4.5. Subroutine user_tbelastic (Defining Material Linear Elastic Properties)
2.4.6. Subroutine userfc (Defining Your Own Failure Criteria)
2.4.7. Subroutine userCZM (Defining Your Own Cohesive Zone Material)
2.4.8. Subroutine userswstrain (Defining Your Own Swelling Laws)
2.4.9. Subroutine userck (Checking User - Defined Material Data)
2.4.10. Supporting Function egen
2.4.11. Subroutine UserFld (Update User - Defined Field Variables)

An example of ANSYS script template for Linux

```bash
#!/bin/bash

#echo "pwd: $(pwd)"

#DIR="${dirname ""$(pwd)""}"

DIR="/home/userx/ANSYS/User_LP";

printf "This is a shell script. It sets ANS_USER_PATH=$DIR and reloads ".bashrc"

sed -i "s;ANS_USER_PATH=.*;ANS_USER_PATH='$DIR';" ~/.bashrc

sed -i "s;os.environ['ANS_USER_PATH']=.*;os.environ['ANS_USER_PATH']="$DIR";" ~/ANSYS/SetAUP.py

#sed -i 's;ANS_USER_PATH=.*;ANS_USER_PATH="/home/userx/ANSYS/User1";' .bashrc

source ~/.bashrc

printf "Path set to: ANS_USER_PATH=

SetAUP.sh file

#!/bin/bash

#echo "pwd: $(pwd)"

#DIR="${dirname ""$(pwd)""}"

#spaces are ignored by ANSYS!

DIR="/home/userx/ANSYS/User_LP";

"#/home/userx/ANSYS/User"

printf "This is a shell script. It sets ANS_USER_PATH=$DIR and evaluates ""ANSUSERSHARED"" file located there."

sed -i "s;ANS_USER_PATH=.*;ANS_USER_PATH='$DIR';" ~/.bashrc

sed -i "s;os.environ['ANS_USER_PATH']=.*;os.environ['ANS_USER_PATH']="$DIR";" ~/ANSYS/SetAUP.py

#sed -i 's;ANS_USER_PATH=.*;ANS_USER_PATH="/home/userx/ANSYS/User1";' .bashrc

source ~/.bashrc

cd "$ANS_USER_PATH"

printf "I am at $(pwd)"

./ANSUSERSHARED

printf ""ANSUSERSHARED" file run"
```
compileAUP.sh file
MathLink, Matlab Environments

The AceGen can build, compile and install C functions so that functions defined in the source code can be called directly from Mathematica using the MathLink protocol. The SMSInstallMathLink command builds the executable program, starts the program and installs Mathematica definitions to call functions in it.

**SMSInstallMathLink[source]**
compile source.c and source.tm source files, build the executable program, start the program and install Mathematica definitions to call functions in it

**SMSInstallMathLink[]**
create MathLink executable from the last generated AceGen source code

<table>
<thead>
<tr>
<th>option</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Optimize&quot;</td>
<td>Automatic</td>
<td>use additional compiler optimization</td>
</tr>
<tr>
<td>&quot;PauseOnExit&quot;</td>
<td>False</td>
<td>pause before exiting the MathLink executable</td>
</tr>
<tr>
<td>&quot;Console&quot;</td>
<td>True</td>
<td>on Windows start the executable as console application (see also SMSPrintMessage)</td>
</tr>
<tr>
<td>&quot;Platform&quot;</td>
<td>Automatic</td>
<td>&quot;32&quot; ⇒ 32 bit operating system (all operating systems Windows, Unix, Mac)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;64&quot; ⇒ 64 bit operating systems (Mac and Windows)</td>
</tr>
<tr>
<td>&quot;AdditionalLinkerParameters&quot;</td>
<td>&quot;&quot;</td>
<td>additional string added to the linker command line</td>
</tr>
<tr>
<td></td>
<td></td>
<td>verbatim (it can specify e.g. additional object files or libraries)</td>
</tr>
</tbody>
</table>

Options of the SMSInstallMathLink function.

The SMSInstallMathLink command executes the standard C compiler and linker. For unsupported C compilers, the user should write his own SMSInstallMathLink function that creates MathLink executable on a basis of the element source file, the sms.h header file and the SMSUtility.c file. Files can be found at the directory $BaseDirectory/Applications/AceGen/Include/MathLink/).

At run time one can effect the way how the functions are executed with an additional function SMSSetLinkOptions.

**SMSSetLinkOptions[source,options]**
sets the options for MathLink functions compiled from source source code file (run time command)

**SMSSetLinkOptions[options] ≡ SMSLinkNoEvaluations[last_AceGen_session,options]**

<table>
<thead>
<tr>
<th>option name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;PauseOnExit&quot;→value</td>
<td>True ⇒ pause before exiting the MathLink executable</td>
</tr>
<tr>
<td></td>
<td>False ⇒ exit without stopping</td>
</tr>
<tr>
<td>&quot;SparseArray&quot;→value</td>
<td>True ⇒ return all matrices in sparse format</td>
</tr>
<tr>
<td></td>
<td>False ⇒ return all matrices in full format</td>
</tr>
<tr>
<td></td>
<td>Automatic ⇒ return the matrices in a format that depends on the sparsity of the actual matrix</td>
</tr>
</tbody>
</table>

Options for SMSSetLinkOptions.

Options for SMSSetLinkOptions.

**SMSLinkNoEvaluations[source]**
returns the number of evaluations of MathLink functions compiled from source source code file during the Mathematica session (run time command)
For more examples see Standard AceGen Procedure, Minimization of Free Energy, Solution to the System of Nonlinear Equations.

The AceGen generated M-file functions can be directly imported into Matlab. See also Standard AceGen Procedure.

Example: MathLink

(MathLink)

(* The example illustrates the use of the MathLink Package.

The generated M-file can be directly imported into Matlab.

The generated executable can be used to calculate the gradient for the numerical test example.

Example: MathLink

(MathLink)

\[\text{MathLink}
\]

In[74]:= << AceGen;

\text{SMSInitialize["test", "Environment" \to "MathLink"]};

\text{SMSModule["Test", Real[\{u\$, x\$, l\$, g\$\[3\]]],

\text{"Input" \to \{u\$, x\$, l\$, g\$\", "Output" \to g\$\};

\text{(x, l) \to \{SMSReal[x\$], SMSReal[l\$]\]},

\text{ui \to SMSReal[Table[u\$[i], \{i, 3\}]]}

\text{Ni = \{x, \frac{1-x}{l} - \frac{x - L}{L}\}};

\text{u = Ni.ui;}

\text{f = u\^2;}

\text{g = SMSD[f, ui];}

\text{SMSExport[g, g\$];}

\text{SMSWrite[];}

Out[74]= \{\text{\{ui, 1, ui\}, \{ui, 2, ui\}, \{ui, 3, ui\}}\}

\text{File: test.c Size: 1838}

\text{Methods No. Formulae No. Leafs}

\text{Test 6 81}

\text{MathLink[\{test, i\_Integer, j\_Integer\}], SMSLinkNoEvaluations[\text{test}], Test[\text{u_?}

\text{\\text{ArrayQ[\#, 1, \text{Head[\#1] = Real || Head[\#1] = Integer && Dimensions[\#1] == \{3\} &&},

\text{x_? \text{\{Head[\#1] = Real || Head[\#1] = Integer \},

\text{L_? \text{\{Head[\#1] = Real || Head[\#1] = Integer \}]}}]

\text{Here the generated executable is used to calculate gradient for the numerical test example.}

In[75]:= \text{Test[\{0., 1., 7.\}, \\text{\pi / N, 10.\}]

Out[75]= \{1.37858, 3.00958, 0.945489\}
Appendix
Bibliography

AceGen


Applications


Wang P.S. (1991), Symbolic computation and parallel software, Technical Report ICM-9109-12, Department of Mathematics and Computer Science, Kent State University, USA.
AceGen Troubleshooting

General
- Rerun the input in **debug** mode ([SMSInitialize[..., "Mode" -> "Debug"]).
- Divide the input statements into the separate cells (Shift+Ctrl+D), remove the ; character at the end of the statement and check the result of each statement separately.
- Check the precedence of the special AceGen operators ⊨, ⊢, ⊣. They have lower precedence than e.g. // operator. (see also SMSR)
- Check the input parameters of the SMSVerbatim, SMSReal, SMSInteger, SMSLogical commands. They are passed into the source code verbatim, without checking the syntax, thus the resulting code may not compile correctly.
- Check that all used functions have equivalent function in the chosen compiled language. **No additional libraries** are included automatically by AceGen.
- Try to minimize the number of calls to automatic differentiation procedure. Remember that in backward mode of automatic differentiation the expression SMSD[a,c]+SMSD[b,c] can result in code that is twice as large and twice slower than the code produced by the equivalent expression SMSD[a+b,c].
- The situation when the new AceGen version gives different results than the old version does not necessary mean that there is a bug in AceGen. Even when the two versions produce mathematically equivalent expressions, the results can be different when evaluated within the finite precision arithmetics due to the different structure of the formulas. It is not only the different AceGen version but also the different Mathematica version that can produce formulae that are equivalent but not the same (e.g. formulas Sin[x]^2 + Cos[x]^2 and 1 are equivalent, but not the same).
- The expression optimization procedure can recognize various relations between expressions, however that is no assurance that relations will be in fact recognized. Thus, the users input must not rely on expression optimization as such and it must produce the same result with or without expression optimization (see Automatic Differentiation).
- Check the argument of the SMSIf command for incorrect comparitions. The expressions a===b or a=!b are executed in Mathematica and not later in a source code!!! Use the a==b and a !=b form instead of the a===b or a=!b form.
- Check the information given at www.fgg.uni-lj.si/symech/FAQ/.

Message: Variables out of scope

See extensive documentation and examples in Auxiliary Variables, SMSIf, SMSDo, SMSFictive and additional examples below.

Symbol appears outside the "If" or "Do" construct

- Erroneous input

```
In[47]= << AceGen`
SMSInitialize["test", "Language" -> "C"]
SMSModule["test", Real[<<$, f$$]]
x ⊢ SMSReal[<<$]
SMSIf[x <= $$];
f = x^2;
SMSElse[];
f + Sin[x];
SMSEndIf[];
SMSExport[f, f$$];
```

- Corrected input
Symbol is defined in other branch of "If" construct

- Erroneous input

```mathematica
<< AceGen`
SMSInitialize["test", "Language" -> "Fortran"];
SMSModule["test", Real[x$$, f$$]];
x ⊢ SMSReal[x$$];
SMSIf[x <= 0];
f = x^2;
SMSElse[];
f = Sin[x];
SMSEndIf[f];
SMSExport[f, f$$];
```

- Corrected input

```mathematica
<< AceGen`
SMSInitialize["test", "Language" -> "C"];
SMSModule["test", Real[x$$, f$$]];
x ⊢ SMSReal[x$$];
f = x;
SMSElse[];
y = 2 f;
```

Generated code does not compile correctly

The actual source code of a single formula is produced directly by *Mathematica* using CForm or FortranForm commands and not by *AceGen*. However, *Mathematica* will produce compiled language equivalent code only in the case that there exist equivalent command in compiled language. The standard form of *Mathematica* expressions can hide some special functions. Please use FullForm to see all used functions. *Mathematica* has several hundred functions and number of possible combinations that have no equivalent compiled language form. There are to ways how to get compiled language code out of symbolic input:

- one can include special libraries or write compiled language code for functions without compiled language equivalent
- make sure that symbolic input contains only functions with the compiled language equivalent or define additional transformations as in example below

Examples of erroneous and corrected input:

- Erroneous input

```mathematica
a < b < c
```

```mathematica
FullForm[a < b < c]
```

```mathematica
CForm[a < b < c]
```
There exist no standard C equivalent for Less so it is left in original form and the resulting code would probably failed to compile correctly.

- Corrected input

```mathematica
Unprotect[CForm];
CForm[Less[a_, b_, c_]] := a < b && b < c;
Protect[CForm];

CForm[a < b < c]
```

**MathLink**

- if the compilation is to slow restrict compiler optimization with `SMSInstallMathLink["Optimize" -> False]`
- in the case of sudden crash of the MathLink program use `SMSInstallMathLink["PauseOnExit" -> True]` to see the printouts generated by `SMSPrint`