UNIVERSITY OF LJUBLJANA

© **Prof. Dr.-Ing. Jože Korelc**, 2006-2023
University of Ljubljana, 2023
E–mail : AceProducts@fgg.uni-lj.si http://symech.fgg.uni-lj.si

AceGen Contents

Contents 2

Introduction 7

Introduction to AceGen 8

Preface 8

General 8

AceGen 9

Mathematica and AceGen 10

AceGen Palettes 11

AceGen Overview 13

• Installation instructions • 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 19
• Example 1: Assignments • Example 2: Conditional statements (If construct) • Example 3: Loops (Do construct) • Example 4: Conditional statements (Which construct)
AceGen Session Functions 26
• SMSInitialize • SMSModule • SMSWrite • SMSVerbatim
Examples of Multi-language Code Generation 35
• Generation of C code • Generation of MathLink code • Generation of Matlab code
User Interface 40
Contents 40
General Description 40
Presentation of Formulas 41
• 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 44
Run time debugging 44
Auxiliary Variables 45
Contents 45
Expression Optimization 45
• Probability of incorrect simplifications • Functions with unique signatures • Expression optimization and automatic differentiation
Types of Auxiliary Variables 49
• Example: real, integer and logical variables • Example: multi-valued variables
Auxiliary Variables Functions 52
• SMSR • SMSV • SMSM • SMSS • SMSSimplify • SMSVariables • SMSFreeze, SMSUnFreeze • SMSFictive
Local and Non-local Operations 63
• General Description • SMSSmartReduce • SMSSmartRestore • SMSRestore • SMSReplaceAll
Program Flow Control 70
Contents 70
General Description 70
Conditionals 71
• SMSIf, SMSElse, SMSEndIf • SMSSwitch • SMSWhich
Loops 81
• SMSDo, SMSEndDo • SMSReturn, SMSBreak, SMSContinue
Symbolic-Numeric Interface 90
Contents 90
General Description 90
Input Parameters 91
• SMSReal • SMSInteger • SMSLogical • SMSRealList
Output Parameters 95
• SMSExport
Automatic Differentiation 97
Contents 97
Theory of Automatic Differentiation 97
SMSD Function 99
Differentiation: Mathematica syntax versus AceGen syntax 100
• Mathematica • AceGen
Automatic Differentiation Examples 102
• 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 Formulae 106
• Example 1: characteristic formulae - one subset • Example 2: characteristic formulae - two subsets
Exceptions in Differentiation 110
• SMSDefineDerivative • Exception Type A: Generic Example • Exception Type C: Implicit dependencies •

Exception Type D: Alternative definition of partial derivatives
Limitations: Incorrect structure of the program 114

Verification and Debugging 116

Contents 116

General Verification Procedures 116

Printing to Output Devices 119

• 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

Run Time Debugging 127

• Example of run-time debugging • SMSSetBreak • SMSLoadSession • SMSClearBreak • SMSActivateBreak

General Utility Functions 132

Contents 132

Standard Functions With Unique Signature 132

• SMSSqrt ($\sqrt{x}$) • SMSPower (x^y)

Standard Functions Without Unique Signature 134

System Utility Functions 135

Compressing Structures 135

Example 136

Utility Functions for Numerical Environments 137

Specialized Functions 138

Linear Algebra 139

• SMSLinearSolve • SMSEigensystem • SMSLUFactor • SMSLUSolve • SMSFactorSim • SMSInverse • SMSDet • SMSKrammer • SMSEigenvalues • SMSInvariantsI • SMSInvariantsJ • SMSRotationMatrix

Tensor Algebra 144

Matrix Functions 148

• Examples

Numerical Root Finding 151

• 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

Mechanics of Solids 157

Advanced Features 158

User Defined Functions 159

Contents 159

General Description 159

• SMSCall

Examples 161

• **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 • **Embeded sequence with** *SMSVerbatim* **command:** User defined source code is embedded into generated source code • MathLink program with modules

Arrays 176

Contents 176

General Description 176

• SMSArray

Manipulating Arrays 180

• SMSPart, SMSReplacePart, SMSArrayLength • SMSDot • SMSSum

Manipulating Notebooks 185

SMSEvaluateCellsWithTag 185

• Example:

SMSRecreateNotebook 186

• Example:

Manipulate the contents of the generated notebook 186

• Example SMSRecreateNotebook: • Example: create a function with SMSTagInclude

Signatures of Expressions 191
• Example 1 • Example 2

AceGen Examples 193
Summary of AceGen Examples 194
Solution to the System of Nonlinear Equations 195
Minimization of Free Energy 197
Contents 197

Numerical Environments 210
Finite Element Environments 211
Standard FE Procedure 214
Description of FE Characteristic Steps 214
Description of Introductory Example 214
• Data interface
AceGen input using high level I/O data management 215
• 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 217
• 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 220
Contents 220
General Description 220
• SMSTemplate
Template Constants 221
• 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 defiend environments, ...)
Data Structures 226
Contents 226
General Description 227
IO Data Management 227
• General Input data • Nodal Input data • Element Input data • 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 Variables 251
• 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 Variables 257
Node Specification Data 259
Node Data 259
Domain Specification Data 261
• General Data • Run Mathematica from code • Memory allocation • Mesh generation • Numerical integration • Graphics post-processing • Sensitivity analysis
Element Data 268
Element Topology 270
Contents 270
• SMCFEMTopologyData
Undefined 270
Zero dimensional 271
One dimensional 271
Two dimensional 271
Three dimensional 274
Node Identification 277
Contents 277
Node Types and Switches 277
AceFEM Code Generation Utility Functions 278

Self-transforming meshes 279
Node Identification Examples 279
• Examples: Non-standard elements • Examples: Additional nodes generated from the element • Examples: An arbitrary transformation of the list of nodes
Numerical Integration 283
Contents 283
One dimensional 283
Quadrilateral 284
Triangle 285
Tetrahedra 286
Hexahedra 286
Implementation of Numerical Integration 289
• Example 1: SMSIO Input/Output data management • Example 2: hard coded integration rules • Example 3: low-level definition of IO data
Elimination of Local Unknowns 292
Example 293
Standard User Subroutines 295
Contents 295
SMSStandardModule 295
Standard User Subroutines 298
• Subroutine: "Tangent and residual" • Subroutine: "Postprocessing" • Subroutine: sensitivity analysis related subroutines • Subroutine: "Tasks"
User Defined Environment Interface 301
AceFEM 303
• About AceFEM
FEAP - ELFEN - ABAQUS - ANSYS 304
Contents 304
Running the generated code with the target numerical environment 304
SMSFEEnvironment 305
ABAQUS 305
• User material subroutine for ABAQUS • Example of 3D hyper-elastic user material subroutine for ABAQUS • User element subroutine for ABAQUS • ABAQUS input data file and simulation • An examples of ABAQUS script templates
FEAP 310
• Specific FEAP Interface Data • Example: Mixed 3D Solid FE for FEAP
ELFEN 313
• Specific ELFEN Interface Data • Example: 3D Solid FE for ELFEN
ANSYS 315
• User material subroutine for ANSYS • Example Neo-Hooke user material for ANSYS • Compiling the ANSYS user materials • Using the ANSYS user materials • An examples of ANSYS script template for Linux
MathLink, Matlab Environments 320
Appendix 322
Bibliography 323
AceGen Troubleshooting 326
General 326
Message: Variables out of scope 326
Symbol appears outside the "If" or "Do" construct 326
Symbol is defined in other branch of "If" construct 327
Generated code does not compile correctly 327
MathLink 328

CHAPTER 1

Introduction

Introduction to AceGen

Preface

© **Prof. Dr.-Ing. Jože Korelc**, 2006-2023
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*, hybrid symbolic-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

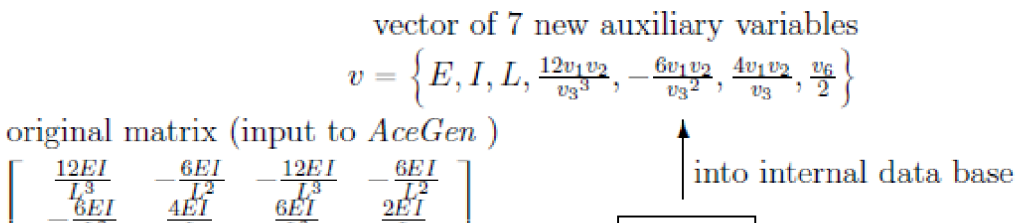
The idea implemented in *AceGen* is not to try to combine different systems, but to combine different techniques inside one system in order to avoid the above mentioned problems. Thus, the main objective is to combine techniques in such a way that will lead to an optimal environment for the design and 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 hybrid symbolic-automatic 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.

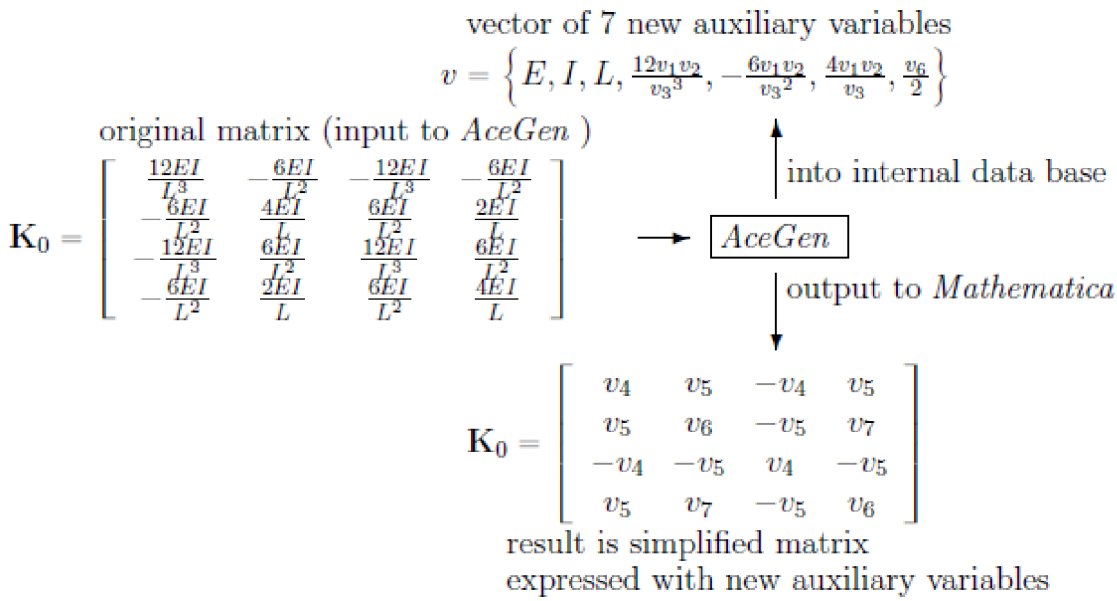
General characteristics of *AceGen* code generator:

- simultaneous optimization of expressions immediately after they have been derived (`Expression Optimization`),
- hybrid symbolic-automatic 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 (v_i), or in an original expression in parts replaced by previously created auxiliary symbols. In the first case *AceGen* stores the new expression in an internal 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.





Simultaneous simplification procedure.

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.

KORELC, Jože, Multi-language and Multi-environment Generation of Nonlinear Finite Element Codes, Engineering with Computers, 2002, 18(4):312-327

KORELC, Jože. Automatic generation of finite-element code by simultaneous optimization of expressions. Theor. comput. sci.. , 1997, 187:231-248.

Mathematica and AceGen

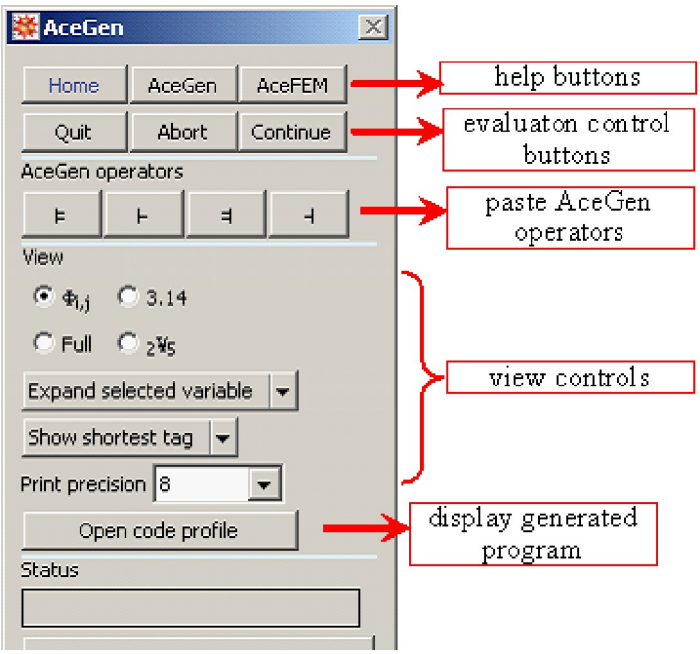
Since *AceGen* runs in parallel with *Mathematica* we can use all the capabilities of *Mathematica*. The major algebraic computations which play crucial role in the development of any numerical code are:

- analytical differentiation,
- symbolic evaluation,
- symbolic solution to the system of linear equations,
- symbolic integration,
- symbolic solution to the system of algebraic equations.

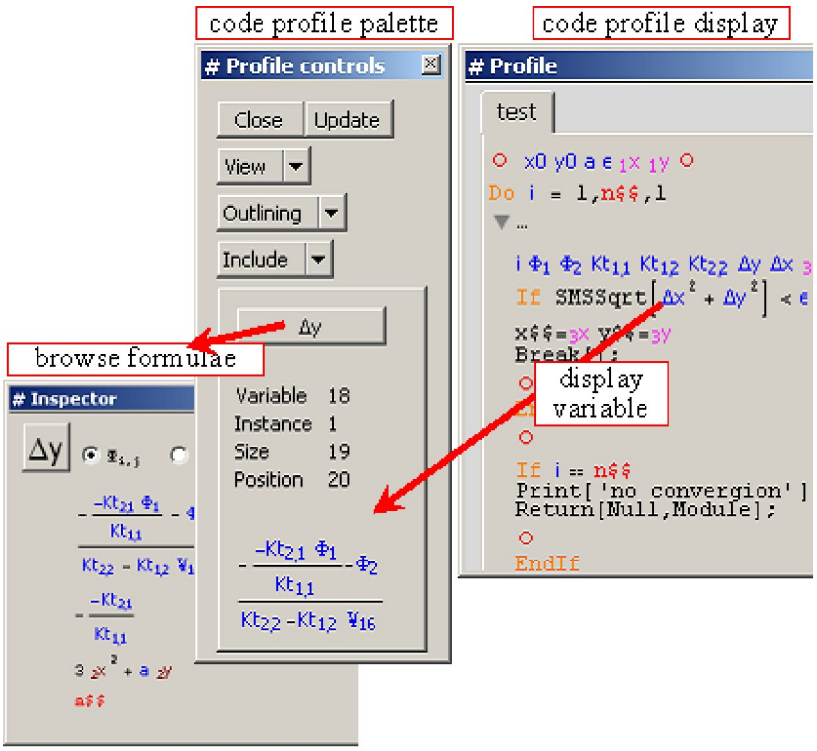
Each of these operations can be directly implemented also with the built-in Mathematica functions and the result optimized by *AceGen*. However, by using equivalent functions in *AceGen* with simultaneous optimization of expressions, much larger problems can be efficiently treated. Unfortunately, the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

AceGen Palettes

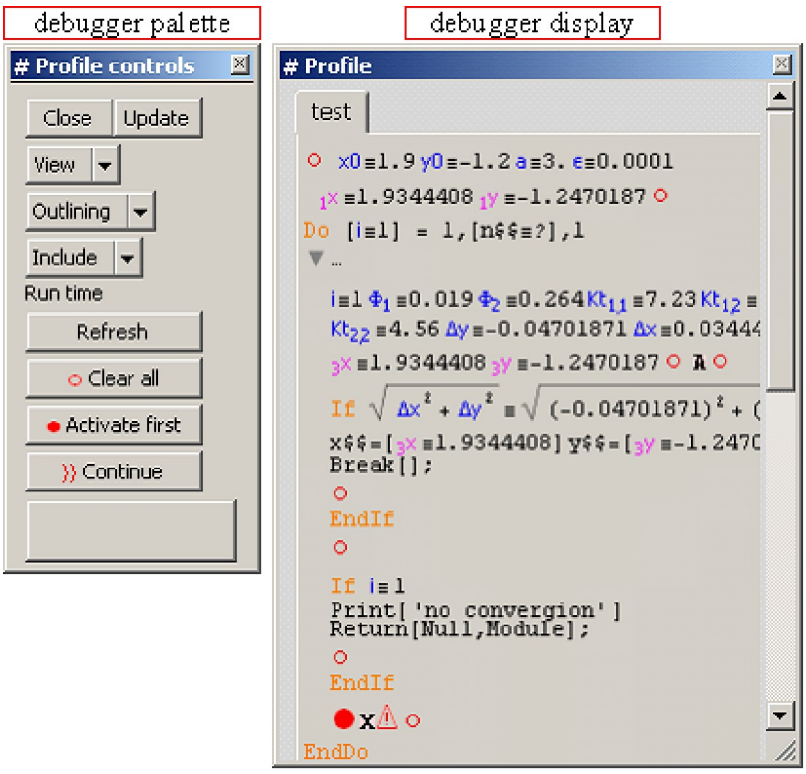
- Main AceGen palette.



- Display generated program.



- Debugger palette and display.



AceGen Overview

Installation instructions

Installation instructions

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

SMSD . 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 . SMSWhich — 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 . SMSLUSolve . SMSFactorSim . SMSInverse . SMSDet . SMSKrammer — linear algebra functions

SMSCovariantBase . SMSCovariantMetric . SMSContravariantMetric . SMSChristoffell1 . SMSChristoffell2 . 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

Standard AceGen Procedure . Solution to the System of Nonlinear Equations . Minimization of Free Energy

Finite element environments

Template Constants

Element Topology

Numerical Integration

Data Structures

Standard User Subroutines

CHAPTER 2

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.

	step		example
1	Initialization	⟹	SMSInitialize["test","Language"->"C"]
2	Definition of input and output parameters	⟹	SMSModule["Test",Real[u$$[3],x$$,L$$,g$$[3]]];
3	Definition of numeric-symbolic interface variables	⟹	{x,L}⊢{SMSReal[x$$],SMSReal[L$$]}; ui⊢SMSReal[Table[u$$[i],{i,3}]];
4	Derivation of the problem	⟹	Ni⊢$\{\frac{x}{L},1-\frac{x}{L},\frac{x}{L}(1-\frac{x}{L})\}$; u⊢Ni.ui; f⊢$u^2$; g⊢SMSD[f,ui];
5	Definition of symbolic-numeric interface variables	⟹	SMSExport[g,g$$];
6	Code generation	⟹	SMSWrite[];

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.

In[1]:=
```
<<AceGen`
```

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

In[2]:=
```
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.

In[3]:=
```
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 ⊨. Operator ⊨ performs stochastic simultaneous optimization of expressions. See also Auxiliary Variables, `SMSReal`.

In[4]:=
```
x ⊢ SMSReal[x$$]
```

Out[ ]=
```
x
```

In[5]:=
```
L ⊢ SMSReal[L$$]
```

Out[ ]=
```
L
```

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

In[6]:=
```
ui ⊢ SMSReal[Table[u$$[i] , {i, 3}]]
```
Out[•]=

$\{ui_1, ui_2, ui_3\}$

Step 4: Description of the Problem

- Here is the body of the subroutine.

In[7]:=
```
Ni ⊨ {x / L, 1 - x / L, x / L * (1 - x / L)}
```
Out[•]=

$\{Ni_1, Ni_2, Ni_3\}$

In[8]:=
```
u ⊨ Ni . ui
```
Out[•]=

u

In[9]:=
```
f ⊨ u^2
```
Out[•]=

f

In[10]:=
```
g ⊨ SMSD[f, ui]
```
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`.

In[11]:=
```
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`.

In[12]:=
```
SMSWrite[];
```

File:	test.f	Size: 988
Methods	No.Formulae	No.Leafs
Test	6	81

- This displays the contents of the generated file.

In[13]:=

```
FilePrint["test.f"]
```

```
!*************************************************************
!* AceGen    5.001 Windows (3 Jan 13)                        *
!*           Co. J. Korelc  2007            3 Jan 13 14:52:07 *
!*************************************************************
! User     : USER
! Notebook : AceGenTutorials.nb
! Evaluation time                 : 0 s     Mode  : Optimal
! Number of formulae              : 6       Method: Automatic
! Subroutine                      : Test size :81
! Total size of Mathematica  code : 81 subexpressions
! Total size of Fortran code      : 384 bytes

!******************* S U B R O U T I N E *********************
      SUBROUTINE Test(v,u,x,L,g)
      IMPLICIT NONE
      include 'sms.h'
      DOUBLE PRECISION v(5001),u(3),x,L,g(3)
      v(15)=x/L
      v(7)=1d0-v(15)
      v(8)=v(15)*v(7)
      v(9)=u(1)*v(15)+u(2)*v(7)+u(3)*v(8)
      v(16)=2d0*v(9)
      g(1)=v(15)*v(16)
      g(2)=v(16)*v(7)
      g(3)=v(16)*v(8)
      END
```

In[ ]:=

Mathematica Syntax - AceGen Syntax

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⊨rhs	Evaluates and optimizes rhs and assigns the result to be the value of lhs (see also: Auxiliary Variables).
lhs ⫤ rhs1	Evaluates and optimizes rhs1 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 ⫣ rhs2	A new value rhs2 is assigned to the previously created variable lhs (see also: Auxiliary Variables).
lhs⊨SMSIf[condition, t, f]	Creates code that evaluates t if condition evaluates to True, and f if it evaluates to False. The value assigned to lhs during the AceGen session represents both options (see also: Program Flow Control, `SMSIf`).
lhs⊨SMSWhich[$test_1$, $value_1$,$test_2$,$value_2$,…]	Creates code that evaluates each of the $test_i$ in turn, returning the value of the $value_i$ 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, `SMSSwitch`).
lhs⊨ SMSSwitch[expr,$form_1$, $value_1$,$form_2$,$value_2$,…]	Creates code that evaluates expr, then compares it with each of the $form_i$ in turn, evaluating and returning the $value_i$ corresponding to the first match found. The value assigned to lhs during the AceGen session represents all options (see also: Program Flow Control, `SMSWhich`).
SMSDo[expr, {i, i_{min}, i_{max}, Δi}]	Creates code that evaluates expr with the variable i successively taking on the values i_{min} through i_{max} in steps of di (see also: Program Flow Control, `SMSDo`).
lhs⫤lhs_0 SMSDo[lhs⫣func[lhs] ,{i, i_{min}, i_{max}, Δi,lhs}]	An initial value lhs_0 is first assigned to the variable lhs. lhs is in a loop continuously changed. After the loop the variable lhs (during the AceGen session) represents all possible values.

Syntax of the basic AceGen commands.

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). AceGen extends the functionality of Mathematica with the cross-cell form of If and Do control structures as presented in Program Flow Control chapter.

Example 1: Assignments

- Mathematica input

In[66]:=
```
x =.; n =.;
y = Sin[x]^n
```
Out[ ]=
```
Sin[x]^n
```

- AceGen input

In[10]:=
```
<< AceGen`;
SMSInitialize["test", "Language" → "C"];
SMSModule["Test", Real[x$$, y$$], Integer[n$$]];
x ⊢ SMSReal[x$$];
n ⊢ SMSInteger[n$$];
y ⊨ Sin[x]^n;
```

- AceGen code profile

Test

x≡x$$ n≡n$$ y≡Sin[x]^n

In[16]:=

```
SMSExport[y, y$$];
SMSWrite[];
```

```
Method : Test 1 formulae, 13 sub-expressions
[0] File created : test.c Size : 726
```

In[18]:=

```
FilePrint["test.c"]
```

```
/*************************************************************
* AceGen    2.115 Windows (20 Nov 08)                        *
*           Co. J. Korelc  2007            20 Nov 08 00:18:33*
*************************************************************
User : USER
Evaluation time                 : 0 s     Mode  : Optimal
Number of formulae              : 1       Method: Automatic
Subroutine                      : Test size :13
Total size of Mathematica  code : 13 subexpressions
Total size of C code            : 164 bytes*/
#include "sms.h"

/******************* S U B R O U T I N E *********************/
void Test(double v[5001],double (*x),double (*y),int (*n))
{
(*y)=Power(sin((*x)),(int)((*n)));
};
```

Example 2: Conditional statements (If construct)

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

$$z = \mathrm{Sin}(y) + 1$$

- Mathematica input

In[315]:=

```
y = If[x ≥ 0
    , If[x ≥ 7
     , 7
     , x
    ]
    , x^2
   ];
z = Sin[y] + 1
```

Out[•]=

```
1 + Sin[If[$V[1, 1] ≥ 0, If[x ≥ 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

Test

```
x ≡ x$$
If x ≥ 0
 If x ≥ 7
  _1v ≡ 7
 Else
  _2v ≡ x
 EndIf
 _1v ≡ _3v
Else
 _2v ≡ x^2
EndIf
z ≡ 1 + Sin[_3v]
```

In[323]:=

```
SMSExport[z, z$$];
SMSWrite[];
FilePrint["test.c"]
```

```
 File:     test.c    Size: 839
Methods No.Formulae No.Leafs
 Test        5         22

/*************************************************************
* AceGen    2.502 Windows (18 Nov 10)                        *
*           Co. J. Korelc  2007            24 Nov 10 13:33:36*
*************************************************************
User : USER
Evaluation time                 : 0 s     Mode  : Optimal
Number of formulae              : 5       Method: Automatic
Subroutine                      : Test size :22
Total size of Mathematica  code : 22 subexpressions
Total size of C code            : 266 bytes*/
#include "sms.h"

/******************* S U B R O U T I N E *********************/
void Test(double v[5001],double (*x),double (*z))
{
int b2,b3;
if((*x)>=0e0){
 if((*x)>=7e0){
  v[4]=7e0;
 } else {
  v[4]=(*x);
 };
 v[5]=v[4];
} else {
 v[5]=Power((*x),2);
};
(*z)=1e0+sin(v[5]);
};
```

Example 3: Loops (Do construct)

$z(i) = \mathrm{Sin}(x^i)$

$y = \sum_{i=1}^{n} \left(z(i) + \frac{i}{z(i)}\right)^i$

- Mathematica input

NOTE: Upper limit *n* in *Do* can only have specific integer value!

In[345]:=

```
Clear[x]; n = 5;
y = 0;
Do[
 z = Sin[x^i];
 y = y + (z + i/z)^i;
 , {i, 1, n, 1}]
y
```

Out[ ]=

$$\mathrm{Csc}[x] + \mathrm{Sin}[x] + \left(2\,\mathrm{Csc}[x^2] + \mathrm{Sin}[x^2]\right)^2 + \left(3\,\mathrm{Csc}[x^3] + \mathrm{Sin}[x^3]\right)^3 + \left(4\,\mathrm{Csc}[x^4] + \mathrm{Sin}[x^4]\right)^4 + \left(5\,\mathrm{Csc}[x^5] + \mathrm{Sin}[x^5]\right)^5$$

- AceGen input

NOTE: Upper limit *n* in SMSDo can have arbitrary value!

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.

In[330]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "C#"];
SMSModule["Test", Real[x$$, y$$], Integer[n$$]];
x ⊢ SMSReal[x$$];
n ⊢ SMSInteger[n$$];
y ⊣ 0;
SMSDo[
  z ⊨ Sin[x^i];
  y ⊣ y + (z + i/z)^i;
  , {i, 1, n, 1, y}];
```

- AceGen code profile

```
Test
x≡x$$ n≡n$$ ₁y≡0
Do i = 1,[n≡n$$],1
  ₃y≡(i + x)^i + ₂y
EndDo
```

In[337]:=

```
SMSExport[y, y$$];
SMSWrite[];
FilePrint["test.cs"]
```

File:	test.cs	Size: 883
Methods	No.Formulae	No.Leafs
Test	5	41

```
/*************************************************************
* AceGen    2.502 Windows (18 Nov 10)                        *
*           Co. J. Korelc  2007            24 Nov 10 13:33:52*
*************************************************************
User : USER
Evaluation time                 : 0 s     Mode   : Optimal
Number of formulae              : 5       Method: Automatic
Subroutine                      : Test size :41
Total size of Mathematica  code : 41 subexpressions
Total size of C# code            : 265 bytes*/

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

/******************* S U B R O U T I N E *********************/
void Test(ref double[] v,ref double x,ref double y,ref int n)
{
i2=(int)(n);
v[3]=0e0;
for(i4=1;i4<=i2;i4++){
 v[5]=Math.Sin(Power(x,i4));
 v[3]=v[3]+Power(i4/v[5]+v[5],i4);
};/* end for */
y=v[3];
}
```

Example 4: Conditional statements (Which construct)

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

$z = \mathrm{Sin}[y] + 1$

- Mathematica input

In[342]:=

```
Clear[x];
y = Which[x ≥ 0 && x ≥ 7, 7, x ≥ 0 && x < 7, x, x < 0, x^2]
z = Sin[y] + 1
```

Out[•]=

```
Which[x ≥ 0 && x ≥ 7, 7, x ≥ 0 && x < 7, x, x < 0, x^2]
```

Out[•]=

```
1 + Sin[Which[x ≥ 0 && x ≥ 7, 7, x ≥ 0 && x < 7, x, x < 0, x^2]]
```

- AceGen input

In[358]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "Fortran"];
SMSModule["Test", Real[x$$, z$$]];
x ⊢ SMSReal[x$$];
y ⊢ SMSWhich[x ≥ 0 && x ≥ 7, 7, x ≥ 0 && x < 7, x, x < 0, x^2];
z ⊢ Sin[y] + 1;
```

- AceGen code profile

Test

```
x ≡ x$$ 1bsw ≡ True
If x ≥ 0 && x ≥ 7
  2bsw ≡ False 1v ≡ 7
EndIf
If 3bsw && x ≥ 0 && x < 7
  4bsw ≡ False 3v ≡ x
EndIf
If 5bsw && x < 0
  6bsw ≡ False 5v ≡ x^2
EndIf
z ≡ 1 + Sin[6v]
```

In[364]:=

```
SMSExport[z, z$$];
SMSWrite[];
FilePrint["test.f"]
```

File:	test.f	Size: 1105
Methods	No.Formulae	No.Leafs
Test	8	35

```
!*************************************************************
!* AceGen    2.502 Windows (18 Nov 10)                       *
!*           Co. J. Korelc  2007            24 Nov 10 13:34:40*
!*************************************************************
! User : USER
! Evaluation time                 : 0 s     Mode   : Optimal
! Number of formulae              : 8       Method: Automatic
! Subroutine                      : Test size :35
! Total size of Mathematica  code : 35 subexpressions
! Total size of Fortran code      : 528 bytes

!******************** S U B R O U T I N E **********************
      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	default	description
"Language"	"Mathematica"	source code language (see Installation instructions for instructions how to compile and link the generated codes)
"Environment"	"None"	is a character constant that identifies the numerical environment for which the code is generated
"VectorLength"	500	length of the system vectors (very large system vectors can considerably slow down execution)
"Mode"	"Optimal"	define initial settings for the options of the *AceGen* functions
"GlobalNames"	{"v","i","b"}	first letter of the automatically generated auxiliary real, integer, and logical type variables
"SubroutineName"	#&	pure function applied on the names of all generated subroutines
"Debug"	for "Mode": "Debug"⇒True "Prototype"⇒False "Optimal"⇒False	if True extra (time consuming) tests of code correctness are performed during derivation of formulas and also included into generated source code
"Precision"	100	default precision of the `Signatures of Expressions`

Options of the *SMSInitialize* function.

Language	description	Generic name
"Fortran"	fixed form FORTRAN 77 code	"Fortran"
"Fortran90"	free form FORTRAN 90 code	"Fortran"
"Mathematica"	code written in Mathematica programming language	"Mathematica"
"C"	ANSI C code	"C"
"C++"	ANSI C++ code	"C"
"Matlab"	standard Matlab "M" file	"Matlab"

Supported languages.

mode	
"Plain"	all `Expression Optimization` procedures are excluded
"Debug"	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
"Prototype"	options are set for the fastest derivation of the code, with moderate level of code optimization
"Optimal"	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)

Supported optimization modes.

environment	Language	description
"None"	defined by "Language" option	plain code
"MathLink"	"C"	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`)
"User"	defined by "Language" option	arbitrary user defined finite element environment (see Standard FE Procedure, User Defined Environment Interface)
"AceFEM"	"C"	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)
"AceFEM–MDriver"	"Mathematica"	*AceFEM* finite element environment with symbolic numerical module (elements and all procedures written entirely in Mathematica' s programming language) (see Standard FE Procedure, AceFEM Structure)
"FEAP"	"Fortran"	research finite element environment written in *FORTRAN* (see `FEAP – ELFEN – ABAQUS – ANSYS`)
"ELFEN"	"Fortran"	commercial finite element environment written in *FORTRAN* (see `FEAP – ELFEN – ABAQUS – ANSYS`)
"ABAQUS"	"Fortran"	commercial finite element environment written in *FORTRAN* (see `FEAP – ELFEN – ABAQUS – ANSYS`)

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.

In[2]:=
```
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	description
Real[p_1,p_2,...]	list of real type parameters
Integer[p_1,p_2,...]	list of integer type parameters
Logical[p_1,p_2,...]	list of logical type parameters
typename[p_1,p_2,...]	list of the user defined type *typename* parameters
Automatic[p_1,p_2,...]	list of parameters for which type is not defined (only allowed for interpreters like Mathematica and Matlab)

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.

option	default	description
"Verbatim"->*Start*	None	string or a list of strings *Start* is included at the end of the declaration block of the source code verbatim
"VerbatimEnd"->*End*	None	string or a list of strings *End* is included at the end of the subroutine code verbatim
"Input"	All	list of input parameters
"Output"	All	list of output parameters

Options of the *SMSModule* function.

substitution	character
'	"
[/	\
/'	'
[/'	\"
[/n	\n

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’ .

In[455]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "Fortran"];
SMSModule["sub1", Real[x$$, y$$[5]], Integer[i$$], Real[z$$], "mytype"[m$$]
  , "Verbatim" → "COMMON /xxx/a(5)\nwrite(*,*)'start'"
  , "VerbatimEnd" → "write(*,*)'end'"];
SMSExport[π, x$$];
SMSWrite[];
FilePrint["test.f"]
```

File: test.f **Size:** 946 **Time:** 0

Method	sub1
No.Formulae	1
No.Leafs	7

```
!*************************************************************
!* AceGen    6.922 Windows (16 Feb 19)                       *
!*           Co. J. Korelc  2013           16 Feb 19 19:29:58 *
!*************************************************************
! User     : Full professional version
! Notebook : AceGenTutorials
! 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      : 328 bytes

!******************* S U B R O U T I N E **********************
      SUBROUTINE sub1(v,x,y,i,z,m)
      IMPLICIT NONE
      include 'sms.h'
      INTEGER i
      DOUBLE PRECISION v(111),x,y(5),z
      TYPE (mytype)::m
      COMMON /xxx/a(5)
      write(*,*)"start"
      x=0.3141592653589793d1
      write(*,*)"end"
      END
```

Example 2

Create a C subroutine that calulactes series $r_{ij} = \sum_{s=1}^{k} x^{s}_{ij}$, i = 1, ... n; j = 1, ... m.

- Solution 1: elements of the series are calculated and added to globaly defined r_{ij} directly. r_{ij} is alloocated on heap.

In[481]:=

```
<< AceGen`;
SMSInitialize["test", "Environment" → "MathLink"];
SMSModule["test", Real[x$$[n$$, m$$]], Integer[n$$, m$$, k$$],
  Real[r$$[n$$, m$$]], "Input" → {x$$, n$$, m$$, k$$}, "Output" → r$$];
{n, m, k} ⊨ SMSInteger[{n$$, m$$, k$$}];
SMSDo[
  SMSDo[
    SMSDo[
      SMSExport[SMSPower[SMSReal[x$$[i, j]], s], r$$[i, j], "AddIn" → True]
      , {s, 1, k}];
    , {i, 1, n}];
  , {j, 1, m}];
SMSWrite[];
```

File: test.c Size: 2538 Time: 0

Method	test
No.Formulae	2
No.Leafs	38

In[487]:=

```
SMSInstallMathLink[];
```

In[468]:=

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

Out[ ]=

138.131

In[470]:=

```
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!

In[471]:=

```
<< AceGen`;
SMSInitialize["test", "Environment" → "MathLink"];
SMSModule["test", Real[x$$[n$$, m$$]], Integer[n$$, m$$, k$$],
  Real[r$$[n$$, m$$]], "Input" → {x$$, n$$, m$$, k$$}, "Output" → r$$
  , "Verbatim" → "double locr[*n][*m];\nint
     ii,jj;\nfor(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];"
 ];
{n, m, k} ⊨ SMSInteger[{n$$, m$$, k$$}];
SMSDo[
  SMSDo[
    SMSDo[
      SMSExport[SMSPower[SMSReal[x$$[i, j]], s], locr$$[i, j], "AddIn" → True]
      , {s, 1, k}];
    , {i, 1, n}];
  , {j, 1, m}];
SMSWrite[];
```

File: test.c Size: 2711 Time: 0

Method	test
No.Formulae	2
No.Leafs	38

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[ ]=
```
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.

option	default	description
"Splice"	{}	list of files interpreted (see FileTemplate) and prepended to the generated source code file (in the case of standard numerical invironment a special interface file is added to the list automatically)
"Substitutions"	{}	list of rules applied on all expressions before the code is generated (see also User Defined Functions)
"IncludeNames"	False	the name of the auxiliary variable is printed as a comment before definition
"IncludeAllFormulas"	False	also the formulae that have no effect on the output parameters of the generated subroutines are printed
"OptimizingLoops"	1	number of additional optimization loops over the whole code
"IncludeHeaders"	{}	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: "Fortran" ⇒ {"sms.h"} "Fortran90" ⇒ {"SMS"} "Mathematica" ⇒ {} "C" ⇒ {"sms.h"} "C++" ⇒ {"sms.h"}
"MaxLeafCount"	3000	large Fortran expressions are split into subexpressions of the size less than "MaxLeafCount" due to the limitations of Fortran compilers (size is measured by the LeafCount function)
"LocalAuxiliaryVariables"	False	True ⇒ The vector of auxiliary variables is defined locally for each module. False ⇒ The vector of auxiliary variables is to be an argument of the subroutines, e.g. double *v. Automatic ⇒ For standard numerical environments the vector of auxiliary variables is defined locally for each module if length of working vector is less than 25 000. _Integer ⇒ 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.
"WorkingVectorSize"	Automatic	The length of vector where AceGen generated auxiliary variables are stored (CDriver, ABAQUS, etc.). In some cases (e.g. for backward differentiation of complex program structures or calling the external subroutines) significant storage might be required. Default value is set by FE environment. E.g. CDriver sets it to 100 000. (see also General Data). Automatic ⇒ only possible when there are no variable length arrays present *i*_Integer ⇒ set by user *f*_Function ⇒ AceGen make an estimate how large the vector should be for all generated subroutines. Function *f*[list of estimates] is appled on that list to get the actual WorkingVectorSize. E.g. if an input data has constants "m", "n" and we know that they cannot be larger than 10 then we set "WorkingVectorSize"-> Function[{lenghts},Max[lenghts/.{"m"→10,"n"→10}]].

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 `Installation instructions` for instructions how to compile and link the generated codes.

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.

In[6]:=
```
<<AceGen`;

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

In[10]:=
```
FilePrint["test.mc"]
```

```
/*This is a "splice" file <*100+1*> */
```

In[11]:=
```
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[i_,j_]:>"mybessel"[i,j]}];
```

```
Method : sub1 1 formulae, 13 sub-expressions
[0] File created : source.c Size : 742
```

In[15]:=
```
FilePrint["source.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                      : sub1 size :13
Total size of Mathematica  code : 13 subexpressions
Total size of C code            : 146 bytes*/
#include "sms.h"
/*This is a "splice" file 101 */

/******************* 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[$language_1$->$source_1$,$language_2$->$source_2$,...]
write textual form of the *source* which corresponds to the currently used program language into the automatically generated file verbatim

option	default	description
"Close"	True	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.
"DeclareSymbol"	{}	List of symbols used within the code that have to have declared data type (e.g "DeclareSymbol"→{Real[x$$,y$$],Integer[n$$]}).
"AddStructure"	End	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.

Options of the *SMSVerbatim* function.

Input parameters *source, source*$_1$, *source*$_2$,... 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.

substitution	character
'	"
[/	\
/'	'
[/'	\"
[/n	\n

Character substitution table.

Character substitution table.

The parameter "language" can be any of the languages supported by *AceGen* (''Mathematica'', ''Fortran'', "Fortan90", "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 differentiation procedure unless additional declarations are included as presented in `User Defined Functions`.

In[16]:=

```
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"

/******************* S U B R O U T I N E *********************/
void test(double v[5001])
{
printf("Hello");
exit(0);
};
```

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$$], SMSReal[L$$]};
ui ⊢ SMSReal[Table[u$$[i], {i, 3}]]
Ni ⊨ {x/L, 1 - x/L, x/L (1 - x/L)};
u ⊨ Ni.ui;
f ⊨ u^2;
g ⊨ SMSD[f, ui];
SMSExport[g, g$$];
SMSWrite[];
```

```
Method : Test 6 formulae, 81 sub-expressions
[0] File created : test.c Size : 863
```

In[25]:=

```
FilePrint["test.c"]
```

```
/*************************************************************
* AceGen    2.103 Windows (17 Jul 08)                        *
*           Co. J. Korelc  2007            17 Jul 08 13:04:01*
*************************************************************
User : USER
Evaluation time                 : 0 s     Mode  : Optimal
Number of formulae              : 6       Method: Automatic
Subroutine                      : Test size :81
Total size of Mathematica  code : 81 subexpressions
Total size of C code            : 294 bytes*/
#include "sms.h"

/******************* S U B R O U T I N E *********************/
void Test(double v[5001],double u[3],double (*x),double (*L),double g[3])
{
v[6]=(*x)/(*L);
v[7]=1e0-v[6];
v[8]=v[6]*v[7];
v[9]=u[0]*v[6]+u[1]*v[7]+u[2]*v[8];
v[15]=2e0*v[9];
g[0]=v[15]*v[6];
g[1]=v[15]*v[7];
g[2]=v[15]*v[8];
};
```

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.

In[26]:=

```
<< AceGen`;
SMSInitialize["test", "Environment" → "MathLink"];
SMSModule["Test", Real[u$$[3], x$$, L$$, g$$[3]],
  "Input" → {u$$, x$$, L$$}, "Output" → g$$];
{x, L} ⊢ {SMSReal[x$$], SMSReal[L$$]};
ui ⊢ SMSReal[Table[u$$[i], {i, 3}]]
Ni ⊨ {x/L, 1 - x/L, x/L (1 - x/L)};
u ⊨ Ni.ui;
f ⊨ u^2;
g ⊨ SMSD[f, ui];
SMSExport[g, g$$];
SMSWrite[];
```

```
Method : Test 6 formulae, 81 sub-expressions
[0] File created : test.c Size : 1787
```

In[37]:=

```
FilePrint["test.c"]
```

```
/*************************************************************
* AceGen    2.103 Windows (17 Jul 08)                        *
*           Co. J. Korelc  2007            17 Jul 08 13:04:03*
*************************************************************
User : USER
Evaluation time                 : 0 s     Mode  : Optimal
Number of formulae              : 6       Method: Automatic
Subroutine                      : Test size :81
Total size of Mathematica  code : 81 subexpressions
Total size of C code            : 294 bytes*/
```

```
#include "sms.h"
#include "stdlib.h"
#include "stdio.h"
#include "mathlink.h"
double workingvector[5101];
void Test(double v[5001],double u[3],double (*x),double (*L),double g[3]);



void TestMathLink(){
int i1000,i1001,i1002,i1003,i1004,i1j1,i4j1,i1s1,i4s1;
char *b1; double *b2;int *b3;
double u[3];
double *x;
double *L;
double g[3];
++MathLinkCallCount[0];

/* read from link */
MLGetRealList(stdlink,&b2,&i1j1);
for(i1j1=0;i1j1<3;i1j1++){
  u[i1j1]=b2[i1j1];
}
MLDisownRealList(stdlink,b2,3);
x=(double*) calloc(1,sizeof(double));
MLGetReal(stdlink,x);
L=(double*) calloc(1,sizeof(double));
MLGetReal(stdlink,L);

/* allocate output parameters */
i1s1=3;
i4s1=3;

/* call module */
Test(workingvector,u,x,L,g);

/* write to link */
free(x);
free(L);
PutRealList(g,i4s1);
};

void MathLinkInitialize()
{
  MathLinkOptions[CO_NoSubroutines]=1;
  printf("MathLink module: %s\n","test");
};


/******************* S U B R O U T I N E *********************/
void Test(double v[5001],double u[3],double (*x),double (*L),double g[3])
{
v[6]=(*x)/(*L);
v[7]=1e0-v[6];
v[8]=v[6]*v[7];
v[9]=u[0]*v[6]+u[1]*v[7]+u[2]*v[8];
v[15]=2e0*v[9];
g[0]=v[15]*v[6];
g[1]=v[15]*v[7];
g[2]=v[15]*v[8];
};
```

- 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`).

In[38]:=

```
SMSInstallMathLink[]
```

Out[ ]=

```
{SMSSetLinkOption[test, {i_Integer, j_Integer}], SMSLinkNoEvaluations[test],
 Test[u_?(ArrayQ[#1, 1, NumberQ] && Dimensions[#1] === {3} &), x_?NumberQ, L_?NumberQ]}
```

- Here the generated executable is used to calculate gradient for the numerical test example.

In[39]:=

```
Test[{0., 1., 7.}, π // N, 10.]
```

Out[ ]=

```
{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.

In[40]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "Matlab"];
SMSModule["Test", Real[u$$[3], x$$, L$$, g$$[3]],
  "Input" → {u$$, x$$, L$$}, "Output" → g$$];
{x, L} ⊢ {SMSReal[x$$], SMSReal[L$$]};
ui ⊢ SMSReal[Table[u$$[i], {i, 3}]]
Ni ⊨ {x/L, 1 - x/L, x/L (1 - x/L)};
u ⊨ Ni.ui;
f ⊨ u^2;
g ⊨ SMSD[f, ui];
SMSExport[g, g$$];
SMSWrite[];
```

```
Method : Test 6 formulae, 81 sub-expressions
[0] File created : test.m Size : 1084
```

In[51]:=

```
FilePrint["test.m"]
```

```
%*************************************************************
%* AceGen    2.103 Windows (17 Jul 08)                       *
%*           Co. J. Korelc  2007            17 Jul 08 13:04:06*
%*************************************************************
% User : USER
% Evaluation time                 : 0 s     Mode  : Optimal
% Number of formulae              : 6       Method: Automatic
% Subroutine                      : Test size :81
% Total size of Mathematica  code : 81 subexpressions
% Total size of Matlab code      : 299 bytes

%************************ F U N C T I O N **************************
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);

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

$$\Phi = \begin{Bmatrix} a\,x\,y + x^3 = 0 \\ a - x\,y^2 = 0 \end{Bmatrix}$$

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$$, tol$$], Integer[nmax$$]];
{x0, y0, a, ϵ} ⊢ SMSReal[{x$$, y$$, a$$, tol$$}];
nmax ⊢ SMSInteger[nmax$$];
{x, y} ⫤ {x0, y0};
SMSDo[
  Φ ⊨ {a x y + x^3, a - x y^2};
  Kt ⊨ SMSD[Φ, {x, y}];
  {Δx, Δy} ⊨ SMSLinearSolve[Kt, -Φ];
  {x, y} ⫤ {x, y} + {Δx, Δy};
  SMSIf[SMSSqrt[{Δx, Δy}.{Δx, Δy}] < ϵ
   , SMSExport[{x, y}, {x$$, y$$}];
   SMSBreak[];
  ];
  SMSIf[i == nmax
   , SMSPrintMessage["no convergion"];
   SMSReturn[];
  ];
  , {i, 1, nmax, 1, {x, y}}
 ];
SMSWrite[];
```

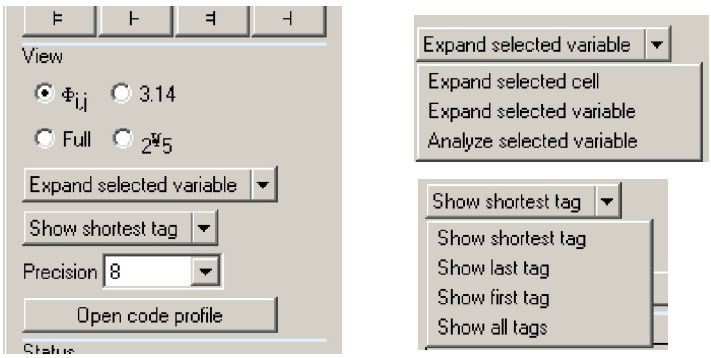
```
time=0  variable= 0 ≡ {}

[0]  Consistency check - global

[0]  Consistency check - expressions

[0]  Generate source code :

Events: 0

[0] Final formating
```

File:	test.m	Size: 2401
Methods	No.Formulae	No.Leafs
test	31	194

Presentation of Formulas

Exploring the structure of the formula - Browser mode submenu

AceGen palette offers buttons that control how expressions are represented on a screen.



Palette for entering *AceGen* commands that control user-*AceGen* interactions.

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 $\Phi_{1;x} = \frac{\partial \Phi_1}{\partial x}$).

In[112]:=

```
Kt[[1, 1]]
```

Out[•]=

$Kt_{1,1}$

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[•]=

$$\{\{Kt_{1,1}, Kt_{1,2}\}, \{-Kt_{2,1}, Kt_{2,2}\}\}$$

$$3\,{}_{2}x^{2} + a\,{}_{2}y$$

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[•]=

$$\{\{Kt_{1,1}, Kt_{1,2}\}, \{-Kt_{2,1}, Kt_{2,2}\}\}$$

$$\{\{3\,{}_{2}x^{2} + a\,{}_{2}y, Kt_{1,2}\}, \{-Kt_{2,1}, Kt_{2,2}\}\}$$

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

Button: Analyze selected variable

In[115]:=
```
Kt
```
Out[•]=

$\{\{Kt_{1,1}, Kt_{1,2}\}, \{-Kt_{2,1}, Kt_{2,2}\}\}$

Variable=$V[13, 1] Tags= $Kt_{1,1}$ | $\Phi_{1;x}$

Definition= $3\,{}_{2}x^{2} + a\,{}_{2}y$

Position in program={1, 2, 10, 2, 7} Position in data base=15 Module=1

Skope={Do[i, 1, nmax, 1]}

Type=Real Singlevalued=True No. of instances=1 Signature=0.756549

Defined derivatives={}

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.

Button: $\Phi_{i,j}$

In[116]:=
```
Kt
```
Out[•]=

$\{\{Kt_{1,1}, Kt_{1,2}\}, \{-Kt_{2,1}, Kt_{2,2}\}\}$

FullForm

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

Button: Full

In[117]:=
```
Kt
```
Out[•]=
```
{{$V[13, 1], $V[15, 1]}, {-$V[14, 1], $V[16, 1]}}
```

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.

Button: 2¥5

In[118]:=
```
Kt
```
Out[•]=

$\{\{¥_{13}, ¥_{15}\}, \{-¥_{14}, ¥_{16}\}\}$

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.

3.14

Button: 3.14

In[119]:=
```
Kt
```
Out[•]=
```
{{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
```
Out[•]=

$$\{\{Kt_{1,1}, Kt_{1,2}\}, \{-\,-Kt_{2,1}, Kt_{2,2}\}\}$$

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

In[122]:=
```
Kt
```
Out[•]=

$$\{\{Kt_{1,1}, Kt_{1,2}\}, \{-\,-\Phi_{2,x}, Kt_{2,2}\}\}$$

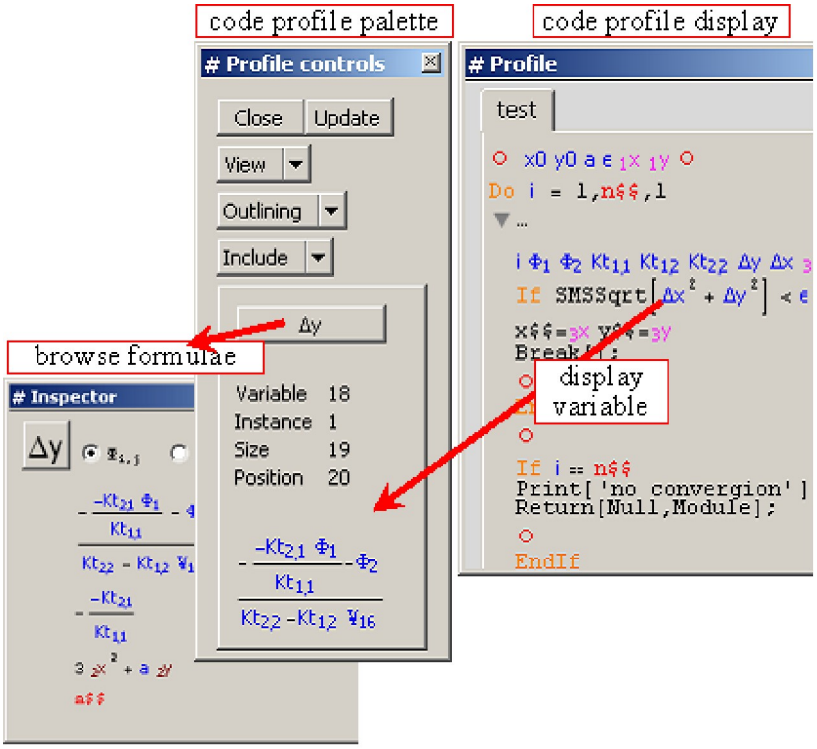
All meanings (names) of the auxiliary variables can also be explored (Show all tags).

In[124]:=
```
Kt
```
Out[•]=

$$\{\{Kt_{1,1} \mid \Phi_{1,x}, Kt_{1,2} \mid \Phi_{1,y}\}, \{-\,-\Phi_{2,x} \mid -Kt_{2,1}, Kt_{2,2} \mid \Phi_{2,y}\}\}$$

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".

KORELC, Jože, Multi-language and Multi-environment Generation of Nonlinear Finite Element Codes, Engineering with Computers, 2002, 18(4):312-327

KORELC, Jože. Automatic generation of finite-element code by simultaneous optimization of expressions. Theor. comput. sci.. , 1997, 187:231-248.

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:

	description	simplification
(a)	two expressions or sub–expressions are the same	$e_1 \equiv e_2 \Longrightarrow \begin{cases} v_1 := e_1 \\ e_2 \Rightarrow v_1 \end{cases}$
(b)	result is an integer value	$e_1 \equiv Z \Longrightarrow \quad e_1 \Rightarrow Z$
(c)	opposite value	$e_1 \equiv -e_2 \Longrightarrow \begin{cases} v_1 := e_1 \\ e_2 \Rightarrow -v_1 \end{cases}$
(d)	intersection of common parts for multiplication and addition	$\begin{matrix} a_{1\cdots i} \circ b_{1\cdots j} \\ c_{1\cdots k} \circ d_{1\cdots j} \\ b_n \equiv d_n \end{matrix} \Longrightarrow \begin{matrix} v_1 := b_{1\cdots j} \\ a_{1\cdots i} \circ b_{1\cdots j} \Rightarrow a_{1\cdots i} \circ v_1 \\ c_{1\cdots k} \circ d_{1\cdots j} \Rightarrow c_{1\cdots k} \circ v_1 \end{matrix}$
(e)	inverse value	$e_1 \equiv \frac{1}{e_2} \Longrightarrow \begin{cases} v_1 := e_2 \\ e_1 \Rightarrow \frac{1}{v_1} \end{cases}$

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.

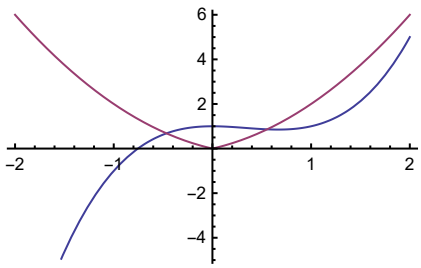
Sub-expressions in the above cases do not need to be syntactically identical, which means that higher relations are recognized also in cases where term rewriting and pattern matching algorithms in *Mathematica* fail. The disadvantage of the procedure is that the code is generated correctly only with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

Probability of incorrect simplifications

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

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

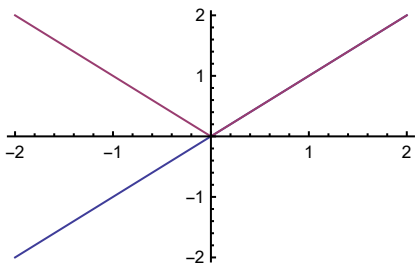
Out[ ]=



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}]
```

Out[ ]=



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.

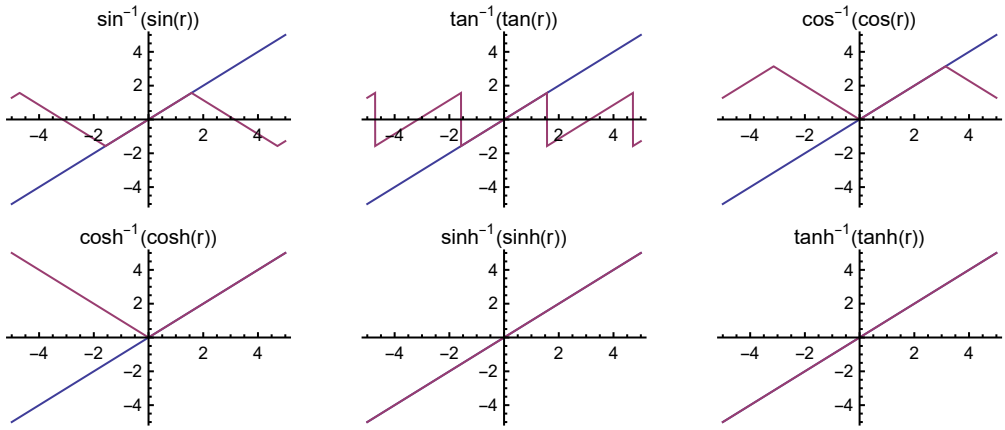
A typical situation when probability to make incorrect simplification is always high is when a non-monotonic function and its inverse counterpart together appear as a part of the problem description.

- 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]]]
   }}
]
```

Out[ ]=



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:

SMSAbs[*exp*]	absolute value of *exp*
SMSSign[*exp*]	`-1`, `0` or `1` depending on whether *exp* is negative, zero, or positive
SMSKroneckerDelta[*i*, *j*]	1 or 0 depending on whether *i* is equal to *j* or not
SMSSqrt[*exp*]	square root of *exp*
SMSMin[*exp1*,*exp2*]	≡ Min[*exp1*,*exp2*]
SMSMax[*exp1*,*exp2*]	≡ Max[*exp1*,*exp2*]

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 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*.

In[213]:=
```
r ⊢ SMSReal[r$$];
r // SMSEvaluate
```

Out[•]=
```
0.62196351
```

- This input leads to incorrect simplification of Sqrt[x^2] expression.

In[215]:=
```
ϕ ⊨ Sqrt[r^2];
SMSRestore[x]
```

```
Function Sqrt is potentialy unsafe. See also: Expression Optimization
```

Out[•]=
```
r$$
```

- This input prevents incorrect simplification.

In[217]:=
```
ϕ ⊨ SMSSqrt[r^2];
SMSRestore[ϕ]
```

Out[•]=
```
SMSSqrt[ϕ^2]
```

- This input leads to incorrect simplification of ArcCos[Cos[r]] expression.

In[219]:=
```
ϕ ⊨ ArcCos[Cos[r]];
SMSRestore[ϕ]
```

```
Function ArcCos is potentialy unsafe. See also: Expression Optimization
```

Out[•]=
```
r$$
```

- This is incorrect since the simplification of ArcCos[Cos[r]] is not prevented yet.

In[221]:=
```
ϕ ⊨ SMSFreeze[ArcCos[Cos[r]], "Code" → True];
SMSRestore[ϕ]
```

```
Function ArcCos is potentialy unsafe. See also: Expression Optimization
```

Out[•]=
```
Freeze[r]
```

- This input represents correct formulation.

In[223]:=
```
ϕ ⊨ ArcCos[SMSFreeze[Cos[r], "Code" → True]];
SMSRestore[ϕ]
```

```
Function ArcCos is potentialy unsafe. See also: Expression Optimization
```

Out[•]=
```
ArcCos[Freeze[Cos[r]]]
```

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.

Original	*Simplification A*	*Simplification B*
$x := L(a)$	$x := L(a)$	$v_1 := L(a)$
$y := L(a)+x^2$	$y := x+x^2$	$x := v_1$
$\frac{dy}{dx}=2\,x$	$\frac{dy}{dx}=1+2\,x$	$y := v_1+x^2$
		$\frac{dy}{dx}=2\,x$

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 ⊨,⊢, ⫤, and ⊣. Operators ⊨ and ⊢ 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 ⫤ and ⊣ have to be used. These operators are replacement for the simple assignment command in *Mathematica* (see `Mathematica Syntax – AceGen Syntax`).

v ⊨ 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 formulae. Ordinary Mathematica input can be converted to the AceGen input by replacing the Set operator (a=b) with the ⊨ operator (a⊨b).

v ⊢ 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 ⫤ 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 ⊣ 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 ⫤ exp command. At the output there is the same variable v, but with the new signature (new instance of v).

Syntax of AceGen assignment operators.

If *x* is a symbol with the value *$V[i,j]*, then after the execution of the expression *x⊣exp, x* has a new value *$V[i,j+1]*. The value *$V[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 `SMSFreeze` function imposes various restrictions in how expression is evaluated, simplified and differentiated. The `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 N function on the result (N[*expr*, SMSEvaluatePrecision]). The expression that does not yield a real number as the result of 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 *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 (`SMSIf`, 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 `Expression Optimization`, `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 most 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 *π*, integer variable *i* with value 1, and logical variable *l* with value True.

In[324]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "Fortran", "Mode" → "Debug"];
SMSModule["Test"];
x ⊢ SMSReal[π];
i ⊢ SMSInteger[1];
l ⊢ SMSLogical[True];
SMSWrite[];
```

```
time=0  variable= 0 ≡ {x}

[0]  Consistency check - global

[0]  Consistency check - expressions

[0]  Generate source code :

Method : Test 3 formulae, 13 sub-expressions

Events: 0

[0] Final formating

Export source code.

[0] File created : test.f Size : 860
```

- Intermediate variables are labeled consecutively regardless of the type of variable. This displays how internal variables really look like.

In[331]:=

```
{x, i, l} // ToString
```

Out[ ]=

```
{$V[1, 1], $V[2, 1], $V[3, 1]}
```

- This displays the generated FORTRAN code. *AceGen* translates internal representation of auxiliary variables accordingly to the type of variable as follows:
  x := $V[1, 1] ⇒ v(1)
  i := $V[2, 1] ⇒ i2
  l := $V[3, 1] ⇒ b3

In[332]:=

```
FilePrint["test.f"]
```

```
!*************************************************************
!* AceGen    2.103 Windows (17 Jul 08)                       *
!*           Co. J. Korelc  2007            17 Jul 08 22:29:46*
!*************************************************************
! User : USER
! Evaluation time                 : 0 s     Mode  : Debug
! Number of formulae              : 3       Method: Automatic
! Subroutine                      : Test size :13
! Total size of Mathematica  code : 13 subexpressions
! Total size of Fortran code      : 295 bytes

!******************** S U B R O U T I N E **********************
      SUBROUTINE Test(v)
      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
```

Example: multi-valued variables

- This generates two instances of the same variable x. The first instance has value π and the second instance has value π^2.

In[333]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "Fortran", "Mode" → "Debug"];
SMSModule["Test"];
x ⫤ SMSReal[π];
x ⫤ π^2;
SMSWrite[];
```

```
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 formating

Export source code.

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

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

In[339]:=

```
x // ToString
```

Out[•]=

```
$V[1, 2]
```

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

```
x := $V[1, 1] ⇒ v (1)
x := $V[1, 2] ⇒ v (1)
```

In[340]:=

```
FilePrint["test.f"]
```

```
!*************************************************************
!* AceGen    2.103 Windows (17 Jul 08)                       *
!*           Co. J. Korelc  2007            17 Jul 08 22:29:52*
!*************************************************************
! User : USER
! Evaluation time                 : 0 s     Mode  : Debug
! Number of formulae              : 2       Method: Automatic
! Subroutine                      : Test size :7
! Total size of Mathematica  code : 7 subexpressions
! Total size of Fortran code      : 253 bytes

!******************* S U B R O U T I N E *********************
      SUBROUTINE Test(v)
      IMPLICIT NONE
      include 'sms.h'
      DOUBLE PRECISION v(5001)
! 1 = x
      v(1)=0.3141592653589793d1
! 1 = x
      v(1)=0.9869604401089358d1
      END
```

Auxiliary Variables Functions

SMSR

SMSR[*symbol*,*exp*]
create a new auxiliary variable if introduction of a new variable is necessary, otherwise *symbol*=*exp*

symbol ⊨ *exp*
infix form of the SMSR function is equivalent to the standard form 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].

Extensions of symbol names	$x_$,#2,e::s,etc.
Function application variants	$e[e]$,e@@e,etc.
Power-related operators	$\sqrt{e}$,e^e,etc.
Multiplication-related operators	∇e , e/e , $e\otimes e$, $e\,e$, etc.
Addition-related operators	$e\oplus e$,$e+e$,$e\bigcup e$,etc.
Relational operators	$e==e$,$e\sim e$,$e\ll e$,$e\lhd e$,$e\in e$,etc.
Arrow and vector operators	$e\longrightarrow e$,$e\nearrow e$,$e\rightleftharpoons e$,$e\rightarrow e$,etc.
Logic operators	$\forall_e e$, e&&e , $e\vee e$, $e\vdash e$, etc.
AceGen operators	⊨,⊢,⫤,⊣
Postfix and rule operators	$e//e$, $e/.e$, , etc.
Pure function operator	e&
Assignment operators	$e=e$,$e:=e$,etc.
Compound expression	e;e

Precedence of AceGen operators.

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

In[97]:=

```
SMSInitialize["test", "Language" → "Fortran"];
SMSModule["sub"];
x ⊨ 1
y ⊨ Sin[5]
```

```
1
```

SMSV

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

symbol ⊢ *exp*
an infix form of the SMSR function is equivalent to the standard form SMSV[*symbol*,*exp*]

The SMSV function first evaluates *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 *exp* is then stored into the AceGen database.

Precedence level of ⊢ operator is specified in *Mathematica* precedence table and described in SMSR.

- The new auxiliary variables are created for all expressions.

In[101]:=

```
SMSInitialize["test", "Language" → "Fortran"];
SMSModule["sub"];
x ⊢ 1
y ⊢ Sin[5]
```

x

y

SMSM

SMSM[*symbol*,*exp*]
create a new multi-valued auxiliary variable

symbol ⫤ *exp*
an infix form of the SMSM function is equivalent to the standard form SMSM[*symbol*,*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 *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 exp is then stored into the AceGen database. The new auxiliary variable will not be created if 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 ⫤ operator is described in SMSR.

SMSS

SMSS[*symbol*,*exp*]
a new instance of the previously created multi-valued auxiliary variable is created

symbol ⊣ *exp*
this is an infix form of the SMSS function and is equivalent to the standard form SMSS[*symbol*,*exp*]

At the input the value of the *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 *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 ⊣ operator is described in SMSR.

- Successive use of the ⫤ and ⊣ operators will produce several instances of the same variable *x*.

In[105]:=

```
SMSInitialize["test", "Language" → "Fortran", "Mode" → "Prototype"];
SMSModule["sub", Real[x$$]];
x ⫤ 1
x ⊣ x + 2
x ⊣ 5 x
```

$_1$x

$_2$x

$_3$x

In[110]:=

```
SMSExport[x, x$$];
SMSWrite[];
```

```
Method : sub 4 formulae, 16 sub-expressions
[0] File created : test.f Size : 808
```

In[112]:=

```
FilePrint["test.f"]
```

```
!*************************************************************
!* AceGen    2.103 Windows (18 Jul 08)                       *
!*           Co. J. Korelc  2007            18 Jul 08 16:48:31*
!*************************************************************
! User : USER
! Evaluation time                 : 0 s     Mode  : Prototype
! Number of formulae              : 4       Method: Automatic
! Subroutine                      : sub size :16
! Total size of Mathematica  code : 16 subexpressions
! Total size of Fortran code      : 244 bytes

!******************* S U B R O U T I N E *********************
      SUBROUTINE sub(v,x)
      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

In[187]:=

```
<< AceGen``;
SMSInitialize["test"];
SMSModule["Test", Real[a$$]];
a ⊢ SMSReal[a$$];
M ⊨ ( a   a^2 )
    ( a^2  0  )
```

$\{\{a, M_{2,1}\}, \{M_{2,1}, 0\}\}$

In[192]:=

```
SMSVariables[M]
```

{a, $M_{2,1}$}

SMSFreeze, SMSUnFreeze

SMSFreeze[*symbol*,{exp_1,exp_2,{exp_3…}..},*allOptions*]
create data objects that represent elements of arbitrarily structured list of expressions {exp_1,exp_2,{exp_3…}..} accordingly to given options *allOptions*. New auxiliary variables with the values {exp_1,exp_2,{exp_3…}...} and random signature are then generated and the resulting arbitrarily structured list is then assigned to symbol *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.

SMSFreeze[*symbol*,{exp_1,exp_2,...},*generalOptions*]
≡ *symbol*←SMSFreeze[{exp_1,exp_2,...}, *generalOptions*]

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

SMSFreeze[*exp*]
create data object (*SMSFreezeF*) that represents expression *exp*, however its numerical evaluation yields an unique signature obtained by the random perturbation of the original signature of *exp*

(original signature is restored for the second code optimization cycle at the code generation phase!!

SMSFreeze[*exp*,*generalOptions*]
create data object (*SMSFreezeF*) that represents expression *exp* accordingly to given general options *generalOptions*

SMSFreeze[{exp_1,exp_2,...},*generalOptions*]
create list of data objects (*SMSFreezeF*) that represent expressions {exp_1,exp_2,...} accordingly to given general options *generalOptions* (note that special options that apply on lists of expressions cannot be used in this form)

SMSUnFreeze[*exp*]
first search *exp* argument for all auxiliary variables that have been frozen by the SMSFreeze command and after then replace any appearance of those variables in expression *exp* by their definitions

Imposing restrictions on an optimization procedure.

Imposing restrictions on an optimization procedure.

option	default	description
"Dependency"	False	see below
"Contents"	False	whether to prevent the search for common sub expressions inside the expression *exp*
"Code"	False	whether to keep all options valid also at the code generation phase
"Differentiation"	False	whether to use SMSFreeze also for the derivatives of given expression *exp*
"Verbatim"	False	SMSFreeze[*exp*,"Verbatim"->True] ≡ SMSFreeze[*exp*,"Contents"->True , "Code"->True, "Differentiation"->True]
"Subordinate"->_List	{}	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

Options of the SMSFreeze function.

options for structured lists	default	description
"Ignore"->*crit*	(False&)	the *SMSFreeze* function is applied only on parts of the list for which *crit*[e] yields False (NumberQ[*exp*] yields True)
"Symmetric"	False	if an input is a matrix (symmetric or not) then the output is a symmetric matrix
"IgnoreNumbers"	False	True ≡ "Ignore"->NumberQ whether to apply *SMSFreeze* function only on parts of the list that are not numbers
"KeepStructure"	False	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)
"Variables"	False	apply SMSFreeze function on auxiliary variables that explicitly appear as a part of expression instead of expression as a whole

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

SMSFreeze[*exp*, "Dependency"→*value*]	description
True	assume that *SMSFreezeF* data object is independent variable (all partial derivatives of *exp* are 0)
False	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*)
$\left\{\left\{p_1,\frac{\partial exp}{\partial p_1}\right\},\left\{p_2,\frac{\partial exp}{\partial p_2}\right\},\ldots\right\}$	assume that *SMSFreezeF* data object depends on given auxiliary variables p_1, p_2,… and define the partial derivatives of *SMSFreezeF* data object with respect to given auxiliary variables p_1,p_2,…
$\left\{\left\{p_1,\frac{\partial exp}{\partial p_1}\right\},\ldots.\right.$ $\left\{\left\{p_2,\frac{\partial exp}{\partial p_2}\right\},\left\{p_3,\frac{\partial exp}{\partial p_1}\right\}\ldots.\right\},\ldots\right\}$	a set of definitions $\left\{p_i,\frac{\partial exp}{\partial p_i}\right\}$ can be arbitrarily structured, thus allowing definition of partial derivatives of components of tensors of an arbitrary order, etc.

Values for "Dependency" option when the input is a single expression.

SMSFreeze[{exp_1,exp_2, …}, "Dependency"→ *value*]	description
True	assume that all expressions are independent variables (all partial derivatives of exp_i are 0)
False	assume that after *SMSFreeze* expressions depend on the same auxiliary variables as original expressions
$\left\{p,\left\{\frac{\partial exp_1}{\partial p},\frac{\partial exp_2}{\partial p},\ldots\right\}\right\}$	define partial derivatives of $\{exp_1,exp_2,\ldots\}$ with respect to variable p to be $\left\{\frac{\partial exp_1}{\partial p},\frac{\partial exp_2}{\partial p},\ldots\right\}$
$\left\{\{p_1,p_2,\ldots\},\left\{\left\{\frac{\partial exp_1}{\partial p_1},\frac{\partial exp_1}{\partial p_2},\ldots\right\},\left\{\frac{\partial exp_2}{\partial p_1},\frac{\partial exp_2}{\partial p_2},\ldots\right\},\ldots\right\}\right\}$	define Jacobian matrix of the transformation from $\{exp_1,exp_2,\ldots\}$ to $\{p_1,p_2,\ldots\}$ to be matrix $\left\{\left\{\frac{\partial exp_1}{\partial p_1},\frac{\partial exp_1}{\partial p_2},\ldots\right\},\left\{\frac{\partial exp_2}{\partial p_1},\frac{\partial exp_2}{\partial p_2},\ldots\right\},\ldots\right\}$
$\left\{\left\{\left\{p_{11},\frac{\partial exp_1}{\partial p_{11}}\right\},\left\{p_{12},\frac{\partial exp_1}{\partial p_{12}}\right\},\ldots\right\},\left\{\left\{p_{21},\frac{\partial exp_2}{\partial p_{21}}\right\},\left\{p_{22},\frac{\partial exp_2}{\partial p_{22}}\right\},\ldots\right\},\ldots\right\}$	define arbitrary partial derivatives of vector of expressions $\{exp_1,exp_2,\ldots\}$

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$\left[exp, \text{"Dependency"} \to \left\{\left\{p_1, \frac{\partial exp}{\partial p_1}\right\}, \left\{p_2, \frac{\partial exp}{\partial p_2}\right\}, \ldots, \left\{p_n, \frac{\partial exp}{\partial p_n}\right\}\right\}\right]$ the true dependencies of *exp* are ignored and it is assumed that *exp* depends on auxiliary variables $p_1, \ldots, p_n$. Partial derivatives of *exp* with respect to auxiliary variables $p_1, \ldots, p_n$ are taken to be $\frac{\partial exp}{\partial p_1}, \frac{\partial exp}{\partial p_2}, \ldots, \frac{\partial exp}{\partial p_n}$ (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[57]:=
```
SMSFreeze[Sin[x]]
```

```
Freeze[Sin[x]]
```

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

In[58]:=
```
{Sin[x], SMSFreeze[Sin[x]]} // SMSEvaluate
```

```
{0.67104233, 0.66222981}
```

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

In[59]:=
```
xf ⊢ SMSFreeze[Sin[x]];
xf
```

xf

- or by

In[61]:=
```
SMSFreeze[xf, Sin[x]];
xf
```

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[1]:=
```
<< AceGen`;
SMSInitialize["test"];
SMSModule["sub", Real[x$$]];
x ⊢ SMSReal[x$$];
M = ( x      2 x Cos[x]
      2 x    2     2 x
      -Cos[x] 0    1/2 );
```

- The random signatures of elements of the original matrix are

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

$$\begin{pmatrix} 0.68150910 & 1.3630182 & 0.77662292 \\ 1.3630182 & 2.0000000 & 1.3630182 \\ -0.77662292 & 0 & 0.50000000 \end{pmatrix}$$

- 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 ⊢ SMSFreeze[M];
Mf // MatrixForm
```

$$\begin{pmatrix} Mf_{1,1} & Mf_{1,2} & Mf_{1,3} \\ Mf_{2,1} & Mf_{2,2} & Mf_{2,3} \\ Mf_{3,1} & Mf_{3,2} & Mf_{3,3} \end{pmatrix}$$

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

In[3502]:=

```
Mf // SMSEvaluate // MatrixForm
```

$$\begin{pmatrix} 0.62543972 & 1.3144220 & 0.74021934 \\ 1.3511905 & 1.9694969 & 1.3127609 \\ -0.73412511 & 0.00083184279 & 0.45988398 \end{pmatrix}$$

- Here the new auxiliary variables with random signatures are generated only for the elements of the matrix for which NumberQ[x] yields true. 6 new auxiliary variables are generated.

In[3509]:=

```
SMSFreeze[Mf, M, "IgnoreNumbers" → True];
Mf // MatrixForm
Mf // SMSEvaluate // MatrixForm
```

$$\begin{pmatrix} \boxed{Mf_{1,1}} & \boxed{Mf_{1,2}} & \boxed{Mf_{1,3}} \\ \boxed{Mf_{2,1}} & 2 & \boxed{Mf_{2,3}} \\ \boxed{Mf_{3,1}} & 0 & \frac{1}{2} \end{pmatrix}$$

$$\begin{pmatrix} 0.63975118 & 1.2659541 & 0.73419665 \\ 1.3304645 & 2.0000000 & 1.3240185 \\ -0.71005037 & 0 & 0.50000000 \end{pmatrix}$$

- 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.

In[3506]:=

```
SMSFreeze[Mf, M, "KeepStructure" → True];
Mf // MatrixForm
Mf // SMSEvaluate // MatrixForm
```

$$\begin{pmatrix} \boxed{Mf_{1,1}} & \boxed{Mf_{2,2}} & \boxed{Mf_{1,2}} \\ \boxed{Mf_{2,3}} & 2 & \boxed{Mf_{2,3}} \\ -\boxed{Mf_{1,3}} & 0 & \frac{1}{2} \end{pmatrix}$$

$$\begin{pmatrix} 0.66082676 & 1.2621213 & 0.76593957 \\ 1.2621213 & 2.0000000 & 1.2621213 \\ -0.76593957 & 0 & 0.50000000 \end{pmatrix}$$

- Here the new auxiliary variables with random signatures are generated only for the elements of the matrix for which NumberQ[x] yields true. Additionally, symmetry of the resulting matrix is enforced.

In[3512]:=

```
SMSFreeze[Mf, M, "Symmetric" → True, "IgnoreNumbers" → True];
Mf // MatrixForm
Mf // SMSEvaluate // MatrixForm
```

$$\begin{pmatrix} \boxed{Mf_{1,1}} & \boxed{Mf_{2,1}} & \boxed{Mf_{3,1}} \\ \boxed{Mf_{2,1}} & 2 & 0 \\ \boxed{Mf_{3,1}} & 0 & \frac{1}{2} \end{pmatrix}$$

$$\begin{pmatrix} 0.64944950 & 1.3471358 & -0.73228175 \\ 1.3471358 & 2.0000000 & 0 \\ -0.73228175 & 0 & 0.50000000 \end{pmatrix}$$

- 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
```

$$\begin{pmatrix} F_1 & 2\,F_1 & \mathrm{Cos}[F_1] \\ 2\,F_1 & 2 & 2\,F_1 \\ -\mathrm{Cos}[F_1] & 0 & \frac{1}{2} \end{pmatrix}$$

$$\begin{pmatrix} 0.65690486 & 1.3138097 & 0.79188613 \\ 1.3138097 & 2.0000000 & 1.3138097 \\ -0.79188613 & 0 & 0.50000000 \end{pmatrix}$$

Option dependency

In[55]:=
```
<< AceGen`;
SMSInitialize["test"];
SMSModule["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:
$\frac{\partial e1}{\partial p1} = 5$.
The derivatives $\left\{\frac{D\,\mathrm{Log}(e1)}{D\,p_1}, \frac{D\,\mathrm{Log}(e1)}{D\,p_2}\right\}$ are then evaluated assuming explicitly defined partial derivatives.

In[60]:=
```
f1 ⊢ SMSFreeze[e1, "Dependency" → {p1, 5}];
SMSD[Log[f1], {p1, p2}]
```

$$\left\{\frac{5}{f1}, 0\right\}$$

- Here all partial derivatives of expression e1 are set to 0 except:
$\frac{\partial e1}{\partial p1} = 5$, $\frac{\partial e1}{\partial p2} = 10$.
The derivatives $\left\{\frac{D\,\mathrm{Log}(e1)}{D\,p_1}, \frac{D\,\mathrm{Log}(e1)}{D\,p_2}\right\}$ are then evaluated assuming explicitly defined partial derivatives.

In[62]:=
```
f1 ⊢ SMSFreeze[e1, "Dependency" → {{p1, 5}, {p2, 10}}];
SMSD[Log[f1], {p1, p2}]
```

$$\left\{\frac{5}{f1}, \frac{10}{f1}\right\}$$

- Here all partial derivatives of expressions e1 and e2 are set to 0 except:
$\frac{\partial e1}{\partial p1} = 5$, $\frac{\partial e1}{\partial p2} = 10$,
$\frac{\partial e2}{\partial p1} = 15$, $\frac{\partial e2}{\partial p2} = 20$.
The derivatives $\begin{pmatrix} \frac{D\,\mathrm{Log}(e1)}{D\,p_1} & \frac{D\,\mathrm{Log}(e1)}{D\,p_2} \\ \frac{D\,\mathrm{Log}(e2)}{D\,p_1} & \frac{D\,\mathrm{Log}(e2)}{D\,p_2} \end{pmatrix}$ are then evaluated assuming explicitly defined gradients of expressions.

In[70]:=

```
{f1, f2} ⊢ SMSFreeze[{e1, e2}, "Dependency" →
    {
     {{p1, 5}, {p2, 10}}  (*∇e1*)
     , {{p1, 15}, {p2, 20}}  (*∇e2*)
    }];
SMSD[{Log[f1], Log[f2]}, {p1, p2}]
```

$$\left\{\left\{\frac{5}{f1}, \frac{10}{f1}\right\}, \left\{\frac{15}{f2}, \frac{20}{f2}\right\}\right\}$$

- The above result can be also obtained by defining a set of unknowns p_i and the Jacobian matrix $J_{i,j} = \frac{\partial e_i}{\partial p_j}$.

In[72]:=

```
{f1, f2} ⊢ SMSFreeze[{e1, e2}, "Dependency" →
    {{p1, p2},  (*pi*)
     {{5, 10}, {15, 20}} (*Jacobian ∂ei/∂pj *)
    }];
SMSD[{Log[f1], Log[f2]}, {p1, p2}]
```

$$\left\{\left\{\frac{5}{f1}, \frac{10}{f1}\right\}, \left\{\frac{15}{f2}, \frac{20}{f2}\right\}\right\}$$

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.

In[24]:=

```
<< AceGen`;
SMSInitialize["test"];
SMSModule["sub", Real[x$$]];
x ⊢ SMSReal[x$$];
```

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

In[3555]:=

```
vf ⊢ SMSFreeze[{Sin[x], 0}, "IgnoreNumbers" → True];
vf
```

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

$\{vf_1, vf_2\}$

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

In[3559]:=

```
SMSFreeze[vf, {Sin[x], 0}, "IgnoreNumbers" → True];
vf
```

$\{vf_1, 0\}$

SMSFictive

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

SMSFictive[] ≡ 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 $f(n) = \sum_{n=1}^{m} \frac{\partial g(x)}{\partial x}\Big|_{x=0}$, 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$$], Integer[m$$]];
f ⫤ 0;
SMSDo[n, 1, SMSInteger[m$$], 1, f];
x ⊢ SMSFictive[];
g ⊨ Sin[x/n] + Cos[x/n];
f ⫞ f + SMSReplaceAll[SMSD[g, x], x → 0];
SMSEndDo[f];
SMSExport[f, f$$];
```

In[185]:=

```
SMSWrite[];
```

File:	test.c	Size: 835
Methods	No.Formulae	No.Leafs
sub	3	15

In[186]:=

```
FilePrint["test.c"]
```

```
/*************************************************************
* AceGen    3.304 Windows (7 Jun 12)                         *
*           Co. J. Korelc  2007            8 Jun 12 11:33:43 *
*************************************************************
User     : USER
Notebook : AceGenSymbols.nb
Evaluation time                 : 0 s     Mode  : Optimal
Number of formulae              : 3       Method: Automatic
Subroutine                      : sub size :15
Total size of Mathematica  code : 15 subexpressions
Total size of C code            : 236 bytes*/
#include "sms.h"

/******************* S U B R O U T I N E *********************/
void sub(double v[5005],double (*f),double (*a),double (*b),int (*m))
{
int i2;
v[1]=0e0;
for(i2=1;i2<=(int)((*m));i2++){
 v[1]=1e0/i2+v[1];
};/* end for */
(*f)=v[1];
};
```

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.

Non-local operations are performed by corresponding Mathematica commands (Solve, Integrate, ...). The results of non-local operations is an general a regular Mathematica expression. However only those Mathematica expressions that are composed solely from elementary functions (+,-./,sin,Cos,...) that have C/FORTRAN/MATLAB equivalent are acceptable for AceGen automatic code generator. Thus one can define that the function is Integrable in a Mathematica/AceGen sense when the use of Mathematica command Integrate results in an expression that is solely composed from elementary functions (+,-./,sin,Cos,...) that have C/FORTRAN/MATLAB equivalent.

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.*

In[21]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "Mathematica"];
SMSModule["Test", Real[x$$, y$$, z$$]];
{x, y, z} ⊢ {SMSReal[x$$], SMSReal[y$$], SMSReal[z$$]};

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

Out[ ]=

$$x^2 + 4\, x\, y + y^2 + 2\, y\, z + z^2$$

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 *fx.* Additionally Normal converts *expr* to a normal expression, from a variety of *AceGen* special forms.

In[26]:=

```
fx = SMSSmartReduce[f, x, Collect[#, x] &] // Normal
```

Out[ ]=

$$x^2 + \S1 + x\, \S2$$

The following vector of auxiliary variables is created.

In[27]:=

```
SMSShowVector[]
```

```
$S[Method, Null, 1]
```

V[1,1] ≡ $_1x$ ≡ x$$

V[2,1] ≡ $_2y$ ≡ y$$

V[3,1] ≡ $_3z$ ≡ z$$

V[4,1] ≡ $_4\S1$ ≡ $_2y^2 + 2\, {}_2y\, {}_3z + {}_3z^2$

V[5,1] ≡ $_5\S2$ ≡ $4\, {}_2y$

```
$S[End, Null, {}]
```

In[28]:=

$$\text{fint} = \int \text{fx}\, dx$$

Out[ ]=

$$\frac{x^3}{3} + x\, \S1 + \frac{x^2\, \S2}{2}$$

Note that the function Sin[fx] is not integrable in Mathematica/AceGen sens although the command Integrate produces an expression. It can become integrable if the user adds the definition of FresnelS and FresnelC as the user defined functions (see `SMSCall`**).**

In[32]:=

```
∫Sin[fx] dx
```

Out[•]=

$$\sqrt{\frac{\pi}{2}}\left(\text{Cos}\left[\boxed{\S1}-\frac{\boxed{\S2}^2}{4}\right]\text{FresnelS}\left[\frac{2\boxed{x}+\boxed{\S2}}{\sqrt{2\pi}}\right]+\text{FresnelC}\left[\frac{2\boxed{x}+\boxed{\S2}}{\sqrt{2\pi}}\right]\text{Sin}\left[\boxed{\S1}-\frac{\boxed{\S2}^2}{4}\right]\right)$$

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*.

In[289]:=

```
fr = SMSSmartRestore[fint, y, Collect[#, y] &] // Normal
```

Out[•]=

$$\boxed{x}\,\boxed{y}^2+\boxed{\S3}+\boxed{y}\,\boxed{\S4}$$

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

SMSSmartReduce

SMSSmartReduce[*exp*,v_1|v_2|...]
replace those parts of the expression *exp* that do not depend on any of the auxiliary variables v_1|v_2|.... by a new auxiliary variable

SMSSmartReduce[*exp*,v_1|v_2|...,*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[#,v1|v2|...]&. The function *func* should perform only correctness preserving transformations, so that the value of expression *exp* remains the same.

SMSSmartRestore

SMSSmartRestore[*exp*,v_1|v_2|...]
replace those parts of expression *exp* that depend on any of the auxiliary variables v_1|v_2|... by their definitions and simplify the result

SMSSmartRestore[*exp*,v_1|v_2|...,*func*]
apply pure function *func* to the sub-expressions that do not depend on v_1|v_2|... before they are replaced by a new auxiliary variable

SMSSmartRestore[*exp*,v_1|v_2|...,{*evaluation_rules*},*func*]

restore expression *exp* and apply list of rules {*evaluation_rules*} to all sub-expressions that depend on any of auxiliary variables $v_1, v_2, \ldots$

At the output, all variables $v_1|v_2|\ldots$ 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[#,v1|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 v_1, v_2, ... 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*,$v_1|v_2|\ldots$]
replace those parts of expression *exp* that depend on any of the auxiliary variables $v_1|v_2|\ldots$. by their definitions

SMSRestore[*exp*,$v_1|v_2|\ldots$,{*evaluation_rules*}]
restore expression *exp* and apply list of rules {*evaluation_rules*} to all sub-expressions that depend on any of auxiliary variables $v_1, v_2, \ldots$

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 $v_1|v_2|\ldots$ 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*,v_1->new_1,v_2->new_2, ...]
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 $exp\,|_{v_1=new_1,v_2=new_2,\ldots}$. 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,

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

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 $3x^2 + \mathrm{Sin}[x^2] - \mathrm{Log}[x^2 - 1]$.

In[89]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "Fortran"];
SMSModule["Test", Real[x0$$, f0$$, fx0$$]];
x0 ⊢ SMSReal[x0$$];
x ⊨ SMSFictive[];
f ⊨ 3 x^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}, {f0$$, fx0$$}];
SMSWrite[];
FilePrint["test.f"];
```

File:	test.f	Size: 908
Methods	No.Formulae	No.Leafs
Test	3	48

```
!*************************************************************
!* AceGen    2.502 Windows (24 Nov 10)                       *
!*           Co. J. Korelc  2007            29 Nov 10 15:07:22*
!*************************************************************
! User : Full professional version
! Evaluation time                 : 0 s     Mode   : Optimal
! Number of formulae              : 3       Method: Automatic
! Subroutine                      : Test size :48
! Total size of Mathematica  code : 48 subexpressions
! Total size of Fortran code      : 324 bytes

!******************* S U B R O U T I N E *********************
      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
```

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.

In[138]:=

```
<< AceGen`;
SMSInitialize["test"];
SMSModule["sub", Real[x$$]];
x ⊢ SMSReal[x$$];
y ⊨ - x;
z ⊨ Sin[y];
SMSReplaceAll[z, y → π / 3]
```

Out[ ]=

z

- The ⊢ command always creates new variable and leads to the correct results.

In[145]:=

```
<< AceGen`;
SMSInitialize["test"];
SMSModule["sub", Real[x$$]];
x ⊢ SMSReal[x$$];
y ⊢ - x;
z ⊨ Sin[y];
SMSReplaceAll[z, y → π / 3]
```

Out[ ]=

$$\frac{\sqrt{3}}{2}$$

Example 3: repeated use of SMSReplaceAll

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

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];
```

- Better formulation.

In[160]:=

```
<< AceGen`;
SMSInitialize["test"];
SMSModule["sub", Real[x$$]];
x ⊢ SMSReal[x$$];
y ⊨ Sin[x];
z ⊣ Cos[x];
{y0, z0} ⊨ SMSReplaceAll[{y, z}, x → 0];
```

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

```
Do[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

```
SMSDo[i, 1, 5]
SMSPrintMessage[i];
SMSEndDo[]
```

and using in-cell form as

```
SMSDo[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 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 *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 perturbated 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 perturbated 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 hybrid symbolic-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

$$f(x) = \begin{cases} x \le 0 & x^2 \\ x > 0 & \mathrm{Sin}[x] \end{cases}.$$

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

In[191]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "Fortran"];
SMSModule["test", Real[x$$, f$$]];
x ⊢ SMSReal[x$$];
f ⊨ SMSIf[x ≤ 0, x^2, Sin[x]];
SMSExport[f, f$$];
SMSWrite["test"];
FilePrint["test.f"]
```

```
Method : test 3 formulae, 16 sub-expressions

[0] File created : test.f Size : 861

!*************************************************************
!* AceGen    2.103 Windows (18 Jul 08)                       *
!*           Co. J. Korelc  2007            18 Jul 08 15:41:18*
!*************************************************************
! User : USER
! Evaluation time                 : 0 s     Mode   : Optimal
! Number of formulae              : 3       Method: Automatic
! Subroutine                      : test size :16
! Total size of Mathematica  code : 16 subexpressions
! Total size of Fortran code      : 295 bytes

!******************* S U B R O U T I N E *********************
      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

$$f(x) = \begin{cases} x \le 0 & x^2 \\ x > 0 & \mathrm{Sin}[x] \end{cases}.$$

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

In[207]:=
```
<< AceGen`;
SMSInitialize["test", "Language" → "Fortran"];
SMSModule["test", Real[x$$, f$$]];
x ⊢ SMSReal[x$$];
```

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

In[211]:=
```
SMSIf[x ≤ 0]
```

$x \le 0$

In[212]:=
```
f ⊣ x^2;
```

In[213]:=
```
SMSElse[]
```

$x \le 0$

In[214]:=
```
f ⊣ Sin[x];
```

In[215]:=
```
SMSEndIf[f]
```

f

In[216]:=
```
SMSExport[f, f$$];
SMSWrite["test"];
```

```
Method : test 3 formulae, 16 sub-expressions
[0] File created : test.f Size : 861
```

- This displays the contents of the generated file.

In[218]:=

```
FilePrint["test.f"]
```

```
!*************************************************************
!* AceGen    2.103 Windows (18 Jul 08)                       *
!*           Co. J. Korelc  2007            18 Jul 08 15:41:19*
!*************************************************************
! User : USER
! Evaluation time                 : 0 s     Mode  : Optimal
! Number of formulae              : 3       Method: Automatic
! Subroutine                      : test size :16
! Total size of Mathematica  code : 16 subexpressions
! Total size of Fortran code      : 295 bytes

!******************* S U B R O U T I N E *********************
      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 3: Generic example - using ternary operator

Generation of the Fortran subroutine which evaluates the following function

$$f(x) = \begin{cases} x \le 0 & x^2 \\ x > 0 & \mathrm{Sin}[x] \end{cases}.$$

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

In[124]:=

```
<< 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"];
FilePrint["test.f"]
```

```
File: test.f  Size: 809  Time: 0
```

Method	test
No.Formulae	1
No.Leafs	34

```
!*************************************************************
!* AceGen    6.819 Windows (14 Oct 17)                      *
!*           Co. J. Korelc  2013          15 Oct 17 08:43:23 *
!*************************************************************
! User     : USER
! Notebook : AceGenTutorials
! Evaluation time                 : 0 s     Mode   : Optimal
! Number of formulae              : 1       Method: Automatic
! Subroutine                      : test size: 34
! Total size of Mathematica  code : 34 subexpressions
! Total size of Fortran code      : 215 bytes

!******************* S U B R O U T I N E *********************
      SUBROUTINE test(v,x,f)
      IMPLICIT NONE
      include 'sms.h'
      DOUBLE PRECISION v(8),x,f
      f=MERGE(x**2,dsin(x),x.le.0d0)
      END
```

Example 4: Incorrect logical expression

The expression x<=0 && i=="0" in this example is evaluated already in *Mathematica* because the === operator always yields True or False. The correct form of the logical condition would be x<=0 && i=="0".

In[199]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "Fortran"];
SMSModule["test", Real[x$$, f$$], Integer[i$$]];
x ⊢ SMSReal[x$$];
f ⊨ SMSIf[x ≤ 0 && i === "0", x^2, Sin[x]];
SMSExport[f, f$$];
SMSWrite[];
```

```
The expressions of the form a===b or a=!=b in Hold[x ≤ 0 && i === 0]
  are evaluated in Mathematica and not later in the source code !!!
  Consider using a==b or a!=b instead. See also: SMSIf

Method : test 1 formulae, 7 sub-expressions

[0] File created : test.f Size : 774
```

In[206]:=

```
FilePrint["test.f"]
```

```
!*************************************************************
!* AceGen    2.103 Windows (18 Jul 08)                       *
!*           Co. J. Korelc  2007            18 Jul 08 15:41:19*
!*************************************************************
! User : USER
! Evaluation time                 : 0 s     Mode  : Optimal
! Number of formulae              : 1       Method: Automatic
! Subroutine                      : test size :7
! Total size of Mathematica  code : 7 subexpressions
! Total size of Fortran code      : 215 bytes

!******************* S U B R O U T I N E *********************
      SUBROUTINE test(v,x,f,i)
      IMPLICIT NONE
      include 'sms.h'
      INTEGER i
      DOUBLE PRECISION v(5001),x,f
      f=dsin(x)
      END
```

Example 5: Incorrect use of ternary operator

Generation of the C subroutine which evaluates the following function

$$f(x) = \begin{cases} x \le 0 & x^2 \\ x > 0 & y = \mathrm{Cos}[x];\ y + y^2 \end{cases}.$$

In[132]:=

```
<< 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<=0 as seen from the generated code.

In[136]:=

```
f ⊨ SMSIf[x ≤ 0, x^2, y ⊨ Cos[x]; y + y^2, "Inline" → True];
SMSExport[f, f$$];
SMSWrite["test"];
FilePrint["test.c"]
```

```
File: test.c  Size: 780  Time: 0
```

Method	test
No.Formulae	2
No.Leafs	43

```
/*************************************************************
* AceGen    6.819 Windows (14 Oct 17)                       *
*           Co. J. Korelc  2013          15 Oct 17 08:53:28 *
*************************************************************
User     : USER
Notebook : AceGenTutorials
Evaluation time                 : 0 s     Mode  : Optimal
Number of formulae              : 2       Method: Automatic
Subroutine                      : test size: 43
Total size of Mathematica  code : 43 subexpressions
Total size of C code            : 184 bytes */
#include "sms.h"

/******************* S U B R O U T I N E *********************/
void test(double v[9],double (*x),double (*f))
{
v[2]=cos((*x));
(*f)=((*x)<=0e0?Power((*x),2):v[2]+Power(v[2],2));
};
```

Example 6: Redundant code

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

$$f(x) = \begin{cases} x \le 0 & x^2 \\ x > 0 & \mathrm{Sin}[x] \end{cases}.$$

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.

In[343]:=

```
<< 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
[0] File created : test.c Size : 931
0.471
```

In[357]:=

```
FilePrint["test.c"]
```

```
/*************************************************************
* AceGen    2.103 Windows (18 Jul 08)                        *
*           Co. J. Korelc  2007            18 Jul 08 02:35:50*
*************************************************************
User : USER
Evaluation time                 : 0 s     Mode  : Optimal
Number of formulae              : 7       Method: Automatic
Subroutine                      : test size :39
Total size of Mathematica  code : 39 subexpressions
Total size of C code            : 351 bytes*/
#include "sms.h"

/******************* S U B R O U T I N E *********************/
void test(double v[5001],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^2;
  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
```

In[372]:=

```
FilePrint["test.c"]
```

```
/*************************************************************
* AceGen    2.103 Windows (18 Jul 08)                        *
*           Co. J. Korelc  2007            18 Jul 08 02:35:51*
*************************************************************
User : USER
Evaluation time                 : 0 s     Mode  : Optimal
Number of formulae              : 5       Method: Automatic
Subroutine                      : test size :30
Total size of Mathematica  code : 30 subexpressions
Total size of C code            : 289 bytes*/
#include "sms.h"

/******************* S U B R O U T I N E *********************/
void test(double v[5001],double (*x),double (*f),double (*d))
{
int b2,b6;
if((*x)<=0e0){
 v[3]=Power((*x),2);
 v[5]=2e0*(*x);
} else {
};
if((*x)>0e0){
 v[3]=sin((*x));
 v[5]=cos((*x));
} else {
};
(*f)=v[3];
(*d)=v[5];
};
```

Example 7: Variables out of scope 1

Generation of the Fortran subroutine which evaluates the following function

$$f(x) = \begin{cases} x \le 0 & x^2 \\ x > 0 & \mathrm{Sin}[x] \end{cases}.$$

In[1]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "Fortran"];
SMSModule["test", Real[x$$, f$$]];
x ⊢ SMSReal[x$$];
SMSIf[x ≤ 0];
f ⊣ x^2;
SMSElse[];
f ⊣ Sin[x];
SMSEndIf[];
```

- Here symbol *f* appears outside the "If" construct. Since *f* is not specified in the *SMSEndIf* statement, we get "variable out of scope" error message.

In[1]:=

```
SMSExport[f, f$$];
```

Some of the auxilirary variables in expression
are defined out of the scope of the current position.

```
Module: test Description: Error in user input parameters for function: SMSExport
                          Input parameter: {2f} Current scope: {}
                          Misplaced variables :
                           2f ≡ $V[3, 2] Scope: If-False[x ≤ 0]

Events: 0
Version: 3.001 Windows (1 Mar 11) (MMA 7.)
  See also: Auxiliary Variables  AceGen Troubleshooting
```

SMC::Fatal : System cannot proceed with the evaluation due to the fatal error in SMSExport .

```
$Aborted
```

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.

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

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)

In[425]:=

```
    SMSIf[NN < 5];
        NN ⫤ 0;
    SMSEndIf[];
SMSEndIf[NN];
SMSExport[NN, r$$];
SMSWrite[];
```

```
[0]  Consistency check - global

[0]  Consistency check - expressions

Fatal error: Variables {{$V[2, 2], ≡, 2NN}} in formula $V[7, 1]≡¥7 are out of scope.
```

SMS::SystemCheck : Fatal errors were detected.()

```
[0]  Generate source code :

Events: 0

[0] Final formating
```

File:	test.m	Size: 1311
Methods	No.Formulae	No.Leafs
test	11	25

- 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 formating
```

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

SMSSwitch

SMSSwitch[*expr*,*form*$_1$,*value*$_1$,*form*$_2$,*value*$_2$,...]
Creates code that evaluates *expr*, then compares it with each of the *form*$_i$ in turn, evaluating and returning the *value*$_i$ corresponding to the first match found. The value returned during the *AceGen* session represents all options (see also: Program Flow Control)

SMSSwitch[*expr*,*form*$_1$,*value*$_1$,*form*$_2$,*value*$_2$,...,_,*default_value*]
definition of the *default_value*

SMSSwitch[*expr*,*form*$_1$,*value*$_1$,*form*$_2$,*value*$_2$,..., "Inline"->True]
creates an expression that evaluates *expr*, then compares it with each of the *form*$_i$ in turn, evaluating and returning the *value*$_i$ 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 $form_i$ are integer type expressions. The SMSSwitch command returns multi-valued auxiliary variable with random signature that represents all options. If all $value_i$ evaluate to Null then SMSSwitch returns Null. If all $value_i$ evaluate to vectors of the same length then SMSSwitch returns a corresponding vector of multi-valued auxiliary variables.

Warning: If none of the $form_i$ match *expr*, the SMSSwitch returns arbitrary value.

SMSWhich

SMSWhich[$test_1$,$value_1$,$test_2$,$value_2$,...]
Creates code that evaluates each of the $test_i$ in turn, returning the value of the $value_i$ corresponding to the first one that yields True. The value returned during the *AceGen* session represents all options (see also: Program Flow Control).

SMSWhich[$test_1$,$value_1$,$test_2$,$value_2$,...,True,*default_value*]
definition of the *default_value*

SMSWhich[$test_1$,$value_1$,$test_2$,$value_2$,..., "Inline"->True]
creates an expression evaluates each of the $test_i$ in turn, returning the value of the $value_i$ 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 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 SMSWhich construct.

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

Warning: If none of the $test_i$ 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, SMSEndDo

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}}] ≡ SMSDo[*expr*,{i,1,i_{max},1}]

SMSDo[*expr*,{i,i_{min},i_{max}}] ≡ SMSDo[*expr*,{i,1,i_{max},1}]

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

SMSDo[i,i_{min},i_{max},di,in_var]
start the “Do” loop with an auxiliary variable i successively taken on the values i_{min} through i_{max} in steps of di and create fictive instances of the *in_var* auxiliary variables

SMSEndDo[]
end the loop

SMSEndDo[out_var]
end the loop and create fictive instances of the *out_var* auxiliary variables

Syntax of the "cross-cell" loop construct.

option	default	description
"MaxIndexRange"	Automatic	$\{i_{min_constant}, i_{max_constant}\}$ ⇒ absolute range of loop index Althout the values of i_{min}, i_{max} can be an arbitrary integer type expressions, in efficient expression optimisation sometimes requires the knowledge of the absolutely minimum nad maximum values of i_{min} and i_{max} that would remain constant during the evaluation of code.

Options of "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 SMSEndDo 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 $f(x) = 1 + \sum_{i=1}^{n} x^i$.

In[122]:=
```
<< AceGen`;
SMSInitialize["test", "Language" → "Fortran"];
SMSModule["test", Real[x$$, f$$], Integer[n$$]];
x ⊢ SMSReal[x$$]; n ⊢ SMSInteger[n$$];
f ⊣ 1;
SMSDo[
  f ⊣ f + x^i;
, {i, 1, n, 1, f}];
SMSExport[f, f$$];
SMSWrite["test"];
FilePrint["test.f"]
```

```
 File:     test.f    Size: 902
Methods No.Formulae No.Leafs
 test        4          23
```

```
!*************************************************************
!* AceGen    5.001 Windows (26 Dec 12)                       *
!*           Co. J. Korelc  2007            27 Dec 12 10:21:43*
!*************************************************************
! User     : USER
! Notebook : AceGenSymbols.nb
! Evaluation time                 : 0 s     Mode   : Optimal
! Number of formulae              : 4       Method: Automatic
! Subroutine                      : test size :23
! Total size of Mathematica  code : 23 subexpressions
! Total size of Fortran code      : 301 bytes

!******************* S U B R O U T I N E *********************
      SUBROUTINE test(v,x,f,n)
      IMPLICIT NONE
      include 'sms.h'
      INTEGER n,i2,i4
      DOUBLE PRECISION v(5005),x,f
      i2=int(n)
      v(3)=1d0
      DO i4=1,i2
       v(3)=v(3)+x**i4
      ENDDO
      f=v(3)
      END
```

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.

In[131]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "Fortran"];
SMSModule["test", Real[x$$, f$$], Integer[n$$]];
x ⊢ SMSReal[x$$];
n ⊢ SMSInteger[n$$];
```

- Description of the loop.

In[135]:=

```
f ⊣ 1;
SMSDo[i, 1, n, 1, f];
    f ⊣ f + x^i;
SMSEndDo[f];
```

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

In[139]:=

```
SMSExport[f, f$$];
SMSWrite["test"];
```

```
 File:     test.f    Size: 902
Methods No.Formulae No.Leafs
 test        4          23
```

- This displays the contents of the generated file.

In[141]:=

```
FilePrint["test.f"]
```

```
!*************************************************************
!* AceGen    5.001 Windows (26 Dec 12)                       *
!*           Co. J. Korelc  2007            27 Dec 12 10:22:42*
!*************************************************************
! User     : USER
! Notebook : AceGenSymbols.nb
! Evaluation time                 : 0 s     Mode   : Optimal
! Number of formulae              : 4       Method: Automatic
! Subroutine                      : test size :23
! Total size of Mathematica  code : 23 subexpressions
! Total size of Fortran code      : 301 bytes

!******************* S U B R O U T I N E *********************
      SUBROUTINE test(v,x,f,n)
      IMPLICIT NONE
      include 'sms.h'
      INTEGER n,i2,i4
      DOUBLE PRECISION v(5005),x,f
      i2=int(n)
      v(3)=1d0
      DO i4=1,i2
       v(3)=v(3)+x**i4
      ENDDO
      f=v(3)
      END
```

Example 3: Incorrect and correct use of "Do" construct

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

- 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.

In[142]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "Fortran", "Mode" → "Optimal"];
SMSModule["test", Real[S$$], Integer[n$$]];
n ⊢ SMSInteger[n$$];
S ⊣ 0;
SMSDo[i, 1, n, 1];
    S ⊣ Cos[S];
SMSEndDo[S];
SMSExport[S, S$$];
SMSWrite["test"];
```

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

File:	test.f	Size: 891
Methods	No.Formulae	No.Leafs
test	4	12

In[152]:=

```
FilePrint["test.f"]
```

```
!*************************************************************
!* AceGen    5.001 Windows (26 Dec 12)                       *
!*           Co. J. Korelc  2007            27 Dec 12 10:23:10*
!*************************************************************
! User     : USER
! Notebook : AceGenSymbols.nb
! Evaluation time                 : 0 s     Mode  : Optimal
! Number of formulae              : 4       Method: Automatic
! Subroutine                      : test size :12
! Total size of Mathematica  code : 12 subexpressions
! Total size of Fortran code      : 290 bytes

!******************* S U B R O U T I N E *********************
      SUBROUTINE test(v,S,n)
      IMPLICIT NONE
      include 'sms.h'
      INTEGER n,i1,i3
      DOUBLE PRECISION v(5005),S
      i1=int(n)
      v(2)=0d0
      DO i3=1,i1
       v(2)=1d0
      ENDDO
      S=v(2)
      END
```

- Correct formulation

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

In[163]:=

```
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"]
```

File:	test.f	Size: 898
Methods	No.Formulae	No.Leafs
test	4	15

```
!*************************************************************
!* AceGen    5.001 Windows (26 Dec 12)                       *
!*           Co. J. Korelc  2007            27 Dec 12 10:23:46*
!*************************************************************
! User     : USER
! Notebook : AceGenSymbols.nb
! Evaluation time                 : 0 s     Mode   : Optimal
! Number of formulae              : 4       Method: Automatic
! Subroutine                      : test size :15
! Total size of Mathematica  code : 15 subexpressions
! Total size of Fortran code      : 297 bytes

!******************* S U B R O U T I N E *********************
      SUBROUTINE test(v,S,n)
      IMPLICIT NONE
      include 'sms.h'
      INTEGER n,i1,i3
      DOUBLE PRECISION v(5005),S
      i1=int(n)
      v(2)=0d0
      DO i3=1,i1
       v(2)=dcos(v(2))
      ENDDO
      S=v(2)
      END
```

The "in-cell" form by default assignes the random signature to *S* at the beginning and at the end of the loop, thus gives correct result.

In[173]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "Fortran"];
SMSModule["test", Real[S$$], Integer[n$$]];
n ⊢ SMSInteger[n$$];
S ⫤ 0;
SMSDo[
 S ⫤ Cos[S];
, {i, 1, n, 1, S}];
SMSExport[S, S$$];
SMSWrite["test"];
FilePrint["test.f"]
```

```
 File:     test.f    Size: 898
Methods No.Formulae No.Leafs
 test        4         15

!*************************************************************
!* AceGen    5.001 Windows (26 Dec 12)                       *
!*           Co. J. Korelc  2007            27 Dec 12 10:23:50*
!*************************************************************
! User     : USER
! Notebook : AceGenSymbols.nb
! Evaluation time                 : 0 s     Mode   : Optimal
! Number of formulae              : 4       Method: Automatic
! Subroutine                      : test size :15
! Total size of Mathematica  code : 15 subexpressions
! Total size of Fortran code      : 297 bytes

!******************* S U B R O U T I N E *********************
      SUBROUTINE test(v,S,n)
      IMPLICIT NONE
      include 'sms.h'
      INTEGER n,i1,i3
      DOUBLE PRECISION v(5005),S
      i1=int(n)
      v(2)=0d0
      DO i3=1,i1
       v(2)=dcos(v(2))
      ENDDO
      S=v(2)
      END
```

In[353]:=

Example 4: How to use variables defined inside the loop outside the loop?

Only the multi-valued variables (introduced by the ⫤ or ⫞ command) can be used outside the loop. The use of the single-valued variables (introduced by the ⊨ or ⊢ 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[S$$], Integer[n$$]];
n ⊢ SMSInteger[n$$];
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: ,X^2 Current scope: {}
Misplaced variables :
,X ≡ $V[4, 1] Scope: Do[i, 1, n, 1]

Events: 0
Version: 5.001 Windows (26 Dec 12) (MMA 9.) **Module:** ⊨

See also: Auxiliary Variables AceGen Troubleshooting Continue

SMC::Fatal : System cannot proceed with the evaluation due to the fatal error in ⊨ .

$Aborted

- Correct formulation for "in-cell" form

In[189]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "Fortran"];
SMSModule["test", Real[S$$], Integer[n$$]];
n ⊢ SMSInteger[n$$];
S ⫤ 0;
SMSDo[
  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[S$$], Integer[n$$]];
n ⊢ SMSInteger[n$$];
S ⫤ 0;
SMSDo[i, 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)

SMSBreak[] ≡ SMSVerbatim["C"->"break;" , "Fortran"->"exit", "*Mathematica*"->"Break[];"]
(see *Mathematica* command Break)

SMSContinue[] ≡ 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 environment 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.

definition of the input/output parameters	example
SMSModule["name", Real[real variables], Integer[integer type variables], Logical[logical variables]]	SMSModule["test",Real[y$$[2,5]]]

external variables as a part of expression	example
SMSReal[real external data]	y ⊨ 2 Sin[SMSReal[y$$[2,5]]]
SMSInteger[integer external data]	i ⊨ SMSInteger[ii$$]
SMSLogical[logical data]	l ⊨ SMSLogical[bool$$] && y<0

exporting values	example
SMSExport[value, real external]	SMSExport[x+5, y$$[2,5]]
SMSExport[value, integer external]	SMSExport[2 i+7, ii$$]
SMSExport[value, logical external]	SMSExport[True, bool$$]

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:

type	*AceGen definition*	*FORTRAN definition*	*C definition*
real variable	x$$ x$$$	real* 8 x real* 8 x	double *x double x
real array	x$$[10] x$$[i$$, "*"] x$$[3, 5]	real* 8 x (10) real* 8 x (i,*) real* 8 x (3,5)	double x[10] double **x double x[3][5]
integer variable	i$$ i$$$	integer i integer i	int *i int i
integer array	i$$[10] i$$[i$$, "*"] i$$[3,5,7]	integer x (10) integer x (i,*) integer x (3,5,7)	int i[10] int **i int i[3][5][7]
logical variable	l$$ l$$$	logical l logical l	int *l int l

External variables in a subroutine declaration.

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:

type	AceGen form	FORTRAN form	C form
real variable	SMSReal[x$$] SMSReal[x$$$]	x x	*x x
real array	SMSReal[x$$[10]] SMSReal[x$$[i$$, "->name",5]] SMSReal[x$$[i$$, ".name",5]]	x (10) illegal illegal	x[10] x[i-1]->name[4] x[i-1].name[4]
integer variable	SMSInteger[i$$] SMSInteger[i$$$]	i i	*i i
integer array	SMSInteger[i$$[10]] SMSInteger[i$$["10"]] SMSInteger[i$$[j$$, "->name",5]] SMSInteger[i$$[j$$, ".name",5]]	i (10) i (10) illegal illegal	i[10] i[10] i[j-1]->name[4] i[j-1].name[4]
logical variable	SMSLogical[l$$] SMSLogical[l$$$]	l l	*l l

External variables as a part of expression.

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.

option	default	description
"Dependency"	True	define partial derivatives of external data object (*SMSExternalF*) with respect to given auxiliary variables (for the detailed syntax see `SMSFreeze`, User Defined Functions)
"Subordinate"	{}	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)

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. r=SMSReal[r$$]). 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.

option	default	description
"Subordinate"-> $\{v_1, v_2 \ldots\}$	{}	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)
"Subordinate"-> v_1		≡ "Subordinate"-> $\{v_1\}$

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*

option	default	description
"Subordinate"-> $\{v_1, v_2 \ldots\}$	{}	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)
"Subordinate"->v_1		≡ "Subordinate"->$\{v_1\}$

Options of the SMSLogical function.

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[$\{eID_1, eID_2, \ldots\}$, *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 $\{eID_1, eID_2, \ldots\}$ 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.

option	default	description
"Description"->{…}	$\{eID_1, eID_2, \ldots\}$	a list of descriptions that corresponds to the list of array element identifications $\{eID_1, eID_2, \ldots\}$
"Length"->*l*	1	each array element identification eID_i can also represent a part of *array* with the given length
"Index"->*i*	1	index of the actual array element taken from the part of *array* associated with the array element identification eID_i (index starts with 1)
"Signature"	{1,1,…}	a list of characteristic real type values that corresponds to the list of array element identifications $\{eID_1, eID_2, \ldots\}$

Options of the SMSRealList function.

return_code	description
"Description"	the values of the option "Description"
"Signature"	the values of the option "Signature"
"Export"	the patterns (e.g. ed$$[5]) suitable as parameter for SMSExport function
"Length"	the accumulated length of all elements which array element identification *eID* match pattern
"ID"	array element identifications
"Plain"	external data objects with all auxiliary variables replaced by their definition
"Exist"	True if the data with the given pattern exists
"Start"	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

Return codes for SMSRealList.

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$$]];
SMSRealList[{"a1", "a2"}, a$$[#] &]
```
Out[•]=

$\{a\$\$_1, a\$\$_2\}$

In[21]:=
```
SMSRealList[{"a3", "a4"}, a$$[#] &]
```
Out[•]=

$\{a\$\$_3, a\$\$_4\}$

In[22]:=
```
SMSRealList["a3"]
```
Out[•]=

$a\$\$_3$

In[23]:=
```
SMSRealList[{"b1", "b2"}, b$$[#] &, "Length" → 5, "Index" → 2]
```
Out[•]=

$\{b\$\$_2, b\$\$_7\}$

In[24]:=
```
SMSRealList[{"b1", "b2"}, b$$[#] &, "Index" → 3]
```
Out[•]=

$\{b\$\$_3, b\$\$_8\}$

In[25]:=
```
SMSRealList[{"b3", "b4"}, b$$[#] &, "Length" → 20, "Index" → 4]
```
Out[•]=

$\{b\$\$_{14}, b\$\$_{34}\}$

- The arguments "Length" is left unevaluated by the use of Hold function in order to be able to retrieve the same array elements through all the *AceGen* session. The actual auxiliary variable assigned to *L* can be different in different subroutines!!

In[26]:=
```
{L, i} ⊨ SMSInteger[{L$$, i$$}];
SMSRealList[{"c1", "c2"}, c$$[#] &, "Length" → Hold[2 L], "Index" → i + 1]
```
Out[•]=

$\{c\$\$_{¥_{13}}, c\$\$_{¥_{13}+¥_{15}}\}$

In[28]:=
```
SMSRealList[Array["β", 2], c$$[#] &, "Length" → Hold[ L], "Index" → i]
```
Out[•]=

$\{c\$\$_{i+¥_{16}}, c\$\$_{i+¥_{19}}\}$

In[29]:=

```
TableForm[{SMSRealList["β"[_], "ID"], SMSRealList["β"[_]], SMSRealList["β"[_], "Plain"],
  SMSRealList["β"[_], "Export"]}, TableHeadings → {{"ID", "AceGen", "Plain", "Export"}, None}]
```

Out[ ]//TableForm=

ID	β[1]	β[2]
AceGen	c$$ $_{i+¥16}$	c$$ $_{i+¥19}$
Plain	c$$[i$$ + 4 L$$]	c$$[i$$ + 5 L$$]
Export	c$$[i + 4 L]	c$$[i + 5 L]

In[30]:=

```
SMSRealList["β"[_], "Length"]
```

Out[ ]=

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_1,exp_2,...,exp_N},{ext_1,ext_2,...,ext_N}]
export the list of expressions {exp_1,exp_2,...,exp_N} to a matching list of the external variables {ext_1,ext_2,...,ext_N}

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

option	default	description
"AddIn"	False	add the value of *exp* to the current value of the external variable *ext*
"Atomic"	False	the "#pragma omp atomic" statement is added to prevent the race condition in the case of parallelization
"Position"	"Current"	Regulates how the "export" object is treated during the code optimisation. "Current" ⇒ the object remains at the position in the program where it was called "Optimal" ⇒ optimal position is determined based on the dependency of *exp* and *ext* on other variables "Automatic" ⇒ "AddIn"=== True ⇒ "Current" "AddIn"=== False ⇒ "Optimal"

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.

In[13]:=

```
<< AceGen`;
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[{1, 2}, a$$];
SMSExport[{3, 4}, {a$$[1], a$$[2]}];
SMSExport[{5, 6}, Table[a$$[i], {i, 1, 2}]];
(* two equivalent forms how to export two-dimensional array*)
SMSExport[Table[Sin[i j], {i, 2}, {j, 2}], r$$];
SMSExport[Table[Sin[i j], {i, 2}, {j, 2}], Table[r$$[i, j], {i, 2}, {j, 2}]];
SMSWrite["test"];
```

File:	test.f	Size: 1058
Methods	No.Formulae	No.Leafs
test	6	40

In[24]:=

```
FilePrint["test.f"]
```

```
!*************************************************************
!* AceGen    3.001 Windows (8 Mar 11)                        *
!*           Co. J. Korelc  2007            13 Mar 11 19:31:04*
!*************************************************************
! User : USER
! Evaluation time                 : 0 s     Mode  : Optimal
! Number of formulae              : 6       Method: Automatic
! Subroutine                      : test size :40
! Total size of Mathematica  code : 40 subexpressions
! Total size of Fortran code      : 484 bytes

!******************* S U B R O U T I N E *********************
      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
```

Automatic Differentiation

Contents

- Theory of Automatic Differentiation
- SMSD
- Differentiation : Mathematica syntax versus AceGen syntax
  - Mathematica
  - AceGen
- Automatic Differentiation Examples
  - 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 Formulae
  - Example 1 : characteristic formulae – one subset
  - Example 2 : characteristic formulae – two subsets
- Exceptions in Differentiation
  - SMSDefineDerivative
  - Exception Type A : Generic Example
  - Exception Type C : Implicit dependencies
  - Exception Type D : Alternative definition of partial derivatives
- Limitations : Incorrect structure of the program

Theory of Automatic Differentiation

Differentiation is 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

For $i = n+1, n+2, .., m$

$$v_i = f_i\left(v_j\right)_{j \epsilon A_i}$$

$y = v_m$

$A_i = \{1, 2, \ldots, i-1\}$

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_i, $i = n+1, n+2, \ldots, m$ a set of auxiliary variables. The goal is to calculate the gradient of y with respect to the set of independent variables $\nabla y = \left\{\frac{\partial y}{\partial v_1}, \frac{\partial y}{\partial v_2}, \ldots, \frac{\partial y}{\partial v_n}\right\}$. 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 ∇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:

$\nabla v_i = \{\delta_{ij}\}_{j=1,2,\ldots,n}$ *for i=1,2,...,n*

For i=n+1,n+2,...,m

$$\nabla v_i = \sum_{j=1}^{i-1} \frac{\partial f_i}{\partial v_j} \nabla v_j$$

$\nabla y = \nabla v_m$

In practical cases gradients ∇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 $\overline{v_i}$ with each auxiliary variable v_i.

$\overline{v_i} = \frac{\partial y}{\partial v_i}$ *for i=m,m-1,...,n*

$\nabla y = \left\{\frac{\partial y}{\partial v_i}\right\} = \{\overline{v_i}\}$ *for i=1,2,...,n*

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

$$\overline{v_i} = \sum_{j=i+1}^{m} \frac{\partial f_j}{\partial v_i} \overline{v_j}$$

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

For i=m,m-1,...,n-1

$$\overline{v_i} = \sum_{j=i+1}^{m} \frac{\partial f_j}{\partial v_i} \overline{v_j}$$

$\nabla y = \{\overline{v_1}, \overline{v_2}, \ldots, \overline{v_n}\}$

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 $\nabla v_3 = \left\{\frac{\partial v_3}{\partial x_i}\right\}$ leads to

$$v_1 = f_1(x_i) \qquad \frac{\partial v_1}{\partial x_i} = \frac{\partial f_1}{\partial x_i} \qquad i = 1, 2, \ldots, n$$

$$v_2 = f_2(x_i, v_1) \qquad \frac{\partial v_2}{\partial x_i} = \frac{\partial f_2}{\partial x_i} + \frac{\partial f_2}{\partial v_1}\frac{\partial v_1}{\partial x_i} \qquad i = 1, 2, \ldots, n$$

$$v_3 = f_3(x_i, v_2, v_3) \qquad \frac{\partial v_3}{\partial x_i} = \frac{\partial f_3}{\partial x_i} + \frac{\partial f_3}{\partial v_1}\frac{\partial v_1}{\partial x_i} + \frac{\partial f_3}{\partial v_2}\frac{\partial v_2}{\partial x_i} \qquad i = 1, 2, \ldots, n.$$

The reverse mode is implemented as follows

$$v_3 = f_3\,(x_i, v_2, v_3) \qquad \bar{v}_3 = \frac{\partial v_3}{\partial v_3} = 1$$
$$v_2 = f_2\,(x_i, v_1) \qquad \bar{v}_2 = \frac{\partial v_3}{\partial v_2} = \frac{\partial f_3}{\partial v_2}\,\bar{v}_3$$
$$v_1 = f_1\,(x_i) \qquad \bar{v}_1 = \frac{\partial v_3}{\partial v_1} = \frac{\partial f_3}{\partial v_1}\,\bar{v}_3 + \frac{\partial f_2}{\partial v_1}\,\bar{v}_2$$
$$x_i \qquad \frac{\partial v_3}{\partial x_i} = \frac{\partial f_3}{\partial x_i}\,\bar{v}_3 + \frac{\partial f_2}{\partial x_i}\,\bar{v}_2 + \frac{\partial f_1}{\partial x_i}\,\bar{v}_1 \qquad i = 1, 2, \ldots, n.$$

By comparing both techniques, it is obvious that the reverse mode leads to a more efficient solution when the number of functions to differentiate is higher than the number of independent variables. The above formulae can be algorithmically implemented in two fundamentally different ways:

A) the standard automatic differentiation tools perform differentiation at program run time using operator overloading capabilities of modern computer languages,

B) in the case of code-to-code implementation derivatives are derived symbolically, program is written in standard computer language and compiled and link together with the rest of the program.

The code-to-code implementation does not require any additional data handling as operator overloading implementation. Consequently, it leads to faster codes. However, it is rarely used due to difficult implementation. The procedure implemented in the *AceGen* system represents a code-to-code transformation version of automatic differentiation technique combined with powerful simultaneous code optimisation. The resulting codes are much more numerically efficient than the codes based on the standard operator overloading automatic differentiation.

AceGen does automatic differentiation by using forward or backward mode of automatic differentiation. 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. *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 for the collection of derivatives of variables which implicitly dependent on differentiation 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.

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`.

KORELC, Jože, Multi-language and Multi-environment Generation of Nonlinear Finite Element Codes, Engineering with Computers, 2002, 18(4):312-327

Korelc J. Automation of primal and sensitivity analysis of transient coupled problems. Computational mechanics, 44(5):631-649 (2009).

SMSD Function

SMSD[*exp*,*v*]
partial derivative $\frac{\partial exp}{\partial v}$

SMSD[*exp*,{v_1,v_2,...}]
gradient of *exp* $\{\frac{\partial exp}{\partial v1}, \frac{\partial exp}{\partial v2}, \ldots\}$

SMSD[{exp_1,exp_2,...},{v_1,v_2,...}]
the Jacobian matrix **J**=$\left[\frac{\partial exp_i}{\partial v_j}\right]$

SMSD[*exp*,{{v_{11},v_{12},...},{v_{21},v_{22},...},...}]

differentiation of scalar with respect to matrix $\left[\frac{\partial exp}{\partial v_{ij}}\right]$

SMSD[*exp*,{v_1,v_2,...}, *index*]
create a characteristic expression for an arbitrary element of the gradient $\{\frac{\partial exp}{\partial v1},\frac{\partial exp}{\partial v2},...\}$ and return an index data object that represents characteristic element of the gradient with the index *index*

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

SMSD[*exp_structure, var_structure*]
differentiation of an arbitrary *exp_structure* with respect to an arbitrary *var_structure*.

The result $\frac{\partial exp_structure}{\partial var_structure}$ has the same global structure as the *exp_structure* with each scalar *exp* replaced by the substructure $\frac{\partial exp}{\partial var_structure}$.

(e.g. derivatives of second order tensors can be generated $D_{i,j,k,l}=\frac{\partial f_{i,j}}{\partial x_{k,l}}$)

Automatic differentiation procedures.

option	default	description
"Constant"-> {v_1,v_2,...}	{}	perform differentiation under assumption that formulas involved do not depend on given variables {v_1,v_2,...} (see also `Exceptions in Differentiation`)
"Constant"->*v*		≡ "Constant"->{*v*}
"Mode"->*AD_mode*	''Automatic''	Method used to perform differentiation: "Forward" ⇒ forward mode of automatic differentiation "Backward" ⇒ backward mode of automatic differentiation "Automatic" ⇒ appropriate AD mode is selected automatically
"Dependency"-> {...,{*v*,*z*,$\frac{\partial v}{\partial z}$},...}	{}	during differentiation assume that derivative of auxiliary variable *v* with respect to auxiliary variable *z* is $\frac{\partial v}{\partial z}$ (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!)
"Symmetric"-> *truefalse*	False	see example below
"Ignore"->*crit*	(False&)	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 "Differentiation with respect to matrix")
"SaveLast"-> *truefalse*	True	True ⇒ the finial results is optimized False ⇒ the final formula is left unoptimized
"SaveAll"-> *truefalse*	True	True ⇒ all partial derivatives are optimized False ⇒ partial derivatives are left unoptimized

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

```
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: $\nabla_{\mathbf{x}} = \frac{\partial f}{\partial x_j}$

```
D[f, {{x, y, z, k}}]
```

$$\{1, 2, 3, 4\}$$

- Jacobian: $J_{i,j} = \frac{\partial f_i}{\partial x_j}$

```
D[{f, f^2}, {{x, y}}] // MatrixForm
```

$$\begin{pmatrix} 1 & 2 \\ 2\,(4\,k + x + 2\,y + 3\,z) & 4\,(4\,k + x + 2\,y + 3\,z) \end{pmatrix}$$

- Derivatives of second order tensors: $D_{i,j,k,l} = \frac{\partial f_{i,j}}{\partial x_{k,l}}$

```
D[{{f, f^2}, {f^3, f^4}}, {{{x, y}, {z, k}}}] // MatrixForm
```

$$\begin{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} & \begin{pmatrix} 2\,(4\,k + x + 2\,y + 3\,z) & 4\,(4\,k + x + 2\,y + 3\,z) \\ 6\,(4\,k + x + 2\,y + 3\,z) & 8\,(4\,k + x + 2\,y + 3\,z) \end{pmatrix} \\ \begin{pmatrix} 3\,(4\,k + x + 2\,y + 3\,z)^2 & 6\,(4\,k + x + 2\,y + 3\,z)^2 \\ 9\,(4\,k + x + 2\,y + 3\,z)^2 & 12\,(4\,k + x + 2\,y + 3\,z)^2 \end{pmatrix} & \begin{pmatrix} 4\,(4\,k + x + 2\,y + 3\,z)^3 & 8\,(4\,k + x + 2\,y + 3\,z)^3 \\ 12\,(4\,k + x + 2\,y + 3\,z)^3 & 16\,(4\,k + x + 2\,y + 3\,z)^3 \end{pmatrix} \end{pmatrix}$$

AceGen

```
<< AceGen`;
SMSInitialize["test", "Language" → "C"];
SMSModule["test", Real[x$$, y$$, z$$, k$$]];
{x, y, z, k} ⊢ SMSReal[{x$$, y$$, z$$, k$$}];
f ⊨ x + 2 y + 3 z + 4 k;
```

- Partial derivative: $\frac{\partial f}{\partial x}$

```
dx ⊨ SMSD[f, x]
```

$$1$$

- Gradient: $\nabla_{\mathbf{x}} = \frac{\partial f}{\partial x_j}$

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

```
∇x ⊨ SMSD[f, {x, y, z}]
```

$$\{1, 2, 3\}$$

- Jacobian: $J_{i,j} = \frac{\partial f_i}{\partial x_j}$

```
Jx ⊨ SMSD[{f, f^2}, {x, y}]
```

$$\{\{1, 2\}, \{Jx_{2,1}, Jx_{2,2}\}\}$$

```
SMSRestore[Jx, "Global"] // MatrixForm
```

$$\begin{pmatrix} 1 & 2 \\ 2\,(4\,k + x + 2\,y + 3\,z) & 4\,(4\,k + x + 2\,y + 3\,z) \end{pmatrix}$$

- Derivatives of second order tensors: $D_{i,j,k,l} = \frac{\partial f_{i,j}}{\partial x_{k,l}}$

```
Dx ⊨ SMSD[{{f, f^2}, {f^3, f^4}}, {{x, y}, {z, k}}]
```

$$\{\{\{\{1, 2\}, \{3, 4\}\}, \{\{Jx_{2,1}, Jx_{2,2}\}, \{Dx_{1,2,2,1}, Dx_{1,2,2,2}\}\}\}, \{\{\{Dx_{2,1,1,1}, Dx_{2,1,1,2}\}, \{Dx_{2,1,2,1}, Dx_{2,1,2,2}\}\}, \{\{Dx_{2,2,1,1}, Dx_{2,2,1,2}\}, \{Dx_{2,2,2,1}, Dx_{2,2,2,2}\}\}\}\}$$

```
SMSRestore[Dx, "Global"] // MatrixForm
```

$$\begin{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} & \begin{pmatrix} 2\,(4\,k + x + 2\,y + 3\,z) & 4\,(4\,k + x + 2\,y + 3\,z) \\ 6\,(4\,k + x + 2\,y + 3\,z) & 8\,(4\,k + x + 2\,y + 3\,z) \end{pmatrix} \\ \begin{pmatrix} 3\,(4\,k + x + 2\,y + 3\,z)^2 & 6\,(4\,k + x + 2\,y + 3\,z)^2 \\ 9\,(4\,k + x + 2\,y + 3\,z)^2 & 12\,(4\,k + x + 2\,y + 3\,z)^2 \end{pmatrix} & \begin{pmatrix} 4\,(4\,k + x + 2\,y + 3\,z)^3 & 8\,(4\,k + x + 2\,y + 3\,z)^3 \\ 12\,(4\,k + x + 2\,y + 3\,z)^3 & 16\,(4\,k + x + 2\,y + 3\,z)^3 \end{pmatrix} \end{pmatrix}$$

See also 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.

$z(x) = 3\,x^2 + 2\,y + \mathrm{Log}[y]$.

$y(x) = \mathrm{Sin}[x^2]$.

```
<< AceGen`;
SMSInitialize["test", "Language" → "C"];
SMSModule["test", Real[x$$, r$$]];
x ⊨ SMSReal[x$$];
y ⊨ Sin[x^2];
z ⊨ 3 x^2 + 2 y + Log[y];
```

- Here the derivative of z with respect to x is calculated.

```
zx ⊨ SMSD[z, x];
```

```
SMSExport[zx, r$$];
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"

/******************* S U B R O U T I N E *********************/
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 $f(x) = 3\,z^2$ with respect to x, where z is

$$z(x) = \begin{cases} x^2 + 2\,y + \mathrm{Log}[y] & x > 0 \\ \mathrm{Cos}[x^3] & x \le 0 \end{cases},$$

and y is $y = \mathrm{Sin}[x^2]$.

```
<< AceGen`;
SMSInitialize["test", "Language" → "C"];
SMSModule["test", Real[x$$, r$$]];
x ⊢ SMSReal[x$$];
z ⊨ SMSIf[x > 0
    , y ⊨ Sin[x^2];
    3 x^2 + 2 y + Log[y]
    , Cos[x^3]
   ];
fx ⊨ SMSD[3 z^2, x];
SMSExport[fx, r$$];
SMSWrite[];
FilePrint["test.c"]
```

```
 File:     test.c    Size: 1018
Methods No.Formulae  No.Leafs
 test       11         88

/*************************************************************
* AceGen    5.001 Windows (3 Jan 13)                        *
*           Co. J. Korelc  2007            3 Jan 13 13:53:07 *
*************************************************************
User     : USER
Notebook : AceGenTutorials.nb
Evaluation time                 : 0 s     Mode  : Optimal
Number of formulae              : 11      Method: Automatic
Subroutine                      : test size :88
Total size of Mathematica  code : 88 subexpressions
Total size of C code            : 407 bytes*/
#include "sms.h"

/******************* S U B R O U T I N E *********************/
void test(double v[5001],double (*x),double (*r))
{
int b2;
v[13]=Power((*x),2);
v[16]=3e0*v[13];
if((*x)>0e0){
 v[14]=2e0*(*x);
 v[7]=v[14]*cos(v[13]);
 v[3]=sin(v[13]);
 v[8]=3e0*v[14]+(2e0+1e0/v[3])*v[7];
 v[5]=v[16]+2e0*v[3]+log(v[3]);
} else {
 v[15]=Power((*x),3);
 v[8]=-(v[16]*sin(v[15]));
 v[5]=cos(v[15]);
};
(*r)=6e0*v[5]*v[8];
};
```

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

$$\boldsymbol{M} \vDash \begin{pmatrix} v_{11} & v_{12} & \dots \\ v_{12} & v_{22} & \dots \\ \dots & \dots & \dots \end{pmatrix}$$

then we have two possibilities to make proper differentiation:

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

M𝔽⊨SMSFreeze[**M**];

*δ*exp⊨SMSD[exp(**M**𝔽),**M**𝔽];

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

$$\delta\text{exp} \vDash \text{SMSD[exp(}\mathbf{M},\mathbf{M},\text{"Symmetric"->True]} \equiv \begin{pmatrix} 1 & \frac{1}{2} & \dots \\ \frac{1}{2} & 1 & \dots \\ \dots & \dots & \dots \end{pmatrix} * \begin{pmatrix} \frac{\partial exp}{\partial v_{11}} & \frac{\partial exp}{\partial v_{12}} & \dots \\ \frac{\partial exp}{\partial v_{12}} & \frac{\partial exp}{\partial v_{22}} & \dots \\ \dots & \dots & \dots \end{pmatrix}$$

Example:

Lets have matrix $\mathrm{M} = \begin{pmatrix} a & b \\ b & c \end{pmatrix}$ and functions $f_{iso} = \det(\mathbf{M})$ and $f_{general} = M_{1,1}{}^2 + 5\,M_{1,2} - \mathrm{Sin}\,(M_{2,1}) - 3\,M_{2,2}$, thus

$\begin{pmatrix} & \\ & \end{pmatrix}$ $\begin{pmatrix} & \\ & \end{pmatrix}$

$$\frac{\partial f_{iso}}{\partial \mathbf{M}} = \begin{pmatrix} a & -b \\ -b & c \end{pmatrix} \text{ and } \frac{\partial f_{general}}{\partial \mathbf{M}} = \begin{pmatrix} 2a & 5 \\ -\mathrm{Cos}(b) & -3 \end{pmatrix}.$$

```
<< AceGen`;
SMSInitialize["test"];
SMSModule["test", Real[a$$, b$$, c$$]];
{a, b, c} ⊢ SMSReal[{a$$, b$$, c$$}];

M ⊨ {{a, b}, {b, c}};
fiso = Det[M];
fgeneral = M[[1, 1]]^2 + 5 M[[1, 2]] - Sin[M[[2, 1]]] - 3 M[[2, 2]];
```

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

```
SMSD[fiso, M] // MatrixForm
```

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

$$\begin{pmatrix} c & -2b \\ -2b & a \end{pmatrix}$$

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

```
SMSD[fiso, M, "Symmetric" → True] // MatrixForm
```

$$\begin{pmatrix} c & -b \\ -b & a \end{pmatrix}$$

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

```
SMSD[{fiso, Sin[fiso]}, M, "Symmetric" → True] // MatrixForm
```

$$\begin{pmatrix} \begin{pmatrix} c \\ -b \end{pmatrix} & \begin{pmatrix} -b \\ a \end{pmatrix} \\ \begin{pmatrix} \mathrm{Cos}\left[b^2 - a\,c\right] c \\ -\mathrm{Cos}\left[b^2 - a\,c\right] b \end{pmatrix} & \begin{pmatrix} -\mathrm{Cos}\left[b^2 - a\,c\right] b \\ \mathrm{Cos}\left[b^2 - a\,c\right] a \end{pmatrix} \end{pmatrix}$$

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

```
SMSD[fgeneral, M, "Symmetric" → True] // MatrixForm
```

$$\begin{pmatrix} 2a & \frac{1}{2}\left(5 - \mathrm{Cos}\left[b\right]\right) \\ \frac{1}{2}\left(5 - \mathrm{Cos}\left[b\right]\right) & -3 \end{pmatrix}$$

- 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.

```
M ⊨ SMSFreeze[M];
fiso = Det[M];
fgeneral = M[[1, 1]]^2 + 5 M[[1, 2]] - Sin[M[[2, 1]]] - 3 M[[2, 2]];
```

```
SMSD[fiso, M] // MatrixForm
```

$$\begin{pmatrix} M_{2,2} & -M_{2,1} \\ -M_{1,2} & M_{1,1} \end{pmatrix}$$

```
SMSD[fgeneral, M] // MatrixForm
```

$$\begin{pmatrix} 2\,M_{1,1} & 5 \\ -\mathrm{Cos}\left[M_{2,1}\right] & -3 \end{pmatrix}$$

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.

$$\mathrm{SMSD}[exp, \begin{pmatrix} v_{11} & v_{12} & \dots \\ v_{21} & 0 & \dots \\ \dots & \dots & \dots \end{pmatrix}, \text{“Ignore”->NumberQ}] \equiv \begin{pmatrix} \frac{\partial exp}{\partial v_{11}} & \frac{\partial exp}{\partial v_{12}} & \dots \\ \frac{\partial exp}{\partial v_{12}} & 0 & \dots \\ \dots & \dots & \dots \end{pmatrix}$$

```
<< AceGen`;
SMSInitialize["test"];
SMSModule["test", Real[a$$, b$$, c$$]];
{a, b, c} ⊢ SMSReal[{a$$, b$$, c$$}];

x ⊨ {a, b, c};
f = a + 2 b + 3 c;

SMSD[f, {a, b, 5}, "Ignore" → NumberQ]
```

```
{1, 2, 0}
```

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

```
SMSD[f, {a, b, 5}]
```

```
Syntax error in differentiation.
  Independent variables have to be true variables.
Module: test Description: a b 5
Events: 0
Version: 5.001 Windows (3 Jan 13) (MMA 9.) Module: SMSD
See also: SMSD  AceGen Troubleshooting  Continue
SMC::Fatal : System cannot proceed with the evaluation due to the fatal error in SMSD .
$Aborted
```

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 $w = x^2 + 1$.

$$w = x^2 + 1$$

$$\frac{\partial \mathrm{Sin}(w)}{\partial w}$$

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

```
<< AceGen`;
SMSInitialize["test", "Language" → "C"];
SMSModule["test", Real[x$$]];
x ⊢ SMSReal[x$$];
w ⊨ x^2 + 1;
SMSD[Sin[w], w]
```

Differentiation variables do not have unique signature. They should be introduced by SMSReal, SMSInteger, SMSFreeze or SMSFictive statement.

Module: test **Description:** {w, $V[2, 1]}

Events: 0

Version: 5.001 Windows (3 Jan 13) (MMA 9.) **Module:** SMSD-1

See also: SMSD AceGen Troubleshooting Continue

SMC::Fatal : System cannot proceed with the evaluation due to the fatal error in SMSD-1 .

$Aborted

- SMSFreeze creates unique signature for the intermediate auxiliary variable.

```
<< AceGen`;
SMSInitialize["test"];
SMSModule["test", Real[x$$]];
x ⊢ SMSReal[x$$];
w ⊢ SMSFreeze[x^2 + 1];
SMSD[Sin[w], w]
```

Cos[w]

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 $N_{d.o.f}$ unknown parameters are used in our numerical procedure, then an explicit form of the gradient and the Hessian will have at least $N_{d.o.f} + (N_{d.o.f})^2$ 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 $\mathbf{a_i}$, is defined by

$$\mathbf{a_i} \subset \mathbf{a}$$

$$\bigcup_{i=1}^{L} \mathbf{a}_i = \mathbf{a}$$

$$\mathbf{a_i} \cap \mathbf{a_j} = \phi \quad , i \neq j.$$

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

$$\frac{\partial f}{\partial \mathbf{a}} = \left\{\frac{\partial f}{\partial \mathbf{a}_1}, \frac{\partial f}{\partial \mathbf{a}_2}, \cdots \frac{\partial f}{\partial \mathbf{a}_L}\right\}$$

Let $\bar{a}_i$ be an arbitrary element of the i-th subset. At the evaluation time of the program, the actual index of an arbitrary element $\bar{a}_i$ becomes known. Thus, $\bar{a}_{ij}$ 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

$\frac{\partial f}{\partial \bar{\mathbf{a}}_{\mathbf{ij}}} = \text{SMSD}[f, \mathbf{a}_{\mathbf{i}}, j]$.

Let $\mathbf{a}_{\mathbf{kl}}$ 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

$$\frac{\partial^2 f}{\partial \bar{\mathbf{a}}_{\mathbf{ij}} \partial \bar{\mathbf{a}}_{\mathbf{kl}}} = \text{SMSD}\left[\frac{\partial f}{\partial \bar{\mathbf{a}}_{\mathbf{ij}}}, \mathbf{a}_{\mathbf{k}}, l\right]$$

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.

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

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`;
SMSInitialize["test", "Language" → "Fortran"]
SMSModule["Test", Real[u$$[3], x$$, L$$, g$$[3]]];
{x, L} ⊢ {SMSReal[x$$], SMSReal[L$$]};
ui ⊢ SMSReal[Table[u$$[i], {i, 3}]];
Ni ⊨ {x/L, 1 - x/L, x/L (1 - x/L)};
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 ui is calculated.

```
fui ⊨ SMSD[f, ui, i];

SMSExport[fui, g$$[i]];
SMSEndDo[];
SMSWrite[];
```

```
Method : Test 6 formulae, 95 sub-expressions
[1] File created : test.f Size : 1011
```

```
FilePrint["test.f"]
```

```
!*************************************************************
!* AceGen    2.103 Windows (17 Jul 08)                       *
!*           Co. J. Korelc  2007            18 Jul 08 00:58:38*
!*************************************************************
! User : USER
! Evaluation time                 : 1 s     Mode  : Optimal
! Number of formulae              : 6       Method: Automatic
! Subroutine                      : Test size :95
! Total size of Mathematica  code : 95 subexpressions
! Total size of Fortran code      : 441 bytes

!******************* S U B R O U T I N E *********************
      SUBROUTINE Test(v,u,x,L,g)
      IMPLICIT NONE
      include 'sms.h'
      INTEGER i11
      DOUBLE PRECISION v(5011),u(3),x,L,g(3)
      v(6)=x/L
      v(7)=1d0-v(6)
      v(8)=v(6)*v(7)
      v(9)=u(1)*v(6)+u(2)*v(7)+u(3)*v(8)
      v(5007)=v(6)*v(9)
      v(5008)=v(7)*v(9)
      v(5009)=v(8)*v(9)
      DO i11=1,3
       g(i11)=2d0*v(5006+i11)
      ENDDO
      END
```

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,

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

with respect to unknown parameters u_i and v_i, where

$u = \sum_{i=1}^{4} N_i\, u_i$
$u = \sum_{i=1}^{4} N_i\, v_i$
and
$N = \{(1-X)(1-Y), (1+X)(1-Y), (1+X)(1+Y), (1-X)(1+Y)\}.$

We make two subsets $\mathbf{u_i}$ and $\mathbf{v}_i$ of the set of independent variables $\mathbf{a_i}$.

$\mathbf{a_i}=\{u_1, v_1, u_2, v_2, u_3, v_3, u_4, v_4\}$

$\mathbf{u_i} = \{u_1, u_2, u_3, u_4\}$, $\mathbf{v}_i = \{v_1, v_2, v_3, v_4\}$

```
<< AceGen`;
SMSInitialize["test", "Language" → "C"] ×
 SMSModule["Test", Real[ul$$[4], vl$$[4], X$$, Y$$, g$$[8], H$$[8, 8]]];
{X, Y} ⊢ {SMSReal[X$$], SMSReal[Y$$]};
ui ⊢ SMSReal[Table[ul$$[i], {i, 4}]];
vi ⊢ SMSReal[Table[vl$$[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 + u v;
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 2*2 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"]
```

File:	test.c	Size: 1508
Methods	No.Formulae	No.Leafs
Test	19	258

```
/*************************************************************
* AceGen    2.502 Windows (18 Nov 10)                        *
*           Co. J. Korelc  2007            24 Nov 10 13:29:27*
*************************************************************
User : USER
Evaluation time                 : 1 s     Mode  : Optimal
Number of formulae              : 19      Method: Automatic
Subroutine                      : Test size :258
Total size of Mathematica  code : 258 subexpressions
Total size of C code            : 913 bytes*/
#include "sms.h"

/******************* S U B R O U T I N E *********************/
void Test(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[17]=1e0+(*Y);
v[12]=1e0-(*Y);
v[11]=v[12]*v[16];
v[13]=v[12]*v[14];
v[15]=v[14]*v[17];
v[18]=v[16]*v[17];
v[5012]=v[11];
v[5013]=v[13];
v[5014]=v[15];
v[5015]=v[18];
v[19]=ul[0]*v[11]+ul[1]*v[13]+ul[2]*v[15]+ul[3]*v[18];
v[20]=v[11]*vl[0]+v[13]*vl[1]+v[15]*vl[2]+v[18]*vl[3];
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[(-2+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[(-2+2*i22)][(-2+2*i31)]=v[37];
  H[(-2+2*i22)][(-1+2*i31)]=v[39];
  H[(-1+2*i22)][(-2+2*i31)]=v[39];
  H[(-1+2*i22)][(-1+2*i31)]=v[37];
 };/* end for */
};/* end for */
};
```

Exceptions in Differentiation

The *SMSDefineDerivative* function should be used cautiously since derivatives are defined permanently and globally. The "Dependency" option of the `SMSFreeze`, `SMSReal` 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:

- **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 follows 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**.

Type	Local AD exception	Schematic *AceGen* input

A	$\nabla f_A := \frac{\hat{\delta} f(\mathbf{a},\mathbf{b}(\mathbf{a}))}{\hat{\delta}\mathbf{a}}\Big\vert_{\frac{D\mathbf{b}}{D\mathbf{a}}=\mathbf{M}}$	`a▸SMSReal[a$$]`
		`b▸SMSFreeze[G[a]]`
		`∇fA▸SMSD[f[a,b],a,"Dependency"→{b,a,M}]`

B	$\nabla f_B := \frac{\hat{\delta} f(\mathbf{a},\mathbf{b}(\mathbf{a}))}{\hat{\delta}\mathbf{a}}\Big\vert_{\frac{D\mathbf{b}}{D\mathbf{a}}=\mathbf{0}}$	`a▸SMSReal[a$$]`
		`b▸SMSFreeze[G[a]]`
		`∇fB▸SMSD[f[a,b],a,"Constant"→b]`

C	$\nabla f_C := \frac{\hat{\delta} f(\mathbf{a},\mathbf{b})}{\hat{\delta}\mathbf{a}}\Big\vert_{\frac{D\mathbf{b}}{D\mathbf{a}}=\mathbf{M}}$	`a▸SMSReal[a$$]`
		`b▸SMSReal[b$$]`
		`∇fC▸SMSD[f[a,b],a,"Dependency"→{b,a,M}]`

D	$\nabla f_D := \frac{\hat{\delta} f(\mathbf{a},\mathbf{b}(\mathbf{c}(\mathbf{a})))}{\hat{\delta}\mathbf{a}}\Big\vert_{\frac{D\mathbf{b}}{D\mathbf{c}}=\mathbf{M}}$	`a▸SMSReal[a$$]`
		`c▸SMSFreeze[H[a]]`
		`b▸SMSFreeze[G[c]]`
		`∇fD▸SMSD[f[a,b],a,"Dependency"→{b,c,M}]`

Type	Global AD exception	Schematic *AceGen* input

A	$\mathbf{b} := \mathbf{G}(\mathbf{a})\vert_{\frac{D\mathbf{b}}{D\mathbf{a}}=\mathbf{M}(\mathbf{a})}$
	$\nabla f_A := \frac{\hat{\delta} f(\mathbf{a},\mathbf{b}(\mathbf{a}))}{\hat{\delta}\mathbf{a}}$
		`a▸SMSReal[a$$]`
		`b▸SMSFreeze[G[a],"Dependency"→{a,M}]`
		`∇fA▸SMSD[f[a,b],a]`

B	$\mathbf{b} := \mathbf{G}(\mathbf{a})\vert_{\frac{D\mathbf{b}}{D\mathbf{a}}=\mathbf{0}}$
	$\nabla f_B := \frac{\hat{\delta} f(\mathbf{a},\mathbf{b}(\mathbf{a}))}{\hat{\delta}\mathbf{a}}$
		`a▸SMSReal[a$$]`
		`b▸SMSFreeze[G[a],"Dependency"→{a,0}]`
		`∇fB▸SMSD[f[a,b],a]`

C	$\mathbf{b} := \mathbf{G}\vert_{\frac{D\mathbf{b}}{D\mathbf{a}}=\mathbf{M}}$
	$\nabla f_C := \frac{\hat{\delta} f(\mathbf{a},\mathbf{b})}{\hat{\delta}\mathbf{a}}$
		`a▸SMSReal[a$$]`
		`b▸SMSReal[b$$,"Dependency"→{a,M}]`
		`∇fC▸SMSD[f[a,b],a]`

D	$\mathbf{c} := \mathbf{H}(\mathbf{a})$
	$\mathbf{b} := \mathbf{G}(\mathbf{c})\vert_{\frac{D\mathbf{b}}{D\mathbf{c}}=\mathbf{M}}$
	$\nabla f_D := \frac{\hat{\delta} f(\mathbf{a},\mathbf{b}(\mathbf{c}(\mathbf{a})))}{\hat{\delta}\mathbf{a}}$
		`a▸SMSReal[a$$]`
		`c▸SMSFreeze[H[a]]`
		`b▸SMSFreeze[G[c],"Dependency"→{c,M}]`
		`∇fD▸SMSD[f[a,b],a]`

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, "Dependency" → {{p_1, $\frac{\partial exp}{\partial p_1}$}, {$p_2$, $\frac{\partial exp}{\partial p_2}$}, ..., {$p_n$, $\frac{\partial exp}{\partial p_n}$}}] the true dependencies of the *exp* are ignored and it is assumed that *exp* depends on auxiliary variables $p_1, \ldots, p_n$. Partial derivatives of *exp* with respect to auxiliary variables $p_1, \ldots, p_n$ are then taken to be $\frac{\partial exp}{\partial p_1}, \frac{\partial exp}{\partial p_2}, \ldots, \frac{\partial exp}{\partial p_n}$.

SMSDefineDerivative

SMSDefineDerivative[*v*,*z*,*exp*]
define the derivative of auxiliary variable *v* with respect to auxiliary variable *z* to be *exp*

$\frac{\partial v}{\partial z}$:=*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*:={$\frac{\partial v}{\partial z_i}$} ...*i*=1,2,...,*N* and set $\frac{\partial z_i}{\partial z_j}=\delta^i_j$

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*:=$\left[\frac{\partial v_i}{\partial z_j}\right]$ ...*i*=1,2,...,*M*; *j*=1,2,...,*N*, and set $\frac{\partial z_i}{\partial z_j}=\delta^i_j$

Exception Type A: Generic Example

Lets take a function *f*=Sin(*w*) where *w* is intermediate auxiliary variable defined as $w = a\,x^2 + 1$ and evaluate the following derivative

$\frac{\partial f}{\partial x}\Big|_{\frac{Dw}{Dx}=5} = 5\,\mathrm{Cos}(w)$

- Local definition of AD exception

```
<< AceGen`;
SMSInitialize["test"];
SMSModule["test", Real[x$$, a$$]];
x ⊢ SMSReal[x$$]; a ⊢ SMSReal[a$$];
w ⊢ SMSFreeze[a x^2 + 1];
f ⊢ Sin[w];
SMSRestore[SMSD[f, x, "Dependency" → {w, x, 5}], "Global"]
```

5 Cos[w]

- Global definition of AD exception with SMSFreeze

```
<< AceGen`;
SMSInitialize["test"];
SMSModule["test", Real[x$$, a$$]];
x ⊢ SMSReal[x$$]; a ⊢ SMSReal[a$$];
w ⊢ SMSFreeze[a x^2 + 1, "Dependency" → {x, 5}];
f ⊢ Sin[w];
SMSRestore[SMSD[f, x], "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.

```
SMSD[f, a]
```

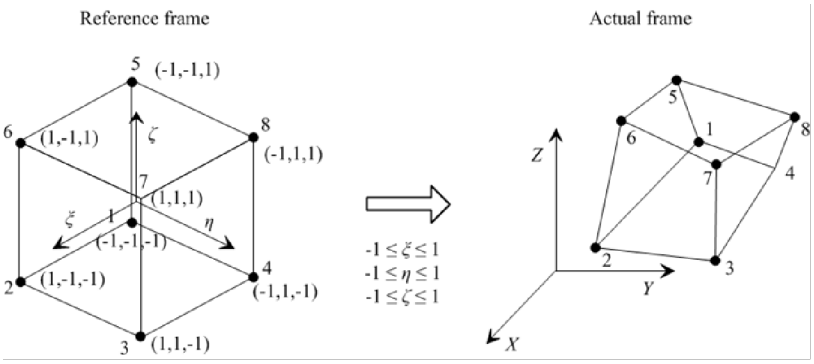
0

Exception Type C: Implicit dependencies

The generation of the subroutine that calculates displacement gradient Dg defined by

$\xi = \{\xi, \eta, \zeta\}$	reference coordinates
$\mathbf{X}(\xi) = \sum_k N(\xi)_k \mathbf{X}_k$	actual coordinates
$\mathbf{u}(\xi) = \sum_k N(\xi)_k \mathbf{u}_k$	displacements
$\mathbf{D} = \frac{\partial \mathbf{u}}{\partial \mathbf{X}}$	displacement gradient

where $N_k = 1/8\,(1 + \xi\,\xi_k)\,(1 + \eta\,\eta_k)\,(1 + \zeta\,\zeta_k)$ 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$$]];
Ξ = {ξ, η, ζ} ⊢ Table[SMSReal[es$$["IntPoints", i, Ig]], {i, 3}];
XI ⊢ Table[SMSReal[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 + ξ Ξn[[i, 1]]) (1 + η Ξn[[i, 2]]) (1 + ζ Ξn[[i, 3]]), {i, 1, 8}];
```

- Coordinates $\mathbf{X} = \{X_g, Y_g, Z_g\}$ 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]
```

$$\left\{\left\{Jg_{1,1}, Jg_{1,2}, Jg_{1,3}\right\}, \left\{Jg_{2,1}, Jg_{2,2}, Jg_{2,3}\right\}, \left\{Jg_{3,1}, Jg_{3,2}, Jg_{3,3}\right\}\right\}$$

- 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.

C $\nabla f_C := \frac{\delta f(\mathbf{a},\mathbf{b})}{\delta \mathbf{a}}\Big|_{\frac{D\mathbf{b}}{D\mathbf{a}}=\mathbf{M}}$

```
a ⊢ SMSReal[a$$]
b ⊢ SMSReal[b$$]
∇fC ⊢ SMSD[f[a,b],a,"Dependency"→{b,a,M}]
```

```
SMSD[u, X, "Dependency" → {Ξ, X, Simplify[SMSInverse[Jg]]}]
```

$\{\{u_{1;X_1}, u_{1;X_2}, u_{1;X_3}\}, \{u_{2;X_1}, u_{2;X_2}, u_{2;X_3}\}, \{u_{3;X_1}, u_{3;X_2}, u_{3;X_3}\}\}$

Exception Type D: Alternative definition of partial derivatives

The generation of the *FORTRAN* subroutine calculates the derivative of function $f = \frac{\mathrm{Sin}(2\,\alpha^2)}{\alpha}$ where $\alpha = \mathrm{Cos}(x)$ 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:

$$f := \begin{cases} \frac{\mathrm{Sin}(2\,\alpha^2)}{\alpha} & \alpha \neq 0 \\ \lim\limits_{\alpha\to 0} \frac{\mathrm{Sin}(2\,\alpha^2)}{\alpha} & \alpha = 0 \end{cases}, \qquad \frac{\partial f}{\partial \alpha} := \begin{cases} \frac{\partial}{\partial \alpha}\left(\frac{\mathrm{Sin}(2\,\alpha^2)}{\alpha}\right) & \alpha \neq 0 \\ \lim\limits_{\alpha\to 0} \frac{\partial}{\partial \alpha}\left(\frac{\mathrm{Sin}(2\,\alpha^2)}{\alpha}\right) & \alpha = 0 \end{cases}.$$

```
<< 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 α^2] / α
    , SMSFreeze[Limit[Sin[2 α^2] / α, α → 0],
     "Dependency" → {{α, Limit[D[Sin[2 α^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"]
```

```
!*************************************************************
!* AceGen    5.001 Windows (3 Jan 13)                        *
!*           Co. J. Korelc  2007            3 Jan 13 13:53:47 *
!*************************************************************
! User     : USER
! Notebook : AceGenTutorials.nb
! Evaluation time                 : 0 s     Mode  : Optimal
! Number of formulae              : 7       Method: Automatic
! Subroutine                      : Test size :54
! Total size of Mathematica  code : 54 subexpressions
! Total size of Fortran code      : 448 bytes

!******************* 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.

```
SMSIf[x > 0];
 f ⇐ Sin[x];
  ...
SMSElse[];
 f ← x^2;
 fx ⇐ SMSD[f, x];
  ...
SMSEndIf[f];
```

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.*

```
SMSIf[x > 0];
 f ⫤ Sin[x];
SMSElse[];
 tmp ⊨ x^2;
 fx ⊨ SMSD[tmp, x];
 f ⫞ tmp;
SMSEndIf[];
```

If the differentiation is placed outside the "If" construct, both instances of *f* are considered for the differentiation.

```
SMSIf[x > 0];
 f ⫤ Sin[x];
SMSElse[];
 f ⫞ x^2;
SMSEndIf[];
fx ⊨ SMSD[f, x];
```

If *f* does not appear outside the "If" construct, then *f* should be defined as a single-valued variable (*f*⊨...) and not as multi-valued variable (*f*⫤...). 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*.

```
SMSIf[x > 0];
 f ⊨ Sin[x];
SMSElse[];
 f ⊨ x^2;
 fx ⊨ SMSD[f, x];
SMSEndIf[];
```

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).

In[125]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "Mathematica", "Mode" → "Debug"];
SMSModule["Test", Real[u$$[3], x$$, L$$, g$$[3]]];
{x, L} ⊢ {SMSReal[x$$], SMSReal[L$$]};
ui ⊢ Table[SMSReal[u$$[i]], {i, 3}];
Ni ⊨ {x/L, 1 - x/L, x/L (1 - x/L)};
u ⊨ Ni.ui;
f ⊨ u^2;
g ⊨ SMSD[f, ui];
SMSExport[g, g$$];
SMSWrite[];
```

```
time=0  variable= 0 ≡ {}

[0]  Consistency check - global

[0]  Consistency check - expressions

[0]  Generate source code :

Events: 0

[0] Final formating
```

File:	test.m	Size: 1383
Methods	No.Formulae	No.Leafs
Test	16	117

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.

In[136]:=

```
u
```

Out[ ]=

```
u
```

In the case of more complex code, the code profile can be explored (Open code profile) 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.

(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.

(Dialog) In[410]:=

```
xv = π; Lv = 10.; uv = {0., 1., 7.}; gv = {Null, Null, Null};
```

- Here the generated code is used to calculate gradient for the numerical test example.

(Dialog) In[411]:=

```
Test[uv, xv, Lv, gv]
```

- Here the numerical results are displayed.

(Dialog) In[412]:=

```
gv
```

(Dialog) Out[ ]=

```
{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.

(Dialog) In[413]:=

```
xv = π // 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.

(Dialog) In[414]:=

```
Test[uv, xv, Lv, gv]
```

- Here the partially evaluated gradient is displayed.

(Dialog) In[415]:=

```
gv // Expand
```

(Dialog) Out[ ]=

$$\left\{-\frac{434.088}{Lv^3}+\frac{118.435}{Lv^2}+\frac{6.28319}{Lv},\ 2.+\frac{434.088}{Lv^3}-\frac{256.61}{Lv^2}+\frac{31.4159}{Lv},\ \frac{1363.73}{Lv^4}-\frac{806.163}{Lv^3}+\frac{98.696}{Lv^2}+\frac{6.28319}{Lv}\right\}$$

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*.

(Dialog) In[416]:=

```
Clear[x, L, up, g];
{x, L} = {x, L};
ui = Array[up, 3];
Ni = {x / L, 1 - x / L, x / L (1 - x / L)};
u = Ni.ui;
f = u^2;
g = Map[D[f, #] &, ui] // Simplify
```

(Dialog) Out[ ]=

$$\left\{\frac{2\,x\,\left(L^2\,up[2]-x^2\,up[3]+L\,x\,(up[1]-up[2]+up[3])\right)}{L^3},\ \frac{2\,(L-x)\,\left(L^2\,up[2]-x^2\,up[3]+L\,x\,(up[1]-up[2]+up[3])\right)}{L^3},\ \frac{2\,(L-x)\,x\,\left(L^2\,up[2]-x^2\,up[3]+L\,x\,(up[1]-up[2]+up[3])\right)}{L^4}\right\}$$

- Here the numerical results are calculated and displayed for the same numerical example as before. We can se that we get the same results.

(Dialog) In[423]:=

```
x = π; L = 10; up[1] = 0; up[2] = 1; up[3] = 7.;
g
```

(Dialog) Out[ ]=

{1.37858, 3.00958, 0.945489}

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.

(Dialog) In[425]:=

```
FilePrint["test.m"]
```

```
(*************************************************************
* AceGen    2.103 Windows (17 Jul 08)                        *
*           Co. J. Korelc  2007            17 Jul 08 22:41:00*
*************************************************************
User : USER
Evaluation time                 : 0 s     Mode  : Debug
Number of formulae              : 16      Method: Automatic
Module                          : Test size : 117
Total size of Mathematica  code : 117 subexpressions       *)
(************************ M O D U L E **************************)
SetAttributes[Test,HoldAll];
Test[u$$_,x$$_,L$$_,g$$_]:=Module[{},
SMSExecuteBreakPoint["1","test",1,1];
$VV[1]=0;(*debug*)
(*2= x *)
$VV[2]=x$$;
(*3= L *)
$VV[3]=L$$;
(*4= ui_1 *)
$VV[4]=u$$[[1]];
(*5= ui_2 *)
$VV[5]=u$$[[2]];
(*6= ui_3 *)
$VV[6]=u$$[[3]];
(*7= Ni_1 *)
$VV[7]=$VV[2]/$VV[3];
(*8= Ni_2 *)
$VV[8]=1-$VV[7];
(*9= Ni_3 *)
$VV[9]=($VV[2]*$VV[8])/$VV[3];
(*10= u *)
$VV[10]=$VV[4]*$VV[7]+$VV[5]*$VV[8]+$VV[6]*$VV[9];
(*11= f *)
$VV[11]=$VV[10]^2;
(*12= [g_1][f_;ui_1] *)
$VV[12]=2*$VV[7]*$VV[10];
(*13= [g_2][f_;ui_2] *)
$VV[13]=2*$VV[8]*$VV[10];
(*14= [g_3][f_;ui_3] *)
$VV[14]=2*$VV[9]*$VV[10];
g$$[[1]]=$VV[12];
g$$[[2]]=$VV[13];
g$$[[3]]=$VV[14];
$VV[15]=0;(*debug*)
SMSExecuteBreakPoint["2","test",1,2];
$VV[16]=0;(*debug*)
];
```

Several modifications of the above procedures are possible.

Printing to Output Devices

SMSPrint

SMSPrint[$expr_1$,$expr_2$,...,*options*]
create a source code sequence that prints out all the expressions $expr_i$ accordingly to the given options

SMSPrintMessage[$expr_1$,$expr_2$,...,]

≡ SMSPrint[$expr_1$,$expr_2$,...,"Optimal"->True, "Output"->"Console","Condition"->None]

prints out to standard output device

option	default	description
"Output"	"Console"	"Console" ⇒ standard output device as defined in table below {"File", *file*} ⇒ create a source code sequence that prints out expressions $expr_i$ to file *file* (*file* is in general identified by the file name. For FORTRAN source codes it can also be identified by the FORTRAN I/O unit number) "AceFEM console" ⇒ create a source code sequence that prints out to CDriver console if possible (SMTInputData["Console"→True]) "AceFEM notebook" ⇒ create a source code sequence that prints out to current AceFEM notebook
"Optimal"	False	By default the code is included into source code only in "Debug" and "Prototype" mode. With the option "Optimal"→True the source code is always generated.
"Condition"	None	at the run time the print out is actually executed only if the given logical expression yields True

Options of the *SMSPrint* function.

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 $expr_i$ 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.

Language or Environment	*default output device* ("Console")
"Mathematica"	current notebook
"C"	console window (printf (…)
"Fortran"	console window (write(*,*) …)
"Matlab"	matlab window (disp (…)
"AceFEM"	MDriver ⇒ current notebook CDriver ⇒ current notebook CDriver console ⇒ console window (SMTInputData["Console"→True])

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 print-outs.

Printing is executed accordingly to the value of the SMTIData["DebugElement"] environment variable:

SMTIData["DebugElement",-1] ⇒ printouts are active for all elements (default value)

SMTIData["DebugElement",0] ⇒ printouts are disabled

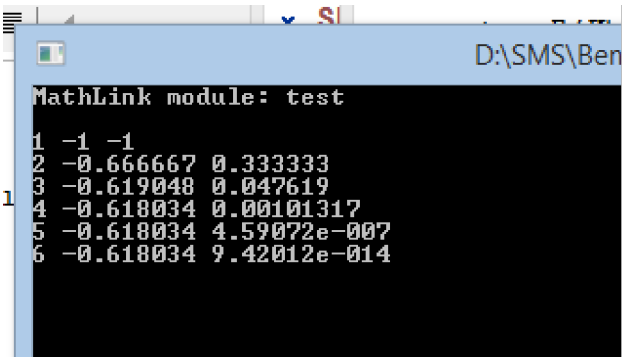
SMTIData["DebugElement",i] ⇒ printout is active for i-th element

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 `SMSInstallMathLink` command (e.g. 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 *Mathematica* has to be started from the terminal as follows:

  - open Terminal (look under Applications-Utilities for Terminal.app),
  - start *Mathematica* form Terminal (e.g. /Applications/Mathematica.app/Contents/MacOS/*Mathematica*),
  - after SMTAnalysis[] set SMTIData[“ConsoleWindow”,1]
  - messages will now be printed to terminal window.

- Linux

  In order to direct printouts to terminal window on Linux *Mathematica* has to be started from the terminal as follows:

  - open Terminal (e.g. search for Terminal),
  - start *Mathematica* form Terminal (e.g.*Mathematica* &),
  - after SMTAnalysis[] set SMTIData[“ConsoleWindow”,1]
  - messages will now be printed to terminal window.

Example 1: printing from Mathematica code

In[1]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "Mathematica", "Mode" → "Prototype"];
SMSModule["test", Real[x$$, r$$], "Input" → {x$$}, "Output" → {r$$}];
SMSPrint["pi=", π];
SMSExport[π, r$$];
SMSWrite[];
```

```
[1]  Consistency check - expressions
```

File: test.m **Size:** 694 **Time:** 1

Method	test
No.Formulae	2
No.Leafs	9

In[7]:=

```
Get["test.m"]
```

In[8]:=

```
FilePrint["test.m"]
```

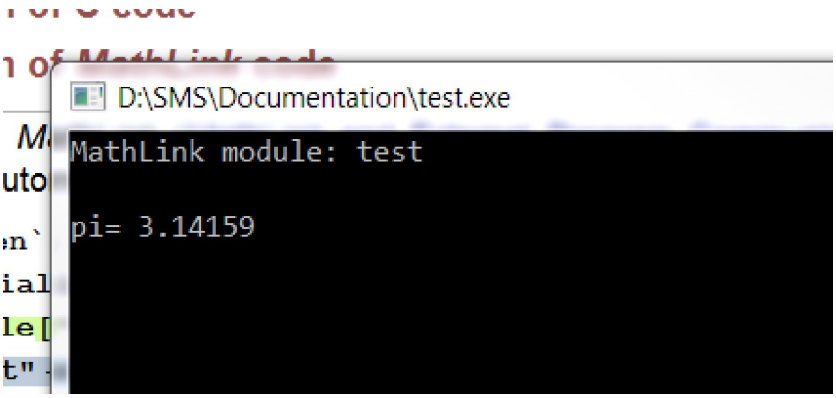
```
(*************************************************************
* AceGen    6.922 Windows (21 Feb 19)                       *
*           Co. J. Korelc  2013          21 Feb 19 21:25:09 *
*************************************************************
User     : USER
Notebook : AceGenTutorials
Evaluation 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[9]:=

```
r = 0.;
test[1., r]
```

```
pi= π
```

- Messages are printed to "console window" and to "test.out" file. Printing to notebook currently not supported!



In[61]:=

```
FilePrint["test.out"]
```

```
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[ ]=

```
True

[1]  Consistency check - expressions
```

File: test.c Size: 1062 Time: 1

Method	test
No.Formulae	4
No.Leafs	11

In[8]:=

```
FilePrint["test.c"]
```

```
/*************************************************************
* 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"

/******************* S U B R O U T I N E *********************/
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

Method	test
No.Formulae	6
No.Leafs	15

In[16]:=

```
FilePrint["test.f"]
```

```
!*************************************************************
!* AceGen    6.922 Windows (21 Feb 19)                      *
!*           Co. J. Korelc  2013           21 Feb 19 21:48:06 *
!*************************************************************
! 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

!******************** S U B R O U T I N E **********************
      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=", 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$$] > 0];
SMSExport[π, r$$];
SMSWrite[];
```

```
[0]  Consistency check - expressions
```

File: test.c Size: 1797 Time: 0

Method	test
No.Formulae	5
No.Leafs	18

In[25]:=

```
SMSInstallMathLink["Console" → True]
```

Out[•]=

```
{SMSSetLinkOption[test, {i_Integer, j_Integer}],
 SMSLinkNoEvaluations[test], test[x_?(Head[#1] == Real || Head[#1] == Integer &)]}
```

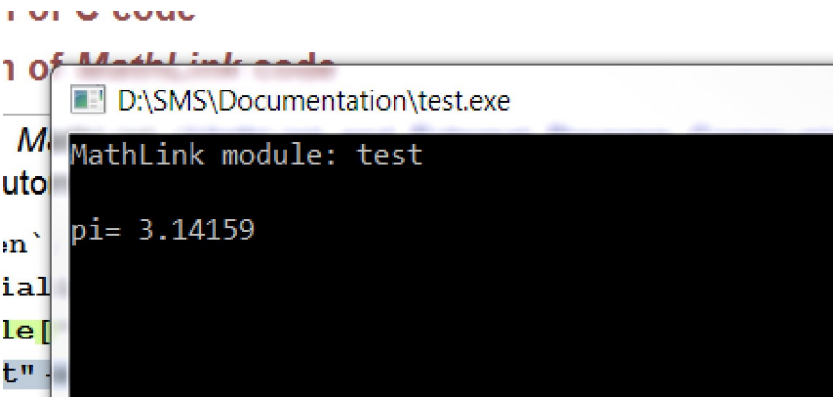
In[26]:=

```
test[1.]
```

Out[•]=

```
3.14159
```

- Messages are printed to "console window" and to "test.out" file. Printing to notebook is currently not supported!



In[61]:=

```
FilePrint["test.out"]
```

```
time= 1.3681e+009
e= 2.71828
```

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='", π];
SMSPrint["'load='", rdata$$["Multiplier"], "Output" → "File", "Condition" → "DebugElement"];
SMSWrite[];
```

```
[0]  Consistency check - expressions
```

File: test.m Size: 2244 Time: 0

Method	SMT`SKR
No.Formulae	2
No.Leafs	4

In[34]:=

```
FilePrint["test.m"]
```

```
(*************************************************************
* AceGen    6.922 Windows (21 Feb 19)                       *
*           Co. J. Korelc  2013          21 Feb 19 21:48:35 *
*************************************************************
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[{}],
 SMCToMMAString[{}],
 SMCToMMAString[{}],
 {1, 2, 3, 0, 1, 2, 3, -1},{2, 2, 2},{},"SensPVFIndex","NoNodeStorage","NoNodeData"
,If[gd===={}, {},gd]
,q3[[2]]//Transpose,{{1, 0, 0}, {0, 1, 0}, {0, 0, 1}},
{-1, -1, -1},"AdditionalData","OutFileName",
 SMCToMMAString["{}&"],
 {"D", "D", "D"},
 SMCToMMAString["{}&"],
 {},
 {},
 {},
 SMCToMMAString[{}],
 SMCToMMAString[""],
 SMCToMMAString[""],
 SMCToMMAString[""],
 {},
 "",
```

```
 "",
 "",
 {},
 {},
 {},
 {},
 {},
 "",{},{},
 {1., 1., 1.},
  FromCharacterCode[{123, 72, 111, 108, 100, 91, 78, 117, 108, 108, 93, 44, 32, 72, 111, 108, 100, 91, 78,
  {6.922,6.922,11.3},
  {},{},{},{},{},{},{}
  };

If[gd=!={} && Not[Floor[True]]
   ,SMC`SMCError={"Given data:",gd,"Required data:"
       ,SMCToMMAString[{}]};
    SMC`SMCAbort["Incorrect domain input data values","SMTAddDomain","SMTAddDomain"];
];

q1[[3,9]]=Round[0];
q1[[13]]=Round[{0, 0, 0}];
q1[[12]]=Round[{0, 0, 0}];
q1[[3,10]]=Round[0];
q1[[3,24]]=Round[0];
q1[[18]]=Array[0.&,Round[0]];
q1];


(********************** M O D U L E *************************)
SetAttributes[SMT`SKR,HoldAll];
SMT`SKR["test",es$$_,ed$$_,ns$$_,nd$$_,rdata$$_,idata$$_
     ,p$$_,s$$_]:=Module[{},
Print["pi=","  ",Pi];
PutAppend[{"load=",rdata$$[[1]]},SMTSession[[10]]];
];
```

Run Time Debugging

Example of run-time debugging

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

$$\Phi = \begin{Bmatrix} a\,x\,y + x^3 = 0 \\ a - x\,y^2 = 0 \end{Bmatrix}$$

where *x* and *y* are unknowns and *a* 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$$, tol$$], Integer[nmax$$]];
{x0, y0, a, ϵ} ⊢ SMSReal[{x$$, y$$, a$$, tol$$}];
nmax ⊢ SMSInteger[nmax$$];
{x, y} ⫤ {x0, y0};
SMSDo[
  Φ ⊨ {a x y + x^3, a - x y^2};
  Kt ⊨ SMSD[Φ, {x, y}];
  {Δx, Δy} ⊨ SMSLinearSolve[Kt, -Φ];
  {x, y} ⫤ {x, y} + {Δx, Δy};
  SMSSetBreak["A", "Active" → False];
  SMSIf[SMSSqrt[{Δx, Δy}.{Δx, Δy}] < ϵ
   , 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 formating
```

File: test.m **Size:** 2560 **Time:** 0

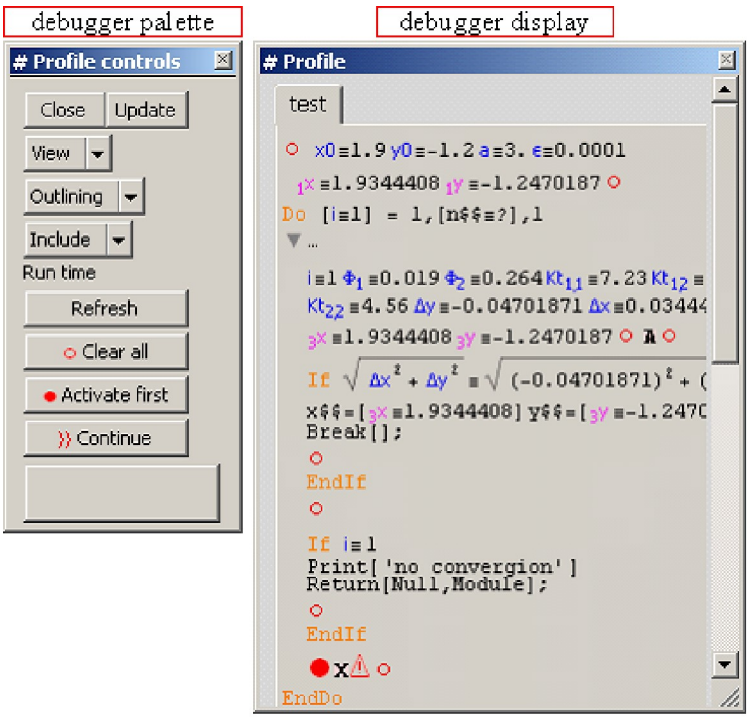
Method	test
No.Formulae	35
No.Leafs	246

- 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 *SMSExecuteBreakPoint* 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 *Kt* 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]
```

$$K=\begin{pmatrix} 7.23 & 5.7 \\ -1.44 & 4.56 \end{pmatrix}$$

$$K=\begin{pmatrix} 7.48513 & 5.80332 \\ -1.55506 & 4.82457 \end{pmatrix}$$

$$K=\begin{pmatrix} 7.4744 & 5.79955 \\ -1.55185 & 4.81646 \end{pmatrix}$$

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.

SMSSetBreak[*breakID*]
insert break point call into the source code with the string *breakID* as the break identification

option	default	description
"Active"	True	break point is by default active
"Optimal"	False	by default the break point is included into source code only in "Debug" mode. With the option "Optimal" the break point is always generated.

Options of the `SMSSetBreak` function.

Break points are inserted only if the code is generated with the "Mode"→"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"→False the break point becomes initially inactive. The break points are also automatically generated at the end of If.. else..endif and Do...enddo 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

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

SMSClearBreak[*breakID*]
disable break point with the break identification *breakID*

SMSClearBreak["Default"]
set all options to default values

SMSClearBreak[]
disable all break points

SMSActivateBreak

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[b, func]≡ SMSActivateBreak[b,"Function"→func,"Window"→False,"Interactive"→False]

SMSActivateBreak[]≡ SMSActivateBreak[1]

option	default	description
"Interactive"	True	initiates dialog (see also Dialog)
"Window"	True	open new window for debugging
"Function"	None	execute pure user defined function at the break point

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

SMSAbs[*exp*]	absolute value of *exp* (real type function)
SMSSign[*exp*]	−1, 0 or 1 depending on whether *exp* is negative, zero, or positive (real type function)
SMSKroneckerDelta[*i*, *j*]	1 or 0 depending on whether *i* is equal to *j* or not (real type function)
SMSMin[*exp1*, *exp2*]	≡ Min[*exp1*, *exp2*] (the type is the same as the type of arguments)
SMSMax[*exp1*, *exp2*]	≡ Max[*exp1*, *exp2*] (the type is the same as the type of arguments)
SMSPower[x,y]	x^y (see details below)
SMSSqrt[x]	$\sqrt{x}$ (see details below)
SMSMod[i,j]	reminder of i/j (the type is the same as the type of arguments)
SMSFloor[x]	returns the nearest integer not greater than the given value (real type function)
SMSCeiling[x]	returns the smallest integer value greater than or equal to x (real type function)
SMSArcCos[x], SMSArcSin[x], SMSArcTan[x], SMSArcTan[dx,dy]	inverse trigonometric functions
SMSRandomizeFunction[*func*[i_]:→outputForm,*derivatives*]	an arbitrary function is replaced by its SMS equivalent with random signature (e.g. SMSRandomizeFunction[ArcCos[i_]:→"acos"[i]] would define SMSArcCos function with unique signature and define C/C++/C # source code output format). Derivatives definition can be: *expression* – derivatives are user defined Automatic – derivatives are derivatives of the corresponding elementary function *func*

Functions with unique signature and support for derivatives.

SMSSqrt (√x)

SMSSqrt[*x*]

$\sqrt{x}$ with an unique signature (see Expression Optimization)

(SMSSqrt[x]≡SMSSqrt[x,0,∞])

SMSSqrt[*x*,*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 ⇒ $\sqrt{x}$

type=1 ⇒ function $\sqrt{x}$ is approximated with 5-th order polynomial approximation on interval [-∞,x0]

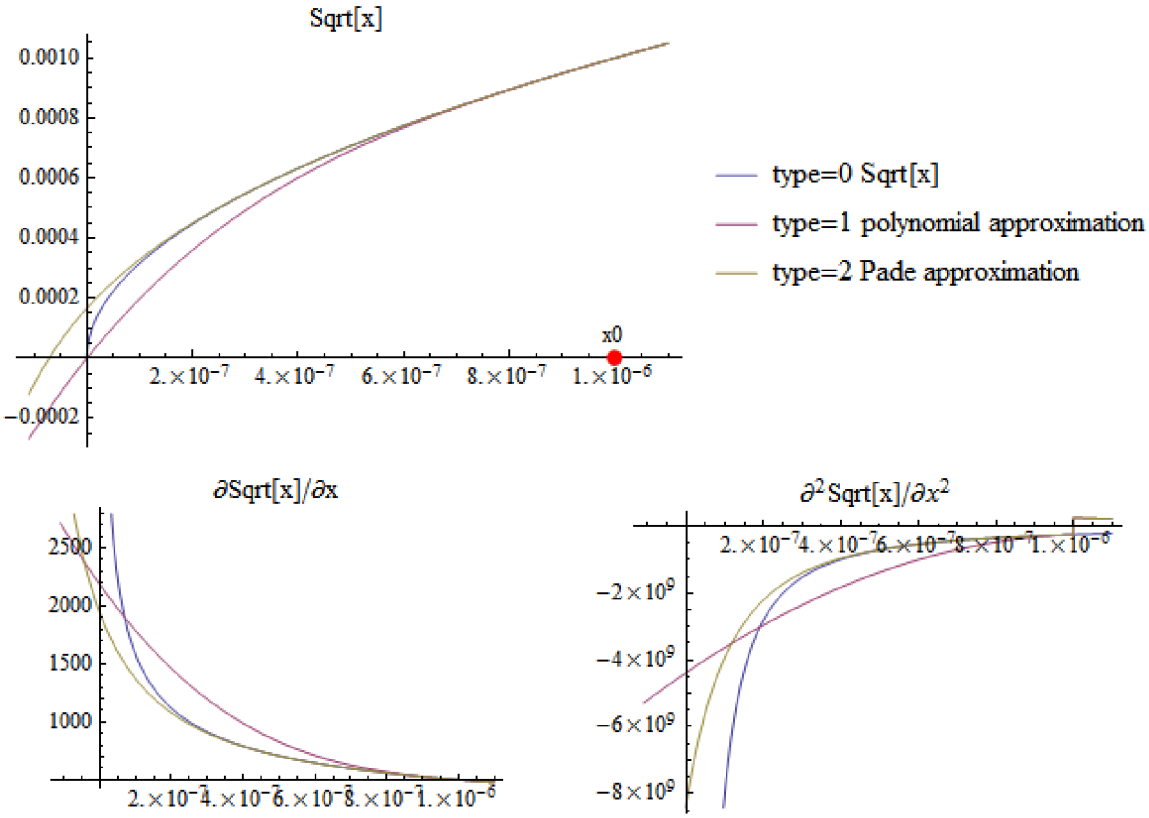
type=2 ⇒ function $\sqrt{x}$ is approximated with Pade approximation on interval [-∞,x0]

option	default	description
"x0"	10^-6	define interval $[0,x_0]$
"PadeOrder"	{3,2}	see PadeApproximant

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,x0*] in a way that singularity in derivatives is avoided.

Out[ ]=



SMSPower (x^y)

SMSPower[x,y]

x^y

this form is appropriate when x>0 or y is positive integer

SMSPower[*x,y, 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^y is approximated with 5-th order polynomial approximation on interval [-∞,x0]

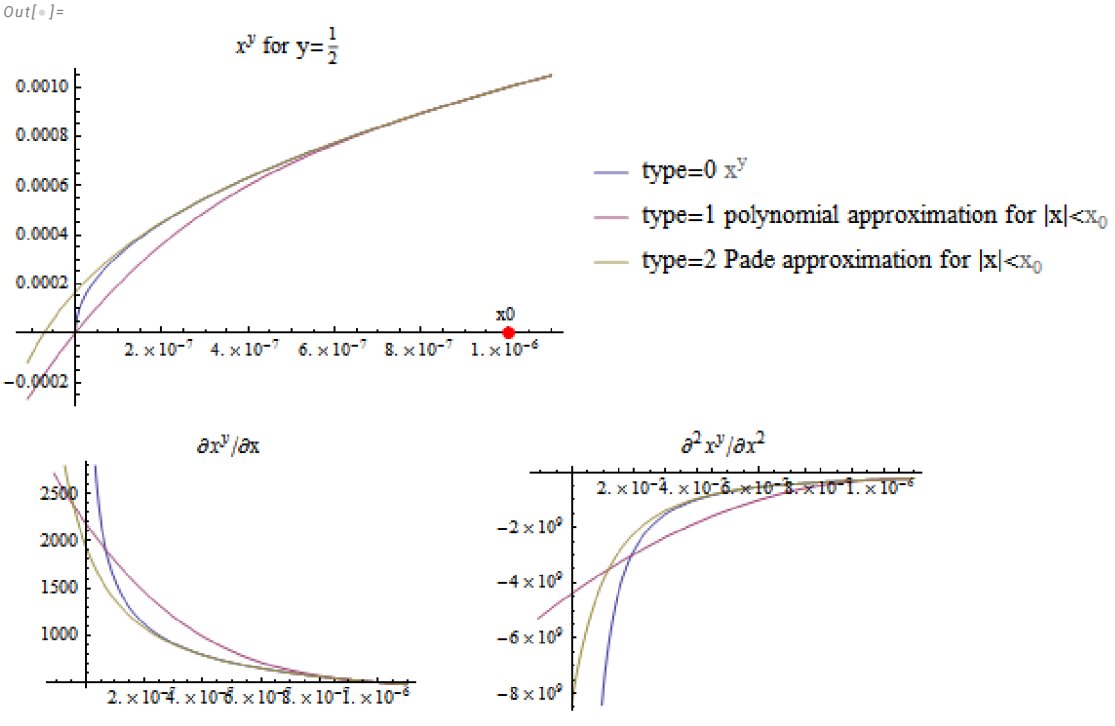
type=2 ⇒ function x^y is approximated with Pade approximation for 0<y<1 and 5-th order polynomial approximation for y<0 on interval [-∞,x0]

option	default	description
"x0"	10^-6	define interval $[0,x_0]$
"PadeOrder"	{3,2}	see PadeApproximant

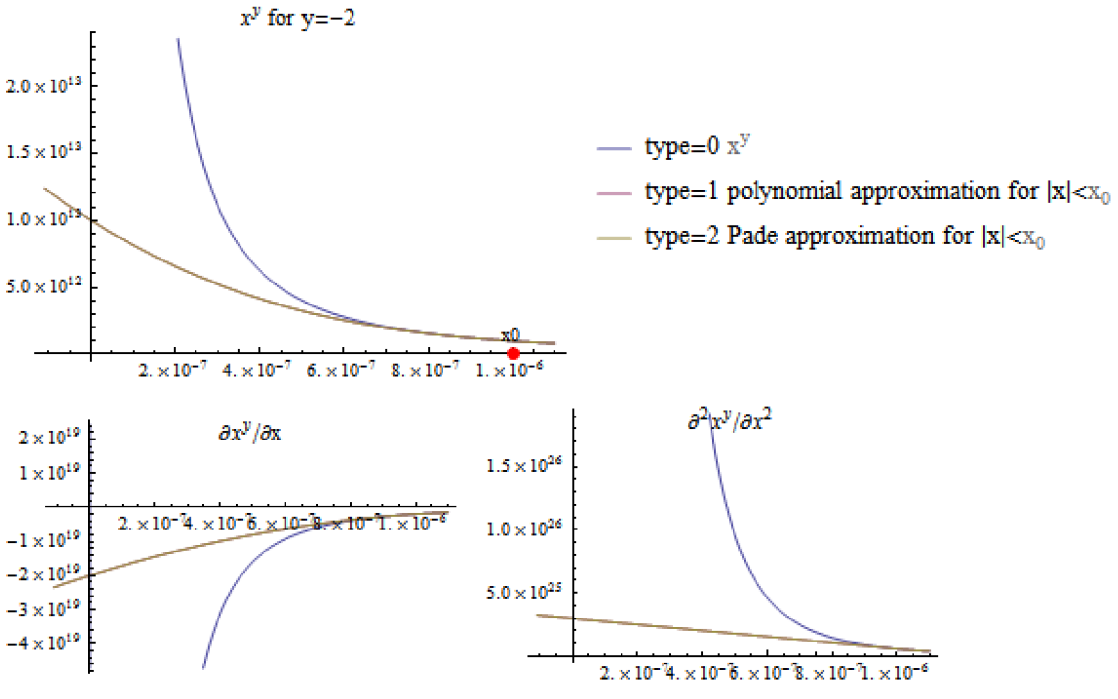
Options of the SMSPower function.

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,x0]* in a way that singularities are avoided.

Out[ ]=



Out[ ]=



Standard Functions Without Unique Signature

Tan[*x*], Cot[x], Cos[x], Sec[x], Sin[x], Scs[x], ArcTan[x], ArcCos[x], ArcSin[x]	trigonometric functions
Log[x], Exp[x]	
Cosh[x], Sinh[x]	
SMSCbrt[x]	$x^{1/3}$ (also works for negative real x)

Functions without unique signature and support for derivatives provided by Mathematica.

System Utility Functions

SMSNumberQ[*exp*]	gives True if *exp* is a real number and False if the results of the evaluation is N/A (not a number). SMSNumberQ produces a code segment that test results at run−time.
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

structCompressed=SMSCompress[*struct*]
analyzes structure and finds pattern inside the structure that can be used to compress the structure into vector. Function returns *structCompressed* data structure.

structCompressed["Data"]
returns compressed *struct*

structCompressed["Length"]

returns the length of vector into which the structure is compressed.

SMSUncompress[*structCompressed*]
returns uncompressed structure *struct*

SMSUncompress[*structCompressed*,*compressedVector*]
returns uncompressed *compressedVector*

SMSUncompress[*structCompressed*,*otherStruct*]
returns *otherStruct* modified in such a way that *otherStruct* has the same pattern as *struct*.

structCompressed=SMSCompress[*struct*, "Export"->Function[{i},*f(i)*]]
struct is compressed into vector which is then exported (stored) into used defined memory space

SMSUncompress[*structCompressed*,"Import"->*importKey*]
compressedVector is imported from user defined memory space accordingly to the *importKey*

option	default	description
"Export"	False	False ⇒ nothing 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*, "Export"->Function[{i},cVector$$[i]]]) will export compressed *struct* into array cVector[maxLength]).
"AddIn"	False	compressed data is added to the already stored compressed data
"Import"	False	False ⇒ nothing True ⇒ compressed vector is taken from the same memory space is 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["Length"]]

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$$], SMSReal[L$$]};
ui ⊨ SMSReal[Array[u$$[#1] & , 3]];
Ni ⊨ {x/L, 1 - x/L, x/L (1 - x/L)};
u ⊨ Ni.ui;
f ⊨ u^2;
g ⊨ SMSD[f, ui];
struct ⊨ SMSD[g, ui];
```

In[203]:=

```
struct
```

Out[•]=

$$\left\{\left\{\text{struct}_{1,1}, \text{struct}_{1,2}, \text{struct}_{1,3}\right\}, \left\{\text{struct}_{1,2}, \text{struct}_{2,2}, \text{struct}_{2,3}\right\}, \left\{\text{struct}_{1,3}, \text{struct}_{2,3}, \text{struct}_{3,3}\right\}\right\}$$

- 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[•]=
```
6
```

- returns uncompress structure accordingly to the given signature

In[206]:=
```
structUncompressed = SMSUncompress[structCompressed]
```

Out[•]=

$$\{\{\text{struct}_{1,1}, \text{struct}_{1,2}, \text{struct}_{1,3}\}, \{\text{struct}_{1,2}, \text{struct}_{2,2}, \text{struct}_{2,3}\}, \{\text{struct}_{1,3}, \text{struct}_{2,3}, \text{struct}_{3,3}\}\}$$

- returns uncompressed *compressedVector*

In[207]:=
```
compressedVector = Table[i, {i, 6}]
```

Out[•]=
```
{1, 2, 3, 4, 5, 6}
```

In[208]:=
```
SMSUncompress[structCompressed, compressedVector]
```

Out[•]=
```
{{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[•]=

$$\{\{\text{struct}_{1,1}, \text{struct}_{1,2}, \text{struct}_{1,3}\}, \{0, \text{struct}_{2,2}, \text{struct}_{2,3}\}, \{\text{struct}_{1,3}, \text{struct}_{2,3}, \text{struct}_{3,3}\}\}$$

- *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[•]=

$$\{\{\text{struct}_{1,1}, \text{struct}_{1,2}, \text{struct}_{1,3}\}, \{\text{struct}_{1,2}, \text{struct}_{2,2}, \text{struct}_{2,3}\}, \{\text{struct}_{1,3}, \text{struct}_{2,3}, \text{struct}_{3,3}\}\}$$

- compress *struct* and export compressed vector into working vector c$$

In[211]:=
```
structCompressed2 = SMSCompress[struct, "Export" → Function[{i}, c$$[i]]];
```

- import compressed data from working vector c$$ and uncompress it

In[212]:=
```
Table[Function[{i}, c$$[i]][i], {i, 6}]
```

Out[•]=
```
{c$$[1], c$$[2], c$$[3], c$$[4], c$$[5], c$$[6]}
```

In[213]:=
```
structUncompressed2 = SMSUncompress[structCompressed2, "Import" → True]
```

Out[•]=

$$\{\{c\$\$_1, c\$\$_2, c\$\$_3\}, \{c\$\$_2, c\$\$_4, c\$\$_5\}, \{c\$\$_3, c\$\$_5, c\$\$_6\}\}$$

- export element 2,1 obtained by various transformations

In[214]:=

```
SMSExport[{struct[[2, 1]], structUncompressed[[2, 1]],
   otherStructModified[[2, 1]], structUncompressed2[[2, 1]]}, k21$$];
```

In[215]:=

```
SMSWrite[];
```

File: tmptest.c Size: 2060 Time: 0

Method	Test
No.Formulae	6
No.Leafs	136

In[216]:=

```
SMSInstallMathLink["Console" → False];
```

- function returns element 2,1 obtained by various transformations and working vector c$$ used for compression. Note that only 6 elements of the working vector c$$ were used.

In[217]:=

```
Test[{0., 1., 7.}, π // N, 10.]
```

Out[ ]=

$$\{\{0.430926, 0.430926, 0.430926, 0.430926\}, \{0.197392, 0.430926, 0.13538, 0.940755, 0.295547, 0.0928488, 3.16\times10^{-322}, 3.56\times10^{-322}, 2.08458\times10^{-317}\}\}$$

In[219]:=

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:

operation	solver available	description
SMSLinearSolve	"Closed form"/ "LAPACK"	solution to the system of linear equations
SMSEigensystem	"LAPACK"	eigenvalues and eigenvectors (only supported for AceFEM finite elements)
SMSLUFactor	"Closed form"	the LU decomposition
SMSFactorSim	"Closed form"	the LU decomposition along with the pivot list of symmetric matrix M
SMSLUSolve	"Closed form"	solution of the linear system represented by decomposed matrix LU and right-hand side B
SMSInverse	"Closed form"	the inverse of square matrix M
SMSDet	"Closed form"	the determinant of square matrix M
SMSKrammer[M,B]	"Closed form"	generate a code sequence that solves the system of linear equations $A\ x= B$ analytically and return the solution vector
SMSEigenvalues	"Closed form"	create code sequence that calculates the eigenvalues of the third order (3×3) matrix and return the vector of 3 eigenvalues
SMSInvariantsI	"Closed form"	I_1, I_2, I_3 invariants of the third order matrix
SMSInvariantsJ	"Closed form"	J_1, J_2, J_3 invariants of the deviator of the third order matrix
SMSRotationMatrix	"Closed form"	rotation matrix **R** that belongs to given rotation vector $\boldsymbol{\phi}$

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). In the case of closed form solution, the Gauss elimination procedure is used without pivoting! Pivoting is automatically done for the numerical solution by LAPACK package.

option	default	description
"Symmetric"	Automatic	Automatic ≡ $\mathbf{A}^T=\mathbf{A}$ True - assume that **A** is symmetric False - assume that **A** is nonsymmetric
"Solver"	"Closed form"	"Closed form" ⇒ closed form solution for **x** "LAPACK" ⇒ numerical solution for **x** using symmetric or nonsymmetric solver from MKL LAPACK numerical library. **ONLY AVAILABLE FOR AceFEM elements!!!**
"Report"	False	False ⇒ returns **x** True ⇒ returns {**x**, *info*} If *info*=0, the execution is successful. If *info* = -i, parameter i had an illegal value. If *info* = i, Ui (computed in double precision for mixed precision subroutines) is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution could not be computed.

Options of SMSLinearSolve function.

Example

This generates the *FORTRAN* code that returns the solution to the general linear system of equations:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{21} & \dots & a_{2N} \\ \dots & \dots & \dots & \dots \\ a_{N1} & a_{N2} & \dots & a_{NN} \end{bmatrix} \bullet \begin{Bmatrix} x_1 \\ x_2 \\ \dots \\ x_N \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ \dots \\ b_N \end{Bmatrix}$$

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
Methods	No.Formulae	No.Leafs
Test	18	429

In[30]:=

```
FilePrint["test.c"]
```

```
/*************************************************************
* AceGen    2.502 Windows (18 Nov 10)                        *
*           Co. J. Korelc  2007            24 Nov 10 13:05:43*
*************************************************************
User : USER
Evaluation time                 : 0 s     Mode  : Optimal
Number of formulae              : 18      Method: Automatic
Subroutine                      : Test size :429
Total size of Mathematica  code : 429 subexpressions
Total size of C code            : 841 bytes*/
#include "sms.h"

/******************* S U B R O U T I N E *********************/
void Test(double v[5001],double a[4][4],double b[4],double x[4])
{
v[40]=1e0/a[0][0];
v[21]=a[1][0]*v[40];
v[22]=a[1][1]-a[0][1]*v[21];
v[23]=a[1][2]-a[0][2]*v[21];
v[24]=a[1][3]-a[0][3]*v[21];
v[25]=a[2][0]*v[40];
v[26]=a[3][0]*v[40];
v[27]=b[1]-b[0]*v[21];
v[28]=(a[2][1]-a[0][1]*v[25])/v[22];
v[29]=a[2][2]-a[0][2]*v[25]-v[23]*v[28];
v[30]=a[2][3]-a[0][3]*v[25]-v[24]*v[28];
v[31]=(a[3][1]-a[0][1]*v[26])/v[22];
v[32]=b[2]-b[0]*v[25]-v[27]*v[28];
v[33]=(a[3][2]-a[0][2]*v[26]-v[23]*v[31])/v[29];
v[35]=(-b[3]+b[0]*v[26]+v[27]*v[31]+v[32]*v[33])/(-a[3][3]+a[0][3]*v[26]+v[24]*v[31]+v[30]*v[33]);
v[36]=(v[32]-v[30]*v[35])/v[29];
v[37]=(v[27]-v[24]*v[35]-v[23]*v[36])/v[22];
x[0]=(b[0]-a[0][3]*v[35]-a[0][2]*v[36]-a[0][1]*v[37])*v[40];
x[1]=v[37];
x[2]=v[36];
x[3]=v[35];
};
```

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!

option	default	description
"Solver"	"LAPACK"	numerical solution only
"Symmetric"	Automatic	Automatic ≡ $\mathbf{A}^T=\mathbf{A}$ True - assume that **A** is symmetric False - assume that **A** is nonsymmetric
"Report"	False	True ⇒ adds *info* to the list of results returned {…, *info*} If *info*=0, the execution is successful. If *info* = -i, parameter i had an illegal value. If *info* = i, Ui (computed in double precision for mixed precision subroutines) is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution could not be computed.

Options of SMSEigensystem function.

Example

```
{σMain, σMainDirection} ⊨ SMSEigensystem[σ, "Solver" → "LAPACK", "Symmetric" → 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.

option	default	description
"Module"	False	With the "Module"-> True a separate subroutine is formed in order to reduce the size of the code.
"Report"	False	False ⇒ return decomposed LU decompose **A** True ⇒ return {LU, *info*} If *info*=0, the execution is successful. If *info* = -i, parameter i had an illegal value. If *info* = i, Ui (computed in double precision for mixed precision subroutines) is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution could not be computed.

Options of SMSLUFactor function.

SMSLUSolve

SMSLUSolve[*LU*,*B*]
solution of the linear system represented by decomposed matrix *LU* and right-hand side *B*

SMSLinearSolve[*A*,*B*]=SMSLUSolve[SMSLUFactor[*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

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//Length]] instead.

SMSDet

SMSDet[*M*]
the determinant of square matrix *M*

Simultaneous simplification is performed during the process.

SMSKrammer

SMSKrammer[*M*,*B*]
generate a code sequence that solves the system of linear equations *A x=B* analytically and return the solution vector

The Krammer's rule is used and simultaneous simplification is performed during the process.

SMSEigenvalues

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

SMSInvariantsI[*M*]
I_1, I_2, I_3 invariants of the third order matrix

SMSInvariantsJ

SMSInvariantsJ[*M*]
J_1, J_2, J_3 invariants of the deviator of the third order matrix

SMSRotationMatrix

SMSRotationMatrix[$\boldsymbol{\phi}$, dc]
SMSRotationMatrix[$\boldsymbol{\phi}$] ≡ SMSRotationMatrix[$\boldsymbol{\phi}$, Infinity]
returns rotation matrix **R** that belongs to given rotation vector $\boldsymbol{\phi}$ (**R**=exp(**Ψ**), **Ψ** is skew-symmetric tensor of rotation vector $\boldsymbol{\phi}$), where *dc* is a code that defines the number of derivatives calculated together with the function evaluation as follows:

dc = 0 ⇒ rotation matrix only
dc = 1 ⇒ rotation matrix and its first derivative
dc = 2 ⇒ rotation matrix and its first and second derivative
dc = Infnity ⇒ generated code suport derivatives of an arbitrary order

The function uses Rodrigues' rotation formula for large rotations and polynomial approximations for small rotations in a way that the result is always machine precision accurate. Function returns calculated functions value. Derivatives are not returned directly. An appropriate AD exceptions are defined that define first and second order derivatives when required.

Tensor Algebra

SMSCovariantBase[{ϕ_1,ϕ_2,ϕ_3}, {η_1,η_2,η_3}]
the covariant base vectors of transformation from the coordinates {η_1,η_2,η_3} to coordinates {ϕ_1,ϕ_2,ϕ_3}

Transformations ϕ_1,ϕ_2,ϕ_3 are arbitrary functions of independent variables η_1,η_2,η_3. Independent variables η_1,η_2,η_3 have to be proper auxiliary variables with unique signature (see also SMSD).

Example: Cylindrical coordinates

In[368]:=
```
<< AceGen`;
SMSInitialize["test", "Language" → "Mathematica"];
SMSModule["test"];
{r, ϕ, z} ⊨ Array[SMSFictive[] &, {3}];
SMSCovariantBase[{r Cos[ϕ], r Sin[ϕ], z}, {r, ϕ, z}] // MatrixForm
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} \mathrm{Cos}[\phi] & \mathrm{Sin}[\phi] & 0 \\ -r\,\mathrm{Sin}[\phi] & \mathrm{Cos}[\phi]\,r & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

SMSCovariantMetric[{ϕ_1,ϕ_2,ϕ_3}, {η_1,η_2,η_3}]
the covariant metric tensor of transformation from coordinates {η_1,η_2,η_3} to coordinates {ϕ_1,ϕ_2,ϕ_3}

Transformations ϕ_1,ϕ_2,ϕ_3 are arbitrary functions of independent variables η_1,η_2,η_3. Independent variables η_1,η_2,η_3 have to be proper auxiliary variables with unique signature (see also SMSD).

Example: Cylindrical coordinates

In[373]:=
```
<< AceGen`;
SMSInitialize["test", "Language" → "Mathematica"];
SMSModule["test"];
{r, ϕ, z} ⊨ Array[SMSFictive[] &, {3}];
SMSCovariantMetric[{r Cos[ϕ], r Sin[ϕ], z}, {r, ϕ, z}] // MatrixForm
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & r^2 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

SMSContravariantMetric[{ϕ_1,ϕ_2,ϕ_3}, {η_1,η_2,η_3}]
the contravariant metric tensor of transformation from coordinates {η_1,η_2,η_3} to coordinates {ϕ_1,ϕ_2,ϕ_3}

Transformations ϕ_1,ϕ_2,ϕ_3 are arbitrary functions of independent variables η_1,η_2,η_3. Independent variables η_1,η_2,η_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[ ]//MatrixForm=

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{r^2} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

SMSChristoffell1[{ϕ_1,ϕ_2,ϕ_3}, {η_1,η_2,η_3}]
the first Christoffell symbol {i,j,k} of transformation from coordinates {η_1,η_2,η_3} to coordinates {ϕ_1,ϕ_2,ϕ_3}

Transformations ϕ_1,ϕ_2,ϕ_3 are arbitrary functions of independent variables η_1,η_2,η_3. Independent variables η_1,η_2,η_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}];
SMSChristoffell1[{r Cos[ϕ], r Sin[ϕ], z}, {r, ϕ, z}] // MatrixForm
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ r \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \\ \begin{pmatrix} 0 \\ r \\ 0 \end{pmatrix} & \begin{pmatrix} -r \\ 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \\ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \end{pmatrix}$$

SMSChristoffell2[{ϕ_1,ϕ_2,ϕ_3}, {η_1,η_2,η_3}]
the second Christoffell symbol Γ^k_{ij} of transformation from coordinates {η_1,η_2,η_3} to coordinates {ϕ_1,ϕ_2,ϕ_3}

Transformations ϕ_1,ϕ_2,ϕ_3 are arbitrary functions of independent variables η_1,η_2,η_3. Independent variables η_1,η_2,η_3 have to be proper auxiliary variables with unique signature (see also SMSD).

Example: Cylindrical coordinates

In[388]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "Mathematica"];
SMSModule["test"];
{r, ϕ, z} ⊨ Array[SMSFictive[] &, {3}];
SMSChristoffel2[{r Cos[ϕ], r Sin[ϕ], z}, {r, ϕ, z}] // MatrixForm
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ \frac{1}{r} \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \\ \begin{pmatrix} 0 \\ \frac{1}{r} \\ 0 \end{pmatrix} & \begin{pmatrix} -r \\ 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \\ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \end{pmatrix}$$

SMSTensorTransformation[*tensor,transf,coord,index_types*]
tensor transformation of arbitrary tensor field *tensor* with indices *index_types* defined in curvilinear coordinates *coord* under transformation *transf*

Transformations *transf* are arbitrary functions while coordinates *coord* have to be proper auxiliary variables with the unique signature (see also SMSD). The type of tensor indices is specified by the array *index_types* where *True* means covariant index and *False* contravariant index.

Example: Cylindrical coordinates

Transform contravariant tensor $u^i = \{r^2, r\ \mathrm{Sin}[\phi], r\ z\}$ defined in cylindrical coordinates $\{r,\phi,z\}$ into Cartesian coordinates.

In[393]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "Mathematica"];
SMSModule["test"];
{r, ϕ, z} ⊨ Array[SMSFictive[] &, {3}];
SMSTensorTransformation[{r^2, r Sin[ϕ], r z}, {r Cos[ϕ], r Sin[ϕ], z}, {r, ϕ, z}, {False}]
```

Out[ ]=

$$\left\{\mathrm{Cos}[\phi]\ r^2 + r\ \mathrm{Sin}[\phi]^2,\ \mathrm{Cos}[\phi]\ \mathrm{Sin}[\phi] - r\ \mathrm{Sin}[\phi],\ r\ z\right\}$$

SMSDCovariant[*tensor,transf,coord,index_types*]
covariant derivative of arbitrary tensor field *tensor* with indices *index_types* defined in curvilinear coordinates *coord* under transformation *transf*

Transformations *transf* are arbitrary functions while coordinates *coord* have to be proper auxiliary variables with unique signature (see also SMSD). The type of tensor indices is specified by the array *index_types* where *True* means covariant index and *False* contravariant index.

The SMSDCovariant function accepts the same options as SMSD function.

Example: Cylindrical coordinates

Derive covariant derivatives $u^i|_j$ of contravariant tensor $u^i = \{r^2, r\ \mathrm{Sin}[\phi], r\ z\}$ defined in cylindrical coordinates $\{r,\phi,z\}$.

In[398]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "Mathematica"];
SMSModule["test"];
{r, ϕ, z} ⊨ Array[SMSFictive[] &, {3}];
SMSDCovariant[{r^2, r Sin[ϕ], r z}, {r Cos[ϕ], r Sin[ϕ], z}, {r, ϕ, z}, {False}] // MatrixForm
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} 2\,r & -r^2\,\mathrm{Sin}[\phi] & 0 \\ 2\,\mathrm{Sin}[\phi] & r + \mathrm{Cos}[\phi]\,r & 0 \\ z & 0 & r \end{pmatrix}$$

Matrix Functions

SMSMatrixExp[**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 ⇒ function only f(**M**) dc=1 ⇒ function and its first derivative $\left(f(\mathbf{M}),\frac{\partial f(\mathbf{M})}{\partial \mathbf{M}}\right)$ dc=2 ⇒ 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[**M**,*dc*]	returns logarithm of positive definite 3×3 matrix,
SMSMatrixSqrt[**M**,*dc*]	returns square root of positive definite 3×3 matrix,
SMSMatrixPower[**M**,p,*dc*]	returns $\mathbf{M}^p$ where **M** is positive definite 3×3 matrix and p is an arbitrary real number,
SMSMatrixPowerSeries[**M**,{{c_1,p_1},…,{c_n,p_n}},*dc*],	returns $\sum_{i=1}^{n} c_i \mathbf{M}^{p_i}$ where **M** is positive definite 3×3 matrix and c_i and p_i are arbitrary real numbers,
SMSMatrixPowerSeries[**M**,**cp**,n,*dc*],	returns $\sum_{i=1}^{n} c_i \mathbf{M}^{p_i}$ where **M** is positive definite 3×3 matrix and **cp**={c_1,p_1,...} 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.

option name	*default value*	
"Report"	False	False ⇒ function returns calculated function value: f(**M**) True ⇒ function returns calculated function value and report value *r*: {f(**M**),*r*}
"Order"	Infinity	Infinity ⇒ closed form representation r_Integer ⇒ Taylor series expansion of order r ϵ_Real ⇒ Taylor series expansion truncated when the norm of term is less than ϵ

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

HUDOBIVNIK, 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

KORELC, Jože, STUPKIEWICZ, Stanislaw. Closed-form matrix exponential and its application in finite-strain plasticity. International journal for numerical methods in engineering, ISSN 0029-5981, 2014, 98(13):960-987

WARNNING: 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 define first and second order derivatives.
In the case of "Report"->True option, function additional returns integer *r* with the following value:
r = 0 ⇒ 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 ⇒ regular point,
r = 2 ⇒ singular point A (multiple eigenvalues),
r = 3 ⇒ singular point B (complex eigenvalues),
r = 4 ⇒ 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[M$$[3, 3], EM$$[3, 3]], "Input" → M$$, "Output" → EM$$];
M0 ⊢ SMSReal[Array[M$$, {3, 3}]];
EM ⊨ SMSMatrixExp[M0, 0];
SMSExport[EM, EM$$];
SMSWrite[];
SMSInstallMathLink["ExpM", "Console" → True];
```

Closed form matrix function module. Derivatives are not supported. See also: Matrix Functions

File:	ExpM.c	Size: 2201
Methods	No.Formulae	No.Leafs
ExpM	3	214

In[78]:=

- Test: calculate e^{I}.

In[79]:=

```
M = {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}};
ExpM[M] // MatrixForm
MatrixExp[M] // MatrixForm
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} 2.71828 & 0. & 0. \\ 0. & 2.71828 & 0. \\ 0. & 0. & 2.71828 \end{pmatrix}$$

Out[ ]//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 $\mathbf{M}^p$ together with its first derivative.

In[1]:=

```
<< AceGen`;
SMSInitialize["PowerM", "Environment" → "MathLink"];
SMSModule["PowerM", Real[M$$[3, 3], p$$, EM$$[3, 3], DEM$$[3, 3, 3, 3]],
  "Input" → {M$$, p$$}, "Output" → {EM$$, DEM$$}];
M0 ⊢ SMSReal[Array[M$$, {3, 3}]];
p ⊢ SMSReal[p$$];
EM ⊨ SMSMatrixPower[M0, p, 1];
SMSExport[EM, EM$$];
DEM ⊨ SMSD[EM, M0];
SMSExport[DEM, DEM$$];
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: Matrix Functions

File: PowerM.c	Size: 5709	Time: 1

Method	PowerM
No.Formulae	41
No.Leafs	1203

In[78]:=

- Test: calculate $(2\,\mathbf{I})^{3.5}$ and its first derivatives.

In[12]:=

```
M = 2 {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}};
{powerM, dpowerM} = PowerM[M, 3.5];
```

In[14]:=

```
powerM // MatrixForm
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} 11.3137 & 0. & 0. \\ 0. & 11.3137 & 0. \\ 0. & 0. & 11.3137 \end{pmatrix}$$

In[15]:=

```
MatrixPower[M, 3.5] // MatrixForm
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} 11.3137 & 0. & 0. \\ 0. & 11.3137 & 0. \\ 0. & 0. & 11.3137 \end{pmatrix}$$

In[17]:=

```
dpowerM // MatrixForm
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} \begin{pmatrix} 19.799 & 0. & 0. \\ 0. & 1.77636\times10^{-15} & 0. \\ 0. & 0. & 1.77636\times10^{-15} \end{pmatrix} & \begin{pmatrix} 0. & 19.799 & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 0. \end{pmatrix} \\ \begin{pmatrix} 0. & 0. & 0. \\ 19.799 & 0. & 0. \\ 0. & 0. & 0. \end{pmatrix} & \begin{pmatrix} 1.77636\times10^{-15} & 0. & 0. \\ 0. & 19.799 & 0. \\ 0. & 0. & 1.77636\times10^{-15} \end{pmatrix} \\ \begin{pmatrix} 0. & 0. & 0. \\ 0. & 0. & 0. \\ 19.799 & 0. & 0. \end{pmatrix} & \begin{pmatrix} 0. & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 19.799 & 0. \end{pmatrix} \quad \begin{pmatrix} 1.77636 \\ 0 \\ 0 \end{pmatrix} \end{pmatrix}$$

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

SMSFindRoot[f,{x,x_0}]	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.
SMSFindRoot[f_Function, x_0]	inserts a code that searches for a numerical root of scalar function defined by a pure function (e.g Function[{x},x^2]), starting from the point x_0. Function can create an algorithm.

Function SMSFindRoot returns a vector:
{
x - calculated root,
True if convergence was obtained and False if not,
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") 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").

option name	*default value*	
"MaxIterations"	30	maximum number of iterations
"Tolerance"	10^{-10}	accuracy of the solution
"Derivatives"	0	0 ⇒ find root only 1 ⇒ find root together with the first derivatives of the root with respect to implicit variables 2 ⇒ 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 a x^3+ b x+c together with its first and second derivative with respect to a, b, and c

In[1]:=
```
<< AceGen`;
```

In[21]:=
```
SMSInitialize["tmptest", "Environment" → "MathLink"];
SMSModule["test", Real[a$$, b$$, c$$, x0$$, tol$$, x$$, dxdfi$$[3], d2xdfi2$$[3, 3]],
  Integer[maxNR$$, NR$$], "Input" → {a$$, b$$, c$$, x0$$, tol$$, maxNR$$},
  "Output" → {x$$, dxdfi$$, d2xdfi2$$, NR$$}];
{a, b, c, tol} ⊢ SMSReal[{a$$, b$$, c$$, tol$$}];
x ⫤ SMSReal[x0$$];
```

In[25]:=

```
ϕ = {a, b, c};
maxNR ⊢ SMSInteger[maxNR$$];
{x, conv, NR} ⊨ SMSFindRoot[a x^3 + b x + c, {x, x},
   "Derivatives" → 2, "MaxIterations" → maxNR, "Tolerance" → tol];
SMSExport[x, x$$];
```

In[29]:=

```
Δx ⊨ SMSD[x, ϕ];
SMSExport[Δx, dxdfi$$];
ΔΔx ⊨ SMSD[Δx, ϕ];
SMSExport[ΔΔx, d2xdfi2$$];
SMSExport[NR, NR$$];
SMSWrite[];
```

File: tmptest.c Size: 3625 Time: 1

Method	test
No.Formulae	35
No.Leafs	387

In[35]:=

```
SMSInstallMathLink["Console" → False]
```

Out[•]=

```
{SMSSetLinkOption[tmptest, {i_Integer, j_Integer}],
 SMSLinkNoEvaluations[tmptest], test[a_?(Head[#1] == Real || Head[#1] == Integer &),
  b_?(Head[#1] == Real || Head[#1] == Integer &),
  c_?(Head[#1] == Real || Head[#1] == Integer &),
  x0_?(Head[#1] == Real || Head[#1] == Integer &),
  tol_?(Head[#1] == Real || Head[#1] == Integer &),
  maxNR_?(Head[#1] == Real || Head[#1] == Integer &)]}
```

In[36]:=

```
test[1, 1, 1, -0.68, 10.^-12, 10]
```

Out[•]=

```
{-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}
```

- verification

In[37]:=

```
x =.;
rr = Solve[aa x^3 + bb x + cc == 0, x][[1, 1, 2]]
```

Out[•]=

$$-\frac{\left(\frac{2}{3}\right)^{1/3} bb}{\left(-9\, aa^2\, cc + \sqrt{3}\, \sqrt{4\, aa^3\, bb^3 + 27\, aa^4\, cc^2}\right)^{1/3}} + \frac{\left(-9\, aa^2\, cc + \sqrt{3}\, \sqrt{4\, aa^3\, bb^3 + 27\, aa^4\, cc^2}\right)^{1/3}}{2^{1/3} \times 3^{2/3}\, aa}$$

In[39]:=

```
dd = D[rr, {{aa, bb, cc}}];
```

In[40]:=

```
ddd = D[dd, {{aa, bb, cc}}];
```

In[41]:=

```
{rr, dd, ddd} /. {aa → 1., bb → 1, cc → 1}
```

Out[•]=

```
{-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}}}
```

Numerical root finding of a system of nonlinear equations

SMSFindRoot[Function[{**h**}, {**Q** (**h**), *data* }], $\mathbf{h}_0$]	inserts a code that searches for a numerical root of a system of nonlinear, equations $\mathbf{Q}(\mathbf{h})$. The iterative procedure starts from the point $\mathbf{h}_0$ (predictor) and it uses the standard Newton–Raphson corrector

Given pure function must return for the given input parameters {h,hn,λ} the following vector:
{
Q … vector of equations
,*data* - arbitrary data
}

Function SMSFindRoot returns a vector:
{
h calculated root (**Q(h)=0**),
depending on reqired derivatives:
Null ⇒if only root is required

$\frac{\partial \mathbf{h}}{\partial \mathbf{r}}$ ⇒ first derivatives of the root with respect to given set of implicit variables
True if convergence was obtained and False if not,
number of iterations performed,
data - an arbitrary data returned by the given pure function
}

option name	*default value*	
"MaxIterations"	30	maximum number of iterations
"Tolerance"	10^{-10}	accuracy of the solution
"ImplicitVariables"	{}	a set of variables **r** on which the equations depend, thus $\mathbf{Q}(\mathbf{h}(\mathbf{r}),\mathbf{r})$ for which derivatives of the root $\frac{\partial \mathbf{h}}{\partial \mathbf{r}}$ have to be calculated
"Derivatives"	0	0 ⇒ find root only 1 ⇒ find root together with the first derivatives of the root with respect to implicit variables $\frac{\partial \mathbf{h}}{\partial \mathbf{r}}$ 2 ⇒ **CURRENTLY NOT SUPPORTED!**
"SkipIterations"	False	assume that predictor $\mathbf{h}_0$ is the correct solution and calculate the implicit derivatives $\frac{\partial \mathbf{h}}{\partial \mathbf{r}}$ only

Options for the solution of system of equations.

Example: Axisymmetric, finite strain elasto-plastic element .

Example

Generate and verify the *MathLink* program that returns solution to the system of nonlinear equations:

$$\Phi=\begin{pmatrix} a\,x\,y+x^3=0 \\ a-x\,y^2=0 \end{pmatrix}$$

where x and y are unknowns and a is parameter.

In[22]:=

```
<< AceGen`;
SMSInitialize["test", "Environment" → "MathLink"];
SMSModule["test", Real[x$$, y$$, a$$, tol$$, dx$$, dy$$], Integer[n$$],
  "Input" → {x$$, y$$, a$$, tol$$, n$$},
  "Output" → {x$$, y$$, dx$$, dy$$}];
{x0, y0, a, ϵ} ⊢ SMSReal[{x$$, y$$, a$$, tol$$}];
nmax ⊢ SMSInteger[n$$];
{{xc, yc}, dx, conv, niter, data} = SMSFindRoot[
   Function[{h},
    {x, y} = h;
    {{a x y + x^3, a - x y^2}, Null}
   ]
   , {x0, y0}
   , "MaxIterations" → nmax
   , "Tolerance" → ϵ
   , "Derivatives" → 1
   , "ImplicitVariables" → {a}
  ];
SMSIf[Not[conv], SMSPrintMessage["no convergence"];];
SMSExport[{xc, yc}, {x$$, y$$}];
SMSExport[dx // Flatten, {dx$$, dy$$}];
SMSWrite[];
```

File: test.c Size: 2858 Time: 0

Method	test
No.Formulae	26
No.Leafs	243

- 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`)

In[32]:=

```
SMSInstallMathLink[]
```

Out[•]=

```
{SMSSetLinkOption[test, {i_Integer, j_Integer}],
 SMSLinkNoEvaluations[test], test[x_?(Head[#1] == Real || Head[#1] == Integer &),
  y_?(Head[#1] == Real || Head[#1] == Integer &),
  a_?(Head[#1] == Real || Head[#1] == Integer &),
  tol_?(Head[#1] == Real || Head[#1] == Integer &),
  n_?(Head[#1] == Real || Head[#1] == Integer &)]}
```

- 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.

In[33]:=

```
test[1.9, -1.2, 3., 0.0001, 10]
```

Out[•]=

```
{1.93318, -1.24573, 0.386636, -0.0830487}
```

In[34]:=

```
x =.; y =.; a =.;
sol = Solve[{a x y + x^3 == 0, a - x y^2 == 0}, {x, y}]
```

Out[ ]=

$$\{\{x \to a^{3/5},\ y \to -a^{1/5}\},\ \{x \to -(-1)^{3/5} a^{3/5},\ y \to (-1)^{1/5} a^{1/5}\},\ \{x \to -(-1)^{1/5} a^{3/5},\ y \to -(-1)^{2/5} a^{1/5}\},\ \{x \to (-1)^{4/5} a^{3/5},\ y \to (-1)^{3/5} a^{1/5}\},\ \{x \to (-1)^{2/5} a^{3/5},\ y \to -(-1)^{4/5} a^{1/5}\}\}$$

In[36]:=

```
{sol[[1]], D[sol[[1]], a]} /. a → 3.
```

Out[ ]=

$$\{\{x \to 1.93318,\ y \to -1.24573\},\ \{0 \to 0.386636,\ 0 \to -0.0830487\}\}$$

Numerical root finding of a system of nonlinear, path dependent, parameterized equations

SMSFindRoot[
Function[{$\mathbf{h}$,$\mathbf{h}_n$,λ}, {$\mathbf{Q}(\mathbf{h},\mathbf{h}_n,\lambda)$,True,True,data}]
,$\mathbf{h_0}$,$\mathbf{h_s}$]

inserts a code that searches for a numerical root of a system of nonlinear, path dependent, parameterized equations $\mathbf{Q}(\mathbf{h},\mathbf{h}_n,\lambda)=\mathbf{0}$ where parameter $\lambda\in[0,1]$, where $\mathbf{h}_s=\mathbf{h}|_{\lambda=0}$ is known solution at $\lambda=0$. The iterative procedure starts from the point $\mathbf{h}_0$ (predictor) and it uses the standard Newton-Raphson corrector to calculate $\mathbf{h}|_{\lambda=1}$. 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".

Note that the index n+1 is avoided, thus $\mathbf{h}=\mathbf{h}_{n+1}$ and that commonly $\mathbf{h_0}=\mathbf{h_s}$.

SMSFindRoot[
Function[{$\mathbf{h}$,$\mathbf{h}_n$,λ}, {$\mathbf{Q}(\mathbf{h},\mathbf{h}_n,\lambda)$,f$(\mathbf{h},\mathbf{h}_n,\lambda)$, p$(\mathbf{h},\mathbf{h}_n,\lambda)$, *data*}]
,$\mathbf{h}_0$,$\mathbf{h}_s$]

inserts a code that searches for a numerical root of a system of nonlinear, path dependent, parameterized equations

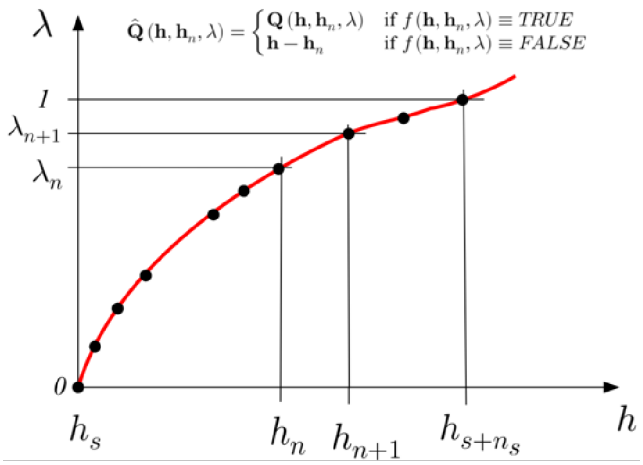
$$\hat{\mathbf{Q}}(\mathbf{h},\mathbf{h}_n,\lambda)=\begin{cases}\mathbf{Q}(\mathbf{h},\mathbf{h}_n,\lambda) & f(\mathbf{h},\mathbf{h}_n,\lambda)=\text{True}\\ \mathbf{h}-\mathbf{h}_n & f(\mathbf{h},\mathbf{h}_n,\lambda)=\text{False}\end{cases}=\mathbf{0}$$

where parameter $\lambda\in[0,1]$, $\mathbf{h}_s=\mathbf{h}|_{\lambda=0}$ is known solution at $\lambda=0$. The iterative procedure starts from the point $\mathbf{h}_0$ (predictor) and it uses the standard Newton-Raphson corrector to calculate $\mathbf{h}|_{\lambda=1}$. If convergence is not achieved in one step, then the algorithm splits the parameter interval $\lambda\in[0,1]$ into several sub-steps adaptively.

If condition $f(\mathbf{h},\mathbf{h}_n,\lambda)$=False then for the given sub-step the iterations are avoided and $\mathbf{h}=\mathbf{h}_n$.

If an additional condition $p(\mathbf{h},\mathbf{h}_n,\lambda)$=False then step cutting is performed regardless on convergence (solution is not acceptable from the physical point of view).

An additional arbitrary data *data* is passed unchanged form the given user function as an output of the SMSFindRoot function.



Given pure function must return for the given input parameters (**h,** $\mathbf{h}_n$, λ) the following vector:

{**Q** (**h,** $\mathbf{h}_n$, λ) … vector of equations

,f $(\mathbf{h}, \mathbf{h}_n, \lambda)$ ... if f $(\mathbf{h}, \mathbf{h}_n, \lambda)$ = True then **Q** (**h,** $\mathbf{h}_n$, λ) = **0** else **h** = $\mathbf{h}_n$

,p (**h,** $\mathbf{h}_n$, λ) ... if p $(\mathbf{h}, \mathbf{h}_n, \lambda)$ = False then step cutting is performed regardless on convergence (solution is not acceptable from the physical point of view)

,an arbitrary data

}

Function SMSFindRoot returns a vector:

{

h - calculated root,

$\partial \mathbf{h} / \partial \mathbf{r}$ - derivatives of the root with respect to given set of implicit variables,

expression that yields True if the problem was solved and False if not,

actual number of sub-steps performed,

the last arbitrary data returned by given user function

}

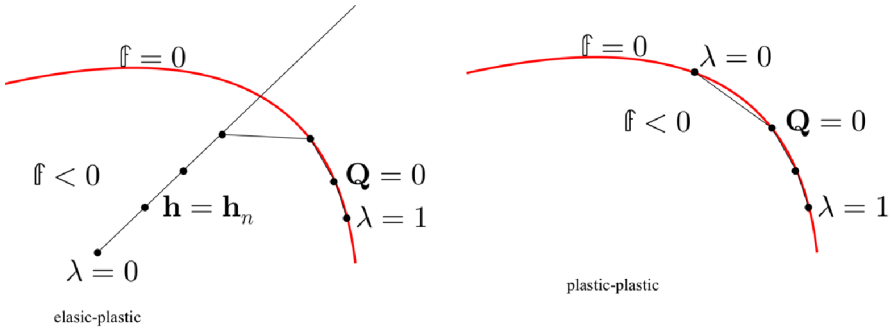
option name	*default value*	
"MaxIterations"	30	maximum number of iterations
"Tolerance"	10^{-10}	accuracy of the solution
"ImplicitVariables"	{}	a set of variables **r** on which the equations depend, thus **Q** $(\mathbf{h}(\mathbf{r}), \mathbf{h}_n, \lambda, \mathbf{r})$ for which derivatives of the root $\partial\mathbf{h}/\partial\mathbf{r}$ have to be calculated
"Derivatives"	1	0 ⇒ find root only, **CURRENTLY NOT SUPPORTED!** 1 ⇒ find root together with the first derivatives of the root with respect to implicit variables $\partial\mathbf{h}/\partial\mathbf{r}$ 2 ⇒ **CURRENTLY NOT SUPPORTED!**
"MinSubsteps"	1	minimum number of sub−steps
"MaxSubsteps"	10	maximum number of sub−steps
"SkipIterations"	False	assume that predictor $\mathbf{h}_0$ is the correct solution and calculate the implicit derivatives $\partial\mathbf{h}/\partial\mathbf{r}$

Options for the solution of parameterized system of equations.

Example: elasto - plastic problems

- Let $\mathbb{f}$ (**h,** $\mathbf{h}_n$) be a yield function, $\boldsymbol{\epsilon}_k$ and $\boldsymbol{\epsilon}_{k+1}$ total strains at the last converged global step k and current step k + 1, then for the last calculated local sub-step n and the current local sub-step n + 1, a set of time discretized plastic evolution equations **Q** and corresponding state variables **h** is given by:
  $\boldsymbol{\epsilon} = \lambda\, \boldsymbol{\epsilon}_{k+1} + (\lambda - 1)\, \boldsymbol{\epsilon}_k$ ... parameterization of local plastic equations

$\boldsymbol{\sigma}=\boldsymbol{\sigma}(\boldsymbol{\epsilon}-\boldsymbol{\epsilon}^{p})$;
$\mathcal{Z} = \boldsymbol{\epsilon}^{p} - \boldsymbol{\epsilon}^{p}_{n} - (\lambda_{m} - \lambda_{m,n})\,(\partial \mathbb{f} / \partial \boldsymbol{\sigma})$
$\mathbf{Q} = \{\mathcal{Z}_{1,1}, \mathcal{Z}_{2,2}, \mathcal{Z}_{3,3}, \mathcal{Z}_{1,2}, \mathcal{Z}_{1,3}, \mathcal{Z}_{2,3}, \mathbb{f}\} = 0$
$\mathbf{h} = \{\epsilon^{p}_{1,1}, \epsilon^{p}_{2,2}, \epsilon^{p}_{3,3}, \epsilon^{p}_{1,2}, \epsilon^{p}_{1,3}, \epsilon^{p}_{2,3}, \lambda_{m}\}$



- The sub-step is in elastic regime when $\mathbb{f}\,(\mathbf{h}, \mathbf{h}_n) < 0$. In that case $\mathbf{h} = \mathbf{h}_n$ leading to additional condition $f\,(\mathbf{h}, \mathbf{h}_n) = \mathbb{f}\,(\mathbf{h}, \mathbf{h}_n) < 0$.
- Plastic process is dissipative leading to additional condition on physical admissibility of the solution $p\,(\mathbf{h}, \mathbf{h}_n) = \lambda_m - \lambda_{m,n} > 0$.
- For the actual AceGen inputs look at the elasto-plastic elements available in AceShare.

Mechanics of Solids

SMSLameToHooke[λ,G]	transform Lame' s constants λ, G (μ≡G) to Hooke' s constants E, ν
SMSHookeToLame[E, ν]	transform Hooke' s constants E, ν to Lame' s constants λ, G
SMSHookeToBulk[E,ν]	transform Hooke' s constants E, ν to shear modulus G and bulk modulus K
SMSBulkToHooke[G,K]	transform shear modulus G and bulk modulus K to Hooke' s constants E, ν

Transformations of mechanical constants in mechanics of solids.

- This transforms Lame's constants λ, G to Hooke's constants E, ν. No simplification is preformed!

In[16]:=

```
SMSLameToHooke[λ, G] // Simplify
```

Out[•]=

$$\left\{\frac{G\,(2\,G+3\,\lambda)}{G+\lambda},\ \frac{\lambda}{2\,(G+\lambda)}\right\}$$

SMSPlaneStressMatrix[E, ν]	linear elastic plane strain constitutive matrix for the Hooke' s constants E, ν
SMSPlaneStrainMatrix[E, ν]	linear elastic plane stress constitutive matrix for the Hooke' s constants E, ν

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, ν] // MatrixForm
```

Out[•]//MatrixForm=

$$\begin{pmatrix} \frac{e}{1-\nu^2} & \frac{e\,\nu}{1-\nu^2} & 0 \\ \frac{e\,\nu}{1-\nu^2} & \frac{e}{1-\nu^2} & 0 \\ 0 & 0 & \frac{e}{2\,(1+\nu)} \end{pmatrix}$$

CHAPTER 4

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
  - Embeded 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 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

sc=SMSCall["sub",p_1,p_2,...]
returns auxiliary variable *sc* that represents the call of external subroutine *sub* with the given set of input and output parameters

option	default	description
"ArgumentsByValue"	False	By default all arguments are passed to subroutine by reference. This can be changed with "ArgumentsByValue"-> True option. ONLY FOR ADVANCED USERS!!
"Position"	"Current"	Regulates how the "export" object is treated during the code optimisation. "Current" ⇒ the object remains at the position in the program where it was called "Optimal" ⇒ optimal position is determined based on the dependency of *exp* and *ext* on other variables "Automatic" ⇒ "Optimal"
"Subordinate"-> {v_1, v_2 ...}	{}	list of auxiliary variables that represent control structures (e.g. SMSVerbatim, SMSExport) that have to be executed before the evaluation of the current expression
"System"-> *truefalse*	True	the subroutine that is called has been generated by *AceGen*

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.

"type"[a$$]	type "type" I/O parameter
"type"[a$$[3]]	array of type "type" I/O parameters
"string"	strings are included verbatim into list of arguments and into list of declarations
Protected[*expr*]	protected expression is transformed into string and included verbatim into list of parameters, but not into the list of declarations

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. x⊨SMSReal[x$$,"Subordinate"→*sc*], ni⊨SMSInteger[i$$[5],"Subordinate"→*sc*], b⊨SMSLogical[b$$,"Subordinate"→*sc*]). The proper order of evaluation of expressions is assured by the "Subordinate"→*sc* option where the parameter *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 `SMSFreeze`.

Examples

Intrinsic user function 1: Scalar function exists but has different syntax in source code language

In[52]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "Fortran"];
```

- This is an additional definition of output format for function tangent.

In[54]:=

```
SMSAddFormat[
  Tan[i_] :→ 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[];
```

File:	test.f	Size: 761
Methods	No.Formulae	No.Leafs
sub1	1	7

The final code can also be formatted by the "Substitutions" option of the `SMSWrite` command.

In[59]:=
```
FilePrint["test.f"]
```

```
!*************************************************************
!* AceGen    2.502 Windows (5 Nov 10)                        *
!*           Co. J. Korelc  2007            5 Nov 10 10:53:53 *
!*************************************************************
! 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

!******************** S U B R O U T I N E *********************
      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 example below). Consequently, if the first derivatives are not derived and exported to the main module, then the first derivatives (and all higher derivatives as well) will be 0. If the first derivatives are defined and higher derivatives are not then in general the higher derivatives of the general function can be nonzero (see example below), however they are incorrect. NO WARNING is given about the possibility of incorrect derivatives.

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] = $y \frac{\text{MyPower}[x,y]}{x}$,
  D[MyPower[x,y],y] = MyPower[x, y] Log[x].

In[88]:=
```
<< AceGen`;
SMSInitialize["test", "Language" → "C"];
```

- This is an additional definition of output format for function MyPower.

In[90]:=
```
SMSAddFormat[
  MyPower[i_, j_] :→ Switch[SMSLanguage, "Mathematica", i^j, "Fortran", i^j, "C", "Power"[i, j]]
 ];
```

- Here the derivatives of MyPower with respect to all parameters are defined.

In[91]:=
```
Unprotect[Derivative];
Derivative[1, 0][MyPower][i_, j_] := j MyPower[i, j] / i;
Derivative[0, 1][MyPower][i_, j_] := MyPower[i, j] Log[i];
Protect[Derivative];
```

- Here is defined the numerical evaluation of MyPower with the *p*-digit precision.

In[95]:=
```
N[MyPower[i_, j_], p_] := i^j;
```

In[96]:=
```
SMSModule["sub1", Real[x$$, y$$, z$$]];
x ⊢ SMSReal[x$$];
y ⊢ SMSReal[y$$];

SMSExport[SMSD[MyPower[x, y], x], z$$];
```

In[100]:=
```
SMSWrite[];
```

File:	test.c	Size: 729
Methods	No.Formulae	No.Leafs
sub1	1	22

In[101]:=
```
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"

/******************* S U B R O U T I N E *********************/
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 u**ser AceGen module** $f = \mathrm{Sin}(a_1 x + a_2 x\wedge 2 + a_3 x^3)$ with an input parameter *x* and constants a[3] and the output parameters y = f(x) and first dy = $\frac{\partial f}{\partial x}$ derivatives.

In[326]:=
```
SMSModule["f", Real[x$$, a$$[3], y$$, dy$$]];
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];
SMSExport[y, y$$];
SMSExport[dy, dy$$];
```

- This generates subroutine *mf* that calls subroutine *f*.

In[333]:=
```
SMSModule["mf", Real[w$$, r$$]];
w ⊢ SMSReal[w$$];
```

- This use of ⊢ operator here is obligatory to ensure that auxiliary variables is generated that can be used later for the definition of the partial derivatives.

In[335]:=
```
z ⊢ w^2;
```

- The `SMSCall` commands inserts into the generated source code the call of external subroutine with the given set of input and output parameters. All the arguments are passed to subroutine by reference (pointer). Input arguments are first assigned to an additional auxiliary variables before they are passed to subroutine. `SMSCall` returns auxiliary variable *fo* that represents the call of external subroutine *f* .

In[336]:=
```
fo = SMSCall["f", z, {1/2, 1/3, 1/4}, Real[y$$], Real[dy$$]];
```

- 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[338]:=
```
f ⊢ SMSReal[y$$, "Subordinate" → fo, "Dependency" → {z, dfdz}];
```

- First derivatives are derived and displayed here.

In[339]:=
```
dw ⊨ SMSD[f, w];
SMSRestore[dw, "Global"]
```

Out[•]=

2 dfdz w

- 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[dw, w];
SMSRestore[ddw, "Global"]
```

Out[•]=

2 dfdz

In[343]:=
```
SMSExport[dw, dy$$];
SMSWrite[];
```

File:	test.c	Size: 1255
Methods	No.Formulae	No.Leafs
f	6	84
main	4	46

In[345]:=

```
FilePrint["test.c"]
```

```
/*************************************************************
* AceGen    5.001 Windows (3 Jan 13)                         *
*           Co. J. Korelc  2007            3 Jan 13 13:55:39 *
*************************************************************
User     : USER
Notebook : AceGenTutorials.nb
Evaluation time                 : 0 s     Mode  : Optimal
Number of formulae              : 10      Method: Automatic
Subroutine                      : f size :84
Subroutine                      : main size :46
Total size of Mathematica  code : 130 subexpressions
Total size of C code            : 594 bytes*/
#include "sms.h"

/******************* S U B R O U T I N E *********************/
void f(double v[5001],double (*x),double a[3],double (*y),double (*dy))
{
v[24]=a[2]*Power((*x),3);
v[23]=a[0]*(*x);
v[22]=Power((*x),2);
v[6]=a[1]*v[22]+v[23]+v[24];
(*y)=sin(v[6]);
(*dy)=(a[0]+3e0*a[2]*v[22]+2e0*a[1]*(*x))*cos(v[6]);
};

/******************* S U B R O U T I N E *********************/
void main(double v[5001],double (*w),double (*r))
{
double dy;double v01;double y;double v02[3];
v01=Power((*w),2);
v02[0]=0.5e0;
v02[1]=0.3333333333333333e0;
v02[2]=0.25e0;
f(&v[5009],&v01,v02,&y,&dy);
(*dy)=2e0*dy*(*w);
};
```

User AceGen module 2: Definition of the user subroutine and first and second derivatives

- This generates u**ser AceGen module** $f = \mathrm{Sin}(a_1 x + a_2 x\wedge 2 + a_3 x^3)$ with an input parameter *x* and constants a[3] and the output parameters y = f(x) and first $dy = \frac{\partial f}{\partial x}$ and second $ddy = \frac{\partial^2 f}{\partial x^2}$ derivatives.

In[346]:=

```
<< 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[ ]=

$$\left\{2\,\boxed{\text{dfdz}}\,\boxed{\text{w}},\ 2\,\boxed{\text{dfdz}} + 4\,\boxed{\text{dfdz2}}\,\boxed{\text{w}}^2\right\}$$

In[365]:=

```
SMSExport[{dw, ddw}, {dy$$, ddy$$}];
```

In[366]:=

```
SMSWrite[];
```

File:	test.c	Size: 1397
Methods	No.Formulae	No.Leafs
f	7	91
main	6	81

In[367]:=

```
FilePrint["test.c"]
```

```
/*************************************************************
* AceGen    5.001 Windows (3 Jan 13)                        *
*           Co. J. Korelc  2007            3 Jan 13 13:55:51 *
*************************************************************
User     : USER
Notebook : AceGenTutorials.nb
Evaluation time                 : 1 s     Mode  : Optimal
Number of formulae              : 13      Method: Automatic
Subroutine                      : f size :91
Subroutine                      : main size :81
Total size of Mathematica  code : 172 subexpressions
Total size of C code            : 729 bytes*/
#include "sms.h"

/******************* S U B R O U T I N E *********************/
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);
v[6]=a[1]*v[25]+v[26]+v[27];
v[8]=(a[0]+3e0*a[2]*v[25]+v[28])*cos(v[6]);
(*y)=sin(v[6]);
(*dy)=v[8];
(*ddy)=v[8];
};

/******************* S U B R O U T I N E *********************/
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

In[58]:=

```
Export["Energy.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. - (C2*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);
 }", "Text"]
```

Out[•]=

```
Energy.c
```

and the C header file "Energy.h" with the following contents

In[59]:=

```
Export["Energy.h", " void Energy (double *I1p, double *I3p, double *C1p, double *C2p,
   double *C3p, double *pi, double dp[2], double ddp[2][2])", "Text"];
```

Subroutine Energy calculates the strain energy Π(I1,I3) where I1 and I3 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 I1 and I2.

- 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.

In[74]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "C"];
SMSModule["Stress", 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];
ddp ⊢ SMSReal[Table[ddp$$[i, j], {i, 2}, {j, 2}], "Subordinate" → pcall];
dp ⊢ SMSReal[Table[dp$$[i], {i, 2}], "Subordinate" → pcall, "Dependency" → {{I1, I3}, ddp}];
Π ⊢ SMSReal[pi$$, "Subordinate" → pcall, "Dependency" → {{I1, dp[[1]]}, {I3, dp[[2]]}}];
S ⊨ 2 SMSD[Π, ℂ, "Symmetric" → True];
SMSExport[S, S$$];
SMSWrite["Splice" → {"Energy.c"}];
```

File:	test.c	Size: 2117
Methods	No.Formulae	No.Leafs
Stress	21	361

In[88]:=

```
FilePrint["test.c"]
```

```
/*************************************************************
* AceGen    3.306 Windows (25 Aug 12)                       *
*           Co. J. Korelc  2007            26 Aug 12 12:54:54*
*************************************************************
User     : USER
Notebook : AceGenTutorials.nb
Evaluation time                 : 1 s     Mode  : Optimal
Number of formulae              : 21      Method: Automatic
Subroutine                      : Stress size :361
Total size of Mathematica  code : 361 subexpressions
Total size of C code            : 1012 bytes*/
#include "sms.h"

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. - (C2*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);
 }

/******************* S U B R O U T I N E *********************/
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[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]=dp[0];
v[26]=dp[1];
v[31]=v[26]*(v[41]+v[43]);
v[32]=v[26]*(-(C[1][1]*v[44])+C[1][2]*v[45]);
v[35]=v[26]*(C[0][1]*v[44]+v[46]);
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];
S[2][2]=2e0*(v[25]+v[26]*v[47]);
};
```

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.

In[89]:=

```
<< 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];
ddp ⊢ SMSReal[Table[ddp$$[i, j], {i, 2}, {j, 2}], "Subordinate" → pcall];
dp ⊢ SMSReal[Table[dp$$[i], {i, 2}], "Subordinate" → pcall, "Dependency" → {{I1, I3}, ddp}];
Π ⊢ SMSReal[pi$$, "Subordinate" → pcall, "Dependency" → {{I1, dp[[1]]}, {I3, dp[[2]]}}];
S ⊨ 2 SMSD[Π, ℂ, "Symmetric" → True];
SMSExport[S, S$$];
SMSWrite["IncludeHeaders" → {"Energy.h"}];
```

File:	test.c	Size: 1662
Methods	No.Formulae	No.Leafs
main	21	361

In[103]:=

```
FilePrint["test.c"]
```

```
/*************************************************************
* AceGen    3.306 Windows (25 Aug 12)                        *
*           Co. J. Korelc  2007            26 Aug 12 12:55:05*
*************************************************************
User     : USER
Notebook : AceGenTutorials.nb
Evaluation time                 : 1 s     Mode  : Optimal
Number of formulae              : 21      Method: Automatic
Subroutine                      : main size :361
Total size of Mathematica  code : 361 subexpressions
Total size of C code            : 1010 bytes*/
#include "Energy.h"
#include "sms.h"

/******************* S U B R O U T I N E *********************/
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]=dp[0];
v[26]=dp[1];
v[31]=v[26]*(v[41]+v[43]);
v[32]=v[26]*(-(C[1][1]*v[44])+C[1][2]*v[45]);
v[35]=v[26]*(C[0][1]*v[44]+v[46]);
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];
S[2][2]=2e0*(v[25]+v[26]*v[47]);
};
```

Embeded sequence with *SMSVerbatim* command: User defined source code is embedded into generated source code

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 Π(I1,I3) and first derivative of the strain energy with respect to the invariants I1 and I2.

In[341]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "C"];
SMSModule["Stress", 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]};

pcode = SMSVerbatim[
   "I1c=", I1, ";I3c=", I3, ";\nC1c=", C1, ";C2c=", C2, ";C3c=", C3, ";",
"\npi = (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[C1c$$, C2c$$, C3c$$, I1c$$, I3c$$, pi$$, dp$$[2]]}];

dp ⊢ SMSReal[Table[dp$$[i], {i, 2}], "Subordinate" → pcode];

Π ⊢ SMSReal[pi$$, "Subordinate" → pcode, "Dependency" → {{I1, dp[[1]]}, {I3, dp[[2]]}}];

S ⊨ 2 SMSD[Π, ℂ, "Symmetric" → True];
SMSExport[S, S$$];
SMSWrite[];
```

File:	test.c	Size: 1703
Methods	No.Formulae	No.Leafs
Stress	16	350

In[354]:=

```
FilePrint["test.c"]
```

```
/*************************************************************
* AceGen    6.002 Windows (25 Jul 13)                        *
*           Co. J. Korelc  2013           9 Aug 13 18:29:23  *
*************************************************************
User     : USER
Notebook : AceGenTutorials.nb
Evaluation time                 : 1 s     Mode  : Optimal
Number of formulae              : 16      Method: Automatic
Subroutine                      : Stress size: 350
Total size of Mathematica  code : 350 subexpressions
Total size of C code            : 1071 bytes */
#include "sms.h"

/******************* S U B R O U T I N E *********************/
void Stress(double v[43],double C[3][3],double S[3][3],double (*C1)
     ,double (*C2),double (*C3))
{
double C1c;double C2c;double C3c;double I1c;double I3c;double pi;double dp[2];
v[37]=-2e0*C[0][0]*C[1][2];
v[36]=2e0*C[0][1];
v[35]=2e0*C[0][2];
v[34]=-(C[2][2]*v[36]);
v[33]=Power(C[1][2],2);
v[32]=C[1][2]*v[35];
v[31]=Power(C[0][2],2);
v[30]=Power(C[0][1],2);
v[38]=C[0][0]*C[1][1]-v[30];
I1c=C[0][0]+C[1][1]+C[2][2];I3c=(-(C[1][1]*v[31])+C[0][1]*v[32]-C[0][0]*v[33]+C[2][2]*v[38]);
C1c=(*C1);C2c=(*C2);C3c=(*C3);
pi = (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);
v[16]=dp[0];
v[17]=dp[1];
v[22]=v[17]*(v[32]+v[34]);
v[23]=v[17]*(-(C[1][1]*v[35])+C[1][2]*v[36]);
v[26]=v[17]*(C[0][1]*v[35]+v[37]);
S[0][0]=2e0*(v[16]+v[17]*(C[1][1]*C[2][2]-v[33]));
S[0][1]=v[22];
S[0][2]=v[23];
S[1][0]=v[22];
S[1][1]=2e0*(v[16]+v[17]*(C[0][0]*C[2][2]-v[31]));
S[1][2]=v[26];
S[2][0]=v[23];
S[2][1]=v[26];
S[2][2]=2e0*(v[16]+v[17]*v[38]);
};
```

MathLink program with modules

In[103]:=

```
<< AceGen`;
SMSInitialize["test", "Environment" → "MathLink"];
```

- This generates the user AceGen module "f" that defines function $f(x) = \sum_{i=1}^{n} a_i x^{i-1}$ 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 $ddy = \frac{\partial^2 f}{\partial x^2}$.

In[105]:=

```
SMSModule["f", Real[x$$, a$$["*"]], Integer[n$$], Real[y$$, dy$$, ddy$$]];
x ⊢ SMSReal[x$$];
n ⊢ SMSInteger[n$$];
y ⫤ 0;
SMSDo[
  y ⫤ y + SMSReal[a$$[i]] x^(i-1);
  , {i, 1, n, 1, y}];
dy ⊨ SMSD[y, x];
ddy ⊨ SMSD[dy, x];
SMSExport[{y, dy, ddy}, {y$$, dy$$, ddy$$}];
```

- 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 $a = \{1/2, 1/3, 1/4\}$.

In[113]:=

```
SMSModule["mlinkA", Real[w$$, ddw$$], "Input" → w$$, "Output" → ddw$$];
w ⊢ SMSReal[w$$];
z ⊢ Exp[w];
fcall = SMSCall["f", z, {1/2, 1/3, 1/4}, SMSInteger[3], Real[y$$], Real[dy$$], Real[ddy$$]];
dfdz2 ⊢ SMSReal[ddy$$, "Subordinate" → fcall];
dfdz ⊢ SMSReal[dy$$, "Subordinate" → fcall, "Dependency" → {z, dfdz2}];
f ⊢ SMSReal[y$$, "Subordinate" → fcall, "Dependency" → {z, dfdz}];
dw ⊨ SMSD[f, w];
ddw = SMSD[dw, w];
SMSExport[ddw, ddw$$];
```

- 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.

In[123]:=

```
SMSModule["mlinkB", Real[w$$, a$$["*"]], Integer[n$$],
  Real[ddw$$], "Input" → {w$$, a$$, n$$}, "Output" → ddw$$];
w ⊢ SMSReal[w$$];
z ⊢ Exp[w];
n ⊢ SMSInteger[n$$];
fcall = SMSCall["f", z, Real[a$$["*"]], n, Real[y$$], Real[dy$$], Real[ddy$$]];
dfdz2 ⊢ SMSReal[ddy$$, "Subordinate" → fcall];
dfdz ⊢ SMSReal[dy$$, "Subordinate" → fcall, "Dependency" → {z, dfdz2}];
f ⊢ SMSReal[y$$, "Subordinate" → fcall, "Dependency" → {z, dfdz}];
dw ⊨ SMSD[f, w];
ddw = SMSD[dw, w];
SMSExport[ddw, ddw$$];
```

```
SMSCall argument a$$ is also I/O argument. See also: SMSCall
```

In[134]:=

```
SMSWrite[];
```

File:	test.c	Size: 4519
Methods	No.Formulae	No.Leafs
f	11	97
mlinkA	6	77
mlinkB	5	66

In[135]:=

```
SMSInstallMathLink["test", "Console" → True]
```

```
{SMSSetLinkOption[test, {i_Integer, j_Integer}],
 SMSLinkNoEvaluations[test], f[x_?(Head[#1] == Real || Head[#1] == Integer &),
  a_?(ArrayQ[#1, 1, Head[#1] == Real || Head[#1] == Integer &] &),
  n_?(Head[#1] == Real || Head[#1] == Integer &),
  y_?(Head[#1] == Real || Head[#1] == Integer &),
  dy_?(Head[#1] == Real || Head[#1] == Integer &),
  ddy_?(Head[#1] == Real || Head[#1] == Integer &)],
 mlinkA[w_?(Head[#1] == Real || Head[#1] == Integer &)],
 mlinkB[w_?(Head[#1] == Real || Head[#1] == Integer &),
  a_?(ArrayQ[#1, 1, Head[#1] == Real || Head[#1] == Integer &] &),
  n_?(Head[#1] == Real || Head[#1] == Integer &)]}
```

- Test with MMA for $w = 2.5$

In[136]:=

```
w =.; x = Exp[w]; D[{1/2, 1/3, 1/4}.{1, x, x^2}, w, w] /. w → 2.5
```

```
152.474
```

- "mlinkA" and "mlinkB" give correct results.

In[137]:=

```
mlinkA[2.5]
```

```
152.474
```

In[138]:=

```
mlinkB[2.5, {1/2, 1/3, 1/4} // N, 3]
```

```
152.474
```

- Results is correct even when the length of a is more than 3, provided that only 3 coefficients are used.

In[139]:=

```
mlinkB[2.5, {1/2, 1/3, 1/4, 1/5, 1/6} // N, 3]
```

```
152.474
```

Arrays

Contents

- General Description
  - SMSArray
- Manipulating Arrays
  - SMSPart
  - SMSReplacePart
  - SMSArrayLength
  - SMSDot
  - 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[[*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:

In[44]:=
```
<< AceGen`;
SMSInitialize["test", "Language" → "C"];
SMSModule["test", Real[x$$[5]], Integer[i$$]];
x ⊢ SMSReal[x$$];
i ⊢ SMSInteger[i$$];
```

- Here a standard *Mathematica* vector is defined and a vector of new auxiliary variables is created to store the result.

In[49]:=
```
X ⊢ Table[SMSReal[x$$[i]], {i, 5}]
```

Out[ ]=

$\{x_1, x_2, x_3, x_4, x_5\}$

- Here the third component of the vector X is displayed.

In[50]:=
```
X[[3]]
```

Out[ ]=

x_3

- Note that an arbitrary symbol cannot be used as an index together with Mathematica arrays.

In[51]:=

```
X[[i]]
```

Part: The expression $V[2, 1] cannot be used as a part specification.

Part: The expression $V[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.

Out[ ]=

$$\{X_1, X_2, X_3, X_4, X_5\}[[i]]$$

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⊢SMSArray[{1,2,3400+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⊢SMSArray[10] allocates memory for the real array M with length 10 and SMSArray[M,10,π&] would allocates memory and initialize all elements of M to π). **General** arrays are always introduced as a multi-valued auxiliary variables.

In[239]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "C"];
SMSModule["test", Real[x$$[5]], Integer[i$$, n$$]];
x ⊢ SMSReal[x$$];
i ⊢ SMSInteger[i$$];
```

- Here a constant *AceGen* array is created. Result is a single auxiliary variable.

In[243]:=

```
X ⊢ SMSArray[Table[SMSReal[x$$[k]], {k, 5}]]
```

Out[ ]=

X

- In this case, the third component of Sin[X] cannot be accessed by *Mathematica* since X is a symbol not an array.

In[244]:=

```
X[[3]]
```

Part: Part 3 of $V[3, 1] does not exist.

Part: Part 3 of $V[3, 1] does not exist.

Part: Part 3 of X does not exist.

General: Further output of Part::partw will be suppressed during this calculation.

Out[•]=

$(X)[[3]]$

- The third component of Sin[X] can be accessed by *AceGen* using SMSPart command.

In[245]:=

```
SMSPart[X, 3]
```

Out[•]=

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.

In[246]:=

```
SMSPart[X, i]
```

Out[•]=

X_i

- Here a general array **M** of length *n* is created with the values set to Sin[i π/n],i=1,...,n.

In[247]:=

```
n ⊢ SMSInteger[n$$];
SMSArray[M, n, Sin[π # / n] &]
```

Out[•]=

$_4M$

- The third component of **M** can be accessed by *AceGen* using SMSPart command.

In[249]:=

```
SMSPart[M, 3]
```

Out[•]=

$_4M_3$

- Here, the 3-th element is not actually the 3-th element but an SMSIndexF object that points to the 3-th element.

In[250]:=

```
SMSPart[M, 4] // ToString
```

Out[•]=

```
SMSIndexF[$V[6, 4], 4, False,
  0.0746770859789364094115061671880629449214401393952135935465602901023361933293731 10
   1486112959003965038]
```

- General *i*-th component of **M**.

In[251]:=

```
SMSPart[M, i]
```

Out[•]=

$_4M_i$

- Here, the 3-th element is changed to Cos[2 π/n].

In[252]:=
```
M ← SMSReplacePart[M, Cos[3 π / n], 3]
```

Out[ ]=

$_5$M

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,...}]
create an array data object that represents a constant array of expressions {exp_1,exp_2,...}

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,…,*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]],...,*func*[*i*] [[*n*]]}, *i*=1,…,*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*.

option	default	description
"Type"	Automatic	Real ⇒ real type array Integer ⇒ integer type array Automatic ⇒ 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)

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[{exp_1,exp_2,...}] 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. r←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[•]=
```
Array[{x, x^2, 0, π}, Real]
```

General arrays: Definition of the general array only allocates space on the global vector. An *a*rray 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[•]=
```
Array[n, 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[•]=
```
4M
```

- 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[•]=
```
4M
```

- The length of the array is obtained by `SMSArrayLength`

In[12]:=
```
SMSArrayLength[M]
```
Out[•]=
```
n
```

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

SMSPart[{exp_1,exp_2,...},*index*]
create an index data object that represents the *index*-th element of the array of expressions {exp_1,exp_2,...} (only real type arrays!)

SMSPart[*array*,*index*]
create an index data object that represents the *index*-th element of the array of expressions represented by the array data object *array*

SMSArrayLength[*array*]
returns the length of the array *array*

SMSReplacePart[*array*, i->*new*]
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)

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. r←SMSPart[{1,2,3,4},i]).

Example: SMSPart

```
SMSInitialize["test"];
SMSModule["test", Real[x$$, r$$], Integer[i$$]];
x ← SMSReal[x$$];
i ← SMSInteger[i$$];
g ⊨ SMSArray[{x, x^2, 0, π}];
gi ⊨ SMSPart[g, i];
SMSExport[gi, r$$];
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$$_,r$$_,i$$_]:=Module[{},
$VV[5000]=x$$;
$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

Method	test
No.Formulae	5
No.Leafs	58

In[286]:=

```
FilePrint["test.c"]
```

```
/*************************************************************
* AceGen    7.007 Windows (17 Jan 20)                        *
*           Co. J. Korelc  2020           5 Feb 20 18:20:06  *
*************************************************************
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[$arrayo_1$,$arrayo_2$]

dot product of the two arrays of expressions represented by the array data objects $arrayo_1$ and $arrayo_2$

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$$]];
x ⊢ SMSReal[x$$];
n ⊢ SMSInteger[n$$];
m ⊢ SMSInteger[m$$];
```

- 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, 0, π}]
```

Out[ ]=

A

- 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[ ]=

B

- This calculates a dot product of vectors A and B

In[234]:=

```
dot = SMSDot[A, B]
```

Out[ ]=

A • B

- This creates an index to the *n*-th element of the second vector.

In[235]:=

```
Bn = SMSPart[B, n]
```

Out[ ]=

B_n

- The same can be obtained with command

In[236]:=

```
SMSExport[{dot, Bn}, {r$$, s$$}];
SMSWrite["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"]
```

```
!*************************************************************
!* AceGen    7.007 Windows (17 Jan 20)                       *
!*           Co. J. Korelc  2020           5 Feb 20 18:11:09  *
!*************************************************************
! User     : USER
! Notebook : AceGenTutorials
! Evaluation time                 : 0 s     Mode  : Optimal
! Number of formulae              : 3       Method: Automatic
! Subroutine                      : test size: 72
! Total size of Mathematica  code : 72 subexpressions
! Total size of Fortran code      : 448 bytes

!******************* S U B R O U T I N E *********************
      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)=0d0
      v(21)=0.3141592653589793d1
      r=SMSDot(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.

SMSEvaluateCellsWithTag[*tag*]
find and evaluate all notebook cells with the cell tag *tag*

SMSEvaluateCellsWithTag[*tag*,"Session"]
find and reevaluate notebook cells with the cell tag *tag* where search is limited to the cells that has already been evaluated once during the current AceGen session

option	default	description
"CollectInputStart"	False	starts the process of collecting the unevaluated contents of all the notebook cells evaluated by the SMSEvaluateCellsWithTag command during the session
"RemoveTag"	False	True ⇒ remove the tag *tag* from the cells included into recreated notebook All ⇒remove all tags from the cells included into recreated notebook
"CollectInput"	True	on False temporarily suspends the process of collecting cells for the current SMSEvaluateCellsWithTag call
"BlockStyle"	False	True ⇒ do not change cell style *level* ⇒ change style to given level
"Position"	-1	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]])
"CreateFunction"	False	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**)
"Active"	True	collected cells are evaluatable
"Echo"	True	print the *tag* to the current notebook
"Evaluate"	True	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

Options of the SMSEvaluateCellsWithTag function.

Example:

In[131]:=
CELLTAG

```
Print["this is cell with the tag CELLTAG"];
```

In[136]:=
```
<<AceGen`;
SMSInitialize["test", "Language" -> "C"];
SMSModule["sub1"];
SMSEvaluateCellsWithTag["CELLTAG"];
```

```
[0-0] Include Tag : CELLTAG (1 cells found, 1 evaluated)

this is cell with the tag CELLTAG
```

SMSRecreateNotebook

The *SMSRecreateNotebook* command creates a new notebook that includes an **unevaluated** 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.

SMSRecreateNotebook[]
creates a new notebook that includes cells evaluated by the SMSEvaluateCellsWithTag command during the AceGen session

option	default	description
"File"	current session name	notebook file name
"Head"	{}	list of additional Cells included at the head of the notebook
"Close"	False	close notebook after creation

Options of the SMSRecreateNotebook function.

Example:

In[131]:=
CELLTAG-1
```
Print["This is cell with the tag CELLTAG-1."];
```

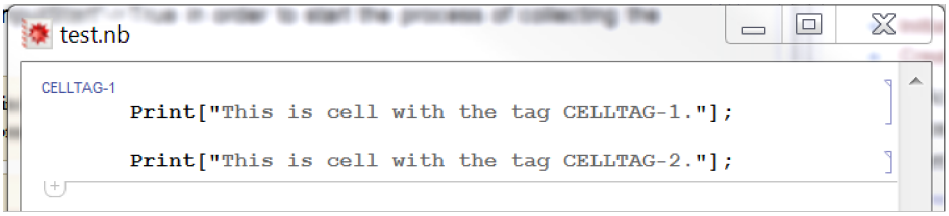
In[131]:=
CELLTAG-2
```
Print["This is cell with the tag CELLTAG-2."];
```

In[31]:=
```
<<AceGen`;
SMSInitialize["test", "Language" -> "C"];
SMSModule["sub1"];
SMSEvaluateCellsWithTag["CELLTAG-1","CollectInputStart"->True];
SMSEvaluateCellsWithTag["CELLTAG-2","RemoveTag"->True];
```

```
[0-0] Include Tag : CELLTAG-1 (1 cells found, 1 evaluated)

This is cell with the tag CELLTAG-1.

[0-0] Include Tag : CELLTAG-2 (1 cells found, 1 evaluated)

This is cell with the tag CELLTAG-2.
```

- 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.

In[36]:=
```
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,**

a) **recreated cell.**

Evaluation cell is directly evaluated immediately after 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*,$form_1$,$value_1$,$form_2$,$value_2$,...]
evaluates *switchexp*, then compares it with each of $form_i$ 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[$test_1$,$value_1$,$test_2$,$value_2$,...]
evaluates each of the $test_i$ in turn, choosing the value of the $value_i$ 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 *test* 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 ofthe 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:

In[49]:=
CELLTAG-3

```
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 = SMSTagEvaluate[1 + 1]; (*1+1 is evaluated before it is included in new notebook*)
```

In[75]:=

```
<<AceGen`;
SMSInitialize["test", "Language" -> "C"];
SMSModule["sub1"];
condition=True;
expr="C";
SMSEvaluateCellsWithTag["CELLTAG-3","CollectInputStart"->True];
```

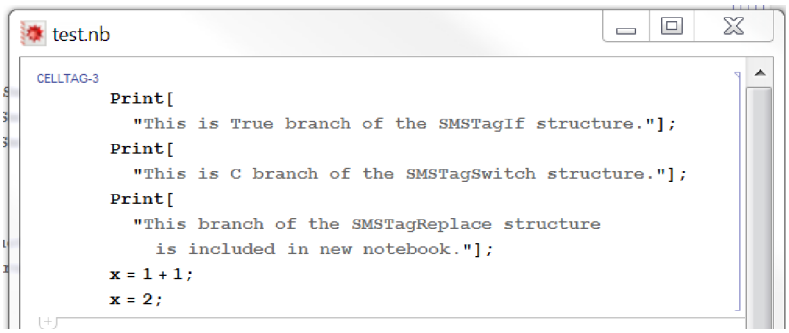
```
[0-0] Include Tag : CELLTAG-3 (1 cells found, 1 evaluated)

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".

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

In[136]:=

```
<< AceGen`;
```

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$$]];
r ⊢ SMSReal[r$$];
t ⊢ SMSReal[t$$];
```

- Here function *func* is symbolically evaluated for the given set of parameters.

In[145]:=
```
e ⊨ func[r, t];
```

In[146]:=
```
SMSExport[e, e$$];
SMSWrite[];
```

File: tmptest.c **Size:** 790 **Time:** 0

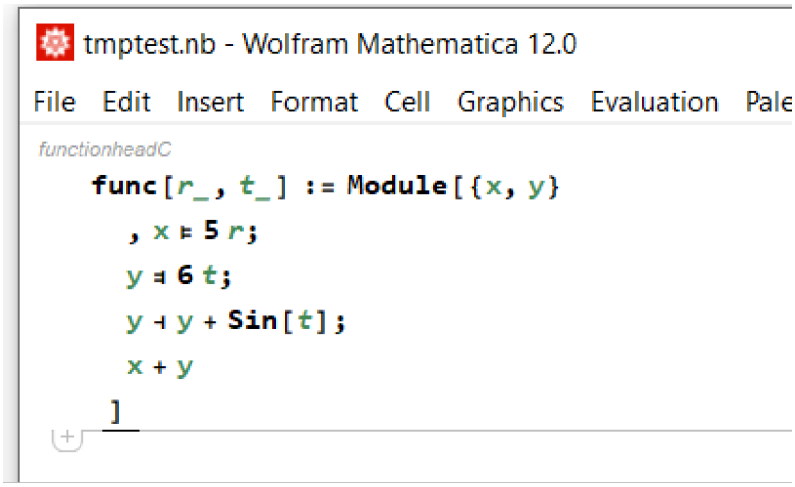
Method	Include
No.Formulae	3
No.Leafs	31

In[148]:=
```
FilePrint["tmptest.c", -8]
```

```
/******************* S U B R O U T I N E *********************/
void Include(double v[117],double (*r),double (*t),double (*e))
{
v[4]=6e0*(*t);
v[4]=v[4]+sin((*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[];
```



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.

KORELC, Jože. Automatic generation of finite-element code by simultaneous optimization of expressions. Theor. comput. sci.. , 1997, 187:231-248.

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 it`s 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*] ≡ Map[SMSReal[#,code]&,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]
{*vmin_Real*,*vmax_Real*}	real type random number form interval [vmin,vmax]
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*]
andom 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*] ≡ 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
```

```
{0.5126296357476784249473038651688948998 75,
 47.741233930866187312379418124941701519 2, 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:

$$\Phi=\begin{pmatrix} a\,x\,y+x^3=0 \\ a-x\,y^2=0 \end{pmatrix}$$

where x and y are unknowns and a is parameter.

Solution

- Here the appropriate *MathLink* module is created.

In[37]:=

```
<< 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, ϵ} ⊢ SMSReal[{x$$, y$$, a$$, tol$$}];
nmax ⊢ SMSInteger[n$$];
{x, y} ⫤ {x0, y0};
SMSDo[
  Φ ⊨ {a x y + x^3, a - x y^2};
  Kt ⊨ SMSD[Φ, {x, y}];
  {Δx, Δy} ⊨ SMSLinearSolve[Kt, -Φ];
  {x, y} ⫤ {x, y} + {Δx, Δy};
  SMSIf[SMSSqrt[{Δx, Δy}.{Δx, Δy}] < ϵ
   , SMSExport[{x, y}, {x$$, y$$}];
   SMSBreak[];
  ];
  SMSIf[i == nmax
  , SMSPrintMessage["no convergence"];
   SMSReturn[];
  ];
  , {i, 1, nmax, 1, {x, y}}
 ];
SMSWrite[];
```

File:	test.c	Size: 2343
Methods	No.Formulae	No.Leafs
test	16	149

- 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`)

In[45]:=

```
SMSInstallMathLink[]
```

Out[•]=

```
{SMSSetLinkOption[test, {i_Integer, j_Integer}], SMSLinkNoEvaluations[test],
 test[x_?NumberQ, y_?NumberQ, a_?NumberQ, tol_?NumberQ, n_?NumberQ]}
```

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.

In[46]:=

```
test[1.9,-1.2,3.,0.0001,10]
```

Out[ ]=

```
{1.93318, -1.24573}
```

In[47]:=

```
x =.; y =.; a = 3.;
Solve[{a x y + x^3 == 0, a - x y^2 == 0}, {x, y}]
```

Out[ ]=

```
{{y → -1.24573, x → 1.93318}, {y → -0.384952 + 1.18476 i, x → -1.56398 + 1.1363 i},
 {y → -0.384952 - 1.18476 i, x → -1.56398 - 1.1363 i},
 {y → 1.00782 + 0.732222 i, x → 0.597386 - 1.83857 i},
 {y → 1.00782 - 0.732222 i, x → 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

$$\Pi = \int_\Omega \left(\frac{1}{2} k\, \Delta\phi.\Delta\phi - \phi\, Q\right) d\Omega$$

where ϕ indicates temperature, k is the conductivity and Q is the heat generation per unit volume and Ω 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 ϕ=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 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 (ϕ(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
- number of evaluations of function, gradient and hessian.

Method	mesh	ϕ	derivatio-n *time (s)*	solution time (s)	No. of variables	memo-ry (MB)	No. of calls
A.MMA.Gradient	5×5×5	55.9	8.6	59.5	80	136	964
A.MMA.Newton	5×5×5	55.9	8.6	177.6	80	1050	4
A.AceGen. Gradient	5×5×5	55.9	6.8	3.3	80	4	962
A.AceGen. Newton	5×5×5	55.9	13.0	0.8	80	4	4
B.MMA.Gradient	11×11×11	57.5	0.3	11.7	810	10	1685
B.MMA.Newton	11×11×11	57.5	0.3	1.1	810	16	4
B.AceGen. Gradient	11×11×11	57.5	1.4	6.30	810	4	1598
B.AceGen. Newton	11×11×11	57.5	4.0	0.8	810	4	4
C.AceFEM	10×10×10	56.5	5.0	2.0	810	6	2
C.AceFEM	20×20×20	55.9	5.0	3.2	7220	32	2
C.AceFEM	30×30×30	55.9	5.0	16.8	25 230	139	2

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.

REMMARK: 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 ϕ 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.

In[1]:=

```
<< AceGen`;
Clear[x, y, z, α];
kcond = 0.58; Q = 500;
order = 5;
nterm = (order - 1) (order - 1) (order)
```

Out[•]=

```
80
```

- Here the fifth order Lagrange polynomials are constructed in three dimensions.

In[6]:=

```
toc = Table[{x, 0}, {x, -0.5, 0.5, 1/order}];
xp = MapIndexed[InterpolatingPolynomial[ReplacePart[toc, 1, {#, 2}], x] &, Range[2, order]];
yp = MapIndexed[InterpolatingPolynomial[ReplacePart[toc, 1, {#, 2}], y] &, Range[2, order]];
toc = Table[{x, 0}, {x, 0., 1., 1/order}];
zp = MapIndexed[InterpolatingPolynomial[ReplacePart[toc, 1, {#, 2}], z] &, Range[2, order + 1]];
ϕi = Array[α, nterm];
poly = Flatten[Outer[Times, xp, yp, zp]] // Chop;
ϕ = poly.ϕi;
```

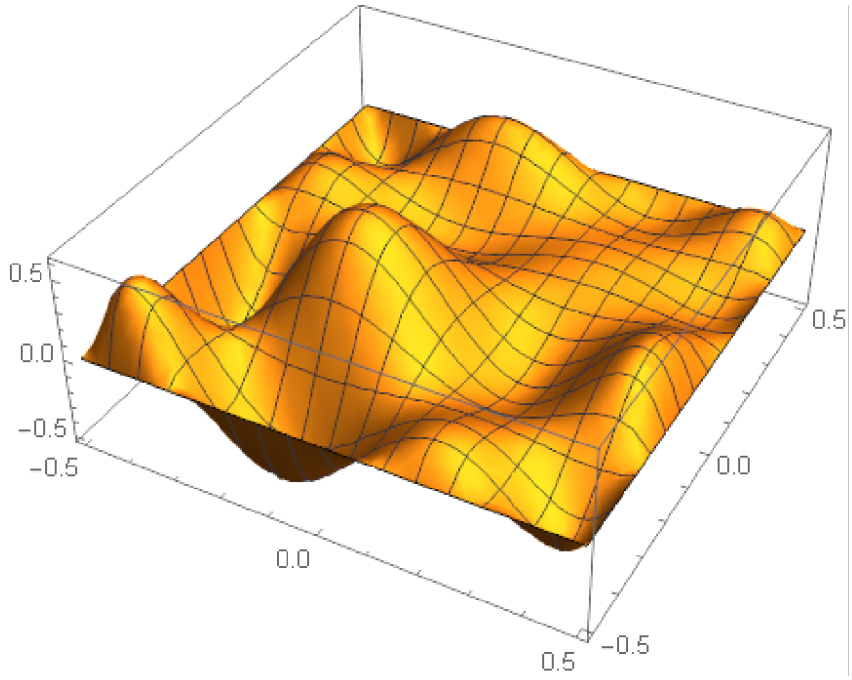
In[13]:=

```
poly[[28]]
Plot3D[poly[[28]] /. z → 0.5, {x, -0.5, 0.5}, {y, -0.5, 0.5}, PlotRange → All]
```

Out[ ]=

```
260.417 (-0.5 + x) (0.5 + x)
 (-4.16667 + (0.1 + x) (10.4167 + (-0.3 + x) (52.0833 - 260.417 (0.3 + x)))) (-0.5 + y)
 (0.5 + y) (-4.16667 + (0.1 + y) (10.4167 + (-0.3 + y) (52.0833 - 260.417 (0.3 + y))))
 (-1. + z) (-0.8 + z) (-0.4 + z) (-0.2 + z) z
```

Out[ ]=



- Here the Gauss points and weights are calculated for ngp×ngp×ngp Gauss numerical integration of the free energy over the domain [-.0.5m,0.5m]×[-0.5m,0.5m]×[0,1m].

In[15]:=

```
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[{#3, #2, #1} &, {ngp, ngp, ngp}], 2];
```

Direct use of Mathematica

- The subsection Definitions has to be executed before the current subsection.

In[21]:=

```
start = SessionTime[];
Δϕ = {D[ϕ, x], D[ϕ, y], D[ϕ, z]};
Π = 1 / 2 kcond Δϕ.Δϕ - ϕ Q;
Πi = Total[Map[(#[[4]] Π /. {x → #[[1]], y → #[[2]], z → #[[3]]}) &, gp]];
derivation = SessionTime[] - start
```

Out[•]=

```
5.7288032
```

- G. Gradient based optimization

In[26]:=

```
start = SessionTime[];
ii = 0;
sol = FindMinimum[Πi, Array[{α[#] , 0.} &, nterm],
  Method → "Gradient", EvaluationMonitor :→ (ii++;)];
{ii, ϕ /. sol[[2]] /. {x → 0, y → 0, z → 0.5}}
SessionTime[] - start
```

Out[•]=

```
{1016, 55.8724}
```

Out[•]=

```
9.7794951
```

- N. Newton method based optimization

In[30]:=

```
start = SessionTime[];
ii = 0;
sol = FindMinimum[Πi, Array[{α[#] , 0.} &, nterm],
  Method → "Newton", EvaluationMonitor :→ (ii++;)];
{ii, ϕ /. sol[[2]] /. {x → 0, y → 0, z → 0.5}}
SessionTime[] - start
```

Out[•]=

```
{4, 55.8724}
```

Out[•]=

```
163.5393726
```

AceGen code generation

- The subsection Definitions has to be executed before the current subsection.

In[72]:=

```
start = SessionTime[];
SMSInitialize["Thermal", "Environment" → "MathLink"]

Πf[i_] := (
  ai ⊢ SMSReal[Array[a$$, nterm]];
  ag ⊨ SMSArray[ai];
  {xa, ya, za, wa} ⊨ Map[SMSArray, Transpose[gp]];
  {xi, yi, zi} ⊢ SMSFreeze[{SMSPart[xa, i], SMSPart[ya, i], SMSPart[za, i]}];
  {xpr, ypr, zpr} ⊨ {xp /. x → xi, yp /. y → yi, zp /. z → zi};
  poly ⊨ SMSArray[Flatten[Outer[Times, xpr, ypr, zpr]]];
  ϕt ⊨ SMSDot[poly, ag];
  Δϕ ⊨ SMSD[ϕt, {xi, yi, zi}];
  wi ⊨ SMSPart[wa, i];
  wi (1 / 2 kcond Δϕ.Δϕ - ϕt Q)
 )
```

Out[ ]=

```
True
```

In[75]:=

```
SMSModule["FThermal", Real[a$$[nterm], f$$], "Input" → a$$, "Output" → f$$];
SMSExport[0, f$$];
SMSDo[i, 1, gp // Length];
 Π ⊨ Πf[i];
 SMSExport[ Π, f$$, "AddIn" → True];
SMSEndDo[];
```

In[81]:=

```
SMSModule["GThermal", Real[a$$[nterm], g$$[nterm]], "Input" → a$$, "Output" → g$$];
SMSExport[Table[0, {nterm}], g$$];
SMSDo[i, 1, gp // Length];
 Π ⊨ Πf[i];
 SMSDo[j, 1, nterm];
  δΠ ⊨ SMSD[Π, ag, j, "Mode" → "Forward"];
  SMSExport[δΠ, g$$[j], "AddIn" → True];
 SMSEndDo[];
SMSEndDo[];
```

In[90]:=

```
derivation = SessionTime[] - start
```

Out[ ]=

```
5.2324732
```

In[91]:=

```
SMSModule["HThermal", Real[a$$[nterm], h$$[nterm, nterm]], "Input" → a$$, "Output" → h$$];
SMSDo[i, 1, nterm];
 SMSDo[j, 1, nterm];
  SMSExport[0, h$$[i, j]];
 SMSEndDo[];
SMSEndDo[];
SMSDo[i, 1, gp // Length];
 Π ⊨ Πf[i];
 SMSDo[j, 1, nterm];
  δΠ ⊨ SMSD[Π, ag, j, "Mode" → "Forward"];
  SMSDo[k, 1, nterm];
   hij ⊨ SMSD[δΠ, ag, k, "Mode" → "Forward"];
   SMSExport[hij, h$$[j, k], "AddIn" → True];
  SMSEndDo[];
 SMSEndDo[];
SMSEndDo[];
```

In[107]:=

```
SMSWrite[];
```

```
[16]  Consistency check - expressions
```

File: Thermal.c Size: 125 709 Time: 17

Method	FThermal	GThermal	HThermal
No.Formulae	150	162	160
No.Leafs	4816	5377	3940

In[108]:=

```
SMSInstallMathLink[]
derivation = SessionTime[] - start
```

Out[•]=

```
{SMSSetLinkOption[Thermal, {i_Integer, j_Integer}],
 SMSLinkNoEvaluations[Thermal], FThermal[a_?
   (ArrayQ[#1, 1, Head[#1] == Real || Head[#1] == Integer &] && Dimensions[#1] === {80} &)],
 GThermal[a_? (ArrayQ[#1, 1, Head[#1] == Real || Head[#1] == Integer &] &&
      Dimensions[#1] === {80} &)], HThermal[a_?
   (ArrayQ[#1, 1, Head[#1] == Real || Head[#1] == Integer &] && Dimensions[#1] === {80} &)]}
```

Out[•]=

```
27.8514925
```

AceGen Solution

- G. Gradient based optimization

In[110]:=

```
start = SessionTime[]; ii = 0;
sol = FindMinimum[FThermal[ϕi], {ϕi, Table[0, {nterm}]},
   Method → "Gradient", Gradient → GThermal[ϕi], EvaluationMonitor :→ (ii++;)];
{ii, ϕ /. MapThread[Rule, List @@ sol[[2, 1]]] /. {x → 0, y → 0, z → 0.5}, SessionTime[] - start}
```

Out[•]=

```
{915, 55.8724, 0.4513012}
```

- N. Newton method based optimization

In[113]:=

```
start = SessionTime[]; ii = 0;
sol = FindMinimum[FThermal[ϕi], {ϕi, Table[0, {nterm}]},
   Method → {"Newton", Hessian → HThermal[ϕi]},
   Gradient → GThermal[ϕi], EvaluationMonitor :→ (ii++;)];
{ii, ϕ /. MapThread[Rule, List @@ sol[[2, 1]]] /. {x → 0, y → 0, z → 0.5}, SessionTime[] - start}
```

Out[ ]=

```
{4, 55.8724, 0.0580390}
```

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.

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;
```

In[125]:=

```
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[i_] :→ nterm + i;
ϕi = Array[α, nterm];
nterm
```

Out[ ]=

```
810
```

Direct use of Mathematica

- The subsection Definitions have to be executed before the current subsection.

In[129]:=

```
start = SessionTime[];
Πi = 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]])/(2 dlxt), (aijk[[5]] - aijk[[4]])/(2 dlyt), (aijk[[7]] - aijk[[6]])/(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[•]=

```
0.1040714
```

- G. Gradient based optimization

In[132]:=

```
start = SessionTime[]; ii = 0;
sol =
  FindMinimum[Πi, 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[•]=

```
{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[[(nx + 3) / 2, (ny + 3) / 2, (nz + 3) / 2]]] /. sol[[2]], SessionTime[] - start}
```

Out[•]=

```
{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$$[(#[[1]] - 1) * (nyp + 2) (nzp + 2) + (#[[2]] - 1) * (nzp + 2) + #[[3]]] &,
      {{i, j, k}, {i - 1, j, k}, {i + 1, j, k}, {i, j - 1, k},
       {i, j + 1, k}, {i, j, k - 1}, {i, j, k + 1}}]];
  aijk ⊢ SMSReal[Map[a$$[#] &, 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;
  grad = {(aijk[[3]] - aijk[[2]])/(2 dlxt), (aijk[[5]] - aijk[[4]])/(2 dlyt), (aijk[[7]] - aijk[[6]])/(2 dlzt)};
  vol (1 / 2 kc grad.grad - Qt aijk[[1]])
 )
```

Out[ ]=

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 ⊨ Πf[i, j, k];
   SMSExport[Π, f$$, "AddIn" → True];
  SMSEndDo[];
 SMSEndDo[];
SMSEndDo[];
```

In[152]:=

```
SMSModule["GThermal", Integer[ndof$$, nt$$[3], index$$["*"]],
  Real[a$$["*"], mc$$["*"], g$$[ndof$$]],
  "Input" → {ndof$$, nt$$, index$$, a$$, mc$$}, "Output" → g$$];
ndof ⊢ SMSInteger[ndof$$];
{nxp, nyp, nzp} ⊢ SMSInteger[Array[nt$$, 3]];
SMSDo[i, 1, ndof];
 SMSExport[0, g$$[i]];
SMSEndDo[];

SMSDo[i, 2, nxp + 1];
 SMSDo[j, 2, nyp + 1];
  SMSDo[k, 2, nzp + 1];
   Π ⊨ Πf[i, j, k];
   SMSDo[i1, 1, indexp // Length];
    dof ⊨ SMSPart[indexp, i1];
    SMSIf[dof ≤ ndof];
     gi ⊨ SMSD[Π, aijk, i1];
     SMSExport[gi, g$$[dof], "AddIn" → True];
    SMSEndIf[];
   SMSEndDo[];
  SMSEndDo[];
 SMSEndDo[];
SMSEndDo[];
```

In[172]:=

```
derivation = SessionTime[] - start
```

Out[ ]=

```
0.9526325
```

In[173]:=

```
SMSModule["HThermal", Integer[ndof$$, nt$$[3], index$$["*"]],
  Real[a$$["*"], mc$$["*"], h$$[ndof$$, ndof$$]],
  "Input" → {ndof$$, nt$$, index$$, a$$, mc$$}, "Output" → h$$];
ndof ⊢ SMSInteger[ndof$$];
{nxp, nyp, nzp} ⊢ SMSInteger[Array[nt$$, 3]];
SMSDo[i, 1, ndof];
 SMSDo[j, 1, ndof];
  SMSExport[0, h$$[i, j]];
 SMSEndDo[];
SMSEndDo[];

SMSDo[i, 2, nxp + 1];
 SMSDo[j, 2, nyp + 1];
  SMSDo[k, 2, nzp + 1];
   Π ⊨ Πf[i, j, k];
   SMSDo[i1, 1, indexp // Length];
    dofi ⊨ SMSPart[indexp, i1];
    SMSIf[dofi ≤ ndof];
     gi ⊨ SMSD[Π, aijk, i1];
     SMSDo[j1, 1, indexp // Length];
      dofj ⊨ SMSPart[indexp, j1];
      SMSIf[dofj ≤ ndof];
       hij ⊨ SMSD[gi, aijk, j1];
       SMSExport[hij, h$$[dofi, dofj], "AddIn" → True];
      SMSEndIf[];
     SMSEndDo[];
    SMSEndIf[];
   SMSEndDo[];
  SMSEndDo[];
 SMSEndDo[];
SMSEndDo[];
```

In[201]:=

```
SMSWrite[];
```

[2] Consistency check - expressions

File: Thermal.c **Size:** 10921 **Time:** 2

Method	FThermal	GThermal	HThermal
No.Formulae	21	33	31
No.Leafs	398	488	495

In[202]:=

```
SMSInstallMathLink[]
derivation = SessionTime[] - start
```

Out[•]=

```
{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 &] &)],
 HThermal[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[•]=

```
7.2057857
```

AceGen Solution

- G. Gradient based optimization

In[204]:=

```
start = SessionTime[]; ii = 0;
indexb = Flatten[index];
sol = FindMinimum[
  FThermal[nterm, {nx, ny, nz}, indexb, Join[ϕi, bound], {dlx, dly, dlz, kcond, Q}]
  , {ϕi, Table[0, {nterm}]},
  Method → "Gradient",
  Gradient → GThermal[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}
```

FindMinimum: Failed to converge to the requested accuracy or precision within 100 iterations.

Out[•]=

```
{1596, 57.5034, 10.2127803}
```

- N. Newton method based optimization

In[207]:=

```
start = SessionTime[]; ii = 0;
indexb = Flatten[index /. b[i_] :> nterm + i];
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 → GThermal[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[ ]=

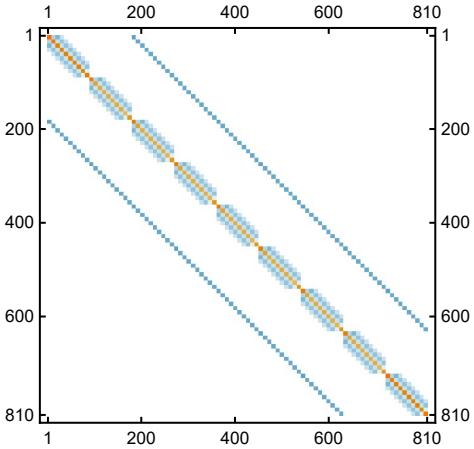
```
{4, 57.5034, 0.2761834}
```

- The tangent matrix is in the case of finite difference approximation extremely sparse.

In[210]:=

```
MatrixPlot[HThermal[nterm, {nx, ny, nz}, indexb, Join[0 ϕi, bound], {dlx, dly, dlz, kcond, Q}]]
```

Out[ ]=



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[211]:=

```
<< AceFEM`;
start = SessionTime[];
SMTInputData[];
k = 0.58; Q = 500;
nn = 30;
SMTAddDomain["cube", "ExamplesHeatConduction", {"k0 *" → k, "Q *" → Q}];
SMTAddEssentialBoundary[
  {"X" == -0.5 || "X" == 0.5 || "Y" == -0.5 || "Y" == 0.5 || "Z" == 0. &, 1 → 0}];
SMTMesh["cube", "H1", {nn, nn, nn}, {
   {{{-0.5, -0.5, 0}, {0.5, -0.5, 0}}, {{-0.5, 0.5, 0}, {0.5, 0.5, 0}}},
   {{{-0.5, -0.5, 1}, {0.5, -0.5, 1}}, {{-0.5, 0.5, 1}, {0.5, 0.5, 1}}}
  }];
SMTAnalysis[];
```

In[220]:=

```
SMTNextStep[0, 1];
SMTNewtonIteration[];
```

In[222]:=

```
SMTPostData["T*", {0, 0, 0.5}]
SessionTime[] - start
```

Out[ ]=

```
55.8765
```

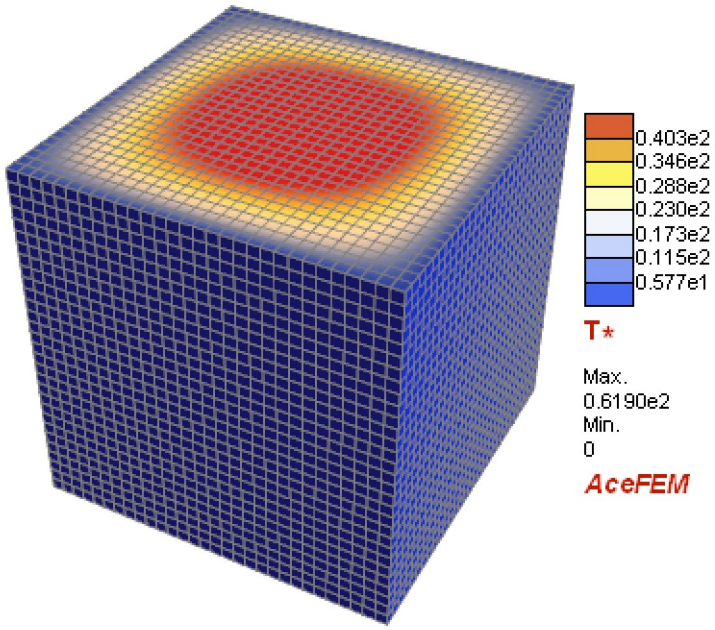
Out[ ]=

```
5.1254023
```

In[224]:=

```
SMTShowMesh["Mesh" → True, "FillElements" → True, "Field" → "T*"]
```

Out[ ]=



CHAPTER 6

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, Jacobean, Hessian and coding of the program in a 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:

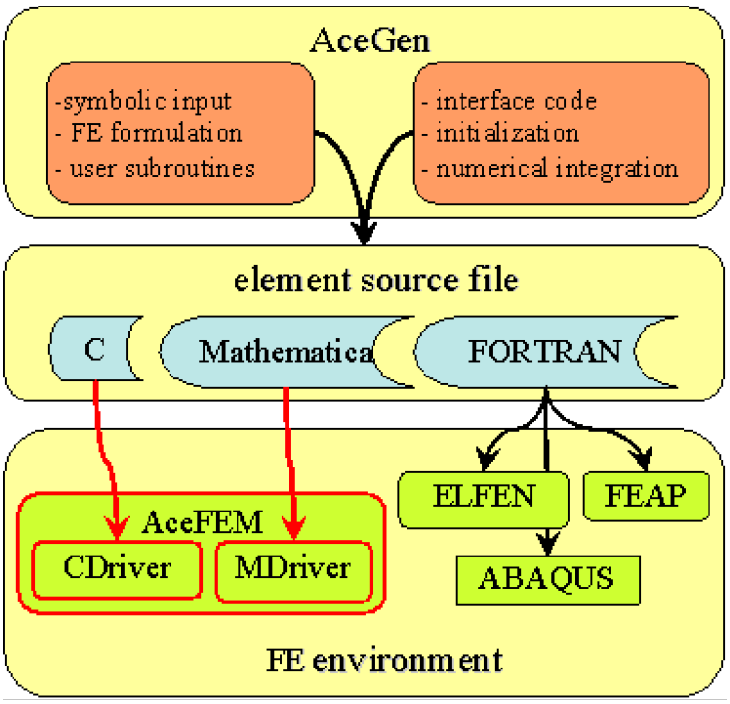
⇒ automatic translation to other FE packages,

⇒ 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.

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 `Standard User Subroutines`). 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 `Standard User Subroutines`). 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*". See also `Installation instructions` for instructions how to compile and link the generated codes.

Since the complexity of the problem description mostly appears in a symbolic input, we can keep the number of data structures (see `Data Structures`) 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 Variables`, `Real Type Environment Variables`),

⇒ 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).

KORELC, Jože, Multi-language and Multi-environment Generation of Nonlinear Finite Element Codes, Engineering with Computers, 2002, 18(4):312-327

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:

$$\frac{\partial}{\partial x}\left(k\frac{\partial\phi}{\partial x}\right)+\frac{\partial}{\partial y}\left(k\frac{\partial\phi}{\partial y}\right)+\frac{\partial}{\partial z}\left(k\frac{\partial\phi}{\partial z}\right)+Q=0 \quad \text{on domain } \Omega,$$

$$\phi-\overline{\phi}=0 \qquad \text{essential boundary condition on } \Gamma_{\phi},$$

$$k\frac{\partial\phi}{\partial n}-\overline{q}=0 \qquad \text{natural boundary condition on } \Gamma_{q},$$

where ϕ indicates temperature, k is conductivity, Q heat generation per unit volume, and $\overline{\phi}$ and $\overline{q}$ 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

$$\int_{\Omega}\left[\nabla^{T}\delta\phi\;k\;\nabla\phi-\delta\phi\,Q\right]d\Omega-\int_{\Gamma_q}\delta\phi\,\overline{q}\,d\Gamma=0.$$

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,
  "SMSSymmetricTangent" → 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 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`). 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.

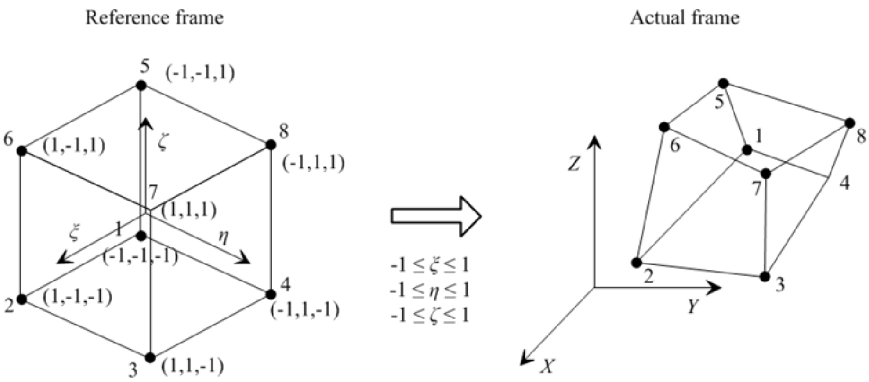
In[178]:=
```
{k0, k1, k2, Q} ⊨ SMSIO["Domain data"];
```

- 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[179]:=
```
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 N_i, *i=1,2,…,8* and interpolation of the physical coordinates within the element. J_m is Jacobian matrix of the isoparametric mapping from actual coordinate system X, Y, Z to reference coordinates ξ, η, ζ.



In[181]:=

```
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 + ξ Ξn[[i, 1]]) (1 + η Ξn[[i, 2]]) (1 + ζ Ξn[[i, 3]]), {i, 1, 8}];
X ⊢ SMSFreeze[NI.XIO];
Jg ⊨ SMSD[X, Ξ];
Jgd ⊨ Det[Jg];
```

- The trial function for the temperature distribution within the element is given as linear combination of the shape functions and the nodal temperatures $\phi = N_n.\phi_n$. The ϕ_n are unknown parameters of the variational problem.

In[185]:=

```
ϕ ⊨ NI.ϕI;
```

Step 5: Definition of the governing equations

- 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.

In[186]:=

```
Dϕ ⊨ SMSD[ϕ, X, "Dependency" → {Ξ, X, SMSInverse[Jg]}];
δϕ ⊨ SMSD[ϕ, ϕI];
Dδϕ ⊨ SMSD[δϕ, X, "Dependency" → {Ξ, X, SMSInverse[Jg]}];
```

- 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 `Real Type Environment Variables`).

In[189]:=

```
k ⊨ k0 + k1 ϕ + k2 ϕ^2;
λ ⊨ SMSIO["Multiplier"];
```

In[191]:=

```
wgp ⊨ SMSIO["Integration weight"[Ig]];
Rg ⊨ Jgd wgp (k Dδϕ.Dϕ - δϕ λ Q);
```

- Element contribution to global residual vector R_g is exported into the *p$$* output parameter of the "*Tangent and residual*" subroutine (see `Standard User Subroutines`).

In[193]:=

```
SMSIO[Rg, "Add to", "Residual"];
```

Step 6: Definition of the Jacobian matrix

- This evaluates the explicit form of the Jacobian (tangent) matrix and exports result into the *s$$* output parameter of the user subroutine. Another possibility would be to generate a characteristic formula for the arbitrary element of the residual and the tangent matrix. This would substantially reduce the code size.

In[194]:=
```
Kg ⊨ SMSD[Rg, ϕI];
SMSIO[Kg, "Add to", "Tangent"];
```

- This is the end of the integration loop.

In[196]:=
```
SMSEndDo[];
```

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.

In[197]:=
```
SMSStandardModule["Postprocessing"];
```

- Here the nodal point post-processing quantity "Temperature" is introduced and exported to array of the nodal point quantities *npost$$*.

In[198]:=
```
{XIO, ϕIO} ⊨ SMSIO["All coordinates and DOFs"];
ϕI = ϕIO[[All, 1]];
SMSIO[{"Temperature" → ϕI}, "Export to", "Nodal point post"];
```

- Here the integration point post-processing quantity "Conductivity" is introduced and exported to array of the integration point quantities *gpost$$*.

In[202]:=
```
{k0, k1, k2, Q} ⊨ SMSIO["Domain data"];
SMSDo[
  Ξ = {ξ, η, ζ} ⊨ SMSIO["Integration point"[Ig]];
  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 + ξ Ξn[[i, 1]]) (1 + η Ξn[[i, 2]]) (1 + ζ Ξn[[i, 3]]), {i, 1, 8}];
  ϕ ⊨ NI.ϕI;
  k ⊨ k0 + k1 ϕ + k2 ϕ^2;
  SMSIO[{"Conductivity" → k}, "Export to", "Integration point"[Ig]];
  , {Ig, 1, SMSIO["No. integration points"]}
 ];
```

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.

In[204]:=
```
SMSWrite[];
```

File: ExamplesHeatConduction.c **Size:** 17 001 **Time:** 6

Method	SKR	SPP
No.Formulae	243	27
No.Leafs	4862	366

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

In[205]:=
```
<< AceGen`;
SMSInitialize["tmpExamplesHeatConduction", "Environment" → "AceFEM"];
SMSTemplate["SMSTopology" → "H1", "SMSDOFGlobal" → 1,
  "SMSSymmetricTangent" → 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

In[208]:=
```
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 nd$$[i,"X",j] and nd$$[i,"at",1] (see Node Data).

In[209]:=
```
XI ⊢ Table[SMSReal[nd$$[i, "X", j]], {i, SMSNoNodes}, {j, SMSNoDimensions}];
ϕI ⊢ Table[SMSReal[nd$$[i, "at", 1]], {i, 8}];
```

- 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).

In[210]:=
```
{k0, k1, k2, Q} ⊢ SMSReal[Table[es$$["Data", i], {i, 4}]];
```

- 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[211]:=
```
SMSDo[Ig, 1, SMSInteger[es$$["id", "NoIntPoints"]]];
Ξ = {ξ, η, ζ} ⊢ Table[SMSReal[es$$["IntPoints", i, Ig]], {i, 3}];
wgp ⊢ SMSReal[es$$["IntPoints", 4, Ig]];
```

Step 4: Definition of the trial functions

In[214]:=
```
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 + ξ Ξn[[i, 1]]) (1 + η Ξn[[i, 2]]) (1 + ζ Ξn[[i, 3]]), {i, 1, 8}];
X ⊢ SMSFreeze[NI.XI];
Jg ⊨ SMSD[X, Ξ]; Jgd ⊨ Det[Jg];
ϕ ⊨ NI.ϕI;
```

Step 5: Definition of the governing equations

In[219]:=
```
Dϕ ⊨ SMSD[ϕ, X, "Dependency" → {Ξ, X, SMSInverse[Jg]}];
δϕ ⊨ SMSD[ϕ, ϕI];
Dδϕ ⊨ SMSD[δϕ, X, "Dependency" → {Ξ, X, SMSInverse[Jg]}];
k ⊨ k0 + k1 ϕ + k2 ϕ^2;
```

- The strength of the heat source is multiplied by the global variable *rdata$$["Multiplier"]* (see Real Type Environment Variables).

In[223]:=
```
λ ⊢ SMSReal[rdata$$["Multiplier"]];
```

In[224]:=
```
Rg ⊨ Jgd wgp (k Dδϕ.Dϕ - δϕ λ Q);
SMSExport[Rg, p$$, "AddIn" → True];
```

Step 6: Definition of the Jacobian matrix

In[226]:=
```
Kg ⊨ SMSD[Rg, ϕI];
SMSExport[Kg, s$$, "AddIn" → True];
```

In[228]:=
```
SMSEndDo[];
```

Step 7: Post-processing subroutine

In[229]:=
```
SMSStandardModule["Postprocessing"];
ϕI ⊢ Table[SMSReal[nd$$[i, "at", 1]], {i, 8}];
SMSNPostNames = {"Temperature"};
SMSExport[ϕI, Table[npost$$[i, 1], {i, 8}]];
{k0, k1, k2, Q} ⊢ SMSReal[Table[es$$["Data", i], {i, Length[SMSDomainDataNames]}]];
SMSDo[
  Ξ = {ξ, η, ζ} ⊢ Table[SMSReal[es$$["IntPoints", i, Ig]], {i, 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 + ξ Ξn[[i, 1]]) (1 + η Ξn[[i, 2]]) (1 + ζ Ξn[[i, 3]]), {i, 1, 8}];
  ϕ ⊨ NI.ϕI;
  k ⊨ k0 + k1 ϕ + k2 ϕ^2;
  SMSGPostNames = {"Conductivity"};
  SMSExport[k, gpost$$[Ig, 1]];
  , {Ig, 1, SMSInteger[es$$["id", "NoIntPoints"]]}
 ];
```

Step 8: Code Generation

In[235]:=
```
SMSWrite[];
```

File: tmpExamplesHeatConduction.c Size: 16977 Time: 6

Method	SKR	SPP
No.Formulae	242	26
No.Leafs	4856	360

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

Abbreviation	default	description
SMSTopology		element topology (see Element Topology)
SMSNoNodes	Automatic	number of nodes (only if different than the one defined by element topology, see Node Identification, accessed by SMSIO["Nodal coordinates"[*node,coordinate*]])
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

Abbreviation	default	description
SMSDOFGlobal	Table[SMSNoDimensions, SMSNoNodes]	number of d.o.f per node for all nodes, accessed by SMSIO["Nodal DOFs"[*node,dof*]]
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)
SMSResidualSign	Automatic	1 ⇒ equations are formed in the form $\boldsymbol{K\ a} + \boldsymbol{\Psi} = \boldsymbol{0}$ -1 ⇒ equations are formed in the form $\boldsymbol{K\ a} = \boldsymbol{\Psi}$ (used to ensure compatibility between the numerical environments)
SMSDefaultIntegrationCod-e	Automatic	default numerical integration code (see Numerical Integration, points and weights are accessed by SMSIO["Integration point"[*Ig*]], SMSIO["Integration weight"[*Ig*]], SMSIO["No. integration points"])
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
SMSVariableNumberOfNo-des	False	a mechanism is initialized that makes easy formulation of elements with variable number of nodes and DOFs (see Nodal Input data, Virtual Element Method)

Data Management

Abbreviation	default	description
SMSDomainDataNames	{}	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$$["Data",j] data field, accessed by SMSIO["Domain data"[*name*]])
SMSDefaultData	Table[0., SMSDomainDataNames// Length]	default values for input data values
SMSDataCheck	True	logical expression that checks the correctness of the user supplied constants stored in es$$["Data",*i*] . It should return True if the data is correct.
SMSNoTimeStorage	0	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$$["ht",j] and ed$$["hp",j] data fields, , accessed by SMSIO["Element time dependent data"[*index*]], SMSIO["Element time dependent data n"[*index*]])
SMSNoElementData	0	total number of arbitrary real values per element (defines the dimension of the ed$$["Data",j] data field, accessed by SMSIO["Element data"[*index*]])
SMSNoNodeStorage	Table[0, SMSNoNodes]	total number of history dependent real type values per node that have to be stored in the memory for transient type of problems (can be different for each node) (defines the dimension of the nd$$[i,"ht",j] and nd$$[i,"hp",j] data fields, accessed by SMSIO["Nodal time dependent data"[*node,index*]], SMSIO["Nodal time dependent data n"[*node,index*]])
SMSNoNodeData	0	total number of arbitrary real values per node (can be different for each node) (defines the dimension of the nd$$[i,"Data",j] data field, accessed by SMSIO["Nodal data"[*node,index*]])
SMSIDataNames	{}	list of the keywords of additional integer type environment data variables (global), see Integer Type Environment Variables, accessed by SMSIO[*idataCode*]
SMSRDataNames	{}	list of the keywords of additional real type environment data variables (global), see Real Type Environment Variables, accessed by SMSIO[*rdataCode*]
SMSNoAdditionalData	0	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$$["AdditionalData",*i*] data field)
SMSCharSwitch	{}	list of character type user defined constants (local) (e.g. "SMSCharSwitch"->{"A","B"}, also automatically generated by SMSIO["Char type switch","*switchValue*"->*switchDescription*])
SMSIntSwitch	{}	list of integer type user defined constants (local) (e.g. "SMSIntSwitch"->{1,20}, also automatically generated by SMSIO["Integer type switch",, "*keyword*"->{*switchValue*, *switchDescription*}])
SMSDoubleSwitch	{}	list of double type user defined constants (local) (e.g. "SMSDoubleSwitch"->{1.41,100.}, also automatically generated by SMSIO["Real type switch", "*keyword*"->{*switchValue*, *switchDescription*}])

Graphics and Postprocessing

Abbreviation	default	description
SMSGPostNames	{}	description of the post-processing quantities defined per material point (see Subroutine: Postprocessing, automatically generated by SMSIO[{*key1->val1*, *key2->val2, ...*}, "Export to", "Integration point post"[Ig]])
SMSNPostNames	{}	description of the post-processing quantities defined per node (see Subroutine: Postprocessing, utomatically generated by SMSIO[{*key1->{val11,val12,…,val_NoNodes}*, *key2->{val21,val22,…}, ...*},"Export to", "Nodal point post"])
SMSSegments	Automatic	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) SMSSegments={{1,2,3,4}} SMSSegments={} ... no post-processing (see example Cubic triangle, Additional nodes)
SMSSegmentsTriangulation	Automatic	for all segments define a rule that splits the segments specified by SMSSegments into triangular or quadrilateral sub-segments (the data is used to color the interior of the segments and post-processing of field variables) SMSSegments={{{1,2,3},{1,3,4}}} SMSSegments={{}} ... no field post-processing (see axample Cubic triangle, Additional nodes)
SMSReferenceNodes	Automatic	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)
SMSPostNodeWeights	Automatic	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.
SMSAdditionalGraphics	{Null,Null,Null}	A set of three pure functions {addGraphicsPrimitives, trueSegments, trueTriangulation} (see Function) that are called for each element as desribed below or Null if additional graphics is not defined for specific element type. (see also SMTUpdatePostData) 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.
  Global variable AceFEM`SMTSession[[60]] contains some of the SMTShowMesh options:

```
AceFEM`SMTSession[[60]]=
        {1 - "DeformedMesh" True/False
         ,2 - "Scale"
         ,3 - "Mesh"
```

```
,4 - "LineFieldScale"
}
```

In the case that "DeformedMesh"===True the provided node coordinates are positions of the nodes of the **deformed mesh.**
Examples:

- "SMSAdditionalGraphics"->{Function[{element,marks,bc,X},{Line[{X[[1]],X[[2]]}]}],Null,Null}
  would produce a line connecting first and second element node)
- "SMSAdditionalGraphics"->{Function[{element,marks,bc,X},{Red,Dashed,Line[{AceFEM`SMTNodeData[element[[3,1]],"X"],X[[1]]}]}],Null,Null}
  would produce Red, Dashed, Line connecting the first element node in undeformed mesh with corresponding node in deformed mesh.

- trueSegments
  trueSegments[element index, domain index,list of node indices]
  must return all segments on the surface of the specific element as 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}}})
  - This defines the post-processing for elements with variable number of nodes. It is assumed that given topological nodes form a polygon. Dummy nodes are neglected.

```
"SMSAdditionalGraphics" → {
  Null,
  Function[{ei, si, ni}, {Range[Length[ni] - Count[ni, AceFEM`SMTDummyNodes]]}],
  Function[{ei, si, ni}, {{Range[Length[ni] - Count[ni, AceFEM`SMTDummyNodes]]}}]
 }
```

Sensitivity Analysis

Abbreviation	default	description
SMSSensitivityNames	{}	description of sensitivity dependent input parameters (e.g. spatial coordinates, material parameters,..) for which sensitivity pseudo-load code is derived
SMSShapeSensitivity	False	True ⇒ shape sensitivity pseudo-load code is supported False ⇒ shape sensitivity is not supported (informative data)
SMSEBCSensitivity	False	True ⇒ essential boundary condition sensitivity pseudo-load code is supported False ⇒ essential boundary condition sensitivity is not supported (informative data)

SMSExtraSensitivityData	{0}	{
		The length of additional data storage per element needed for the execution of backward mode sensitivity analysis
		}
		see Backward mode for time dependent locally coupled problems

See also AceFEM manual sections `Sensitivity Analysis`.

AceFEM Solution Procedure Specific

Abbreviation	default	description
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* ≡ {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)
SMSPostIterationCall	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)
SMSDOFScaling	False	True ⇒ scaling of DOF is allowed False ⇒ scaling of DOF is not allowed

Description of the Element for AceShare

Abbreviation	default	description
SMSMainTitle	""	description of the element (see `SMSVerbatim` how to insert special characters such as \n or ")
SMSSubTitle	""	description of the element
SMSSubSubTitle	""	description of the element
SMSBibliography	""	reference

Environment Specific (FEAP,ELFEN, user defiend environments, ...)

Abbreviation	default	description
SMSUserSubroutine	None	generation of environment specific user subroutines (e.g. ABAQUS or ANSYS UMAT, (see `FEAP – ELFEN – ABAQUS – ANSYS`))
SMSTemplateFile	None	an alternative template file for the generation of AceGen - FE environment interface
SMSUserDataRules	{}	user defined replacement rules that transform standard input/output parameters to user defined input/output parameters (see also User Defined Environment Interface)
FEAP$*	""	FEAP specific template constants described in chapter FEAP (see `FEAP – ELFEN – ABAQUS – ANSYS`)
ELFEN$*	""	ELFEN specific template constants described in chapter ELFEN (see `FEAP – ELFEN – ABAQUS – ANSYS`)
SMSBibliography	""	reference

Constants defining the general element characteristics.

Data Structures

Contents

- General Description
- IO Data Management
  - General Input data
  - Nodal Input data
  - Element Input data
  - 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 Variables
  - 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 Variables
- 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 scalar environment variables are stored into two vectors, one for the integer type variables and the other for the real type variables. 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 Variables (in AceGen idata$$,in AceFEM SMTIData)
- Real Type Environment Variables (in AceGen rdata$$, 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 `SMSTemplate` constants (SMSTopology, SMSAdditionalNodes, SMSNodeID, SMSDOFGlobal, SMSDomainDataNames, etc.). Specific data is identified by its name (*dataName*) defined in (`IO Data Management`, `Sensitivity analysis Input / Output data`, `Tasks Input / Output data`, `AceFEM Code Generation Utility Functions`) and it can be imported form or exported to IO parameters of corresponding user subroutine with the following commands:

form	description
SMSIO[*dataName*]	returns all data associated with the *dataName* (scalar, vector of matrix)
SMSIO[*dataName*[*index*]]	if *dataName* defines a vector commands returns *index -th* element of vector
SMSIO[*dataName*[*indexes_List*]]	if *dataName* defines a vector commands returns a subset of elements defined by a set of indexes
SMSIO[*dataName*[*indexi,indexj*]]	if *dataName* defines a matrix commands returns [*indexi-th,indexj-th*] element of matrix
SMSIO[*value*, "Add to", *dataName*]	adds the *value* (scalar, vector or matrix) to the output parameter of the subroutine defined by the *dataName*
SMSIO[*value*, "Add to", *dataName*[*index*]]	adds the *value* to the *index -th* element of the vector defined by the *dataName*
SMSIO[*values*, "Add to", *dataName*[*indexes_List*]]	adds a set of *values* to the a subset of elements of vectors defined by a set of *indexes*
SMSIO[*value*, "Add to", *dataName*[*indexi,indexj*]]	adds the *value* to the [*indexi-th,indexj-th*] element of the matrix defined by the *dataName*
SMSIO[*value*, "Export to", *dataName*] SMSIO[*value*, "Export to", *dataName*[*index*]] SMSIO[*values*, "Export to", *dataName*[*indexes_List*]] SMSIO[*value*, "Export to", *dataName*[*indexi,indexj*]]	exports the *value* (scalar, vector or matrix) to the output parameter of the subroutine defined by the *dataName*

Syntax of generic SMSIO commands for static exporting and importing of IO data.

- 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 `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 `SMSTemplate` constants (SMSTopology, SMSAdditionalNodes, SMSNodeID, SMSDOFGlobal, SMSDomainDataNames, etc.) are automatically updated and they don't have to be defined with the `SMSTemplate` command.

form	description
SMSIO[*dataName*]	returns all data associated with the *dataName* (scalar, vector of matrix)
SMSIO[*dataType*, *key->addData*]	creates a new auxiliary variable of the type associated with *dataName* and associates the value and the corresponding additional *addData* to the *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*

Syntax of generic SMSIO commands for dynamic creation and exporting of IO data.

- Level 3: low-level definition of IO data definition (`Example : low – level IO data definitions`)

  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 commands.

General Input 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. es$$["IntPoints",coordinate,Ig]).

form	I/O parameter	description
Integer Type Environment Variables		
SMSIO[*idataCode*]	idata$$[*idataCode*]	returns an integer variable holding *idata$$* with the given code *idataCode* (see Integer Type Environment Variables)
Real Type Environment Variables		
SMSIO[*rdataCode*]	rdata$$[*rdataCode*]	returns an real variable holding *rdata$$* with the given code *rdataCode* (see Real Type Environment Variables)
selected set of Domain Specification Data		
SMSIO["Domain data"]	es$$["Data", *index*]	returns all domain data accordingly to the given SMSDomainDataNames and SMSDefaultData
SMSIO["Domain data"[*index*]]	es$$["Data", *index*]	returns *index -th* domain data accordingly to the given SMSDomainDataNames
SMSIO["Domain data"[*name*]]	es$$["Data", *index*]	returns domain data with the name *name* accordingly to the given SMSDomainDataNames
Numerical Integration		
SMSIO["Integration point"[*Ig*]]	es$$["IntPoints", *coordinate*,*Ig*]	returns coordinates (ξ,η,ζ) of *Ig*-th integration point (see Domain Specification Data)
SMSIO["Integration weight"[*Ig*]]	es$$["IntPoints",4,*Ig*]	returns weight in *Ig*-th integration point (see Domain Specification Data)
SMSIO["No. integration points"]	es$$["id", "NoIntPoints"]	returns number of integration points (see Domain Specification Data)
SMSIO["Integration point"[*Ig*][*code*]]	hard coded	returns coordinates (ξ,η,ζ) of *Ig*-th integration point of numerical integration rule defined by given *code* (see Numerical Integration)
SMSIO["Integration weight"[*Ig*][*code*]]	hard coded	returns weight in *Ig*-th integration point of numerical integration rule defined by given *code* (see Numerical Integration)
SMSIO["No. integration points"[*code*]]	hard coded	returns number of integration points of numerical integration rule defined by given *code* (see Numerical Integration)
SMSIO["Integration point"[All][*code*]]	hard coded	returns coordinates (ξ,η,ζ) of all integration points of numerical integration rule defined by given *code* (see Numerical Integration)
SMSIO["Integration weight"[All][*code*]]	hard coded	returns weight in all integration points of numerical integration rule defined by given *code* (see Numerical Integration)

AceFEM Code Generation Utility Functions		additional macro commands for specialized IO (SMSIsDOFConstrained, SMSNoDummyNodes, SMSLastTrueNode, SMSLastTrueDOF, SMSIsDummyNode, SMSIsRealNode)
accessing general user supplied vectors (see e.g. Globally Defined User Tasks)		
SMSIO["Integer input vector"[*index*]]	SMT$$["->TasksStructure. IntegerInput",i]	*index* -th component of integer type user supplied vector
SMSIO["Integer input vector length"]	SMT$$["->TasksStructure. IntegerInputLength "]	length of user defined integer type vector
SMSIO["Real input vector"[*index*]]	SMT$$["->TasksStructure. RealInput",i]	*index* -th component of real type user supplied vector
SMSIO["Real input vector length"]	SMT$$["->TasksStructure. RealInputLength"]	length of user defined real type vector

SMSIO commands for importing general named IO data. Argument *index* can be an integer constant or variable or a list of indexes.

Nodal Input data

form	I/O parameter	description
selected set of Node Data		
SMSIO["Node index"[*node*]]	nd$$[*node*,"id", "NodeIndex"]+1	global node index of *node -th* element node
SMSIO["All coordinates and DOFs"]	nd$$[*node*, "X", *coordinate*] nd$$[*node*, "at", *dof*]	returns {all nodes, all DOFs} accordingly to the given element topology, SMSDOFGlobal, SMSNodeID (see Node Data)
SMSIO["Nodal coordinates"]	nd$$[*node*, "X", *coordinate*]	returns coordinates of all nodes accordingly to the given element topology and SMSNodeID
SMSIO["Nodal coordinates"[*node*]]	nd$$[*node*, "X",*coordinate*]	returns coordinates of *node* where *node* is an arbitrary expressions
SMSIO["Nodal coordinates"[*node,coordinate*]]	nd$$[*node*, "X",*coordinate*]	returns *coordinate*-th spatial coordinate in node *node* where *node* and *coordinate* are an arbitrary expressions
SMSIO["Nodal coordinates"[*NodeID*]]	nd$$[*node*, "X",*coordinate*]	returns coordinates of all nodes with node identification *NodeID*
SMSIO["Nodal DOFs"]	nd$$[*node*, "at", *dof*]	returns all DOFs accordingly to the given SMSDOFGlobal, SMSNodeID in a form of a list ("SMSVariableNumberOfNodes"->False) or AceGen array ("SMSVariableNumberOfNodes"->True) at time n+1
SMSIO["Nodal DOFs"[*node,dof*]]	nd$$[*node*, "at", *dof*]	returns *dof* -th nodal DOF in node *node* where *node* and *dof* are an arbitrary expressions
SMSIO["Nodal DOFs"[*node*]]	nd$$[*node*, "at", *dof*]	returns all nodal DOFs in node *node* where 1≤ node≤NoNodes and *node* is an integer number
SMSIO["Nodal DOFs"[*NodeID*]]	nd$$[*node*, "at", *dof*]	returns all nodal DOF of all nodes with node identification *NodeID*
SMSIO["Nodal DOFs"[node,*NodeID*]]	nd$$[*node*, "at", *dof*]	returns all nodal DOF in node *node* assuming that node has node identification *NodeID* (the form is only needed when *node* is not an integer number and "SMSVariableNumberOfNodes"->True, otherwise it is equivalent to SMSIO["Nodal DOFs"[*node*]])

SMSIO["Nodal DOFs n"]	nd$$[*node*, "ap", *dof*]	returns all DOFs accordingly to the given SMSDOFGlobal, SMSNodeID at time *n*
SMSIO["Nodal DOFs n"[*node,dof*]]	nd$$[*node*, "ap", *dof*]	returns *dof*-th nodal DOF in node *node* where *node* and *dof* are an arbitrary expressions at time *n*
SMSIO["Nodal DOFs n"[*node*]]	nd$$[*node*, "at", *dof*]	returns all nodal DOF in node *node* where 1≤ node≤NoNodes at time *n*
SMSIO["Nodal DOFs n"[*NodeID*]]	nd$$[*node*, "at", *dof*]	returns all nodal DOF of all nodes with node identification *NodeID* at time *n*
SMSIO["Nodal DOFs n"[node,*NodeID*]]	nd$$[*node*, "at", *dof*]	returns all nodal DOF in node *node* assuming that node has node identification *NodeID*
SMSIO["Nodal data"]	nd$$[*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 in node *node* where *node* and *dof* are an arbitrary expressions at time *n*

SMSIO commands for importing element and node related input data.

Element Input data

form	I/O parameter	description
selected set of Element Data		
SMSIO["Element index"]	ed$$["id", "ElemIndex"]+1	global index of current element
SMSIO["Element time dependent data"[*index*]]	ed$$["ht", *index*]	returns *index*-th element time dependent data accordingly to the given SMSNoTimeStorage (data can also be dynamically allocated on ht vector, see Dynamically created IO data variables)
SMSIO["Element time dependent data"]	ed$$["ht", *index*]	returns all element time dependent data accordingly to the given SMSNoTimeStorage
SMSIO["Element time dependent data n"[*index*]]	ed$$["hp", *index*]	returns *index*-th element time dependent data at time *n* accordingly to the given SMSNoTimeStorage (data can also be dynamically allocated on ht vector, see Dynamically created IO data variables)
SMSIO["Element time dependent data n"]	ed$$["hp", *index*]	returns all element time dependent data accordingly to the given SMSNoTimeStorage
SMSIO["Element data"[*index*]]	ed$$["Data", *index*]	returns *index*-th element data accordingly to the given SMSNoElementData
SMSIO["Element data"]	ed$$["Data", *index*]	returns all element data accordingly to the given SMSNoElementData

SMSIO commands for importing element related IO data. Argument *index* can be an integer or a list of integers.

Examples: generic example

In[12]:=
```
<< "AceGen`";
SMSInitialize["test", "Environment" → "AceFEM"];
SMSTemplate["SMSTopology" → "Q1", "SMSNoTimeStorage" → 2];
SMSStandardModule["Tangent and residual"];
```

In[16]:=
```
SMSIO["All coordinates and DOFs"]
```

Out[•]=

$$\{\{\{XD_{1,1}, XD_{1,2}\}, \{XD_{2,1}, XD_{2,2}\}, \{XD_{3,1}, XD_{3,2}\}, \{XD_{4,1}, XD_{4,2}\}\}, \{\{D_{1,1}, D_{1,2}\}, \{D_{2,1}, D_{2,2}\}, \{D_{3,1}, D_{3,2}\}, \{D_{4,1}, D_{4,2}\}\}\}$$

In[17]:=
```
SMSIO["Nodal coordinates"]
```

Out[•]=

$$\{\{XD_{1,1}, XD_{1,2}\}, \{XD_{2,1}, XD_{2,2}\}, \{XD_{3,1}, XD_{3,2}\}, \{XD_{4,1}, XD_{4,2}\}\}$$

In[18]:=
```
SMSIO["Nodal coordinates"[{1, 3}]]
```

Out[•]=

$$\{\{XD_{1,1}, XD_{1,2}\}, \{XD_{3,1}, XD_{3,2}\}\}$$

In[19]:=
```
SMSIO["Nodal DOFs"[3]]
```

Out[•]=

$$\{D_{3,1}, D_{3,2}\}$$

In[23]:=
```
SMSIO[SMSInteger[2], "Export to", "ErrorStatus"]
```

Out[•]=

$$Export[Cast[2], idata\$\$_{ID_ErrorStatus}]$$

In[21]:=
```
SMSIO["Element time dependent data"[{1, 2}]]
```

Out[•]=

$$\{ed\$\$_{->ht,1}, ed\$\$_{->ht,2}\}$$

In[22]:=
```
SMSIO[{π, π^2}, "Export to", "Element time dependent data"[{1, 2}]]
```

Out[•]=

$$Export[\pi, \pi^2, ed\$\$_{->ht,1}, ed\$\$_{->ht,2}]$$

Examples: The new Input/Output mechanism makes formulating the elements with a variable number of nodes easy.

In[22]:=
```
<< "AceGen`";
SMSInitialize["test", "Environment" → "AceFEM"];
SMSTemplate["SMSTopology" → "Q1", "SMSNoNodes" → 100,
  "SMSAdditionalNodes" → Null, "SMSVariableNumberOfNodes" → True];
SMSStandardModule["Tangent and residual"];
```

- when the true number of nodes is unknown, the SMSIO command returns an SMSArray array data object with the length equal to the maximum number of nodes

In[26]:=
```
SMSIO["All coordinates and DOFs"]
```
Out[•]=

$\{\{{}_4\text{SMSXe}, {}_4\text{SMSYe}\}, {}_6\text{SMSAllDOF}\}$

- components are then taken from the arrays

In[27]:=
```
SMSIO["Nodal coordinates"[{1, 3}]]
```
Out[•]=

$\{\{{}_4\text{SMSXe}_1, {}_4\text{SMSYe}_1\}, \{{}_4\text{SMSXe}_3, {}_4\text{SMSYe}_3\}\}$

- SMSLastTrueNode[] returns an index of the last element node that is not a dummy node (at AceFEM run time!, see also `AceFEM Code Generation Utility Functions`)

In[28]:=
```
SMSDo[node, 1, SMSLastTrueNode[]];
```

- node index can be an arbitrary expression in the case of variable number of nodes “SMSVariableNumberOfNodes”->True

In[29]:=
```
X ⊨ SMSIO["Nodal coordinates"[node]]
```
Out[•]=

$\{X_1, X_2\}$

- additional information about the node identification is required to get the number of DOFs at an arbitrary node

In[30]:=
```
p ⊨ SMSIO["Nodal DOFs"[node, "D"]]
```
Out[•]=

$\{p_1, p_2\}$

In[31]:=
```
SMSEndDo[]
```

Output data

form	I/O parameter	description
SMSIO[*value*, "Add to" or "Export to", "Residual"[*index*]]	p$$[*index*]	adds or exports to residual vector where *value* can be a scalar or a vector and *index* is the position or a list of positions on the residual vector
SMSIO[*value*, "Add to" or "Export to", "Tangent"[*indexi, indexj*]]	s$$[*indexi, indexj*]	adds or exports to tangent matrix where *value* can be a scalar, vector or matrix and *indexi, indexj* are the positions or a list of positions on the tangent matrix
SMSIO[{*key1->val1, key2->val2, ...*}, "Export to", "Nodal point post"[*node*]]	npost$$[*node, keyIndex*]	export post-processing quantities {*val1,val1,..*} to node with the index *node* and associate them with the keywords SMSNPostNames={*key1, key2,...*} (see `Subroutine: Postprocessing`)
SMSIO[{*key1->* {*val11,val12,...,val_NoNodes*}, *key2->*{*val21,val22,...*}, ...}, "Export to", "Nodal point post"]	npost$$[*node, keyIndex*]	export post-processing quantities val_{ij} to j-th node and associate them with the i-th keywords SMSNPostNames={*key1, key2,...*} (see `Subroutine: Postprocessing`)
SMSIO[{*key1->val1, key2->val2, ...*}, "Export to", "Integration point post"[Ig]]	gpost$$[*Ig, keyIndex*]	export post-processing quantities {*val1,val1,..*} to integration point with the index *Ig* and associate them with the keywords SMSGPostNames={*key1, key2,...*} (see `Subroutine: Postprocessing`)

SMSIO commands for exporting specific named data.

Examples:

```
SMSIO[Rgi, "Add to", "Residual"[i]];
SMSIO[Kgij, "Add to", "Tangent"[i, j]];
SMSIO[{"DeformedMeshX" → u, ...}, "Export to", "Nodal point post"[node]];
SMSIO[{"Sxx" → σ, ...}, "Export to", "Integration point post"[Ig]];
SMSIO[{"DeformedMeshX" → {u1, u2, ...}, ...}, "Export to", "Nodal point post"];
```

Sensitivity analysis Input/Output data

form	I/O parameter	description
SMSIO["Shape velocity field"[*is*][*node, coor*]]	nd$$[*node*, "SDVF", *index (is), index (coor)*]	returns value of first order shape velocity field for *is*-th sensitivity parameter in *node*-th element node for *coor* -th coordinate ($X_1 \equiv X$, $X_2 \equiv Y$, $X_3 \equiv Z$) ($\frac{\partial X_{e,coor}}{\partial \phi_{is}}$)
SMSIO["Shape velocity field"[*is, js*][*node, coor*]]	nd$$[*node*, "SDVF", *index (is, js), index (coor)*]	returns value of second order shape velocity field for *is, js*-th sensitivity parameters in *node*-th element node for *coor* -th coordinate ($\frac{\partial^2 X_{e,coor}}{\partial \phi_{is} \partial \phi_{js}}$)
SMSIO["Shape velocity field"[*is*][*node*]] SMSIO["Shape velocity field"[*is, js*][*node*]]		returns first or second order shape velocity field in *node*-th node for all coordinates
SMSIO["Shape velocity field"[*is*]] SMSIO["Shape velocity field"[*is, js*]]		returns first or second order shape velocity field in all nodes for all coordinates
SMSIO["Parameter velocity field"[*is*][*node, iparm*]]	nd$$[*node*, "SDVF", *index (is), index (iparm)*]	returns the value of first order parameter velocity fields for *is*-th sensitivity parameter in *node*-th element node for *iparm*-th constant defined by SMSSensitivityNames ($\frac{\partial d_{e,iparm}}{\partial \phi_{is}}$)
SMSIO["Parameter velocity field"[*is, js*][*node, iparm*]]	nd$$[*node*, "SDVF", *index (is,js), index (iparm)*]	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 ($\frac{\partial^2 d_{e,par}}{\partial \phi_{is} \partial \phi_{js}}$)
SMSIO["Parameter velocity field"[*is*][*node*]] SMSIO["Parameter velocity field"[*is, js*][*node*]]		returns first or second order parameter velocity field in *node*-th node for all constant defined by SMSSensitivityNames
SMSIO["Parameter velocity field"[*is*]] SMSIO["Parameter velocity field"[*is, js*]]		returns first or second order parameter velocity field in all nodes for all constant defined by SMSSensitivityNames
SMSIO["Sensitivity DOFs"[*is*][*node, dof*]]	nd$$[*node*,"st", *index (is), dof*]	returns first derivative of *dof* -th nodal DOF for *is*-th sensitivity parameter in *node*-th element node ($\frac{\partial p_{e,dof}}{\partial \phi_{is}}$)
SMSIO["Sensitivity DOFs"[*is, js*][*node, dof*]]	nd$$[*node*,"st", *index (is, js), dof*]	returns second derivative of *dof* -th nodal DOF for *is, js*-th sensitivity parameters in *node*-th element node ($\frac{\partial^2 p_{e,dof}}{\partial \phi_{is} \partial \phi_{js}}$)
SMSIO["Sensitivity DOFs"[*is*][*node*]] SMSIO["Sensitivity DOFs"[*is, js*][*node*]]		returns first or second derivative of all nodal DOF in *node*-th element node
SMSIO["Sensitivity DOFs"[*is*]] SMSIO["Sensitivity DOFs"[*is, js*]]		returns first or second derivative of all nodal DOF in all element nodes

SMSIO["Sensitivity DOFs n"[*is*][*node, dof*]] SMSIO["Sensitivity DOFs n"[*is, js*][*node, dof*]] SMSIO["Sensitivity DOFs n"[*is*][*node*]] SMSIO["Sensitivity DOFs n"[*is, js*][*node*]] SMSIO["Sensitivity DOFs n"[*is*]] SMSIO["Sensitivity DOFs n"[*is, js*]]	nd$$[*node*,"sp", *index (is), dof*]	derivatives of nodal DOFs in time *n*
SMSIO["Adjoint vector DOFs"[*ifunc*][*node, dof*]] SMSIO["Adjoint vector DOFs"[*ifunc*][*node*]] SMSIO["Adjoint vector DOFs"[*ifunc*]]	nd$$[*node*, "BSAV", *ifunc, dof*]	returns the value of adjoint vector for *ifunc -th* functional in *node*-th node and *dof*-th nodal DOF (see Backward Mode Implementation Notes)
SMSIO["Backward sensitivity element storage"[{*pos1,pos2,…*}]] SMSIO["Backward sensitivity element storage"[*pos*]]	ed$$["BSETS", *pos*]	returns data stored for each element for the implementation of bacward sensitivity at position pos_i on the allocated "BSETS" data field (see Backward Mode Implementation Notes)
SMSIO["Integer input vector"[*pos*]]	SMT$$["->TasksStructure. IntegerInput", *pos*]	additional user input data accessible from backward mode element subroutines (see Backward sensitivity load, Backward sensitivity derivatives)
SMSIO["Integer input vector length"[*pos*]]	SMT$$["->TasksStructure. IntegerInputLength "]	
SMSIO["Real input vector"[*pos*]]	SMT$$["->TasksStructure. RealInput", *pos*]	
SMSIO["Real input vector length"[*pos*]]	SMT$$["->TasksStructure. RealInputLength"]	

SMSIO commands for importing sensitivity analysis related data.

form	I/O parameter	description
SMSIO[*value*, "Add to" or "Export to", "First order pseudo load"[*is*][*dof*]]	s$$[*index (is), dof*]	adds or exports to first order sensitivity pseudo load *value* for *is-th* sensitvity parameter and *dof-th* noda
SMSIO[*values*, "Add to" or "Export to", "First order pseudo load"[*is*]]	s$$[*index (is), dof*]	adds or exports to first order sensitivity load *values* for *is-th* sensitvity paramet
SMSIO[*value*, "Add to" or "Export to", "Second order pseudo load"[*is,js*][*dof*]] SMSIO[*values*, "Add to" or "Export to", "Second order pseudo load"[*is,js*]]	s$$[*index (is,js), dof*]	adds or exports to second order sensitivity pseudo load *value* for *is, js-th* sensitvity parameters and *dof -th*
SMSIO[*value*, "Add to" or "Export to", "Adjoint pseudo–load"[*ifunc*][*dof*]] SMSIO[*values*, "Add to" or "Export to", "Adjoint pseudo–load"[*ifunc*]]	s$$[*ifunc, dof*]	adds or exports adjoint vector of *sof -th* nodal DOF for *ifunc -th* functional (see Backward sensitivity
SMSIO[*value*, "Add to" or "Export to", "Backward sensitivity element storage"[*pos*][fog]]	ed$$["BSETS", *pos*]	adds or exports *value* at position *pos* on the allocated "BSETS" data fiel (see Backward Mode Implementation N
SMSIO[*value*, "Add to" or "Export to", "Functional derivative"[*ifunc, is*]]	fd$$[*ifunc, index (is)*]	adds or exports *value* of first derivative *-th* functional for *is*-th parameter ($\frac{\partial F_{ifunc}}{\partial \phi_{is}}$) (see Backward Mode Implementation N
SMSIO[*value*, "Add to" or "Export to", "Functional derivative"[*ifunc,is,js*]]	fd$$[*ifunc, index (is,js)*]	adds or exports *value* of second derivative of *ifunc -th* functional for *is, js*-th parameter ($\frac{\partial^2 F_{ifunc}}{\partial \phi_{is} \partial \phi_{js}}$) (see Backward Mode Implementation N

SMSIO commands for exporting sensitivity analysis related data.

- For details see `Forward Mode Implementation Notes`, `Backward Mode Implementation Notes`.

Tasks Input/Output data

form	I/O parameter	description
SMSIO["Task index"]	Tasks$$	see User Defined Tasks
SMSIO["Task data"[*index*]]	TasksData$$[*index*]	see User Defined Tasks
SMSIO["Task real input"[*index*]]	RealInput$$[*index*]	see User Defined Tasks
SMSIO["Task integer input"[*index*]]	IntegerInput$$[*index*]	see User Defined Tasks
SMSIO[*value*, "Add to" or "Export to", "Task data"[*index*]]	TasksData$$[*index*]	see User Defined Tasks
SMSIO[*value*, "Add to" or "Export to", "Task real output"[*index*]]	RealOutput$$[*index*]	see User Defined Tasks
SMSIO[*value*, "Add to" or "Export to", "Task integer output"[*index*]]	IntegerOutput$$[*index*]	see User Defined Tasks
SMSIO[*value*, "Add to" or "Export to", "Task local vector"[*index*]]	p$$[*index*]	see User Defined Tasks
SMSIO[*value*, "Add to" or "Export to", "Task local matrix"[*indexi,indexj*]]	s$$[*indexi,indexj*]	see User Defined Tasks

SMSIO commands for I/O of data related to tasks.

Example: IO data defined by SMSTemplate constants

In[103]:=

```
<< "AceGen`";
SMSInitialize["tmpTest", "Environment" → "AceFEM"];
SMSTemplate["SMSTopology" → "Q1", "SMSSymmetricTangent" → 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 ⊨ 1/4 {(1 - ξ) (1 - η), (1 + ξ) (1 - η), (1 + ξ) (1 + η), (1 - ξ) (1 + η)};
  X ⊢ SMSFreeze[Append[Nh.XIO, ζ]];
  Je ⊨ SMSD[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, tζ} ⊨ SMSIO["Domain data"];
  {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 tζ 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]], "Szx" → σ[[3, 1]], "Szy" → σ[[3, 2]], "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[];
```

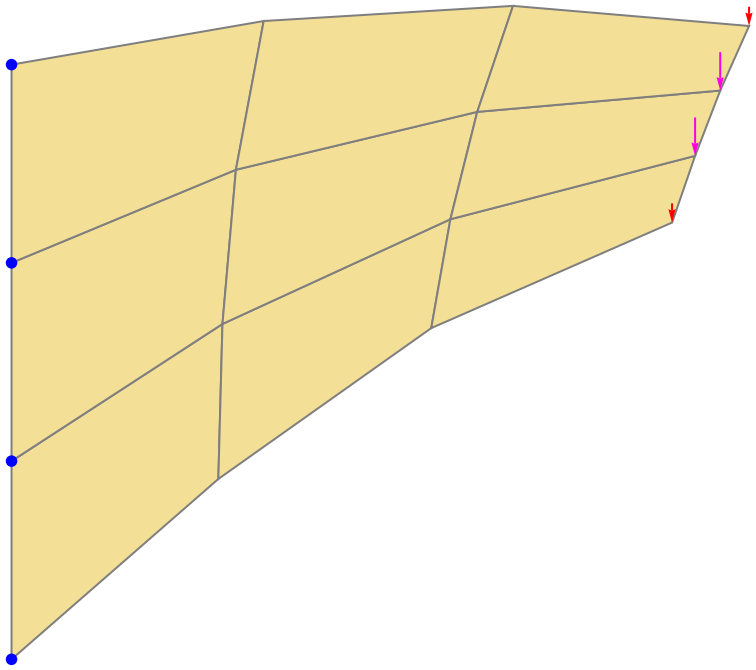
File: tmpTest.c Size: 11475 Time: 2

Method	SKR	SPP
No.Formulae	94	76
No.Leafs	1395	1286

In[122]:=

```
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; Δλ = 0.1; tolNR = 10^-8; maxNR = 15;
Do[
 SMTNextStep["Δλ" → Δλ];
 While[SMTConvergence[tolNR, maxNR], SMTNewtonIteration[];];
 , {i, 1, nstep}]
SMTShowMesh["DeformedMesh" → True, "BoundaryConditions" → True]
```

Out[ ]=



Dynamically created IO data variables

A simple IO interface presented in previous chapter uses template constants such as SMSTopology to create proper I/O 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	data type	description
SMSIO["Nodal coordinates", *key*]	nd$$[*node*, "X", *coordinate*]	adds and/or returns a new topological node
SMSIO["Nodal DOFs", *key*->{*nDOF*,*nodeID*}]	nd$$[*node*, "at", *dof*]	declares that the next node has given *nodeID* and *nDOF* nodal DOFs and returns a set nodal DOFs at time n+1 (see Node Data)
SMSIO["Nodal DOFs", *key*]	nd$$[*node*, "at", *dof*]	returns a set of nodal DOFs at time n+ 1 associated with the given key (see Node Data)
SMSIO["Nodal DOFs n", *key*]	nd$$[*node*, "ap", *dof*]	returns nodal DOFs at time n in the node associated with the given *key* (*key* must be the same as previously defined for the time n+1) (see Node Data)
SMSIO["Element time dependent data", *key*]	ed$$["ht",*i*]	adds and returns a new element time dependent variable at time n+1 (see Element Data)
SMSIO["Element time dependent data n", *key*]	ed$$["hp",*i*]	returns time dependent variable at time n associated with the given key (*key* must be the same as previously defined for the time n+1) (see Element Data)
SMSIO["Element data", *key*]	ed$$["Data",_]	adds and/or returns general element data variable associated with the given key (see Element Data)
SMSIO["Domain data", *key*->{*domainDataName*, {*defaultData*, *benchmarkValue*}, *isSensitivityParameter*}]	es$$["Data",*i*]	adds and returns a new domain data variable (see Domain Specification Data) defined by identification *domainDataName* and with the default value *defaultData.* The *benchmarkValue* is numerical value of the data used within the standard benchmark tests. If *isSensitivityParameter* is True then the parameter will be one of general sensitivity parameters.
SMSIO["Domain data", *key*]	es$$["Data",*i*]	returns domain data variable associated with the given key
SMSIO["Integer type switch", *key*->{*switchValue*, *switchDescription*}]	es$$["IntSwitch",*i*]	adds and returns the integer type switch variable (see Domain Specification Data, SMSIntSwitch). The *switchValue* is added to SMSIntSwitch and *switchDescription* is used to make documentation.
SMSIO["Integer type switch", *key*]	es$$["IntSwitch",*i*]	returns the value of integer type switch associated with the given key (see Domain Specification Data)
SMSIO["Real type switch", *key*->{*switchValue*, *switchDescription*}]	es$$["DoubleSwitch",*i*]	adds and returns the real type switch variable (see Domain Specification Data, SMSDoubleSwitch). The *switchValue* is added to SMSDoubleSwitch template constant and *switchDescription* is used to make documentation.
SMSIO["Real type switch", *key*]	es$$["DoubleSwitch",*i*]	returns the value of real type switch associated with the given key (see Domain Specification Data)
SMSIO["Char type switch", *key*->*switchDescription*]	es$$["CharSwitch",*i*]	*the key* is added to SMSCharSwitch template constant and *switchDescription* is used to make documentation. Character type switch does not have a value that can be changed, it is only a switch stored into es$$["CharSwitch",i]!

Basis SMSIO commands for dynamically created IO variables associated with the given 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*, {key_1->$addData_1$, key_2->$addData_2$,...}]	creates a set of new auxiliary variables of the type associated with *dataType* and associates the values and the corresponding additional data $addData_i$ to the given keys key_i.
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.

All SMSIO commands for dynamically importing and exporting an arbitrary data associated with the given key/keys.

In[24]:=
```
<< AceFEM`;
```

Examples:

```
SMSIO["Nodal coordinates", Table["X"[i], {i, SMSNoNodes}]];
SMSIO["Nodal DOFs", Table["u"[i] → {SMSNoDimensions, "D"}, {i, SMSNoNodes}]];
SMSIO["Domain data", "Em" → {"E -elastic modulus", 21000}];
SMSIO["Element time dependent data", Table["ℍeIO"[i], {i, nhe}]];
SMSIO[Table[π, {nhe}], "Export to",
  "Element time dependent data", Table["ℍeIO"[i], {i, nhe}]];
SMSIO["Element time dependent data", Table["ℍgIO"[i] → {Hold[NoIp], Ig}, {i, nhg}]];
SMSIO["Element data", "weighth"];
SMSIO["Integer type switch", "N" → {10, "number of ..."}];
SMSIO["Char type switch", "contact" → "element is a contact finite element"];
```

- key can also be a pattern, e.g. SMSIO["Nodal coordinates", "u"[_]] would return a set of nodal coordinates of all nodes with associated key that matches pattern "u"[_].

dataType	*addData*	description

"Nodal coordinates" ≡nd$$[_,"X",_]	*nodeType*	Pattern defines a new node (see `Node Data`, Node Identification) (nd$$[i,"X",j] is j^{th} component of the spatial coordinate of ith element node). The value of the *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 *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. 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" ≡nd$$[_,"at",_]	{*nDOF*,*nodeID*}	Pattern defines DOF–s of the node with node identification *nodeID and* with *nDOF* unknowns (see `Node Data`) at time n+1 (nd$$[i,"at",j] is j^{th} component of the vector of DOF–s of ith element node at time n+1).
"Nodal DOFs n" ≡nd$$[_,"ap",_]	Automatic	Pattern defines DOFs associated with the node at time n (see `Node Data`) (nd$$[i,"ap",j] is j^{th} component of the vector of DOF–s of ith element node at time n). *addData* is taken from the corresponding "Nodal DOFs".
"Element time dependent data" ≡ed$$["ht",_]	Null	Pattern defines a current value of history dependent real type variable automatically allocated on element data field "ht" (see Element Data). The SMSIO returns auxiliary variable that points to corresponding element on "ht".
"Element time dependent data" ≡ed$$["ht",_]	{*length*, *index1*}	Pattern defines a current value of history dependent real type vector of length *length* automatically allocated on element data field "ht" (see Element Data). The SMSIO returns auxiliary variable that points to the *index*–th element of the allocated vector. If *index1*= Null then only vector is allocated.
"Element time dependent data" ≡ed$$["ht",_]	*index2*	Real type vector allocated on element data field "ht" can have an arbitrary number of indices assigned. Only the first time the length has to be specified as well.
"Element time dependent data n" ≡ed$$["hp",_]	*index*	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 n" data! The SMSIO returns auxiliary variable that points to the *index*–th element of the allocated vector.

"Domain data" ≡ es$$["Data",_]	{*domainDataName*, {*defaultData*, *benchmarkValue*}, *isSensitivityParameter*}	Pattern defines domain data or material data (see Domain Specification Data) defined by identification *domainDataName* and with the default value *defaultData.* The *data is* associated to the specific type of elements (es$$["Data",i] is i^{th} component of the domain data vector). The *benchmarkValue* is numerical value of the data used within the standard benchmark tests. If *isSensitivityParameter* is True then the parameter will be one of general sensitivity parameters.
"Domain data" ≡ es$$["Data",_]	{*domainDataName*, *defaultData*}	≡ {*domainDataName*, {*defaultData*, *defaultData*}, *True*} simple input
"Char type switch" ≡es$$["CharSwitch", _]	*switchDescription*	Pattern defines character type switch associated to the specific type of elements (see Domain Specification Data, SMSCharSwitch, es$$["CharSwitch",i] is i^{th} character type constant). The *key* is added to SMSCharSwitch template constant and written into domain data structure es$$["CharSwitch",i] and *switchDescription* is used to make documentation. Character type switch does not have a value that can be changed, it is only a switch stored in es$$["CharSwitch",i]!
"Integer type switch" ≡es$$["IntSwitch",_]	{*switchValue*, *switchDescription*}	Pattern defines itialization value *switchValue of* integer type switch associated to the specific type of elements (see Domain Specification Data, es$$["IntSwitch",i] is i^{th} integer type constant). The *switchValue* is written into domain data structure and *switchDescription* is used to make documentation.
"Real type switch" ≡es$$["DoubleSwitch ",_]	{*switchValue*, *switchDescription*}	Pattern defines itialization value *switchValue* of real type switch associated to the specific type of elements (see Domain Specification Data, es$$["DoubleSwitch",i] is i^{th} real type constant). The *switchValue* is written into domain data structure and *switchDescription* is used to make documentation.
"Element data" ≡ ed$$["Data",_]	Null {*length*, *index1*} *index2*	Pattern defines real type variable(s) automatically allocated on element data field "Data" (see Element Data). Depending on *addData* command does: Null ⇒ allocates and returns scalar variable {*length*, *index1*} ⇒ allocates vector of length *length* and returns scalar variable that points to *index1*–th element of allocated vector {*length*, Null} ⇒ allocates vector of length *length* *index2* ⇒ returns scalar variable that points to *index2*–th element of previously allocated vector

Data structures with an automatic data management support.

Example: dynamically created IO data variables

In[59]:=

```
<< "AceGen`";
SMSInitialize["tmpTest", "Environment" → "AceFEM"];
SMSTemplate["SMSTopology" → "Q1", "SMSSymmetricTangent" → True];

SMSStandardModule["Tangent and residual"];
SMSDo[Ig, 1, SMSIO["No. integration points"]];
SMSSaveAndEvaluate["Element definitions",
  Ξ = {ξ, η, ζ} ⊨ 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, ζ]];
  Je ⊨ SMSD[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, tζ} ⊨ SMSIO["Domain data", {
     "Em" → {"E -elastic modulus", 21000},
     "ν" → {"ν -poisson ratio", 0.3},
     "tζ" → {"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 tζ 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", "u"[_]]];
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]], "Szx" → σ[[3, 1]], "Szy" → σ[[3, 2]], "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[];
```

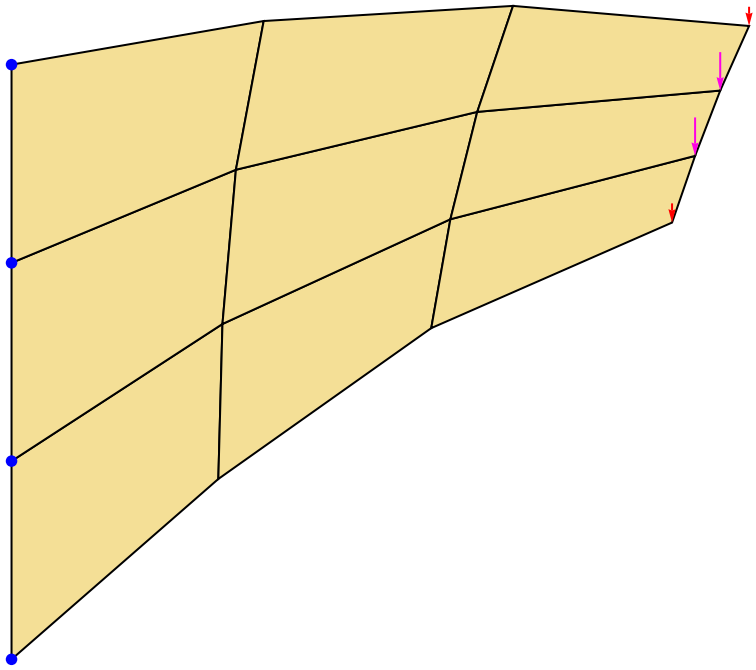
File: tmpTest.c **Size:** 11413 **Time:** 3

Method	SKR	SPP
No.Formulae	94	76
No.Leafs	1395	1286

In[79]:=

```
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; Δλ = 0.1; tolNR = 10^-8; maxNR = 15;
Do[
 SMTNextStep["Δλ" → Δλ];
 While[SMTConvergence[tolNR, maxNR], SMTNewtonIteration[];];
 , {i, 1, nstep}]
SMTShowMesh["DeformedMesh" → True, "BoundaryConditions" → True]
```

Out[ ]=



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

In[81]:=

```
<< "AceGen`";
SMSInitialize["tmpTest", "Environment" → "AceFEM"];
SMSTemplate["SMSTopology" → "Q1"];
```

- Definition of the first user subroutine.

In[84]:=

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

- definition of topological nodes and nodal unknowns for discretization of displacements

In[85]:=
```
X = SMSIO["Nodal coordinates", Table["X"[i], {i, SMSNoNodes}]]
p = SMSIO["Nodal DOFs", Table["u"[i] → {SMSNoDimensions, "D"}, {i, SMSNoNodes}]]
```

Out[•]=

$$\{\{X_{1,1}, X_{1,2}\}, \{X_{2,1}, X_{2,2}\}, \{X_{3,1}, X_{3,2}\}, \{X_{4,1}, X_{4,2}\}\}$$

Out[•]=

$$\{\{u_{1,1}, u_{1,2}\}, \{u_{2,1}, u_{2,2}\}, \{u_{3,1}, u_{3,2}\}, \{u_{4,1}, u_{4,2}\}\}$$

In[87]:=
```
{SMSNoNodes, SMSNodeID, SMSDOFGlobal, SMSAdditionalNodes}
```

Out[•]=
```
{4, {D, D, D, D}, {2, 2, 2, 2}, {}&}
```

In[88]:=
```
SMSIO["Nodal coordinates"]
```

Out[•]=

$$\{\{X_{1,1}, X_{1,2}\}, \{X_{2,1}, X_{2,2}\}, \{X_{3,1}, X_{3,2}\}, \{X_{4,1}, X_{4,2}\}\}$$

In[89]:=
```
SMSIO["Nodal coordinates", "X"[3]]
```

Out[•]=

$$\{X_{3,1}, X_{3,2}\}$$

In[90]:=
```
SMSIOInfo["Nodal coordinates", "X"[3], "Export"]
```

Out[•]=
```
{nd$$[3, X, 1], nd$$[3, X, 2]}
```

In[91]:=
```
SMSIOInfo["Nodal coordinates", "Key"]
```

Out[•]=
```
{X[1], X[2], X[3], X[4]}
```

- definition of additional nodes and nodal unknowns for discretization of temperature field

In[92]:=
```
𝕏T = SMSIO["Nodal coordinates", Table["XT"[i] → i, {i, 4}]]
𝕋IO = SMSIO["Nodal DOFs", Table["Te"[i] → {1, "T -M"}, {i, 4}]]
```

Out[•]=

$$\{\{XT_{5,1}, XT_{5,2}\}, \{XT_{6,1}, XT_{6,2}\}, \{XT_{7,1}, XT_{7,2}\}, \{XT_{8,1}, XT_{8,2}\}\}$$

Out[•]=

$$\{\{Te_{5,1}\}, \{Te_{6,1}\}, \{Te_{7,1}\}, \{Te_{8,1}\}\}$$

In[94]:=
```
{SMSNoNodes, SMSNodeID, SMSDOFGlobal, SMSAdditionalNodes}
```

Out[•]=
```
{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[95]:=
```
SMSIO["Nodal coordinates", "LC" → "Auxiliary"]
```

Out[•]=

$$\{LC_{9,1}, LC_{9,2}\}$$

In[96]:=

```
𝕃IO = SMSIO["Nodal DOFs", "L" → {1, "LC -LP"}]
```

Out[•]=

$$\{\{L_{9,1}\}\}$$

In[97]:=

```
{SMSNoNodes, SMSNodeID, SMSDOFGlobal, SMSAdditionalNodes}
```

Out[•]=

```
{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[98]:=

```
SMSStandardModule["Postprocessing"];
```

- IO data structures are automatically updated based on previous definitions

In[99]:=

```
SMSIO["Nodal coordinates", "X"[3]]
```

Out[•]=

$$\{\{X_{3,1}, X_{3,2}\}\}$$

In[100]:=

```
𝕏IO = SMSIO["Nodal coordinates"]
```

Out[•]=

$$\{\{X_{1,1}, X_{1,2}\}, \{X_{2,1}, X_{2,2}\}, \{X_{3,1}, X_{3,2}\}, \{X_{4,1}, X_{4,2}\}, \{XT_{5,1}, XT_{5,2}\}, \{XT_{6,1}, XT_{6,2}\}, \{XT_{7,1}, XT_{7,2}\}, \{XT_{8,1}, XT_{8,2}\}, \{LC_{9,1}, LC_{9,2}\}\}$$

In[101]:=

```
𝕡eIO = SMSIO["Nodal DOFs"]
```

Out[•]=

$$\{\{u_{1,1}, u_{1,2}\}, \{u_{2,1}, u_{2,2}\}, \{u_{3,1}, u_{3,2}\}, \{u_{4,1}, u_{4,2}\}, \{Te_{5,1}\}, \{Te_{6,1}\}, \{Te_{7,1}\}, \{Te_{8,1}\}, \{L_{9,1}\}\}$$

Example: low-level IO data definitions

In[88]:=

```
<< "AceGen`";
SMSInitialize["tmpTest", "Environment" → "AceFEM"];
SMSTemplate["SMSTopology" → "Q1", "SMSSymmetricTangent" → True,
  "SMSDomainDataNames" → {"E -elastic modulus", "ν -poisson ratio", "t -thickness"},
  "SMSDefaultData" → {21000, 0.3, 1}
 ];
SMSStandardModule["Tangent and residual"];
SMSDo[Ig, 1, SMSInteger[es$$["id", "NoIntPoints"]]];
Ξ = {ξ, η, ζ} ⊢ Table[SMSReal[es$$["IntPoints", i, Ig]], {i, 3}];
XIO ⊢ Table[SMSReal[nd$$[i, "X", j]], {i, SMSNoNodes}, {j, SMSNoDimensions}];
uIO ⊢ SMSReal[Table[nd$$[i, "at", j], {i, SMSNoNodes}, {j, SMSDOFGlobal[[i]]}]];
Nh ⊨ 1/4 {(1 - ξ) (1 - η), (1 + ξ) (1 - η), (1 + ξ) (1 + η), (1 - ξ) (1 + η)};
X ⊢ SMSFreeze[Append[Nh.XIO, ζ]];
Je ⊨ SMSD[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, tζ} ⊢ SMSReal[Table[es$$["Data", i], {i, Length[SMSDomainDataNames]}]];
{λ, μ} ⊨ SMSHookeToLame[Em, ν];
W ⊨ 1/2 λ (JF - 1)^2 + μ (1/2 (Tr[Ct] - 3) - Log[JF]);
wgp ⊢ SMSReal[es$$["IntPoints", 4, Ig]];
pe = Flatten[uIO];
SMSDo[
  Rgi ⊨ Jed tζ wgp SMSD[W, pe, i];
  SMSExport[Rgi, p$$[i], "AddIn" → True];
  SMSDo[
   Kgij ⊨ SMSD[Rgi, pe, j];
   SMSExport[Kgij, s$$[i, j], "AddIn" → True];
   , {j, i, 8}];
  , {i, 1, 8}];
SMSEndDo[];
SMSWrite[];
```

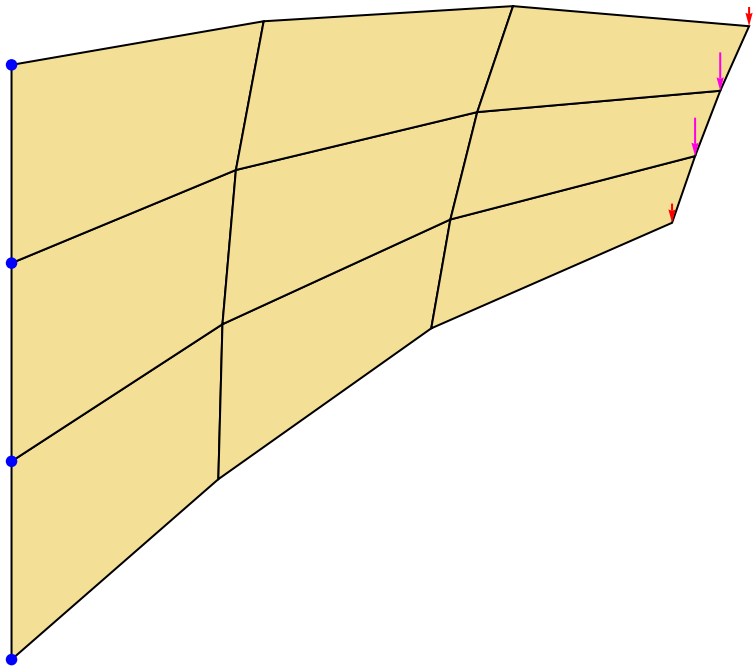
File: tmpTest.c Size: 7494 Time: 2

Method	SKR
No.Formulae	93
No.Leafs	1389

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 → 0, 2 → 0];
SMTAddNaturalBoundary[ Line[{{48, 44}, {48, 44 + 16}}], 2 → Line[{-1000}]];
SMTAnalysis[];
nstep = 10; Δλ = 0.1; tolNR = 10^-8; maxNR = 15;
Do[
 SMTNextStep["Δλ" → Δλ];
 While[SMTConvergence[tolNR, maxNR], SMTNewtonIteration[];];
 , {i, 1, nstep}]
SMTShowMesh["DeformedMesh" → True, "BoundaryConditions" → True]
```

Out[ ]=



Utility IO data management commands

SMSUndefineIO[]

clear all definitions of IO data, thus they can be defined with SMSIO[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*

*taskCode*s for the SMSIOInfo function.

Integer Type Environment Variables

- `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 `IO Data Management`, `General Input data`, `SMTIData`).

General data

Default form	Default	description
idata$$["IDataLength"]	200/R	actual length of idata vector
idata$$["RDataLength"]	200/R	actual length of rdata vector
idata$$["IDataLast"]	?/R	index of the last value reserved on *idata* vector (we can store additional user defined data after this point)
idata$$["RDataLast"]	?/R	index of the last value reserved on r*data* vector (we can store additional user defined data after this point)
idata$$["LastIntCode"]	?/R	last integration code for which numerical integration points and weights were calculated
idata$$["OutputFile"]	?/R	output file number or output channel number
idata$$["SymmetricTangent"]	?/R	1 ⇒ global tangent matrix is symmetric 0 ⇒ global tangent matrix is unsymmetrical
idata$$["MinNoTmpData"]	3	minimum number of real type variables per node stored temporarily (actual number of additional temporary variables per node is calculated as Max["MinNoTmpData", number of nodal d.o.f])

idata$$["Task"]	?/R	code of the current task performed -1 - Quit 0 - undefined 1000+n - User n 1 - standard NR iteration 2 - residual 3 - arc length iteration 4 - arc length estimate 5 - postprocessing 6 - global part of first order forward mode sensitivity 7 - local part of first order forward mode sensitivity 8 - global part of second order forward mode sensitivity 9 - local part of second order forward mode sensitivity 10 - first order backward mode sensitivity 11 - second order backward mode sensitivity
idata$$["CurrentElement"]	0/R	index of the current element processed, **valid only without parallelization!, also ed$$["id","ElemIndex"]**
idata$$["TmpContents"]	0	the meaning of the temporary real type variables stored during the execution of a single analysis into nd$$[i,"tmp", j] data structure 0 ⇒ not used 1 ⇒ residual (reactions) 2 ⇒ used for post-processing
idata$$["AssemblyNodeResidual"]	0	0 ⇒ residual vector is not formed separately 1 ⇒ during the execution of the SMTNewtonIteration command the residual vector is formed separately and stored into nd$$[i,"tmp", j] (at the end the nd$$[i,"tmp", j] contains the j-th component of the nodal reaction in the i-th node) (see SMTResidual)
idata$$["Debug"]	0/RW	1 ⇒ prevent closing of the CDriver console on exit
idata$$["DataMemory"]	0	memory used to store data (bytes)
idata$$["GeometricTangentMatrix"]	0	Used for buckling analysis ($K_0+\lambda K_\sigma$) $\{\Psi\}=\{0\}$): 0 ⇒ form full nonlinear matrix 1 ⇒ form K_0 2 ⇒ form K_σ 3 ⇒ form K_0+K_σ **ONLY VALID FOR ELEMENTS THAT SUPPORT THE OPTION**.
idata$$["ExtrapolationType"]	0/RW	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`)
idata$$["NoThreads"]	All available	number of processors that are available for the parallel execution
idata$$["TempVectorSize"]	100 000	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.
idata$$["ConsoleWindow"]	?	True of CDriver opens console window and False when printing is only possible to notebook

Mesh input related data

Default form	Default	description
idata$$["NoAllNodes"]	?/R	total number of nodes
idata$$["NoNodes"]	?/R	number of mesh nodes
idata$$["NoSystemNodes"]	?/R	number of system nodes (e.g. dummy nodes, global Lagrange nodes, etc..)
idata$$["NoElements"]	?/R	total number of elements
idata$$["NoESpec"]	?/R	total number of domains
idata$$["NoDimensions"]	?/R	number of spatial dimensions of the problem (2 or 3)
idata$$["NoNSpec"]	?/R	total number of node specifications
idata$$["NoEquations"]	?/R	total number of global equations
idata$$["DummyNodes"]	0	1 ⇒ dummy nodes are supported for the current analysis
idata$$["NoMultipliers"]	1	number of boundary conditions multipliers
idata$$["MaxEquations"]	?/R	maximum number of global equations (without constrains)

Iterative procedure related data

See: `Iterative Solution Procedures`, `SMTConvergence`, `SMTStatusReport`, `SMTErrorCheck`

Default form	Default	description
idata$$["Iteration"]	?/R	index of the current iteration within the iterative loop
idata$$["TotalIteration"]	?/R	total number of iterations in session
idata$$["Step"]	0	total number of completed solution steps (set by Newton-Raphson iterative procedure)
idata$$["ErrorStatus"]	0/RW	code for the type of the most important error event (see `SMTErrorCheck`)
idata$$["LinearEstimate"]	0/RW	the value of the "EBCUpdate" option of `SMTNewtonIteration` command
idata$$["ResidualAndTangentTask"]	?	SMTNewtonIteration is performing: 0 ⇒ corrector iteration (standard Newton-Raphson or related iteration) 1 ⇒ predictor iteration (usualy the first iteration) 2 ⇒ recalculate element local equations and update history vectors (**ht,** for locally coupled problems) 3 ⇒ evaluate $\mathbf{R}_e$ and $\mathbf{K}_e$ for current element state (no other actions should be taken) The value is set by various procedures and should be properly interprated by all elements.
idata$$["PostIteration"]	0	is set by the SMTConvergence command to 1 if idata$$["PostIterationCall"]=1 or the SMTConvergence has been called with switch "PostIteration" ->True
idata$$["PostIterationCall"]	0	1 ⇒ additional call of the SKR user subroutines after the convergence of the global solution is enabled in at least one of the elements ("SMSPostIterationCall"->True)
idata$$["SkipTangent"]	0	1 ⇒ the global tangent matrix is not assemled
idata$$["SkipResidual"]	0	1 ⇒ the global residual vector is not assembled
idata$$["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
idata$$["NoSubIterations"]	0/R	maximal number of local sub-iterative process iterations performed during the analysis
idata$$["SubIterationMode"]	0	Switch used in the case that alternating solution has been detected by the `SMTConvergence` function. 0 ⇒ ${}_{i+1}\mathbf{b}_0^t=\mathbf{b}^p$ ≥1 ⇒ ${}_{i+1}\mathbf{b}_0^t={}_i\mathbf{b}^t$
idata$$["GlobalIterationMode"]	0	Switch used in the case that alternating solution has been detected by the `SMTConvergence` function. 0 ⇒ no restrictions on global equations ≥1 ⇒ freeze all "If" statements (e.g. nodes in contact, plastic-elastic regime)
idata$$["MaxPhysicalState"]	0/RW	used for the indication of the physical state of the element (e.g. 0-elastic, 1-plastic, etc., user controlled option)
idata$$["LineSearchUpdate"]	False	activate line search procedure (see also rdata$$["LineSearchStepLength"])
idata$$["NoBackStep"]	0	number of failed iterative solution steps
idata$$["DOFScaling"]	0	1 ⇒ scaling of DOF is performed 0 ⇒ scaling of DOF is not performed
idata$$["NoDiscreteEvents"]	0	number of discrete events recordered during the NR-iteration by the elements (e.g. new contact node, transformation from elastic to plastic regime)

Debugging and errors related data

See: `Iterative Solution Procedures`, `SMTConvergence`, `SMTStatusReport`, `SMTErrorCheck`

Default form	Default	description
idata$$["MaxMessages"]	100	Printing is suspended if the actual number of printouts exceeds the maximum number allowed as defined by SMTIData["MaxMessages"] environment variable. Warning message is produced when the number of printouts reaches SMTIData["MaxMessages"], see SMSPrintMessage.
idata$$["ErrorStatus"]	0/RW	code for the type of the most important error event (see `SMTErrorCheck`)
idata$$["SubDivergence"]	0/RW	number of the "Divergence of the local sub-iterative process" error events detected form the last error check
idata$$["ErrorElement"]	0	last element where error event occurred
idata$$["MaterialState"]	0/RW	number of the "Non-physical material point state" error events detected form the last error check
idata$$["ElementShape"]	0/RW	number of the "Non-physical element shape" error events detected form the last error check
idata$$["MissingSubroutine"]	0/RW	number of the "Missing user defined subroutine" error events detected form the last error check
idata$$["ElementState"]	0/RW	number of the "Non-physical element state" error events detected form the last error check
idata$$["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

Default form	Default	description
idata$$["SkipSolver"]	0	0 ⇒ full Newton-Raphson iteration 1 ⇒ the tangent matrix and the residual vector are assembled but the resulting sistem of equations is not solved
idata$$["SetSolver"]	0	1 ⇒ recalculate solver dependent data structures if needed
idata$$["SolverMemory"]	0	memory used by solver (bytes)
idata$$["TangentMatrixMemory"]	0	memory used to store global tangent matrix (bytes)
idata$$["Solver"]	0	solver identification number
idata$$["Solver1"]× idata$$["Solver2"]× idata$$["Solver3"]× idata$$["Solver4"]× idata$$["Solver5"]		solver specific parameters
idata$$["ZeroPivots"]	0	number of near-zero pivots (see also `SMTSetSolver`)
idata$$["NegativePivots"]	0	number of negative pivots (or -1 if data is not available), (see also `SMTSetSolver`)
idata$$["NoLinearConstraints"]	0	number of linear constraint equations
data$$["MatrixStorageScheme"]	0	0 - not set 1 -symmetric skyline 2 -unsymmetric skyline 3- symmetric sparse 4- unsymmetric sparse

Sensitivity related data

Default form	Forward (F)/ Backward (B) method	description
idata$$["SensitivityScheme"]	0	general sensitivity analysis scheme 0 - no sensitivity analysis 1 - first order forward mode sensitivity analysis 2 - first order forward mode sensitivity, second order forward mode sensitivity 3 - first order backward mode sensitivity analysis 4 - first order forward mode sensitivity, second order backward mode sensitivity
idata$$["NoSensParameters"]	F/B	number of sensitivity parameters of the problem (Φ)
idata$$["NoSensDerivatives"]	F	NoFirstOrderDerivatives+ NoSecondOrderDerivatives+NoThirdOrderDerivatives
idata$$["SensIndexStart"]	F	- an index of the first sensitivity parameter within the currently calculated subset of first order sensitivity parameters - in the case of second order sensitivity the index to derivative that corresponds to diagonal element $\{\phi_{\text{SensRowStart}},\phi_{\text{SensRowStart}}\}$ at row data$$["SensRowStart"]
idata$$["SensIndexEnd"]	F	-an index of the last sensitivity parameter within the currently calculated subset of first order sensitivity parameters - in the case of second order sensitivity the index to derivative that correcponds diagonal element at row data$$["SensRowEnd"]
idata$$["SensRowStart"]	F	an index of the first row in the currently calculated band of matrix of second order derivatives
idata$$["SensRowEnd"]	F	an index of the last row in the currently calculated band of matrix of second order derivatives
idata$$["SensMaxGroupLength"]	F	maximum length of the subset of sensitivity derivatives
idata$$["NoFirstOrderDerivatives"]	F	number of first order derivatives ($\equiv$ idata$$["NoSensParameters"])
idata$$["NoSecondOrderDerivatives"]	F	number of second order derivatives ($\equiv n_\phi\,(n_\phi+1)/2$, symmetric matrix!)
idata$$["NoThirdOrderDerivatives"]	F	number of third order derivatives
idata$$["SensIndex"]	F	index of the currently processed sensitivity derivative for post- processing and debugging
idata$$["NoDesignVelocityFields"]	F	total number of design velocity fields (user defined fields + Identity + Zero)
idata$$["NoBCFields"]	F	number of nonzero bounday conditions related design velocity fields (time dependent velocity fields)
idata$$["NoSensProblems"]	F/B	total number of solved sensitivity problems
idata$$["NoResponseFunctionals"]	B	number of response functionals (relevant for backward mode)
idata$$["NoBackwardSteps"]	B	number of time steps for backward sensitivity
idata$$["BackwardStepIndex"]	B	current time step for backward sensitivity (counts backward, from idata$$["NoBackwardSteps"] to 1!)
idata$$["SymmetricSensHessian"]	B	1 $\Rightarrow$ only upper triangle of Hessian has to be formed by the user subroutine (see Backward Mode Implementation Notes), command SMTBackwardSensitivity returns symmetrized Hessian 0 $\Rightarrow$ compete Hessian have to be formed by user subroutine

The relevance of the specific data depends on the type of sensitivity analysis (forward or backward). See also `SensitivityAnalysis`.

Contact related data

Default form	Default	description
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 Variables

Default form	Default	description
rdata$$["Multiplier"]	0	current values of the natural and essential boundary conditions are obtained by multiplying initial values with the rdata$$["Multiplier"] *(*the value is also known as load level or load factor*)*
rdata$$["ResidualError"]	10^{55}	Modified Euklids norm of the residual vector $\sqrt{\frac{\mathbf{R}.\mathbf{R}}{\text{NoEquations}}}$
rdata$$["IncrementError"]	10^{55}	Modified Euklids norm of the last increment of global d.o.f $\sqrt{\frac{\Delta\mathbf{a}.\Delta\mathbf{a}}{\text{NoEquations}}}$
rdata$$["MFlops"]		estimate of the number of floating point operations per second
rdata$$["SubMFlops"]		number of equivalent floating point operations for the last call of the user subroutine
rdata$$["Time"]	0	real time
rdata$$["TimeIncrement"]	0	value of the current real time increment
rdata$$["MultiplierIncrement"]	0	value of the current boundary conditions multiplier increment
rdata$$["PreviousMultiplierIncrement"]	0	value of the previous boundary conditions multiplier increment (used to calculate initial values of the unknowns at the beginning of the increment)
rdata$$["SubIterationTolerance"]	10^{-9}	tolerance for the local sub-iterative process
rdata$$["LineSearchStepLength"]	Automatic	step size control factor η $({}_{i+1}\mathbf{a}^t = {}_i\mathbf{a}^t + \eta\,\Delta\,{}_i\mathbf{a})$ (see also idata$$["LineSearchUpdate"])
rdata$$["PostMaxValue"]	0	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)
rdata$$["PostMinValue"]	0	the value is set by the post-processing SMTPostData function to the true minimum value of the required quantity
rdata$$["Solver1"] rdata$$["Solver2"] rdata$$["Solver3"] rdata$$["Solver4"] rdata$$["Solver5"]		solver specific parameters
rdata$$["Contact1"] rdata$$["Contact2"] rdata$$["Contact3"] rdata$$["Contact4"] rdata$$["Contact5"]		contact problem specific parameters
rdata$$["ContactSearchTolerance"]	0.001*rdata$$["XYZRange"]	tolerance for the rough contact search procedure

rdata$$["MinX"] rdata$$["MinY"] rdata$$["MinZ"] rdata$$["MaxX"] rdata$$["MaxY"] rdata$$["MaxZ"]		min and max mesh coordinates (mesh bounding box)
rdata$$["XYZRange"]		maximal dimension of the mesh bounding box
rdata$$["MinElementSize"]		minimum of the maximal dimension of the element bounding box for all elements
rdata$$["KAndRTime"]		time to evaluate global tangent matrix and residual
rdata$$["SolverTime"]		time to solve global system of linear equations
rdata$$["SpatialTolerance"]	10^{-9}	tolerance for various spatial search procedures (Tie, find nodes, find elements, etc.)
rdata$$["Parameter"]	0	general parameter used to parameterized the problem (see `Iterative Solution Procedures`, `Arc-length examples`, `SMTNextStep`)
rdata$$["ParameterIncrement"]	0	value of the current increment of the general parameter
rdata$$["ArcLengthPsi1"]	1	scaling for natural BC norm
rdata$$["ArcLengthPsi1"]	1	scaling for essential BC norm
rdata$$["ArcLengthDirectionCheck"]	10^{-8}	check for false direction if rdata$$["IncrementError"]< rdata$$["ArcLengthDirectionCheck"]
rdata$$["ArcLengthDirectionTolerance"]	0.01	direction is false when cos ϕ < rdata$$["ArcLengthDirectionTolerance"]-1
rdata$$["Solver6"]		
rdata$$["SensRTime"]		time to evaluate sensitivity pseudo-load vectors and resolve locally coupled sensistivity problems
rdata$$["SensSolTime"]		time to solve right-hand sides of sensitvity problem

Real type environment data.

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	*Description*	*Dimension*
ns$$[i,"id","SpecIndex"]	global index of the *i* -th node specification data structure	1
ns$$[i,"id","NoDOF"]	number of nodal d.o.f (≡ nd$$[i,"id","NoDOF"])	1
ns$$[i,"id", "NoNodeStorage"]	total number of history dependent real type values per node that have to be stored in the memory for transient type of problems	1
ns$$[i,"id", "NoNodeData"]	total number of arbitrary real values per node	1
ns$$[i,"id","NoData"]	total number of arbitrary real values per node specification	1
ns$$[i,"id", "NoTmpData"]	number of temporary real type variables stored during the execution of a single analysis directive (max (SMTIData["MinNoTmpData"],NoDOF))	1
ns$$[i,"id", "Constrained"]	1 ⇒ node has initially all d.o.f. constrained	1
ns$$[i,"id","Fictive"]	1 ⇒ The node does not represent a topological point. The switch is set automatically for the nodes with the -D and -P node identification switch.	1
ns$$[i,"id","Dummy"]	1 ⇒ node specification describes a dummy node	1
ns$$[i,"id", "DummyNode"]	index of the dummy node	1
ns$$[i,"Data", j]	arbitrary node specification specific data	NoData real numbers
ns$$[i,"NodeID"]	node identification (see Node Identification)	string
ns$$["id","Active"]	1 ⇒ nodes with specific node identification are active 0 ⇒ elements with specific node identification are ignored for all actions	integer
ns$$[i,"DOFScaling", j]	DOFScaling factors	NoDOF real numbers
ns$$[i, "SensBVFIndex", *j,k*]	is 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 "ADVF" node data field. It is used to construct boundary conditions related sensitvity data structures. **AceFEM**: *index* $(\partial p_j/\partial \phi_k)$ ≡ SMTNodeSpecData[*NodeID,*"SensBVFIndex"]⟦(k-1) NoDOF+j⟧ (see `AceFEM implementation` of `Sensitivity Analysis`)	NoDOF* NoSensDerivatives integer numbers

Node specification data structure.

Node Data

- see `Nodal Input data`, `SMTNodeData`

Default form	*Description*	*Dimension*
nd$$[i,"id", "NodeIndex"]	global index of the *i* -th node	1
nd$$[i,"id","NoDOF"]	number of nodal d.o.f	1
nd$$[i,"id","SpecIndex"]	index of the node specification data structure	1
nd$$[i,"id", "NoElements"]	number of elements associated with *i* -th node for calculation of tangent and residual (inactive nodes are excluded)	1
nd$$[i,"DOF", j]	global index of the *j* -th nodal d.o.f or < 0 if there is an essential boundary condition assigned to the *j*-th d.o.f. -1 constrained by input data -2 constrained by user after SMTAnalysis (for e.g. staggered schemes) -3 constrained automatically due to the deactivation of NodeSpec.id.Active - set by SMTSetSolver -4 constrained automatically due to no elements connects with node - set by SMTSetSolver	NoDOF
nd$$[i,"Elements",j]	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)	NoElements
nd$$[i,"X", j]⇒ SMSIO["Nodal coordinates"[i,j]]	initial coordinates of the node	3 (1-X,2-Y,3-Z)
nd$$[i,"at", j]⇒ SMSIO["Nodal DOFs"[i,j]]	current value of the *j* -th nodal d.o.f ($\mathbf{a}_i^t$)	NoDOF
nd$$[i,"ap", j]⇒ SMSIO["Nodal DOFs n"[i,j]]	value of the *j* -th nodal d.o.f at the end of previous step ($\mathbf{a}_i^p$)	NoDOF
nd$$[i,"da", j]	value of the increment of the *j* -th nodal d.o.f in last iteration ($\Delta\mathbf{a}_i$)	NoDOF
nd$$[i,"Bt", j]	nd$$[*i*,"DOF",*j*] ≡ -1 ⇒ current value of the *j* -th essential boundary condition nd$$[*i*,"DOF",*j*] ≥ 0 ⇒current value of the *j*-th natural boundary condition	NoDOF
nd$$[i,"Bp", j]	value of the *j* -th boundary condition (either essential or natural) at the end of previous step	NoDOF
nd$$[i,"dB",j]	reference value of the *j* -th boundary condition in node *i* (current boundary value is defined as $Bt = Bp + \Delta\lambda\ dB$, where $\Delta\lambda$ is the multiplier increment)	NoDOF
nd$$[i,"Data",j]⇒ SMSIO["Nodal data"[*i*, *j*]]	arbitrary node specific data + design sensitvity velocity fields+ adjoint vectors (user defined data length is internaly extended to accomodate sensitvity related data)	NoNodeData+ NoDesignVelocityFields +NoDOF* NoResponseFunctionals real numbers
nd$$[i,"ht",j]⇒ SMSIO["Nodal time dependent data"[*i*, *j*]]	current state of the *j* -th history dependent real type variable in the *i*-th node	NoNodeStorage real numbers
nd$$[i,"hp",j]⇒ SMSIO["Nodal time dependent data n"[*i*, *j*]]	the state of the *j* -th history dependent real type variable in the *i* -th node at the end of the previous step	NoNodeStorage real numbers
nd$$[i,"tmp", j]	temporary real type variables stored during the execution of a single analysis directive (restricted use)	Max[idata$$["MinNoTmpData"], NoDOF])

Nodal data structures.

Default form	*Description*	*Dimension*
nd$$[i,"st", j, k]	current *j*-th derivative of the *k*-th nodal d.o.f in *i -th* node **AceFEM**: $\frac{\partial \hat{p}_{i,k}}{\phi_j} \equiv$ SMTNodeData[i,"st"]⟦(j-1) NoDOF+k⟧ $\frac{\partial^2 \hat{p}_{i,k}}{\phi_j \phi_m} \equiv$ SMTNodeData[i,"st"]⟦(sensPos (j,m)-1) NoDOF+k⟧	NoDOF* NoSensDerivatives
nd$$[i,"sp", j, k]	*j*-th derivative of the *k*-th nodal d.o.f in *i -th* node in time *n*	NoDOF* NoSensDerivatives
nd$$[*i*, "ADVF", *j*]	the *j*-th nonzero velocity field in *i -th* node **AceFEM**: SMTNodeData[i,"ADVF"]⟦j⟧	NoDesignVelocityFields
nd$$[*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",*position (j,k)*]) **AceFEM**: no ekvivalent (interpreted, not an actual data)	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 $\frac{\partial F_j}{\partial p_{i\,k}}$. Data is stored into nodal Data field (nd$$[node,"Data"]) and has length NoDOF*NoResponseFunctional and is stored for n-th node as folows $\{\frac{\partial F_1}{\partial p_{n,1}}, \frac{\partial F_1}{\partial p_{n,2}}, \ldots \frac{\partial F_2}{\partial p_{n,1}}, \frac{\partial F_2}{\partial p_{n,2}}, \ldots \frac{\partial F_{nf}}{\partial p_{n,NoDOF}}\}$	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

Default form	Description	Type
es$$["Code"]	element code according to the general classification	string
es$$["id","SpecIndex"]	global index of the domain specification structure	integer
es$$["id", "NoDimensions"]	number of spatial dimensions (1/2/3)	integer
es$$["Topology"]	element topology code (see Template Constants)	string
es$$["MainTitle"]	description of the element	string
es$$["SubTitle"]	description of the element	string
es$$["SubSubTitle"]	detailed description of the element	string
es$$["Bibliography"]	reference	string
es$$["id", "SymmetricTangent"]	1 ⇒ element tangent matrix is symmetric 0 ⇒ element tangent matrix is unsymmetrical	integer
es$$["id", "PostIterationCall"]	force an additional call of the SKR user subroutines after the convergence of the global solution is achieved	False
es$$["id", "NoDOFGlobal"]	total number of global d.o.f per element	integer
es$$["DOFGlobal", *i*]	number of d.o.f for the *i* -th node (each node can have different number of d.o.f)	NoNodes× integer
es$$["id", "NoDOFCondense"]	number of d.o.f that have to be statically condensed before the element quantities are assembled to global quantities (see also Template Constants)	integer
es$$["user", *i*]	the *i* -th user defined element subroutines (interpretation depends on the FE environment)	link
es$$["id","Active"]	1 ⇒ elements with specific domain are active 0 ⇒ elements with specific domain are ignored for all actions	integer
es$$["Version"]	{AceGen version number, AceFEM version number, Mathemtica version number} used to create the element	Real list
es$$["id", "WorkingVectorSize"]	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.	Integer

General Domain Specification Data

Template Constant	*AceGen external variable*	*AceFEM data*
"SMSTopology"->"Key"	es$$["Topology"]	SMTDomainData[*dID*,"Topology"]
"SMSNoDimensions"->*NoDimensions*	es$$["id","NoDimensions"]	SMTDomainData[*dID*,"NoDimensions"]
"SMSNoNodes"->*NoNodes*	es$$["id","NoNodes"] ed$$["Nodes",i]	SMTDomainData[*dID*,"NoNodes"] SMTElementData[*element*,"Nodes"]
"SMSDOFGlobal"->{N,…}	es$$["DOFGlobal",*i*] nd$$[*element_node*,"id","NoDOF"] es$$["id","NoDOFGlobal"] es$$["id","MaxNoDOFNode"] es$$["id","NoAllDOF"]	SMTDomainData[*dID*,"DOFGlobal"][[i]] SMTNodeData[*global_node*,"NoDOF"] SMTDomainData[*dID*,"NoDOFGlobal"] SMTDomainData[*dID*,"MaxNoDOFNode"] SMTDomainData[*dID*,"NoAllDOF"]
"SMSNoDOFCondense"->*NoDOFCondense*	es$$["id","NoDOFCondense"]	SMTDomainData[*dID*,"NoDOFCondense"]
"SMSNodeID"->{"Key" …}	es$$["NodeID",*i*]	SMTDomainData[*dID*,"NodeID"][[i]]
"SMSMainTitle"->"ab"	es$$["MainTitle"]	SMTDomainData[*dID*,"MainTitle"]
"SMSSubTitle"->"ab"	es$$["SubTitle"]	SMTDomainData[*dID*,"SubTitle"]
"SMSSubSubTitle"->"ab"	es$$["SubSubTitle"]	SMTDomainData[*dID*,"SubSubTitle"]
"SMSSymmetricTangent"->*True-or-False*	es$$["id","SymmetricTangent"]	SMTDomainData[*dID*,"SymmetricTangent"]
"SMSPostIterationCall"->*True-or-False*	es$$["PostIterationCall"]	SMTDomainData["dID","PostIterationCall"]

Relations between the template constants, AceGen and AceFEM

Run Mathematica from code

es$$["MMAInitialisation"]	Mathematica' s code executed after SMTAnalysis command	string
es$$["MMANextStep"]	Mathematica' s code executed after SMTNextStep command	string
es$$["MMAStepBack"]	Mathematica' s code executed after SMTStepBack command	string
es$$["MMAPreIteration"]	Mathematica' s code executed before SMTNextStep command	string

Memory allocation

Default form	Description	Type
es$$["id", "NoTimeStorage"]	total length of the vector of history dependent variables per element (element level)	integer expression
es$$["id", "NoDomainData"]	number of input data values that are common for all elements in domain (e.g material constants) and are provided by the user is input data	integer
es$$["DomainDataNames", *i*]	description of the *i* -th input data value that is common for all elements with the same specification	NoDomainData×string
es$$["Data",j]⇒SMSIO["Domain data"[j]]	data common for all the elements within a particular domain (fixed length)	NoDomainData real numbers
es$$["id", "NoAdditionalData"]	number of additional input data values that are common for all elements in domain (e.g flow curve points) and are provided by the user is input data (variable length)	integer expression
es$$["AdditionalData",*i*]	additional data common for all the elements within a particular domain (variable length)	NoAdditionalData real numbers
es$$["id", "NoElementData"]	total length of vector of history independent variables per element (element level)	integer expression
es$$["id", "NoIData"]	number of additional integer type environment variables (global level)	integer
es$$["IDataNames",*i*]	name of the *i*-th additional integer type environment data variable (the corresponding value can be accessed by idata$$[*name*])	NoIData×string
es$$["IDataIndex",*i*]	position of the *i*-th additional integer type environment data variable on the idata$$ vector	NoIData×integer
es$$["id", "NoRData"]	number of additional real type environment variables (global level)	integer
es$$["RDataNames",*i*]	name of the *i*-th additional real type environment data variable (the corresponding value can be accessed by rdata$$[*name*])	NoRData×string
es$$["RDataIndex",*i*]	position of the *i*-th additional real type environment data variable on the rdata$$ vector	NoRData×integer
es$$["id", "NoCharSwitch"]	number of character type user defined constants (domain level)	0
es$$["CharSwitch",*i*]	*i*-th character type user defined constant	NoCharSwitch*word
es$$["id","NoIntSwitch"]	number of integer type user defined constants (domain level)	0
es$$["IntSwitch",*i*]	*i*-th integer type user defined constant	NoIntegerSwitch*integer
es$$["id", "NoDoubleSwitch"]	number of real type user defined constants (domain level)	0
es$$["DoubleSwitch",*i*]	*i*-th real type user defined constant	NoDoubleSwitch*doube
es$$["NoNodeStorage", *i*]	total length of the vector of history dependent real variables for the *i* -th node (node level)	NoNodes integer numbers
es$$["NoNodeData", *i*]	total length of the vector of arbitrary real values for the *i* -th node (node level)	NoNodes integer numbers

Template constant	*AceGen external variables*	*AceFEM data*
"SMSNoTimeStorage"->e*N*	es$$["id","NoTimeStorage"] ed$$["ht",*i*] ed$$["hp",*i*]	SMTDomainData[*dID*,"NoTimeStorage"] SMTElementData[e,"ht"][[i]]× SMTElementData[e,"hp"][[i]]
"SMSDomainDataNames"->{"Key" …}	es$$["id","NoDomainData"] es$$["DomainDataNames",*i*] es$$["Data",*i*]⇒SMSIO["Domain data"[*i*]]	SMTDomainData[*dID*,"NoDomainData"] SMTDomainData[*dID*, "DomainDataNames"][[i]] SMTDomainData[dID,"Data"][[i]]
"SMSNoAdditionalData"->*NoAdditionalData*	es$$["id","NoAdditionalData"] es$$["AdditionalData",*i*]	SMTDomainData[*dID*,"NoAdditionalData"] SMTDomainData[*dID*,"AdditionalData"][[i]]
"SMSNoElementData"->e*N*	es$$["id","NoElementData"] ed$$["Data",*i*]	SMTDomainData[*dID*,"NoElementData"] SMTElementData[e,"Data"][[i]]
"SMSNoNodeStorage"->e*N*	es$$["id","NoNodeStorage"] nd$$[en,"ht",*i*] nd$$[en,"hp",*i*]	SMTDomainData[*dID*,"NoElementData"] SMTNodeData[gn,"ht"][[i]] SMTNodeData[gn,"hp"][[i]]
"SMSNoNodeData"->e*N*	es$$["id","NoNodeData"] nd$$[en,"Data",*i*]	SMTDomainData[*dID*,"NoNodeData"] SMTNodeData[gn,"Data"][[i]]
"SMSIDataNames"->{"Key" …}	es$$["id", "NoIData"] es$$["IDataIndex",i] idata$$["K"]	SMTDomainData[*dID*,"NoIData"] SMTDomainData[*dID*,"IDataNames"] SMTIData["K"]
"SMSRDataNames"->{"Key" …}	es$$["id", "NoRData"] es$$["RDataIndex",i] rdata$$["K"]	SMTDomainData[*dID*,"NoRData"] SMTDomainData[*dID*,"RDataNames"] SMTRData["K"]
"SMSCharSwitch"->{"Key" …}	es$$["id","NoCharSwitch"] es$$["CharSwitch",*i*]	SMTDomainData[*dID*,"NoCharSwitch"] SMTDomainData[*dID*,"CharSwitch"][[i]]
"SMSIntSwitch"->{v_1,…}	es$$["id","NoIntSwitch"] es$$["IntSwitch",*i*]	SMTDomainData[*dID*,"NoIntSwitch"] SMTDomainData[*dID*,"IntSwitch"][[i]]
"SMSDoubleSwitch"->{v_1,…}	es$$["id","NoDoubleSwitch"] es$$["DoubleSwitch",*i*]	SMTDomainData[*dID*,"NoDoubleSwitch"] SMTDomainData[*dID*,"DoubleSwitch"][[i]]

Relations between the template constants, AceGen and AceFEM

Mesh generation

es$$["id","NoNodes"]	number of nodes per element	integer
es$$["NodeSpec",*i*]	node specification index for the *i* -th node	NoNodes integer numbers
es$$["NodeID",*i*]	integer number that is used for identification of the nodes in the case of multi-field problems for all nodes	NoNodes* integer numbers
es$$["AdditionalNodes"]	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)	pure function

Numerical integration

Default form	Description	Type
es$$["id","IntCode"]	integration code according to the general classification (see Numerical Integration)	integer
es$$["id", "DefaultIntegrationCode"]	default numerical integration code (Numerical Integration). Value is initialized by template constant SMSDefaultIntegrationCode (see Template Constants).	integer
es$$["id","NoIntPoints"]	total number of integration points for numerical integration (see Numerical Integration)	integer
es$$["id", "NoIntPointsA"]	number of integration points for first integration code (see Numerical Integration)	integer
es$$["id", "NoIntPointsB"]	number of integration points for second integration code (see Numerical Integration)	integer
es$$["id", "NoIntPointsC"]	number of integration points for third integration code (see Numerical Integration)	integer
es$$["IntPoints",*v*,*p*]	coordinates and weights of the numerical integration points ξ_p=es$$["IntPoints",1,*p*], η_p=es$$["IntPoints",2,*p*], ζ_p=es$$["IntPoints",3,*p*], w_p=es$$["IntPoints",4,*p*]	NoIntPoints*4 real numbers

Template Constant	*AceGen external variables*	*AceFEM data*
"SMSDefaultIntegration-Code"->n	es$$["id","DefaultIntegrationCode"] es$$["id","IntCode"] es$$["id","NoIntPoints"]⇒ SMSIO["No. integration points"] es$$["id","NoIntPointsA"] es$$["id","NoIntPointsB"] es$$["id","NoIntPointsC"] es$$["IntPoints",*v*,*p*]⇒ SMSIO["Integration point"[*p*]], SMSIO["Integration weight"[*p*]]	SMTDomainData["dID", "DefaultIntegrationCode"] SMTDomainData["dID","IntCode"] SMTDomainData["dID","NoIntPoints"] SMTDomainData["dID","NoIntPointsA"] SMTDomainData["dID","NoIntPointsB"] SMTDomainData[*dID*,"NoIntPointsC"] SMTDomainData[*dID*,"IntPoints"][[4 (*p*-1) + *v*]]

Relations between the template constants, AceGen and AceFEM

Graphics post-processing

es$$["id", "NoGPostData"]	number of post-processing quantities per material point (see Standard User Subroutines)	integer
es$$["id", "NoNPostData"]	number of post-processing quantities per node (see Standard User Subroutines)	integer
es$$["GPostNames", *i*]	description of the *i* -th post-processing quantities evaluated at each material point (see Standard User Subroutines)	NoGPostData×string
es$$["NPostNames", *i*]	description of the *i* -th post-processing quantities evaluated at each nodal point (see Standard User Subroutines)	NoNPostData× string
es$$["Segments", *i*]	sequence of element node indices that defines the segments on the surface or outline of the element (e.g. for *"Q1"* topology {1,2,3,4,0})	NoSegmentPoints ×integer
es$$["id", "NoSegmentPoints"]	the length of the *es$$["Segments"]* field	integer
es$$["ReferenceNodes",*i*]	coordinates of the nodes in a reference coordinate system (reference coordinate system is specified by the integration code)	NoNodes*3 real numbers
es$$["PostNodeWeights",*i*]	see `SMTPostData`	NoNodes real numbers
es$$["AdditionalGraphics"]	pure function (see Function) that is called for each element and returns additional graphics primitives per element (see Template Constants)	string

Template Constant	*AceGen external variables*	*AceFEM data*
"SMSGPostNames"->{"Key" …}	es$$["id","NoGPostData"]×es$$["GPostNames",*i*]	SMTDomainData[*dID*,"NoGPostData"] SMTDomainData[*dID*,"GPostNames"][[i]]
"SMSNPostNames"->{"Key" …}	es$$["id","NoNPostData"]×es$$["NPostNames",*i*]	SMTDomainData[*dID*,"NoNPostData"] SMTDomainData[*dID*,"NPostNames"][[i]]
"SMSSegments"->{N…}	es$$["id","NoSegmentPoints"] es$$["Segments", *i*]	SMTDomainData[*dID*,"NoSegmentPoints"] SMTDomainData[*dID*,"Segments"][[i]]
"SMSReferenceNodes"->{N…}	es$$["ReferenceNodes",i]	SMTDomainData[*dID*,"ReferenceNodes"][[i]]
"SMSPostNodeWeights"->{N…}	es$$["PostNodeWeights",i]	SMTDomainData[*dID*,"PostNodeWeights"][[i]]
"SMSAdditionalGraphics"-> Function[{...},...]	es$$["AdditionalGraphics"]	SMTDomainData[*dID*,"AdditionalGraphics"]

Relations between the template constants, AceGen and AceFEM

Sensitivity analysis

Default form	Description	Type
es$$["id", "NoSensNames"]	number of element input data fields (except the nodal unknowns!)	integer
es$$["SensitivityNames",*i*]	set of element input data fields identifiers (e.g. "X", "Y","u",..., except the nodal unknowns!)	NoSensNames*string
es$$["SensPVFIndex", *i*]	is a vector of indeces that point to the "ADVF" node data field where the element general design velocity fields are stored $\{\frac{d\psi_{e,1}}{d\phi_1}, \frac{d\psi_{e,2}}{d\phi_1}, \ldots, \frac{d\psi_{e,NoSensNames}}{d\phi_1}, \ldots, \frac{d^2\psi_{e,NoSensNames}}{d^2\phi_{ns}}\}$ Data is used to interpret the "SDVF" keyword. **AceFEM**: *index* $(\partial\psi_{e,k}/\partial\phi_j)$ ≡ SMTIDomainData[*dID*,"SensPVFIndex"][[(j-1) NoSensNames+k]] *index* $(\partial^2\psi_{e,k}/\partial\phi_i\,\partial\phi_j)$ ≡ SMTIDomainData[*dID*,"SensPVFIndex"][[(sensPos (i,j)-1) NoSensNames+k]]	NoSensDerivatives* NoSensNames integer numbers
es$$["SensLowerOrderIndex", *l*]	for all higher order sensitivity derivatives indeces of the sensitvity parameters (or 0 if the lower order derivatives does not exist) {…, sensPos (ϕ_i^n), sensPos (ϕ_j^n), sensPos (ϕ_k^n), sensPos (ϕ_i^n,ϕ_j^n), sensPos (ϕ_i^n,ϕ_k^n), sensPos (ϕ_j^n,ϕ_k^n), ... n=1,…,NoSensDerivatives}	6*NoSensDerivatives integer numbers
es$$["SensIndexi",*n*]	≡ sensPos (ϕ_i^n) (taken from es$$["SensLowerOrderIndex"] vector)	integer
es$$["SensIndexj",*n*]	≡ sensPos (ϕ_j^n) (taken from es$$["SensLowerOrderIndex"] vector)	integer
es$$["id", "ShapeSensitivity"]	1 ⇒ shape sensitivity is supported 0 ⇒ shape sensitivity is not supported	integer
es$$["id", "EBCSensitivity"]	1 ⇒ essential boundary condition sensitivity is supported 0 ⇒ essential boundary condition sensitivity is not supported	integer
es$$["id", "SensitivityOrder"]	Order of sensitivity analysis that is supported by the element: 0 ⇒ sensitivity analysis is not supported 1 ⇒ first order sensitivity analysis is supported 2 ⇒ second order sensitivity analysis is supported	integer

es$$["ExtraSensitivityData", *i*]	{
	1- length of backward sensitivity element
	time storage (BSETS data field attached to the ed$$["Data])
	}
		integer list

For more details see `AceFEM implementation of Sensitivity Analysis`.

Element Data

- see `Element Input data`, `SMTElementData`

Default form	*Description*	*Type*
ed$$["id","ElemIndex"]	global index of the element	integer
ed$$["id","SpecIndex"]	index of the domain specification data structure	integer
ed$$["id","Active"]	>0 active element for assembly 100 - visualization, original 200 - no visualization, automatic hiden surface, original 300 - no visualization by user, potential surface, original 400 - no visualization by user, inner element, original X+1 - added elements after SMTAnalysis X+10 - elements inactive for SMTPost <0 inactive element for assembly -100 - visualization, original -200 - no visualization, automatic hiden surface, original -300 - no visualization by user, potential surface, original -400 - no visualization by user, inner element, original X-1 - added elements after SMTAnalysis X-10 - elements inactive for SMTPost -1000 - deleted from memory	integer
ed$$["Nodes",j]	index of the *j* -th element nodes	NoNodes integer numbers
ed$$["Data",j]⇒SMSIO["Element data"[*j*]]	arbitrary element specific data	NoElementData real numbers
ed$$["ht",j]⇒SMSIO["Element time dependent data"[*j*]]	current state of the *j*-th history dependent real type variable	NoTimeStorage real numbers
ed$$["hp",j]⇒SMSIO["Element time dependent data n"[*j*]]	the state of the *j* -th history dependent real type variable at the end of the previous step	NoTimeStorage real numbers
ed$$["BSETS",j]	backward sensitivity element time storage (attached at the and of ed$$["Data",_])	interprated data SMSExtraSensitivityDat-a[[1]] real numbers

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[ ]=
```
{2, 4, {{1, 2, 3, 4}}, 2, Quadrilateral}
```

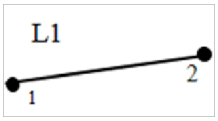
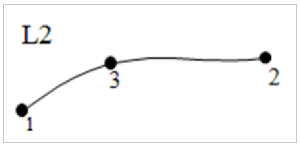
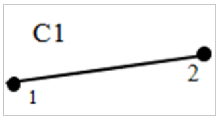
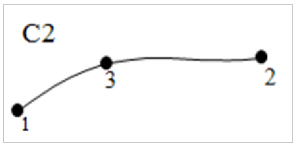
Undefined

Code	Description	Node numbering
"XX"	user defined topology	arbitrary

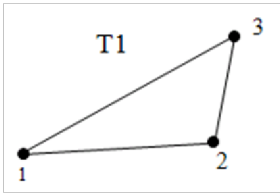
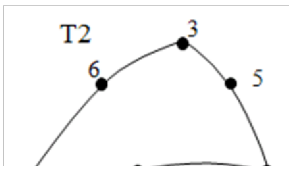
Zero dimensional

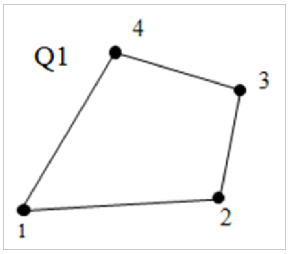
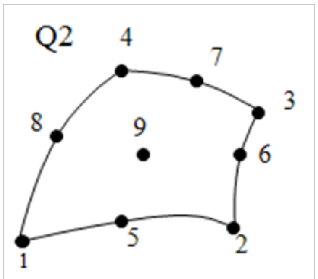
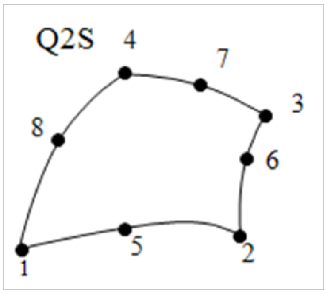
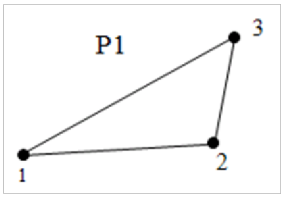
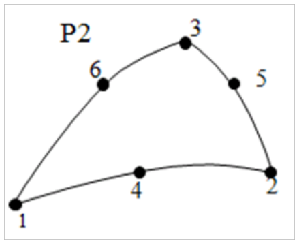
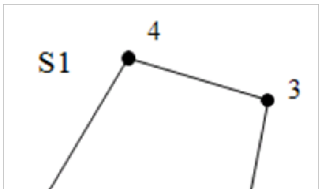
Code	*Description*	*Node numbering*
"V1"	point in 1 D	1
"V2"	point in 2 D	1
"V3"	point in 3 D	1

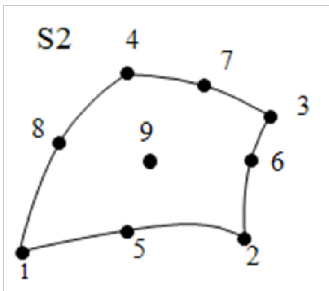
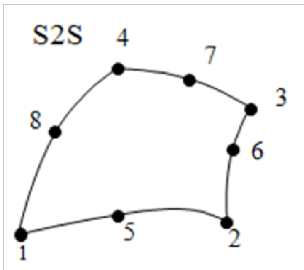
One dimensional

Code	*Description*	*Node numbering*
"D1"	element with 2 nodes in 1 D	1-2
"D2"	element with 3 nodes in 1 D	1-2-3
"DX"	element with arbitrary number of nodes in 1 D	arbitrary
"L1"	straight line segment in 2D bounded by two nodes	
"L2"	curve in 2 D with three nodes	
"LX"	curve with arbitrary number of nodes and arbitrary numbering in 2 D	arbitrary
"C1"	straight line segment in 3D bounded by two nodes	
"C2"	curve with 3 nodes in 3 D	
"CX"	curve with arbitrary number of nodes and arbitrary numbering in 3 D	arbitrary

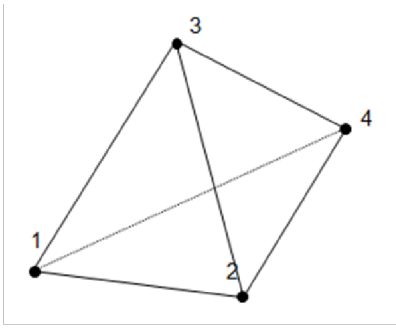
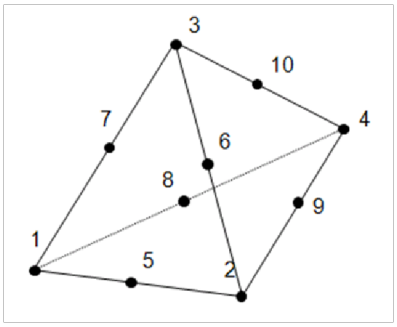
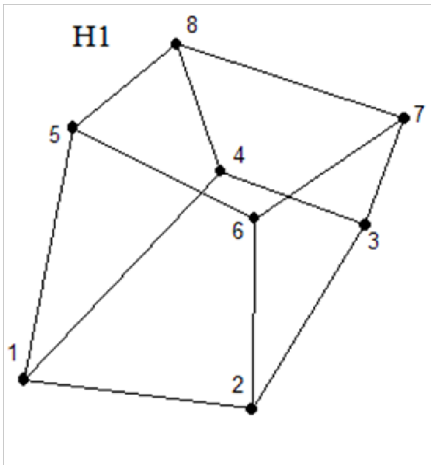
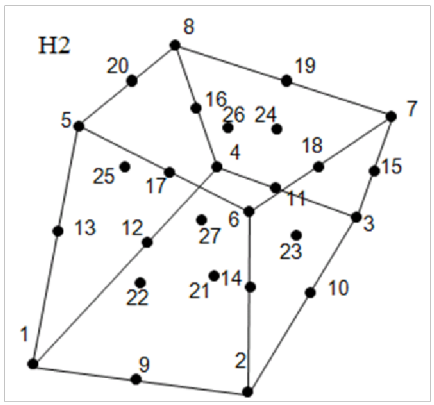
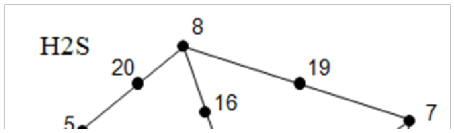
Two dimensional

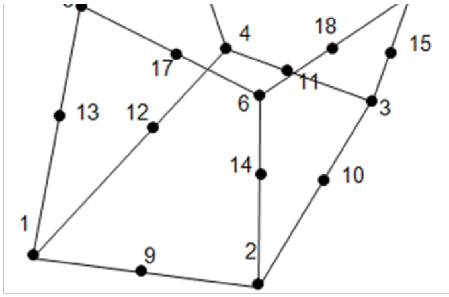
Code	*Description*	*Node numbering*
"T1"	triangle with 3 nodes in 2 D (numerical integration rules and post-processing routines assume "AREA CCORDINATES" of the reference element!)	
"T2"	triangle with 6 nodes in 2 D (numerical integration rules and post-processing routines assume "AREA CCORDINATES" of the reference element!)	

"TX"	triangle with arbitrary number of nodes and arbitrary numbering in 2 D	arbitrary
"Q1"	quadrilateral with 4 nodes in 2 D	
"Q2"	quadrilateral with 9 nodes in 2 D	
"Q2S"	quadrilateral with 8 nodes (serendipity family) in 2 D	
"QX"	quadrilateral with arbitrary number of nodes and arbitrary numbering in 2 D	arbitrary
"P1"	triangle with 3 nodes in 3 D	
"P2"	triangle with 6 nodes in 3 D	
"PX"	triangle with arbitrary number of nodes and arbitrary numbering in 3 D	arbitrary
"S1"	quadrilateral with 4 nodes in 3 D	

"S2"	quadrilateral with 9 nodes in 3 D	
"S2S"	quadrilateral with 8 nodes (serendipity family) in 3 D	
"SX"	quadrilateral with arbitrary number of nodes and arbitrary numbering in 3 D	arbitrary

Three dimensional

Code	*Description*	*Node numbering*
"O1"	tetrahedron with 4 nodes (numerical integration rules and post-processing routines assume "AREA CCORDINATES" of the reference element!)	
"O2"	tetrahedron with 10 nodes (numerical integration rules and post-processing routines assume "AREA CCORDINATES" of the reference element!)	
"OX"	tetrahedron with arbitrary number of nodes and arbitrary numbering	arbitrary
"H1"	hexahedron with 8 nodes	
"H2"	hexahedron with 27 nodes	
"H2S"	hexahedron with 20 nodes (serendipity family)	

"HX"	hexahedron with arbitrary number of nodes and arbitrary numbering	 arbitrary

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
–LP	The node with the switch –LP is a **local auxiliary node**. The –LP switch implies switches –P –S –F –T.
–GP	The node with the switch –GP is a **global auxiliary node**. The –GP switch implies switches –P –S –F –E.
–D	The node with switch –D is a standard **dummy** node. The "–D" switch implies switches –C –F –S. ("*name* –D" is identical to "*name$$*").
–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.

Basic node identifications switches.

Switch	description
–P	The node with the switch –P is auxiliary node. The –P switch implies switches –S –F.
–C	The unknowns associated with the nodes with the switch –C are initially constrained (by default all the unknowns are initially unconstrained).
–T	A node with the switch –T is ignored by the "Tie" command (see also `SMTAnalysis`.)
–S	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`).
–E	An unknown variables associated with the node are placed at the end of the list of unknowns.
–L	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).
–AL	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).
–F	All nodes with the switch –F are ignored by the SMTShowMesh["Marks"→"NodeNumber"] command, but they can be used to define the edge of the elements (see SMSSegments).

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

SMSIsDOFConstrained[*inode, idof, trueValue, falseValue*]
returns *trueValue* if *idof*-s DOF in *inode*-s element node is globally constrained (supported) and *falseValue* otherwise

SMSNoDummyNodes[]
returns the number of dummy element nodes (at AceFEM run time!)

SMSLastTrueNode[]
returns an index of the last element node that is not dummy node (at AceFEM run time!)

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!)

SMSIsDummyNode[*inode*]
returns True if *inode*-s element node is a dummy node and False otherwise (at AceFEM run time!)

SMSIsRealNode[*inode*]
returns True if *inode*-s element node is not 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, 2 D slave node – line master segment element,`.

Node Identification Examples

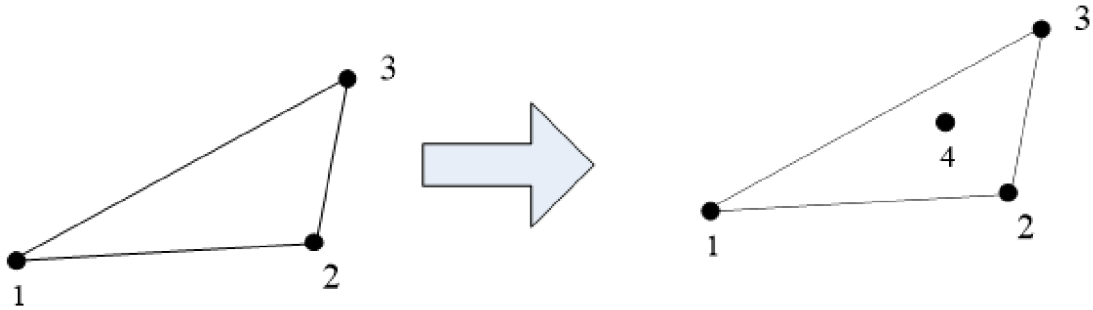
Examples: Non-standard elements

- 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.

```
SMSTemplate["SMSTopology" → "V3", "SMSNodeID" → "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.

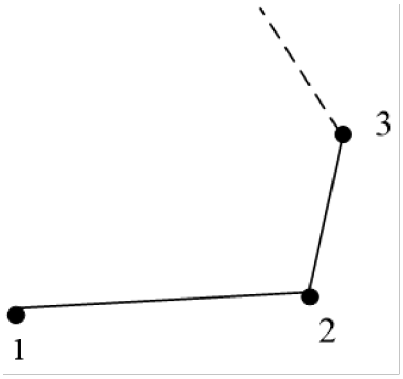



```
SMSTemplate["SMSTopology" → "T1",
  "SMSNoNodes" → 4,
  "SMSNodeID" → {"A", "A", "A", "A"},
  "SMSAdditionalNodes" → Function[{x1, x2, x3}, {(x1 + x2 + x3) / 3}]];
```

- 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.

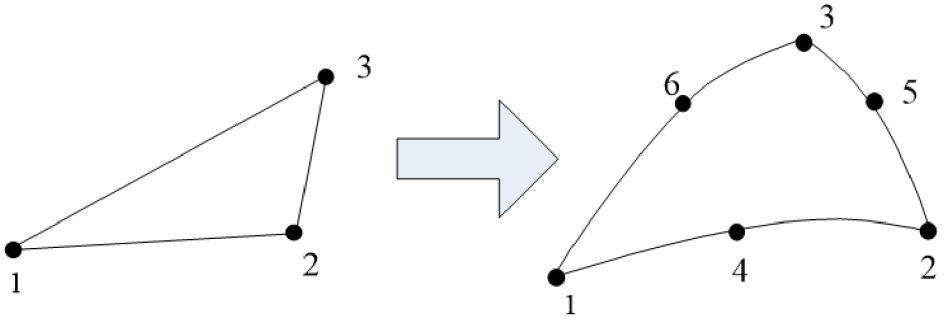
```
SMSTemplate["SMSTopology" → "T1",
  "SMSNoNodes" → 4,
  "SMSNodeID" → {"A", "A", "A", "B -LP"}, "SMSAdditionalNodes" → Function[{}, {Null}], ...];
```

- 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.




```
maxNodes = 15;
SMSTemplate["SMSTopology" → "T1",
  "SMSNoNodes" → maxNodes,
  "SMSNodeID" → Table["D -D", {maxNodes}], "SMSAdditionalNodes" → Null];
```

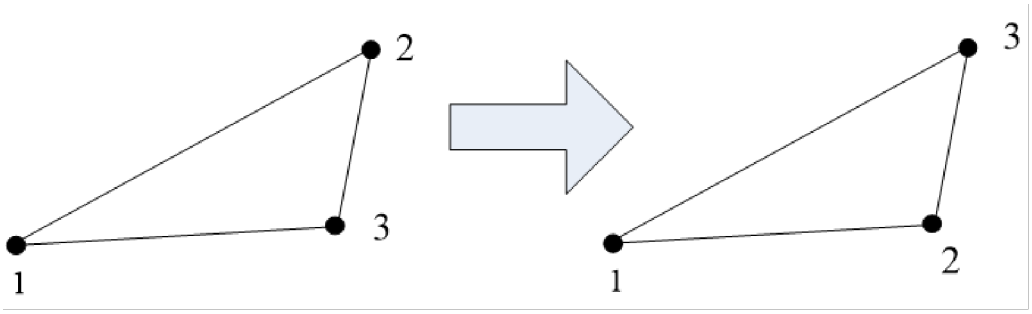
- This changes constant triangle into quadratic one.




```
SMSTemplate["SMSTopology" → "T1",
  "SMSNoNodes" → 6,
  "SMSNodeID" → {"D", "D", "D", "D", "D", "D"},
  "SMSAdditionalNodes" → Function[{x1, x2, x3}, {(x1 + x2) / 2, (x2 + x3) / 2, (x1 + x3) / 2}]];
```

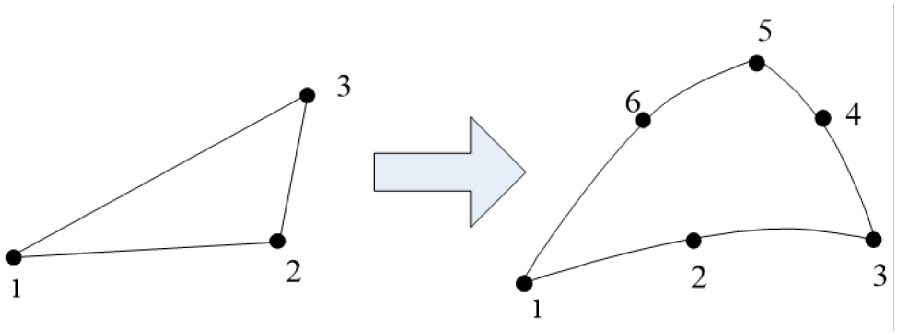
Examples: An arbitrary transformation of the list of nodes

- This changes the clockwise numeration of the nodes of triangle element into counter-clockwise numeration.



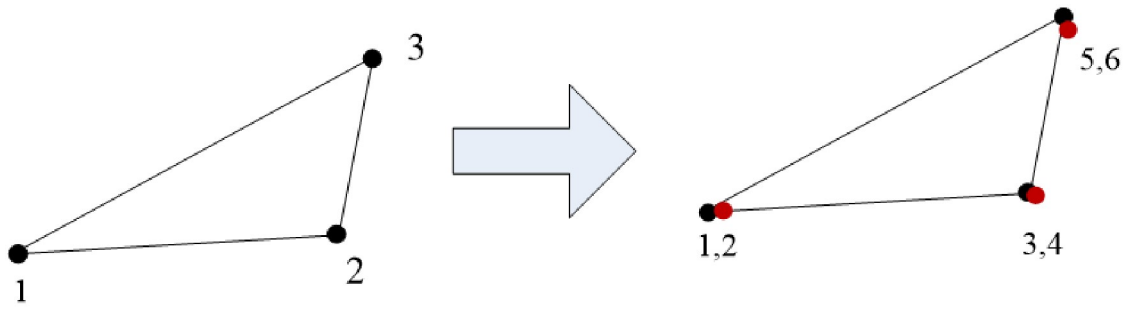

```
SMSTemplate["SMSTopology" → "T1",
  "SMSNoNodes" → 3,
  "SMSNodeID" → {"D", "D", "D"},
  "SMSAdditionalNodes" → Function[{x1, x2, x3},
    If[Cross[{x2[[1]] - x1[[1]], x2[[2]] - x1[[2]], 0}, {x3[[1]] - x1[[1]], x3[[2]] - x1[[2]], 0}][[3]] > 0
     , {1, 2, 3}
     , {1, 3, 2}]
   ]];
```

- This changes constant triangle into quadratic one with an alternative node ordering.



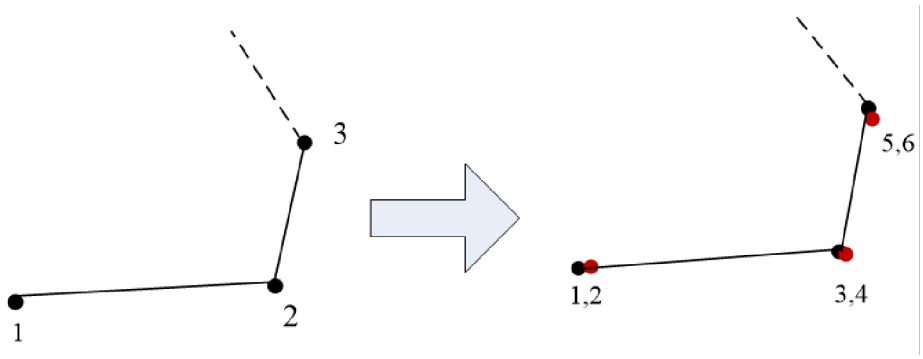

```
SMSTemplate["SMSTopology" → "T1",
  "SMSNoNodes" → 6,
  "SMSNodeID" → {"D", "D", "D", "D", "D", "D"},
  "SMSAdditionalNodes" → Function[{x1, x2, x3}, {1, (x1 + x2) / 2, 2, (x2 + x3) / 2, 3, (x1 + x3) / 2}]];
```

- This doubles the nodes of the triangle.




```
SMSTemplate["SMSTopology" → "T1",
  "SMSNoNodes" → 6,
  "SMSNodeID" → {"D", "T", "D", "T", "D", "T"},
  "SMSAdditionalNodes" → Function[{x1, x2, x3}, {1, x1, 2, x2, 3, x3}]];
```

- 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.




```
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

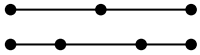
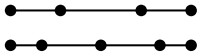
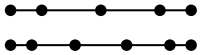
- One dimensional
- Quadrilateral
- Triangle
- Tetrahedra
- Hexahedra
- Implementation of Numerical Integration
  - Example 1 : SMSIO Input / Output data management
  - Example 2 : hard coded integration rules
  - Example 3 : low – level definition of IO data

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.

Code	*Description*	*No. of points*
0	numerical integration is not used	0
-1	default integration code is taken accordingly to the topology of the element	topology dependent
>0	integration code is taken accordingly to the given code	

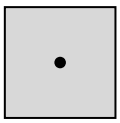
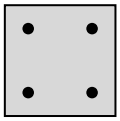
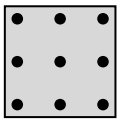
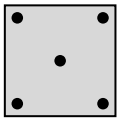
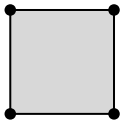
One dimensional

Cartesian coordinates of the reference element: $\{\zeta, \eta, \zeta\} \in [-1,1] \times [0,0] \times [0,0]$

Code	*Description*	*No. of points*	*Disposition*
20	1 point Gauss	1	
21	2 point Gauss	2	
22	3 point Gauss	3	
23	4 point Gauss	4	
24	5 point Gauss	5	
25	6 point Gauss	6	
26	7 point Gauss	7	
27	8 point Gauss	8	
28	9 point Gauss	9	
29	10 point Gauss	10	
30	2 point Lobatto	2	
31	3 point Lobatto	3	
32	4 point Lobatto	4	
33	5 point Lobatto	5	
34	6 point Lobatto	6	

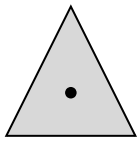
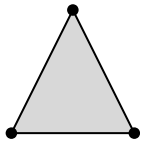
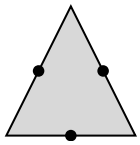
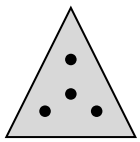
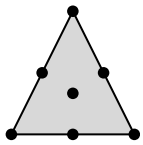
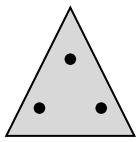
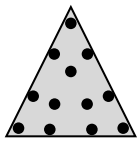
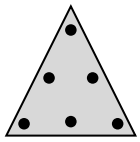
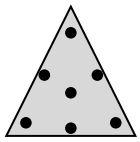
Quadrilateral

Cartesian coordinates of the reference element: $\{\zeta, \eta, \zeta\} \in [-1,1] \times [-1,1] \times [0,0]$

Code	*Description*	*No. of points*	*Disposition*
1	1 point integration	1	
2	2×2 Gauss integration	4	
3	3×3 Gauss integration	9	
4	5 point special rule	5	
5	points in nodes	4	
{19+N,19+N}	N×N Gauss integration (N≤10)	N^2	
{29+N,29+N}	N×N Lobatto integration (2<N≤6)	N^2	

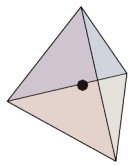
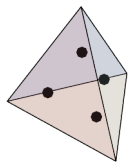
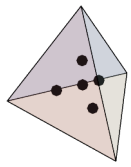
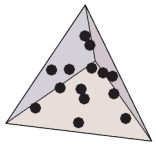
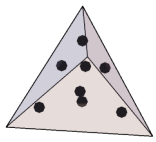
Triangle

AREA coordinates of the reference element: $\{\zeta, \eta, \zeta\} \in [0,1] \times [0,1] \times [0,0]$

Code	*Description*	*No. of points*	*Disposition*
12	1 point integration	1	
13	3 point integration	3	
14	3 point integration	3	
16	4 point integration (negative weight at the centre!)	4	
17	7 point integration	7	
35	3 point integration	3	
39	12 point integration	12	
41	6 point integration	6	
42	7 point integration	7	

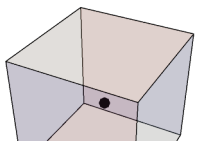
Tetrahedra

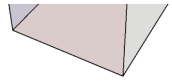
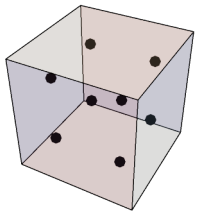
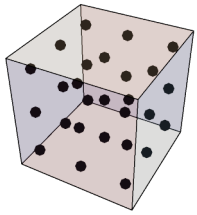
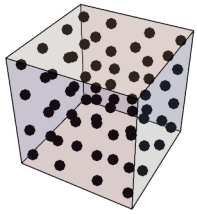
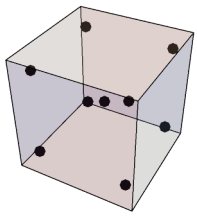
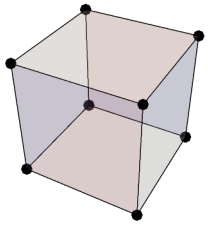
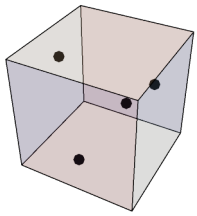
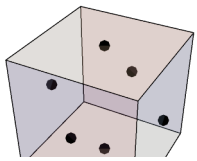
AREA coordinates of the reference element: $\{\zeta, \eta, \zeta\} \in [0,1] \times [0,1] \times [0,1]$

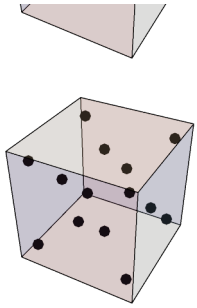
Code	*Description*	*No. of points*	*Disposition*
15	1 point integration	1	
18	4 point integration	4	
19	5 point integration (negative weight at the centre!)	5	
40	14 point integration	14	
43	8 point integration	8	

Hexahedra

Cartesian coordinates of the reference element: $\{\zeta, \eta, \zeta\} \in [-1,1] \times [-1,1] \times [-1,1]$

Code	*Description*	*No. of points*	*Disposition*
6	1 point integration	1	

				
7	2×2×2 Gauss integration	8		
8	3×3×3 Gauss integration	27		
9	4×4×4 Gauss integration	64		
10	9 point special rule	9		
11	points in nodes	8		
{19+N, 19+N,19+N}	N×N×N Gauss integration (N≤10)	N^3		
{29+N, 29+N,29+N}	N×N×N Lobatto integration (2<N≤6)	N^3		
36	4 point special rule	4		
37	6 point special rule	6		

38	14 point special rule	14	

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 a following way

```
ξi ← SMSReal[es$$["IntPoints",1,i]]
ηi ← SMSReal[es$$["IntPoints",2,i]]
ζi ← SMSReal[es$$["IntPoints",3,i]]
wi ← SMSReal[es$$["IntPoints",4,i]]
```

where $\{\xi_i\ \eta_i, \zeta_i\}$ 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 multi-dimensional integration rules. The combined code is given in the domain specification input data as a list of up to three basic codes as follows:

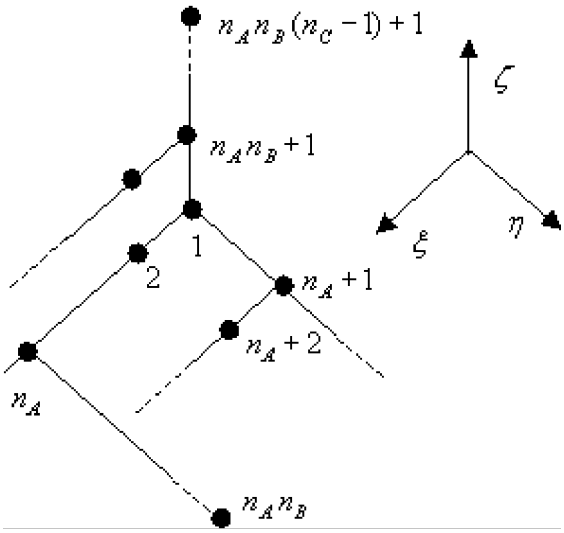
{*codeA*} ≡ *codeA*

{*codeA*,*codeB*}

{*codeA*,*codeB*,*codeC*}

where *codeA*, *codeB* and *codeC* are any of the basic integration codes. For example 2×2×5 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 point (2×2×2) 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: SMSIO Input/Output data management

- This generates simple loop over all given integration points for 3D integration. Proper data interface is created by `IO Data Management`. Integration points and weights are calculated at run time.

```
SMSDo[Ig, 1, SMSIO["No. integration points"]];
    ⌧ = {ξ, η, ζ} ⊨ SMSIO["Integration point"[Ig]];
    wgp ⊨ SMSIO["Integration weight"[Ig]];
    ...
SMSEndDo[];
```

- This generates simple loop over all given integration points for 2D integration.

```
SMSDo[Ig, 1, SMSIO["No. integration points"]];
    {ξ, η} ⊨ SMSIO["Integration point"[Ig]]⟦{1, 2}⟧;
    wgp ⊨ SMSIO["Integration weight"[Ig]];
    ...
SMSEndDo[];
```

- This generates simple loop over all given integration points for 1D integration.

```
SMSDo[Ig, 1, SMSIO["No. integration points"]];
    ξ ⊨ SMSIO["Integration point"[Ig]]⟦1⟧;
    wgp ⊨ SMSIO["Integration weight"[Ig]];
    ...
SMSEndDo[];
```

Example 2: hard coded integration rules

- Integration points and weights can be calculated in advance and hard coded into element code as well. This is useful if one needs several different integration rules within the same finite element.
- This generates simple loop over all given integration points for 3×3×3 Gauss integration (code 27).

```
SMSDo[Ig, 1, SMSIO["No. integration points"[27]]];
    ⌧ = {ξ, η, ζ} ⊨ SMSIO["Integration point"[Ig][27]];
    wgp ⊨ SMSIO["Integration weight"[Ig][27]];
    ...
SMSEndDo[];
```

- This generates simple loop over all given integration points for 10×2×2 Gauss integration (code {29,21,21}).

```
SMSDo[Ig, 1, SMSIO["No. integration points"[{29, 21, 21}]]];
    Ξ = {ξ, η, ζ} ⊨ SMSIO["Integration point"[Ig][{29, 21, 21}]];
    wgp ⊨ SMSIO["Integration weight"[Ig][{29, 21, 21}]];
    ...
SMSEndDo[];
```

Example 3: low-level definition of IO data

- In the case of the combined integration code, the integration can be also performed separately for each set of points. low-level definition of IO data has to be used in this case (see `IO Data Management`).

```
{nA, nB, nC} ⊢ SMSInteger[
  {es$$["id", "NoIntPointsA"], es$$["id", "NoIntPointsB"], es$$["id", "NoIntPointsC"]}]
SMSDo[iξ, 1, nA];
 ξ ⊢ SMSReal[es$$["IntPoints", 1, iξ]];
 ...
 SMSDo[iη, 1, nB];
  η ⊢ SMSReal[es$$["IntPoints", 2, (iη - 1) nA + 1]];
  ...
  SMSDo[iζ, 1, nC];
   ζ ⊢ SMSReal[es$$["IntPoints", 3, (iζ - 1) nA nB + 1]];
   w ⊢ SMSReal[es$$["IntPoints", 4, iξ + (iη - 1) nA + (iζ - 1) nA nB]];
   ...
  SMSEndDo[];  ...
    SMSEndDo[];
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:

$$\begin{pmatrix} \mathbf{K}^{\mathbf{n}}_{\mathrm{uu}} & \mathbf{K}^{\mathbf{n}}_{\mathrm{uh}} \\ \mathbf{K}^{\mathbf{n}}_{\mathbf{h}\mathrm{u}} & \mathbf{K}^{\mathbf{n}}_{\mathrm{hh}} \end{pmatrix} \cdot \begin{pmatrix} \Delta \mathbf{u}^{\mathrm{n}} \\ \Delta \mathbf{h}^{\mathrm{n}} \end{pmatrix} = \begin{pmatrix} -\mathbf{R}^{\mathbf{n}}_{\mathbf{u}} \\ -\mathbf{R}^{\mathbf{n}}_{\mathbf{h}} \end{pmatrix} \implies \mathbf{K}_{\mathrm{cond}}\, \Delta \mathbf{u}^{\mathrm{n}} = -\mathbf{R}_{\mathrm{cond}}$$

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:

$$\mathbf{K}_{\mathrm{cond}} = \mathbf{K}^{\mathbf{n}}_{\mathrm{uu}} - \mathbf{K}^{\mathbf{n}}_{\mathrm{uh}}\, \mathbf{H}^{\mathbf{n}}_{\mathbf{a}}$$
$$\mathbf{R}_{\mathrm{cond}} = \mathbf{R}^{\mathbf{n}}_{\mathbf{u}} + \mathbf{K}^{\mathbf{n}}_{\mathrm{uh}}\, \mathbf{H}^{\mathbf{n}}_{\mathbf{b}}$$

where $\mathbf{H}_{\mathrm{a}}$ is a matrix and $\mathbf{H}_{\mathrm{b}}$ a vector defined as

$$\mathbf{H}^{\mathbf{n}}_{\mathbf{a}} = \mathbf{K}^{\mathbf{n}}_{\mathrm{hh}}{}^{-1}\, \mathbf{K}^{\mathbf{n}}_{\mathbf{h}\mathrm{u}}\,.$$
$$\mathbf{H}^{\mathbf{n}}_{\mathbf{b}} = -\mathbf{K}^{\mathbf{n}}_{\mathrm{hh}}{}^{-1}\, \mathbf{R}^{\mathbf{n}}_{\mathbf{h}}$$

The actual values of the local unknowns are calculated first time when the element tangent and residual subroutine is called by:

$$\mathbf{h}^{\mathrm{n}+1} = \mathbf{h}^{\mathrm{n}} + \mathbf{H}_{\mathrm{b}} - \mathbf{H}_{\mathrm{a}}\, \Delta \mathbf{u}^{\mathbf{n}}\,.$$

Three quantities have to be stored at the element level for the presented scheme: the values of the local unknowns $\mathbf{h}^{\mathrm{n}}$, the $\mathbf{H}^{\mathrm{n}}_{\mathrm{b}}$ matrix and the $\mathbf{H}^{\mathrm{n}}_{\mathrm{a}}$ 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:

SMSNoDOFCondense+SMSNoDOFCondense+SMSNoDOFCondense*SMSNoDOFGlobal

It is assumed that the sensitivity of the local unknowns ($\delta \mathbf{h}^{\mathrm{n}}$) is stored as follows:

$$\left\{ \frac{\partial h_1}{\partial p_1}, \frac{\partial h_1}{\partial p_2}, \ldots, \frac{\partial h_1}{\partial p_{\mathrm{NoSensDerivatives}}}, \frac{\partial h_2}{\partial p_1}, \frac{\partial h_2}{\partial p_1}, \ldots, \frac{\partial h_{\mathrm{SMSNoDOFCondense}}}{\partial p_{\mathrm{NoSensDerivatives}}} \right\}$$

The template constant SMSCondensationData stores pointers to the beginning of the corresponding data field as follows:

Data	*Position*	Dimension
$\mathbf{h}^{\mathrm{n}}$	SMSCondensationData[[1]]	SMSNoDOFCondense
$\mathbf{H}^{\mathrm{n}}_{\mathrm{b}}$	SMSCondensationData[[2]]	SMSNoDOFCondense
$\mathbf{H}^{\mathrm{n}}_{\mathrm{a}}$	SMSCondensationData[[3]]	SMSNoDOFCondense* SMSNoDOFGlobal
$\delta \mathbf{h}^{\mathrm{n}}$	SMSCondensationData[[4]]	SMSNoDOFCondense* NoSensDerivatives

Storage scheme for the elimination of the local unknowns.

The the actual position is calculated accordingly to the data provided to the SMSTemplate command as follows:

option to SMSTemplate command	actual SMSCondensationData
"SMSNoDOFCondense"→*nhe*	{ed$$["ht",1], ed$$["ht",*nhe*+1], ed$$["ht",2 *nhe*+1], ed$$["ht",2 *nhe*+*nhe*∗ SMSNoDOFGlobal]}
"SMSNoDOFCondense"→*nhe* "SMSCondensationData"→*h_position*	{*h_position*, ed$$["ht",SMSNoTimeStorage+1], ed$$["ht",SMSNoTimeStorage+ *nhe*+1], ed$$["ht",SMSNoTimeStorage+ *nhe*+ *nhe*∗MSNoDOFGlobal+1]}
"SMSNoDOFCondense"→*nhe* "SMSCondensationData"→ {*h_position*,*δh_position*}	{*h_position*, ed$$["ht",SMSNoTimeStorage+1], ed$$["ht",SMSNoTimeStorage+ *nhe*+1], *δh_position*}
"SMSCondensationData"→{*h_position*, *Hb_position*,*Ha_position*,*δh_position*}	{*h_position*,*Hb_position*,*Ha_position*,*δh_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 .

```
SMSTemplate["SMSTopology" → "H1", "SMSNoDOFCondense" → 9]

SMSTemplate["SMSTopology" → "H1", "SMSNoDOFCondense" → 9,
 "SMSCondensationData" → ed$$["ht", 1], "SMSNoTimeStorage" → 9]

SMSTemplate["SMSTopology" → "H1", "SMSNoDOFCondense" → 9,
 "SMSCondensationData" → {ed$$["ht", 1], ed$$["ht", 10], ed$$["ht", 19], ed$$["ht", 235]},
 "SMSNoTimeStorage" → 234 + 9 idata$$["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 field that corresponds to the input is as follows:

```
SMSInitialize["test", "Environment" → "AceFEM"];
SMSTemplate["SMSTopology" → "Q1", "SMSNoTimeStorage" → le, "SMSNoDOFCondense" → nhe]

hi ⊢ SMSReal[Array[ed$$["ht", le + #] &, nhe]];
j ⊢ SMSInteger[idata$$["SensIndex"]];
δhi ⊢ SMSReal[Array[ed$$["ht", le + 2 nhe + nhe * SMSNoDOFGlobal + (j - 1) * nhe + #] &, nhe]];
 ....
SMSWrite[]
```

Data	*Position of i-th element*	position
$\mathbf{h}^{\mathrm{n}}$	$\mathbf{h}^{\mathrm{n}}[i]$	ed$$["ht",*le*+i]
$\mathbf{H}^{\mathrm{n}}_{\mathrm{b}}$	$\mathbf{H}^{\mathrm{n}}_{\mathrm{b}}[i]$	ed$$["ht",*le*+*nhe*+i]
$\mathbf{H}^{\mathrm{n}}_{\mathrm{a}}$	$\mathbf{H}^{\mathrm{n}}_{\mathrm{a}}[i]$	ed$$["ht",*le*+2 *nhe*+*i*]
$\delta\mathbf{h}^{\mathrm{n}}$	$\delta\mathbf{h}^{\mathrm{n}}[i]$ for j-th sensitivity parameter	ed$$["ht",*le*+2 *nhe*+*nhe** SMSNoDOFGlobal+ (*j*-1)**nhe*+*i*]

See also: `Mixed 3 D 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.

codes for the user defined subroutines	default subroutine name	description
"Tangent and residual"	"SKR"	See: `Subroutine: Tangent and residual` Standard subroutine that returns the tangent matrix and residual for the current values of nodal and element data.
"Postprocessing"	"SPP"	See: `Subroutine: Postprocessing` Standard subroutine that returns post-processing quantities.
"Element initialisation"	"ELI"	User controled additional initialisation of data (e.g. "ht", "hp", etc..) after the SMTAnalysis. Subroutine is called once for each element where it is defined.
"Forward mode first order pseudo-load"	"SSE"	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.
"Forward mode dependent sensitivity"	"SHI"	See: Dependent sensitivity user subroutine Standard subroutine that resolves forward mode sensitivities of the dependent variables defined at the element level.
"Forward mode second order pseudo-load"	"SS2"	See: Sensitivity 2 nd order user subroutine Standard subroutine that returns the forward mode second order pseudo-load vector for the current subset of sensitivity parameters.
"Backward mode pseudo-load"	"SBL"	See: Backward sensitivity pseudo-load user subroutine Standard subroutine that returns the pseudo-load vector for backward mode sensitivity analysis.
"Backward mode first order derivatives"	"SBG"	See: Backward sensitivity derivatives user subroutine Standard subroutine that calculates element contribution to gradient of functional.
"Backward mode second order derivatives"	"SBH"	See: Backward sensitivity derivatives user subroutine Standard subroutine that calculates element contribution to Hessian of functional.
"Residual"	"SRE"	Standard subroutine that returns residual for the current values of the nodal and element data.
"Nodal information"	"PAN"	Standard subroutine that returns position of the nodes at current and previous time step (e.g. used for slave nodes of contact elements).
"Tasks"	"Tasks"	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 "Tasks" 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.
"User n"	"Usern"	*n*-th user defined system subroutine (low-level system feature intended to be used by advanced users)

Standard set of user subroutines.

option	description
"Name"→"name"	use a given name for the generated subroutine instead of default name (for the default names see table below)
"AdditionalArguments"→{*arg1,arg2,…*}	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`)

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 *SMSRe-*

al[nd$$[i,"X",1]] 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.

parameter	description
es$$[…]	element specification data structure (see Element Data)
ed$$[…]	element data structure (see `Element Data`)
ns$$[1,…], ns$$[2,…], …,ns$$[*SMSNoNodes*,…]	node specification data structure for all element nodes (see Node Data)
nd$$[1,…], nd$$[2,…], …,nd$$[*SMSNoNodes*,…]	nodal data structure for all element nodes (see Node Data)
idata$$	integer type environment variables (see `Integer Type Environment Variables`)
rdata$$	real type environment variables (see Real Type Environment Variables)

The standard set of input/output arguments passed to all user subroutines.

Some additional I/O arguments are needed for specific tasks as follows:

user subroutine	argument	description
"Tangent and residual"	p$$[*NoDOFGlobal*] s$$[*NoDOFGlobal*,*NoDOFGlobal*]	element residual vector element tangent matrix
"Postprocessing"	gpost$$[*NoIntPoints*,*NoGPostData*] npost$$[*NoNodes*,*NoNPostData*]	integration point post-processing quantities nodal point post-processing quantities
"Element initialisation"	-	-
"Tangent"	s$$[*NoDOFGlobal*,*NoDOFGlobal*]	element tangent matrix
"Residual"	p$$[*NoDOFGlobal*]	element residual vector
"Nodal information"	d$$[*problem dependent* , 6]	$\{\{x_1^t,y_1^t,z_1^t,x_1^p,y_1^p,z_1^p\},\{x_2^t,y_2^t...\},...\}$
"Tasks"	Task$$ TasksData$$[5] IntegerInput$$[TasksData$$[2]] RealInput$$[TasksData$$[3]] IntegerOutput$$[TasksData$$[4]] RealOutput$$[TasksData$$[5]]	see User Defined Tasks, Standard User Subroutines, `SMTTask`
"Nodal information"	d$$[MaxNoNodes,12]	position of the node at current and previous time step and normal vector for all nodes
"User n"	-	-

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.

user subroutine	AceFEM	*FEAP*	*ELFEN*	ABAQUS
"Tangent and residual"	●	●	●	●
"Postprocessing"	●	●		
"Element initialisation"	●			
sensitivity analysis related subroutines	●			
"Nodal information"	●			
"Tasks"	●			
"User n"	●			

- 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
[0] File created : test.c Size : 3570
```

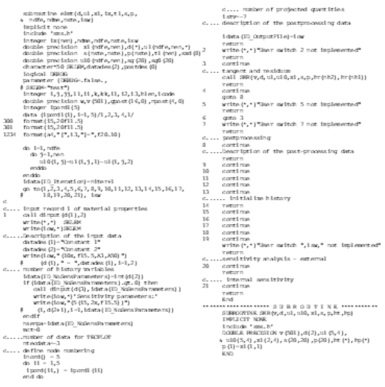
In[484]:=

```
FilePrint["test.c"]
```

In[235]:=

```
<< 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
[0] 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"×"number of integration point quantities",*

⇒ *npost$$* array of the nodal point quantities with the dimension *"number of nodes"×"number of nodal point quantities".*

The dimension and the contents of the arrays are defined by the two vectors of strings *SMSGPostNames* and *SMSNPostNames.* 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:

keyword	description
"DeformedMeshX"	(see "DeformedMesh"→True option of `SMTShowMesh` command)
"DeformedMeshY"	(see "DeformedMesh"→True option of `SMTShowMesh` command)
"DeformedMeshZ"	(see "DeformedMesh"→True option of `SMTShowMesh` command)

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.

```
(* template constants related to the post-processing*)
SMSTemplate[
 "SMSSegments" → ..., "SMSReferenceNodes" → ...,
 "SMSPostNodeWeights" → ..., "SMSAdditionalGraphics" → ...
]

SMSStandardModule["Postprocessing"];
(* nodal post-processing *)
{u, v} ⊨ Transpose[SMSIO["Nodal DOFs"]];
SMSIO[{"DeformedMeshX" → u, "DeformedMeshY" → v, "u" → u, "v" → v},
  "Export to", "Nodal point post"];
(* integration point post-processing *)
SMSDo[Ig, 1, SMSIO["No. integration points"]];
    ...
    SMSIO[{"Sxx" → σ⟦1, 1⟧, "Sxy" → σ⟦1, 2⟧, ...}, "Export to", "Integration point post"[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" → ..., "SMSReferenceNodes" → ...,
 "SMSPostNodeWeights" → ..., "SMSAdditionalGraphics" → ...
]

SMSStandardModule["Postprocessing"];

(* integration point post-processing *)
SMSGPostNames = {"Sxx", "Syy", "Sxy", ...};
SMSDo[Ig, 1, SMSInteger[es$$["id", "NoIntPoints"]]];
 (* export integration point post-processing values to Ig-th integration point*)
 SMSExport[{Sxx, Syy, Sxy, ...}, gpost$$[Ig, #1] &];
SMSEndDo[];

(* nodal post-processing *)
SMSNPostNames = {"DeformedMeshX", "DeformedMeshY", "DeformedMeshZ", "a", "b", ...};
(* Example: export nodal point post-processing values for all nodes*)
SMSExport[Table[{ui⟦i⟧, vi⟦i⟧, w⟦i⟧, a⟦i⟧, b⟦i⟧, ...}, {i, 1, SMSNoNodes}], npost$$];

(* Example: export nodal point post-processing values to i-th nodes*)
SMSDo[node, 1, SMSNoNodes]
uNode = ...; ...
SMSExport[{uNode, vNode, wNode, aNode, bNode, ...}, npost$$[node, #] &];
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$$["ExtrapolationType"]. The nodal value is additionally multiplied by the user defined nodal wight factor that is stored in element specification data structure for each node (es$$["PostNodeWeights",*node*-

number]). 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!!!

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.

In[502]:=
```
datarules = {nd$$[i_, "X", j_] :> X$$[i, j],
    nd$$[i_, "at", j_] :> U$$[i, j],
    es$$["Data", i_] :> D$$[i],
    s$$[i_, j_] :> K$$[i, j],
    p$$[i_] :> S$$[i],
    rdata$$["Multiplier"] → load$$};
```

- 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"}
  , "SMSUserDataRules" → datarules];
SMSStandardModule["Tangent and residual"];
SMSDo[Ig, 1, SMSInteger[es$$["id", "NoIntPoints"]]];
Ξ = {ξ, η, ζ} ⊢ Table[SMSReal[es$$["IntPoints", i, Ig]], {i, 3}];
XI ⊢ Table[SMSReal[nd$$[i, "X", j]], {i, SMSNoNodes}, {j, SMSNoDimensions}];
Ξn = {{-1, -1, -1}, {1, -1, -1}, {1, 1, -1}, {-1, 1, -1},
  {-1, -1, 1}, {1, -1, 1}, {1, 1, 1}, {-1, 1, 1}};
NI ⊨ Table[1/8 (1 + ξ Ξn[[i, 1]]) (1 + η Ξn[[i, 2]]) (1 + ζ Ξn[[i, 3]]), {i, 1, 8}];
X ⊢ SMSFreeze[NI.XI]; Jg ⊨ SMSD[X, Ξ]; Jgd ⊨ Det[Jg];
ϕI ⊢ SMSReal[Table[nd$$[i, "at", 1], {i, SMSNoNodes}]];
ϕ ⊨ NI.ϕI;
{k0, k1, k2, Q} ⊢ SMSReal[Table[es$$["Data", i], {i, Length[SMSDomainDataNames]}]];
k ⊨ k0 + k1 ϕ + k2 ϕ^2;
SMSSetBreak["k"];
λ ⊢ SMSReal[rdata$$["Multiplier"]];
wgp ⊢ SMSReal[es$$["IntPoints", 4, Ig]];
SMSDo[
  Dϕ ⊨ SMSD[ϕ, X, "Dependency" → {Ξ, X, SMSInverse[Jg]}];
  δϕ ⊨ SMSD[ϕ, ϕI, i];
  Dδϕ ⊨ SMSD[δϕ, X, "Dependency" → {Ξ, X, SMSInverse[Jg]}];
  Rg ⊨ Jgd wgp (k Dδϕ.Dϕ - δϕ λ Q);
  SMSExport[SMSResidualSign Rg, p$$[i], "AddIn" → True];
  SMSDo[
   Kg ⊨ SMSD[Rg, ϕI, j];
   SMSExport[Kg, s$$[i, j], "AddIn" → True];
   , {j, 1, 8}
  ];
  , {i, 1, 8}
 ];
SMSEndDo[];
SMSWrite[];
```

File: UserEnvironment.c Size: 6769		
Methods	No.Formulae	No.Leafs
RKt	132	2616

In[521]:=

```
Quit[];
```

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).

FEAP - ELFEN - ABAQUS - ANSYS

Contents

- Running the generated code with the target numerical environment
- SMSFEEnvironment
- ABAQUS
- FEAP
- ELFEN
- ANSYS

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 FileNameJoin[{$BaseDirectory,"Applications","AceGen","Include", *environment*}] directory.

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:

environment	script template files
FEAP	FEAPCompile.bat FEAPRun.bat
ABAQUS	ABAQUSCompile.bat ABAQUSRun.bat abaqus_v6.env
ELFEN	ELFENCompile.bat ELFENRun.bat
ANSYS	compileAUP.sh, SetAUP.sh

Script command files

The *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:

slotname	script files
SourceCodeFileName	the source code file name (without path and extension, e.g. Hooke2D)
IncludeDirectory	full path of the directory where the header file (sms.h) and utility file (SMSUtility.f) are located
UtilityLibraryFileName	full path and name of the utility library
SimulationInputFile	the name of the simulation input data file
SimulationOutputFile	the results of simulation are writen to files defined by the given file name
Debug	True/False

Available template slots.

In order to compile and link a new element or user subroutine with *environment* we need:

- header file *sms.h* (available at FileNameJoin[{$BaseDirectory,"Applications","AceGen","Include", *environment*}])
- utility library "libaceutilityWinIntelF.lib"/"libaceutilityMacF.lib"/"libaceutilityLinuxF.lib" (available at FileNameJoin[{$BaseDirectory,"Applications","AceGen","Include","Fortran"}])
- 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.

option	default	description
"SourceCode"->*fname*	None	compiles user subroutines specified by the file name or `Unified Element Code` using the *environmentCompile.bat* script template file.
"SimulationInputFile"->*fname*	None	runs FE *environment* with the *fname* as input data file using the *environmentRun.bat* script template file.
"SimulationTemplate"->*fname*	None	given template file is first processed and then written to the *"SimulationData"* file. The result is then used as input data file to perform simulation using the *environmentRun.bat* script template file.
"SimulationOutputFile"->*fname*	"tmp.txt"	the results of simulation are writen to files defined by the given file name
"CompileTemplate"->*fname*	Automatic	*environmentCompile.bat* script template file
"RunTemplate"->*fname*	Automatic	*environmentRun.bat* script template file
"Debug"->*truefalse*	False	pause before exiting the script files

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. Supplementary files are available at FileNameJoin[{$BaseDirectory,"Applications","AceGen","Include", "ABAQUS"}] directory. See also `Installation instructions` for instructions how to compile and link the generated codes.

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**.

Abbreviation	description
S$$[SMSNoTensorComponents]	components of Cauchy stress tensor ($\boldsymbol{\sigma}$) (output)
F$$[9]	current deformation gradient ($\boldsymbol{F}_{n+1}$)
Fn$$[9]	deformation gradient at time n ($\boldsymbol{F}_n$)
W$$	specific strain energy
En$$[SMSNoTensorComponents]	deformation tensor at time n
dE$$[SMSNoTensorComponents]	increment of deformation tensor
DSDE$$[SMSNoTensorComponents, SMSNoTensorComponents]	material constitutive matrix (output)
ed$$["ht",SMSNoTimeStorage]	history dependent real type values per material point
ed$$["hp",SMSNoTimeStorage]	history dependent real type values per material point at time n
es$$["Data", SMSDomainDataNames//Length]	vector of material constants (in input file USER MATERIAL, CONSTANTS=SMSDomainDataNames//Length)
TEMP$$	Temperature at the start of the increment
DTEMP$$	Increment of temperature
environment variables SMSIO["keyword"]	Keywords: "Time", "TimeIncrement", "CurrentElement", "SubIterationTolerance", "Iteration","TotalIteration"

IO parameters of the UMAT subroutine.

Abbreviation	Default value	description
SMSNoTensorComponents	Switch[SMSNoDimensions, 1,1,2,4,3,6]	the lenght of the vectorized stress/strain tensor

Additional ABAQUS UMAT template constants.

Stress tensor and constitutive matrix must be vectorized accordingly to ABAQUS manual. Stress tensor is vectorized as:

In[144]:=
```
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}, {1, 3}, {2, 3}}
]
Extract[σ, TensorComponents]
```

Constitutive tensor is vectorized by:

```
Table[Extract[DσDE, Join[i1, i2]], {i1, TensorComponents}, {i2, TensorComponents}]
```

Example of 3D hyper-elastic user material subroutine for ABAQUS

In[146]:=

```
<< "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[ ]:=

- deformation gradient: needs NLGEOM=YES set in ABAQUS input

In[151]:=

```
F ⊢ SMSReal[Array[F$$, {3, 3}]];
JF ⊨ Det[F];
SMSFreeze[be, F.Fᵀ, "Symmetric" → True];
Jbe = Det[be];
{μ, K} ⊨ SMSHookeToBulk[Em, ν];
W ⊨ μ/2 (Jbe^(-1/3) Tr[be] - 3) + K/β^2 (Jbe^(-β/2) - 1 + β/2 Log[Jbe]);
τ ⊨ Simplify[2 be.SMSD[W, be, "Ignore" → NumberQ, "Symmetric" → True]];
σ ⊨ τ / JF;
DτDF ⊨ SMSD[τ, F];
(*material constitutive matrix for isotropic materials*)
DσDE ⊨ Table[Sum[
      F[[l, m]] × DτDF[[i, j, k, m]] + F[[k, m]] × DτDF[[i, j, l, m]], {m, 3}],
     {l, 3}, {k, 3}, {j, 3}, {i, 3}] / JF / 2;
SMSExport[W, W$$];
```

- stress tensor and constitutive matrix is vectorized accordingly to ABAQUS manual

In[17]:=

```
TensorComponents = {{1, 1}, {2, 2}, {3, 3}, {1, 2}, {1, 3}, {2, 3}};
SMSExport[Extract[σ, TensorComponents], S$$];
SMSExport[
  Table[Extract[DσDE, Join[i1, i2]], {i1, TensorComponents}, {i2, TensorComponents}], DSDE$$];
```

In[20]:=

```
SMSWrite[];
```

File: test.for Size: 16111 Time: 7

Method	ConstitutiveEquations
No.Formulae	182
No.Leafs	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]:=

```
SMSFEEnvironment["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

In[1]:=

```
<< "AceGen`";
SMSInitialize["test", "Environment" → "ABAQUS"];
SMSTemplate["SMSTopology" → "H1", "SMSSymmetricTangent" → True
  , "SMSDomainDataNames" → {"E -elastic modulus", "ν -poisson ratio",
    "Qx -volume load X", "Qy -volume load Y", "Qz -volume load Z"}
  , "SMSDefaultData" → {21000, 0.3, 0, 0, 0}];
SMSStandardModule["Tangent and residual"];
SMSDo[Ig, 1, SMSInteger[es$$["id", "NoIntPoints"]]];
Ξ = {ξ, η, ζ} ⊢ Table[SMSReal[es$$["IntPoints", i, Ig]], {i, 3}];
XI ⊢ Table[SMSReal[nd$$[i, "X", j]], {i, SMSNoNodes}, {j, SMSNoDimensions}];
Ξn = {{-1, -1, -1}, {1, -1, -1}, {1, 1, -1}, {-1, 1, -1},
  {-1, -1, 1}, {1, -1, 1}, {1, 1, 1}, {-1, 1, 1}};
NI ⊨ Table[1/8 (1 + ξ Ξn⟦i, 1⟧) (1 + η Ξn⟦i, 2⟧) (1 + ζ Ξn⟦i, 3⟧), {i, 1, 8}];
X ⊢ SMSFreeze[NI.XI]; Jg ⊨ SMSD[X, Ξ]; Jgd ⊨ Det[Jg];
uI ⊢ SMSReal[Table[nd$$[i, "at", j], {i, SMSNoNodes}, {j, SMSNoDimensions}]];
pe = Flatten[uI]; u ⊨ NI.uI;
Dg ⊨ SMSD[u, X, "Dependency" → {Ξ, X, SMSInverse[Jg]}];
F ⊨ IdentityMatrix[3] + Dg;
JF ⊨ Det[F];
Cg ⊨ Transpose[F].F;
{Em, ν, Qx, Qy, Qz} ⊢ SMSReal[Table[es$$["Data", i], {i, Length[SMSDomainDataNames]}]];
{λ, μ} ⊨ SMSHookeToLame[Em, ν];
W ⊨ 1/2 λ (JF - 1)^2 + μ (1/2 (Tr[Cg] - 3) - Log[JF]) - {Qx, Qy, Qz}.u;
wgp ⊢ SMSReal[es$$["IntPoints", 4, Ig]];
SMSDo[Rg ⊨ Jgd wgp SMSD[W, pe, i];
  SMSExport[SMSResidualSign Rg, p$$[i], "AddIn" → True];
  SMSDo[Kg ⊨ SMSD[Rg, pe, j];
   SMSExport[Kg, s$$[i, j], "AddIn" → True];
   , {j, i, 24}];
  , {i, 1, 24}];
SMSEndDo[];
SMSWrite[];
```

File: test.for Size: 24900 Time: 5

Method	SKR
No.Formulae	180
No.Leafs	5296

- Here is the generated element compiled and linked into the ABAQUS. The SMSFEEnvironment compiles the source code file, links the resulting object file into ABAQUS and starts ABAQUS with a ABAQUSExample.dat file as a input file and tmp.res as output file. The ABAQUS input data file for the one element test example is available at the $BaseDirectory/Applications/AceGen/Include/ABAQUS directory.

In[348]:=

```
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"

In[ ]:=

- 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

```
call "C:\Program Files (x86)\IntelSWTools\compilers_and_libraries_2016.3.207\windows\bin\ifortvars.bat" intel64 vs2012
call ifort /c /DABQ_WIN86_64 /extend-source /fpp /iface:cref /recursive /Qauto-scalar /QxSSE3 /QaxAVX /heap-arrays:1 /Qdiag-file:SMSCompile.txt -o `SourceCodeFileName`.obj /I"`IncludeDirectory`" `SourceCodeFileName`.for

<*If[#Debug,"pause",""]*>
```

ABAQUSCompile.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

<*If[#Debug,"pause",""]*>
```

ABAQUSRun.bat file

```
compile_fmu=['cl', '/LD', '/D_WINDOWS', '/TC', '/W0',  '/I%I']

compile_fortran=['ifort',
        '/c','/DABQ_WIN86_64', '/extend-source', '/fpp',
        '/iface:cref', '/recursive', '/Qauto-scalar',
        '/QxSSE3', '/QaxAVX',
        '/heap-arrays:1',
        # '/Od', '/Ob0',  # <-- Optimization Debugging
        # '/Zi',       # <-- Debugging
        '/I"<*StringReplace[#IncludeDirectory,"\\"->"/"]*>"'
        ,'/include:%I']

link_sl=['LINK',
    '/nologo', '/NOENTRY', '/INCREMENTAL:NO', '/subsystem:console', '/machine:AMD64',
    '/NODEFAULTLIB:LIBC.LIB', '/NODEFAULTLIB:LIBCMT.LIB',
    '/DEFAULTLIB:OLDNAMES.LIB', '/DEFAULTLIB:LIBIFCOREMD.LIB', '/DEFAULTLIB:LIBIFPORTMD.LIB',
'/DEFAULTLIB:LIBMMD.LIB',
    '/DEFAULTLIB:kernel32.lib', '/DEFAULTLIB:user32.lib', '/DEFAULTLIB:advapi32.lib',
    '/FIXED:NO', '/dll',
    #'/debug', # <-- Debugging
    '/def:%E', '/out:%U', '%F', '%A', '%L', '%B',
    'oldnames.lib', 'user32.lib', 'ws2_32.lib', 'netapi32.lib', 'advapi32.lib',
    '<*StringReplace[#UtilityLibraryFileName,"\\"->"/"]*>']
```

abaqus_v6.env 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. Supplementary files are available at FileNameJoin[{$BaseDirectory,"Applications","AceGen","Include", "FEAP"}] directory. See also `Installation instructions` for instructions how to compile and link the generated codes.

Specific FEAP Interface Data

Additional template constants (see Template Constants) have to be specified in order to process the *FEAP's* "splice-file" correctly.

Abbreviation	default	description
FEAP$ElementNumber	"10"	element user subroutine number (*elmt ??*)

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.

In[141]:=

```
<< "AceGen`";
SMSInitialize["test", "Environment" → "FEAP"];
SMSTemplate["SMSTopology" → "H1", "SMSSymmetricTangent" → True, "SMSNoDOFCondense" → 9
  , "SMSDomainDataNames" → {"E -elastic modulus", "ν -poisson ratio",
    "Qx -volume load X", "Qy -volume load Y", "Qz -volume load Z"}
  , "SMSDefaultData" → {21000, 0.3, 0, 0, 0}];
SMSStandardModule["Tangent and residual"];
SMSDo[Ig, 1, SMSInteger[es$$["id", "NoIntPoints"]]];
Ξ = {ξ, η, ζ} ⊢ Table[SMSReal[es$$["IntPoints", i, Ig]], {i, 3}];
XI ⊢ Table[SMSReal[nd$$[i, "X", j]], {i, SMSNoNodes}, {j, SMSNoDimensions}];
Ξn = {{-1, -1, -1}, {1, -1, -1}, {1, 1, -1}, {-1, 1, -1},
  {-1, -1, 1}, {1, -1, 1}, {1, 1, 1}, {-1, 1, 1}};
NI ⊨ Table[1/8 (1 + ξ Ξn[[i, 1]]) (1 + η Ξn[[i, 2]]) (1 + ζ Ξn[[i, 3]]), {i, 1, 8}];
X ⊢ SMSFreeze[NI.XI]; Jg ⊨ SMSD[X, Ξ]; Jgd ⊨ Det[Jg];
uI ⊢ SMSReal[Table[nd$$[i, "at", j], {i, SMSNoNodes}, {j, SMSNoDimensions}]];
pe = Flatten[uI]; u ⊨ NI.uI;
Dg ⊨ SMSD[u, X, "Dependency" → {Ξ, X, SMSInverse[Jg]}];
J0 ⊨ SMSReplaceAll[Jg, {ξ → 0, η → 0, ζ → 0}]; J0d ⊨ Det[J0];
αe ⊢ Table[SMSReal[ed$$["ht", i]], {i, SMSNoDOFCondense}];
ph = Join[pe, αe];
Hbξ = ( ξ αe[[1]] | η αe[[2]] | ζ αe[[3]]
        ξ αe[[4]] | η αe[[5]] | ζ αe[[6]]
        ξ αe[[7]] | η αe[[8]] | ζ αe[[9]] ); Hb ⊨ J0d/Jgd Hbξ.SMSInverse[J0];
F ⊨ IdentityMatrix[3] + Dg + Hb;
JF ⊨ Det[F];
Cg ⊨ Transpose[F].F;
{Em, ν, Qx, Qy, Qz} ⊢ SMSReal[Table[es$$["Data", i], {i, Length[SMSDomainDataNames]}]];
{λ, μ} ⊨ SMSHookeToLame[Em, ν];
W ⊨ 1/2 λ (JF - 1)^2 + μ (1/2 (Tr[Cg] - 3) - Log[JF]) - {Qx, Qy, Qz}.u;
wgp ⊢ SMSReal[es$$["IntPoints", 4, Ig]];
SMSDo[Rg ⊨ Jgd wgp SMSD[W, ph, i];
  SMSExport[SMSResidualSign Rg, p$$[i], "AddIn" → True];
  SMSDo[Kg ⊨ SMSD[Rg, ph, j];
   SMSExport[Kg, s$$[i, j], "AddIn" → True];
   , {j, i, SMSNoAllDOF}];
  , {i, 1, SMSNoAllDOF}];
SMSEndDo[];
SMSWrite[];
```

```
Elimination of local unknowns requires additional memory. Corresponding constants are set to:

SMSCondensationData= {ed$$[ht, 1], ed$$[ht, 10], ed$$[ht, 19], ed$$[ht, 235]}
SMSNoTimeStorage= 234 + 9 idata$$[NoSensDerivatives]  See also: Elimination of local unknowns
```

File: test.f **Size:** 29000 **Time:** 7

Method	SKR10
No.Formulae	269
No.Leafs	7004

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.

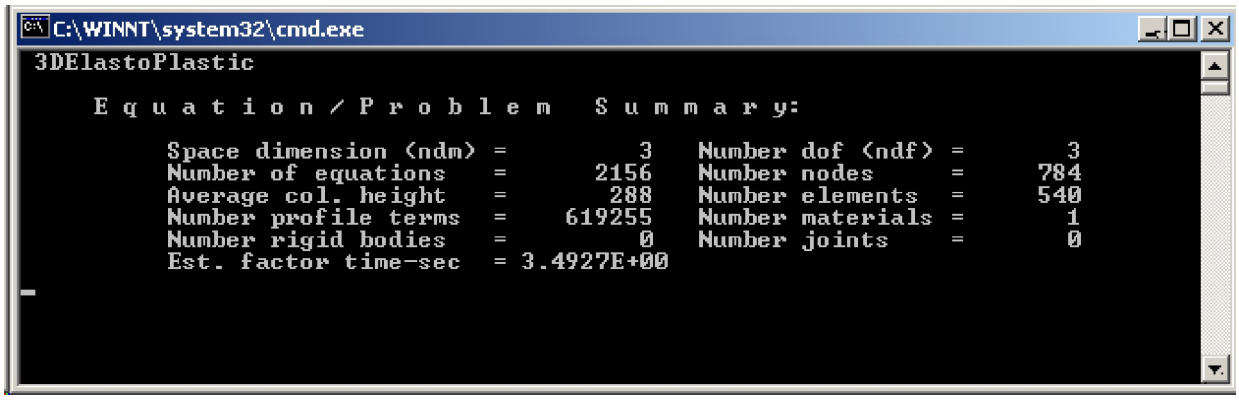
```
feap
0,0,0,3,3,8
```

```
block
cart,6,15,6,1,1,1,10
1,10.,0.,0.
2,10.,2.,0.
3,0.,2.,0.
4,0.,0.,0.
5,10.,0.,2.
6,10.,2.,2.
7,0.,2.,3.
8,0.,0.,3.

ebou
1,0,1,1,1
1,10.,,,1

edisp,add
1,10.,,,-1.

mate,1
user,10
1000,0.3

end

macr
tol,,1e-9
prop,,1
dt,,1
loop,,5
time
loop,,10
tang,,1
next
disp,,340
next
end

stop
```

- 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**.

Supplementary files are available at FileNameJoin[{$BaseDirectory,"Applications","AceGen","Include", "ELFEN"}] directory. See also `Installation instructions` for instructions how to compile and link the generated codes.

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.

Abbreviation	Default value	description
ELFEN$ElementModel	"L1","LX"⇒"B2" "C1","CX"⇒"B3" "T1","T2","TX", "Q1","Q2", "QX"⇒"PE" "P1","P2","PX","S1", "S2","SX"⇒"SH" "O1","O2", "OX","H1","H2", "HX"⇒"D3"	"B2" ⇒ two dimensional beam elements "B3" ⇒ three dimensional beam elements "PS " ⇒ two dimensional plane stress elements "PE " ⇒ two dimensional plane strain elements "D3" ⇒ three dimensional solid elements "AX" ⇒ axi-symmetric elements "PL" ⇒ plate elements "ME" ⇒ membrane elements "SH" ⇒ shell elements
ELFEN$NoState	0	number of state variables
SMSNoStressComponents	Switch[SMSNoDimensions, 1,1,2,4,3,6]	the lenght of the vectorized stress tensor
SMSNoStrainComponents	Switch[SMSNoDimensions, 1,1,2,4,3,6]	the lenght of the vectorized strain tensor

Additional ELFEN constants.

- Here the additional constants for the 2D, plane strain element are defined.

Parameter	type	description
mswitch	integer *mswitch*	dimensions of the integer switch data array
switch	integer *switch (mswitch)*	integer type switches
meuvbl	integer *meuvbl*	dimensions of the element variables vlues array
lesvbl	integer *lesvbl (meuvbl)*	array of the element variables vlues
nehist	integer *nehist*	number of element dependent history variables
jfile	integer *jfile*	output file (FORTRAN unit number)
morder	integer *m order*	dimension of the node ordering array
order	integer *orde (morder)*	node ordering
mgdata	integer *mgdata*	dimension of the element group data array
gdata	character*32 *gdata (mgdata)*	description of the element group specific input data values
ngdata	integer *ngdata*	number of the element group specific input data values
mstate	integer *mstate*	dimension of the state data array
state	character* 32 *state (mstate)*	description of the element state data values
nstate	integer *nstate*	number of the element state data values
mgpost	integer *mgpost*	dimension of the integration point post-processing data array
gpost	character*32 *gpost (mgpost)*	description of the integration point post-processing values
ngpost	integer *ngpost*	total number of the integration point post-processing values
ngspost	integer *ngspost*	number of sensitivity parameter dependent integration point post-processing values
mnpost	integer *mgpost*	dimension of the integration point post-processing data array
npost	character*32 *npost (mnpost)*	description of the integration point post-processing values
nnpost	integer *nnpost*	total number of the integration point post-processing values
nnspost	integer *nnspost*	number of sensitivity parameter dependent integration point post-processing values

Parameter list for the SMSI*nnn* ELFEN *nnnn*'th user element subroutine.

Switch	type	description
1	output	number of gauss points
2	input	number of sensitivity parameters

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

idata$$["SkipTangent"] = 0,

idata$$["SkipResidual"] = 1,

idata$$["SubIterationMode"] = 1

and when the residual is required the variables are set to

idata$$["SkipTangent"] = 1,

idata$$["SkipResidual"] = 0,

idata$$["SubIterationMode"] = 0.

In[20]:=

```
<< "AceGen`";
SMSInitialize["test", "Environment" → "ELFEN"];
SMSTemplate["SMSTopology" → "H1", "SMSSymmetricTangent" → True
  , "SMSDomainDataNames" → {"E -elastic modulus", "ν -poisson ratio",
    "Qx -volume load X", "Qy -volume load Y", "Qz -volume load Z"}
  , "SMSDefaultData" → {21000, 0.3, 0, 0, 0}];
SMSStandardModule["Tangent and residual"];
SMSDo[Ig, 1, SMSInteger[es$$["id", "NoIntPoints"]]];
Ξ = {ξ, η, ζ} ⊢ Table[SMSReal[es$$["IntPoints", i, Ig]], {i, 3}];
XI ⊢ Table[SMSReal[nd$$[i, "X", j]], {i, SMSNoNodes}, {j, SMSNoDimensions}];
Ξn = {{-1, -1, -1}, {1, -1, -1}, {1, 1, -1}, {-1, 1, -1},
  {-1, -1, 1}, {1, -1, 1}, {1, 1, 1}, {-1, 1, 1}};
NI ⊨ Table[1/8 (1 + ξ Ξn⟦i, 1⟧) (1 + η Ξn⟦i, 2⟧) (1 + ζ Ξn⟦i, 3⟧), {i, 1, 8}];
X ⊢ SMSFreeze[NI.XI]; Jg ⊨ SMSD[X, Ξ]; Jgd ⊨ Det[Jg];
uI ⊢ SMSReal[Table[nd$$[i, "at", j], {i, SMSNoNodes}, {j, SMSNoDimensions}]];
pe = Flatten[uI]; u ⊨ NI.uI;
Dg ⊨ SMSD[u, X, "Dependency" → {Ξ, X, SMSInverse[Jg]}];
F ⊨ IdentityMatrix[3] + Dg;
JF ⊨ Det[F];
Cg ⊨ Transpose[F].F;
{Em, ν, Qx, Qy, Qz} ⊢ SMSReal[Table[es$$["Data", i], {i, Length[SMSDomainDataNames]}]];
{λ, μ} ⊨ SMSHookeToLame[Em, ν];
W ⊨ 1/2 λ (JF - 1)^2 + μ (1/2 (Tr[Cg] - 3) - Log[JF]) - {Qx, Qy, Qz}.u;
wgp ⊢ SMSReal[es$$["IntPoints", 4, Ig]];
SMSDo[Rg ⊨ Jgd wgp SMSD[W, pe, i];
  SMSExport[SMSResidualSign Rg, p$$[i], "AddIn" → True];
  SMSDo[Kg ⊨ SMSD[Rg, pe, j];
   SMSExport[Kg, s$$[i, j], "AddIn" → True];
   , {j, i, 24}];
  , {i, 1, 24}];
SMSEndDo[];
SMSWrite[];
```

```
Default value for ELFEN$ElementModel is set to: D3 ≡  three dimensional solid elements
```

File: test.f Size: 27891 Time: 5

Method	SKR2999
No.Formulae	180
No.Leafs	5224

- 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 a 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.

In[348]:=

```
SMSFEEnvironment["ELFEN", "SourceCode" → "test",
 "SimulationInputFile" → "ELFENExample.dat", "SimulationOutputFile" → "tmp.res"]
```

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. Supplementary files are available at FileNameJoin[{$BaseDirectory,"Applications","AceGen","Include", "ANSYS"}] directory. See also `Installation instructions` for instructions how to compile and link the generated codes.

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
S$$[SMSNoTensorComponents]	components of Cauchy stress tensor ($\boldsymbol{\sigma}$) (output)
F$$[9]	current deformation gradient ($\boldsymbol{F}_{n+1}$)
Fn$$[9]	deformation gradient at time n ($\boldsymbol{F}_n$)
W$$	specific strain energy
En$$[SMSNoTensorComponents]	deformation tensor at time n
dE$$[SMSNoTensorComponents]	increment of deformation tensor
DSDE$$[SMSNoTensorComponents, SMSNoTensorComponents]	material constitutive matrix (output)
ed$$["ht",SMSNoTimeStorage]	history dependent real type values per material point
ed$$["hp",SMSNoTimeStorage]	history dependent real type values per material point at time n
es$$["Data", SMSDomainDataNames//Length]	vector of material constants

IO parameters of the USERMAT subroutine.

Abbreviation	Default value	description
SMSNoTensorComponents	Switch[SMSNoDimensions, 1,1,2,4,3,6]	the lenght of the vectorized stress/strain tensor

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[DσDE, 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"}
  , "SMSDefaultData" → {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]}]];
```

In[6]:=
```
𝔽n ⊢ SMSReal[Array[Fn$$, {3, 3}]];
𝔽 ⊢ SMSReal[Array[F$$, {3, 3}]];
JF ⊨ Det[𝔽];
```

In[9]:=
```
SMSFreeze[𝕓e, 𝔽.𝔽ᵀ, "Symmetric" → True];
Jbe = Det[𝕓e];
```

In[11]:=
```
{λ, μ} ⊨ SMSHookeToLame[Em, ν];
{μ, K} ⊨ SMSHookeToBulk[Em, ν];
W ⊨ μ/2 (Jbe^(-1/3) Tr[𝕓e] - 3) + K/β^2 (Jbe^(-β/2) - 1 + β/2 Log[Jbe]);
τ ⊨ Simplify[2 𝕓e.SMSD[W, 𝕓e, "Ignore" → NumberQ, "Symmetric" → True]];
σ ⊨ τ / JF;
DτD𝔽 ⊨ SMSD[τ, 𝔽];
DσDE ⊨ (Table[Sum[
        𝔽[[l, m]] × DτD𝔽[[i, j, k, m]] + 𝔽[[k, m]] × DτD𝔽[[i, j, l, m]], {m, 3}],
       {l, 3}, {k, 3}, {j, 3}, {i, 3}]) / JF / 2;
SMSExport[W, W$$];
```

In[19]:=
```
TensorComponents = {{1, 1}, {2, 2}, {3, 3}, {1, 2}, {2, 3}, {1, 3}};
SMSExport[Extract[σ, TensorComponents], S$$];
SMSExport[
  Table[Extract[DσDE, Join[i1, i2]], {i1, TensorComponents}, {i2, TensorComponents}], DSDE$$];
```

In[22]:=
```
SMSWrite[];
```

File: ANSYS_USERMAT_NH.for Size: 21518 Time: 7

Method	ConstitutiveEquations
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 examples of ANSYS script template for Linux

```
#!/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\".\n"

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/User 1";' .bashrc
source ~/.bashrc

printf "Path set to: ANS_USER_PATH=$ANS_USER_PATH.\n"
```

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\" script located there.\n"

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)\n"

./ANSUSERSHARED
printf "\"ANSUSERSHARED\"file run\n"
```

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

option	default	description
"Optimize"	Automatic	use additional compiler optimization
"PauseOnExit"	False	pause before exiting the *MathLink* executable
"Console"	True	on Windows start the executable as console application (see also `SMSPrintMessage`)
"Platform"	Automatic	"32" ⇒ 32 bit operating system (all operating systems Windows, Unix, Mac) "64" ⇒ 64 bit operating systems (Mac and Windows)
"AdditionalLinkerParameters"	""	additional string added to the linker command line verbatim (it can specify e.g. additional object files or libraries)

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*]

option name	description
"PauseOnExit"→value	True ⇒ pause before exiting the MathLink executable False ⇒ exit without stopping
"SparseArray"→value	True ⇒ return all matrices in sparse format False ⇒ return all matrices in full format Automatic ⇒ return the matrices in a format that depends on the sparsity of the actual matrix

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)

SMSLinkNoEvaluations[] ≡ SMSLinkNoEvaluations[*last_AceGen_session*]

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*

In[14]:=

```
<< AceGen`;
SMSInitialize["test", "Environment" → "MathLink"];
SMSModule["Test", Real[u$$[3], x$$, L$$, g$$[3]],
  "Input" → {u$$, x$$, L$$}, "Output" → g$$];
{x, L} ⊢ {SMSReal[x$$], SMSReal[L$$]};
ui ⊢ SMSReal[Table[u$$[i], {i, 3}]]
Ni ⊨ {x/L, 1 - x/L, x/L (1 - x/L)};
u ⊨ Ni.ui;
f ⊨ u^2;
g ⊨ SMSD[f, ui];
SMSExport[g, g$$];
SMSWrite[];
```

Out[ ]=

$\{ui_1, ui_2, ui_3\}$

File:	test.c	Size: 1838
Methods	No.Formulae	No.Leafs
Test	6	81

In[73]:=

```
SMSInstallMathLink[]
```

Out[ ]=

```
{SMSSetLinkOption[test, {i_Integer, j_Integer}], SMSLinkNoEvaluations[test], Test[u_?
    (ArrayQ[#1, 1, Head[#1] == Real || Head[#1] == Integer &] && Dimensions[#1] === {3} &),
  x_? (Head[#1] == Real || Head[#1] == Integer &),
  L_? (Head[#1] == Real || Head[#1] == Integer &)]}
```

- Here the generated executable is used to calculate gradient for the numerical test example.

In[74]:=

```
Test[{0., 1., 7.}, π // N, 10.]
```

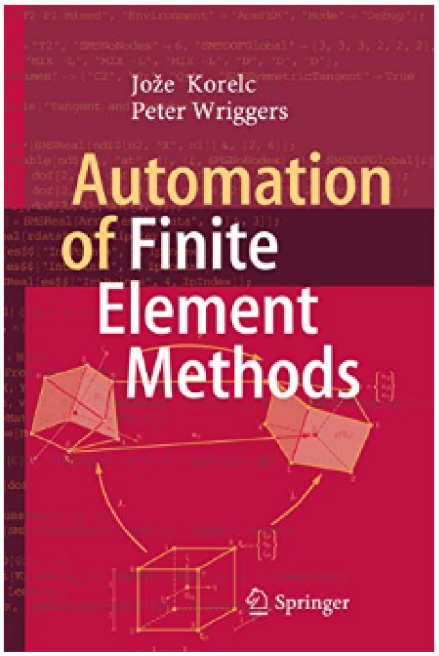
Out[ ]=

```
{1.37858, 3.00958, 0.945489}
```

CHAPTER 7

Appendix

Bibliography



AceGen

KORELC, J. Automation of primal and sensitivity analysis of transient coupled problems. Computational mechanics, 44(5):631-649 (2009).

KORELC, Jože, Multi-language and Multi-environment Generation of Nonlinear Finite Element Codes, Engineering with Computers, 2002, 18(4):312-327

KORELC, Jože. Automatic generation of finite-element code by simultaneous optimization of expressions. Theor. comput. sci.. , 1997, 187:231-248.

HUDOBIVNIK, 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

KORELC, J. Semi-analytical solution of path-independent nonlinear finite element models. Finite elem. anal. des., 2011, 47:281-287.

KORELC, J. Automation of the finite element method. V: WRIGGERS, Peter. Nonlinear finite element methods. Springer, 483-508 (2008).

Applications

ZUPAN, Nina, KORELC, Jože. Sensitivity analysis based multi-scale methods of coupled path-dependent problems, Computational Mechanics, 2019, DOI 10.1007/s00466-019-01762-8

ŠOLINC, Urša, KORELC, Jože. A simple way to improved formulation of FE2 analysis. Computational mechanics, 2015, 56:905-915, doi: 10.1007/s00466-015-1208-4.

MELINK, Teja, KORELC, Jože. Stability of Karhunen- Loève expansion for the simulation of Gaussian stochastic fields using Galerkin scheme. Probabilistic Engineering Mechanics, 2014, 37:7-15, doi: 10.1016/j.probengmech.2014.03.006.

KORELC, Jože, STUPKIEWICZ, Stanislaw. Closed-form matrix exponential and its application in finite-strain plasticity. International journal for numerical methods in engineering, ISSN 0029-5981, 2014, 98(13):960-987, ilustr., doi: 10.1002/nme.4653.

CLASEN, Heiko, HIRSCHBERGER, C. Britta, KORELC, Jože, WRIGGERS, Peter, FE2-homogenization of micromorphic elasto-plastic materials, XII International Conference on Computational Plasticity. Fundamentals and Applications, COMPLAS XII, 2013

LENGIEWICZ, Jakub, KORELC, Joze, STUPKIEWICZ, Stanislaw., Automation of finite element formulations for large deformation contact problems. Int. j. numer. methods eng., 2011, 85: 1252-1279.

KORELC, J. Direct computation of critical points based on Crout`s elimination and diagonal subset test function, Computers and Structures, 88:189-197 (2010).

WRIGGERS, Peter, KRSTULOVIC-OPARA, Lovre, KORELC, Jože.(2001), Smooth C1-interpolations for two-dimensional frictional contact problems. *Int. j. numer. methods eng.*, 2001, vol. 51, issue 12, str. 1469-1495

KRSTULOVIC-OPARA, Lovre, WRIGGERS, Peter, KORELC, Jože. (2002), A C1-continuous formulation for 3D finite deformation frictional contact. *Comput. mech.*, vol. 29, issue 1, 27-42

STUPKIEWICZ, Stanislaw, KORELC, Jože, DUTKO, Martin, RODIC, Tomaž. (2002), Shape sensitivity analysis of large deformation frictional contact problems. *Comput. methods appl. mech. eng.*, 2002, vol. 191, issue 33, 3555-3581

BRANK, Boštjan, KORELC, Jože, IBRAHIMBEGOVIC, Adnan. (2002), Nonlinear shell problem formulation accounting for through-the-tickness stretching and its finite element implementation. *Comput. struct.*. vol. 80, n. 9/10, 699-717

BRANK, Boštjan, KORELC, Jože, IBRAHIMBEGOVIC, Adnan. (2003),Dynamic and time-stepping schemes for elastic shells undergoing finite rotations. *Comput. struct.*, vol. 81, issue 12, 1193-1210

STADLER, Michael, HOLZAPFEL, Gerhard A., KORELC, Jože. (2003) Cn continuous modelling of smooth contact surfaces using NURBS and application to 2D problems. *Int. j. numer. methods eng.*, 2177-2203

KUNC, Robert, PREBIL, Ivan, RODIC, Tomaž, KORELC, Jože. (2002),Low cycle elastoplastic properties of normalised and tempered 42CrMo4 steel. *Mater. sci. technol.*, Vol. 18, 1363-1368.

Bialas M, Majerus P, Herzog R, Mroz Z, Numerical simulation of segmentation cracking in thermal barrier coatings by means of cohesive zone elements, MATERIALS SCIENCE AND ENGINEERING A-STRUCTURAL MATERIALS PROPERTIES MICROSTRUCTURE AND PROCESSING 412 (1-2): 241-251 Sp. Iss. SI, DEC 5 2005

Maciejewski G, Kret S, Ruterana P, Piezoelectric field around threading dislocation in GaN determined on the basis of high-resolution transmission electron microscopy image , JOURNAL OF MICROSCOPY-OXFORD 223: 212-215 Part 3 SEP 2006

Wisniewski K, Turska E, Enhanced Allman quadrilateral for finite drilling rotations, COMPUTER METHODS IN APPLIED MECHANICS AND ENGINEERING 195 (44-47): 6086-6109 2006

Maciejewski G, Stupkiewicz S, Petryk H, Elastic micro-strain energy at the austenite-twinned martensite interface, ARCHIVES OF MECHANICS 57 (4): 277-297 2005

Stupkiewicz S, The effect of stacking fault energy on the formation of stress-induced internally faulted martensite plates, EUROPEAN JOURNAL OF MECHANICS A-SOLIDS 23 (1): 107-126 JAN-FEB 2004

Symbolic methods

Korelc J. (1997b), A symbolic system for cooperative problem solving in computational mechanics, *Computational Plasticity Fundamentals and Applications*, (Owen D.R.J., Oñate E. and Hinton E., editors), CIMNE, Barcelona, 447-451.

Korelc J., and Wriggers P. (1997c), Symbolic approach in computational mechanics, *Computational Plasticity Fundamentals and Applications*, (Owen D.R.J., Oñate E. and Hinton E., editors), CIMNE, Barcelona, 286-304.

Korelc J., (2001), Hybrid system for multi-language and multi-environment generation of numerical codes, Proceedings of the ISSAC'2001 Symposium on Symbolic and Algebraic Computation, New York, ACM:Press, 209-216

Korelc, J. (2003) Automatic Generation of Numerical Code. MITIC, Peter. Challenging the boundaries of symbolic computation : proceedings of the 5th International Mathematica Symposium. London: Imperial College Press, 9-16.

Gonnet G. (1986), New results for random determination of equivalence of expression, Proc. of 1986 ACM Symp. on Symbolic and Algebraic Comp, (Char B.W., editor), Waterloo, July 1986, 127-131.

Griewank A. (1989), On Automatic Differentiation, *Mathematical Programming: Recent Developments and Applications*, Kluwer Academic Publisher, Amsterdam, 83-108.

Hulzen J.A. (1983), Code optimization of multivariate polynomial schemes: A pragmatic approach. Proc. of IEUROCAL'83, (Hulzen J.A., editor), Springer-Verlag LNCS Series Nr. 162.

Kant E. (1993), Synthesis of Mathematical Modeling Software, *IEEE Software*, May 1993.

Leff L. and Yun D.Y.Y. (1991), The symbolic finite element analysis system. *Computers & Structures*, **41**, 227-231.

Noor A.K. (1994), Computerized Symbolic Manipulation in Structural Mechanics, *Computerized symbolic manipulation in mechanics*, (Kreuzer E., editor), Springer-Verlag, New York, 149-200.

Schwartz J.T. (1980), Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM*, **27**(4), 701-717.

Sofroniou M. (1993), An efficient symbolic-numeric environment by extending mathematica's format rules. Proceedings of Workshop on Symbolic and Numerical Computation, (Apiola H., editor), University of Helsinki, Technical Report Series, 69-83.

Wang P.S. (1986), Finger: A symbolic system for automatic generation of numerical programs in finite element analysis, *J. Symb. Comput,* **2**, 305-316.

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.q. formulas $\text{Sin}[x]^2+\text{Cos}[x]^2$ 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[14]:=
```
<< AceGen`;
SMSInitialize["test", "Language" → "C"];
SMSModule["test", Real[x$$, f$$]];
x ⊢ SMSReal[x$$];
SMSIf[x ≤ 0];
 f ⫤ x^2;
SMSElse[];
 f ⫞ Sin[x];
SMSEndIf[];
SMSExport[f, f$$];
```

- Corrected input

In[24]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "Fortran"];
SMSModule["test", Real[x$$, f$$]];
x ⊢ SMSReal[x$$];
SMSIf[x ≤ 0];
 f ⫤ x^2;
SMSElse[];
 f ⫤ Sin[x];
SMSEndIf[f];
SMSExport[f, f$$];
```

Symbol is defined in other branch of "If" construct

- Erroneous input

In[34]:=

```
<< AceGen`;
SMSInitialize["test", "Language" → "C"];
SMSModule["test", Real[x$$, f$$]];
x ⊢ SMSReal[x$$];
f ⫤ x;
SMSIf[x ≤ 0];
 f ⫤ x^2;
SMSElse[];
 y ⊨ 2 f;
```

- Corrected input

In[43]:=

```
SMSInitialize["test", "Language" → "C"];
SMSModule["test", Real[x$$, f$$]];
x ⊢ SMSReal[x$$];
f ⫤ x;
tmp ⊢ f;
SMSIf[x ≤ 0];
 f ⫤ x^2;
SMSElse[];
 y ⊨ 2 tmp;
```

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 erroneus and corrected input:

- Erroneous input

In[52]:=

```
a < b < c
```

In[53]:=
```
FullForm[a < b < c]
```

In[54]:=
```
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

In[55]:=
```
Unprotect[CForm];
CForm[Less[a_, b_, c_]] := a < b && b < c;
Protect[CForm];
```

In[58]:=
```
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`