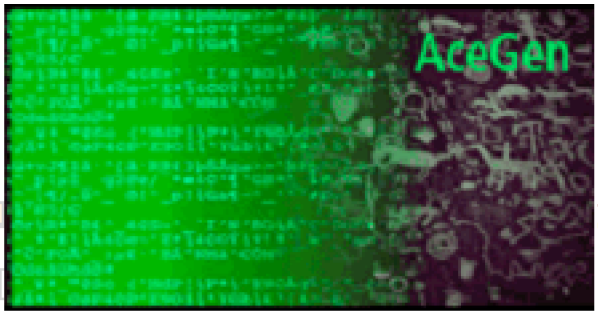



```
SMSInitial                                  Fortran"];
SMSModule[
x = SMSReal[x$$];

Module : test

SMSIf[x <= 0];
 f = x^2;
SMSElse[];
 f = Sin[x];
SMSEndIf[f];

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

Function : test 4 formulae, 18 sub expressions

[0] File created : test.f Size : 850
```

AceGen

Jože Korelc

University of Ljubljana

Ljubljana, 2012

Slovenia

AceGen Contents

AceGen Tutorials

AceGen Preface

AceGen

© **Prof. Dr.Jože Korelc**, 2006, 2007, 2008, 2009, 2010
Ravnikova 4, SI - 1000, Ljubljana, Slovenia
E–mail : AceProducts@fgg.uni - lj.si
www.fgg.uni - lj.si/Symech/

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

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

environment AceFEM the AceGen package represents ideal tool for a rapid development of new numerical models.

AceGen Overview

General AceGen Session

`SMSInitialize` — start AceGen session

`SMSModule` — start new user subroutine

`SMSWrite` — end AceGen session and create source file

Assignments and expression manipulations

`⊨ . ⊢ . ⫤ . ⊣` — assignment operators

`SMSInt` . `SMSFreeze` . `SMSFictive` — special assignments

`SMSSimplify` . `SMSReplaceAll` . `SMSSmartReduce` . `SMSSmartRestore` . `SMSRestore` . `SMSVariables` — auxiliary variables manipulations

`SMSArray` . `SMSPart` . `SMSReplacePart` . `SMSDot` . `SMSSum` — aperations with arrays

`SMSD` . `SMSDefineDerivative` — automatic differentiation

Symbolic-numeric interface

`SMSReal` . `SMSInteger` . `SMSLogical` . `SMSRealList` . — import from input parameters

`SMSExport` — export to output parameters

`SMSCall` — call external subroutines

Program Flow Control

`SMSIf` . `SMSElse` . `SMSEndIf` . SMSSwitch . SMSWhich — conditionals

`SMSDo` . `SMSEndDo` — loop construct

`SMSReturn` . `SMSBreak` . `SMSContinue` .

Special 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` — creates new notebook that includes only evaluated cells

`SMSTagIf` . `SMSTagSwitch` . `SMSTagReplace` . — manipulate break points

Debugging

`SMSSetBreak` — insert break point

`SMSLoadSession` — reload the data and definitions for debugging session

`SMSClearBreak` . `SMSActivateBreak` — creates 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

`SMSInitialize` . `SMSTemplate` . `SMSStandardModule` . `SMSWrite` — start AceGen session, et finite element attributes, create element user subroutines and create element source file

`SMSFEAPMake` . `SMSFEAPRun` . `SMSELFENMake` . `SMSELFENRun` . `SMSABAQUSMake` . `SMSABAQUSRun` . — link and run generated element with choosen environment

Integer Type Environment Data (idata$$), Real Type Environment Data (rdata$$), Domain Specification Data (es$$), Element Data (ed$$), Node Specification Data (ns$$), Node Data (nd$$) — FEM data structures

Standard FE Procedure . Summary of Examples . ABAQUS . FEAP . ELFEN . User defined environment interface — FEM examples

Introduction

■ General

Symbolic and algebraic computer systems such as *Mathematica* are general and very powerful tools for the manipulation of formulae and for performing various mathematical operations by computer. However, in the case of complex numerical models, direct use of these systems is not possible. Two reasons are responsible for this fact: a) during the development stage the symbolic derivation of formulae leads to uncontrollable growth of expressions and consequently

redundant operations and inefficient programs, b) for numerical implementation SAC systems can not keep up with the run-time efficiency of programming languages like FORTRAN and C and by no means with highly problem oriented and efficient numerical environments used for finite element analysis.

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

⇒ symbolic and algebraic computations (SAC) systems,

⇒ automatic differentiation (AD) tools,

⇒ problem Solving Environments (PSE),

⇒ theorem proving systems (TP),

⇒ numerical libraries,

⇒ specialized systems for FEM.

■ AceGen

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

General characteristics of *AceGen* code generator:

⇒ simultaneous optimization of expressions immediately after they have been derived (Expression Optimization),

⇒ automatic differentiation technique (Automatic Differentiation , Exceptions in Differentiation),

⇒ automatic selection of the appropriate intermediate variables,

⇒ the whole program structure can be generated (Mathematica syntax - AceGen syntax),

⇒ appropriate for large problems where also intermediate expressions can be subjected to the uncontrolled swell,

⇒ improved optimization procedures with stochastic evaluation of expressions,

⇒ differentiation with respect to indexed variables,

⇒ automatic interface to other numerical environments (by using Splice command of Mathematica),

⇒ multi-language code generation (Fortran/Fortran90, C/C++, Mathematica language, Matlab language),

⇒ advanced user interface,

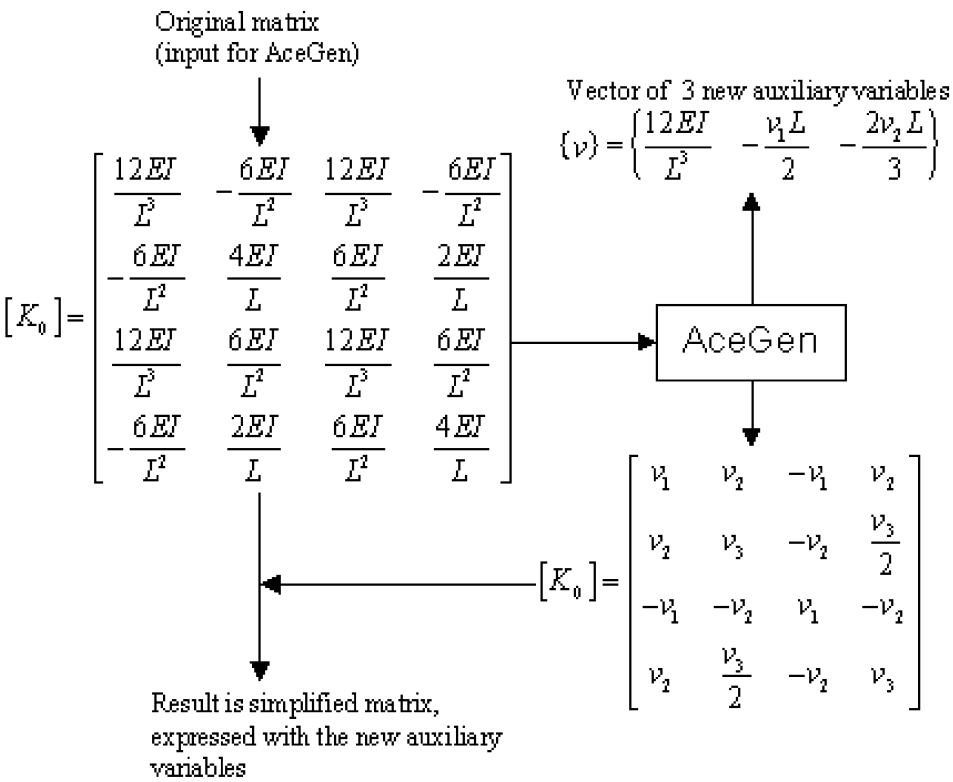
⇒ advanced methods for exploring and debugging of generated formulae,

⇒ special procedures are needed for 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 interac-

tively 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.

■ *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,

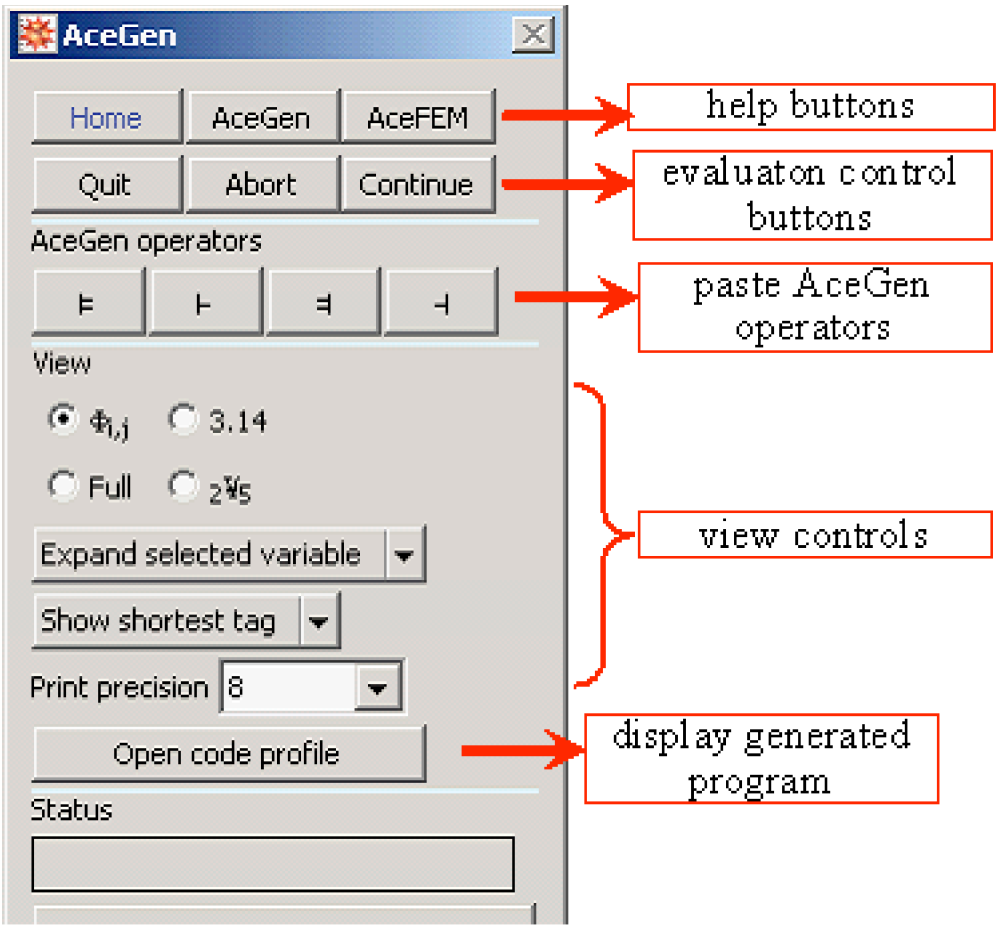
⇒ symbolic integration,

⇒ symbolic solution to the system of algebraic equations.

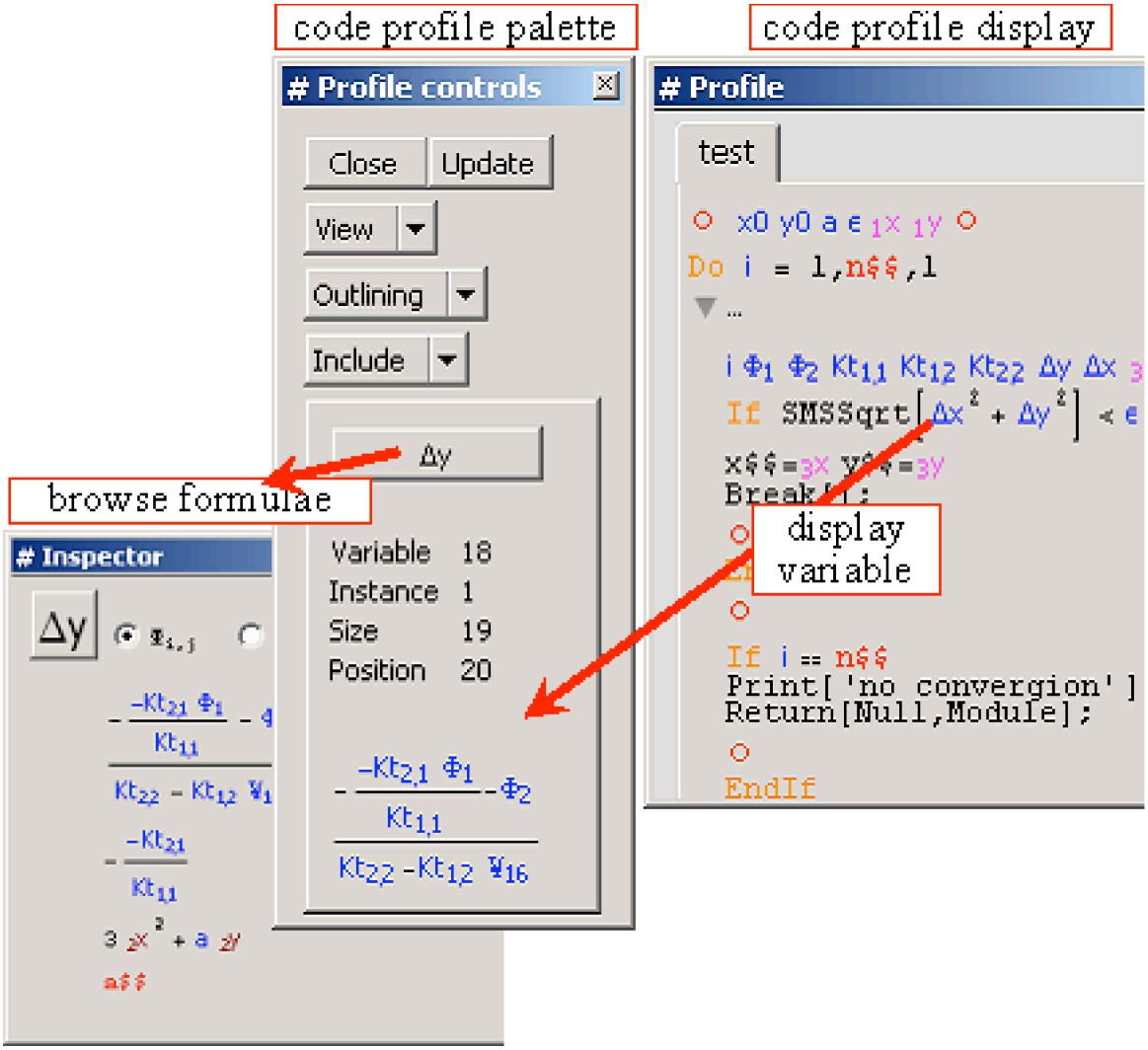
Each of these operations can be directly implemented also with the built-in Mathematica functions and the result optimized by *AceGen*. However, by using equivalent functions in *AceGen* with simultaneous optimization of expressions, much larger problems can be efficiently treated. Unfortunately, the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

AceGen Palettes

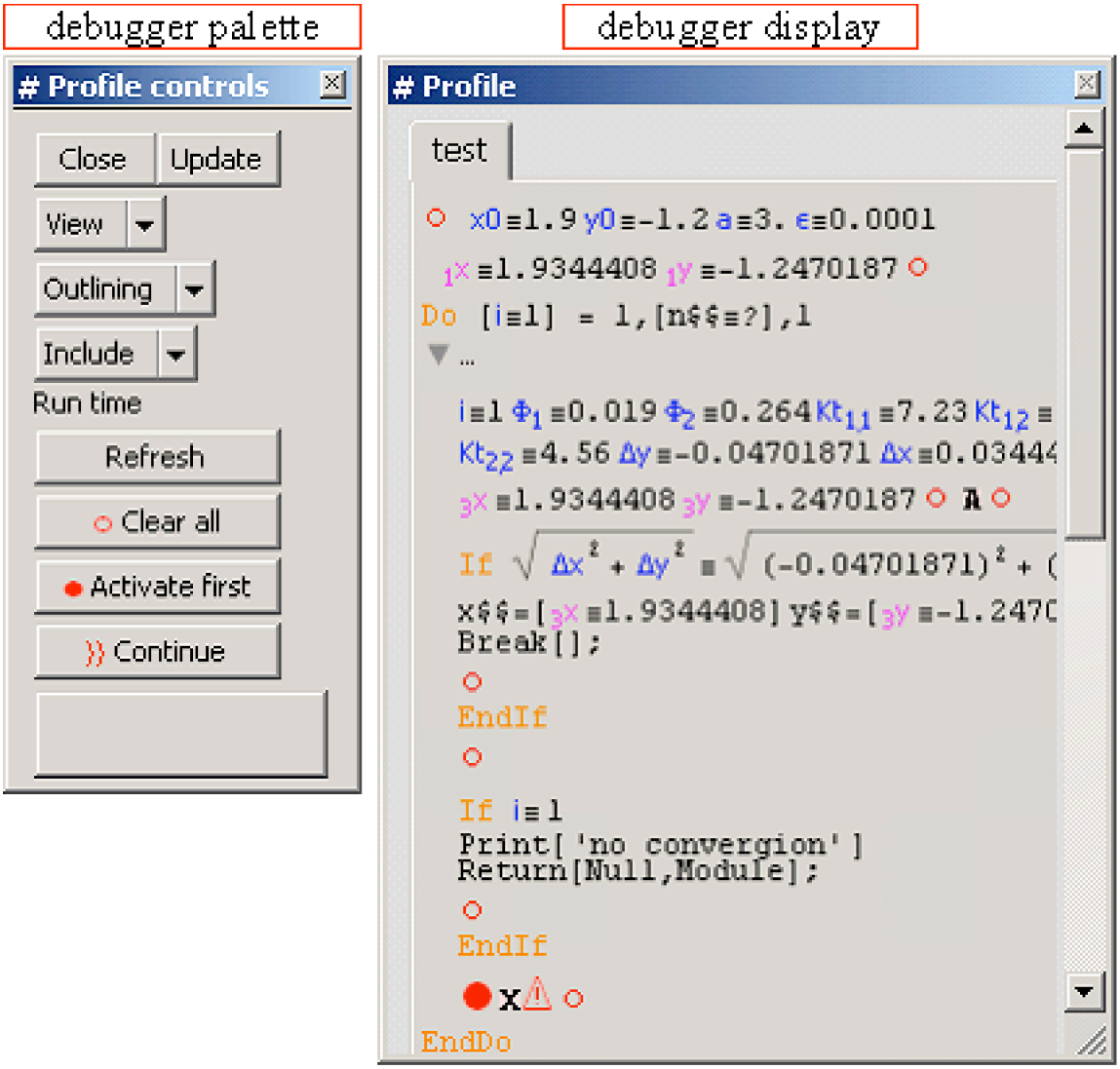
Main AceGen palette.



Display generated program.



Debugger palette and display.



Standard AceGen Procedure

■ 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$$[i],{i,3}]];
4	Derivation of the problem	⟹	$Ni\vDash\{\frac{x}{L},1-\frac{x}{L},\frac{x}{L}\left(1-\frac{x}{L}\right)\}$; u⊨Ni.ui; $f\vDash u^2$; g⊨SMSD[f,ui];
5	Definition of symbolic–numeric interface variables	⟹	SMSExport[g,g$$];
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.

```
<<AceGen`
```

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

```
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.

```
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.

```
x ⊢ SMSReal[x$$]
```

x

```
L ⊢ SMSReal[L$$]
```

L

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

```
ui ⊢ SMSReal[Table[u$$[i] , {i, 3}]]
```

$\{ui_1, ui_2, ui_3\}$

Step 4: Description of the Problem

Here is the body of the subroutine.

```
Ni ⊨ {x / L, 1 - x / L, x / L * (1 - x / L)}
```

$\{Ni_1, Ni_2, Ni_3\}$

```
u ⊨ Ni . ui
```

u

```
f ⊨ u^2
```

f

```
g ⊨ SMSD[f, ui]
```

$\{g_1, g_2, g_3\}$

Step 5: Definition of Symbolic - Numeric Interface Variables

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

```
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.

```
SMSWrite[];
```

File:	test.f	Size: 946
Methods	No.Formulae	No.Leafs
Test	6	81

This displays the contents of the generated file.

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

```
!*************************************************************
!* AceGen    2.502 Windows (18 Nov 10)                       *
!*           Co. J. Korelc  2007            24 Nov 10 13:30:52*
!*************************************************************
! 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 Fortran code      : 379 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(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(15)=2d0*v(9)
      g(1)=v(15)*v(6)
      g(2)=v(15)*v(7)
      g(3)=v(15)*v(8)
      END
```

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.

```
<< 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
```

```
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* and External Program Communication) version of the source code is generated. The generated code is automatically enhanced by an additional modules necessary for the proper *MathLink* connection.

```
<< 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
```

```
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)

```
SMSInstallMathLink[]
```

```
{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. (see also Verification of Automatically Generated Code).

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

```
{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.

```
<< 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
```

```
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
```

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, SMSWhich and SMSSwitch 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*, *Expression Optimization*).
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

```
x =.; n =.;
y = Sin[x]^n
```

```
Sin[x]^n
```

AceGen input

```
<< 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$$ n≡n$$ v≡Sin[x]^n

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

```
Method : Test 1 formulae, 13 sub-expressions

[0] File created : test.c Size : 726
```

```
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

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

```
1 + Sin[If[$V[1, 1] ≥ 0, If[x ≥ 7, 7, x], x^2]]
```

AceGen input

```
<< 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]
```

```
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!

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

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

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.

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

```
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)

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

$$z = Sin[y] + 1$$

Mathematica input

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

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

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

AceGen input

```
<< 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]
```

```
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
```

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.

See also: SMSR, SMSS, SMSReal, SMSInteger , Mathematica syntax - AceGen syntax

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 the 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.

```
<< 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.

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

```
{$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
```

```
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.

```
<< 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.

```
x // ToString
```

```
$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)
```

```
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
```

User Interface

An important question arises: how to understand the automatically generated formulae? The automatically generated code should not act like a "black box". For example, after using the automatic differentiation tools we have no insight in the actual structure of the derivatives. While formulae are derived automatically with *AceGen*, *AceGen* tries to find the actual meaning of the auxiliary variables and assigns appropriate names. By asking *Mathematica* in an interactive dialog about certain symbols, we can retain this information and explore the structure of the generated expressions. In the following *AceGen* sessions various possibilities how to explore the structure of the program are presented.

Example

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

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

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.

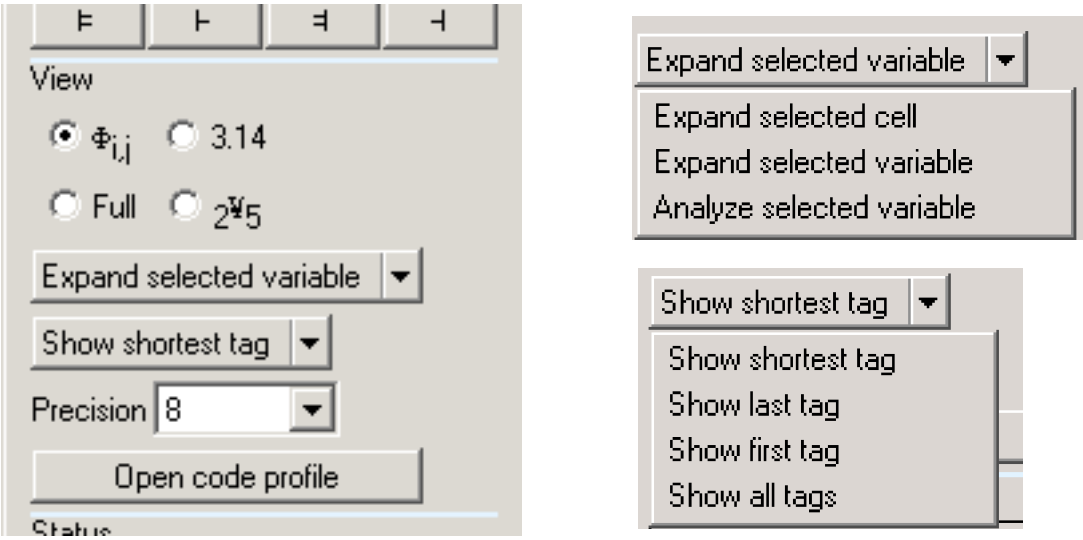
```
<< AceGen`;
SMSInitialize["test", "Language" -> "Mathematica", "Mode" → "Debug"];
SMSModule["test", Real[x$$, y$$, a$$, tol$$], Integer[nmax$$]];
{x0, y0, a, ϵ} ⊢ SMSReal[{x$$, y$$, a$$, tol$$}];
nmax ⊢ SMSInteger[nmax$$];
{x, y} ⫤ {x0, y0};
SMSDo[
  Φ ⊨ {a x y + x^3, a - x y^2};
  Kt ⊨ SMSD[Φ, {x, 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: 2491
Methods	No.Formulae	No.Leafs
test	33	198

Exploring the structure of the formula - Browser mode submenu

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

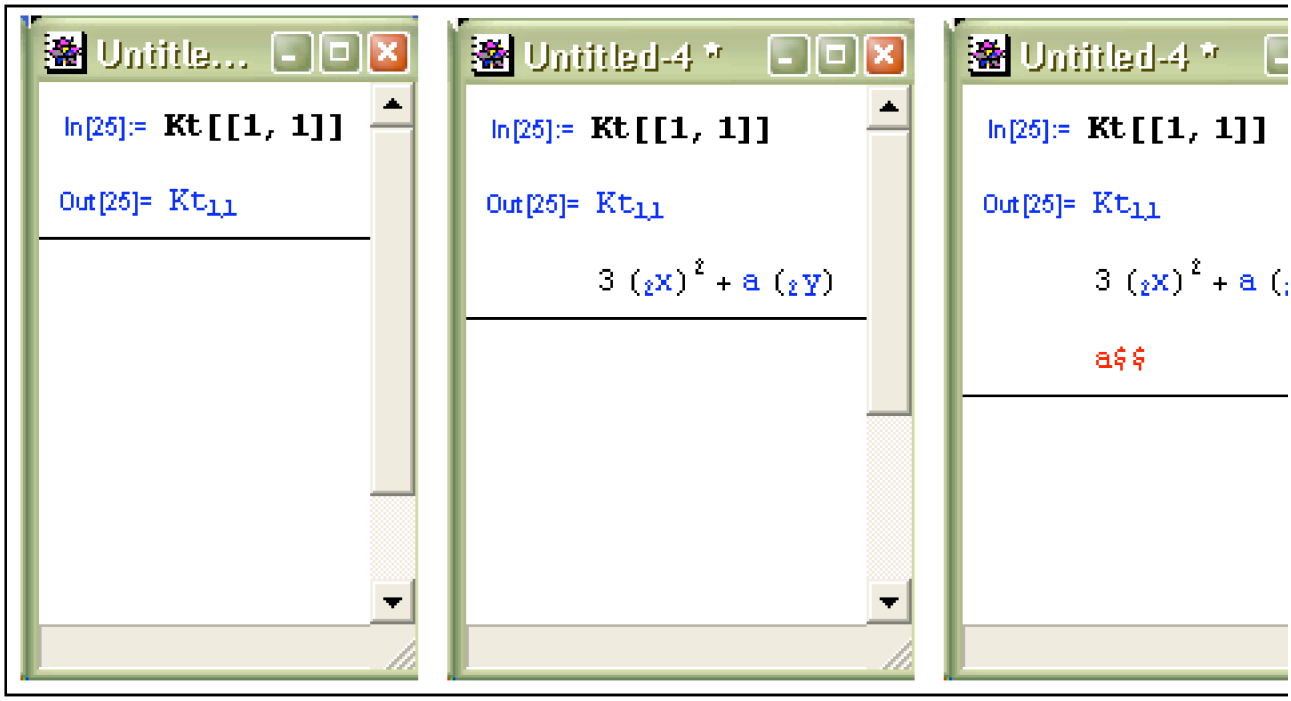


Palette for entering *AceGen* commands that control user-*AceGen* interactions.

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 $\Phi_{1\cdot x} = \frac{\partial \Phi 1}{\partial x}$).

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

$Kt_{1\,1}$



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

```
Kt
```

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

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

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

```
Kt
```

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

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

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

Button: Analyze selected variable

```
Kt
```

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

Variable=$V[12, 1] Tags= $Kt_{1,1}$ | $\Phi_{1;x}$

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

Position in program={1, 2, 9, 2, 7} Position in data base=14 Module=1

Skope={Do[i, 1, n$$, 1]}

Type=Real Singlevalued=True No. of instances=1 Stochastic values={1.22245}

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: $\Phi_{i,j}$

```
Kt
```

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

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

```
Kt
```

```
{{$V[12, 1], $V[14, 1]}, {-$V[13, 1], $V[15, 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: $2^{¥}5$

```
Kt
```

$\{\{¥_{12}, ¥_{14}\}, \{-¥_{13}, ¥_{15}\}\}$

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.

Button: 3.14

```
Kt
```

```
{{1.2224526, 0.44704429}, {-0.025662073, -0.19439409}}
```

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).

```
Kt
```

$\{\{Kt_{11}, Kt_{12}\}, \{-\,-Kt_{21}, Kt_{22}\}\}$

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

```
Kt
```

$\{\{Kt_{11}, Kt_{12}\}, \{-\,-\Phi_{2\cdot x}, Kt_{22}\}\}$

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

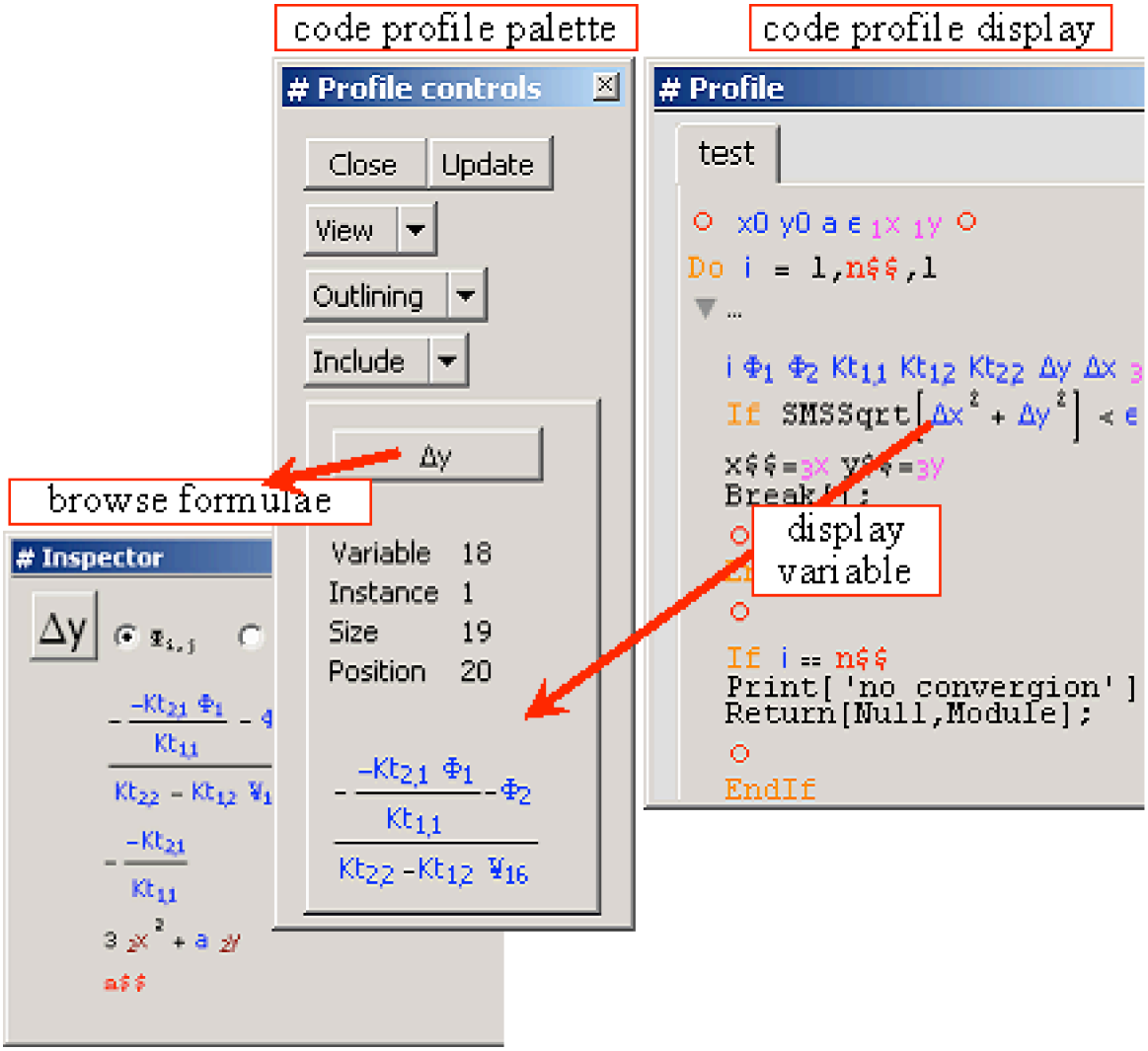
```
Kt
```

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

Analyzing the structure of the program

The 

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.

Verification of Automatically Generated Code

We can verify the correctness of the generated code directly in *Mathematica*. To do this, we need to rerun the problem and to generate the code in a script language of *Mathematica*. The *SMSSetBreak* function inserts a break point into the generated code where the program stops and enters interactive debugger (see also User Interface).

```
<< AceGen`;
SMSInitialize["test", "Language" -> "Mathematica", "Mode" -> "Debug"];
SMSModule["Test", Real[u$$[3], x$$, L$$, g$$[3]]];
{x, L} ⊢ {SMSReal[x$$], SMSReal[L$$]};
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: 1348
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.

```
u
```

u

In the case of more complex code, the code profile 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.

```
<<"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 ald ones. Thus the names of the arguments cannot be the same as the symbols used in the definition of the subroutine.

```
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.

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

Here the numerical results are displayed.

```
gv
```

```
{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.

```
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.

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

Here the partially evaluated gradient is displayed.

```
gv // Expand
```

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

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*.

```
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
```

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

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

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

```
{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.

```
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.

Program Flow Control

AceGen can automatically generate conditionals (SMSIf, SMSSwitch, 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 conditional 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}]
```

See also: SMSIf, SMSElse, SMSEndIf, SMSSwitch, SMSWhich, SMSVerbatim, SMSDo, SMSEndDo.

Example 1: Gauss integration

Generation of the Fortran subroutine calculates the integral $\int_a^b x^2 + 2\,\mathrm{Sin}[x^3]\,dx$ by employing Gauss integration scheme. The source code is written in FORTRAN language. The input for the subroutine are the Gauss points and the Gauss weights defined on interval [-1,1] and an integration interval [a,b].

```
<< AceGen`;
SMSInitialize["test", "Language" -> "Fortran"];
SMSModule["test", Real[gp$$[ng$$], wg$$[ng$$], a$$, b$$, r$$], Integer[ng$$]];
{a, b} ⊢ SMSReal[{a$$, b$$}];
ng ⊢ SMSInteger[ng$$];
intg ⊣ 0;
SMSDo[
  (* map the interval [-1,1] to interval [a,b]*)
  {ξg, wg} ⊢ SMSReal[{gp$$[i], wg$$[i]}];
  x ⊨ (a + b)/2 + 1/2 (-a + b) ξg;
  Jg ⊨ 1/2 (-a + b);
  intg ⊣ intg + wg Jg (x^2 + 2 Sin[x^3]) ;
, {i, 1, ng, 1, intg}];
SMSExport[intg, r$$];
SMSWrite[];
FilePrint["test.f"]
```

File: test.f	Size: 1028
Methods	No.Formulae	No.Leafs
test	7	82

```
!*************************************************************
!* AceGen    2.502 Windows (24 Nov 10)                       *
!*           Co. J. Korelc  2007            29 Nov 10 15:03:23*
!*************************************************************
! User : Full professional version
! Evaluation time                 : 0 s      Mode   : Optimal
! Number of formulae              : 7        Method: Automatic
! Subroutine                      : test size :82
! Total size of Mathematica  code : 82 subexpressions
! Total size of Fortran code      : 438 bytes

!******************* S U B R O U T I N E *********************
      SUBROUTINE test(v,gp,wg,a,b,r,ng)
      IMPLICIT NONE
      include 'sms.h'
      INTEGER ng,i3,i5
      DOUBLE PRECISION v(5005),gp(ng),wg(ng),a,b,r
      v(9)=-a+b
      v(10)=v(9)/2d0
      i3=int(ng)
      v(4)=0d0
      DO i5=1,i3
       v(8)=(a+b+gp(i5)*v(9))/2d0
       v(4)=v(4)+v(10)*wg(i5)*((v(8)*v(8))+2d0*dsin(v(8)**3))
      ENDDO
      r=v(4)
      END
```

Example 1: Newton-Raphson (in-cell form)

The generation of the *Mathematica* subroutine that calculates the zero of function f(x) $=x^2 + 2\,\mathrm{Sin}\left[x^3\right]$ by using Newton-Raphson iterative procedure.

```
<< AceGen`;
SMSInitialize["test", "Language" -> "Mathematica"];
SMSModule["test", Real[x0$$, r$$]];
x ⇐ SMSReal[x0$$];
(*This starts iterative loop.*)
SMSDo[
  (*Description of the Newton-Raphson iterative procedure.*)
  f ⇐ x^2 + 2 Sin[x^3];
  dx ⇐ -f/SMSD[f, x];
  x ← x + dx;
  (*Here we exit the "Do" loop when
   the convergence of the iterative solution is obtained.*)
  SMSIf[Abs[dx] < .00000001, SMSBreak[];];
  (*Here the divergence of the Newton-
   Raphson procedure is recognized and reported and the program is aborted.*)
  SMSIf[i == 15, SMSPrintMessage["no convergence"]; SMSReturn[];];
, {i, 1, 30, 1, {x}}];
SMSExport[x, r$$];
SMSWrite[];
FilePrint["test.m"]
```

```
Method : test 9 formulae, 61 sub-expressions

[0] File created : test.m Size : 969

(*************************************************************
* AceGen    2.103 Windows (18 Jul 08)                        *
*           Co. J. Korelc  2007            18 Jul 08 16:36:45*
*************************************************************
User : USER
Evaluation time                 : 0 s     Mode  : Optimal
Number of formulae              : 9       Method: Automatic
Module                          : test size : 61
Total size of Mathematica  code : 61 subexpressions        *)
(********************** M O D U L E *************************)
SetAttributes[test,HoldAll];
test[x0$$_,r$$_]:=Module[{},
$VV[1]=x0$$;
Do[
 $VV[5]=$VV[1]^2;
 $VV[4]=$VV[1]^3;
 $VV[7]=-((2*Sin[$VV[4]]+$VV[5])/(2*$VV[1]+6*Cos[$VV[4]]*$VV[5]));
 $VV[1]=$VV[1]+$VV[7];
 If[Abs[$VV[7]] < 0.1*10^-7,
  Break[];
 ]; (* endif *)
 If[$VV[2]==15,
  Print["no convergence"];
  Return[Null,Module];
 ]; (* endif *)
,{$VV[2],1,30,1}]; (*EndDo*)
r$$=$VV[1];
];
```

Example 1: Newton-Raphson (cross-cell form)

The generation of the C subroutine calculates the zero of function f(x) $=x^2 + 2\,\mathrm{Sin}\left[x^3\right]$ by using Newton-Raphson iterative procedure. The formulation is the same as before exept the "cross-cell" form of the control structure is used insted of the "in-cell" form.

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

```
<< AceGen`;
SMSInitialize["test", "Language" -> "C"];
SMSModule["test", Real[x0$$, r$$]];
x ⫤ SMSReal[x0$$];

SMSDo[i, 1, 30, 1, {x}];

 f ⊨ x^2 + 2 Sin[x^3];
 dx ⊨ -f/SMSD[f, x];
 x ⫤ x + dx;

 SMSIf[Abs[dx] < .00000001];

  SMSBreak[];

 SMSEndIf[];

 SMSIf[i == 15];
    SMSPrint["no convergence"]
    SMSReturn[];
  SMSEndIf[];

True

SMSEndDo[x];

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

```
Method : test 9 formulae, 61 sub-expressions

[1] File created : test.c Size : 1015
```

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

```
/************************************************************
* AceGen    2.103 Windows (17 Jul 08)                       *
*           Co. J. Korelc  2007            17 Jul 08 23:57:50*
************************************************************
User : USER
Evaluation time                 : 1 s     Mode  : Optimal
Number of formulae              : 9       Method: Automatic
Subroutine                      : test size :61
Total size of Mathematica  code : 61 subexpressions
Total size of C code            : 436 bytes*/
#include "sms.h"

/****************** S U B R O U T I N E ********************/
void test(double v[5005],double (*x0),double (*r))
{
int i2,b8,b10;
v[1]=(*x0);
for(i2=1;i2<=30;i2++){
 v[5]=(v[1]*v[1]);
 v[4]=Power(v[1],3);
 v[7]=-((v[5]+2e0*sin(v[4]))/(2e0*v[1]+6e0*v[5]*cos(v[4])));
 v[1]=v[1]+v[7];
 if(fabs(v[7])<0.1e-7){
  break;
 } else {
 };
 if(i2==15){
  printf("\n%s ","no convergence");
  return;
 } else {
 };
};/* end for */
(*r)=v[1];
};
```

Symbolic-Numeric Interface

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$$, ".name",5]]	x (10) illegal illegal	x[10] x[i-1]->name[5] x[i-1].name[5]
integer variable	SMSInteger[i$$] SMSInteger[i$$$]	i i	*i i
integer array	SMSInteger[i$$[10]] SMSInteger[i$$["10"]] SMSInteger[i$$[j$$, "->name",5]] SMSInteger[i$$[j$$, ".name",5]]	i (10) i (10) illegal illegal	i[10] i[10] i[j-1]->name[5] i[j-1].name[5]
logical variable	SMSLogical[l$$] 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.

See also: SMSReal, SMSExport.

Automatic Differentiation

Theory of Automatic Differentiation

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

⇒ evaluation of consistent tangent matrices for non-standard physical models,

⇒ sensitivity analysis according to arbitrary parameters,

⇒ optimization problems,

⇒ inverse analysis.

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

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

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

$$v_i = f_i(v_j)_{j \in A_i}$$

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

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,...,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:

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

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

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

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

SMSD function

SMSD[*exp*,*v*]	partial derivative $\frac{\partial exp}{\partial v}$
SMSD[*exp*,{*v1*,*v2*,...}]	gradient of *exp* $\left\{\frac{\partial exp}{\partial v_1}, \frac{\partial exp}{\partial v_2}, \ldots\right\}$
SMSD[{*exp1*,*exp2*,...},{*v1*,*v2*,...}]	the Jacobian matrix $J=\left[\frac{\partial exp_i}{\partial v_j}\right]$
SMSD$\left[exp,\{\{v_{11},v_{12},\ldots\},\{v_{21},v_{22},\ldots\},\ldots\}\right]$	differentiation of scalar with respect to matrix $\left[\frac{\partial \mathrm{exp}}{\partial v_{ij}}\right]$
SMSD[*exp*,{*v1*,*v2*,...}, *index*]	create a characteristic expression for an arbitrary element of the gradient $\left\{\frac{\partial exp}{\partial v_1}, \frac{\partial exp}{\partial v_2}, \ldots\right\}$ 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 $\left\{\frac{\partial exp}{\partial acegenarray}\right\}$ 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_structu* with each scalar *exp* replaced by the substructure $\frac{\partial exp}{\partial var\_structure}$. $\left(\text{e.g. derivatives of second order tensors can be generated } D_{i,j,k,l}=\frac{\partial f_{i,j}}{\partial x_{k,l}}\right)$

Automatic differentiation procedures.

option name	*default value*	
"Constant"->{*v1*,*v2*,...}	{}	perform differentiation under assumption that formulas involved do not depend on given variables (directional derivative)
"Constant"->*v*		≡ "Constant"->{*v*}
"Method"->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 , note 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")
"PartialDerivatives"->*truefalse*	False	whether to account also for partial derivatives of auxiliary variables with respect to arbitrary auxiliary variable defined by SMSDefineDerivatives command (by default only total derivatives of auxiliary variables with respect to independent variables are accounted for) TO BE USED ONLY BY THE ADVANCED USERS!!

Options for SMSD.

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 hare 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: $\nabla_x = \frac{\partial f}{\partial x_j}$

```
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
```

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

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: $\nabla_x = \frac{\partial f}{\partial x_j}$

  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}, {Jx₂₁, Jx₂₂}}

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

$$\begin{pmatrix} 2\left(4\,k+x+2\,v+3\,z\right)^{1} & 4\left(4\,k+x+2\,v+3\,z\right)^{2} \end{pmatrix}$$

- 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}}]
```

$$\{\{\{\{1,2\},\{3,4\}\},\{\{Ix_{21},Ix_{22}\},\{Dx_{1221},Dx_{1222}\}\}\},$$
$$\{\{\{Dx_{2111},Dx_{2112}\},\{Dx_{2121},Dx_{2122}\}\},$$
$$\{\{Dx_{2211},Dx_{2212}\},\{Dx_{2221},Dx_{2222}\}\}\}\}$$

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

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

See also SMSD for additional examples.

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: 817
Methods	No.Formulae	No.Leafs
test	4	38

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

```
/************************************************************
* AceGen    3.305 Windows (6 Jul 12)                        *
*           Co. J. Korelc  2007            6 Jul 12 19:59:17 *
************************************************************
User      : USER
Notebook  : AceGenSymbols.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

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

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

```
<< 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: 1016
Methods	No.Formulae	No.Leafs
test	11	88

```
/*************************************************************
* AceGen    3.305 Windows (6 Jul 12)                         *
*           Co. J. Korelc  2007            6 Jul 12 19:59:43 *
*************************************************************
User      : USER
Notebook  : AceGenSymbols.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 to possibilities to make proper differentiation:

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

MF⊨SMSFreeze[**M**];

δexp⊨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 M = $\begin{array}{|c|c|}\hline a & b \\ \hline b & c \\ \hline\end{array}$ and functions $f_{\text{iso}} = \det(\boldsymbol{M})$ and $f_{\text{general}} = M_{1,1}{}^2 + 5\,M_{1,2} - \sin(M_{2,1}) - 3\,M_{2,2}$, thus

$$\frac{\partial f_{\text{iso}}}{\partial \boldsymbol{M}} = \begin{array}{|c|c|}\hline a & -b \\ \hline -b & c \\ \hline\end{array} \text{ and } \frac{\partial f_{\text{general}}}{\partial \boldsymbol{M}} = \begin{array}{|c|c|}\hline 2\,a & 5 \\ \hline -\cos(b) & -3 \\ \hline\end{array}.$$

```
<< 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 x 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

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

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
```

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

SMSFreeze creates unique variables for all component 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
```

$$\begin{pmatrix} M_{22} & -M_{21} \\ -M_{12} & M_{11} \end{pmatrix}$$

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

$$\begin{pmatrix} 2\,M_{11} & 5 \\ -\mathrm{Cos}\left[M_{21}\right] & -3 \end{pmatrix}$$

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.

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

```
<< 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 0
Events: 0
Version: 3.305 Windows (6 Jul 12) (MMA 8.)
See also: SMSD AceGen Troubleshooting

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 $w = x^2 + 1$.

$$w = x^2 + 1$$

$$\frac{\partial \sin(w)}{\partial w}$$

- 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: 3.305 Windows (6 Jul 12) (MMA 8.)

See also: SMSD AceGen Troubleshooting

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]

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[];
```

Exceptions in Differentiation

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 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.

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.

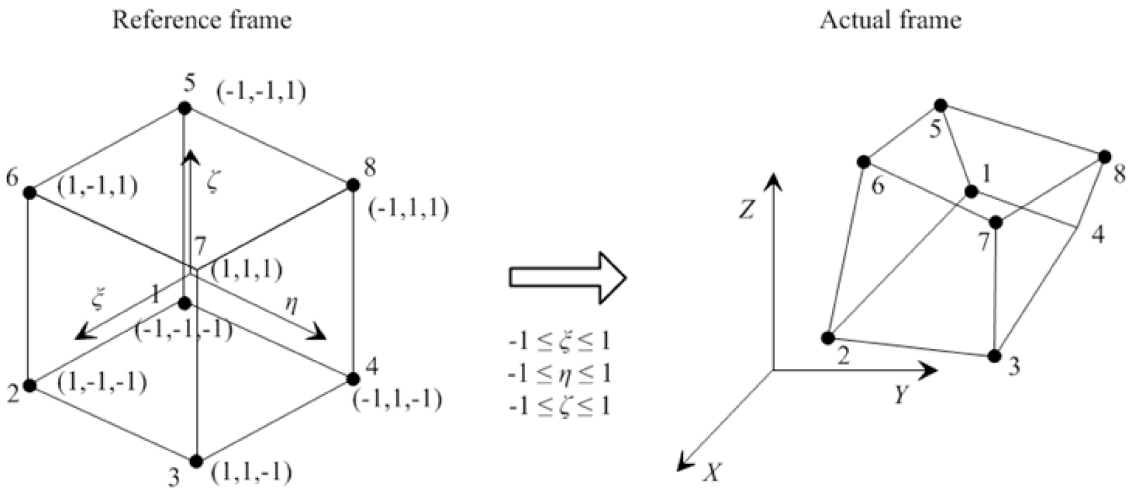
With the SMSFreeze$\left[\text{exp, "Dependency" -> }\left\{\left\{p_1, \frac{\partial exp}{\partial p_1}\right\}, \left\{p_2, \frac{\partial exp}{\partial p_2}\right\}, \ldots, \left\{p_n, \frac{\partial exp}{\partial p_n}\right\}\right\}\right]$ the true dependencies of the *exp* are ignored and it 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 $\frac{\partial exp}{\partial p_1}, \frac{\partial exp}{\partial p_n}, \ldots, \frac{\partial exp}{\partial p_n}$ (see also SMSFreeze).

Example Type C: Implicit dependencies

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

$\boldsymbol{\xi} = \{\xi, \eta, \zeta\}$	reference coordinates
$\mathbf{X}(\boldsymbol{\xi}) = \sum_k N(\boldsymbol{\xi})_k \mathbf{X}_k$	actual coordinates
$\mathbf{u}(\boldsymbol{\xi}) = \sum_k N(\boldsymbol{\xi})_k \mathbf{u}_k$	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 $X = \{X_g, Y_g, Z_g\}$ 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]
```

$$\{\{\boxed{Ia_{11}}, \boxed{Ia_{12}}, \boxed{Ia_{13}}\}, \{\boxed{Ia_{21}}, \boxed{Ia_{22}}, \boxed{Ia_{23}}\}, \{\boxed{Ia_{31}}, \boxed{Ia_{32}}, \boxed{Ia_{33}}\}\}$$

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

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

$$\{\{u_{1\cdot x_1}, u_{1\cdot x_2}, u_{1\cdot x_3}\}, \{u_{2\cdot x_1}, u_{2\cdot x_2}, u_{2\cdot x_3}\}, \{u_{3\cdot x_1}, u_{3\cdot x_2}, u_{3\cdot x_3}\}\}$$

Example 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: 993
Methods	No.Formulae	No.Leafs
Test	6	51

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

```
!************************************************************
!* AceGen    3.102 Windows (11 Jun 11)                      *
!*           Co. J. Korelc  2007            17 Jul 11 13:16:56*
!************************************************************
! User : USER
! Evaluation time                 : 0 s     Mode  : Optimal
! Number of formulae              : 6       Method: Automatic
! Subroutine                      : Test size :51
! Total size of Mathematica  code : 51 subexpressions
! Total size of Fortran code      : 424 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(5)=-dsin(x)
      v(2)=dcos(x)
      IF(dabs(v(2)).gt.0.1d-9) THEN
       v(6)=2d0*(v(2)*v(2))
       v(8)=v(5)*(4d0*dcos(v(6))-dsin(v(6))/v(2)**2)
      ELSE
       v(8)=2d0*v(5)
      ENDIF
      dfdx=v(8)
      END
```

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 $N_{d.o.f} + \left(N_{d.o.f}\right)^2$ 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 $\boldsymbol{a_i}$, is defined by

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

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

$$\boldsymbol{a_i} \cap \boldsymbol{a}_j = \phi \quad ,\mathrm{i} \neq \mathrm{j}.$$

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

$$\frac{\partial f}{\partial \boldsymbol{a}} = \left\{ \frac{\partial f}{\partial \boldsymbol{a}_1}, \frac{\partial f}{\partial \boldsymbol{a}_2}, \ldots \frac{\partial f}{\partial \boldsymbol{a}_L} \right\}$$

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

$$\frac{\partial f}{\partial \overline{\boldsymbol{a}}_{\mathrm{ij}}} = \mathrm{SMSD}[f, \boldsymbol{a_i}, j].$$

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

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

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 $\boldsymbol{u_i}$ and $\boldsymbol{v_i}$ of the set of independent variables $\boldsymbol{a_i}$.

$\boldsymbol{a_i}$=$\{u_{1,}, v_{1,} u_2, v_2, u_3, v_3, u_4, v_4\}$

$\boldsymbol{u_i} = \{u_1, u_2, u_3, u_4\}$, $\boldsymbol{v_i} = \{v_1, v_2\, v_3, v_4\}$

```
<< 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 */
};
```

Non-local Operations

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

Non-local operations have the following characteristics:

⇒ symbolic integration, factorization, nonlinear equations,

⇒ the entire expression has to be considered to get a solution,

⇒ all the relevant variables have to be explicitly “visible”.

Local operations have the following characteristics:

⇒ differentiation, evaluation, linear system of equations,

⇒ operation can be performed on parts of the expression,

⇒ relevant variables can be part of already optimized code.

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

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

```
<< 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
```

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

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.

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

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

The following vector of auxiliary variables is created.

```
SMSShowVector[]
```

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

```
fint = ∫ fx dx
```

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

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

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

$$x\,v^2 + \S3 + v\,\S4$$

At the end of the *Mathematica* session, the global vector of formulae contains the following auxiliary variables:

```
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, {}]
```

See also: SMSSmartRestore `SMSSmartReduce`

Arrays

AceGen has no prearranged matrix, vector, or tensor operations. 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:

```
<< 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.

```
X ⊢ Table[SMSReal[x$$[i]], {i, 5}]
```

$$\{X_1, X_2, X_3, X_4, X_5\}$$

Here the third component of the vector X is displayed.

```
X[[3]]
```

$$X_3$$

The use of AceGen arrays is in this case unnecessary.

```
SMSPart[X, 3]
```

```
Array index is integer number. Consider
   using the standard Mathematica arrays instead
   of AceGen array objects. See also: Arrays
```

X_3

Note that an arbitrary symbol cannot be used as an index.

```
X[[i]]
```

Part::pspec :
Part specification $V[2, 1] is neither an integer nor a list of integers. ≫

Part::pspec :
Part specification $V[2, 1] is neither an integer nor a list of integers. ≫

Part::pspec : Part specification i is neither an integer nor a list of integers. ≫

General::stop :
Further output of Part::pspec will be suppressed during this calculation. ≫

$\{X_1, X_2, X_3, X_4, X_5\}[[i]]$

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 the optimization of the code is 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 vector
- SMSPart - take an arbitrary element of the vector
- SMSReplacePart - replace an arbitrary element of the vector. (WARNING: Currently, differentiation can not be performed with respect to the arrays with the elements that have been changed with the SMSReplacePart !! 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 introductions of a general array (e.g. V ∋ SMSArray[10] allocates memory for the real array with length 10). General arrays HAVE TO BE introduced as a new multi-valued auxiliary variables.

```
<< AceGen`;
SMSInitialize["test", "Language" -> "C"];
SMSModule["test", Real[x$$[5]], Integer[i$$]];
x ⊢ SMSReal[x$$]; i ⊢ SMSInteger[i$$];
```

Here a constant *AceGen* array is created. Result is a single auxiliary variable.

```
X ⊢ SMSArray[Table[SMSReal[x$$[k]], {k, 5}]]
```

X

In this case, the third component of Sin[X] cannot be accessed by *Mathematica* since X is a symbol not an array.

```
X[[3]]
```

Part::partw : Part 3 of $V[3, 1] does not exist. ≫

Part::partw : Part 3 of $V[3, 1] does not exist. ≫

Part::partw : Part 3 of X does not exist. ≫

X[[3]]

However, one can access the i-th component of X. 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.

```
SMSPart[X, i]
```

X_i

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.

■ Example : Arrays

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.

```
<< AceGen`;
SMSInitialize["test", "Language" -> "Fortran"];
SMSModule["test", Real[x$$, r$$, s$$, t$$], Integer[n$$, m$$]];
x ⊢ SMSReal[x$$];
n ⊢ SMSInteger[n$$]; m ⊢ SMSInteger[m$$];
```

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 (*G*[...]) representing the whole array.

```
SMSArray[{x, x^2, 0, π}]
```

$\mathbb{G}[x, x^2, 0, \pi]$

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.

```
A ⊨ SMSArray[{x, x^2, 0, π}]
```

A

This creates the second *AceGen* array object with constant contents.

```
B ⊨ SMSArray[{3 x, 1 + x^2, Sin[x], Cos[x π]}]
```

B

This calculates a dot product of vectors A and B

```
dot = SMSDot[A, B]
```

A • B

This creates an index to the *n*-th element of the second vector.

```
Bn = SMSPart[B, n]
```

B_n

This allocates space on the global vector of formulae and creates a general *AceGen* array object *V*. The values of the vector V are **NOT INITIALIZED**.

```
V ⊣ SMSArray[10]
```

$_1V$

This sets the elements of the *V* array to be equal $V_i = \frac{1}{i}$, $i = 1, 2, \ldots, 10$.

```
SMSDo[
  V ⊣ SMSReplacePart[V, 1 / i, i];
  , {i, 1, 10, 1, V}];
V
```

$_4V$

This creates an index to the *m*-th element of the *V* array.

```
Vm = SMSPart[V, m]
```

$_4V_m$

```
SMSExport[{dot, Bn, Vm}, {r$$, s$$, t$$}];
SMSWrite["test"];
```

File:	test.f	Size: 1135
Methods	No.Formulae	No.Leafs
test	6	96

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

```
!*************************************************************
!* AceGen    2.502 Windows (18 Nov 10)                       *
!*           Co. J. Korelc  2007            24 Nov 10 13:08:14*
!*************************************************************
! User : USER
! Evaluation time                 : 0 s     Mode  : Optimal
! Number of formulae              : 6       Method: Automatic
! Subroutine                      : test size :96
! Total size of Mathematica  code : 96 subexpressions
! Total size of Fortran code      : 560 bytes

!****************** S U B R O U T I N E *********************
      SUBROUTINE test(v,x,r,s,t,n,m)
      IMPLICIT NONE
      include 'sms.h'
      INTEGER n,m,i8
      DOUBLE PRECISION v(5023),x,r,s,t
      v(5)=x**2
      v(5000)=x
      v(5001)=v(5)
      v(5002)=0d0
      v(5003)=0.3141592653589793d1
      v(5004)=3d0*x
      v(5005)=1d0+v(5)
      v(5006)=dsin(x)
      v(5007)=dcos(0.3141592653589793d1*x)
      DO i8=1,10
       v(5007+i8)=1d0/i8
      ENDDO
      r=SMSDot(v(5000),v(5004),4)
      s=v(5003+int(n))
      t=v(5007+int(m))
      END
```

Run Time Debugging

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 name	*default value*	
"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 for *SMSSetBreak*.

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[*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[*breakID*]	disable break point with the break identification *breakID*
SMSClearBreak[" Default "]	set all options to default values
SMSClearBreak[]	disable all break points

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 name	*default value*	
"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 for SMSActivateBreak.

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]).

See also: AceGen Palettes , Interactive Debugging, AceFEM Structure, User Interface

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.

```
<< 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[];
```

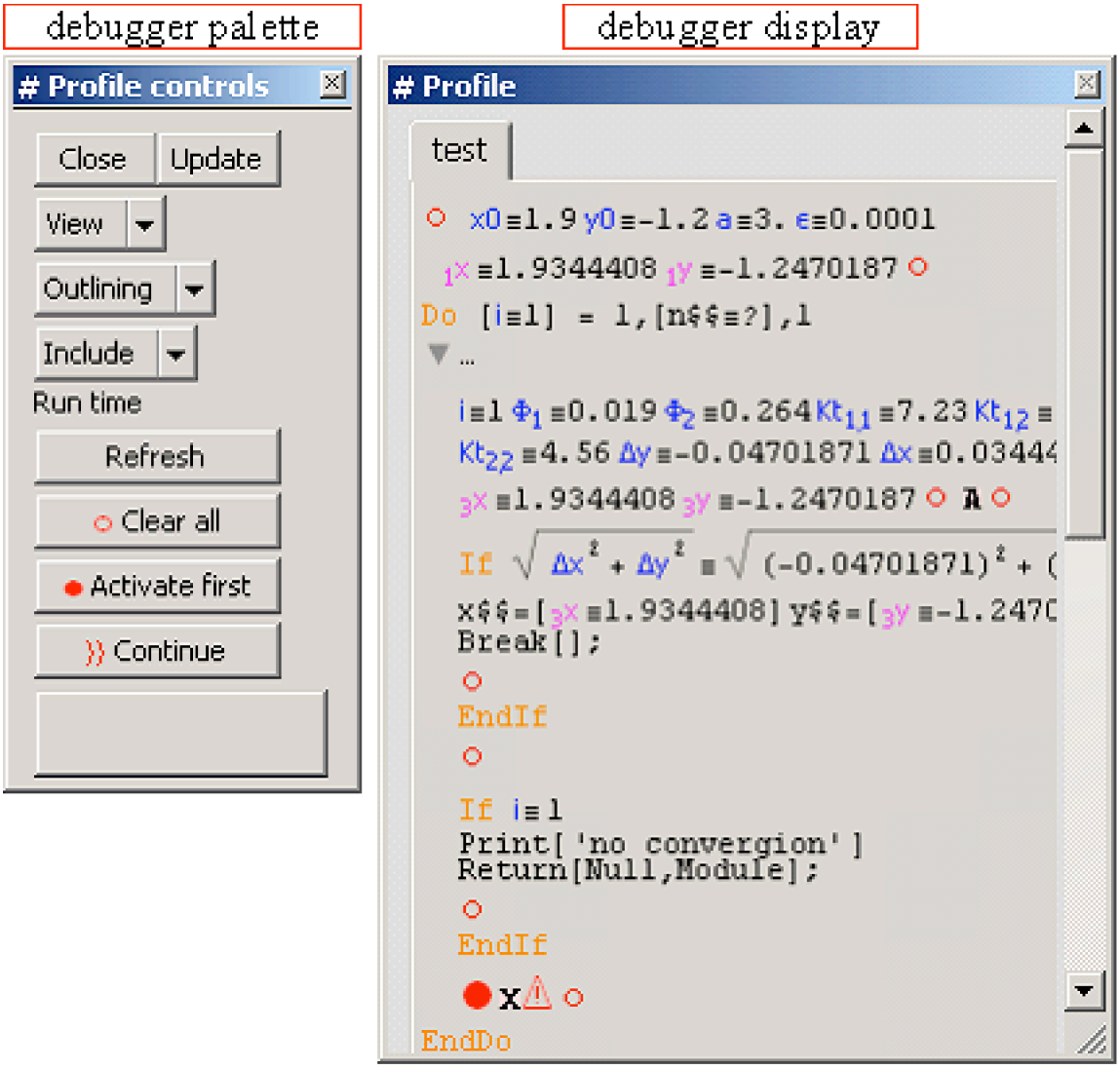
```
time=0  variable= 0 ≡ {}

[0]  Consistency check - global

[0]  Consistency check - expressions

[0]  Generate source code :

Events: 0

[0] Final formating
```

File:	test.m	Size: 2491
Methods	No.Formulae	No.Leafs
test	33	198

Here the program is loaded and the generated subroutine is called.

```
<< 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 additionaly 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).

```
<< 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]
```

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

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

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

User Defined Functions

The user can define additional output formats for standard Mathematica functions or new functions. The advantage of properly defined function is that allows optimization of expressions and automatic differentiation. In general there are several types of user defined functions supported in AceGen:

1. **Intrinsic user function**: 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.
2. **User AceGen module**: 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.
3. **User 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 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:
   a. 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.

b. 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.

See also: Elements that Call User External Subroutines.

Intrinsic user function 1: Scalar function exists but has different syntax in source code language

```
<< AceGen`;
SMSInitialize["test", "Language" → "Fortran"];
```

This is an additional definition of output format for function tangent.

```
SMSAddFormat[
  Tan[i_] :→ Switch[SMSLanguage,
    "Mathematica", "Tan"[i], "Fortran", "dtan"[i], "C", "tan"[i]]
 ];

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.

```
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].

```
<< AceGen`;
SMSInitialize["test", "Language" → "C"];
```

This is an additional definition of output format for function MyPower.

```
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.

```
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.

```
N[MyPower[i_, j_], p_] := i^j;

SMSModule["sub1", Real[x$$, y$$, z$$]];
x ⊢ SMSReal[x$$];
y ⊢ SMSReal[y$$];

SMSExport[SMSD[MyPower[x, y], x], z$$];

SMSWrite[];
```

File:	test.c	Size: 729
Methods	No.Formulae	No.Leafs
sub1	1	22

```
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

```
<< AceGen`;
SMSInitialize["test", "Language" → "C"];
```

This generates **user AceGen module** $f = \text{Sin}(a_1\, x + a_2\, x\text{^}2 + a_3\, x^3)$ 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.

```
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 *main* that calls subroutine *f*.

```
SMSModule["main", 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.

```
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 (see SMSCall).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* .

```
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.

```
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.

```
f ⊢ SMSReal[y$$, "Subordinate" → fo, "Dependency" -> {z, dfdz}];
```

First derivatives are derived and displayed here.

```
dw ⊨ SMSD[f, w];
SMSRestore[dw, "Global"]
```

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.

```
ddw = SMSD[dw, w];
SMSRestore[ddw, "Global"]
```

2 dfdz

```
SMSExport[dw, dy$$];
SMSWrite[];
```

File:	test.c	Size: 1189
Methods	No.Formulae	No.Leafs
f	4	78
main	4	42

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

```
/*************************************************************
* AceGen    2.502 Windows (18 Nov 10)                        *
*           Co. J. Korelc  2007            24 Nov 10 13:18:30*
*************************************************************
User : USER
Evaluation time                 : 0 s     Mode  : Optimal
Number of formulae              : 8       Method: Automatic
Subroutine                      : f size :78
Subroutine                      : main size :42
Total size of Mathematica  code : 120 subexpressions
Total size of C code            : 565 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[7]=Power((*x),2);
v[6]=a[1]*v[7]+a[0]*(*x)+a[2]*Power((*x),3);
(*y)=sin(v[6]);
(*dy)=(a[0]+3e0*a[2]*v[7]+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 **user AceGen module** $f = \text{Sin}(a_1\, x + a_2\, x\text{^}2 + a_3\, x^3)$ 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.

```
<< 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.

```
SMSRestore[{dw, ddw}, "Global"]
```

$$\{2\ \text{dfdz}\ \text{w},\ 2\ \text{dfdz} + 4\ \text{dfdz2}\ \text{w}^2\}$$

```
SMSExport[{dw, ddw}, {dy$$, ddy$$}];

SMSWrite[];
```

File:	test.c	Size: 1317
Methods	No.Formulae	No.Leafs
f	4	82
main	6	73

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

```
/************************************************************
* AceGen    2.502 Windows (18 Nov 10)                       *
*           Co. J. Korelc  2007            24 Nov 10 13:16:18*
************************************************************
User : USER
Evaluation time                 : 0 s     Mode  : Optimal
Number of formulae              : 10      Method: Automatic
Subroutine                      : f size :82
Subroutine                      : main size :73
Total size of Mathematica  code : 155 subexpressions
Total size of C code            : 687 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[7]=Power((*x),2);
v[6]=a[1]*v[7]+a[0]*(*x)+a[2]*Power((*x),3);
v[8]=(a[0]+3e0*a[2]*v[7]+2e0*a[1]*(*x))*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[20]=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[20];
(*ddy)=2e0*v[18]+ddy*(v[20]*v[20]);
};
```

User external subroutines 1: Source code file incorporated into the generated source code

Lets create the C source file "Energy.c" with the following contents

```
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"]
```

```
Energy.c
```

and the C header file "Energy.h" with the following contents

```
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 incorporated into generated source code.

```
<< 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$$, "Subordinate" → pcall,
   "Dependency" → {{I1, dp[[1]]}, {I3, dp[[2]]}}];
S ⊨ 2 SMSD[Π, 𝕮];
SMSExport[S, S$$];
SMSWrite["Splice" -> {"Energy.c"}];
```

File:	test.c	Size: 2051
Methods	No.Formulae	No.Leafs
Stress	18	353

FilePrint["test.c"]

```
/*************************************************************
* AceGen    2.502 Windows (18 Nov 10)                        *
*           Co. J. Korelc  2007            24 Nov 10 13:26:17*
*************************************************************
User : USER
Evaluation time                 : 0 s     Mode  : Optimal
Number of formulae              : 18      Method: Automatic
Subroutine                      : Stress size :353
Total size of Mathematica  code : 353 subexpressions
Total size of C code            : 985 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[36]=Power(C[0][1],2);
v[40]=C[0][0]*C[1][1]-v[36];
v[33]=Power(C[0][2],2);
v[30]=2e0*C[0][2]*C[1][2];
v[28]=Power(C[1][2],2);
v01=C[0][0]+C[1][1]+C[2][2];
v02=-(C[0][0]*v[28])+C[0][1]*v[30]-C[1][1]*v[33]+C[2][2]*v[40];
v03=(*C1);
v04=(*C2);
v05=(*C3);
Energy(&v01,&v02,&v03,&v04,&v05,&pi,dp,ddp);
v[25]=dp[0];
v[26]=dp[1];
v[39]=4e0*v[26];
v[31]=2e0*v[26]*(-2e0*C[0][1]*C[2][2]+v[30]);
v[32]=(-(C[0][2]*C[1][1])+C[0][1]*C[1][2])*v[39];
v[35]=(C[0][1]*C[0][2]-C[0][0]*C[1][2])*v[39];
S[0][0]=2e0*(v[25]+v[26]*(C[1][1]*C[2][2]-v[28]));
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[33]));
S[1][2]=v[35];
S[2][0]=v[32];
S[2][1]=v[35];
S[2][2]=2e0*(v[25]+v[26]*v[40]);
};
```

User external subroutines 2: Header file incorporated into the generated source code

Previous example is here modified in a way that only the header file "Energy.h" is incorporated into generated source code.

```
<< 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$$, "Subordinate" → pcall,
   "Dependency" → {{I1, dp[[1]]}, {I3, dp[[2]]}}];
S ⊨ 2 SMSD[Π, ℂ];
SMSExport[S, S$$];
SMSWrite["IncludeHeaders" -> {"Energy.h"}];
```

File:	test.c	Size: 1596
Methods	No.Formulae	No.Leafs
main	18	353

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

```
/*************************************************************
* AceGen    2.502 Windows (18 Nov 10)                        *
*           Co. J. Korelc  2007            24 Nov 10 13:28:00*
*************************************************************
User : USER
Evaluation time                 : 1 s     Mode  : Optimal
Number of formulae              : 18      Method: Automatic
Subroutine                      : main size :353
Total size of Mathematica  code : 353 subexpressions
Total size of C code            : 983 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[36]=Power(C[0][1],2);
v[40]=C[0][0]*C[1][1]-v[36];
v[33]=Power(C[0][2],2);
v[30]=2e0*C[0][2]*C[1][2];
v[28]=Power(C[1][2],2);
v01=C[0][0]+C[1][1]+C[2][2];
v02=-(C[0][0]*v[28])+C[0][1]*v[30]-C[1][1]*v[33]+C[2][2]*v[40];
v03=(*C1);
v04=(*C2);
v05=(*C3);
Energy(&v01,&v02,&v03,&v04,&v05,&pi,dp,ddp);
v[25]=dp[0];
v[26]=dp[1];
v[39]=4e0*v[26];
v[31]=2e0*v[26]*(-2e0*C[0][1]*C[2][2]+v[30]);
v[32]=(-(C[0][2]*C[1][1])+C[0][1]*C[1][2])*v[39];
v[35]=(C[0][1]*C[0][2]-C[0][0]*C[1][2])*v[39];
S[0][0]=2e0*(v[25]+v[26]*(C[1][1]*C[2][2]-v[28]));
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[33]));
S[1][2]=v[35];
S[2][0]=v[32];
S[2][1]=v[35];
S[2][2]=2e0*(v[25]+v[26]*v[40]);
};
```

Symbolic Evaluation

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.

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*$_i$

At the output the *SMSReplaceAll* function gives $\exp|_{v_1=\text{new}_1, v_2=\text{new}_2, \ldots}$. 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,

$$F(x) = F(x)\,|_{x=x_0} + \frac{\partial F(x)}{\partial x}\,|_{x=x_0}\ (x - \mathrm{x0})\ ,$$

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

```
<< 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.

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

z

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

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

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

Example 3: repeated use of SMSReplaceAll

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

```
<< 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.

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

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". The signatures of the expressions are basis for the heuristic algorithm that can search also for some higher relations among the expressions. The relations between expressions which are automatically recognized by the *AceGen* system are:

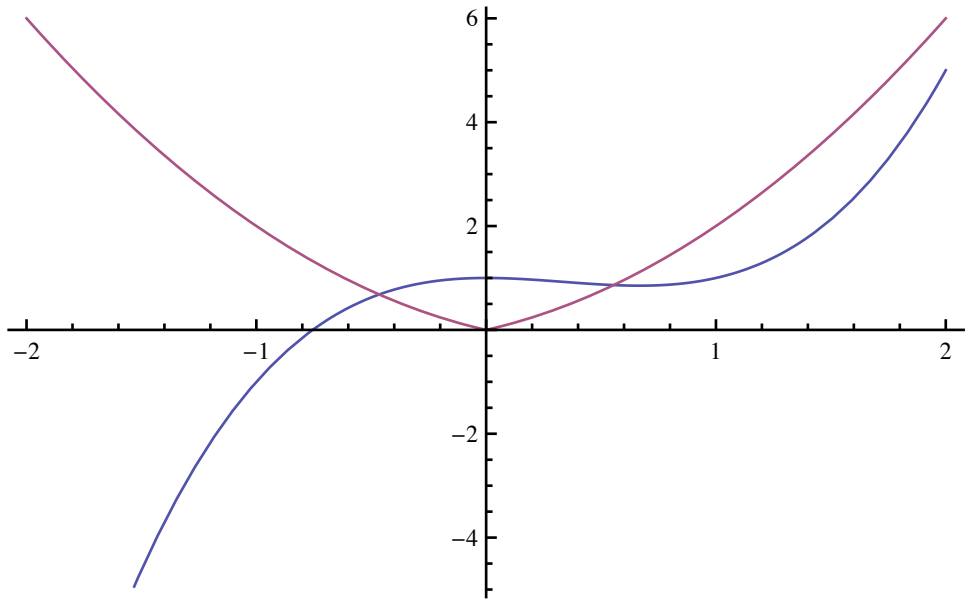
	description	simplification
(a)	two expressions or sub–expressions are the same	$e_1 \equiv e_2 \Longrightarrow \begin{cases} v_1 := e_1 \\ e_2 \Rightarrow v_1 \end{cases}$
(b)	result is an integer value	$e_1 \equiv Z \Longrightarrow \quad e_1 \Rightarrow Z$
(c)	opposite value	$e_1 \equiv -e_2 \Longrightarrow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

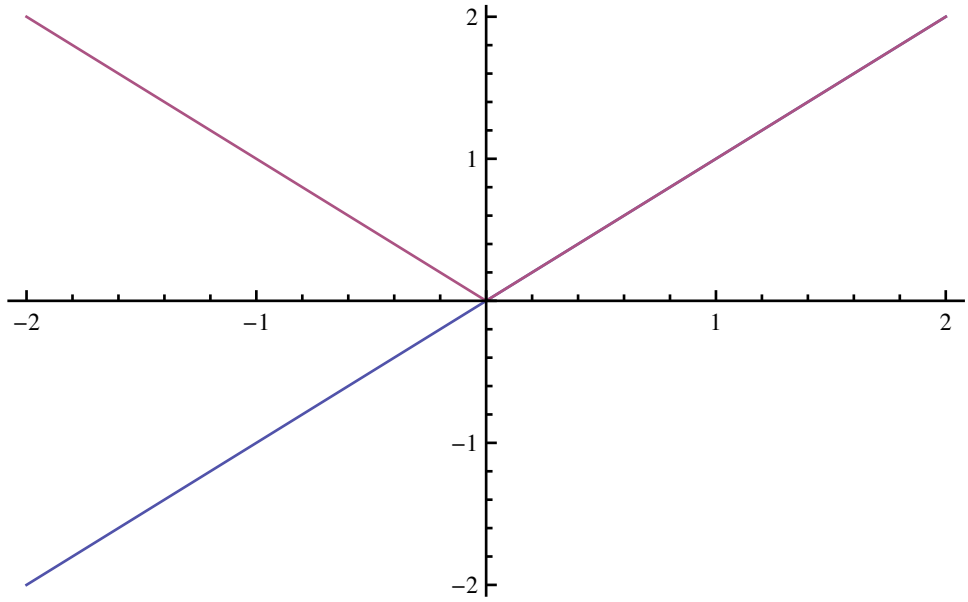
Let us first consider the two functions $f_1 = x^3 - x^2 + 1$ and $f_2 = \mathrm{Abs}[x] + x^2$.

```
Plot[{x^3 - x^2 + 1, Abs[x] + x^2}, {x, -2, 2}]
```



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 $f_2 = \mathrm{Abs}[x]$.

```
Plot[{x, Abs[x]}, {x, -2, 2}]
```



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.

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 as SMSAbs, SMSSqrt, SMSSign. For all other cases we can wrap critical function with the general unique signature function SMSFreeze.

Differentiation (Automatic Differentiation, SMSD) 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 *FORTRAN* 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)]).

See also: `Signatures of the Expressions`

Signatures of the Expressions

The input parameters of the subroutine (independent variables) have assigned a randomly generated high precision real number or an ***unique signature***. The signature of the dependent auxiliary variables is obtained by replacing all auxiliary variables in the definition of variable with corresponding signatures and is thus deterministic. The randomly generated high precision real numbers assigned to the input parameters of the subroutine can have in some cases effects on code optimization procedure or even results in wrong code. One reason for the incorrect optimization of the expressions is presented in section `Expression Optimization`. Two additional reasons for wrong simplification are round-off errors and hidden patterns inside the sets of random numbers. In *AceGen* we can use randomly generated numbers of arbitrary precision, so that we can exclude the possibility of wrong simplifications due to the round-off errors. *AceGen* also combines several different random number generators in order to minimize the risk of hidden patterns inside the sets of random numbers.

The precision of the randomly generated real numbers assigned to the input parameters is specified by the "Precision" option of the SMSInitialize function. Higher precision would slow down execution.

In rare cases user has to provide 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, SMSAbs, SMSSqrt) 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	default signature

Evaluation codes for the generation of the signature.

■ 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 increase precision with the *SetPrecision[exp,SMSEvaluatePrecision]* command.

```
<< AceGen`;
SMSInitialize["test", "Language" -> "Mathematica", "Mode" -> "Debug"];
SMSModule["test"];
```

```
time=0  variable= 0 ≡ {}
```

```
x ⊢ π;
```

```
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.

```
<< 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.

```
x ⊢ SMSReal[x$$];
y ⊢ SMSReal[y$$, {-100, 100}];
n ⊢ SMSInteger[n$$];
```

This displays the signatures of external variables x, y, and n .

```
{x, y, n} // SMSEvaluate // Print
```

```
{0.5126296357476784249473038651688948998 75,
 47.741233930866187312379418124941701519 2,
 4.641185606823421179019188449964375656913}
```

Linear Algebra

Enormous growth of expressions typically appears when the SAC systems such as *Mathematica* are used directly for solving 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. In all linear algebra routines it is assumed that the solution exist ($det(A)\neq 0$).

SMSLinearSolve[A,B]	generate the code sequence that solves the system of linear equations $A\ x= B$ analytically and return the solution vector

Parameter A is a square matrix. Parameter B can be a vector (one right-hand side) or a matrix (multiple right-hand sides). The Gauss elimination procedure is used without pivoting.

SMSLUFactor[A]	the LU decomposition along with the pivot list of M

The Gauss elimination procedure is used and simultaneous simplification is performed during the process. The *SMSLU-Factor* 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.

SMSLUSolve[LU,B]	solution of the linear system represented by LU and right−hand side 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[*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[*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[*M*]	the determinant of square matrix *M*

Simultaneous simplification is performed during the process.

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.

Example

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

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

```
<< 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

```
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];
};
```

Tensor Algebra

SMSCovariantBase[$\{\phi_1,\phi_2,\phi_3\},\{\eta_1,\eta_2,\eta_3\}$]	the covariant base vectors of transformation from the coordinates $\{\eta_1,\eta_2,\eta_3\}$ to coordinates $\{\phi_1,\phi_2,\phi_3\}$

Transformations 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

```
<< 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
```

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

SMSCovariantMetric[$\{\phi_1,\phi_2,\phi_3\}$, $\{\eta_1,\eta_2,\eta_3\}$]	the covariant metrix tensor of transformation from coordinates $\{\eta_1,\eta_2,\eta_3\}$ to coordinates $\{\phi_1,\phi_2,\phi_3\}$

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

```
<< 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
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & r^2 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

SMSContravariantMetric[$\{\phi_1,\phi_2,\phi_3\}$, $\{\eta_1,\eta_2,\eta_3\}$]	the contravariant metrix tensor of transformation from coordinates $\{\eta_1,\eta_2,\eta_3\}$ to coordinates $\{\phi_1,\phi_2,\phi_3\}$

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

```
<< 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
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{r^2} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

SMSChristoffell1[$\{\phi_1,\phi_2,\phi_3\}$, $\{\eta_1,\eta_2,\eta_3\}$]	the first Christoffell symbol {i,j,k} of transformation from coordinates $\{\eta_1,\eta_2,\eta_3\}$ to coordinates $\{\phi_1,\phi_2,\phi_3\}$

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

```
<< 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
```

$$\begin{pmatrix} \begin{pmatrix}0\\0\\0\end{pmatrix} & \begin{pmatrix}0\\r\\0\end{pmatrix} & \begin{pmatrix}0\\0\\0\end{pmatrix} \\ \begin{pmatrix}0\\r\\0\end{pmatrix} & \begin{pmatrix}-r\\0\\0\end{pmatrix} & \begin{pmatrix}0\\0\\0\end{pmatrix} \\ \begin{pmatrix}0\\0\\0\end{pmatrix} & \begin{pmatrix}0\\0\\0\end{pmatrix} & \begin{pmatrix}0\\0\\0\end{pmatrix} \end{pmatrix}$$

SMSChristoffell1[$\{\phi_1,\phi_2,\phi_3\}$, $\{\eta_1,\eta_2,\eta_3\}$]	the second Christoffell symbol Γ^k_{ij} of transformation from coordinates $\{\eta_1,\eta_2,\eta_3\}$ to coordinates $\{\phi_1,\phi_2,\phi_3\}$

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

```
<< AceGen`;
SMSInitialize["test", "Language" -> "Mathematica"];
SMSModule["test"];
{r, ϕ, z} ⊨ Array[SMSFictive[] &, {3}];
SMSChristoffell2[{r Cos[ϕ], r Sin[ϕ], z}, {r, ϕ, z}] // 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 coorinates *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,ϕ,z} into Cartesian coordinates.

```
<< 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}]
```

$$\{\mathrm{Cos}[\phi]\, r^2 + r\, \mathrm{Sin}[\phi]^2,\ \mathrm{Cos}[\phi]\, \mathrm{Sin}[\phi] - r\, \mathrm{Sin}[\phi],\ r\, z\}$$

SMSDCovariant[*tensor*, *transf*, *coord*, *index_types*]	covariant derivative of arbitrary tensor field *tensor* with indices *index_* 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,ϕ,z}.

```
<< 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
```

$$\begin{pmatrix} 2\, r & -r^2\, \mathrm{Sin}[\phi] & 0 \\ 2\, \mathrm{Sin}[\phi]\ r & r + \mathrm{Cos}[\phi]\ r & 0 \\ z & 0 & r \end{pmatrix}$$

Mechanics of Solids

Mechanics of solids functions:

`SMSLameToHooke` `SMSHookeToLame` . `SMSHookeToBulk` . `SMSBulkToHooke` . `SMSPlaneStressMatrix` . `SMSPlaneStrainMatrix` . `SMSEigenvalues` . `SMSMatrixExp` . `SMSInvariantsI` . `SMSInvariantsJ`

Bibliography

KORELC, Joze. Semi-analytical solution of path-independed nonlinear finite element models. Finite elem. anal. des., 2011, 47:281-287.

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

Korelc J., (2002), Multi-language and Multi-environment Generation of Nonlinear Finite Element Codes, Engineering with Computers, 2002, vol. 18, n. 4,str. 312-327

Korelc, J. (1997a), Automatic generation of finite-element code by simultaneous optimization of expressions, *Theoretical Computer Science,* **187,** 231-248.

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.

Korelc J. (1996), Symbolic Approach in Computational Mechanics and its Application to the Enhanced Strain Method, Doctoral Dissertation, Institut of Mechanics, TH Darmstadt, Germany.

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.

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.

Wolfram, S. (1991), *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley.

Numerical Environments Tutorials

Finite Element Environments Introduction

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 environmen. *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©* is the commercial environment written in FORTRAN (see `ELFEN`).

⇒ *ABAQUS©* is the commercial environment written in FORTRAN (see `ABAQUS`).

The AceGen package is often used to generate user subroutines for various other environments. It is advisable for the user to use standardized interface as described in User defined environment interface .

There are several benefits of using the standardized interface:

⇒ automatic translation to other FE packages,

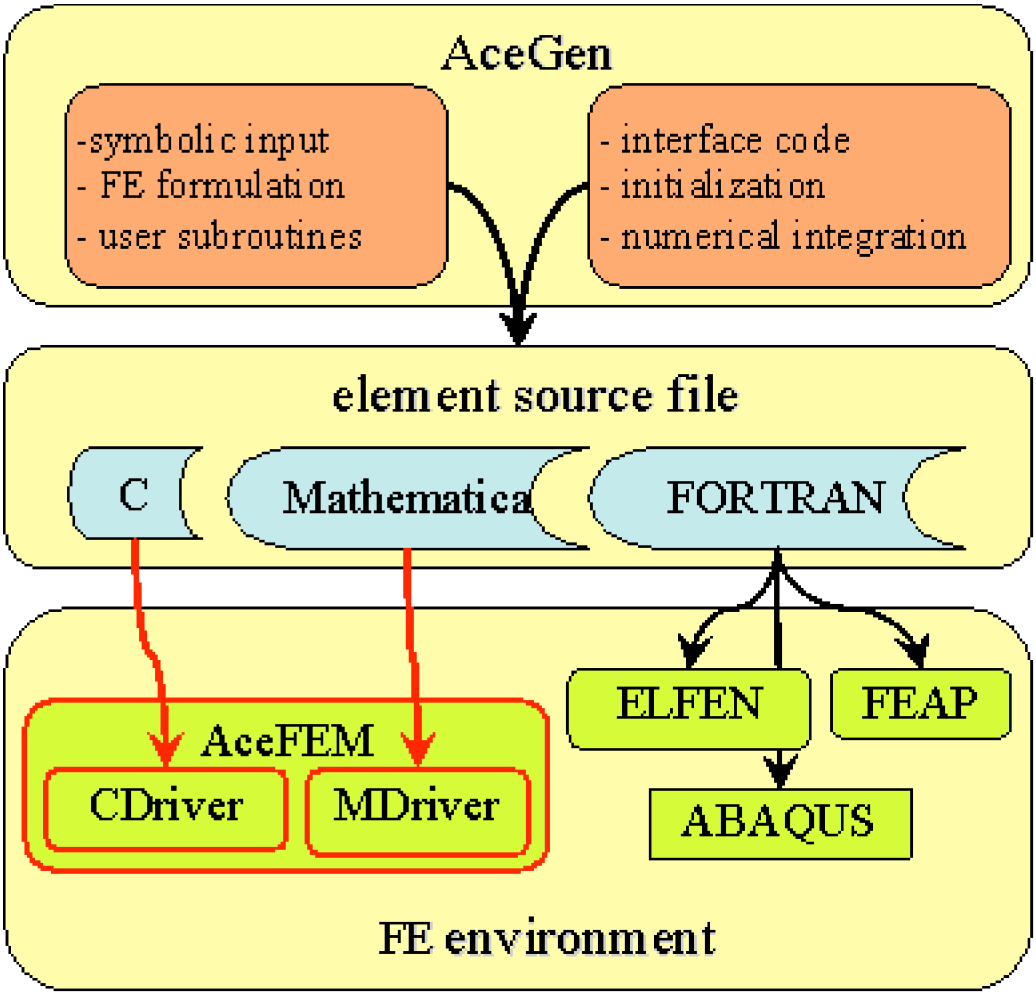
⇒ other researchers are able to repeat the results,

⇒ commercialization of the research is easier,

⇒ eventually, the users interface can be added to the list of standard interfaces.

The number of numercal environments supported by AceGen system is a growing daily. Please visit the www.fgg.uni-lj.si/symech/extensions/ page to see if the numerical environment you are using is already supported or www.fgg.uni-lj.si/consulting/ to order creation of the interface for your specific environment.

All FE environments are essentially treated in the same way. Additional interface code ensures proper data passing to and from automatically generated code for those systems. Interfacing the automatically generated code and FE environment is a two stage process. The purpose of the process is to generate element codes for various languages and environments from the same symbolic input. At the first stage user subroutine codes are generated. Each user subroutine performs specific task (see SMSStandardModule). The input/output arguments of the generated subrutines 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 SMSStandardModule). There are several possibilities in the case of need for an additional functionality. Standard user subroutines can be used as templates by giving them a new name and, if necessary, additional arguments. The additional subroutines can be called directly from the environment or from the enhanced "*splice-file*". Source code of the "*splice-files*" for all supported environments are available at directory $BaseDirectory/Applications/AceGen/Splice/. The additional subroutines can be generated independently just by using the *AceGen* code generator and called directly from the environment or from the enhanced "*splice-file*".

Since the complexity of the problem description mostly appears in a symbolic input, we can keep the number of data structures (see 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 Data , Real Type Environment Data),

⇒ nodal data structure contains all the data that is associated with the node (see Node Data),

⇒ element specification data structure contains information common for all elements of particular type (see Domain Specification Data),

⇒ node specification data structure contains information common for all nodes of particular type (see Node Specification Data),

⇒ element data structure contains all the data that is associated with the specific element (see Element Data).

Standard FE Procedure

Description of FE Characteristic Steps

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

A) **AceGen initialization**

- see SMSInitialize

B) **Template initialization**

- see SMSTemplate
- general characteristics of the element
- rules for symbolic-numeric interface

C) **Definition of user subroutines**

- see SMSStandardModule
- tangent matrix, residual, postprocessing, …

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, postprocessing etc.. The problem considered is non-linear and it has unsymmetric Jacobian matrix.

Step 1: Initialization

- This loads the *AceGen* code generator.

```
<< AceGen`;
```

- This initializes the *AceGen* session. The *AceFEM* is chosen as the target numerical environment. See also SMSInitialize.

```
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.

```
SMSTemplate["SMSTopology" → "H1", "SMSDOFGlobal" → 1,
  "SMSSymmetricTangent" → False,
  "SMSGroupDataNames" ->
   {"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 SMSStandardModule).

```
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.

```
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 the case that material characteristic vary substantially over the domain it is better to use element data field "Data" instead of element specification data.

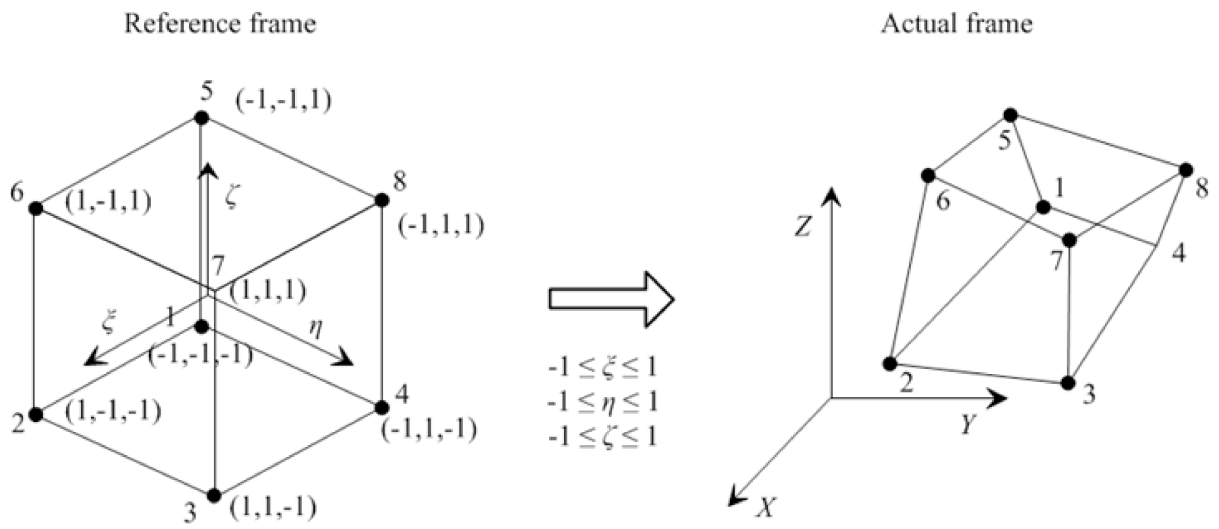
```
{k0, k1, k2, Q} ⊢ SMSReal[Table[es$$["Data", i], {i, 4}]];
```

- 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.

```
SMSDo[Ig, 1, SMSInteger[es$$["id", "NoIntPoints"]]];
Ξ = {ξ, η, ζ} ⊢ Table[SMSReal[es$$["IntPoints", i, Ig]], {i, 3}];
```

Step 4: Definition of the trial functions

- This defines the trilinear shape functions N_i, $i=1,2,\ldots,8$ 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.




```
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];
```

- 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.

```
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.

```
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"]*.

```
k ⊨ k0 + k1 ϕ + k2 ϕ^2;
λ ⊢ SMSReal[rdata$$["Multiplier"]];

wgp ⊢ SMSReal[es$$["IntPoints", 4, 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 SMSStandardModule).

```
SMSExport[SMSResidualSign Rg, p$$, "AddIn" → True];
```

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.

```
Kg ⊨ SMSD[Rg, ϕI];
SMSExport[Kg, s$$, "AddIn" → True];
```

This is the end of the integration loop.

```
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 an obligatory, however it makes the pos-processing much easier.

```
SMSStandardModule["Postprocessing"];
```

Here the nodal point post-processing quantitie "Temperature" is introduced and exported to array of the nodal point quantities *npost$$*.

```
ϕI ⊢ Table[SMSReal[nd$$[i, "at", 1]], {i, 8}];
SMSNPostNames = {"Temperature"};
SMSExport[ϕI, Table[npost$$[i, 1], {i, 8}]];
```

Here the integration point post-processing quantitie "Conductivity" is introduced and exported to array of the integration point quantities *gpost$$*.

```
{k0, k1, k2, Q} ⊢ SMSReal[Table[es$$["Data", i], {i, Length[SMSGroupDataNames]}]];
SMSDo[
  Ξ = {ξ, η, ζ} ⊢ Table[SMSReal[es$$["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

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.

```
SMSWrite[];
```

File:	ExamplesHeatConduction.c	Size: 16 296
Methods	No.Formulae	No.Leafs
SKR	208	4280
SPP	29	312

Template Constants

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 chapter *Problem Solving Environments*).

The template constants are initialized with the SMSTemplate function. Values of the constants can be also set or changed directly after SMSTemplate command.

■ Geometry

Abbreviation	*Description*	Default value
SMSTopology	element topology (see Element Topology)	□
SMSNoNodes	number of nodes	Automatic
SMSAdditionalNodes	pure function (see Function) that returns additional nodes. Arguments of the function are the coordinates of topological nodes given as mesh input data. Example: Hold[{(#1+#2)/2}&] adds one additional node in the middle of nodes 1 and 2. Additional nodes can be topological (see Cubic triangle, Additional nodes) or auxiliary nodes (see Mixed 3 D Solid FE, Auxiliary Nodes).	Hold[{}&]
SMSNodeID	for all nodes a keyword that is used for identification of the nodes in the case of multi–field problems (see Node Identification, Mixed 3 D Solid FE, Auxiliary Nodes)	Array["D"&, SMSNoNodes]
SMSCreateDummyNodes	enable use of dummy nodes (see Node Identification)	False
SMSNoDimensions	number of spatial dimensions	Automatic

Degrees of Freedom, K and R

Abbreviation	*Description*	Default value
SMSDOFGlobal	number of d.o.f per node for all nodes	Array[SMSNoDimensions&, SMSNoNodes]
SMSSymmetricTangent	True ⇒ tangent matrix is symmetric False ⇒ tangent matrix is unsymmetrical	True
SMSNoDOFCondense	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)	0
SMSCondensationData	storage scheme for local condensation (see Elimination of local unknowns)	
SMSResidualSign	1 ⇒ equations are formed in the form $\boldsymbol{K\,a+\Psi=0}$ −1 ⇒ equations are formed in the form $\boldsymbol{K\,a=\Psi}$ (used to ensure compatibility between the numerical environments)	Automatic
SMSDefaultIntegrationCode	default numerical integration code (see Numerical Integration)	Automatic
SMSNoDOFGlobal	total number of global d.o.f.	calculated value
SMSNoAllDOF	total number of all d.o.f.	calculated value
SMSMaxNoDOFNode	maximum number of d.o.f. per node	calculated value

Data Management

Abbreviation	*Description*	Default value
SMSGroupDataNames	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$$["Data",j] data field)	{}
SMSDefaultData	default values for input data values	Table[0., SMSGroupDataNames //Length]
SMSDataCheck	logical expression that checks the correctness of the user supplied constants stored in es$$["Data",*i*] . It should return True if the data is correct.	True
SMSNoTimeStorage	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)	0
SMSNoElementData	total number of arbitrary real values per element (defines the dimension of the ed$$["Data",j] data field)	0
SMSNoNodeStorage	total number of history dependent real type values per node that have to be stored in the memory for transient type of problems (can be different for each node) (defines the dimension of the nd$$[i,"ht",j] and nd$$[i,"hp",j] data fields)	Array[0&, SMSNoNodes]
SMSNoNodeData	total number of arbitrary real values per node (can be different for each node) (defines the dimension of the nd$$[i,"Data",j] data field)	Array[idata$$["NoShapeParameters"] *es$$["id", "NoDimensions"]&, SMSNoNodes]
SMSIDataNames	list of the keywords of additional integer type environment data variables (global)	{}
SMSRDataNames	list of the keywords of additional real type environment data variables (global)	{}
SMSNoAdditionalData	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)	0
SMSCharSwitch	list of character type user defined constants (local)	{}
SMSIntSwitch	list of integer type user defined constants (local)	{}
SMSDoubleSwitch	list of double type user defined constants (local)	{}

Graphics and Postprocessing

Abbreviation	*Description*	Default value
SMSGPostNames	description of the postprocessing quantities defined per material point	{}
SMNPostNames	description of the postprocessing quantities defined per node	{}
SMSSegments	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 postprocessing	Automatic
SMSSegmentsTriangulation	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 postprocessing of field variables) SMSSegments={{{1,2,3},{1,3,4}}} SMSSegments={{}} ... no field postprocessing	Automatic
SMSReferenceNodes	coordinates of the nodes in the reference coordinate system in the case of elements with variable number of nodes (used in post processing)	Automatic
SMSPostNodeWeights	additional weights associated with element nodes and used for postprocessing of the results (see SMTPost). In general, the weight of the nodes that form the segments is 1 and for the others is 0.	Automatic
SMSAdditionalGraphics	pure function (see Function) that is called for each element and returns additional graphics primitives per element SMSAdditionalGraphics[{element index, domain index,list of node indices}, True if node marks are required, True if boundary conditions are required, {node coordinates for all element nodes}] e.g. Hold[{Line[{#4[[1]],#4[[2]]}]}&] would produce a line connecting first and second element node	Hold[{}&]

Sensitivity Analysis

Abbreviation	*Description*	Default value
SMSSensitivityNames	description of the quantities for which parameter sensitivity pseudo−load code is derived	""
SMSShapeSensitivity	True ⇒ shape sensitivity pseudo−load code is derived False ⇒ shape sensitivity is not enabled	False

See also: Standard user subroutines , SMTSensitivity, SMTAddSensitivity, Standard user subroutines, Solid, Finite Strain Element for Direct and Sensitivity Analysis, Parameter, Shape and Load Sensitivity Analysis of Multi-Domain Example .

AceFEM Solution Procedure Specific

Abbreviation	*Description*	Default value
SMSMMAInitialisation	list of arbitrary length *Mathematica*' s codes and additional function definitions executed after the SMTAnalysis command (wrapping the code in Hold prevents evaluation)	Hold[]
SMSMMANextStep	short *Mathematica*' s code executed after SMTNextStep command (wrapping the code in Hold prevents evaluation)	Hold[]
SMSMMAStepBack	short *Mathematica*' s code executed after SMTStepBack command (wrapping the code in Hold prevents evaluation)	Hold[]
SMSMMAPreIteration	short *Mathematica*' s code executed before SMTNextStep command (wrapping the code in Hold prevents evaluation)	Hold[]
SMSPostIterationCall	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)	False

Description of the Element for AceShare

Abbreviation	*Description*	Default value
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, ...)

SMSUserDataRules	user defined replacement rules that transform standard input/output parameters to user defined input/output parameters (see also User defined environment interface)	{}
FEAP$*	FEAP specific template constants described in chapter FEAP (see FEAP)	
ELFEN$*	ELFEN specific template constants described in chapter ELFEN (see ELFEN)	

Constants defining the general element characteristics .

Element Topology

The element topology defines an outline of the element, spatial dimension, 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[“SMSTpology”-> “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 SMSAdditionalNodes , SMSNoNodes). E.g. the SMSTemplate[“SMSTopology”→”T1”, SMSNoNodes →4, SMSAdditionalNodes→ Hold[{(#1+#2+#3)/3}&] 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. See also: Cubic triangle, Additional nodes .
- 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.

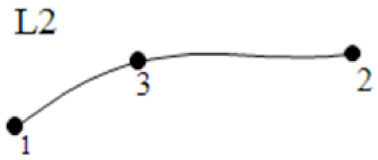
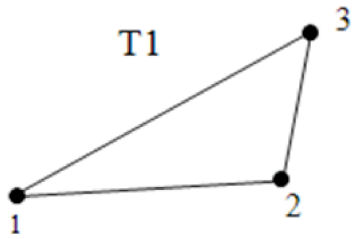
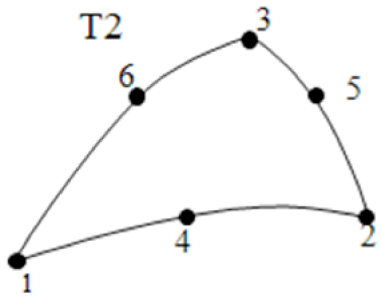
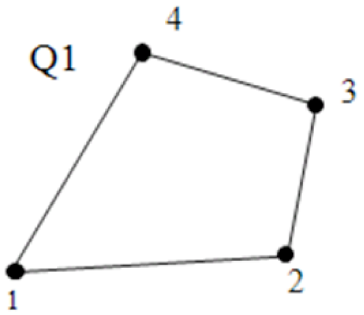
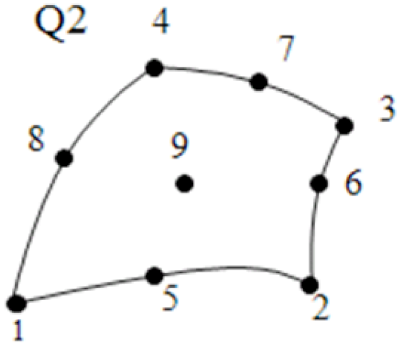
The coordinate systems in the figures below are only informative (e.g. X, Y can also stand for axisymmetric coordinate system X, Y, ϕ).

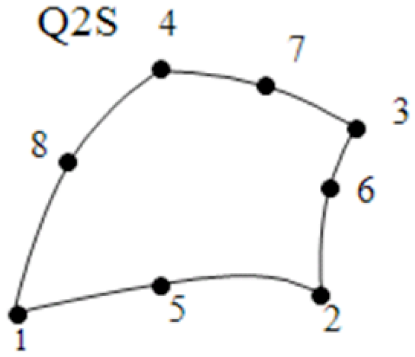
One dimensional

C ode	*Description*	*Node numbering*
"XX"	user defined or unknown topology	arbitrary
"D1"	1 D element with 2 nodes	1−2
"D2"	1 D element with 3 nodes	1−2−3
"DX"	1 D element with arbitrary number of nodes	arbitrary
"V1"	1 D point	1

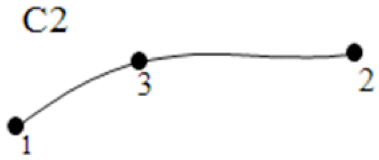
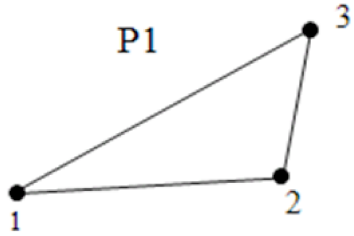
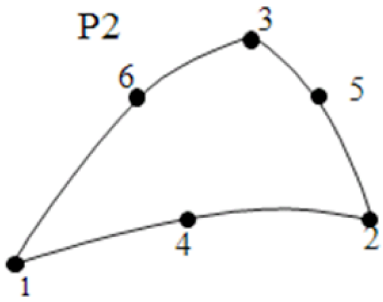
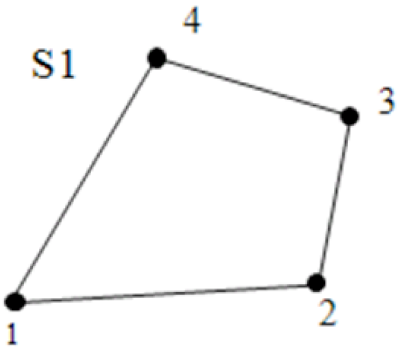
Two dimensional

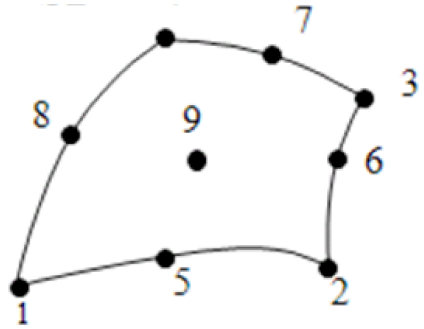
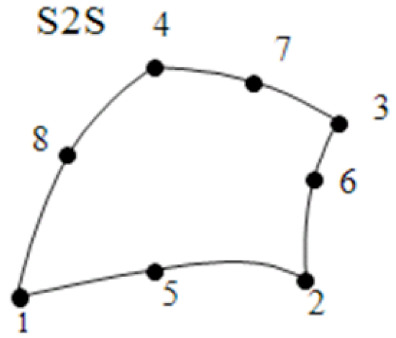
C ode	*Description*	*Node numbering*
"V2"	2 D point	1
"L1"	2 D curve with 2 nodes	

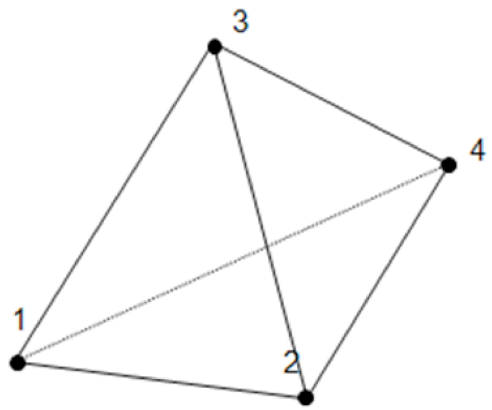
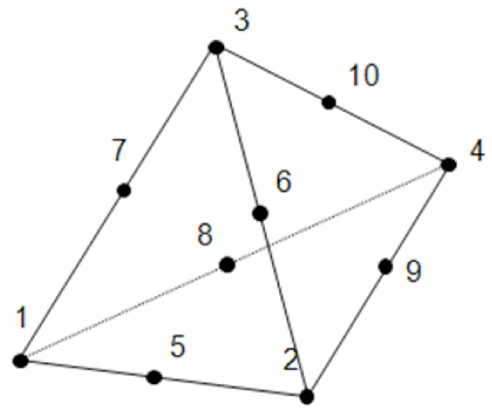
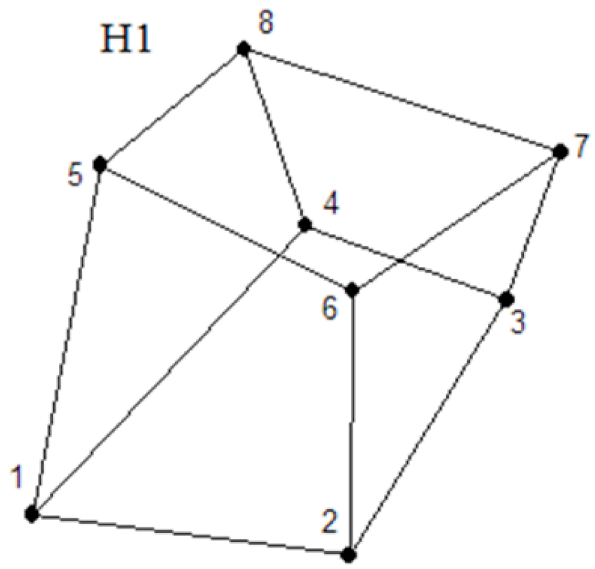
"L2"	2 D curve with 3 nodes	
"LX"	2 D curve with arbitrary number of nodes and arbitrary numbering	arbitrary
"T1"	2 D Triangle with 3 nodes (numerical integration rules and post–processing routines assume "AREA CCORDINATES" of the reference element!)	
"T2"	2 D Triangle with 6 nodes (numerical integration rules and post–processing routines assume "AREA CCORDINATES" of the reference element!)	
"TX"	2 D Triangle with arbitrary number of nodes and arbitrary numbering	arbitrary
"Q1"	2 D Quadrilateral with 4 nodes	
"Q2"	2 D Quadrilateral with 9 nodes	

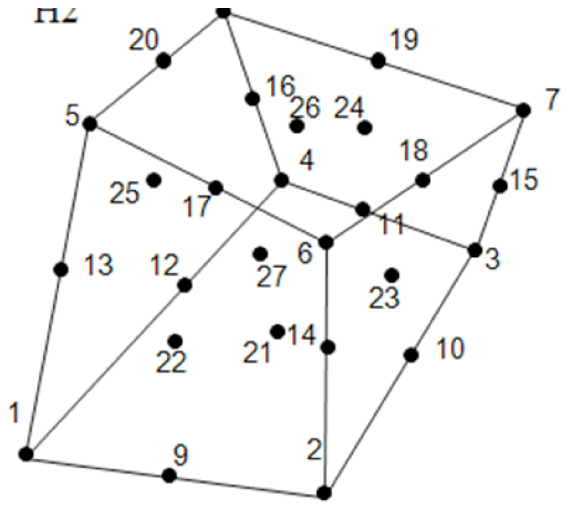
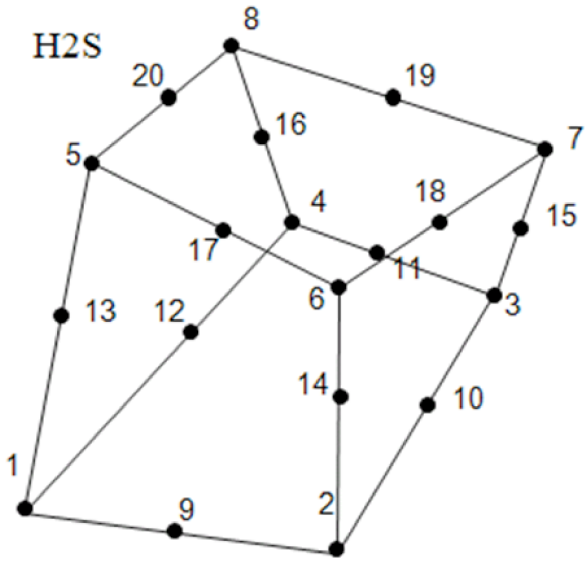
"Q2S"	2 D Quadrilateral with 8 nodes	
"QX"	2 D Quadrilateral with arbitrary number of nodes and arbitrary numbering	arbitrary

Three dimensional

Code	*Description*	*Node numbering*
"V3"	3 D point	1
"C1"	3 D curve with 2 nodes	
"C2"	3 D curve with 3 nodes	
"CX"	3 D curve with arbitrary number of nodes and arbitrary numbering	arbitrary
"P1"	3 D Triangle with 3 nodes	
"P2"	3 D Triangle with 6 nodes	
"PX"	3 D Triangle with arbitrary number of nodes and arbitrary numbering	arbitrary
"S1"	3 D Quadrilateral with 4 nodes	
"S2"	3 D Quadrilateral with 9 nodes	

"S2S"	3 D Quadrilateral with 8 nodes	 
"SX"	3 D Quadrilateral with arbitrary number of nodes and arbitrary numbering	arbitrary

Code	*Description*	*Node numbering*
"O1"	3 D Tetrahedron with 4 nodes (numerical integration rules and post–processing routines assume "AREA CCORDINATES" of the reference element!)	
"O2"	3 D Tetrahedron with 10 nodes (numerical integration rules and post–processing routines assume "AREA CCORDINATES" of the reference element!)	
"OX"	3 D Tetrahedron with arbitrary number of nodes and arbitrary numbering	arbitrary
"H1"	3 D Hexahedron with 8 nodes	
"H2"	3 D Hexahedron with 27 nodes	

"H2S"	3 D Hexahedron with 20 nodes	 
"HX"	3 D Hexahedron with arbitrary number of nodes and arbitrary numbering	arbitrary

Node Identification

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). No names are prescribed in advance, however in order to have consistent set of elements one has to use the same names for the nodes with the same physical meaning. Standard names 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..

Accordingly to the node identification switches a node can be one of three basic types:

- **Topological node**
  Topological node belongs to a specific point in space. It can have associated unknowns.
- **Auxiliary node**
  Auxiliary node does not belong to a specific point in space and is created automatically. Auxiliary node can have associated unknowns. Instead of the nodal coordinates a Null sign must be given. The 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 SMSAdditionalNodes, Mixed 3D Solid FE, Auxiliary Nodes

- **Dummy node**
  Dummy node does not belong to a specific point 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.

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.
−M	A node with the switch M becomes 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 and dummy if the corresponding 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.

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.
- Example: "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 on the main diagonal.

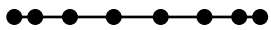
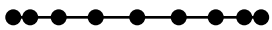
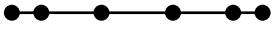
Numerical Integration

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	

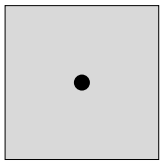
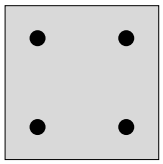
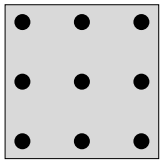
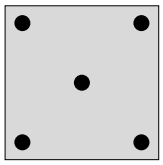
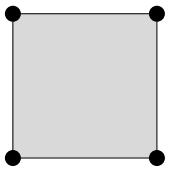
One dimensional

Cartessin 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	

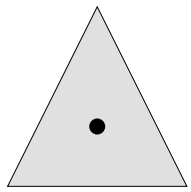
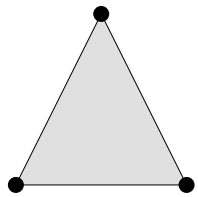
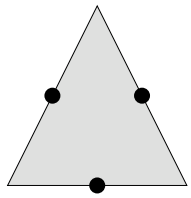
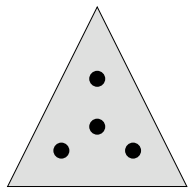
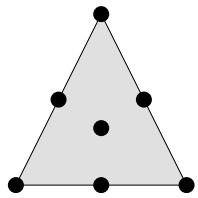
Quadrilateral

Cartessin 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	

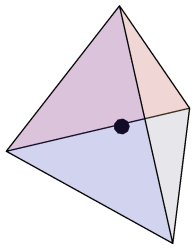
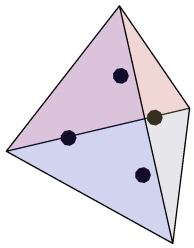
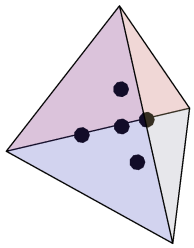
Triangle

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	4	
17	7 point integration	7	

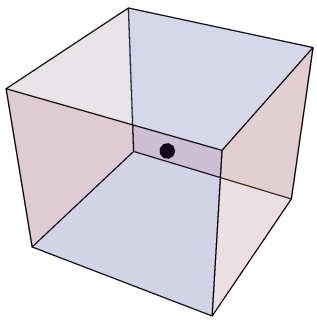
Tetrahedra

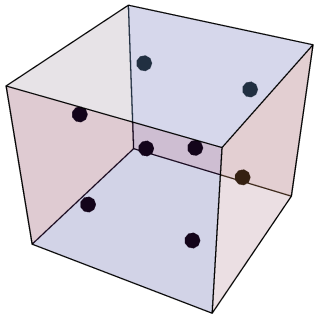
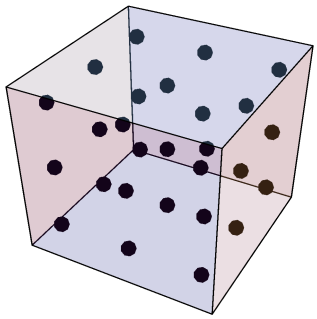
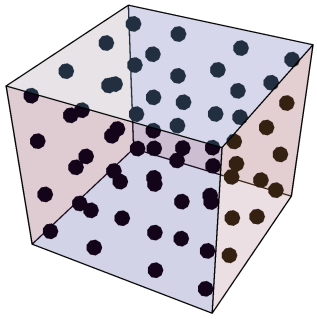
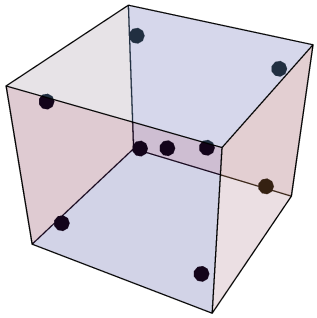
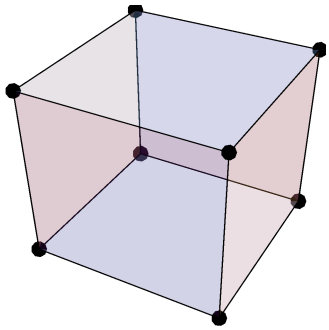
AREA coordinates of the reference element: $\{\zeta, \eta, \zeta\} \in [0,1] \times [0,1] \times [0,1]$

Code	*Description*	*No. of points*	*Disposition*
15	1 point integration	1	
18	4 point integration	4	
19	5 point integration	5	

Hexahedra

Cartessin 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	

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 integrational rules. The combined code is given in the domain specification input data as a list of up to three basic codes as follows:

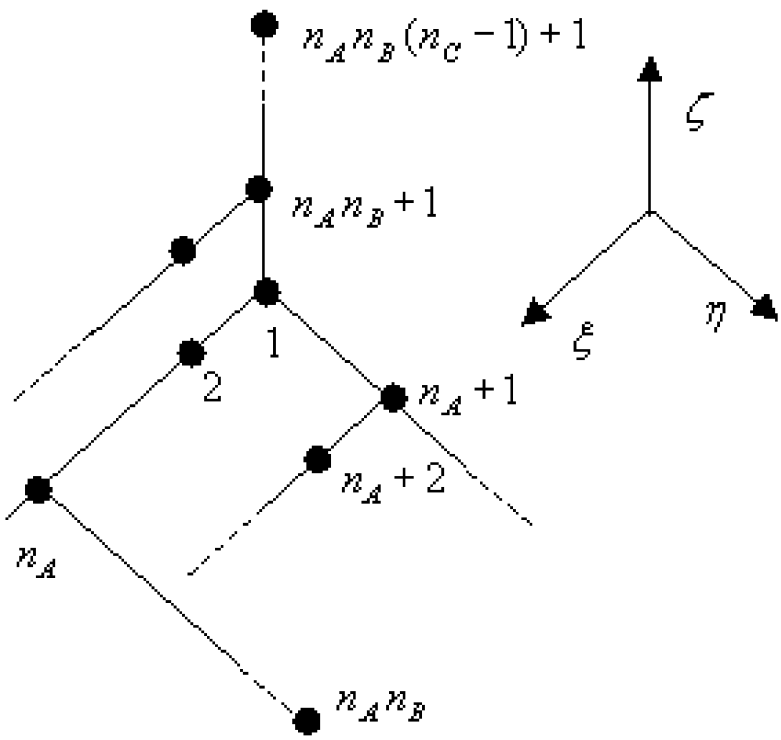
{codeA} ≡ *codeA*

{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

This generates simple loop over all given integration points for 2D integration.

```
SMSDo[Ig, 1, SMSInteger[es$$["id", "NoIntPoints"]]];
    {ξ, w} ⊢ Table[SMSReal[es$$["IntPoints", i, Ig]] &, {i, {1, 4}}];
    …
SMSEndDo[];
```

This generates simple loop over all given integration points for 2D integration.

```
SMSDo[Ig, 1, SMSInteger[es$$["id", "NoIntPoints"]]];
    {ξ, η, w} ⊢ Table[SMSReal[es$$["IntPoints", i, Ig]] &, {i, {1, 2, 4}}];
    …
SMSEndDo[];
```

This generates simple loop over all given integration points for 3D integration.

```
SMSDo[Ig, 1, SMSInteger[es$$["id", "NoIntPoints"]]];
    {ξ, η, ζ, w} ⊢ Table[SMSReal[es$$["IntPoints", i, Ig]] &, {i, {1, 2, 3, 4}}];
    …
SMSEndDo[];
```

Example 2

In the case of the combined integration code, the integration can be also performed separately for each set of points.

```
{nA, nB, nC} ⊢ 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:

$$\begin{pmatrix} K^n_{\mathrm{uu}} & K^n_{\mathrm{uh}} \\ K^n_{hu} & K^n_{\mathrm{hh}} \end{pmatrix} \cdot \begin{pmatrix} \Delta u^n \\ \Delta h^n \end{pmatrix} = \begin{pmatrix} -R^n_u \\ -R^n_h \end{pmatrix} \implies K_{\mathrm{cond}}\, \Delta \mathbf{u}^n = -R_{\mathrm{cond}}$$

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:

$$\boldsymbol{K}_{\text{cond}} = \boldsymbol{K}^{n}_{\text{uu}} - \boldsymbol{K}^{n}_{\text{uh}}\, \boldsymbol{H}^{n}_{a}$$
$$\boldsymbol{R}_{\text{cond}} = \boldsymbol{R}^{n}_{u} + \boldsymbol{K}^{n}_{\text{uh}}\, \boldsymbol{H}^{n}_{b}$$

where $\boldsymbol{H}_a$ is a matrix and $\boldsymbol{H}_b$ a vector defined as

$$\boldsymbol{H}^{n}_{a} = \boldsymbol{K}^{n}_{\text{hh}}{}^{-1}\, \boldsymbol{K}^{n}_{hu}\ .$$
$$\boldsymbol{H}^{n}_{b} = -\boldsymbol{K}^{n}_{\text{hh}}{}^{-1}\, \boldsymbol{R}^{n}_{h}$$

The actual values of the local unknowns are calculated first time when the element tangent and residual subroutine is called by:

$$\boldsymbol{h}^{n+1} = \boldsymbol{h}^{n} + \boldsymbol{H}_b - \boldsymbol{H}_a\, \Delta\mathbf{u}^{n}\ .$$

Three quantities have to be stored at the element level for the presented scheme: the values of the local unknowns $\mathbf{h}^n$, the $\mathbf{H}^n_b$ matrix and the $\mathbf{H}^n_a$ 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

The template constant SMSCondensationData stores pointers at the beginning of the corresponding data field.

Data	*Position*	Dimension	Default for AceFEM
$\mathbf{h}^n$	SMSCondensationData[[1]]	SMSNoDOFCondense	ed$$["ht",1]
$\mathbf{H}^n_b$	SMSCondensationData[[2]]	SMSNoDOFCondense	ed$$["ht", SMSNoDOFCondense+1]
$\mathbf{H}^n_a$	SMSCondensationData[[3]]	SMSNoDOFCondense∗ SMSNoDOFGlobal	ed$$["ht", 2 SMSNoDOFCondense+1]
$\delta\mathbf{h}^n$	SMSCondensationData[[4]]	SMSNoDOFCondense∗ NoSensParameters	ed$$["ht", 2 SMSNoDOFCondense+ SMSNoDOFCondense∗ SMSNoDOFGlobal+ (idata$$["SensIndex"]−1)∗ SMSNoDOFCondense+1]

Storage scheme for the elimination of the local unknowns.

It is assumed that the sensitivity of the local unknowns ($\delta\mathbf{h}^n$) is stored as follows:

$$\left\{\frac{\partial h_1}{\partial p_1}, \frac{\partial h_1}{\partial p_2}, \ldots, \frac{\partial h_1}{\partial p_{\text{NoSensParameters}}}, \frac{\partial h_2}{\partial p_1}, \frac{\partial h_2}{\partial p_1}, \ldots, \frac{\partial h_{\text{SMSNoDOFCondense}}}{\partial p_{\text{NoSensParameters}}}\right\}$$

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 + 9 (-1 + idata$$["SensIndex"])]},
 "SMSNoTimeStorage" → 234 + 9 idata$$["NoSensParameters"]]
```

Example

Let assume that SMSNoTimeStorage constant has value *nht* before the SMSWrite command is executed and that the local unknows were allocated by the "SMSNoDOFCondense" → *nlu* template constant. The true allocation of the storage is then done automatically by the SMSWrite command. The proper AceGen input and the possition of the data within the "ht" history filed that corresponds to the input is as follows:

```
SMSInitialize["test", "Environment" → "AceFEM"];
SMSTemplate["SMSTopology" → "Q1",
 "SMSNoTimeStorage" -> nht, "SMSNoDOFCondense" → nlu]

hi ⊢ SMSReal[Array[ed$$["ht", nht + #] &, nlu]];
j ⊢ SMSInteger[idata$$["SensIndex"]];
δhi ⊨ SMSReal[
   Array[ed$$["ht", nht + 2 nlu + nlu * SMSNoDOFGlobal + (j - 1) * nlu + #] &, nlu]];
 ....
SMSWrite[]
```

Data	*Position of i−th element*	position
$\mathbf{h}^n$	$\mathbf{h}^n[i]$	ed$$["ht",*nht*+i]
$\mathbf{H}^n_b$	$\mathbf{H}^n_b[i]$	ed$$["ht",*nht*+*nlu*+i]
$\mathbf{H}^n_a$	$\mathbf{H}^n_a[i]$	ed$$["ht",*nht*+2 *nlu*+*i*]
$\delta\mathbf{h}^n$	$\delta\mathbf{h}^n[i]$ for j −th sensitivity parameter	ed$$["ht", *nht*+2 *nlu*+*nlu*∗ SMSNoDOFGlobal+ (*j*−1)∗*nlu*+*i*]

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 also SMSStandardModule , Standard FE Procedure .

■ 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 , SMTPost).

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)

This outlines the major parts of the "*Postprocessing*" user subroutine.

```
(* template constants related to the postprocessing*)
SMSTemplate[
 "SMSSegments" → …, "SMSReferenceNodes" → …,
 "SMSPostNodeWeights" → …, "SMSAdditionalGraphics" → …
]
…
SMSStandardModule["Postprocessing"];
…
(* export integration point postprocessing values for all integration points*)
SMSGPostNames = {"Sxx", "Syy", "Sxy", ...};
SMSDo[Ig, 1, SMSInteger[es$$["id", "NoIntPoints"]]];
 …
 SMSExport[{Sxx, Syy, Sxy, ...}, gpost$$[Ig, #1] &];
SMSEndDo[];
…
(* export nodal point postprocessing values for all nodes,
excluded nodes can be omitted*)
SMSNPostNames = {"DeformedMeshX", "DeformedMeshY", "DeformedMeshZ", "u", "v", ...};
SMSExport[{{ui[[1]], vi[[1]], ...}, {ui[[2]], vi[[2]], ...}, ...},
  Table[npost$$[i, j], {i, 1, SMSNoNodes}, {j, 1, Length[SMSNPostNames]}]];
```

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 , SMTPost).

The type of extrapolation is defined by the value of idata$$["ExtrapolationType"] (Integer Type Environment Data) . The nodal value is additionally multiplied by the user defined nodal wight factor that is stored in element specification data structure for each node (es$$["PostNodeWeights",*nodenumber*]). Default value of the nodal weight factor is 1 for all nodes. It can be changed by setting the SMSPostNodeWeights template constant.

■ Subroutine: "Sensitivity pseudo-load" and "Dependent sensitivity"

The "Sensitivity pseudo-load" user subroutine returns pseudo-load vector used in direct implicit analysis to get sensitivities of the global unknowns with respect to arbitrary parameter.

See also: SMTSensitivity, SMTAddSensitivity, Standard user subroutines, Solid, Finite Strain Element for Direct and Sensitivity Analysis, Parameter, Shape and Load Sensitivity Analysis of Multi-Domain Example .

SensType code	*Description*	*SensTypeIndex parameter*
1	parameter sensitivity	an index of the selected parameter as specified in a description of the element (WARNING! The same material constant can have different *SensTypeIndex* in different elements)
2	shape sensitivity	an index of the current shape parameter
3	implicit sensitivity	it has no meaning for implicit sensitivity
4	essential boundary condition sensitivity	an index of the current boundary condition sensitivity parameter (essential or natural)
5	natural boundary condition sensitivity	an index of the current boundary condition sensitivity parameter (essential or natural)

Codes for the "SensType" and "SensTypeIndex" switches.

Here is a shematic example how the sensitivity pseudo-load vector can be evaluated.

```
(*keywords used to identify parameters and a switch
 that indicated availability of the shape sensitivity*)
SMSTemplate[
  ...
  , "SMSSensitivityNames" → {"E -elastic modulus", ...}
  , "SMSShapeSensitivity" -> True
     ...
]

(* index of the current sensitivity parameter*)
SensIndex ⊢ SMSInteger[idata$$["SensIndex"]];
(* type of the parameter*)
SensType ⊢ SMSInteger[es$$["SensType", SensIndex]];
 (* index of the parameter inside the type group*)
SensTypeIndex ⊢ SMSInteger[es$$["SensTypeIndex", SensIndex]];

ϕ ⊨ SMSFictive[];
(*ϕ -current material parameter is introduced as
  fictitious parameter that can represent arbitrary parameter*)

(*define derivatives of all material
 parameters with respect to the current parameter *)
{Em, ν, thick, Qx, Qy} ⊢ Table[
   SMSReal[es$$["Data", i], "Dependency" →
     {ϕ, SMSKroneckerDelta[1, SensType] SMSKroneckerDelta[i, SensTypeIndex]}]
   , {i, 1, 5}];

(*- define derivatives of node coordinates with respect to the current parameter
 - the shape velocity field is by default
  stored in a nodal data field nd$$[i,"sX",j,k] *)
δXYZ ⊨ SMSIf[SensType == 2
   , SMSReal[
    Table[nd$$[i, "sX", SensTypeIndex, j], {i, SMSNoNodes}, {j, SMSNoDimensions}]]
   , Table[0, {i, SMSNoNodes}, {j, SMSNoDimensions}]
  ];
XYZ ⊢ Table[SMSReal[nd$$[i, "X", j], "Dependency" → {ϕ, δXYZ[[i, j]]}],
   {i, SMSNoNodes} , {j, SMSNoDimensions}];

(* - define derivatives of essential boundary conditions
 - the BC velocity field is by default
  stored in a nodal data field nd$$[i,"sBt",j,k]*)
δdof ⊨ SMSIf[SensType == 4
   , Table[
    SMSIsDOFConstrained[SMSInteger[nd$$[i, "DOF", j]],
     SMSReal[nd$$[i, "sBt", SensTypeIndex, j]], 0]
    , {i, SMSNoNodes}, {j, SMSDOFGlobal[[i]]}]
   , Table[0, {i, SMSNoNodes}, {j, SMSDOFGlobal[[i]]}]
  ];
dof ⊢ Table[SMSReal[nd$$[i, "at", j], "Dependency" → {ϕ, δdof[[i, j]]}],
   {i, SMSNoNodes}, {j, SMSDOFGlobal[[i]]}];

(*... body of the subroutine that evaluates residual R ... *)

(*evaluate sensitivity pseudo-load vector for current sensitivity parameter*)
SMSExport[SMSD[R, ϕ], p$$];
```

Subroutine: "Tasks"

The "Tasks" standard user subroutine can be used to perform various tasks that requires the assembly of the results over the complete finite element mesh or over the part of the mesh. The detailed description with examples is given in User Defined Tasks.

argument	*Description*	*Type*
Task$$	Task to be executed: −n ⇒ initialization of task with task identification keyword *taskID* where *n* is position of *taskID* within the vector of character type element switches, thus *es$$["CharSwitch",n]==taskID* *n* ⇒ execute task that corresponds to task *es$$["CharSwitch",n]==taskID*	Integer
TasksData$$[5]	TasksData$$[1] ⇒ task type (see table below) TasksData$$[2] ⇒ the length of the IntegerInput$$ vector TasksData$$[3] ⇒ the length of the RealInput$$ vector TasksData$$[4] ⇒ the length of the IntegerOutput$$ vector TasksData$$[5] ⇒ the length of the RealOutput$$ vector	Real
IntegerInput$$[TasksData$$[2]]	integer input vector (the same for all elements)	Integer
RealInput$$[TasksData$$[3]]	real input vector (the same for all elements)	Real
IntegerOutput$$[TasksData$$[4]]	integer output vector (the values returned from all selected elements are further processed accordingly to the options given to SMTTask command)	Integer
RealOutput$$[TasksData$$[5]]	real output vector (the values returned from all selected elements are further processed accordingly to the options given to SMTTask command)	Real

Additional arguments of the "Tasks" subroutine.

task type	*task description*	active output parameters
1	given integer and real output vectors are evaluated and optionally summarized (see SMTTask command) for selected elements	IntegerOutput$$,RealOutput$$
2	the tasks expects the values of the continuous field to be given for all element nodes and stored in the RealOutput$$ vector (the defined continuous field can then be smoothed and extrapolated to the given spatial point, etc.. , depending on the options given to the SMTTask command)	RealOutput$$
3	the tasks expects the values of the continuous field to be given for all element integration points and stored in the RealOutput$$ vector (the defined continuous field can then be smoothed, extrapolated to the given spatial point, etc.. , depending on the options given to the SMTTask command)	RealOutput$$
4	given local element vectors are assembled accordingly to the standard finite element assembly procedure and given integer and real output vectors are summarized for selected elements	IntegerOutput$$,RealOutput$$,p$$
5	given local element matrices are assembled accordingly to the standard finite element assembly procedure and given integer and real output vectors are summarized for selected elements	IntegerOutput$$,RealOutput$$,s$$
6	types 5 and 6 combined	IntegerOutput$$,RealOutput$$,p$$,s$$

Types of the tasks to be performed.

This outlines the major parts of the "*Tasks*" user subroutine.

Initialization

The SMSCharSwitch constant holds the identifications of the tasks.

```
SMSTemplate[
 …, "SMSCharSwitch" →
  {"TaskType1", "TaskType2", "TaskType3", "TaskType4", "TaskType5", "TaskType6"}, …
]
…
SMSStandardModule["Tasks"];
task ⊨ SMSInteger[Task$$];
…
```

Task type 1

Initialization and execution of the type 1 task with the task identification "TaskType1" that will return 1 integer and 3 real values.

```
SMSIf[task == -1, SMSExport[{1, 0, 0, 1, 3}, TasksData$$]; SMSReturn[];];
SMSIf[task == 1
  , SMSExport[{ival}, IntegerOutput$$];
  SMSExport[{rval1, rval2, rval3}, RealOutput$$];
 ];
```

Task type 2

Initialization and execution of the type 2 task with the task identification "TaskType2" that will return SMTNoNodes real type values

```
SMSIf[task == -2, SMSExport[{2, 0, 0, 0, SMSNoNodes}, TasksData$$]; SMSReturn[];];
SMSIf[task == 2
  , SMSExport[{val_2, val_2, …, val_SMTNoNodes}, RealOutput$$];
 ];
```

Task type 3

Initialization and execution of the type 3 task with the task identification "TaskType3" that will return the number of integration points real type values

```
SMSIf[task == -3, SMSExport[
   {3, 0, 0, 0, SMSInteger[es$$["id", "NoIntPoints"]]}, TasksData$$]; SMSReturn[];];
SMSIf[task == 3
  , SMSExport[{val_2, val_2, …, val_NoIntPoints}, RealOutput$$];
 ];
```

Task type 4

Initialization and execution of the type 4 task with the task identification "TaskType4" that will set the local element vector p$$. The local element vectors are assembled to form a global vector that is result of the SMTTask["-TaskType4"] command.

```
SMSIf[task == -4, SMSExport[{4, 0, 0, 0, 0}, TasksData$$]; SMSReturn[];];
SMSIf[task == 4
  , SMSExport[{val_2, val_2, …, val_SMTNoDofGlobal}, p$$];
 ];
```

Task type 5

Initialization and execution of the type 5 task with the task identification "TaskType5" that will set the local element matrix s$$. The local element matrices are assembled to form a global matrix that is result of the SMTTask["-TaskType5"] command.

```
SMSIf[task == -5, SMSExport[{5, 0, 0, 0, 0}, TasksData$$]; SMSReturn[];];
SMSIf[task == 5
  , SMSExport[localmatrix_{SMTNoDofGlobal,SMTNoDofGlobal}, s$$];
 ];
```

Task type 6

Initialization and execution of the type 6 task with the task identification "TaskType6" that will set the local element matrix s$$ and the local element vector p$$. The local quantities are assembled to form a global quantities that are result of the SMTTask["TaskType6"] command.

```
SMSIf[task == -6, SMSExport[{6, 0, 0, 0, 0}, TasksData$$]; SMSReturn[];];
SMSIf[task == 6
  , SMSExport[{val2, val2, …, valSMTNoDofGlobal}, p$$];
  SMSExport[localmatrixSMTNoDofGlobal,SMTNoDofGlobal, s$$];
 ];
```

Data structures

Environment data structure defines the general information common for all nodes and elements of the problem. If the "default form" of the data is used, then *AceGen* automatically transforms the input into the form that is correct for the selected FE environment. The environment data are stored into two vectors, one for the integer type values (Integer Type Environment Data) and the other for the real type values (Real Type Environment Data). All the environments do not provide all the data, thus automatic translation mechanism can sometimes fails. All data can be in general divided into 6 data structures:

Integer Type Environment Data (in AceGen idata$$,in AceFEM SMTIData)
Real Type Environment Data(in AceGen rdata$$,in AceFEM SMTRData)
Domain Specification Data (in AceGen es$$,in AceFEM SMTDomainData)
Element Data (in AceGen ed$$,in AceFEM SMTElementData)
Node Specification Data (in AceGen ns$$,in AceFEM SMTNodeSpecData)
Node Data (in AceGen nd$$,in AceFEM SMTNodeData)

■ Node Data Structures

Two types of the node specific data structures are defined. The structure (Node Specification Data , ns$$*)* defines the major characteristics of the nodes sharing the same node identification (NodeID, Node Identification). Nodal data structure (Node Data , nd$$*)* contains all the data that are associated with specific node. Nodal data structure can be set and accessed from the element code. For example, the command *SMSReal[nd$$[i,"X",1]]* returns *x*-coordinate of the *i-th* element node. At the analysis phase the data can be set and accessed interactively from the *Mathematica* by the user (see SMTNodeData , SMTElementData ...). The data are always valid for the current element that has been processed by the FE environment. Index *i* is the index of the node accordingly to the definition of the particular element.

■ Element Data Structures

Two types of the element specific data structures are defined. The domain specification data structure (Domain Specification Data , es$$) defines the major characteristics of the element that is used to discretize particular sub-domain of the problem. It can also contain the data that are common for all elements of the domain (e.g. material constants). The element data structure (Element Data , ed$$) holds the data that are specific for each element in the mesh.

For a transient problems several sets of element dependent transient variables have to be stored. Typically there can be two sets: the current (*ht*) and the previous (*hp*) values of the transient variables. The *hp* and *ht* data are switched at the beginning of a new step (see SMTNextStep).

All element data structures can be set and accessed from the element code. For example, the command *SMSInteger[ed$$["nodes",1]]* returns the index of the first element node. The data is always valid for the current element that has been processed by the FE environment.

Integer Type Environment Data

■ General data

Default form	*Description*	*Default/ Read−Write*
idata$$["IDataLength"]	actual length of idata vector	200/R
idata$$["RDataLength"]	actual length of rdata vector	200/R
idata$$["IDataLast"]	index of the last value reserved on *idata* vector (we can store additional user defined data after this point)	?/R
idata$$["RDataLast"]	index of the last value reserved on r*data* vector (we can store additional user defined data after this point)	?/R
idata$$["LastIntCode"]	last integration code for which numerical integration points and weights were calculated	?/R
idata$$["OutputFile"]	output file number or output channel number	?/R
idata$$["SymmetricTangent"]	1 ⇒ global tangent matrix is symmetric 0 ⇒ global tangent matrix is unsymmetrical	?/R
idata$$["MinNoTmpData"]	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])	3
idata$$["Task"]	code of the current task performed	?/R
idata$$["CurrentElement"]	index of the current element processed	0/R
idata$$["TmpContents"]	the meaning of the temporary real type variables stored during the execution of a single analysis into nd$$[i,"tmp", j] data structure 0 ⇒ not used 1 ⇒ residual (reactions) 2 ⇒ used for postprocessing	0
idata$$["AssemblyNodeResidual"]	0 ⇒ residual vector is not formed separately 1 ⇒ during the execution of the SMTNewtonIteration command the residual vector is formed separately and stored into nd$$[i,"tmp", j] (at the end the nd$$[i,"tmp", j] contains the j− th component of the nodal reaction in the i−th node)	0
idata$$["Debug"]	1 ⇒ prevent closing of the CDriver console on exit	0/RW
idata$$["DataMemory"]	memory used to store data (bytes)	0
idata$$["GeometricTangentMatrix"]	Used for buckling analysis ($K_0+\lambda K_\sigma$) {Ψ}={0}): 0 ⇒ form full nonlinear matrix 1 ⇒ form K_0 2 ⇒ form K_σ 3 ⇒ form K_0+K_σ	0
idata$$["ExtrapolationType"]	type of extrapolation of integration point values to nodes 0 ⇒ least square extrapolation ( ) 1⇒ integration point value is multiplied by the user defined weight factors (see SMSStandardModule)	0/RW
idata$$["NoThreads"]	number of processors that are available for the parallel execution	All available

Mesh input related data

Default form	*Description*	*Default/ Read–Write*
idata$$["NoNodes"]	total number of nodes	?/R
idata$$["NoElements"]	total number of elements	?/R
idata$$["NoESpec"]	total number of domains	?/R
idata$$["NoDimensions"]	number of spatial dimensions of the problem (2 or 3)	?/R
idata$$["NoNSpec"]	total number of node specifications	?/R
idata$$["NoEquations"]	total number of global equations	?/R
idata$$["DummyNodes"]	1 ⇒ dummy nodes are supported for the current analysis	0
idata$$["NoMultipliers"]	number of boundary conditions multipliers	1

Iterative procedure related data

See: Iterative solution procedure , SMTConvergence , SMTStatusReport , SMTErrorCheck

idata$$["Iteration"]	index of the current iteration within the iterative loop	?/R
idata$$["TotalIteration"]	total number of iterations in session	?/R
idata$$["Step"]	total number of completed solution steps (set by Newton−Raphson iterative procedure)	0
idata$$["LinearEstimate"]	if 1 then in the first iteration of the NewtonRaphson iterative procedure the prescribed boundary conditions are not updated and the residual is evaluated by R=R (ap)+K (ap)$*\Delta a_{\text{prescribed}}$	0/RW
idata$$["PostIteration"]	is set by the SMTConvergence command to 1 if idata$$["PostIterationCall"]= 1 or the SMTConvergence has been called with switch "PostIteration" −>True	0
idata$$["PostIterationCall"]	1 ⇒ additional call of the SKR user subroutines after the convergence of the global solution is enabled in at least one of the elements ("SMSPostIterationCall"−>True)	0
idata$$["SkipTangent"]	1 ⇒ the global tangent matrix is not assemled	0
idata$$["SkipResidual"]	1 ⇒ the global residual vector is not assembled	0
idata$$["NoSubIterations"]	maximal number of local sub− iterative process iterations performed during the analysis	0/R
idata$$["SubIterationMode"]	Switch used in the case that alternating solution has been detected by the SMTConvergence function. $0 \Rightarrow {}_{i+1}\mathbf{b}_0^t=\mathbf{b}^p$ $\geq 1 \Rightarrow {}_{i+1}\mathbf{b}_0^t={}_i\mathbf{b}^t$	0
idata$$["GlobalIterationMode"]	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)	0
idata$$["MaxPhysicalState"]	used for the indication of the physical state of the element (e.g. 0−elastic, 1−plastic, etc., user controlled option)	0/RW
idata$$["LineSearchUpdate"]	activate line search procedure (see also idata$$["LineSearchStepLength"])	False
idata$$["NoBackStep"]	number of failed iterative solution steps	0

Debugging and errors related data

See: Iterative solution procedure , SMTConvergence , SMTStatusReport , SMTErrorCheck

idata$$["ErrorStatus"]	code for the type of the most important error event	0/RW
idata$$["SubDivergence"]	number of the "Divergence of the local sub−iterative process" error events detected form the last error check	0/RW
idata$$["ErrorElement"]	last element where error event occurred	0
idata$$["NoDiscreteEvents"]	number of discrete events recordered during the NR−iteration by the elements (e.g. new contact node, transformation from elastic to plastic regime)	0
idata$$["MaterialState"]	number of the "Non−physical material point state" error events detected form the last error check	0/RW
idata$$["ElementShape"]	number of the "Non−physical element shape" error events detected form the last error check	0/RW
idata$$["MissingSubroutine"]	number of the "Missing user defined subroutine" error events detected form the last error check	0/RW
idata$$["ElementState"]	number of the "Non−physical element state" error events detected form the last error check	0/RW
idata$$["DebugElement"]	−1 ⇒ break points (see Interactive Debugging) 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"]	0

Linear solver related data

idata$$["SkipSolver"]	0 ⇒ full Newton−Raphson iteration 1 ⇒ the tangent matrix and the residual vector are assembled but the resulting sistem of equations is not solved	0
idata$$["SetSolver"]	1 ⇒ recalculate solver dependent data structures if needed	0
idata$$["SolverMemory"]	memory used by solver (bytes)	0
idata$$["Solver"]	solver identification number	0
idata$$["Solver1"] idata$$["Solver2"] idata$$["Solver3"] idata$$["Solver4"] idata$$["Solver5"]	solver specific parameters	
idata$$["ZeroPivots"]	number of near−zero pivots (see also SMTSetSolver)	0
idata$$["NegativePivots"]	number of negative pivots (or −1 if data is not available), (see also SMTSetSolver)	0
idata$$["NoLinearConstraints"]	number of linear constraint equations	0

Sensitivity related data

idata$$["NoSensParameters"]	total number of sensitivity parameters (see `"Sensitivity pseudo-load"`)	?/R
idata$$["SensIndex"]	index of the current sensitivity parameter – globally to the problem (see `"Sensitivity pseudo-load"`)	?/R
idata$$["NoBCParameters"]	number of bounday conditions sensitivity parameters	0
idata$$["NoShapeParameters"]	total number of shape sensitivity parameters	0

Contact related data

idata$$["ContactProblem"]	1 ⇒ global contact search is enabled 0 ⇒ global contact search is disabled	1/R
idata$$["Contact1"] idata$$["Contact2"] idata$$["Contact3"] idata$$["Contact4"] idata$$["Contact5"]	contact problem specific parameters	

Integer type environment data.

■ All data structures

Real Type Environment Data

Default form	Description	Default
rdata$$["Multiplier"]	current values of the natural and essential boundary conditions are obtained by multiplying initial values with the *rdata$$["Multiplier"]* (the value is also known as load level or load factor)	0
rdata$$["ResidualError"]	Modified Euklid' s norm of the residual vector $\sqrt{\frac{\mathbf{R}.\mathbf{R}}{\text{NoEquations}}}$	10^{55}
rdata$$["IncrementError"]	Modified Euklid' s norm of the last increment of global d.o.f $\sqrt{\frac{\Delta\mathbf{a}.\Delta\mathbf{a}}{\text{NoEquations}}}$	10^{55}
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$$["Time"]	real time	0
rdata$$["TimeIncrement"]	value of the last real time increment	0
rdata$$["MultiplierIncrement"]	value of the last multiplier increment	0
rdata$$["SubIterationTolerance"]	tolerance for the local sub−iterative process	10^{-9}
rdata$$["LineSearchStepLength"]	step size control factor η $\left({}_{i+1}\mathbf{a}^{t}={}_{i}\mathbf{a}^{t}+\eta\,\Delta\,{}_{i}\mathbf{a}\right)$ (see also idata$$["LineSearchUpdate"])	Automatic
rdata$$["PostMaxValue"]	the value is set by the postprocessing SMTPost function to the true maximum value of the required quontitie (note that the values returned by the SMTPost function are smoothed over the patch of elements)	0
rdata$$["PostMinValue"]	the value is set by the postprocessing SMTPost function to the true minimum value of the required quantity	0
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	

Real type environment data.

Node Specification Data

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	ns$$[i, "id","NoData"] real numbers
ns$$[i,"NodeID"]	node identification (see Node Identification)	string

Node specification data structure.

Node Data

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	1
nd$$[i,"DOF", j]	global index of the j −th nodal d.o.f or −1 if there is an essential boundary condition assigned to the j −th d.o.f.	NoDOF
nd$$[i,"Elements"]	list of elements associated with i −th node	NoElements
nd$$[i,"X", j]	initial coordinates of the node	3 (1−X,2−Y,3−Z)
nd$$[i,"at", j]	current value of the j −th nodal d.o.f $(\mathbf{a}_i^t)$	NoDOF
nd$$[i,"ap", j]	value of the j −th nodal d.o.f at the end of previous step $(\mathbf{a}_i^p)$	NoDOF
nd$$[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]* $\equiv -1 \Rightarrow$ current value of the j −th essential boundary condition *nd$$[i,"DOF",j]* $\geq 0 \Rightarrow$ current value of the j−th natural boundary condition	NoDOF
nd$$[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]	arbitrary node specific data (e.g. initial sensitivity in the case of shape sensitivity analysis)	NoNodeData real numbers
nd$$[i,"ht",j]	current state of the j −th transient specific variable in the i−th node	NoNodeStorage real numbers
nd$$[i,"hp",j]	the state of the j −th transient 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])
nd$$[i,"ppd",j]	post−processing data where nd$$[i,"ppd",1] is the sum of all weights and nd$$[i,"ppd",2] is smoothed nodal value ≡ nd$$[i,"tmp",j]	2

Nodal data structure.

Default form	*Description*	*Dimension*
nd$$[i,"st", j, k]	current sensitivities of the $k-$th nodal d.o.f with respect to the $j-$th sensitivity parameter $\left(\frac{\partial \mathbf{a}^{t}{}_{i}}{\partial \phi_{j}}\right)$	NoDOF∗ NoSensParameters
nd$$[i,"sp", j, k]	sensitivities of the $k-$th nodal d.o.f with respect to the $j-$th sensitivity parameter in previous step $\left(\frac{\partial \mathbf{a}^{p}{}_{i}}{\partial \phi_{j}}\right)$	NoDOF∗ NoSensParameters
nd$$[i, "sX", j, k]	initial sensitivity of the $k-$th nodal coordinate of the i −th node with respect to the $j-$th shape sensitivity parameter ≡ nd$$[i,"Data",NoNodeData+SMSNoDimensions∗(j−1)+k]	SMSNoDimensions∗ NoShapeParameters real numbers
nd$$[i,"sBt", j,k]	current sensitivity of the $k-$th dof of the i −th node with respect to the $j-$th boundary sensitivity parameter ≡ nd$$[i,"Bt",NoDOF+NoDOF∗(j−1)+k]	NoDOF∗ NoBCParameters
nd$$[i,"sBp", j,k]	sensitivity of the $k-$th dof of the i −th node with respect to the $j-$th boundary sensitivity parameter at the end of previous step ≡ nd$$[i,"Bp",NoDOF+NoDOF∗(j−1)+k]	NoDOF∗ NoBCParameters
nd$$[i,"sdB", j,k]	reference value of the sensitivity of the $jk-$th dof of the i −th node with respect to the $j-$th boundary sensitivity parameter ≡ nd$$[i,"dB",NoDOF+NoDOF∗(j−1)+k] (current sensitivity value is defined as $sBt=sBp+\Delta\lambda\ sdB$, where $\Delta\lambda$ is the multiplier increment)	NoDOF∗ NoBCParameters

Nodal data related to sensitivity analysis.

Domain Specification Data

■ Memory allocation (element, node, domain and global level)

Default form	Description	Type
es$$["id","NoTimeStorage"]	total length of the vector of history dependent variables per element (element level)	integer expression
es$$["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

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$$["MMAInitialisation"]	*Mathematica*' s code executed after SMTAnalysis command	string
es$$["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

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
es$$["id", "CreateDummyNodes"]	enable use of dummy nodes	False

Domain input data

es$$["id","NoGroupData"]	number of input data values that are common for all elements in domain (e.g material constants) and are provided by the user is input data	integer
es$$["GroupDataNames", *i*]	description of the *i* −th input data value that is common for all elements with the same specification	NoGroupData× string
es$$["Data",j]	data common for all the elements within a particular domain (fixed length)	NoGroupData real numbers
es$$["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

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$$["IntPoints",*i*,*j*]	coordinates and weights of the numerical integration points ξ_i=es$$["IntPoints",1,i], η_i=es$$["IntPoints",2,i], ζ_i=es$$["IntPoints",3,i], w_i=es$$["IntPoints",4,i]	NoIntPoints∗4 real numbers

Graphics postprocessing

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$$["PostNodeWeights",*i*]	see SMTPost	NoNodes real numbers
es$$["AdditionalGraphics"]	pure function (see Function) that is called for each element and returns additional graphics primitives per element (see Template Constants)	string

Sensitivity analysis

Default form	Description	Type
es$$["id","NoSensNames"]	number of quantities for which parameter sensitivity pseudo−load code is derived	integer
es$$["SensitivityNames",i]	description of the quantities for which parameter sensitivity pseudo−load code is derived	NoSensNames∗ string
es$$["SensType", *i*]	type of the *i*−th sensitivity parameter (see Standard user subroutines)	NoSensParameters integer numbers
es$$["SensTypeIndex", *i*]	index of the *i* −th parameter defined locally in a type group (see Standard user subroutines)	NoSensParameters integer numbers
es$$["id","ShapeSensitivity"]	1 ⇒ shape sensitivity pseudo−load code is present 0 ⇒ shape sensitivity is not enabled	integer

Element Data

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$$["id","Active"]	1 ⇒ element is active 0 ⇒ element is ignored for all actions	integer
ed$$["Nodes",j]	index of the *j* −th element nodes	NoNodes integer numbers
ed$$["Data",j]	arbitrary element specific data	NoElementData real numbers
ed$$["ht",j]	current state of the *j* −th transient element specific variable	NoTimeStorage real numbers
ed$$["hp",j]	the state of the *j* −th transient variable at the end of the previous step	NoTimeStorage real numbers

Element data structure.

Interactions Templates-AceGen-AceFEM

Glossary

This chapter explains the relations among the general template constants (SMSTemplate) , the input-output parameters of generated user subroutines (Symbolic-Numeric Interface) and the AceFEM environment data manipulation routines.

symbol	*description*	*symbol*	*description*
N	positive integer number	"ab"	arbitrary string
eN	integer type expression	"K"	keyword
R	real number	TF	True / False
i, j	index	e	element number
n	node number – within the element	"dID"	domain identification
m	node number – within the global mesh	f&	pure function (see Function)

Element Topology

Template Constant	*AceGen external variable*	*AceFEM data*
"SMSTopology"–>"K"	es$$["Topology"]	SMTDomainData["dID","Topology"]
"SMSNoDimensions"–>N	es$$["id","NoDimensions"]	SMTDomainData["dID","NoDimensions"]
"SMSNoNodes"–>N	es$$["id","NoNodes"] ed$$["Nodes",i]	SMTDomainData["dID","NoNodes"] SMTElementData[e,"Nodes"]
"SMSDOFGlobal"–>{N,…}	es$$["DOFGlobal",*i*] nd$$[n,"id","NoDOF"] es$$["id","NoDOFGlobal"] es$$["id","MaxNoDOFNode"] es$$["id","NoAllDOF"]	SMTDomainData["dID","DOFGlobal"] SMTNodeData[m,"NoDOF"] SMTDomainData["dID","NoDOFGlobal"] SMTDomainData["dID","MaxNoDOFNode"] SMTDomainData["dID","NoAllDOF"]
"SMSNoDOFCondense"–>N	es$$["id","NoDOFCondense"]	SMTDomainData["dID","NoDOFCondense"]
"SMSCondensationData"–> {N,N,N}	–	–

Template Constant	*AceGen external variable*	*AceFEM data*
"SMSAdditionalNodes"–*f&*	–	–
"SMSNodeID"–>{"K" …}	es$$["NodeID",*i*]	SMTDomainData["dID","NodeID"]
"SMSCreateDummyNodes"–> TF	es$$["id", "CreateDummyNodes"]	SMTDomainData["dID","CreateDummyNodes"]

Automatic mesh generation.

Memory Management

Template Constant	*AceGen external variables*	*AceFEM data*
"SMSNoTimeStorage"->e*N*	es$$["id","NoTimeStorage"] ed$$["ht",*i*] ed$$["hp",*i*]	SMTDomainData["dID","NoTimeStorage"] SMTElementData[e,"ht",i] SMTElementData[e,"hp",i]
"SMSNoElementData"->e*N*	es$$["id","NoElementData"] ed$$["Data",*i*]	SMTDomainData["dID","NoElementData"] SMTElementData[e,"Data",i]
"SMSNoNodeStorage"->e*N*	es$$["id","NoNodeStorage"] nd$$[n,"ht",*i*] nd$$[n,"hp",*i*]	SMTDomainData["dID","NoElementData"] SMTNodeData[n,"ht",i] SMTNodeData[n,"hp",i]
"SMSNoNodeData"->e*N*	es$$["id","NoNodeData"] nd$$[n,"Data",*i*]	SMTDomainData["dID","NoNodeData"] SMTNodeData[n,"Data",i]
"SMSIDataNames"->{"K" …}	es$$["id", "NoIData"] es$$["IDataNames",i] es$$["IDataIndex",i] idata$$["K"]	SMTDomainData["dID","NoIData"] SMTDomainData["dID","IDataNames"] SMTIData["K"]
"SMSRDataNames"->{"K" …}	es$$["id", "NoRData"] es$$["RDataNames",i] es$$["RDataIndex",i] rdata$$["K"]	SMTDomainData["dID","NoRData"] SMTDomainData["dID","RDataNames"] SMTRData["K"]

Element Description

Template Constant	*AceGen external variable*	*AceFEM data*
"SMSMainTitle"->"ab"	es$$["MainTitle"]	SMTDomainData["dID","MainTitle"]
"SMSSubTitle"->"ab"	es$$["SubTitle"]	SMTDomainData["dID","SubTitle"]
"SMSSubSubTitle"->"ab"	es$$["SubSubTitle"]	SMTDomainData["dID","SubSubTitle"]
"SMSBibliography"->"ab"	es$$["Bibliography"]	SMTDomainData["dID","Bibliography"]

Input Data

Template Constant	*AceGen external variables*	*AceFEM data*
"SMSGroupDataNames"–> {"ab" …}	es$$["id","NoGroupData"] es$$["GroupDataNames",*i*]	SMTDomainData["dID","NoGroupData"] SMTDomainData["dID","GroupDataNames"] SMTDomainData["dID","Data"]
"SMSNoAdditionalData"–>eN	es$$["id","NoAdditionalData"] es$$["AdditionalData",*i*]	SMTDomainData["dID","NoAdditionalData"] SMTDomainData["dID","AdditionalData"]
"SMSCharSwitch"–>{"ab" …}	es$$["id","NoCharSwitch"] es$$["CharSwitch",*i*]	SMTDomainData["dID","NoCharSwitch"] SMTDomainData["dID","CharSwitch"]
"SMSIntSwitch"–>{i…}	es$$["id","NoIntSwitch"] es$$["IntSwitch",*i*]	SMTDomainData["dID","NoIntSwitch"] SMTDomainData["dID","IntSwitch"]
"SMSDoubleSwitch"–>{i…}	es$$["id","NoDoubleSwitch"] es$$["DoubleSwitch",*i*]	SMTDomainData["dID","NoDoubleSwitch"] SMTDomainData["dID","DoubleSwitch"]

Mathematica

Template Constant	*AceGen external variables*	*AceFEM data*
"SMSMMAInitialisation"–> "ab"	es$$["MMAInitialisation"]	SMTDomainData["dID","MMAInitialisation"]
"SMSMMANextStep"–>"ab"	es$$["MMANextStep"]	SMTDomainData["dID","MMANextStep"]
"SMSMMAStepBack"–>"ab"	es$$["MMAStepBack"]	SMTDomainData["dID","MMAStepBack"]
"SMSMMAPreIteration"–>"ab"	es$$["MMAPreIteration"]	SMTDomainData["dID","MMAPreIteration"]
"SMSMMAInitialisation"–> "ab"	es$$["MMAInitialisation"]	SMTDomainData["dID","MMAInitialisation"]

Presentation of Results

Template Constant	*AceGen external variables*	*AceFEM data*
"SMSGPostNames"->{"ab" …}	es$$["id","NoGPostData"] es$$["GPostNames",*i*]	SMTDomainData["dID","NoGPostData"] SMTDomainData["dID","GPostNames"]
"SMSNPostNames"->{"ab" …}	es$$["id","NoNPostData"] es$$["NPostNames",*i*]	SMTDomainData["dID","NoNPostData"] SMTDomainData["dID","NPostNames"]
"SMSSegments"->{N…}	es$$["id","NoSegmentPoints"] es$$["Segments", *i*]	SMTDomainData["dID","NoSegmentPoints"] SMTDomainData["dID","Segments"]
"SMSReferenceNodes"->{N…}	es$$["ReferenceNodes",i]	SMTDomainData["dID","ReferenceNodes"]
"SMSPostNodeWeights"->{N…}	es$$["PostNodeWeights",i]	SMTDomainData["dID","PostNodeWeights"]
"SMSAdditionalGraphics"->f&	es$$["AdditionalGraphics"]	SMTDomainData["dID","AdditionalGraphics"]

General

Template Constant	*AceGen external variable*	*AceFEM data*
"SMSPostIterationCall"->TF	es$$["PostIterationCall"]	SMTDomainData["dID","PostIterationCall"]
"SMSSymmetricTangent"->TF	es$$["id","SymmetricTangent"]	SMTDomainData["dID","SymmetricTangent"]
"SMSDefaultIntegrationCode"->N	es$$["id", "DefaultIntegrationCode"] es$$["id","IntCode"] es$$["id","NoIntPoints"] es$$["id","NoIntPointsA"] es$$["id","NoIntPointsB"] es$$["id","NoIntPointsC"] es$$["IntPoints",*i*,*j*]	SMTDomainData["dID", "DefaultIntegrationCode"] SMTDomainData["dID","IntCode"] SMTDomainData["dID","NoIntPoints"] SMTDomainData["dID","NoIntPointsA"] SMTDomainData["dID","NoIntPointsB"] SMTDomainData["dID","NoIntPointsC"] SMTDomainData["dID","IntPoints"]

Options for numerical procedures.

Template Constant	*AceGen external variable*	*AceFEM data*
"SMSSensitivityNames"->{"ab" …}	es$$["id","NoSensNames"] es$$["SensitivityNames",i] es$$["SensType", *i*] es$$["SensTypeIndex", *i*]	SMTDomainData["dID","NoSensNames"] SMTDomainData["dID","SensitivityNames"] SMTDomainData["dID","SensType"] SMTDomainData["dID","SensTypeIndex"]
"SMSShapeSensitivity"->TF	es$$["id","ShapeSensitivity"]	SMTDomainData["dID","ShapeSensitivity"]

Sensitivity related data.

Template Constant	AceGen external variables	AceFEM data
"SMSResidualSign"->R	–	–
"SMSNodeOrder"->{N…}	–	–
"SMSUserDataRules"->rules	–	–

Compatibility related data.

User defined environment interface

Regenerate the heat conduaction 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 !!!

```
<<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.

```
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.

An 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.

```
SMSInitialize["UserEnvironment", "Environment" -> "User", "Language" -> "C"];
SMSTemplate["SMSTopology" → "H1", "SMSDOFGlobal" → 1,
  "SMSSymmetricTangent" → False, "SMSGroupDataNames" ->
   {"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$$["Data", i], {i, Length[SMSGroupDataNames]}]];
k ⊨ k0 + k1 ϕ + k2 ϕ^2;
SMSSetBreak["k"];
λ ⊢ SMSReal[rdata$$["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

```
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 Structure)

Examples related to the automation of the Finite Element Method using AceFEM are part of **AceFEM** documentation (see Summary of Examples).

FEAP

About *FEAP*

FEAP is an FE environment developed by R. L. Tylor, Department of Civil Engineering, University of California at Berkeley, Berkeley, California 94720.

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 SMSStandardModule). By default, the element with the number 10 is generated.

In order to put a new element in *FEAP* we need:

⇒ *FEAP* libraries (refer to http://www.ce.berkeley.edu/~rlt/feap/)

⇒ element source file.

⇒ supplementary files (files can be find at directory $BaseDirectory/Applications/AceGen/Include/FEAP/).

Supplementary files are:

⇒ SMS.h has to be available when we compile element source code

⇒ SMSUtility.f contains supplementary routines for the evaluation of Gauss points, static condensation etc.

⇒ sensitivity.h, Umacr0.f and uplot.f files contain *FEAP* extension for the sensitivity analysis,

⇒ Umacr3.f contain *FEAP* extension for automatic exception and error handling.

Files has to be placed in an appropriate subdirectories of the *FEAP* project and included into the *FEAP* project.

The *FEAP* source codes of the elements presented in the examples section can be obtained by setting environment option of SMSIntialize to "FEAP").

How to set paths to FEAP's Visual Studio project is described in http://www.fgg.uni-lj.si/symech/User/AceInstallation.htm .

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	*Description*	*Default*
FEAP$ElementNumber	element user subroutine number (*elmt ??*)	"10"

Additional FEAP template constants.

Some of the standard interface data are interpreted in a FEAP specific form as follows.

Standard form	*Description*	*FEAP interpretation*
es$$["SensType", *j*]	type of the *j*−th (current) sensitivity parameter	idata$$["SensType"]
es$$["SensTypeIndex", j]	index of the *j*−th (current) sensitivity parameter within the type group	idata$$["SensTypeIndex"]
nd$$[*i*, "sX", *j*, *k*]	initial sensitivity of the *k*−tk nodal coordinate of the *i*−th node with respect to the *j*−th shape sensitivity parameter	sxd$$[(i−1) SMSNoDimensions+k]

The FEAP specific interpretation of the standard interface data.

FEAP extensions

FEAP has built-in command language. Additional commands are defined (see *FEAP* manual) for the tasks that are not supported directly by the *FEAP* command languge .

Command	*Description*
sens,set	allocate working fields for all sensitivity parameters
sens,solv	solve sensitivity problem for all parameters in for current time step
sens,disp	display sensitivities for all parameters and all nodes
sens,disp,n	display sensitivities for the *n*−th parameters and all nodes
sens,disp,n,m	display sensitivities for the *n*−th parameter and the *m*−th node
sens,disp,n,m,k	display sensitivities for the *n*−th parameter and nodes *m* to *k*
plot,uplo,n,m,k	plot the *m*−th component of the *n*−th sensitivity parameter where *k* determines the number of contour lines and the type of contour

Additional FEAP macro commands for sensitivity calculations.

Command	*Description*
chkc	report error status to the screen and to the output file and clear all the error flags
chkc, clea	clear all the error flags and write report to the output file
chkc, clea, tag	*tag* is an arbitrary number included in a report that can be used to locate the error

Additional FEAP macro commands for exception and error handling.

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.

Generation of element source code for *FEAP* environment

```
<< "AceGen`";
SMSInitialize["test", "Environment" → "FEAP"];
SMSTemplate["SMSTopology" → "H1",
  "SMSSymmetricTangent" → True, "SMSNoDOFCondense" → 9
  , "SMSGroupDataNames" -> {"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];
```

$$\text{HbΞ} = \left(\begin{array}{|c|c|c|}\hline \xi\,\alpha e[[1]] & \eta\,\alpha e[[2]] & \zeta\,\alpha e[[3]] \\ \hline \xi\,\alpha e[[4]] & \eta\,\alpha e[[5]] & \zeta\,\alpha e[[6]] \\ \hline \xi\,\alpha e[[7]] & \eta\,\alpha e[[8]] & \zeta\,\alpha e[[9]] \\ \hline\end{array}\right); \quad \text{Hb} \vDash \frac{\text{J0d}}{\text{Jgd}}\ \text{HbΞ.SMSInverse[J0]};$$

```
F ⊨ IdentityMatrix[3] + Dg + Hb; JF ⊨ Det[F]; Cg ⊨ Transpose[F].F; {Em, ν, Qx, Qy, Qz} ⊢
 SMSReal[Table[es$$["Data", i], {i, Length[SMSGroupDataNames]}]];
{λ, μ} ⊨ SMSHookeToLame[Em, ν];
W ⊨ 1 / 2 λ (JF - 1) ^2 + μ (1 / 2 (Tr[Cg] - 3) - Log[JF]) - {Qx, Qy, Qz}.u;
wgp ⊢ SMSReal[es$$["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$$[NoSensParameters]
  See also: Elimination of local unknowns
```

File:	test.f	Size:	29 124
Methods	No.Formulae	No.Leafs	
SKR10	244	6902	

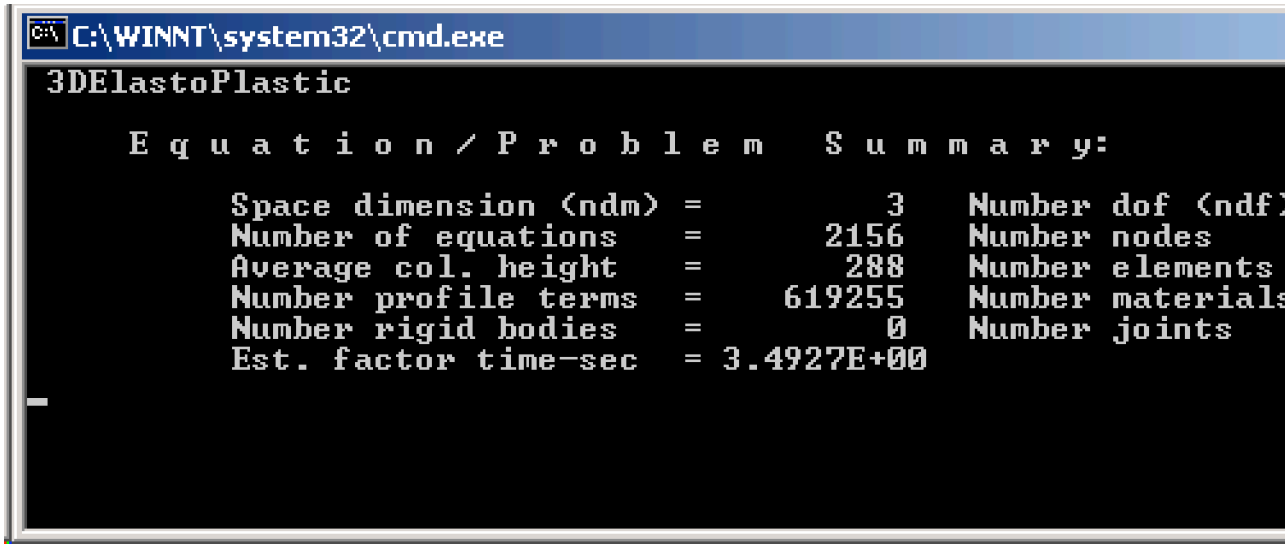
Test example: FEAP

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's Visual Studio project. See http://www.fgg.uni-lj.si/symech/User/AceInstallation.htm for details. The SMSFEAPRun function then starts FEAP with a beam.inp file as a standard FEAP input file and a beam.out file as output file.

```
SMSFEAPMake["ExamplesHypersolidFEAP"]

SMSFEAPRun["feap.inp"]
```




```
ReadList["feap.out", "Record"][[-4]]
```

ELFEN

About *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.

In order to put a new element in *ELFEN*© we need:

⇒ ELFEN© libraries (refer to Rockfield Software),

⇒ SMS.h and SMSUtility.f files (available in $BaseDirectory/Applications/AceGen/Include/ELFEN/ directory),

⇒ element source file.

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. Insted of one "Tangent and residul" user subroutine we have to generate two separate routines for the evaluation of the tangent matrix and the residual .

How to set paths to ELFEN's Visual Studio project is described in http://www.fgg.uni-lj.si/symech/User/AceInstallation.htm.

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 choosen accordingly to the element topology.

Abbreviation	*Description*	*Default value*
ELFEN$ElementModel	"B2" ⇒ two dimensional beam elements "B3" ⇒ three dimensional beam elements "PS " ⇒ two dimensional plane stress elements "PE " ⇒ two dimensional plane strain elements "D3" ⇒ three dimensional solid elements "AX" ⇒ axi−symmetric elements "PL" ⇒ plate elements "ME" ⇒ membrane elements "SH" ⇒ shell elements	"L1","LX"⇒"B2" "C1","CX"⇒"B3" "T1","T2","TX","Q1", "Q2","QX"⇒"PE" "P1","P2","PX","S1", "S2","SX"⇒"SH" "O1","O2","OX","H1", "H2","HX"⇒"D3"
ELFEN$NoStress	number of stress components	accordingly to the SMSTopology
ELFEN$NoStrain	number of strain components	accordingly to the SMSTopology
ELFEN$NoState	number of state variables	0

Additional ELFEN constants.

Here the additional constants for the 2D, plane strain element are defined.

```
ELFEN$ElementModel = "PE";
ELFEN$NoState = 0;
ELFEN$NoStress = 4;
ELFEN$NoStrain = 4;
```

Some of the standard interface data are interpreted in a ELFEN specific form as follows.

Standard form	*Description*	*FEAP interpretation*
es$$["SensType", *j*]	type of the *j*−th (current) sensitivity parameter	idata$$["SensType"]
es$$["SensTypeIndex", *j*]	index of the *j*−th (current) sensitivity parameter within the type group	idata$$["SensTypeIndex"]
nd$$[*i*, "sX", *j*, *k*]	initial sensitivity of the *k*− tk nodal coordinate of the *i* −th node with respect to the *j*− th shape sensitivity parameter	sxd$$[(i−1) SMSNoDimensions+k]

The ELFEN specific interpretation of the interface data.

ELFEN Interface

Parameter	*Description*	*type*
m*switch*	dimensions of the integer switch data array	integer *mswitch*
switch	integer type switches	integer *switch* (*mswitch*)
m*euvbl*	dimensions of the element variables vlues array	integer *meuvbl*
lesvbl	array of the element variables vlues	integer *lesvbl* (*meuvbl*)
nehist	number of element dependent history variables	integer *nehist*
jfile	output file (FORTRAN unit number)	integer *jfile*
m*order*	dimension of the node ordering array	integer *m order*
order	node ordering	integer *orde* (*morder*)
mgdata	dimension of the element group data array	integer *mgdata*
gdata	description of the element group specific input data values	character∗32 *gdata* (*mgdata*)
ngdata	number of the element group specific input data values	integer *ngdata*
mstate	dimension of the state data array	integer *mstate*
state	description of the element state data values	character∗32 *state* (*mstate*)
nstate	number of the element state data values	integer *nstate*
mgpost	dimension of the integration point postprocessing data array	integer *mgpost*
gpost	description of the integration point postprocessing values	character∗32 *gpost* (*mgpost*)
ngpost	total number of the integration point postprocessing values	integer *ngpost*
ngspost	number of sensitivity parameter dependent integration point postprocessing values	integer *ngspost*
mnpost	dimension of the integration point postprocessing data array	integer *mgpost*
npost	description of the integration point postprocessing values	character∗32 *npost* (*mnpost*)
nnpost	total number of the integration point postprocessing values	integer *nnpost*
nnspost	number of sensitivity parameter dependent integration point postprocessing values	integer *nnspost*

Parameter list for the SMSI*nnn* ELFEN *nnnn*'th user element subroutine.

Switch	*Description*	*type*
1	number of gauss points	output
2	number of sensitivity parameters	input

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.

Generation of element source code 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. However, due to the non-standard approach to the implementation of the Newton-Raphson loop in *ELFEN* result would not be the most efficient. More efficient implementation is obtained if the evaluation of the tangent matrix and residual vector are separated. 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.

Additionally, the non-standard evaluation of the Newton-Raphson loop makes implementation of the mixed FE models difficult. Thus only displacement element is generated.

The generated code is then incorporated into ELFEN.

```
<< "AceGen`";
SMSInitialize["test", "Environment" → "ELFEN"];
SMSTemplate["SMSTopology" → "H1", "SMSSymmetricTangent" → True
  , "SMSGroupDataNames" -> {"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$$["Data", i], {i, Length[SMSGroupDataNames]}]];
{λ, μ} ⊨ SMSHookeToLame[Em, ν];
W ⊨ 1 / 2 λ (JF - 1) ^2 + μ (1 / 2 (Tr[Cg] - 3) - Log[JF]) - {Qx, Qy, Qz}.u;
wgp ⊢ SMSReal[es$$["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: 27813
Methods	No.Formulae	No.Leafs
SKR2999	182	5160

Test example: ELFEN

Here is the generated element compiled and linked into the ELFEN's Visual Studio project. See http://www.fgg.uni-lj.si/symech/User/AceInstallation.htm for details. The SMSELFENRun function then starts ELFEN with a ELFENExample.dat file as a input file and a tmp.res file as output file. The ELFEN input data file for the one element test example is available in a $BaseDirectory/Applications/AceGen/Include/ELFEN/ directory.

```
SMSELFENMake["ExamplesHypersolidELFEN"]
```

```
SMSELFENRun["ELFEN.dat"]
```

ABAQUS

■ About *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). Currecntly the interface for *ABAQUS*© support direct, static implicit analysis. The interface does not support elements with the internal degrees of freedom.

In order to put a new element in *ABAQUS*© we need:

⇒ *ABAQUS*©,

⇒ sms.*h* and *SMSUtility.for* files (available in $BaseDirectory/Applications/AceGen/Include/ABAQUS/ directory),

⇒ element source file.

Paths to the *ABAQUS*© are set by the SMSABAQUSProject variable in the initialization file Paths.m. Paths.m initialisation file is located at the directory $BaseDirectory/Applications/AceGen/Paths.m or $BaseDirectory/Applications/AceFEM/Paths.m.

The SMSABAQUSProject variable contains:

1 - command line that compiles the element source file and builds element object file

2 - the name used to run ABAQUS from command line

Example: SMSABAQUSProject = {"df /compile_only /optimize:4 /list:SMSCompile.txt /show:nomap" ,"abaqus"};

The curret ABAQUS interface is tested for Compaq Visual Fortran 6.6 and ABAQUS 6.4!

Example

```
<< AceGen`;
```

This runs ABAQUS with ABAQUSExample.inp as input file and SED3H1DFHYH1NHookeA element as user element.

The element source code is automatically downloaded from OL shared library (see also AceShare).

The ABAQUSExample.inp is available in $BaseDirectory/Applications/AceGen/Include/ABAQUS/ directory.

```
SMSABAQUSRun["ABAQUSExample.inp", "UserElement" → "OL:SED3H1DFHYH1NHookeA"]
```

3D Solid FE 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.

Generation of element source code for ABAQUS environment

The *AceGen* input presented in previous example 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.

```
<< "AceGen`";
SMSInitialize["test", "Environment" → "ABAQUS"];
SMSTemplate["SMSTopology" → "H1", "SMSSymmetricTangent" → True
  , "SMSGroupDataNames" -> {"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$$["Data", i], {i, Length[SMSGroupDataNames]}]];
{λ, μ} ⊨ SMSHookeToLame[Em, ν];
W ⊨ 1 / 2 λ (JF - 1) ^2 + μ (1 / 2 (Tr[Cg] - 3) - Log[JF]) - {Qx, Qy, Qz}.u;
wgp ⊢ SMSReal[es$$["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: 24577
Methods	No.Formulae	No.Leafs
SKR	182	5233

Test example: ABAQUS

Here is the generated element compiled and linked into the ABAQUS's. The SMSABAQUSRun function then starts ABAQUS with a ABAQUSExample.inp file as a input file and a tmp.res file as output file. The ABAQUS input data file for the one element test example is available in a $BaseDirectory/Applications/AceGen/Include/ABAQUS/ directory.

```
SMSABAQUSRun["ABAQUSExample", "UserElement" -> "ExamplesHypersolidABAQUS"]
```

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 pro start the program and install *Mathematica* definitions to call functions
SMSInstallMathLink[]	create *MathLink* executable from the last generated *AceGen* source co

option name	*default value*	
"Optimize"	Automatic	use additional compiler optimization
"PauseOnExit"	False	pause before exiting the *MathLink* executable
"Console"	True	start the executable as console application
"Platform"	Automatic	"32" ⇒ 32 bit operating system (all operating systems Windows, Unix, Mac) "64" ⇒ 64 bit operating systems (Mac and Windows)

Options for SMSInstallMathLink.

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 executued 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	
"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.

SMSLinkNoEvaluations[*source*]	returns the number of evaluations of *MathLink* functions compiled fr *source* source code file during the *Mathematica* session (run time con
SMSLinkNoEvaluations[]	≡ SMSLinkNoEvaluations[*last AceGen session*]

For more examples se 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*

```
<< 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[];
```

{ui1, ui2, ui3}

File:	test.c	Size: 1838
Methods	No.Formulae	No.Leafs
Test	6	81

```
SMSInstallMathLink[]
```

```
{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.

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

```
{1.37858, 3.00958, 0.945489}
```

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/.

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

ABAQUS

FEAP

ELFEN

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.

```
<< 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)

```
SMSInstallMathLink[]
```

```
{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.

```
test[1.9,-1.2,3.,0.0001,10]
```

```
{1.93318, -1.24573}
```

```
x =.; y =.; a = 3.;
Solve[{a x y + x^3 == 0, a - x y^2 == 0}, {x, y}]
```

```
{{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

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 a ϕ indicates temperature, a k is the conductivity and a 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	derivation *time* (*s*)	solution time (s)	No. of variables	memory (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.

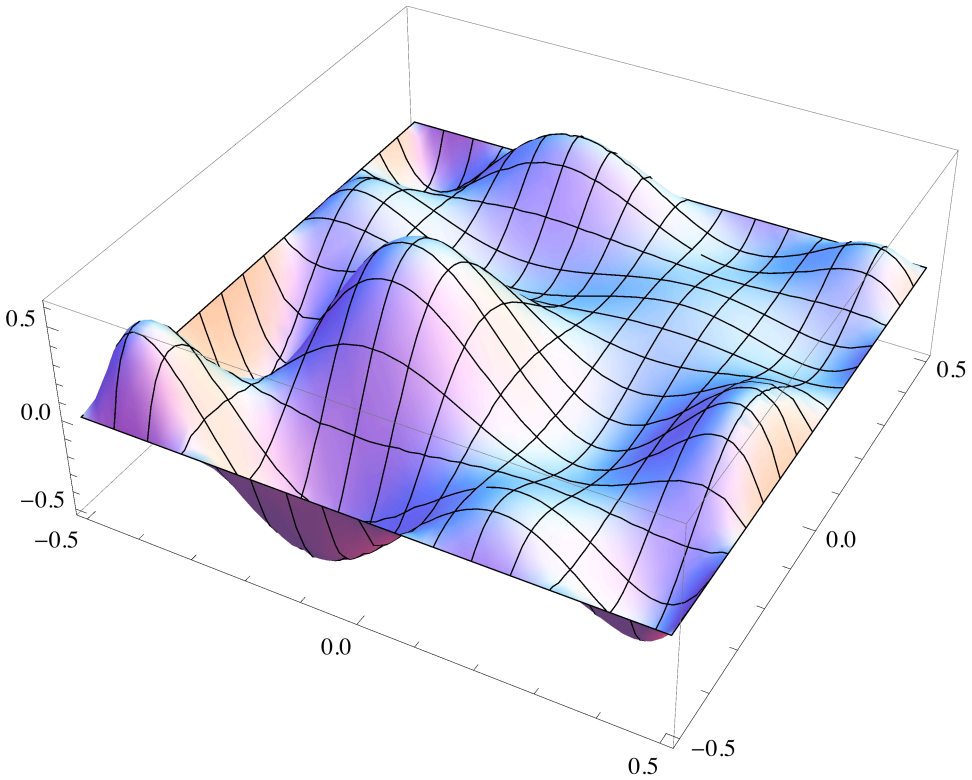
```
<< AceGen`;
Clear[x, y, z, α];
kcond = 0.58; Q = 500;
order = 5;
nterm = (order - 1) (order - 1) (order)
```

```
80
```

Here the fifth order Lagrange polynomials are constructed in three dimensions.

```
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;

poly[[28]]
Plot3D[poly[[28]] /. z → 0.5, {x, -0.5, 0.5}, {y, -0.5, 0.5}, PlotRange → All]
```

$$1.76606\times10^{7}\,(-0.5+x)\,(-0.3+x)\,(-0.1+x)\,(0.3+x)\,(0.5+x)\,(-0.5+y)\,(-0.3+y)\,(-0.1+y)\,(0.3+y)\,(0.5+y)\,(-1.+z)\,(-0.8+z)\,(-0.4+z)\,(-0.2+z)\,z$$



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].

```
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.

```
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
```

```
10.8556096
```

G. Gradient based optimization

```
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
```

```
{993, 55.8724}

59.5956944
```

N. Newton method based optimization

```
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
```

```
{4, 55.8724}

177.6454416
```

AceGen code generation

The subsection Definitions has to be executed before the current subsection.

```
start = SessionTime[]; SMSInitialize["Thermal",
 "Environment" -> "MathLink", "Mode" → "Prototype", "ADMethod" -> "Forward"]

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)
 )
```

```
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[];

SMSModule["GThermal", Real[a$$[nterm], g$$[nterm]], "Input" → a$$, "Output" → g$$];
SMSExport[Table[0, {nterm}], g$$];
SMSDo[i, 1, gp // Length];
 Π ⊨ Πf[i];
 SMSDo[j, 1, nterm];
  δΠ ⊨ SMSD[Π, ag, j, "Method" -> "Forward"];
  SMSExport[δΠ, g$$[j], "AddIn" → True];
 SMSEndDo[];
SMSEndDo[];

derivation = SessionTime[] - start

6.5794608

SMSModule["HThermal",
   Real[a$$[nterm], h$$[nterm, nterm]], "Input" → a$$, "Output" → h$$];
SMSDo[i, 1, nterm];
 SMSDo[j, 1, nterm];
  SMSExport[0, h$$[i, j]];
 SMSEndDo[];
SMSEndDo[];
SMSDo[i, 1, gp // Length];
 Π ⊨ Πf[i];
 SMSDo[j, 1, nterm];
  δΠ ⊨ SMSD[Π, ag, j, "Method" -> "Forward"];
  SMSDo[k, 1, nterm];
   hij ⊨ SMSD[δΠ, ag, k, "Method" -> "Forward"];
   SMSExport[hij, h$$[j, k], "AddIn" → True];
  SMSEndDo[];
 SMSEndDo[];
SMSEndDo[];

SMSWrite[];
```

```
Method : FThermal 162 formulae, 6025 sub-expressions
Method : GThermal 161 formulae, 6133 sub-expressions
Method : HThermal 79 formulae, 4606 sub-expressions
[11] File created : Thermal.c Size : 133 849
```

```
SMSInstallMathLink["Optimize" → False]
derivation = SessionTime[] - start
```

```
{SMSSetLinkOption[Thermal, {i_Integer, j_Integer}], SMSLinkNoEvaluations[Thermal],
 FThermal[a_?(ArrayQ[#1, 1, NumberQ] && Dimensions[#1] === {80} &)],
 GThermal[a_?(ArrayQ[#1, 1, NumberQ] && Dimensions[#1] === {80} &)],
 HThermal[a_?(ArrayQ[#1, 1, NumberQ] && Dimensions[#1] === {80} &)]}
```

```
30.4237472
```

AceGen Solution

G. Gradient based optimization

```
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}
```

```
{931, 55.8724, 2.9943056}
```

N. Newton method based optimization

```
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}
```

```
{4, 55.8724, 0.7811232}
```

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.

```
<< 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;
```

```
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
```

```
810
```

Direct use of Mathematica

The subsection Definitions have to be executed before the current subsection.

```
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
```

```
0.4105904
```

G. Gradient based optimization

```
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}
```

```
{1685, 57.5034, 11.6767904}
```

N. Newton method based optimization

```
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}
```

```
{4, 57.5034, 1.0915696}
```

AceGen code generation

The subsection Definitions have to be executed before the current subsection.

```
start = SessionTime[]; SMSInitialize["Thermal",
 "Environment" -> "MathLink", "Mode" → "Prototype", "ADMethod" -> "Backward"]
```

```
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]])
 )
```

```
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[];

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[];

derivation = SessionTime[] - start
```

```
1.5822752
```

```
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[];

SMSWrite[];
```

```
Method : FThermal 32 formulae, 471 sub-expressions
Method : GThermal 42 formulae, 550 sub-expressions
Method : HThermal 38 formulae, 559 sub-expressions
[2] File created : Thermal.c Size : 11 891
```

```
SMSInstallMathLink["Optimize" → True]
derivation = SessionTime[] - start
```

```
{SMSSetLinkOption[Thermal, {i_Integer, j_Integer}], SMSLinkNoEvaluations[Thermal],
 FThermal[ndof_?NumberQ, nt_?(ArrayQ[#1, 1, NumberQ] && Dimensions[#1] === {3} &),
  index_?(ArrayQ[#1, 1, NumberQ] &),
  a_?(ArrayQ[#1, 1, NumberQ] &), mc_?(ArrayQ[#1, 1, NumberQ] &)],
 GThermal[ndof_?NumberQ, nt_?(ArrayQ[#1, 1, NumberQ] && Dimensions[#1] === {3} &),
  index_?(ArrayQ[#1, 1, NumberQ] &),
  a_?(ArrayQ[#1, 1, NumberQ] &), mc_?(ArrayQ[#1, 1, NumberQ] &)],
 HThermal[ndof_?NumberQ, nt_?(ArrayQ[#1, 1, NumberQ] && Dimensions[#1] === {3} &),
  index_?(ArrayQ[#1, 1, NumberQ] &),
  a_?(ArrayQ[#1, 1, NumberQ] &), mc_?(ArrayQ[#1, 1, NumberQ] &)]}
```

```
7.9414192
```

AceGen Solution

G. Gradient based optimization

```
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}
```

```
{1599, 57.5034, 6.3090720}
```

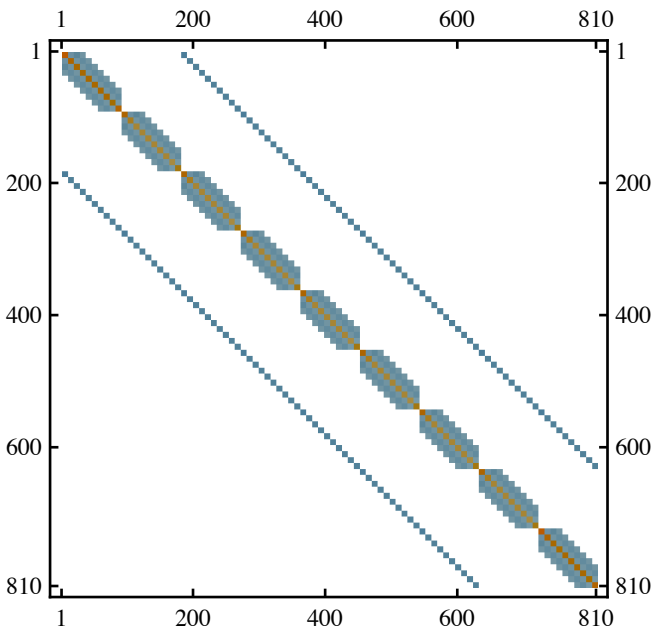
N. Newton method based optimization

```
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}
```

```
{4, 57.5034, 0.8011520}
```

The tangent matrix is in the case of finite difference approximation extremely sparse.

```
MatrixPlot[
 HThermal[nterm, {nx, ny, nz}, indexb, Join[0 ϕi, bound], {dlx, dly, dlz, kcond, Q}]]
```



C) Finite element method

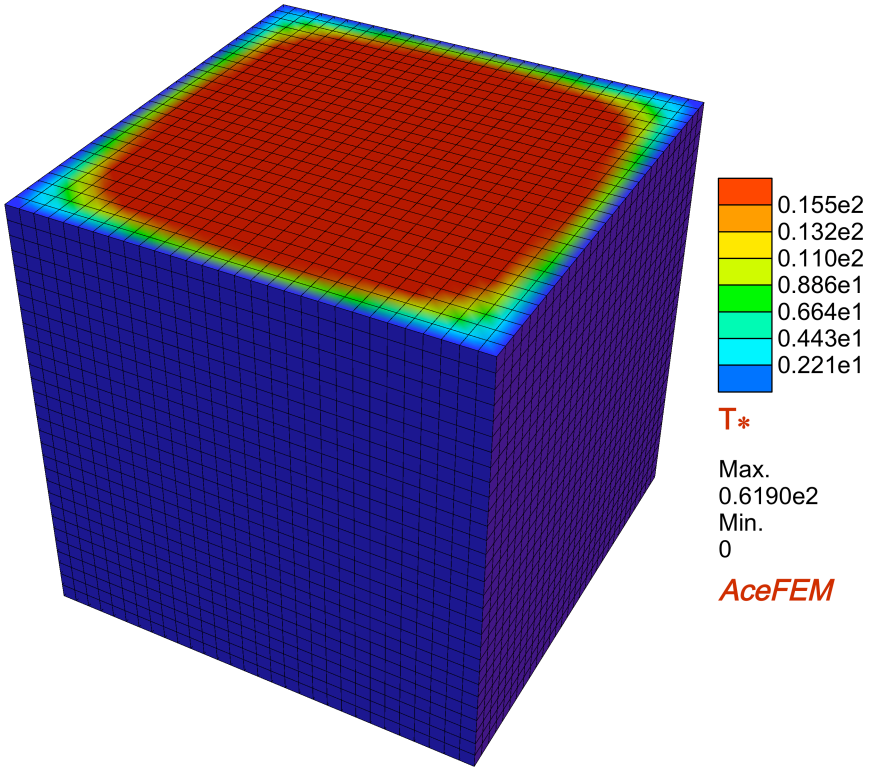
Solution

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 .

```
<< 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[];

SMTNextStep[0, 1];
SMTNewtonIteration[];

SMTPostData["T*", {0, 0, 0.5}]
SessionTime[] - start
```

```
55.8765

27.7098448
```

```
SMTShowMesh["Mesh" → True, "Elements" → True, "Field" → "T*"]
```



Troubleshooting and New in version

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 $\mathrm{Sin}[x]^2 + \mathrm{Cos}[x]^2$ 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 Expression Optimization,).
- 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

```
<< 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

```
<< 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

```
<< 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

```
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 is infinite. 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

Erroneous input

```
a < b < c

FullForm[a < b < c]

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

```
Unprotect[CForm];
CForm[Less[a_, b_, c_]] := a < b && b < c;
Protect[CForm];

CForm[a < b < c]
```

MathLink

- if the compilation is to slow restrict compiler optimization with SMSInstallMathLink["Optimize"→False]
- in the case of sudden crash of the *MathLink* program use SMSInstallMathLink["PauseOnExit"→True] to see the printouts generated by SMSPrint

New in version

1. Mathematica syntax - AceGen syntax
2. new "in-cell" form of the SMSIf and SMSDo constructs (Program Flow Control) enables more direct transition from the *Mathematica* input into the AceGen input
3. new commands `SMSSwitch` `SMSWhich`
4. new boundary conditions sensitivity types (See also: SMTSensitivity, SMTAddSensitivity, Standard user subroutines, Solid, Finite Strain Element for Direct and Sensitivity Analysis, Parameter, Shape and Load Sensitivity Analysis of Multi-Domain Example .)
5. M switch for Node Identification data enables formulation of multi-field problems
6. IMPORTANT! The input syntax of the Dependency option of SMSReal and SMSFreeze commands has been changed. In the case of scalar expression, the input form $\texttt{SMSFreeze}\left[\texttt{exp, Dependency -> }\left\{\{p_1, p_2, \ldots\}, \left\{\frac{\partial \exp}{\partial p_2}, \frac{\partial \exp}{\partial p_1}, \ldots\right\}\right\}\right]$ is no longer supported. Please use the $\texttt{SMSFreeze}\left[\texttt{exp, Dependency -> }\left\{\left\{p_1, \frac{\partial exp}{\partial p_1}\right\}, \left\{p_2, \frac{\partial exp}{\partial p_2}\right\}, \ldots\right\}\right]$ input form instead.
7. Detailed documentation and new options for creation of User Defined Functions
8. New form of SMSFreeze function

Reference Guide

AceGen Session

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 name	*default value*	
"Language"	"*Mathematica*"	source code language
"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 the Expressions

Options for *SMSInitialize*.

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	*description*	Language
"None"	plain code	defined by "Language" option
"*MathLink*"	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)	"C"
"User"	arbitrary user defined finite element environment (see Standard FE Procedure , User defined environment interface)	defined by "Language" option
"AceFEM"	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 Structure)	"C"
"AceFEM−MDriver"	*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)	"*Mathematica*"
"FEAP"	research finite element environment written in *FORTRAN* (see FEAP)	"Fortran"
"ELFEN"	commercial finite element environment written in *FORTRAN* (see ELFEN)	"Fortran"
"ABAQUS"	commercial finite element environment written in *FORTRAN* (see ABAQUS)	"Fortran"

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.

```
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

Syntax of SMSModule function.

parameter types	
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 name	*default value*	
"Verbatim"->"text"	None	string "text" is included at the end of the declaration block of the source code verbatim
"Input"	All	list of input parameters
"Output"	All	list of output parameters

Options for SMSModule.

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 can be used instead of SMSModule for the definition of the standard user subroutines for supported finite element numerical environments.

This creates a 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" .

```
<<AceGen`;
SMSInitialize["test","Language"->"Fortran"];
SMSModule["sub1",Real[x$$,y$$[5]],Integer[i$$],Real[z$$],
          "mytype"[m$$],"Verbatim"->"COMMON /xxx/a(5)"];
SMSWrite[];
FilePrint["test.f"]
```

```
Method : sub1 0 formulae, 0 sub-expressions

[0] File created : test.f Size : 814

!*************************************************************
!* AceGen    2.103 Windows (18 Jul 08)                       *
!*           Co. J. Korelc  2007            18 Jul 08 15:41:06*
!*************************************************************
! User : USER
! Evaluation time                 : 0 s     Mode  : Optimal
! Number of formulae              : 0       Method: Automatic
! Subroutine                      : sub1 size :0
! Total size of Mathematica  code : 0 subexpressions
! Total size of Fortran code      : 254 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(5001),x,y(5),z
      TYPE (mytype)::m
      COMMON /xxx/a(5)
      END
```

SMSWrite

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

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 name	*default value*	
"Splice"	{}	list of files interprated (see Splice) 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	due to the limitations of Fortran compilers, break large Fortran expressions into subexpressions of the size less than "MaxLeafCount" (size is measured by the LeafCount function)
"LocalAuxiliaryVariables "	False	The vector of auxiliary variables is defined locally for each module.

Options for *SMSWrite.*

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. Options "IncludeNames" and "IncludeAllFormulas" are useful during the "debugging" period. They have effect only in the case that *AceGen* session was initiated in the "*Debug*" or "*Prototype*" mode. Option "OptimizingLoops" has effect only in the case that *AceGen* session was initiated in the "*Optimal*" or a higher mode.

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

See also: Standard AceGen Procedure

This write the generated code on the file "source.c" and prepends contents of the file "test.mc" interpreted by the Splice command.

```
<<AceGen`;

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

```
FilePrint["test.mc"]

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

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

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 parameter *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
SMSVerbatim[…,"CheckIf"->False]	Since the effect of the SMSVerbatim statement can not be predicted, some optimization of the code can be prevented by the "verbatim" statement. With the option "CheckIf"->False, the verbatim code is ignored for the code optimization.
SMSVerbatim[…,"Close"->False]	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.

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.

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 automatic differentiation procedure.

```
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
```

```
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);
};
```

SMSPrint

SMSPrint[*expr1*,*expr2*,...,*options*]	create a source code sequence that prints out all the expressions $expr_i$ accordingly to the given options
SMSPrint["Message", *expr1*,*expr2*,...,]	≡ SMSPrint[expr1,expr2,..., "Optimal"−>True, "Output"−>"Console","Condition"−>None] prints out message to standard output device

option	*description*	default
"Output"	"Console" ⇒standard output device {"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)	"Console"
"Optimal"	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.	False
"Condition"	at the run time the print out is actually executed only if the given logical expression yields True	None

General options for the *SMSPrint* function.

The SMSPrint function is active only in **"Debug"** and **"Prototype"** mode while the SMSPrintMessage function **always** creates source code.

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.

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	*standard output device* ("*Console*")
"*Mathematica*"	current notebook
"C"	console window (printf (…)
"Fortran"	console window (write(∗,∗) …)
"Matlab"	matlab window (disp (…)

Options	description
"Output"->"File"	create a source code sequence that prints out to the standard output file associated with the specific numerical environment (if exist)
"Condition"->"DebugElement"	Within some numerical environment there is an additional possibility to limit the print out. With the "DebugElement" option the print out is executed accordingly to the value of the SMTIData["DebugElement"] environment variable (if applicable): SMTIData["DebugElement",−1] ⇒ print outs are active for all elements SMTIData["DebugElement",0] ⇒ print outs are disabled (default value) SMTIData["DebugElement",*i*] ⇒ print out is active for *i*−th element

Additional values for SMSPrint options for numerical environments (AceFEM,...).

Example 1: printing out to all output devices - C language

```
<< 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[];
```

File:	test.c	Size: 1063
Methods	No.Formulae	No.Leafs
test	4	11

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

```
/************************************************************
* AceGen    3.301 Windows (27 Dec 11)                       *
*           Co. J. Korelc  2007            27 Dec 11 19:21:48*
************************************************************
User      : USER
Notebook : AceGenSymbols.nb
Evaluation time                 : 0 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            : 456 bytes*/
#include "sms.h"

/****************** S U B R O U T I N E ********************/
void test(double v[5001],double (*x))
{
FILE *SMSFile;
printf("\n%s %g ","pi=",(double)0.3141592653589793e1);
v[2]=Time();
SMSFile=fopen("test.out","a");if(SMSFile!=NULL){
fprintf(SMSFile,"\n%s %g ","time=",(double)v[2]);
fclose(SMSFile);};
if((*x)>0e0){
SMSFile=fopen("test.out","a");if(SMSFile!=NULL){
fprintf(SMSFile,"\n%s %g ","e=",(double)0.2718281828459045e1);
fclose(SMSFile);}
};
};
```

Example 2: printing out to all output devices - Fortran language

```
<< 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[];
```

File:	test.f	Size: 1055
Methods	No.Formulae	No.Leafs
test	6	15

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

```
!************************************************************
!* AceGen    3.301 Windows (27 Dec 11)                      *
!*           Co. J. Korelc  2007            27 Dec 11 19:21:41*
!************************************************************
! User      : USER
! Notebook  : AceGenSymbols.nb
! 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      : 451 bytes

!****************** S U B R O U T I N E *********************
      SUBROUTINE test(v,x)
      IMPLICIT NONE
      include 'sms.h'
      DOUBLE PRECISION v(5001),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 3: printing out from numerical environment

```
<< 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[];
```

File:	test.m	Size: 2209
Methods	No.Formulae	No.Leafs
SMT`SKR	2	4

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

```
(************************************************************
* AceGen    3.301 Windows (27 Dec 11)                       *
*           Co. J. Korelc  2007            27 Dec 11 19:21:56*
************************************************************
User      : USER
Notebook  : AceGenSymbols.nb
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",
   0,0,0,0,0}
 ,"T1",
 SMCToMMAString[{}],
 SMCToMMAString[{}],
 SMCToMMAString[{}],
 {1, 2, 3, 0, 1, 2, 3, -1},{2, 2, 2},{},{}
 ,If[gd===={}, {},gd]
 ,q3[[2]]//Transpose,{{1, 0, 0}, {0, 1, 0}, {0, 0, 1}},
"AdditionalData","NoNodeStorage","NoNodeData",{-1, -1, -1},
 SMCToMMAString["{}&"],
 {"D", "D", "D"},
 SMCToMMAString["{}&"],
 SMCToMMAString[{}],
 SMCToMMAString[""],
 SMCToMMAString[""],
 SMCToMMAString[""],
 "",
 "",
 "",
 "",
 {},
 {},
 {},
 {},
 "","",{},{},
 {},
 {},
 {},
 {1., 1., 1.},
  FromCharacterCode[{}],
  {3.301,3.301,8.}
  };

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

q1[[3,9]]=Round[0];
q1[[17]]=Round[{Round[q1[[3,2]]*idata$$[[41]]],
Round[q1[[3,2]]*idata$$[[41]]], Round[q1[[3,2]]*idata$$[[41]]]}];
q1[[16]]=Round[{0, 0, 0}];
q1[[3,10]]=Round[0];
q1[[3,24]]=Round[0];
q1[[15]]=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];
If[idata$$[[65]]==-1 || idata$$[[32]]==idata$$[[65]],
PutAppend[{"load=",rdata$$[[1]]},SMTSession[[10]]]
```

```
];
];
```

SMSPrintMessage

SMSPrintMessage[*expr1*,*expr2*,...,]	≡ SMSPrint[expr1,expr2,..., "Optimal"->True, "Output"->"Console","Condition"->None] prints out message to standard output device

The SMSPrint function is active only in **"Debug"** and **"Prototype"** mode while the SMSPrintMessage function **always** creates source code.

See also: SMSPrint

Basic Assignments

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].

See also: Mathematica syntax - AceGen syntax, Auxiliary Variables, Expression Optimization

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	$e\longrightarrow e$,$e\nearrow e$,$e\rightleftharpoons e$,$e\rightarrowtail e$,etc.
Logic operators	$\forall_e e$, e&&e , $e\bigvee e$, $e\vdash e$, etc.
AceGen operators	$\vDash$,$\vdash$,$\dashv$,$\dashv$
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 opreators.

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

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

```
1
```

V

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.

See also: Mathematica syntax - AceGen syntax, Auxiliary Variables, Expression Optimization

The new auxiliary variables are created for all expressions.

```
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.

See also: Mathematica syntax - AceGen syntax, Auxiliary Variables, Expression Optimization, Program Flow Control

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.

See also: Mathematica syntax - AceGen syntax, Auxiliary Variables, Expression Optimization, Program Flow Control

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

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

$_1$X

$_2$X

$_3$X

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

```
Method : sub 4 formulae, 16 sub-expressions

[0] File created : test.f Size : 808
```

```
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
```

SMSInt

SMSInt[*exp*]	create an integer type auxiliary variable

If an expression contains logical type auxiliary or external variables then the expression is automatically considered as logical type expression. Similarly, if an expression contains real type auxiliary or external variables then the expression is automatically considered as real type expression and if it contains only integer type auxiliary variables it is considered as integer type expression. With the *SMSInt* function we force the creation of integer type auxiliary variables also in the case when the expression contains some real type auxiliary variables.

See also: Auxiliary Variables, SMSM.

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.

See also: Auxiliary Variables, SMSM .

SMSVariables

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

Example

```
<< AceGen``;
SMSInitialize["test"];
SMSModule["Test", Real[a$$]];
a ⊢ SMSReal[a$$];
```

$$M \vDash \begin{pmatrix} a & a^2 \\ a^2 & 0 \end{pmatrix}$$

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

```
SMSVariables[M]
```

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

Symbolic-numeric Interface

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 .

option name	*default value*	
"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 for SMSReal.

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 the Expressions).

See also: Expression Optimization , Symbolic-Numeric Interface, User Defined Functions

SMSInteger

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

Introduction of integer type external variables .

option name	*default value*	
"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 for SMSInteger.

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.

See also: SMSReal, Symbolic-Numeric Interface

SMSLogical

SMSLogical[*exte*]	create logical type external data object with definition *exte*

option name	*default value*	
"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 for SMSLogical.

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$$]).

See also: SMSReal, Symbolic-Numeric Interface

SMSRealList

SMSRealList[{eID_1,eID_2,…}, *array_Function*]	create a list of real type external data objects that correspo to the list of array element identifications {eID_1,eID_2,…} and represents consecutive elements of the array
SMSRealList[*pattern*]	gives the real type external data objects that correspond to elements which array element identification *eID* match pat
SMSRealList[*pattern*,*code_String*]	gives the data accordingly to the *code* that correspond to elements which array element identification *eID* match pat

Introduction of the list of real type external variables .

option name	*default value*	
"Description"−>{…}	{eID_1,eID_2,…}	a list of descriptions that corresponds to the list of array element identifications {eID_1,eID_2,…}
"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,…}

Options for SMSRealList

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 commands 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 form 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.

See also: SMSReal

Example

```
<< AceGen`

SMSInitialize["test", "Language" -> "C"];
SMSModule["test", Real[a$$[10], b$$[10], c$$[100]], Integer[L$$, i$$]];
SMSRealList[{"a1", "a2"}, a$$[#] &]
```

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

```
SMSRealList[{"a3", "a4"}, a$$[#] &]
```

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

```
SMSRealList["a3"]
```

$a\$\$_3$

```
SMSRealList[{"b1", "b2"}, b$$[#] &, "Length" → 5, "Index" → 2]
```

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

```
SMSRealList[{"b3", "b4"}, b$$[#] &, "Length" → 20, "Index" → 4]
```

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

The arguments "Length" and "Index" are 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 variables assigned to *L* and *i* can be different in different subroutines!!

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

$\{c\$\$_{1+i}, c\$\$_{1+i+2L}\}$

```
SMSRealList[Array["β", 2], c$$[#] &, "Length" → Hold[ L], "Index" → Hold[i]]
```

$\{c\$\$_{i+4L}, c\$\$_{i+5L}\}$

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

ID	β[1]	β[2]
AceGen	$c\$\$_{i+4L}$	$c\$\$_{i+5L}$
Plain	c$$[(int)[i$$] + 4 (int)[L$$]]	c$$[(int)[i$$] + 5 (int)[L$$]]
Export	$c\$\$[i + 4L]$	$c\$\$[i + 5L]$

```
SMSRealList["β"[_], "Length"]
```

2 (int)[L$$]

SMSExport

SMSExport[*exp*,*ext*]	export the expression *exp* to the external variable *ext*
SMSExport[{*exp1*,*exp2*,...,*expN*},*ext*]	≡ SMSExport[{*exp1*,*exp2*,...,*expN*},Table[*ext*[i],{i,1,*N*}]] export the list of expressions {exp1,exp2,...} to the external array ext formed as Table[ext[i],{i,1,N}]
SMSExport[{*exp1*,*exp2*,...,*expN*}, {*ext1*,*ext2*,...,*ex2N*}]	export the list of expressions {*exp1*,*exp2*,...} to a matching list of the external variables {*ext1*,*ext2*,...}
SMSExport[*exp*, *ext*, "AddIn"->True]	add the value of *exp* to the current value of the external variable *ext*

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.

See also: Symbolic-Numeric Interface

```
<< 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

```
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
```

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

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. Declaration of output parameters and their later use in a program should follow *AceGen* rules for the declaration and use of external variables as described in chapter Symbolic-Numeric Interface (e.g. Real[x$$,"Subordinate"→*sc*], Integer[i$$[5],"Subordinate"→*sc*], Logical[b$$,"Subordinate"→*sc*]). 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.

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"->$\left\{\left\{v_1, \frac{\partial exte}{\partial v_1}\right\}, \left\{v_2, \frac{\partial exte}{\partial v_2}\right\}, \ldots\right\}$ (see also SMSReal).

More detailed description and examples are given in section User Defined Functions .

option name	*description*	*default value*
"Dependency"–> $\left\{\left\{v_1,\frac{\partial exte}{\partial v_1}\right\},\left\{v_2,\frac{\partial exte}{\partial v_2}\right\},\ldots\right\}$	defines partial derivatives of output parameters with respect to input parameters	{}
"System"–>*truefalse*	the subroutine that is called has been generated by *AceGen*	True
"ArgumentsByValue"	By default in *AceGen*, the arguments are passed to subroutine by reference. This can be changed with "ArgumentsByValue"–>True option.	Automatic

Options for *SMSCall*.

Example

This generates **user AceGen module** $f = \text{Sin}(a_1\, x + a_2\, x\text{^}2 + a_3\, x^3)$ 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.

```
<< AceGen`;
SMSInitialize["test", "Language" → {"Fortran", "C", "Mathematica"}[[2]]];

SMSModule["f", Real[x$$, a$$[3], y$$, dy$$, ddy$$]];
x ⊨ SMSReal[x$$];
{a1, a2, a3} ⊨ SMSReal[Array[a$$, 3]];
y ⊨ Sin[a1 x + a2 x^2 + a3 x^3];
dy ⊨ SMSD[y, x];
ddy ⊨ SMSD[y, x];
SMSExport[{y, dy, ddy}, {y$$, 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$$, "Subordinate" → fo, "Dependency" -> {z, dfdz}];
dw ⊨ SMSD[f, w];
ddw = SMSD[dw, w];
SMSExport[dd, r$$];

SMSWrite[];
```

Smart Assignments

SMSFreeze

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*
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)
SMSFreeze[*symbol*, {exp_1,exp_2,{…}..}, *options*]	create data objects that represent elements of arbitrarily structured list of expressions {exp_1,exp_2,{…}..} accordingly to given options *options*. New auxiliary variables with the values {exp_1,exp_2,{…}..} 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)

Imposing restrictions on an optimization procedure.

general option	*default value*	
"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

General options for SMSFreeze.

option for structured lists	*default value*	
"Ignore"−>*crit*	(False&)	the *SMSFreeze* functions 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	≡ "Ignore"−>NumberQ whether to apply *SMSFreeze* functions 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*]	
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,…

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

SMSFreeze[{exp_1,exp_2, …}, "*Dependency*"->*value*]	
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, …} 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,…} to {p_1, p_2,…} 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,…}

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$\left[\text{exp, "Dependency"} \to \left\{\left\{p_1, \frac{\partial exp}{\partial p_1}\right\}, \left\{p_2, \frac{\partial exp}{\partial p_2}\right\}, \ldots, \left\{p_n, \frac{\partial exp}{\partial p_n}\right\}\right\}\right]$ 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 $\frac{\partial exp}{\partial p_1},\frac{\partial exp}{\partial p_n},\ldots,\frac{\partial exp}{\partial p_n}$ (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*.

See also: Exceptions in Differentiation, Auxiliary Variables

Basic Examples

```
<< 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!

```
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.

```
{Sin[x], SMSFreeze[Sin[x]]} // SMSEvaluate
```

```
{0.67104233, 0.66222981}
```

New auxiliary variable that represents original expression can be introduced by

```
xf ⊢ SMSFreeze[Sin[x]];
xf
```

```
xf
```

or by

```
SMSFreeze[xf, Sin[x]];
xf
```

```
xf
```

Options

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

```
<< 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

```
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.

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

$$\begin{pmatrix} Mf_{11} & Mf_{12} & Mf_{13} \\ Mf_{21} & Mf_{22} & Mf_{23} \\ Mf_{31} & Mf_{32} & Mf_{33} \end{pmatrix}$$

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

```
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.

```
SMSFreeze[Mf, M, "Ignore" -> NumberQ];
Mf // MatrixForm
Mf // SMSEvaluate // MatrixForm
```

$$\begin{pmatrix} Mf_{11} & Mf_{12} & Mf_{13} \\ Mf_{21} & 2 & Mf_{23} \\ Mf_{31} & 0 & \frac{1}{2} \end{pmatrix}$$

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

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.

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

$$\begin{pmatrix} Mf_{11} & Mf_{23} & Mf_{13} \\ Mf_{23} & 2 & Mf_{23} \\ -Mf_{13} & 0 & \frac{1}{2} \end{pmatrix}$$

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

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.

```
SMSFreeze[Mf, M, "Symmetric" -> True, "Ignore" -> NumberQ];
Mf // MatrixForm
Mf // SMSEvaluate // MatrixForm
```

$$\begin{pmatrix} \boxed{Mf_{11}} & \boxed{Mf_{21}} & \boxed{Mf_{31}} \\ \boxed{Mf_{21}} & 2 & 0 \\ \boxed{Mf_{31}} & 0 & \frac{1}{2} \end{pmatrix}$$

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

Here all the auxiliary variables in original expression are replaced by new auxiliary variables with random signatures.

```
SMSFreeze[Mf, M, "Variables" -> True];
Mf // MatrixForm
Mf // SMSEvaluate // MatrixForm
```

$$\begin{pmatrix} \boxed{\mathbb{F}_1} & 2\,\boxed{\mathbb{F}_1} & \mathrm{Cos}\left[\boxed{\mathbb{F}_1}\right] \\ 2\,\boxed{\mathbb{F}_1} & 2 & 2\,\boxed{\mathbb{F}_1} \\ -\mathrm{Cos}\left[\boxed{\mathbb{F}_1}\right] & 0 & \frac{1}{2} \end{pmatrix}$$

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

Option dependency

```
<< 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:

$\frac{\partial \mathrm{e1}}{\partial \mathrm{p1}} = 5.$

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.

```
f1 ⊢ SMSFreeze[e1, "Dependency" -> {p1, 5}];
SMSD[Log[f1], {p1, p2}]
```

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

Here all partial derivatives of expression e1 are set to 0 except:

$\frac{\partial \mathrm{e1}}{\partial \mathrm{p1}} = 5, \frac{\partial \mathrm{e1}}{\partial \mathrm{p2}} = 10.$

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.

```
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:

$\frac{\partial \mathrm{e1}}{\partial \mathrm{p1}} = 5, \frac{\partial \mathrm{e1}}{\partial \mathrm{p2}} = 10,$

$\frac{\partial \mathrm{e2}}{\partial \mathrm{p1}} = 15, \frac{\partial \mathrm{e2}}{\partial \mathrm{p2}} = 20.$

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.

```
{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 $J_{i,j} = \frac{\partial e_i}{\partial p_j}$.

```
{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.

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

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

```
vf ⊢ SMSFreeze[{Sin[x], 0}, "Ignore" -> NumberQ];
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.

```
SMSFreeze[vf, {Sin[x], 0}, "Ignore" -> NumberQ];
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 has Auxiliary Variables 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, Non-local Operations.

Example

Here the pure fictive auxiliary variable is used for x in order to evaluate expression $f(n) = \sum_{n=1}^{m} \frac{\partial g(x)}{\partial x} |_{x=0}$, 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.

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

SMSWrite[];
```

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

```
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];
};
```

SMSReplaceAll

See Symbolic Evaluation.

SMSSmartReduce

SMSSmartReduce[*exp*,*v1*|*v2*|...]	replace those parts of the expression *exp* that do not depend on any of the auxiliary variables *v1*|*v2*|.... by a new auxiliary variable
SMSSmartReduce[*exp*,*v1*|*v2*|...,*func*]	apply pure function *func* to the sub− expressions before they are replaced by a new auxiliary variable

The default value for *func* is identity operator #&. Recommended value is Collect[#,v1|v2|...]&. The function *func* should perform only correctness preserving transformations, so that the value of expression *exp* remains the same.

See also: Non-local Operations.

SMSSmartRestore

SMSSmartRestore[*exp*,*v1*|*v2*|...]	replace those parts of expression *exp* that depend on any of the auxiliary variables *v1*| *v2*|.... by their definitions and simplify the result
SMSSmartRestore[*exp*,*v1*|*v2*|...,*func*]	apply pure function *func* to the sub− expressions that do not depend on v1| v2|.. before they are replaced by a new auxiliary variable
SMSSmartRestore[*exp*, *v1*|*v2*|...,{*evaluation_rules*},*func*]	restore expression *exp* and apply list of rules {*evaluation_rules*} to all sub− expressions that depend on any of auxiliary variables *v1*,*v2*,...

At the output, all variables *v1*|*v2*|... become fully visible. The result can be used to perform non-local operations. The default values for *func* is identity operator #&. Recommended value is Collect[#,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 Symbolic Evaluation).

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.

See also: Non-local Operations.

SMSRestore

SMSRestore[*exp*,*v1*|*v2*|...]	replace those parts of expression *exp* that depend on any of the auxiliary variables *v1*|*v2*|.... by their definitions
SMSRestore[*exp*, *v1*|*v2*|...,{*evaluation_rules*}]	restore expression *exp* and apply list of rules {*evaluation_rules*} to all sub−expressions that depend on any of auxiliary variables *v1*,*v2*,..
SMSRestore[*exp*]	replace all visible auxiliary variables in *exp* by their definition
SMSRestore[*exp*,"Global"]	repeatedly replace all auxiliary variables until only basic input variables remain (objects such as SMSExternalF, SMSFreezeF and SMSFictiveF are left intact)

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

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

See also: Non-local Operations.

Arrays

SMSArray

SMSArray[{*exp1*,*exp2*,...}]	create an *SMSGroupF* data object that represents a constant array of expressions {*exp1*, *exp2*,...}
SMSArray[*len*]	create an *SMSArrayF* data object that represents a general real type array of length *len* and allocate space on the global vector of formulas
SMSArray[*len*,*func*]	create a multi−valued auxiliary variable that represents a general array data object of length *len*, with elements *func*[*i*] , i=1,…,*len*
SMSArray[{*n*,*len*},*func*]	create *n* multi−valued auxiliary variables that represents *n* general array data objects of length *len*, with elements {*func*[*i*] ⟦1⟧, *func*[*i*] ⟦2⟧,...,*func*[*i*] ⟦*n*⟧}, i=1,…,*len*

The SMSArray[{*exp1*,*exp2*,...}] function returns the *SMSGroupF* data object. 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}]*).

The SMSArray[*len*] 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.

The SMSArray[*len*,*func*] function returns a multi-valued auxiliary variable that points at the *SMSArrayF* data object. 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.

The SMSArray[{*n*,*len*},*func*] function returns *n* multi-valued auxiliary variables that points at the *nth SMSArrayF* data objects. 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.

A constant array is represented in the final source code as a sequence of auxiliary variables and formulae. Definition of the general array only allocates space on the global vector. Constant array is represented by the data object with the

head *SMSGroupF* (*AceGen* array object). Thegeneral array data object has head *SMSArrayF*. *An a*rray data object 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 head *SMSIndexF* (*AceGen* index object).

See also:Arrays,SMSPart, Characteristic Formulae, SMSReplacePart.

SMSPart

SMSPart[{*exp1*, *exp2*,...},*index*]	create an index data object that represents the *index* −th element of the array of expressions {*exp1*, *exp2*,...}
SMSPart[*arrayo*,*index*]	create an index data object that represents the *index* −th element of the array of expressions represented by the array data object *arrayo*

The argument *arrayo* 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 perturbated 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]).

See also: Arrays.

```
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$$]];
];
```

SMSReplacePart

SMSReplacePart[*array*,*new*,*i*]	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)

See also: Arrays, SMSArray , SMSPart

SMSDot

SMSDot[$arrayo_1$,$arrayo_2$]	dot product of the two arrays of expressions represented by the array data objects $arrayo_1$ and $arrayo_2$

The arguments are the array data objects (see *Arrays*). The signature of the dot product is a dot product of the signatures of the array components.

See also: Arrays, SMSArray , SMSPart

```
SMSInitialize["test", "Language" -> "C"];
SMSModule["test", Real[x$$, r$$]];
x ⊨ SMSReal[x$$];
g1 ⊨ SMSArray[{x, x^2, 0, π}];
g2 ⊨ SMSArray[{3 x, 1 + x^2, Sin[x], Cos[x π]}];
dot ⊨ SMSDot[g1, g2];
SMSExport[dot, r$$];
SMSWrite["test"];
```

```
Method : test 4 formulae, 57 sub-expressions

[0] File created : test.c Size : 911
```

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

```
/************************************************************
* AceGen    2.103 Windows (18 Jul 08)                       *
*           Co. J. Korelc  2007            18 Jul 08 15:41:17*
************************************************************
User : USER
Evaluation time                 : 0 s     Mode  : Optimal
Number of formulae              : 4       Method: Automatic
Subroutine                      : test size :57
Total size of Mathematica  code : 57 subexpressions
Total size of C code            : 340 bytes*/
#include "sms.h"

/****************** S U B R O U T I N E ********************/
void test(double v[5009],double (*x),double (*r))
{
v[3]=Power((*x),2);
v[5004]=3e0*(*x);
v[5005]=1e0+v[3];
v[5006]=sin((*x));
v[5007]=cos(0.3141592653589793e1*(*x));
v[5000]=(*x);
v[5001]=v[3];
v[5002]=0e0;
v[5003]=0.3141592653589793e1;
(*r)=SMSDot(&v[5000],&v[5004],4);
};
```

SMSSum

SMSSum[*arrayo*]	sum of all elements of the array represented by an array data object *arrayo*

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.

See also: Arrays, SMSArray , SMSPart

Differentiation

SMSD

See: Automatic Differentiation

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, $\{z_1,z_2,\ldots, z_N\}$, D]	define gradient of auxiliary variable v with respect to variables $\{z_1, z_2, \ldots,z_N\}$ to be vector $D:=\left\{\frac{\partial v}{\partial z_i}\right\}$ ... $i=1,2,\ldots,N$ and set $\frac{\partial z_i}{\partial z_j}=\delta^i_j$
SMSDefineDerivative[$\{v_1, v_2,\ldots,v_M\}$, z, $\{d_1,d_1,\ldots,d_M\}$]	define the derivatives of the auxiliary variables $\{v_1,v_2,\ldots,v_M\}$ with respect to variable z to be $\frac{\partial v_i}{\partial z}=d_i$
SMSDefineDerivative[$\{v_1, v_2,\ldots,v_M\}$, $\{z_1,z_2,\ldots, z_N\}$, J]	define a Jacobian matrix of the transformation from $\{v_1, v_2, \ldots,v_M\}$ to $\{z_1, z_2, \ldots,z_N\}$ to be matrix $J:=\left[\frac{\partial v_i}{\partial z_j}\right]$ ... $i=1, 2,\ldots,M$; $j=1,2,\ldots,N$, and set $\frac{\partial z_i}{\partial z_j}=\delta^i_j$

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.

TO BE USED ONLY BY THE ADVANCED USERS!!

See also: Automatic Differentiation, Exceptions in Differentiation , SMSFreeze .

In the case of coordinate transformations we usually first define variables z_i in terms of variables v_j as $z_i = f_i(v_j)$. Partial derivatives $\frac{\partial v_i}{\partial z_j}$ are then defined by $\left[\frac{\partial v_i}{\partial z_j}\right]=\left[\frac{\partial f_k}{\partial v_l}\right]^{-1}$. The definition of partial derivatives $\frac{\partial v_i}{\partial z_j}$ will make independent variables z_i dependent, leading to $\frac{\partial z_i}{\partial z_j} = \sum_k \frac{\partial f_i}{\partial v_k}\frac{\partial v_k}{\partial z_j} \neq \delta^i_j$. Correct result $\frac{\partial z_i}{\partial z_j} = \delta^i_j$ is obtained by defining additional partial derivatives with

```
SMSDefineDerivative[{z1, …, zN}, {z1, …, zN}, IdentityMatrix[N]].
```

This is by default done automatically. This automatic correction can also be suppressed as follows

```
SMSDefineDerivative[{v1, …, vM}, {z1, …, zN}, J, False]
```

Program Flow Control

SMSIf

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

See also: Mathematica syntax - AceGen syntax , Program Flow Control , Auxiliary Variables, Signatures of the Expressions

SMSIf[*condition*]	starts the TRUE branch of the *if* .. *else* .. *endif* construct
SMSElse[]	starts the FALSE branch of the *if* .. *else* .. *endif* construct
SMSEndIf[]	ends the *if* .. *else* .. *endif* construct
SMSEndIf[*out_var*]	ends the *if* .. *else* .. *endif* construct and create 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 perturbated 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.

Example 1: Generic example (in-cell)

Generation of the Fortran subroutine which evaluates the following function

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

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

```
<< 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: 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".

```
<< 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
```

```
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 3: Generic example (cross-cell)

Generation of the Fortran subroutine which evaluates the following function

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

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

```
<< AceGen`;
SMSInitialize["test", "Language" -> "Fortran"];
SMSModule["test", Real[x$$, f$$]];
x ⊨ SMSReal[x$$];
```

Description of the cross-cell "If" construct.

```
SMSIf[x <= 0]
```

x ≤ 0

```
 f ← x^2;

SMSElse[]

x ≤ 0

 f ← Sin[x];

SMSEndIf[f]

f

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.

```
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 4: Incorrect use of the "If" structure

Generation of the Fortran subroutine which evaluates the following function

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

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

```
<< AceGen`;
SMSInitialize["test", "Language" -> "Fortran"];
SMSModule["test", Real[x$$, f$$]];
x ⊨ SMSReal[x$$];
SMSIf[x <= 0];
 f ⊣ x^2;
SMSElse[];
 f ⊣ Sin[x];
SMSEndIf[];
SMSExport[f, f$$];
```

```
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: {f} Current scope: {}
Misplaced variables :
 f ≡ $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
```

By combining "if" construct and multivalued auxiliary variables the arbitrary program flow can be generated. When automatic differentiation interacts with the arbitrary program structure a lot of redundant code can be generated. If the construct appears inside the loop, then some indirect dependencies can appear and all branches have to be considered for differentiation. The user is strongly encouraged to keep "if" constructs as simple as possible and to avoid redundant dependencies.

Example 5: Unnecessary dependencies

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.

```
<< 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
```

```
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];
};
```

```
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
```

```
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];
};
```

SMSElse

See:SMSIf.

SMSEndIf

See:SMSIf.

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*]	If the last $form_i$ is the pattern _, then the corresponding $value_i$ is always returned if this case is reached.

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 $value_i$ evaluates 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.

See also: Mathematica syntax - AceGen syntax , Program Flow Control

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*]	If the last $test_i$ is True, then the corresponding $value_i$ is always returned if this case is reached.

Syntax of the SMSWhich construct.

The *SMSWhich* construct is a direct equivalent of the standard Which statement. The $test_1$ are logical expressions. The *SMSWhich* command returns multi-valued auxiliary variable with random signature that represents all options. If all $value_i$ evaluates 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.

See also: Mathematica syntax - AceGen syntax , Program Flow Control

SMSDo

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
SMSDo[*expr*,{i_{max}}]	≡ SMSDo[*expr*,{i, 1, i_{max}, *1*}]
SMSDo[*expr*,{i, i_{min}, i_{max}}]	≡ SMSDo[*expr*,{i, i_{min}, i_{max}, *1*}]
SMSDo[*expr*, {i, i_{min}, i_{max}, *di*, *in_out _var*}]	create code that in evaluates *expr* with the variable i successively taking on the v i_{min} through i_{max} in steps of di and define input/output *in_out _var* variables of th
SMSDo[*expr*, {i, i_{min}, i_{max}, *di*, *in_var*,*out_var*}]	create code that in 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 v successively taken on the values i_{min} through i_{min} (in steps of 1)
SMSDo[i, i_{min}, i_{max}, di]	start the "Do" loop with an auxiliary variable v 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 v successively taken on the values through i_{max} in steps of di and create fictive instances of the *in_var* auxiliary vari
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.

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 *init_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*).

See also: Mathematica syntax - AceGen syntax , Program Flow Control , Auxiliary Variables, Signatures of the Expressions

Example 1: Generic example (in-cell)

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

```
<< 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"]
```

```
Method : test 4 formulae, 23 sub-expressions

[0] File created : test.f Size : 867

!*************************************************************
!* AceGen    2.103 Windows (17 Jul 08)                       *
!*           Co. J. Korelc  2007            18 Jul 08 00:47:03*
!*************************************************************
! User : USER
! 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.

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

Description of the loop.

```
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".

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

File:	test.f	Size: 867
Methods	No.Formulae	No.Leafs
test	4	23

This displays the contents of the generated file.

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

```
!*************************************************************
!* AceGen    2.502 Windows (24 Nov 10)                       *
!*           Co. J. Korelc  2007            25 Nov 10 12:56:07*
!*************************************************************
! User : USER
! 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.

```
<< 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 appearst to have a constant value. x= $_1$S value= 0 See also: SMSS

File:	test.f	Size: 856
Methods	No.Formulae	No.Leafs
test	4	12

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

```
!*************************************************************
!* AceGen    2.502 Windows (24 Nov 10)                       *
!*           Co. J. Korelc  2007            25 Nov 10 12:55:47*
!*************************************************************
! User : USER
! 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.

```
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: 863
Methods	No.Formulae	No.Leafs
test	4	15

```
!*************************************************************
!* AceGen    2.502 Windows (24 Nov 10)                       *
!*           Co. J. Korelc  2007            25 Nov 10 12:55:47*
!*************************************************************
! User : USER
! 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.

```
<< 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: 863
Methods	No.Formulae	No.Leafs
test	4	15

```
!*************************************************************
!* AceGen    2.502 Windows (24 Nov 10)                       *
!*           Co. J. Korelc  2007            25 Nov 10 12:55:47*
!*************************************************************
! User : USER
! 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
```

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

```
<< 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: $_1X^2$ Current scope: {}

Misplaced variables :

$_1X$ ≡ $V[4, 1] Scope: Do[i, 1, n, 1]

Events: 0

Version: 3.001 Windows (7 Mar 11) (MMA 7.)

See also: Auxiliary Variables AceGen Troubleshooting

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

$Aborted

Correct formulation for "in-cell" form

```
<< 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

```
<< 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;
```

SMSEndDo

See: SMSDo.

SMSReturn

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)

SMSBreak

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)

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)

Manipulating notebooks

SMSEvaluateCellsWithTag

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 session

option name	*description*	*default value*
"CollectInputStart"	start the process of collecting the unevaluated contents of all the notebook cells evaluated by the SMSEvaluateCellsWithTag command during the session (by default the SMSInitialize restarts the process)	False
"RemoveTag"	remove the tag *tag* from the cells included into recreated notebook	False
"CollectInput"	on False temporarily suspends the process of collecting cells for the current SMSEvaluateCellsWithTag call	True

Options for SMSEvaluateCellsWithTag command.

Cell tags are used to find single notebook cells or classes of cells in notebook. Add/Remove Cell Tags 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 cells. The cell tags are not visible unless Show Cell Tags in the Find menu is checked. To search for cells according to their cell tags, you can use either the Cell Tags submenu or the Find in Cell Tags command. SMSEvaluateCellsWithTag command finds and evaluates all cells with the specified tag.

See also: Advanced AceShare library , Solid, Finite Strain Element for Direct and Sensitivity Analysis

Example:

```
CELLTAG
    Print["this is cell with tag CELLTAG"]

        this is cell with tag CELLTAG
```

```
<<AceGen`;
SMSInitialize["test", "Language" -> "C"];
SMSModule["sub1"];
SMSEvaluateCellsWithTag["CELLTAG"];
```

```
[0-0] Include Tag : CELLTAG (2 cells found, 1 evaluated)

this is cell with tag CELLTAG
```

SMSRecreateNotebook

SMSRecreateNotebook[]	creates a new notebook that includes **unevaluated** contents of all the notebook cells that were evaluated by the SMSEvaluateCellsWithTag command during the session

option name	*description*	*default value*
"File"	notebook file name	current session name
"Head"	list of additional Cells included at the head of the notebook	{}
"Close"	close notebook after creation	False

Options for SMSRecreateNotebook command.

See also: Advanced AceShare library

SMSTagIf

SMSTagIf[*condition*, *t*, *f*]	*t* is evaluated and included **unevaluated** into recreated notebook if *condition* yields True and *f* if *condition* yields False (True or False value has to be assigned to *condition* before the corresponding SMSEvaluateCellsWithTag call !!!)

See also: Advanced AceShare library

SMSTagSwitch

SMSTagSwitch[*expr*, $form_1$,$value_1$,$form_2$,$value_2$,…]	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 nad including the **unevaluated** $value_i$ into recreated notebook

See also: Advanced AceShare library

SMSTagReplace

SMSTagReplace[*eval*, *include*]	evaluates *eval* but includes **unevaluated** *include* into recreated notebook
SMSTagEvaluate[*exp*]	evaluates *exp* and includes **evaluated** *exp* into recreated notebook

See also: Advanced AceShare library

Debugging

SMSSetBreak

See Run Time Debugging

SMSLoadSession

See Run Time Debugging

SMSClearBreak

See Run Time Debugging

SMSActivateBreak

See Run Time Debugging

Random Signature Functions

SMSAbs

SMSAbs[*exp*]	absolute value of *exp*

The result of the evaluation of the *SMSAbs* function is an unique random value. The *SMSAbs* should be used instead of the *Mathematica´s Abs* function in order to reduce the possibility of incorrect simplification and to insure proper automatic differentiation.

See also: Expression Optimization

SMSSign

SMSSign[*exp*]	`-1`, `0` or `1` depending on whether *exp* is negative, zero, or positive

The result of the evaluation of the *SMSSign* function is an unique random value. The *SMSSign* should be used instead of the *Mathematica´s Sign* function in order to reduce the possibility of incorrect simplification and to insure proper

automatic differentiation.

See also: Expression Optimization

SMSKroneckerDelta

SMSKroneckerDelta[*i*, *j*]	1 or 0 depending on whether *i* is equal to *j* or not

The result of the evaluation of the *SMSKroneckerDelta* function is an unique random value. The *SMSKroneckerDelta* should be used in order to reduce the possibility of incorrect simplification and to insure proper automatic differentiation.

See also: Expression Optimization

SMSSqrt

SMSSqrt[*exp*]	square root of *exp*

The result of the evaluation of the *SMSSqrt* function is a unique random value. The *SMSSqrt* should be used instead of the *Mathematica´s Sqrt* function in order to reduce the possibility of incorrect simplification and to insure proper automatic differentiation.

See also: Expression Optimization

SMSMin

SMSMin[*exp1*,*exp2*]	≡ Min[*exp1*,*exp2*]

SMSMax

SMSMax[*exp1*,*exp2*]	≡ Max[*exp1*,*exp2*]

SMSRandom

SMSRandom[]	random number on interval [0,1] with the precision *SMSEvaluatePrecision*
SMSRandom[*i*,*j*]	random number on interval [*i*,*j*] with the precision *SMSEvaluatePrecision*
SMSRandom[*i*]	gives random number from the interval [0.9∗i ,1.1∗i]
SMSRandom[i_List]	≡ Map[SMSRandom[♯]&, i]

See also: Expression Optimization

General Functions

SMSNumberQ

SMSNumberQ[*exp*]	gives True if *exp* is a real number and False if the results of the evaluation is N/A

SMSPower

SMSPower[x,y]	$\equiv x^y$
SMSPower[x,y,"Positive"]	$\equiv x^y$ under assumption that $x>0$
SMSPower[x,y,"NonNegative"]	$\equiv x^y$ under assumption that $x\geq 0$

SMSTime

SMSTime[]	returns number of seconds elapsed since midnight (00:00:00), January 1,1970, coordinated universal time (UTC)

SMSUnFreeze

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

The *SMSUnFreeze* function searches the entire database. The Normal operator can be used to remove all special object (*SMSFreezeF*, *SMSExternalF*, ...) from the explicit form of the expression.

Linear Algebra

SMSLinearSolve

See Linear Algebra

SMSLUFactor

See Linear Algebra

SMSLUSolve

See Linear Algebra

SMSFactorSim

See Linear Algebra

SMSInverse

See Linear Algebra

SMSDet

See Linear Algebra

SMSKrammer

See Linear Algebra

Tensor Algebra

SMSCovariantBase

See Tensor Algebra

SMSCovariantMetric

See Tensor Algebra

SMSContravariantMetric

See Tensor Algebra

SMSChristoffell1

See Tensor Algebra

SMSChristoffell2

See Tensor Algebra

SMSTensorTransformation

See Tensor Algebra

SMSDCovariant

See Tensor Algebra

Mechanics of Solids

SMSLameToHooke

SMSLameToHooke[λ,μ]	transform Lame' s constants λ, μ to Hooke' s constants E, ν
SMSHookeToLame[E, ν]	transform Hooke' s constants E, ν to Lame' s constants λ, μ
SMSHookeToBulk[E,ν]	transform Hooke' s constants E, ν to shear modulus G and bulk modulus κ
SMSBulkToHooke[G,κ]	transform shear modulus G and bulk modulus κ to Hooke' s constants E, ν

Transformations of mechanical constants in mechanics of solids.

This transforms Lame's constants λ, μ to Hooke's constants E, ν. No simplification is preformed!

```
SMSLameToHooke[λ, μ] // Simplify
```

$$\left\{\frac{\mu\,(3\,\lambda+2\,\mu)}{\lambda+\mu},\ \frac{\lambda}{2\,(\lambda+\mu)}\right\}$$

SMSHookeToLame

SMSLameToHooke[λ,μ]	transform Lame' s constants λ, μ to Hooke' s constants E, ν
SMSHookeToLame[E, ν]	transform Hooke' s constants E, ν to Lame' s constants λ, μ
SMSHookeToBulk[E,ν]	transform Hooke' s constants E, ν to shear modulus G and bulk modulus κ
SMSBulkToHooke[G,κ]	transform shear modulus G and bulk modulus κ to Hooke' s constants E, ν

Transformations of mechanical constants in mechanics of solids.

This transforms Lame's constants λ, μ to Hooke's constants E, ν. No simplification is preformed!

```
SMSLameToHooke[λ, μ] // Simplify
```

$$\left\{\frac{\mu\,(3\,\lambda+2\,\mu)}{\lambda+\mu},\ \frac{\lambda}{2\,(\lambda+\mu)}\right\}$$

SMSHookeToBulk

SMSLameToHooke[λ,μ]	transform Lame' s constants λ, μ to Hooke' s constants E, ν
SMSHookeToLame[E, ν]	transform Hooke' s constants E, ν to Lame' s constants λ, μ
SMSHookeToBulk[E,ν]	transform Hooke' s constants E, ν to shear modulus G and bulk modulus κ
SMSBulkToHooke[G,κ]	transform shear modulus G and bulk modulus κ to Hooke' s constants E, ν

Transformations of mechanical constants in mechanics of solids.

This transforms Lame's constants λ, μ to Hooke's constants E, ν. No simplification is preformed!

```
SMSLameToHooke[λ, μ] // Simplify
```

$$\left\{\frac{\mu\,(3\,\lambda+2\,\mu)}{\lambda+\mu},\ \frac{\lambda}{2\,(\lambda+\mu)}\right\}$$

SMSBulkToHooke

SMSLameToHooke[λ,μ]	transform Lame' s constants λ, μ to Hooke' s constants E, ν
SMSHookeToLame[E, ν]	transform Hooke' s constants E, ν to Lame' s constants λ, μ
SMSHookeToBulk[E,ν]	transform Hooke' s constants E, ν to shear modulus G and bulk modulus κ
SMSBulkToHooke[G,κ]	transform shear modulus G and bulk modulus κ to Hooke' s constants E, ν

Transformations of mechanical constants in mechanics of solids.

This transforms Lame's constants λ, μ to Hooke's constants E, ν. No simplification is preformed!

```
SMSLameToHooke[λ, μ] // Simplify
```

$$\left\{\frac{\mu\,(3\,\lambda+2\,\mu)}{\lambda+\mu},\ \frac{\lambda}{2\,(\lambda+\mu)}\right\}$$

SMSPlaneStressMatrix

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!

```
SMSPlaneStressMatrix[e, ν] // MatrixForm
```

$$\begin{pmatrix} \frac{e}{1-\nu^2} & \frac{e\,\nu}{1-\nu^2} & 0 \\ \frac{e\,\nu}{1-\nu^2} & \frac{e}{1-\nu^2} & 0 \\ 0 & 0 & \frac{e}{2\,(1+\nu)} \end{pmatrix}$$

SMSPlaneStrainMatrix

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!

```
SMSPlaneStressMatrix[e, ν] // MatrixForm
```

$$\begin{pmatrix} \frac{e}{1-\nu^2} & \frac{e\,\nu}{1-\nu^2} & 0 \\ \frac{e\,\nu}{1-\nu^2} & \frac{e}{1-\nu^2} & 0 \\ 0 & 0 & \frac{e}{2\,(1+\nu)} \end{pmatrix}$$

SMSEigenvalues

SMSEigenvalues[*matrix*]	create code sequence that calculates the eigenvalues of the third order 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 might occur.

SMSMatrixExp

SMSMatrixExp[**M**]	create code sequence that calculates exponential of the 3×3 matrix

option name	*default value*	
"Order"	Infinity	Infinity ⇒ analytical solution (all eigenvalues of the matrix have to be real numbers.) r_Integer ⇒ Taylor series expansion of order r (arbitrary matrix) ϵ_Real ⇒ Taylor series expansion truncated when $\|\frac{\mathbf{M}^k}{k!}\| < \epsilon$ (arbitrary matrix)
"Module"	False	False ⇒ generated code is included directly into current module True ⇒ generated code is included as separate module
"Derivatives"	1	Active only when "Module"−>True. 0 ⇒ derivatives are not supported 1 ⇒ generated module includes the definition of the first order derivatives $\left(\frac{\partial \mathrm{Exp}\,(\mathbf{M})}{\partial \mathbf{M}}\right)$

Options for SMSMatrixExp.

SMSInvariantsI

SMSInvariantsI[*matrix*]	I_1, I_2, I_3 invariants of the third order matrix
SMSInvariantsJ[*matrix*]	J_1, J_2, J_3 invariants of the third order matrix

SMSInvariantsJ

SMSInvariantsI[*matrix*]	I_1, I_2, I_3 invariants of the third order matrix
SMSInvariantsJ[*matrix*]	J_1, J_2, J_3 invariants of the third order matrix

MathLink Environment

SMSInstallMathLink

See MathLink, Matlab Environments

SMSLinkNoEvaluations

See MathLink, Matlab Environments

SMSSetLinkOptions

See MathLink, Matlab Environments

Finite Element Environments

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 commands.**

See also Template Constants section for a list of all constants and the Interactions Templates-AceGen-AceFEM 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];
```

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	*description*	*default subroutine name*
"Tangent and residual"	standard subroutine that returns the tangent matrix and residual for the current values of nodal and element data	"SKR"
"Postprocessing"	standard subroutine that returns postprocessing quantities (see Standard user subroutines)	"SPP"
"Sensitivity pseudo−load"	standard subroutine that returns the sensitivity pseudo−load vector for the current sensitivity parameter (see Standard user subroutines)	"SSE"
"Dependent sensitivity"	standard subroutine that resolves sensitivities of the dependent variables defined at the element level (see Standard user subroutines)	"SHI"
"Residual"	standard subroutine that returns residual for the current values of the nodal and element data	"SRE"
"Nodal information"	standard subroutine that returns position of the nodes at current and previous time step and normal vectors if applicable (used for contact elements)	"PAN"
"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. (see Standard user subroutines, User Defined Tasks)	"Tasks"
"User n"	*n* −th user defined system subroutine (low−level system feature intended to be used by advanced users)	"Usern"

Standard set of user subroutines.

option	
"Name"−>"*name*"	use a given name for the generated subroutine insted of defult 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 *SMSReal[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$$	integer type environment variables (see Integer Type Environment Data)
rdata$$	real type environment variables (see Real Type Environment Data)

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
"Sensitivity pseudo−load"	p$$[*NoDOFGlobal*]	sensitivity pseudo−load vector
"Dependent sensitivity"	−	−
"Tangent"	s$$[*NoDOFGlobal*,*NoDOFGlobal*]	element tangent matrix
"Residuum"	p$$[*NoDOFGlobal*]	element residual vector
"Nodal information"	d$$[*problem dependent* , 6]	$\{\{x_1^t, y_1^t, z_1^t, x_1^p, y_1^p, z_1^p\}, \{x_2^t, y_2^t \ldots\}, \ldots\}$
"Tasks"	Task$$ TasksData$$[5] IntegerInput$$[TasksData$$[2]] RealInput$$[TasksData$$[3]] IntegerOutput$$[TasksData$$[4]] RealOutput$$[TasksData$$[5]]	see User Defined Tasks, Standard user subroutines, SMTTask
"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 supprot 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"	●	●		
"Sensitivity pseudo−load"	●	●	●	
"Dependent sensitivity"	●	●	●	
"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.

```
<< AceGen`;
SMSInitialize["test", "Environment" -> "AceFEM"];
SMSTemplate["SMSTopology" → "Q1", "SMSDOFGlobal" → 5,
  "SMSGroupDataNames" → {"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
```

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

```
<< AceGen`;
SMSInitialize["test", "Environment" -> "FEAP"];
SMSTemplate["SMSTopology" → "Q1", "SMSDOFGlobal" → 5,
  "SMSGroupDataNames" → {"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
```

SMSFEAPMake

SMSFEAPMake[*source*]	compiles *source*.*f* source file and builds the FEAP executable program

Create FEAP executable.

The paths to FEAP's Visual Studio project have to be set as described in the Install.txt file available at www.fgg.uni-lj.si/symech/user/install.txt.

SMSFEAPRun

SMSFEAPRun[*input*]	runs FEAP with the *input* as input data file

Run analysis.

option name	*default value*	
"Debug"	False	pause before exiting the *FEAP* executable
"Splice"	False	splice file with the given file name into an FEAP input file *input* (it takes text enclosed between <*and*>in the file, evaluates the text as *Mathematica* input, and replaces the text with the resulting *Mathematica* output)
"Output"	Automatic	name of the FEAP output data file

Options for SMSFEAPRun.

The paths to FEAP's Visual Studio project have to be set as described in the Install.txt file available at www.fgg.uni-lj.si/symech/user/install.txt.

SMSELFENMake

SMSELFENMake[*source*]	compiles *source.f* source file and builds the ELFEN executable program

Create ELFEN executable.

The paths to ELFEN's Visual Studio project have to be set as described in the Install.txt file available at www.fgg.uni-lj.si/symech/user/install.txt.

SMSELFENRun

SMSELFENRun[*input*]	runs ELFEN with the *input* as input data file

Run analysis.

option name	*default value*	
"Debug"	False	pause before exiting the *ELFEN* executable
"Splice"	False	splice file with the given file name into an ELFEN input file *input* (it takes text enclosed between <*and*>in the file, evaluates the text as *Mathematica* input, and replaces the text with the resulting *Mathematica* output)
"Output"	Automatic	name of the ELFEN output data file

Options for SMSELFENRun.

The paths to ELFEN's Visual Studio project have to be set as described in the Install.txt file available at www.fgg.uni-lj.si/symech/user/install.txt.

SMSABAQUSMake

SMSABAQUSMake[*ecode*]	compiles element source file defined by the element code *ecode* and builds the user element object file (*ecode* can be a name of the element FORTRAN source file or an `unified element code` that points to the elements in shared libraries)

Create user element object file.

SMSABAQUSRun

SMSABAQUSRun[*input*]	runs ABAQUS with the *input* as ABAQUS input data file

Run analysis.

option name	*default value*	
"Debug"	False	pause before exiting the *ABAQUS* executable
"Splice"	False	splice file with the given file name into an ABAQUS input file *input* (it takes text enclosed between <*and*>in the file, evaluates the text as *Mathematica* input, and replaces the text with the resulting *Mathematica* output)
"UserElement"	False	run ABAQUS with the specified by element code *ecode*

Options for SMSABAQUSRun.

Additional definitions

idata$$

See: `Integer Type Environment Data`

rdata$$

See: `Real Type Environment Data`

ns$$

See: `Node Specification Data`

nd$$

See: `Node Data`

es$$

See: `Domain Specification Data`

ed$$

See: `Element Data`

SMSTopology

See: Template Constants – SMSTopology

SMSNoDimensions

See: Template Constants – SMSNoDimensions

SMSNoNodes

See: Template Constants – SMSNoNodes

SMSDOFGlobal

See: Template Constants – SMSDOFGlobal

SMSNoDOFGlobal

See: Template Constants – SMSNoDOFGlobal

SMSNoAllDOF

See: Template Constants – SMSNoAllDOF

SMSSymmetricTangent

See: Template Constants – SMSSymmetricTangent

SMSGroupDataNames

See: Template Constants – SMSGroupDataNames

SMSDefaultData

See: Template Constants – SMSDefaultData

SMSGPostNames

See: Template Constants – SMSGPostNames

SMSNPostNames

See: Template Constants – SMSNPostNames

SMSNoDOFCondense

See: Template Constants – SMSNoDOFCondense

SMSNoTimeStorage

See: Template Constants – SMSNoTimeStorage

SMSNoElementData

See: Template Constants – SMSNoElementData

SMSResidualSign

See: Template Constants – SMSResidualSign

SMSSegments

See: Template Constants – SMSSegments

SMSSegmentsTriangulation

See: Template Constants – SMSSegmentsTriangulation

SMSNodeOrder

See: Template Constants – SMSNodeOrder

ELFEN$NoStress

See: Template Constants – ELFEN$NoStress

ELFEN$NoStrain

See: Template Constants – ELFEN$NoStrain

ELFEN$NoState

See: Template Constants – ELFEN$NoState

ELFEN$ElementModel

See: Template Constants – ELFEN$ElementModel

FEAP$ElementNumber

See: Template Constants – FEAP$ElementNumber

SMSReferenceNodes

See: Template Constants – SMSReferenceNodes

SMSNoNodeStorage

See: Template Constants – SMSNoNodeStorage

SMSNoNodeData

See: Template Constants – SMSNoNodeData

SMSDefaultIntegrationCode

See: Template Constants – SMSDefaultIntegrationCode

SMSAdditionalNodes

See: Template Constants – SMSAdditionalNodes

SMSNodeID

See: Template Constants – SMSNodeID

SMSAdditionalGraphics

See: Template Constants – SMSAdditionalGraphics

SMSSensitivityNames

See: Template Constants – SMSSensitivityNames

SMSShapeSensitivity

See: Template Constants – SMSShapeSensitivity

SMSMainTitle

See: Template Constants – SMSMainTitle

SMSSubTitle

See: Template Constants – SMSSubTitle

SMSSubSubTitle

See: Template Constants – SMSSubSubTitle

SMSMMAInitialisation

See: Template Constants – SMSMMAInitialisation

SMSMMANextStep

See: Template Constants – SMSMMANextStep

SMSMMAStepBack

See: Template Constants – SMSMMAStepBack

SMSMMAPreIteration

See: Template Constants – SMSMMAPreIteration

SMSIDataNames

See: Template Constants – SMSIDataNames

SMSRDataNames

See: Template Constants – SMSRDataNames

SMSBibliography

See: Template Constants – SMSBibliography

SMSNoAdditionalData

See: Template Constants – SMSNoAdditionalData

SMSUserDataRules

See: Template Constants – SMSUserDataRules

SMSCharSwitch

See: Template Constants – SMSCharSwitch

SMSIntSwitch

See: Template Constants – SMSIntSwitch

SMSDoubleSwitch

See: Template Constants – SMSDoubleSwitch

SMSCreateDummyNodes

See: Template Constants – SMSCreateDummyNodes

SMSPostIterationCall

See: Template Constants – SMSPostIterationCall

SMSPostNodeWeights

See: Template Constants – SMSPostNodeWeights

SMSCondensationData

See: Template Constants – SMSCondensationData

SMSDataCheck

See: Template Constants – SMSDataCheck