AceGen

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


J. Korelc

UNIVERSITY OF LJUBLJANA
FACULTY OF CIVIL AND GEODETIC ENGINEERING

2006

AceGen Contents

Tutorial

Preface

AceGen 1.0

University of Ljubljana
Faculty of Civil and Geodetic Engng.
Jamova 2, SI – 1000, Ljubljana, Slovenia
jkorelc@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 provides also 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*© is a commercial finite element environment 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.

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 will be to combine techniques in such a way that will lead to an optimal environment for the design and implementation of arbitrary numerical procedures. 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,

⇒ automatic differentiation technique,

⇒ automatic selection of the appropriate intermediate variables,

⇒ the whole program structure can be generated,

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

⇒ improved optimization procedures with stochastic evaluation of expressions,

⇒ generation of characteristic formulae,

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

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

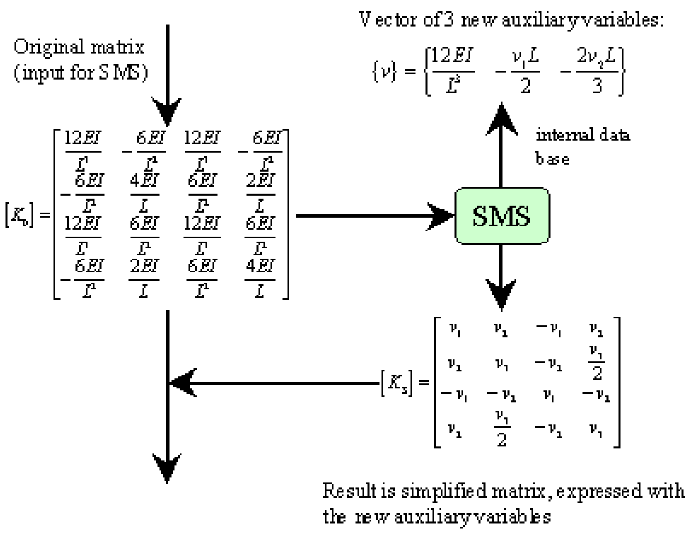
⇒ advanced user interface,

⇒ 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 interactively or in file, and returns an optimized version of the expression. Optimized version of the expression can result in a newly created auxiliary symbol (v_i), or in an original expression in parts replaced by previously created auxiliary symbols. In the first case *AceGen* stores the new expression in an internal data base. The data base contains a global vector of all expressions, information about dependencies of the symbols, labels and names of the symbols, partial derivatives, etc. The data base is a global object which maintains informations 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. By 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).

Bibliography

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

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.

WRIGGERS, Peter, KRSTULOVIC-OPARA, Lovre, KORELC, Jože.(2001), Smooth C1-interpolations for two-dimensional frictional contact problems. *Int. j. numer. methods eng.*, 2001, vol. 51, issue 12, str. 1469-1495

KRSTULOVIC-OPARA, Lovre, WRIGGERS, Peter, KORELC, Jože. (2002), A C1-continuous formulation for 3D finite deformation frictional contact. *Comput. mech.*, vol. 29, issue 1, 27-42

STUPKIEWICZ, Stanislaw, KORELC, Jože, DUTKO, Martin, RODIC, Tomaž. (2002), Shape sensitivity analysis of large deformation frictional contact problems. *Comput. methods appl. mech. eng.*, 2002, vol. 191, issue 33, 3555-3581

BRANK, Boštjan, KORELC, Jože, IBRAHIMBEGOVIC, Adnan. (2002), Nonlinear shell problem formulation accounting for through-the-tickness stretching and its finite element implementation. *Comput. struct.*. vol. 80, n. 9/10, 699-717

BRANK, Boštjan, KORELC, Jože, IBRAHIMBEGOVIC, Adnan. (2003),Dynamic and time-stepping schemes for elastic shells undergoing finite rotations. *Comput. struct.*, vol. 81, issue 12, 1193-1210

STADLER, Michael, HOLZAPFEL, Gerhard A., KORELC, Jože. (2003) Cn continuous modelling of smooth contact surfaces using NURBS and application to 2D problems. *Int. j. numer. methods eng.*, 2177-2203

KUNC, Robert, PREBIL, Ivan, RODIC, Tomaž, KORELC, Jože. (2002),Low cycle elastoplastic properties of normalised and tempered 42CrMo4 steel. *Mater. sci. technol.*, Vol. 18, 1363-1368.

Standard AceGen Procedure

Load *AceGen* package

This loads the *AceGen* package.

```
In[12]:= <<AceGen`
```

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	$\Longrightarrow$	SMSInitialize["test","Language"->"C"]
2	Definition of input and output parameters	$\Longrightarrow$	SMSModule["Test",Real[u$$[3],x$$,L$$,g$$[3]]];
3	Definition of numeric–symbolic interface variables	$\Longrightarrow$	{x,L}⊧{SMSReal[x$$],SMSReal[L$$]}; ui⊧SMSReal[Array[u$$[#],3]&];
4	Derivation of the problem	$\Longrightarrow$	Ni⊧{$\frac{x}{L}$,1−$\frac{x}{L}$,$\frac{x}{L}$ (1−$\frac{x}{L}$)}; u⊧Ni.ui; f⊧u^2; g⊧SMSD[f,ui];
5	Definition of symbolic–numeric interface variables	$\Longrightarrow$	SMSExport[g,g$$];
6	Code generation	$\Longrightarrow$	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 initializes the *AceGen* session. FORTRAN is chosen as the final code language. See also SMSInitialize .

```
In[13]:= 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 External variables .

```
In[14]:= 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 Intermediate variables , SMSReal .

```
In[15]:= x ⊧ SMSReal[x$$]

Out[15]= x

In[16]:= L ⊧ SMSReal[L$$]

Out[16]= L
```

Here the variable u[1], u[2], u[3] are introduced with the signature (characteristic random numbers used within code optimization procedures) taken from the interval [0.1,0.2]. If the interval is omitted, the signature from the default interval [0,1] is generated.

```
In[17]:= ui ⊧ SMSReal[Array[u$$[#1] & , 3]]
```

Out[17]= $\{ui_1, ui_2, ui_3\}$

Step 4: Description of the Problem

Here is the body of the subroutine.

```
In[18]:= Ni ⊨ {x / L, 1 - x / L, x / L * (1 - x / L)}
```

Out[18]= $\{Ni_1, Ni_2, Ni_3\}$

```
In[19]:= u ⊨ Ni . ui
```

```
Out[19]= u
```

```
In[20]:= f ⊨ u^2
```

```
Out[20]= f
```

```
In[21]:= g ⊨ SMSD[f, ui]
```

Out[21]= $\{g_1, g_2, g_3\}$

Step 5: Definition of Symbolic - Numeric Interface Variables

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

```
In[22]:= SMSExport[g, g$$];
```

Step 6: Code Generation

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

```
In[23]:= SMSWrite[];
```

```
Method : Test 6 formulae, 81 sub-expressions

[0] File created : test.f Size : 948
```

This displays the contents of the generated file.

In[24]:= **!!test.f**

```
!**************************************************************
!* AceGen    VERSION                                          *
!*           Co. J. Korelc  2006            20.8.2006 23:31   *
!**************************************************************
! User : Korelc
! 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.

In[1]:=
```
<< AceGen`;
SMSInitialize["test", "Language" -> "C"];
SMSModule["Test", Real[u$$[3], x$$, L$$, g$$[3]]];
{x, L} ⊨ {SMSReal[x$$], SMSReal[L$$]};
ui ⊨ SMSReal[Array[u$$[#1] & , 3]];
Ni ⊨ {x/L, 1 - x/L, x/L (1 - x/L)};
u ⊨ Ni.ui;
f ⊨ u^2;
g ⊨ SMSD[f, ui];
SMSExport[g, g$$];
SMSWrite[];
```

Method : **Test** 6 formulae, 81 sub-expressions

[0] File created : **test.c** Size : 863

In[36]:= **!!test.c**

```
/*************************************************************
* AceGen    VERSION                                          *
*           Co. J. Korelc  2006          20.8.2006 23:31     *
*************************************************************
User : Korelc
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 version of the source code is generated. The generated code is automatically enhanced by an additional modules necessary for the proper *MathLink* connection.

In[1]:=
```
<< 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[Array[u$$[#1] &, 3]];
Ni ⊨ {x/L, 1 - x/L, x/L (1 - x/L)};
u ⊨ Ni.ui;
f ⊨ u^2;
g ⊨ SMSD[f, ui];
SMSExport[g, g$$];
SMSWrite[];
```

Method : **Test** 6 formulae, 81 sub-expressions

[0] File created : **test.c** Size : 1554

In[12]:= **!!test.c**

```
/*************************************************************
* AceGen    VERSION                                          *
*           Co. J. Korelc  2006           27.9.2006 19:46   *
*************************************************************
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[5001];
void Test(double v[5001],double u[3],double (*x),double (*L),double g[3]);

void TestMathLink(){
int i1000,i1001,i1002,i1,i4;
char *b1; double *b2;int *b3;
double u[3];
double x;
double L;
double g[3];
MLGetRealList(stdlink,&b2,&i1);
for(i1001=0;i1001<min(i1,3);i1001++)u[i1001]=b2[i1001];
MLDisownRealList(stdlink,b2,i1);
MLGetReal(stdlink,&x);
MLGetReal(stdlink,&L);
Test(workingvector,u,&x,&L,g);
PutRealList(g,3);
};

int main(int argc,char  *argv[]){
  printf("MathLink module: %s\n","test");
  pauseonexit=0;
  atexit(exit_util);
  return MLMain(argc, argv);
};


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

Here the *MathLink* program Test.exe is build from the generated source code and installed so that functions defined in the source code can be called directly from *Mathematica*. (see also `SMSInstallMathLink`)

In[13]:= **SMSInstallMathLink**[]

```
Out[13]= {SMSMathLinkInitialize[Test, i_Integer, j_Integer],
          Test[u_?(VectorQ[#1, NumberQ] &), 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`).

```
In[14]:= Test[{0., 1., 7.}, π // N, 10.]

Out[14]= {1.37858, 3.00958, 0.945489}
```

Generation of *Matlab* code

Here the Matlab version of the source code is generated.

```
In[15]:= << 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[Array[u$$[#1] &, 3]];
         Ni ⊨ {x/L, 1 - x/L, x/L (1 - x/L)};
         u ⊨ Ni.ui;
         f ⊨ u^2;
         g ⊨ SMSD[f, ui];
         SMSExport[g, g$$];
         SMSWrite[];

           Method : Test 6 formulae, 81 sub-expressions

           [0] File created : test.m Size : 1042
```

```
In[26]:= !!test.m
```

```
%**************************************************************
%* AceGen    VERSION                                          *
%*           Co. J. Korelc  2006            27.9.2006 19:46   *
%**************************************************************
% 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      : 262 bytes

%*********************** F U N C T I O N **************************
function[g]=Test(u,x,L);
v=zeros(5001,'double');
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
```

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 thought 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 , SMSInteger , SMSLogical , SMSExport .

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 *rhs*, creates new auxiliary variable, and assigns the new auxiliary variable to be the value of *lhs*. From then on, lhs is replaced by a new auxiliary variable whenever it appears. rhs is then stored into the *AceGen* database.

In AceGen there are four basic 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* (lhs=rhs). In principle we can get AceGen input simply by replacing = operators in standard *Mathematica* input by one of the AceGen assignment operators.

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 the basic 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 , SMSM , SMSS , SMSReal , SMSInteger , SMSLogical .

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.

```
In[37]:= << 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 : 862
```

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

```
In[44]:= {x, i, l} // ToString

Out[44]= {$V[1, 1], $V[2, 1], $V[3, 1]}
```

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

```
x  :=  $V[1, 1]  ⇒  v(1)
i  :=  $V[2, 1]  ⇒   i2
l  :=  $V[3, 1]  ⇒   b3
```

In[45]:= **!!test.f**

```
!**************************************************************
!* AceGen    VERSION                                          *
!*           Co. J. Korelc  2006            20.8.2006 23:31   *
!**************************************************************
! User : Korelc
! Evaluation time                 : 0 s     Mode   : Debug
! Number of formulae              : 3       Method: Automatic
! Subroutine                      : Test size :13
! Total size of Mathematica  code : 13 subexpressions
! Total size of Fortran code      : 295 bytes

!******************* S U B R O U T I N E **********************
      SUBROUTINE Test(v)
      IMPLICIT NONE
      include 'sms.h'
      INTEGER i2
      LOGICAL b3
      DOUBLE PRECISION v(5001)
! 1 = x
      v(1)=0.3141592653589793d1
! 2 = i
      i2=int(1)
! 3 = l
      b3=.true.
      END
```

Example: multi-valued variables

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

In[46]:=
```
<< 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 : 814
```

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

```
In[52]:=  x // ToString

Out[52]=  $V[1, 2]
```

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

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

```
In[53]:=  !!test.f
```

```
!**************************************************************
!* AceGen    VERSION                                          *
!*           Co. J. Korelc  2006            20.8.2006 23:31   *
!**************************************************************
! User : Korelc
! 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

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

where x and y are unknowns and a is the parameter using the standard Newton-Raphson iterative procedure. The *SMSSetBreak* function inserts the breaks points with the identifications "X" and "A" into the generated code.

```
In[349]:=
<< AceGen`;
SMSInitialize["test", "Language" -> "Mathematica", "Mode" → "Debug"];
SMSModule["test", Real[x$$, y$$, a$$, tol$$], Integer[n$$]];
{x0, y0, a, ϵ} ⊧ SMSReal[{x$$, y$$, a$$, tol$$}];
nmax = SMSInteger[n$$];
{x, y} ⫤ {x0, y0};
SMSDo[i, 1, nmax, 1, {x, y}];
    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[];
    SMSEndIf[];
    SMSIf[i == nmax];
        SMSPrint["'no convergion'"];
        SMSReturn[];
    SMSEndIf[];
    SMSSetBreak["X"];
SMSEndDo[];
SMSWrite[];
```

```
time=0  variable= 0 ≡ {}
Forward differentiation of 6 variables.
Solution of 2 linear equations.
[1]  Consistency check - global
[1]  Consistency check - expressions
[1]  Generate source code :
Method : test 32 formulae, 194 sub-expressions
Events: 0
[1] Final formating
[1] File created : test.m Size : 2408
```

Exploring the structure of the formula

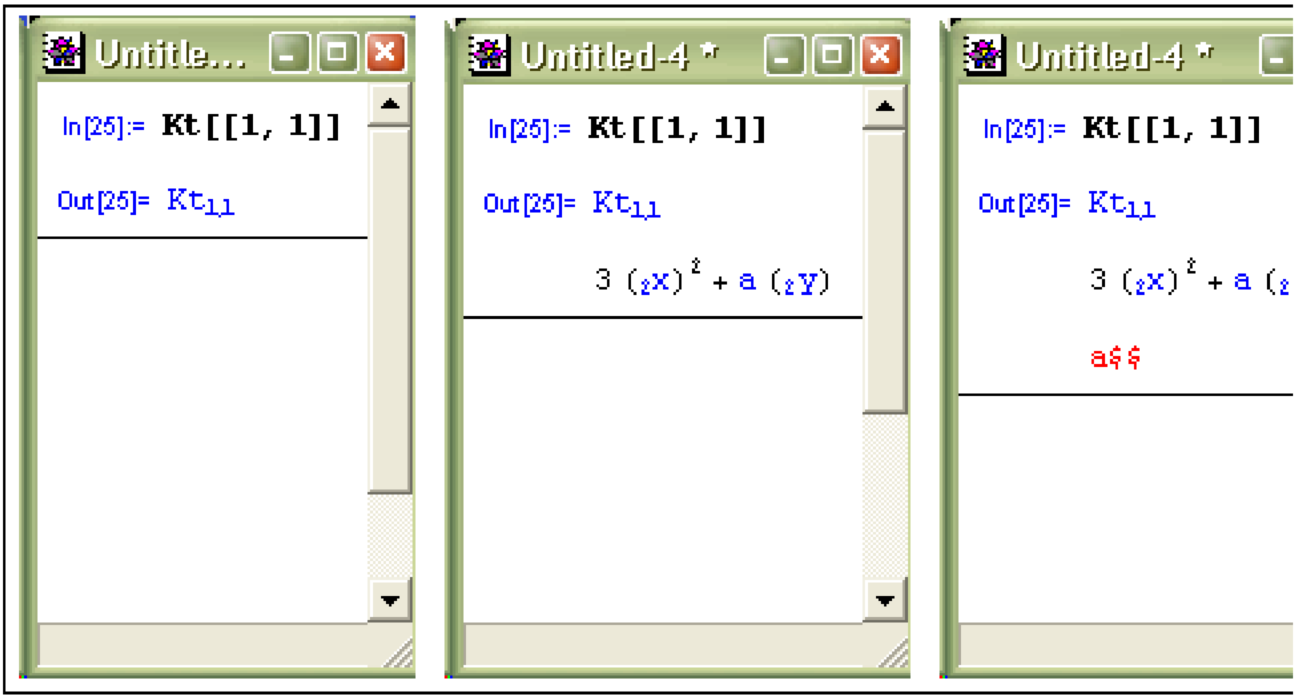
AceGen palette offers buttons that control how expressions are represented on a screen.

$\Phi_{ij}{}^2$
$({}_2¥_5)^2$
`$V[5,2]^2`
`3.14151`
`ZOOM all`
`ZOOM sele.`
`Last name`
`First name`
`All names`

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. $k_{11;x_1} \equiv \frac{\partial k_{11}}{\partial x_1}$).

```
In[233]:=
      Kt[[1, 1]]

Out[233]=
      Kt11
```



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: `ZOOM se.`

```
In[372]:=
      Kt
Out[372]=
      {{Kt11, Kt12}, {--Φ2;x, Kt22}}

      a 2X
```

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

Button: **ZOOM all**

```
In[235]:=
      Kt
Out[235]=
      {{Kt11, Kt12}, {--Φ2;x, Kt22}}

      {{Kt11, a 2X}, {--Φ2;x, Kt22}}
```

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: $\Phi_{ij}{}^2$

```
In[236]:=
      Kt
Out[236]=
      {{Kt11, Kt12}, {--Φ2;x, Kt22}}
```

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: **$V[5,2]^2**

```
In[237]:=
      Kt
Out[237]=
      {{$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: **$({}_{2}¥_{5})^{2}$**

```
In[238]:=
      Kt
Out[238]=
      {{¥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 can be used for debugging.

Button: **3.14151**

```
In[239]:=
      Kt
Out[239]=
      {{1.88136968728346844755671352685112325267369709991604672558212,
        0.35532800737456468387465454062781729688510428083534474135257},
       {-0.199444213279276150918981737408557038630358423942906038354110,
        -0.66612417045124917818278138936071633954807337182830311724512}}
```

Polymorphism of the generated formulae

Sometimes *AceGen* finds more that one meaning (name) for the same auxiliary variable. By default it displays the first name **First name** (first the system has to be put back to the basic display mode with the **$\Phi_{ij}{}^{2}$** button).

```
In[242]:=
      Kt
Out[242]=
      {{Kt11, Kt12}, {--Φ2;x, Kt22}}
```

By pressing button **Last name** the last found meaning (name) of the auxiliary variables will be displayed.

```
In[243]:=
      Kt
Out[243]=
      {{Φ1;x, Φ1;y}, {--Kt21, Φ2;y}}
```

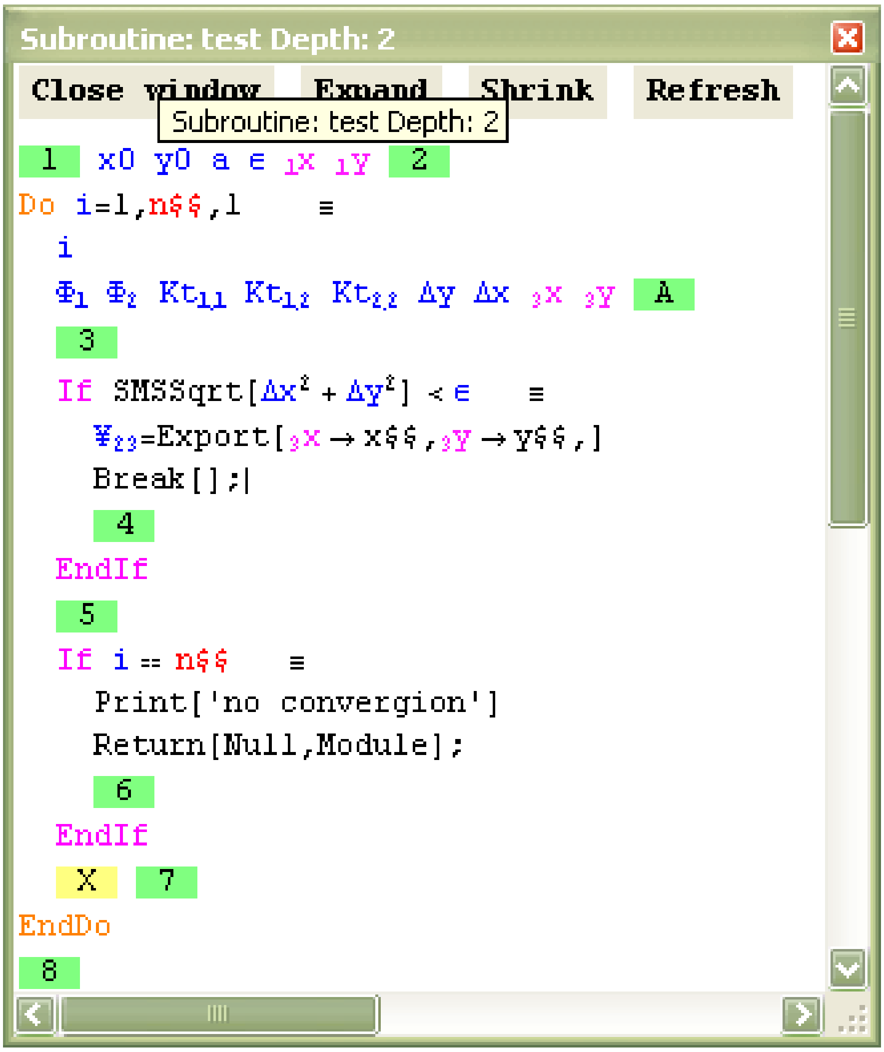
All meanings (names) of the auxiliary variables can also be explored (**All names**).

```
In[244]:=
      Kt

Out[244]=
      {{Kt11 | Φ1;x, Kt12 | Φ1;y}, {--Φ2;x | -Kt21, Kt22 | Φ2;y}}
```

Analyzing the structure of the program

The SMSAnalyze function 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.

```
In[87]:=  SMSAnalyze[]
```



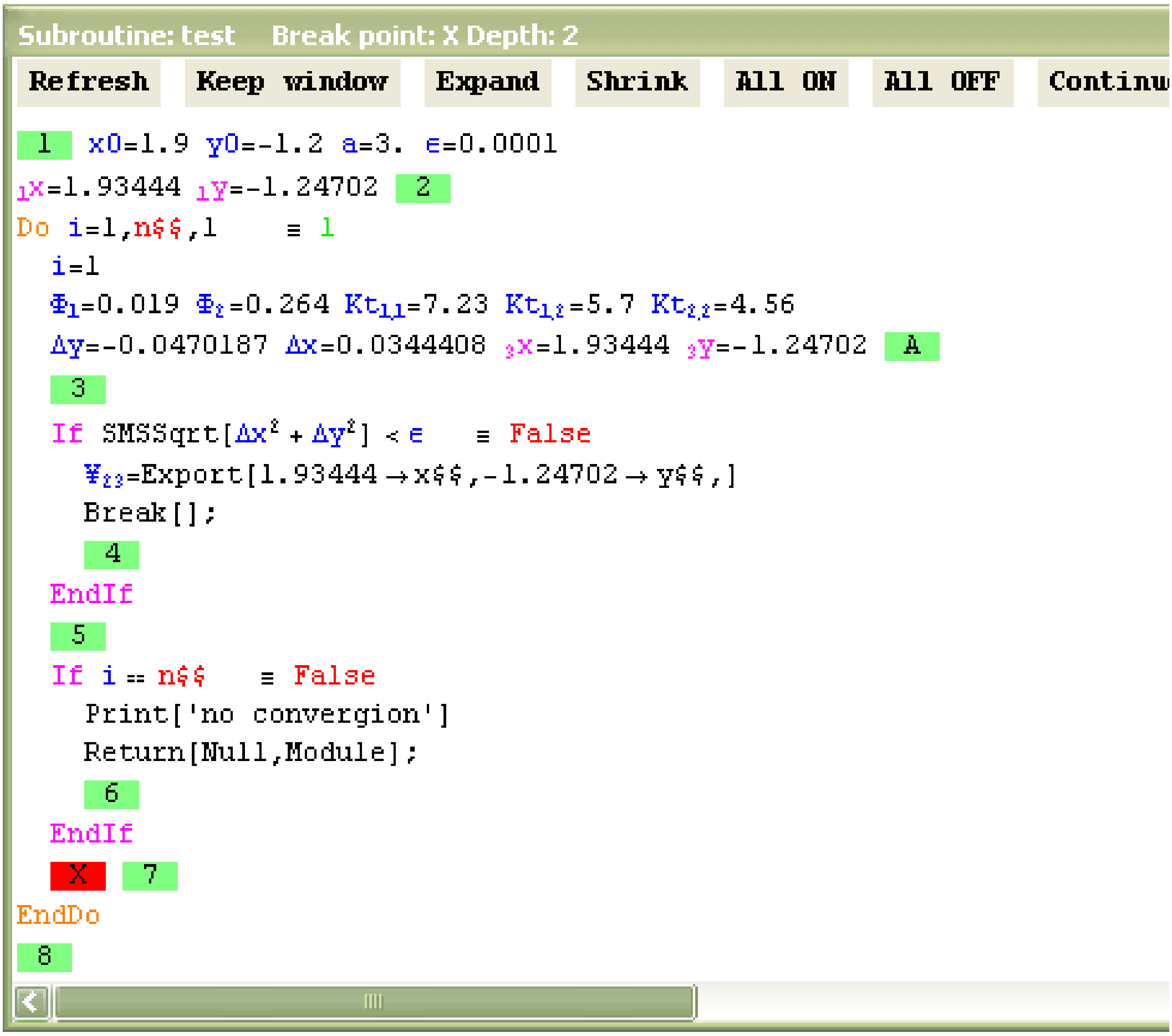
Run time debugging

The SMSAnalyze function is also called automatically during the run time by the SMSExecuteBreakPoint function. The SMSExecuteBreakPoint function can be inserted into the source code by SMSSetBreak function. Break points are inserted only if the code is generated with the "Mode"→"Debug" option. In "Debug" mode the system also automatically generates file with the name *"sessionname.dbg"* where all the information necessary for the run-time debugging is stored. The data is restored from the file by the *SMSLoadSession* command. 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 a 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.

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

```
In[88]:= << AceGen`;
         << "test.m";
         SMSLoadSession["test"];
         x = 1.9; y = -1.2;
         test[x, y, 3., 0.0001, 10]
```

At the break point the *SMSAnalyze* function now produces separate window where the structure of the program is displayed together with the links to all generated formulae and the actual values of the auxiliary variables. The current break point is displayed with the red background.



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 **`Continue`** button. New break point will close previously generated debug window Closing of the window can be prevented by pressing the **`Keep window`** button. Break points can be switched on and off by pressing the button at the position of the break point.

Button legend:

A ⇒ is the button that represents active user defined break point.

B ⇒ is the button that represents the position in a program where the program has stopped.

1 ⇒ is the button that represents automatically generated inactive break point. The break points are automatically generated at the end of If.. else..endif and Do...enddo structures.

`Refresh` ⇒ refresh the contents of the debug window.

`Keep window` ⇒ prevents automatic closing of the debug window

Expand ⇒ increase the extend of the variables that are presented

Shrink ⇒ decrease the extend of the variables that are presented

All ON ⇒ enable all breaks points

All OFF ⇒ disable all breaks points

Continue ⇒ continue to the next break point

Here the break point "X" is inactivated and the break point "A" is activated. The break point "A" is given a pure function that is executed whenever the break point is called. Note that the *SMSLoadSession* also restores all definitions of the symbols that have been assigned value during the *AceGen* session (e.g. the definition of the *Kt* variable in the current example).

```
In[93]:= << AceGen`;
         << "test.m";
         SMSLoadSession["test"];
         SMSClearBreak["X"];
         SMSActivateBreak["A", Print[Kt] &];
         x = 1.9; y = -1.2;
         test[x, y, 3., 0.0001, 10]

           {{7.23, 5.7}, {-1.44, 4.56}}

           {{7.48513, 5.80332}, {-1.55506, 4.82457}}

           {{7.4744, 5.79955}, {-1.55185, 4.81646}}
```

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).

In[100]:=
```
<< AceGen`;
SMSInitialize["test", "Language" -> "Mathematica", "Mode" -> "Debug"];
SMSModule["Test", Real[u$$[3], x$$, L$$, g$$[3]]];
{x, L} ⊨ {SMSReal[x$$], SMSReal[L$$]};
ui ⊨ Array[SMSReal[u$$[#]] &, 3];
Ni ⊨ {x/L, 1 - x/L, x/L (1 - x/L)};
u ⊨ Ni.ui;
f ⊨ u^2;
g ⊨ SMSD[f, ui];
SMSExport[g, g$$];
SMSSetBreak["x"];
SMSWrite[];
```

```
time=0  variable= 0 ≡ {}
Forward differentiation of 5 variables.
[0]  Consistency check - global
[0]  Consistency check - expressions
[0]  Generate source code :
Method : Test 17 formulae, 119 sub-expressions
Events: 0
[0] Final formating
[0] File created : test.m Size : 1410
```

We have several possibilities how to explore the derived formulae and generated code and how to verify the correctness of the model and of the generated code (see also User Interface).

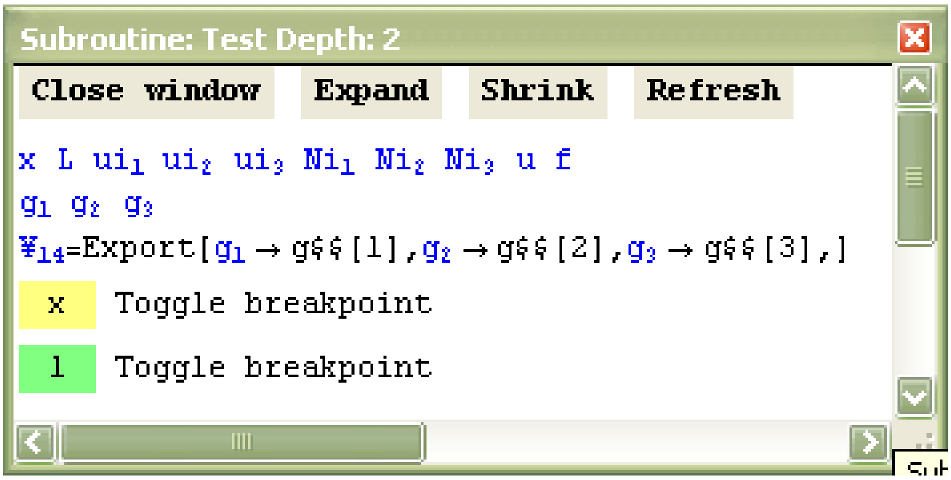
The first possibility is to explore the generated formulae interactively with *Mathematica* in order to see whether their structure is logical.

In[112]:=
```
u
```

Out[112]=
```
u
```

In the case of more complex code, the SMSAnalyze function 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 (see also User Interface).

In[113]:=
```
SMSAnalyze[]
```



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.

In[114]:=

```
<<"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.

In[115]:=

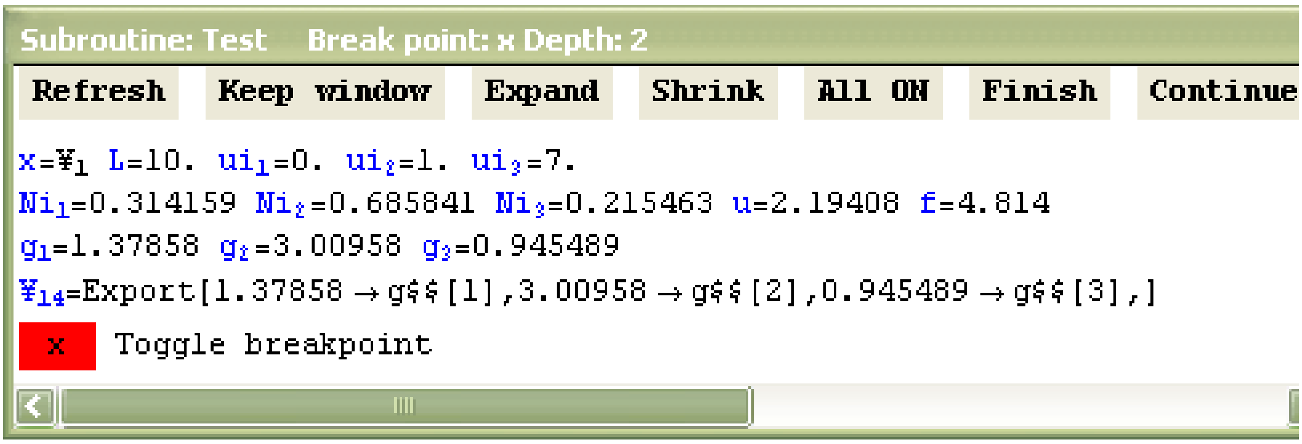
```
xv = π; Lv = 10.; uv = {0., 1., 7.}; gv = {Null, Null, Null};
```

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

In[116]:=

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

Here the contents of the interactive debugger is displayed. See also `SMSAnalyze`.



Here the numerical results are displayed.

```
In[117]:=
    gv
Out[117]=
    {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.

```
In[118]:=
    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.

```
In[119]:=
    Test[uv, xv, Lv, gv]
```

Here the partially evaluated gradient is displayed.

```
In[120]:=
    gv // Expand
Out[120]=
```

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

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*.

```
In[121]:=
    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
Out[127]=
```

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

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

In[128]:=

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

Out[129]=

```
{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 more easily understood and explored.

In[130]:=

```
!!test.m
```

```
(*************************************************************
* AceGen    VERSION                                          *
*           Co. J. Korelc  2006           20.8.2006 23:31    *
**************************************************************
User : Korelc
Evaluation time                 : 0 s     Mode  : Debug
Number of formulae              : 17      Method: Automatic
Module                          : Test size : 119
Total size of Mathematica  code : 119 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["x","test",1,2];
$VV[16]=0;(*debug*)
SMSExecuteBreakPoint["2","test",1,3];
$VV[17]=0;(*debug*)
];
```

Several modifications of the above procedures are possible.

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	$\begin{array}{l} a_{1\ldots i} \circ b_{1\ldots j} \\ c_{1\ldots k} \circ d_{1\ldots j} \\ b_n \equiv d_n \end{array} \Longrightarrow \begin{array}{l} v_1 := b_{1\ldots j} \\ a_{1\ldots i} \circ b_{1\ldots j} \Rightarrow a_{1\ldots i} \circ v_1 \\ c_{1\ldots k} \circ d_{1\ldots j} \Rightarrow c_{1\ldots k} \circ v_1 \end{array}$
(e)	inverse value	$e_1 \equiv \frac{1}{e_2} \Longrightarrow \begin{cases} v_1 := e_2 \\ e_1 \Rightarrow \frac{1}{v_1} \end{cases}$

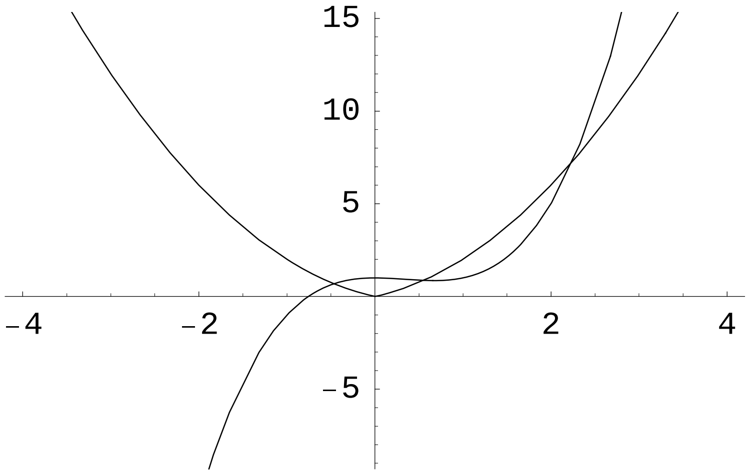
In the formulae above, e_i, a_i, b_i, c_i, d_i are arbitrary expressions or sub-expressions, and v_i are auxiliary variables. Formula $e_i \equiv e_j$ means that the signature of the expression e_i is identical to the signature of the expression e_j. Expressions do not need to be syntactically identical. 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 $f_2 = \text{Abs}[x] + x^2$.

In[131]:=

```
Plot[{x^3 - x^2 + 1, Abs[x] + x^2}, {x, -4, 4}, TextStyle -> {FontSize -> 12}]
```

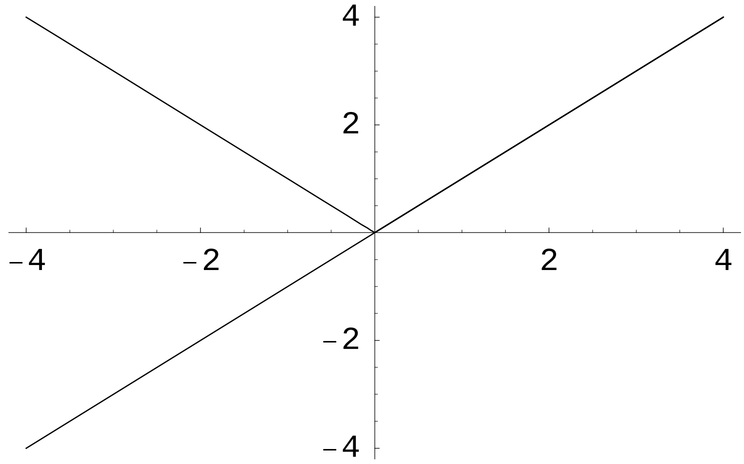


Out[131]=

```
- Graphics -
```

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

In[132]:=
```
Plot[{x, Abs[x]}, {x, -4, 4}, TextStyle -> { FontSize -> 12}]
```



Out[132]=
```
- Graphics -
```

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`.

See also: `Signatures of the expressions`

Program Flow Control

AceGen can automatically generate conditionals (SMSIf, SMSElse, SMSEndIf construct) and loops (SMSDo , SMSEndDo 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. Additionally, we can include parts of the final source code verbatim (SMSVerbatim statement).

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

Example 1: Newton-Raphson

The generation of the Fortran subroutine calculates the zero of function f(x) = $x^2 + 2\,\mathrm{Sin}[x^3]$ by using Newton-Raphson iterative procedure. The source code is written in C language.

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

```
In[133]:=
    << AceGen`;
    SMSInitialize["test", "Language" -> "C"];
    SMSModule["test", Real[x0$$, r$$]];
    x ⊣ SMSReal[x0$$];
```

This starts iterative loop.

```
In[137]:=
    SMSDo[i, 1, 30, 1, {x}];
```

Description of the Newton-Raphson iterative procedure.

In[138]:=

$$f \vDash x^2 + 2\,\mathrm{Sin}[x^3];$$

$$dx \vDash -\frac{f}{\mathrm{SMSD}[f,\ x]};$$

$$x \dashv x + dx;$$

This starts the "If" construct where convergence of the iterative solution is checked.

```
In[141]:=
    SMSIf[Abs[dx] < .00000001];
```

Here we exit the "Do" loop. This is verbatim included in the source code.

```
In[142]:=
    SMSBreak[];
```

This ends the "If" construct .

```
In[143]:=
    SMSEndIf[];
```

Here the divergence of the Newton-Raphson procedure is recognized and reported and the program is aborted.

```
In[144]:=
    SMSIf[i == 15];
      SMSPrint["no convergence"];
      SMSReturn[];
    SMSEndIf[];
```

This ends the "Do" loop.

```
In[148]:=
    SMSEndDo[x];
```

```
In[149]:=
    SMSExport[x, r$$];
    SMSWrite[];
```

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

[0] File created : **test.c** Size : 1017

In[151]:=

!!test.c

```
/*************************************************************
* AceGen    VERSION                                          *
*           Co. J. Korelc  2006           20.8.2006 23:31    *
**************************************************************
User : Korelc
Evaluation time                 : 0 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];
};
```

Example 2: 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].

In[152]:=

```
<< AceGen`;
SMSInitialize["test", "Language" -> "Fortran"];
SMSModule["test", Real[gp$$[ng$$], gw$$[ng$$], a$$, b$$, r$$], Integer[ng$$]];
int ⫤ 0;
SMSDo[i, 1, SMSInteger[ng$$], 1, int];
```

Here the x which corresponds to the i-th Gauss point is calculated by the built-in Solve function.

```
In[157]:=
    Clear[k, n];
     x ⊨ SMSReal[
       (k gp$$[i] + n) /. Solve[{k (-1) + n == a$$, k 1 + n == b$$}, {k, n}][[1]] // Simplify] ;
    int ⊣ int + SMSReal[gw$$[i]] (x^2 + 2 Sin[x^3]);
   SMSEndDo[int];
   SMSExport[int, r$$];
   SMSWrite[];
```

Method : **test** 4 formulae, 52 sub-expressions

[0] File created : **test.f** Size : 950

```
In[163]:=
  !!test.f
```

```
!**************************************************************
!* AceGen    VERSION                                          *
!*           Co. J. Korelc  2006            20.8.2006 23:31   *
!**************************************************************
! User : Korelc
! Evaluation time                 : 0 s     Mode  : Optimal
! Number of formulae              : 4       Method: Automatic
! Subroutine                      : test size :52
! Total size of Mathematica  code : 52 subexpressions
! Total size of Fortran code      : 382 bytes

!******************* S U B R O U T I N E **********************
      SUBROUTINE test(v,gp,gw,a,b,r,ng)
      IMPLICIT NONE
      include 'sms.h'
      INTEGER ng,i2
      DOUBLE PRECISION v(5005),gp(ng),gw(ng),a,b,r
      v(1)=0d0
      DO i2=1,int(ng)
       v(3)=(a+b+(-a+b)*gp(i2))/2d0
       v(1)=v(1)+gw(i2)*((v(3)*v(3))+2d0*dsin(v(3)**3))
      ENDDO
      r=v(1)
      END
```

Algebraic Operations

Automatic Differentiation

Differentiation is an arithmetic operation that plays crucial role in the development of new numerical procedures. The procedure implemented in the *AceGen* system represents a version of automatic differentiation technique. The automatic differentiation generates a program code for the derivative from a code for the basic function. The vector of the new auxiliary variables, generated during the simultaneous simplification of the expressions, is a kind of 'pseudo' code, which makes the automatic differentiation with *AceGen* possible. *AceGen* uses *Mathematica*'s symbolic differentiation functions for the differentiation of explicit parts of the expression. The version of reverse or forward mode of 'automatic differentiation' technique is then employed 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.

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.

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 , $i = 1..n$ 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 $\nabla y = \{\frac{\partial y}{\partial v_1}, \frac{\partial y}{\partial v_2}, \ldots, \frac{\partial y}{\partial v_n}\}$. To do this we must resolve dependencies due to the implicitly contained variables. Two approaches can be used for this, often recalled as forward and reverse mode of automatic differentiation.

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

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

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

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

$\nabla y = \nabla v_m$

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

In contrast to the forward mode, the reverse mode propagates adjoints, that is, the derivatives of the final values, with respect to auxiliary variables. First we associate the scalar derivative $\overline{v_i}$ with each auxiliary variable v_i.

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

$\nabla \mathrm{y} = \left\{\frac{\partial y}{\partial v_i}\right\} = \{\overline{v_i}\}$ *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

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

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

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

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

The reverse mode is implemented as follows

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

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

SMSD (see SMSD) function in *AceGen* does automatic differentiation by using forward or backward mode of automatic differentiation (see examples in Standard AceGen Procedure)

Differentiation is an example where the problems involved in simultaneous simplification are obvious. The table below considers the simple example of the two expressions *x* , *y* and the differentiation of *y* with respect to *x*. *L(a)* is an arbitrary large expression and v_1 is an auxiliary variable. From the computational point of view, simplification A is the most efficient and it gives correct results for both values *x* and *y*. However, when used in a further operations, such as differentiation, it obviously leads to wrong results. On the other hand, simplification B has one more assignment and gives correct results also for the differentiation. To achieve maximal efficiency both types of simplification are used in the *AceGen* system. During the derivation of the formulae type B simplification is performed.

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

At the end of the derivation, 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 basic undependent variables 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 SMSFreeze).

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 .

```
In[164]:=
```

Example 1: simple derivative

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

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

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

```
In[165]:=
     << 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.

```
In[171]:=
     zx ⊨ SMSD[z, x];
```

```
In[172]:=
     SMSExport[zx, r$$];
     SMSWrite[];
```

```
Method : test 4 formulae, 38 sub-expressions
[0] File created : test.c Size : 783
```

In[174]:=

```
!!test.c
```

```
/*************************************************************
* AceGen    VERSION                                          *
*           Co. J. Korelc  2006            20.8.2006 23:31   *
*************************************************************
User : Korelc
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            : 216 bytes*/
#include "sms.h"

/******************* S U B R O U T I N E *********************/
void test(double v[5001],double (*x),double (*r))
{
v[5]=2e0*(*x);
v[3]=Power((*x),2);
v[6]=v[5]*cos(v[3]);
(*r)=3e0*v[5]+v[6]*(2e0+1e0/sin(v[3]));
};
```

Example 2: differentiation of the complex program structure

Generation of the Matlab M-function file which evaluates derivative of function $f(x) = 3\,z^2$ with respect to x, where z is

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

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

In[175]:=

```
<< AceGen`;
SMSInitialize["test", "Language" -> "Matlab"];
SMSModule["test", Real[x$$, r$$]];
x ⊨ SMSReal[x$$];
SMSIf[x > 0];
 y ⊨ Sin[ x^2];
 z ⫤ 3 x^2 + 2 y + Log[y];
SMSElse[];
 z ⊣ Cos[x^3];
SMSEndIf[z];
fx ⊨ SMSD[3 z^2, x];
SMSExport[fx, r$$];
SMSWrite[];
```

```
Method : test 11 formulae, 88 sub-expressions
[0] File created : test.m Size : 1157
```

In[188]:=
!!test.m

```
%**************************************************************
%* AceGen    VERSION                                          *
%*           Co. J. Korelc  2006           20.8.2006 23:31    *
%**************************************************************
% User : Korelc
% 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 Matlab code      : 370 bytes

%*********************** F U N C T I O N **************************
function[x,r]=test(x,r);
v=zeros(5001,'double');
v(10)=Power(x,2);
v(13)=3e0*v(10);
if(x>0)
 v(6)=2e0*x;
 v(7)=v(6)*cos(v(10));
 v(3)=sin(v(10));
 v(8)=3e0*v(6)+(2e0+1/v(3))*v(7);
 v(5)=v(13)+2e0*v(3)+log(v(3));
else;
 v(9)=Power(x,3);
 v(8)=-(v(13)*sin(v(9)));
 v(5)=cos(v(9));
end;
r=6e0*v(5)*v(8);

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

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

end
```

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 *(see* SMSReplaceAll *).*

Example

A typical example is a Taylor series expansion,

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

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

In[189]:=

```
<< 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[];
```

```
Method : Test 3 formulae, 48 sub-expressions

[0] File created : test.f Size : 889
```

In[200]:=

```
!!test.f
```

```
!**************************************************************
!* AceGen    VERSION                                          *
!*           Co. J. Korelc  2006           20.8.2006 23:31    *
!**************************************************************
! User : Korelc
! 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
```

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)≠0).*

Example

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

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}.$$

```
In[201]:=
<< AceGen`;
SMSInitialize["test", "Language" -> "C"];
SMSModule["Test", Real[a$$[4, 4], b$$[4], x$$[4]]];
a ⊨ SMSReal[Array[a$$, {4, 4}]];
b ⊨ SMSReal[Array[b$$, {4}]];
x ⊨ SMSLinearSolve[a, b];
SMSExport[x, x$$];
SMSWrite[];
```

```
Solution of 4 linear equations.

Method : Test 18 formulae, 429 sub-expressions

[1] File created : test.c Size : 1427
```

In[209]:=

!!test.c

```
/*************************************************************
* AceGen    VERSION                                          *
*           Co. J. Korelc  2006           20.8.2006 23:32   *
*************************************************************
User : Korelc
Evaluation time                 : 1 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];
};
```

Other Algebraic Computations

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 (see example in section Non – local operations).

Advanced Features

Arrays

AceGen has no prearranged higher order matrix, vector, or tensor operations. We can use all *Mathematica* built-in functions or any of the external packages to perform those operations. After the operation is performed, we can simplify the result by using *AceGen* optimization capabilities. In this case, one auxiliary variable represents one element of the vector, matrix or tensor.

However, sometimes we wish to express an array of expressions with a single auxiliary variable, or to make a reference to the arbitrary element of the array of expressions. *AceGen* enables some basic operations with one dimensional arrays. We can create one dimensional array of symbolic expressions with the fixed length and contents or variable length and contents one dimensional array.

Arrays are physically stored at the end of the global vector of formulae. The dimension of the global vector (specified in SMSInitialize) is automatically extended in order to accommodate additional arrays. In the final source code, the fixed length array is represented as a sequence of auxiliary variables and formulae. Definition of the variable length array only allocates space on the global vector.

Fixed length array is represented by the data object with the head *SMSGroupF* (*AceGen* array object). The variable length array data object has head *SMSArrayF*. Array data object represents array of expressions together with the information regarding random evaluation. Reference to the particular or arbitrary element of the array is represented by the data object with the head *SMSIndexF* (*AceGen* index object).

See also: SMSArray , SMSPart .

Example : Arrays

We wish to create a function that returns a dot product of the two vectors of expressions and the *i*-th element of the second vector.

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

```
In[541]:=
      << AceGen`;
      SMSInitialize["test", "Language" -> "Fortran"];
      SMSModule["test", Real[x$$, r$$, s$$, t$$], Integer[n$$, m$$]];
      x ⊧ SMSReal[x$$];
      n ⊧ SMSInteger[n$$];
```

This creates the *AceGen* array object with fixed length and 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.

```
In[546]:=
      SMSArray[{x, x^2, 0, π}]
```

```
Out[546]//DisplayForm=
      𝔾[x, x^2, 0, π]
```

If an array is required as auxiliary variable then we have to use one of the functions that introduces a new auxiliary variable. Note that a single auxiliary variable has been created representing arbitrary element of the array. The signature of the array is calculated as perturbed average signature of all array elements.

In[547]:=
```
fixedA ⊨ SMSArray[{x, x^2, 0, π}]
```
Out[547]=
```
fixedA
```

This creates the second *AceGen* array object with fixed length and contents

In[548]:=
```
fixedB ⊨ SMSArray[{3 x, 1 + x^2, Sin[x], Cos[x π]}];
```

This calculates a dot product of vectors g1 and g2.

In[549]:=
```
dot ⊨ SMSDot[fixedA, fixedB];
```

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

In[550]:=
```
fixedBnth ⊨ SMSPart[fixedB, n]
```
Out[550]=
```
fixedBnth
```

This allocates space on the global vector of formulae and creates variable length *AceGen* array object *varlength*.

In[551]:=
```
varlength ⫤ SMSArray[10];
```

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

In[552]:=
```
SMSDo[i, 1, 10];
 varlength ⊣ SMSReplacePart[varlength, 1 / i, i];
SMSEndDo[varlength];
```

This creates an index to the *n*-th element of the *varlength* array.

In[555]:=
```
varlengthmth ⊨ SMSPart[varlength, SMSInteger[m$$]];
```

In[556]:=
```
SMSExport[{dot, fixedBnth, varlengthmth}, {r$$, s$$, t$$}];
SMSWrite["test"];
```
```
Method : test 6 formulae, 96 sub-expressions
[0] File created : test.f Size : 1137
```

In[558]:=

```
!!test.f
```

```
!**************************************************************
!* AceGen    VERSION                                          *
!*           Co. J. Korelc  2006            21.8.2006 11:20   *
!**************************************************************
! User : Korelc
! 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,i9
      DOUBLE PRECISION v(5023),x,r,s,t
      v(4)=x**2
      v(5000)=x
      v(5001)=v(4)
      v(5002)=0d0
      v(5003)=0.3141592653589793d1
      v(5004)=3d0*x
      v(5005)=1d0+v(4)
      v(5006)=dsin(x)
      v(5007)=dcos(0.3141592653589793d1*x)
      DO i9=1,10
       v(5007+i9)=1d0/i9
      ENDDO
      r=SMSDot(v(5000),v(5004),4)
      s=v(5003+int(n))
      t=v(5007+int(m))
      END
```

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.

Optimization is only possible for a scalar function of scalar input parameters and not for user defined subroutines with several input/output parameters of various types (how to incorporate arbitrary subroutines see `SMSCall`).

Example 1: Simple case where function in MMA exists, but it does not produce a proper C or FORTRAN output

In[391]:=

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

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

In[393]:=

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

In[394]:=
```
SMSModule["sub1", Real[x$$, y$$[5]]];
x ⊨ SMSReal[x$$];
SMSExport[Tan[x], y$$[1]];
SMSWrite[];
```

```
Method : sub1 1 formulae, 7 sub-expressions
[0] File created : test.f Size : 765
```

In[398]:=
```
!!test.f
```

```
!**************************************************************
!* AceGen    VERSION                                          *
!*           Co. J. Korelc  2006             20.8.2006 23:39  *
!**************************************************************
! User : Korelc
! 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      : 205 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)=mydtan(x)
      END
```

Example 2: General user defined function

This adds alternative definition of Power function MyPower[x, y] ≡ x^y that assumes that x>0 and

D[MyPower[x,y],x] = y $\frac{\text{MyPower}[x,y]}{x}$,

D[MyPower[x,y],y] = MyPower[x, y] Log[x].

In[399]:=
```
<< AceGen`;
SMSInitialize["test", "Language" → "C"];
```

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

In[401]:=
```
SMSAddFormat[MyPower[i_, j_] ↦
   Switch[SMSLanguage, "Mathematica", i^j, "Fortran", i^j, "C", "Power"[i, j]]
  ];
```

Here the derivatives of MyPower with respect to all parameters are defined.

In[402]:=
```
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 numerical evaluation of MyPower with *p*-digit precision.

```
In[406]:=
     N[MyPower[i_, j_], p_] := i^j;
```

```
In[407]:=
     SMSModule["sub1", Real[x$$, y$$, z$$]];
     x ⊧ SMSReal[x$$];
     y ⊧ SMSReal[y$$];

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

```
Method : sub1 1 formulae, 22 sub-expressions
[0] File created : test.c Size : 731
```

```
In[412]:=
     !! test.c
```

```
/*************************************************************
* AceGen    VERSION                                          *
*           Co. J. Korelc  2006            20.8.2006 23:39   *
*************************************************************
User : Korelc
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);
};
```

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:

⇒ there exists implicit dependency between variables that has to be considered for the differentiation,

⇒ there exists explicit dependency between variables that has to be neglected for the differentiation,

⇒ the evaluation of the derivative code would lead to numerical errors.

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. In the case of implicitly dependent relations, the gradient has to be provided by the user with the SMSDefineDerivative command.

With the SMSFreeze[*exp*, "Dependency"] the true dependencies of *exp* with respect to auxiliary variables are neglected and all partial derivatives are taken to be 0.

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

Example 1: Implicit dependencies

The generation of the *FORTRAN* subroutine calculates the derivative of function $f = (\xi + \eta)^2$ with respect to X. Transformation from (X,Y) coordinate system into the (ξ,η) coordinate system is defined by:

$X = N_i X_i$,
$Y = N_i Y_i$,
where
$N = \{(1-\xi)(1-\eta), (1+\xi)(1-\eta), (1+\xi)(1+\eta), (1-\xi)(1+\eta)\}$,
$X = \{X_1, X_2, X_3, X_4\}$,
$X = \{Y_1, Y_2, Y_3, Y_4\}$.

```
In[413]:=
      << AceGen`;
      SMSInitialize["test", "Language" -> "Fortran"]
      SMSModule["Test", Real[Xi$$[4], Yi$$[4], ksi$$, eta$$, fx$$]];
      Xi ⊨ SMSReal[Array[Xi$$, 4]];
      Yi ⊨ SMSReal[Array[Yi$$, 4]];
      {ξ, η} ⊨ {SMSReal[ksi$$], SMSReal[eta$$]};
      Ni ⊨ 1/4 {(1 - ξ) (1 - η), (1 + ξ) (1 - η), (1 + ξ) (1 + η), (1 - ξ) (1 + η)};
```

X and Y are the basic derivative variables. To prevent wrong simplifications, we have to define unique signatures for the definition of X and Y.

```
In[420]:=
      X ⊨ SMSFreeze[Ni.Xi];
      Y ⊨ SMSFreeze[Ni.Yi];
```

Here the Jacobian matrix of nonlinear coordinate transformation is calculated.

```
In[422]:=
      Jm ⊨ ( SMSD[X, ξ]  SMSD[X, η]
             SMSD[Y, ξ]  SMSD[Y, η] );
      Jmi ⊨ SMSInverse[Jm];
```

Here the implicit derivatives are defined.

```
In[424]:=
      SMSDefineDerivative[{ξ, η}, {X, Y}, Jmi];
```

The implicit dependencies of ξ and η are now taken into account when the derivation of f is made with respect to X.

```
In[425]:=
      f ⊨ (ξ + η)^2;
      fX ⊨ SMSD[f, X]

Out[426]=
      fX
```

In[427]:=

```
SMSExport[fX, fx$$];
SMSWrite[];
```

Method : **Test** 12 formulae, 235 sub-expressions

[1] File created : **test.f** Size : 1253

In[429]:=

```
!!test.f
```

```
!**************************************************************
!* AceGen    VERSION                                          *
!*           Co. J. Korelc  2006            20.8.2006 23:39   *
!**************************************************************
! User : Korelc
! Evaluation time                 : 1 s     Mode  : Optimal
! Number of formulae              : 12      Method: Automatic
! Subroutine                      : Test size :235
! Total size of Mathematica  code : 235 subexpressions
! Total size of Fortran code      : 677 bytes

!****************** S U B R O U T I N E *********************
      SUBROUTINE Test(v,Xi,Yi,ksi,eta,fx)
      IMPLICIT NONE
      include 'sms.h'
      DOUBLE PRECISION v(5001),Xi(4),Yi(4),ksi,eta,fx
      v(16)=1d0-ksi
      v(14)=1d0+ksi
      v(17)=1d0+eta
      v(12)=1d0-eta
      v(23)=(-(v(12)*Yi(1))+v(12)*Yi(2)+v(17)*(Yi(3)-Yi(4)))/4d0
      v(37)=v(14)*v(23)
      v(34)=v(16)*v(23)
      v(24)=(-(v(14)*Yi(2))+v(14)*Yi(3)+v(16)*(-Yi(1)+Yi(4)))/4d0
      v(36)=v(17)*v(24)
      v(35)=v(12)*v(24)
      v(25)=4d0/(v(34)*Xi(1)-v(35)*Xi(1)+(v(35)+v(37))*Xi(2)+(v(36)-v
     &(37))*Xi(3)-v(34)*Xi(4)-v(36)*Xi(4))
      fx=2d0*(eta+ksi)*(-v(23)+v(24))*v(25)
      END
```

Example 2: Partial derivative

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:

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

```
In[430]:=
      SMSInitialize["test", "Language" -> "Fortran"]
      SMSModule["Test", Real[x$$, f$$, dfdx$$]];
      x ⊢ SMSReal[x$$];
      α ⊢ Cos[x];
      SMSIf[SMSAbs[α] > 10^-10];
       f ⫤ Sin[2 α^2] / α;
      SMSElse[];
       f ⊣ SMSFreeze[Limit[Sin[2 α^2] / α, α → 0], "Dependency" ->
             {{α, Limit[D[Sin[2 α^2] / α, α] // Evaluate, α → 0]}}];
      SMSEndIf[True, f];
      dfdx ⊨ SMSD[f, x];
      SMSExport[dfdx, dfdx$$];
      SMSWrite[];

          Method : Test 6 formulae, 51 sub-expressions

          [0] File created : test.f Size : 995

In[442]:=
      !!test.f
```

```
!**************************************************************
!* AceGen    VERSION                                          *
!*           Co. J. Korelc  2006            20.8.2006 23:39    *
!**************************************************************
! User : Korelc
! 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 $N_{d.o.f} + (N_{d.o.f})^2$ terms. Thus, explicit code for all terms can be generated only if the number of unknowns is small. If the number of parameters of the problem is large, then characteristic expressions for arbitrary term of gradient or Hessian have to be derived. The first step is to present a set of parameters as a union of disjoint subsets. The subset of unknown parameters, denoted by $\boldsymbol{a_i}$, is defined by

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

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

$\boldsymbol{a_i} \bigcap \boldsymbol{a}_j = \phi$,i≠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

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

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

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

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.

In[443]:=
```
<< AceGen`;
SMSInitialize["test", "Language" -> "Fortran"]
SMSModule["Test", Real[u$$[3], x$$, L$$, g$$[3]]];
{x, L} ⊨ {SMSReal[x$$], SMSReal[L$$]};
ui ⊨ SMSReal[Array[u$$, 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.

In[452]:=
```
fui ⊨ SMSD[f, ui, i];
```

This is how the formula is displayed if we explore the structure of the created auxiliary variable.

In[453]:=
```
fui
```

Out[453]=
```
fui
```

In[454]:=
```
SMSExport[fui, g$$[i]];
SMSEndDo[];
SMSWrite[];
```

```
Method : Test 6 formulae, 95 sub-expressions
[1] File created : test.f Size : 1013
```

In[457]:=

```
!!test.f
```

```
!**************************************************************
!* AceGen    VERSION                                          *
!*           Co. J. Korelc  2006            20.8.2006 23:39   *
!**************************************************************
! User : Korelc
! 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 $\boldsymbol{a_i}$.

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

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

In[458]:=

```
<< 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[Array[ul$$, 4]];
vi ⊨ SMSReal[Array[vl$$, 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[i, 1, 4];
```

Here the characteristic formulae for the sub-vector of the gradient vector are created.

```
In[468]:=
        {g1i, g2i} ⊨ {SMSD[f, ui, i], SMSD[f, vi, i]};
```

Characteristic formulae have to be exported to the correct places in a gradient vector.

```
In[469]:=
        SMSExport[{g1i, g2i}, {g$$[2 i - 1], g$$[2 i]}];
        SMSDo[j, 1, 4];
```

Here the 2*2 characteristic sub-matrix of the Hessian is created.

```
In[471]:=
          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]}}];
         SMSEndDo[];
        SMSEndDo[];
       SMSWrite[];

        Method : Test 19 formulae, 258 sub-expressions

        [1] File created : test.c Size : 1510
```

In[476]:=

!!test.c

```
/*************************************************************
* AceGen    VERSION                                          *
*           Co. J. Korelc  2006           20.8.2006 23:39   *
*************************************************************
User : Korelc
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.

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

```
In[23]:= << AceGen`;
         SMSInitialize["test", "Language" -> "Mathematica"];
         SMSModule["Test", Real[x$$, y$$, z$$]];
         {x, y, z} ⊨ {SMSReal[x$$], SMSReal[y$$], SMSReal[z$$]};

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

Out[27]= x^2 + 4 x y + y^2 + 2 y z + z^2
```

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

```
In[28]:= fx = SMSSmartReduce[f, x, Collect[#, x] &] // Normal

Out[28]= x^2 + §1 + x §2
```

The following vector of auxiliary variables is created.

```
In[29]:= SMSShowVector[0]

           1 $V[1, 1]    {x} = x$$
           2 $V[2, 1]    {y} = y$$
           3 $V[3, 1]    {z} = z$$
           4 $V[4, 1]    {§1} = y^2 + 2 y z + z^2
           5 $V[5, 1]    {§2} = 4 y
```

```
In[31]:= fint = ∫fx ⅆx
```

Out[31]= $\frac{x^3}{3} + x\ §1 + \frac{x^2\ §2}{2}$

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

```
In[32]:= fr = SMSSmartRestore[fint, y, Collect[#, y] &] // Normal
```

Out[32]= $x\,y^2 + §3 + y\ §4$

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

```
In[33]:= SMSShowVector[0];
```

1	\$V[1, 1]	{x}	=	x\$\$
2	\$V[2, 1]	{y}	=	y\$\$
3	\$V[3, 1]	{z}	=	z\$\$
4	\$V[4, 1]	{§1}	=	$y^2 + 2\,y\,z + ¥_6$
5	\$V[5, 1]	{§2}	=	$4\,y$
6	\$V[6, 1]		=	z^2
7	\$V[7, 1]	{§3}	=	$\frac{x^3}{3} + x\ ¥_6$
8	\$V[8, 1]	{§4}	=	$2\,x^2 + 2\,x\,z$

See also: SMSSmartReduce , SMSSmartRestore .

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 (`SMSInitalize`) then *AceGen* automatically increase precision with the *SetPrecision[exp,SMSEvaluatePrecision]* command.

```
In[41]:= << AceGen`;
         SMSInitialize["test", "Language" -> "Mathematica", "Mode" -> "Debug"];
         SMSModule["test"];

           time=0  variable= 0 ≡ {}

In[44]:= x ⊢ π;

In[45]:= y ⊢ 3.1415;

           Precision of the user input real number
             3.1415 has been automatically increased.
           See also: Signatures of the Expressions  Troubleshooting
```

Example 2

This initializes the *AceGen* system, starts the description of the "test" subroutine and sets default precision of the signatures to 40.

```
In[46]:= << AceGen`;
         SMSInitialize["test", "Language" -> "Fortran", "Precision" → 40];
         SMSModule["test", Real[x$$, y$$], Integer[n$$]];
```

Here variable *x* gets automatically generated real random value from interval [0,1], for variable *y* three interval is explicitly prescribed, and an integer external variable *n* also gets real random value.

```
In[49]:= x ⊨ SMSReal[x$$];
         y ⊨ SMSReal[y$$, {-100, 100}];
         n ⊨ SMSInteger[n$$];
```

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

```
In[52]:= {x, y, n} // SMSEvaluate // Print

{0.35742092377640402551381082565181024766625,
 -81.0812438877469432056774710062273571087,
 7.75117845351252629406397842773737580644062}
```

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`

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 *`MathLink`* program is build from the generated source code and installed (see Install) so that functions defined in the source code can be called directly from *Mathematica* (see `Generation of` *`MathLink`* `code`, `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 `CDriver` 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 `MDriver` 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 `Standard FE Procedure`, `About FEAP`)	"Fortran"
"ELFEN"	commercial finite element environment written in *FORTRAN* (see `Standard FE Procedure`, `About ELFEN`)	"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.

```
In[1]:= 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 External Variables . 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 `Generation of MathLink code`) 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" .

```
In[45]:= <<AceGen`;
        SMSInitialize["test","Language"->"Fortran"];
        SMSModule["sub1",Real[x$$,y$$[5]],Integer[i$$],Real[z$$],
                  "mytype"[m$$],"Verbatim"->"COMMON /xxx/a(5)"];
        SMSWrite[];
        !!test.f
```

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

[0] File created : test.f Size : 816
User : Korelc

!**************************************************************
!* AceGen    VERSION                                          *
!*           Co. J. Korelc  2006            20.8.2006 22:34   *
!**************************************************************
! User : Korelc
! 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.ext*"

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"	""	file prepended to the generated source code file
"Substitutions"	{}	list of rules applied on all expressions before the code is generated
"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 the additional optimization loops over the whole code
"IncludeHeaders"	Automatic	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 values are 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 *Mathematica´s* ../AddOns/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.

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

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

```
In[10]:= !!test.mc
```

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

```
In[11]:= SMSInitialize["test", "Language" -> "C"];
         SMSModule["sub1", Real[x$$, y$$[2]]];
         SMSExport[BesselJ[SMSReal[y$$[1]],SMSReal[y$$[2]]],x$$];
         SMSWrite["source","Splice" -> "test.mc",
            "Substitutions"->{BesselJ[i_,j_]:>"mybessel"[i,j]}];

           Method : sub1 1 formulae, 13 sub-expressions

           [0] File created : source.c Size : 742

In[15]:= !!source.c

         /*************************************************************
         * AceGen    VERSION                                          *
         *           Co. J. Korelc  2006            10.10.2006 17:24  *
         **************************************************************
         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]);
         };
```

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

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: `Solid, Finite Strain Element for Direct and Sensitivity Analysis`

Example:

```
CELLTAG, b:3.0.3
     Print["this is cell with tag CELLTAG"]
```

```
In[186]:=
    <<AceGen`;
    SMSInitialize["test", "Language" -> "C"];
    SMSModule["sub1"];
    SMSEvaluateCellsWithTag["CELLTAG"]
```

```
[0-0] Include Tag : CELLTAG (1 input cells)

this is cell with tag CELLTAG
```

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.

```
In[945]:=
    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 : 685

In[950]:=
    !!test.c
```

```
/*************************************************************
* AceGen    VERSION                                          *
*           Co. J. Korelc  2006            21.8.2006 12:23   *
**************************************************************
User : Korelc
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

option	*description*	*default value*
"Output"	"Console" ⇒standard output device {"File", *filename*} ⇒ create a source code sequence that prints out all the expressions $expr_i$ to the file *filename*	"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	True

Options for the *SMSPrint* function.

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 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.

The actual meaning of the standard output device depends on a chosen language as follows:

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

Standard output device.

Example 1: printing out to standard output device

```
In[17]:= << AceGen`;
SMSInitialize["test", "Language" -> "C", "Mode" → "Prototype"];
SMSModule["test", Real[x$$]];

SMSPrint["pi=", π];

SMSPrint["time=", SMSTime[], "Output" → {"File", "test.out"}];

SMSPrint["e=", E, "Output" → {"File", "test.out"}, "Condition" → SMSReal[x$$] > 0];

SMSWrite[];
```

```
Method : test 4 formulae, 11 sub-expressions

[0] File created : test.c Size : 993
```

In[24]:= **!!test.c**

```
/*************************************************************
* AceGen    VERSION                                          *
*           Co. J. Korelc  2006           21.8.2006 18:22    *
*************************************************************
User : Korelc
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            : 417 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");
fprintf(SMSFile,"\n%s %g ","time=",(double)v[2]);
fclose(SMSFile);
if((*x)>0e0){
SMSFile=fopen("test.out","a");
fprintf(SMSFile,"\n%s %g ","e=",(double)0.2718281828459045e1);
fclose(SMSFile);
};
};
```

Numerical environments specific options

"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): −1 ⇒ print outs are active for all elements 0 ⇒ print outs are disabled (default value) >0 ⇒ print out is active if SMTIData["DebugElement"]= SMTIData["CurrentElement"]

Example 2: printing out from numerical environment

In[33]:=
```
<< AceGen`;
SMSInitialize["test", "Environment" -> "AceFEM", "Mode" → "Prototype"];
SMSTemplate["SMSTopology" -> "T1"];
SMSStandardModule["Tangent and residual"];
SMSPrint["'pi='", π];
SMSPrint["'load='", rdata$$["Multiplier"],
  "Output" → "File", "Condition" -> "DebugElement"];
SMSWrite[];
```

Method : **SKR** 3 formulae, 8 sub−expressions

[0] File created : **test.c** Size : 3807

```
In[40]:= !!test.c
```

```
head deleted ...

/****************** S U B R O U T I N E ********************/
void SKR(double v[5001],ElementSpec *es,ElementData *ed,NodeSpec **ns
     ,NodeData **nd,double *rdata,int *idata,double *p,double **s)
{
v[1]=0.3141592653589793e1;
es->Execute(
"Print[\"pi=\",\" \",SMTVData[1]];"
,"SMSPrint");
if((idata[ID_DebugElement]==(-1) ||
idata[ID_CurrentElement]==idata[ID_DebugElement])){
if(strcmp(es->OutFileName,"NONE")!=0){SMTFile=fopen(es->OutFileName,"a");
fprintf(SMTFile,"\n%s %g ","load=",(double)rdata[RD_Multiplier]);
fclose(SMTFile);}
};
};
```

Basic Assignments

SMSR or ⊨

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 A.2.7. It has higher precedence than arithmetical operators like +, -,*,/, but lower than postfix operators like // and /., /... . In these cases the parentheses or 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: Auxiliary Variables , SMSM , SMSS , SMSSimplify

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

```
In[56]:= SMSInitialize["test", "Language" → "Fortran"];
         SMSModule["sub"];
         x ⊧ 1
         y ⊧ Sin[5]

           Module : sub

Out[58]= 1

Out[59]= y
```

SMSV or ⊢

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 precedence table A.2.7 and described in SMSR .

See also: Auxiliary Variables , SMSM , SMSS , SMSSimplify

The new auxiliary variables are created for all expressions.

```
In[60]:= SMSInitialize["test", "Language" → "Fortran"];
         SMSModule["sub"];
         x ⊢ 1
         y ⊢ Sin[5]

           Module : sub

Out[62]= x

Out[63]= y
```

SMSM or ⫤

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 specified in precedence table A.2.7 and described in SMSR .

See also: SMSR , SMSS , SMSFreeze , Auxiliary Variables , SMSReal , SMSInteger, SMSLogical , SMSFictive , SMSIf , SMSDo .

SMSS or ⊣

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 specified in precedence table A.2.7 and described in SMSR .

See also: Auxiliary Variables , SMSIf , SMSDo .

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

In[129]:=

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

Module : sub

Out[131]=

x

Out[132]=

$_{2}x$

Out[133]=

$_{3}x$

In[134]:=

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

Function : **sub** 4 formulae, 12 sub-expressions

[0] File created : **test.f** Size : 764

In[136]:=

```
!!test.f
```

```
C*********************************************************************
C* SMS 4.0 - Symbolic Mechanics System      - FORTRAN          *
C*          Co. J. Korelc  Sept. 1999      6.7.2000 14:13    *
C*********************************************************************
C Evaluation time                 : 0 seconds   Mode : TFFF0FF
C Number of formulae              : 4
C Subroutine                      : sub size :12
C Total size of Mathematica  code : 12 subexpressions
C Total size of Fortran code      : 238 bytes

******************* S U B R O U T I N E *********************
      SUBROUTINE sub(v,x)
      IMPLICIT NONE
      include 'sms.h'
      DOUBLE PRECISION v(501),x
      v(1)=1d0
      v(1)=2d0+v(1)
      v(1)=5d0
      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 .

This creates a new auxiliary variable inside the formula.

In[234]:=

```
SMSInitialize["test"]; SMSModule["sub"];
```

```
Module : sub
```

```
In[235]:=
    1 + 5 * SMSSimplify[Tan[5] + 1]
Out[235]=
    1 + 5 (1¥1)
```

SMSVariables

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

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 details of syntax see `SMSFreeze`)
"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

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* (N[*exte*]) leads to the real type number then the default signature is calculated by it`s pertubation, else the signature is 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: `External Variables` , `Expression Optimization`

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 …}	{}	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
"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` , `External Variables` .

SMSLogical

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

option name	*default value*	
"Subordinate"−> {v_1, v_2 …}	{}	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
"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` , `External Variables` .

SMSRealList

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

Introduction of the list of real type external variables .

option 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

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

```
In[107]:=
    << AceGen`
    SMSInitialize["test", "Language" -> "C"];
    SMSModule["test", Real[a$$[10], b$$[10], c$$[100]], Integer[L$$, i$$]];
    SMSRealList[{"a1", "a2"}, a$$[#] &]

Out[110]=
    {a$$1, a$$2}

In[111]:=
    SMSRealList[{"a3", "a4"}, a$$[#] &]

Out[111]=
    {a$$3, a$$4}

In[112]:=
    SMSRealList["a3"]

Out[112]//DisplayForm=
    a$$3

In[113]:=
    SMSRealList[{"b1", "b2"}, b$$[#] &, "Length" → 5, "Index" → 2]

Out[113]=
    {b$$2, b$$7}

In[114]:=
    SMSRealList[{"b3", "b4"}, b$$[#] &, "Length" → 20, "Index" → 4]

Out[114]=
    {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!!

```
In[115]:=
    {L, i} ⊧ SMSInteger[{L$$, i$$}];
    SMSRealList[{"c1", "c2"}, c$$[#] &, "Length" → Hold[2 L], "Index" → Hold[i + 1]]

Out[116]=
    {c$$1+i, c$$1+i+2 L}

In[117]:=
    SMSRealList[Array["β", 2], c$$[#] &, "Length" → Hold[ L], "Index" → Hold[i]]

Out[117]=
    {c$$i+4 L, c$$i+5 L}
```

In[118]:=
```
TableForm[{SMSRealList["β"[_], "ID"], SMSRealList["β"[_]],
  SMSRealList["β"[_], "Plain"], SMSRealList["β"[_], "Export"]},
 TableHeadings → {{"ID", "AceGen", "Plain", "Export"}, None}]
```

Out[118]//TableForm=

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

In[119]:=
```
SMSRealList["β"[_], "Length"]
```

Out[119]=
```
2 (int)[L$$]
```

SMSExport

SMSExport[*exp*,*ext*]	export the expression *exp* to the external variable *ext*
SMSExport[{*exp1*,*exp2*,...,*expN*},*ext*]	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 (see External Variables). At the end of the session, the external variables are translated into the *FORTRAN* or *C* format.

See also: External Variables .

In[1]:=
```
<< AceGen`;
SMSInitialize["test", "Language" -> "Fortran"];
SMSModule["test", Real[x$$, a$$[2], r$$[2, 2]]];
x ⊧ SMSReal[x$$];

SMSExport[x^2, x$$];

(* three equivalent forms how to export list of two values*)
SMSExport[{1, 2}, a$$];
SMSExport[{3, 4}, {a$$[1], a$$[2]}];
SMSExport[{5, 6}, a$$[#] &];

SMSExport[Array[#1 #2 &, {2, 2}], r$$];
SMSWrite["test"];
```

```
Method : test 5 formulae, 26 sub-expressions

[0] File created : test.f Size : 936
```

In[11]:= **!!test.f**

```
!**************************************************************
!* AceGen    VERSION                                          *
!*           Co. J. Korelc  2006           20.8.2006 23:31    *
!**************************************************************
! User : Korelc
! Evaluation time                 : 0 s     Mode  : Optimal
! Number of formulae              : 5       Method: Automatic
! Subroutine                      : test size :26
! Total size of Mathematica  code : 26 subexpressions
! Total size of Fortran code      : 364 bytes

!****************** S U B R O U T I N E **********************
      SUBROUTINE test(v,x,a,r)
      IMPLICIT NONE
      include 'sms.h'
      DOUBLE PRECISION v(5001),x,a(2),r(2,2)
      x=x**2
      a(1)=1d0
      a(2)=2d0
      a(1)=3d0
      a(2)=4d0
      a(1)=5d0
      a(2)=6d0
      r(1,1)=1d0
      r(1,2)=2d0
      r(2,1)=2d0
      r(2,2)=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 commands inserts into the generated source code the call of external subroutine with the given set of input and output parameters. The input parameters can be arbitrary expressions.

The 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 External Variables (e.g. Real[x$$$,"Subordinate"→*sc*], Integer[i$$[5],"Subordinate"→*sc*], Logical[b$$$,"Subordinate"→*sc*]). The 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"->$\{\{v_1, \frac{\partial exte}{\partial v_1}\}, \{v_2, \frac{\partial exte}{\partial v_2}\}, ...\}$ (see also SMSReal SMSInteger SMSLogical).

option name	*description*	*default value*
"Dependency"−> $\{\{v_1,\frac{\partial exte}{\partial v_1}\},\{v_2,\frac{\partial exte}{\partial v_2}\},\ldots\}$	defines partial derivatives of output parameters with respect to input parameters	{}
"System"−>*truefalse*	the subroutine that is called has been automatically generated as well	True

Options for *SMSCall*.

Example

```
In[175]:=
    << AceGen`;
     SMSInitialize["test", "Language" → {"Fortran", "C", "Mathematica"}[[2]]];
```

This generates subroutine *f* with an input parameter *x* and the output parameters $y = f(x)$ and $dy = \frac{\partial f}{\partial x}$. The triple $$$ in declaration of input parameter *x* indicates that *x* is transferred by value and not by pointer and it only effects C code.

```
In[177]:=
    SMSModule["f", Real[x$$$, y$$, dy$$]];
    x ⊨ SMSReal[x$$$];
    y ⊨ Sin[x];
    dy ⊨ SMSD[y, x];
    SMSExport[y, y$$];
    SMSExport[dy, dy$$];
```

This generates subroutine *main* that calls subroutine *f*.

```
In[183]:=
    SMSModule["main", Real[x$$, r$$]];
    x ⊨ SMSReal[x$$];
    a ⊢ x^2;

    f = SMSCall["f", a, Real[y$$], Real[dy$$]];
    da ⊨ SMSReal[dy$$$, "Subordinate" → f];
```

The "Dependency"->{sin,{a,da}} option defines that output parameter *y* depends on input parameters of external subroutine call *f* and defines partial derivative of *y* with respect to input parameter *a*. By default all partial derivatives of output parameters with respect to input parameters are set to 0.

The triple $$$ here is required because y is defined as local variable of the master subroutine and it only effects C code.

```
In[188]:=
    sina ⊨ SMSReal[y$$$, "Subordinate" → f, "Dependency" -> {a, da}];
```

```
In[189]:=
    dd ⊨ SMSD[sina, x];
    SMSExport[dd, r$$];

    SMSWrite[];

       Method : f 2 formulae, 14 sub-expressions
       Method : main 2 formulae, 29 sub-expressions
       [1] File created : test.c Size : 954
```

In[192]:=

!!test.c

```
/*************************************************************
* AceGen    VERSION                                          *
*           Co. J. Korelc  2006           30.9.2006 17:14    *
**************************************************************
User : USER
Evaluation time                 : 1 s     Mode  : Optimal
Number of formulae              : 4       Method: Forward
Subroutine                      : f size :14
Subroutine                      : main size :29
Total size of Mathematica  code : 43 subexpressions
Total size of C code            : 339 bytes*/
#include "sms.h"

/******************* S U B R O U T I N E *********************/
void f(double v[5001],double x,double (*y),double (*dy))
{
(*y)=sin(x);
(*dy)=cos(x);
};

/******************* S U B R O U T I N E *********************/
void main(double v[5001],double (*x),double (*r))
{
double dy;double y;
f(v,Power((*x),2),&y,&dy);
(*r)=2e0*dy*(*x);
};
```

Smart Assignments

SMSFreeze

SMSFreeze[*exp*]	create data object (*SMSFreezeF*) that represents expression *exp*, however its numerical evaluation yields an unique signature obtained by the perturbation of numerical value of *exp*
SMSFreeze[{exp_1,exp_2,…}]	create list of data objects (*SMSFreezeF*) that represent expressions {exp_1,exp_2,…}

Imposing restrictions on an optimization procedure.

option name	*default value*	
"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]
"Dependency"	False	see below
"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

Options for SMSFreeze.

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*)
$\left\{\left\{p_1,\frac{\partial exp}{\partial p_1}\right\},\left\{p_2,\frac{\partial exp}{\partial p_2}\right\},...\right\}$	assume that *SMSFreezeF* data object depends on given auxiliary variables p_1, $p_2,\ldots$ and define the partial derivatives of *SMSFreezeF* data object with respect to given auxiliary variables $p_1,p_2,\ldots$
$\left\{\{p_1,p_2,\ldots\},\left\{\frac{\partial exp}{\partial p_2},\frac{\partial exp}{\partial p_1},\ldots\right\}\right\}$	$\equiv\left\{\left\{p_1,\frac{\partial exp}{\partial p_1}\right\},\left\{p_2,\frac{\partial exp}{\partial p_2}\right\},...\right\}$ define gradient of *exp* with respect to variables $\{p_1,p_2,\ldots p_N\}$ to be $\left\{\frac{\partial exp}{\partial p_2},\frac{\partial exp}{\partial p_1},\ldots\right\}$

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
$\{p,\{\frac{\partial exp_1}{\partial p},\frac{\partial exp_2}{\partial p},\ldots\}\}$	define partial derivatives of $\{exp_1,exp_2,\ldots\}$ with respect to variable p to be $\{\frac{\partial exp_1}{\partial p},\frac{\partial exp_2}{\partial p},\ldots\}$
$\{\{p_1,p_2,\ldots\}, \{\{\frac{\partial exp_1}{\partial p_1},\frac{\partial exp_1}{\partial p_2},\ldots\},\{\frac{\partial exp_2}{\partial p_1},\frac{\partial exp_2}{\partial p_2},\ldots\},\ldots\}\}$	define Jacobian matrix of the transformation from $\{exp_1,exp_2,\ldots\}$ to $\{p_1,p_2,\ldots\}$ to be matrix $\{\{\frac{\partial exp_1}{\partial p_1},\frac{\partial exp_1}{\partial p_2},\ldots\},\{\frac{\partial exp_2}{\partial p_1},\frac{\partial exp_2}{\partial p_2},\ldots\},\ldots\}$
$\{\{\{p_{11},\frac{\partial exp_1}{\partial p_{11}}\},\{p_{12},\frac{\partial exp_1}{\partial p_{12}}\},\ldots\}, \{\{p_{21},\frac{\partial exp_2}{\partial p_{21}}\},\{p_{22},\frac{\partial exp_2}{\partial p_{22}}\},\ldots\},\ldots\}$	define arbitrary partial derivatives of vector of expressions $\{exp_1,exp_2,\ldots\}$

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

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

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

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

With SMSFreeze[exp, "Dependency" -> $\{\{p_1,\frac{\partial exp}{\partial p_1}\},\{p_2,\frac{\partial exp}{\partial p_2}\},\ldots,\{p_n,\frac{\partial exp}{\partial p_n}\}\}$] the true dependencies of *exp* are ignored and it is assumed that *exp* depends on auxiliary variables $p_1,\ldots,p_n$. 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.*

option name	*default value*	
"IgnoreNumbers"	False	whether to apply *SMSFreeze* functions only on parts of the list that are not numbers (NumberQ[*exp*] yields True)
"KeepStructure"	False	whether to keep the structure of the input expression (sparsity, symmetry and antisymmetry is preserved)
"Variables"	False	whether to apply SMSFreeze function on all auxiliary variables in expression rather than on expression as a whole

Additional options for input expression that is arbitrarily structured list of expressions.

See also: Auxiliary Variables , `SMSInteger` , `SMSReal` , Exceptions in Differentiation .

Here the various options of *SMSFreeze* function are demonstrated.

```
In[712]:=
      << AceGen`;
      SMSInitialize["test"];
      SMSModule["sub", Real[x$$]];
      x ⊨ SMSReal[x$$];
```

Here the original matrix, the result of numerical evaluation of the matrix and optimized matrix are presented.

In[716]:=

$$\text{matrix} = \begin{pmatrix} x & 2x & \text{Cos}[x] \\ 2x & 2 & 0 \\ -\text{Cos}[x] & 0 & 0 \end{pmatrix};$$

```
In[717]:=
      matrix // SMSEvaluate // MatrixForm
```

Out[717]//MatrixForm=

$$\begin{pmatrix} 0.5304699360852496 & 1.060939872170499 & 0.8625694068930539 \\ 1.060939872170499 & 2.000000000000000 & 0 \\ -0.8625694068930539 & 0 & 0 \end{pmatrix}$$

```
In[718]:=
      mr ⊨ matrix;
      mr // MatrixForm
```

Out[719]//MatrixForm=

$$\begin{pmatrix} mr_{11} & mr_{21} & mr_{13} \\ mr_{21} & 2 & 0 \\ -mr_{13} & 0 & 0 \end{pmatrix}$$

Here the elements of the matrix are replaced by the *SMSFreezeF* data objects.

```
In[720]:=
      a = SMSFreeze[matrix];
      a // MatrixForm
```

Out[721]//MatrixForm=

$$\begin{pmatrix} \mathbb{F}\text{reeze}[mr_{11}] & \mathbb{F}\text{reeze}[2\,mr_{11}] & \mathbb{F}\text{reeze}[\text{Cos}[mr_{11}]] \\ \mathbb{F}\text{reeze}[2\,mr_{11}] & \mathbb{F}\text{reeze}[2] & \mathbb{F}\text{reeze}[0] \\ \mathbb{F}\text{reeze}[-\text{Cos}[mr_{11}]] & \mathbb{F}\text{reeze}[0] & \mathbb{F}\text{reeze}[0] \end{pmatrix}$$

Here the new numerical values of matrix are displayed.

```
In[722]:=
      a // SMSEvaluate // MatrixForm
```

Out[722]//MatrixForm=

$$\begin{pmatrix} 0.5261162843211248 & 0.9930915332003850 & 0.8243787060707944 \\ 0.9799062724161880 & 1.995023026311400 & 0.006499404590254645 \\ -0.7941996183163959 & 0.002476043404718058 & 0.004036575708692753 \end{pmatrix}$$

Optimisation procedures can not optimize the matrix, thus new auxiliary variables are generated for each element of the matrix.

```
In[723]:=
    ar ⊨ a;
    ar // MatrixForm
```

Out[724]//MatrixForm=

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

Here the affects of various options on results are presented .

```
In[725]:=
    b = SMSFreeze[matrix, "IgnoreNumbers" -> True];
    b // MatrixForm
    b // SMSEvaluate // MatrixForm
```

Out[726]//MatrixForm=

$$\begin{pmatrix} \mathbb{F}reeze[mr_{11}] & \mathbb{F}reeze[2\,mr_{11}] & \mathbb{F}reeze[Cos[mr_{11}]] \\ \mathbb{F}reeze[2\,mr_{11}] & 2 & 0 \\ \mathbb{F}reeze[-Cos[mr_{11}]] & 0 & 0 \end{pmatrix}$$

Out[727]//MatrixForm=

$$\begin{pmatrix} 0.4919362278509971 & 0.9940983400487229 & 0.8222124988595418 \\ 1.034330904328028 & 2.000000000000000 & 0 \\ -0.8460541623782876 & 0 & 0 \end{pmatrix}$$

```
In[728]:=
    br ⊨ b;
    br // MatrixForm
```

Out[729]//MatrixForm=

$$\begin{pmatrix} br_{11} & br_{12} & br_{13} \\ br_{21} & 2 & 0 \\ br_{31} & 0 & 0 \end{pmatrix}$$

```
In[730]:=
    c = SMSFreeze[matrix, "KeepStructure" -> True];
    c // MatrixForm
    c // SMSEvaluate // MatrixForm
```

Out[731]//MatrixForm=

$$\begin{pmatrix} \mathbb{F}reeze[mr_{11}] & \mathbb{F}reeze[2\,mr_{11}] & \mathbb{F}reeze[Cos[mr_{11}]] \\ \mathbb{F}reeze[2\,mr_{11}] & 2 & 0 \\ -\mathbb{F}reeze[Cos[mr_{11}]] & 0 & 0 \end{pmatrix}$$

Out[732]//MatrixForm=

$$\begin{pmatrix} 0.5254870667632555 & 1.053916845282593 & 0.8055348146768811 \\ 1.053916845282593 & 2.000000000000000 & 0 \\ -0.8055348146768811 & 0 & 0 \end{pmatrix}$$

```
In[733]:=
    cr ⊨ c;
    cr // MatrixForm
```

Out[734]//MatrixForm=

$$\begin{pmatrix} cr_{11} & cr_{21} & cr_{13} \\ cr_{21} & 2 & 0 \\ -cr_{13} & 0 & 0 \end{pmatrix}$$

```
In[735]:=
      d = SMSFreeze[matrix, "Variables" -> True];
      d // MatrixForm
```

Out[736]//MatrixForm=

$$\begin{pmatrix} \mathbb{F}\text{reeze}[mr_{11}] & 2\,\mathbb{F}\text{reeze}[mr_{11}] & \text{Cos}[\mathbb{F}\text{reeze}[mr_{11}]] \\ 2\,\mathbb{F}\text{reeze}[mr_{11}] & 2 & 0 \\ -\text{Cos}[\mathbb{F}\text{reeze}[mr_{11}]] & 0 & 0 \end{pmatrix}$$

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 `unique signature` 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 $f(n) = \sum_{n=1}^{m} \frac{\partial g(x)}{\partial x}\Big|_{x=0}$, where $g(x)$ is arbitrary expression (can be large expression involving *If* and *Do* structures). Not that 0 cannot be assigned to x before the differentation.

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

In[11]:= SMSWrite[];
```

Method : **sub** 3 formulae, 15 sub-expressions

[0] File created : **test.c** Size : 804

In[12]:= !!test.c

```
/*************************************************************
* AceGen    VERSION                                          *
*           Co. J. Korelc  2006           29.8.2006 18:49    *
*************************************************************
User : Korelc
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

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).

See also: Symbolic Evaluation .

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: the variable that should be replaced does not exist

The ⊧ command creates variables accordingly to the set of rules. Here the expression *y*⊧-*x* did not create a new variable *y* resulting in wrong replacement.

In[749]:=

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

Out[755]=

z

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

In[756]:=

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

Out[762]=

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

Example 2: repeated use of SMSReplaceAll

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

In[763]:=

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

Better formulation.

```
In[771]:=
    << AceGen`;
    SMSInitialize["test"];
    SMSModule["sub", Real[x$$]];
    x ⊨ SMSReal[x$$];
    y ⊨ Sin[x];
    z ⊣ Cos[x];
    {y0, z0} ⊨ SMSReplaceAll[{y, z}, x → 0];
```

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

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 fixed length array of expressions {*exp1*, *exp2*,...}
SMSArray[*len*]	create an *SMSArrayF* data object that represents variable length real typ 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 variable length 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* variable length 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 *n* *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.

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 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 index of an arbitrary element becomes known. Consequently, *AceGen* assigns the 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 .

```
In[813]:=
    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 : 721
```

In[820]:=

```
!!test.m
```

```
(*************************************************************
* AceGen    VERSION                                         *
*           Co. J. Korelc  2006            21.8.2006 12:5   *
**************************************************************
User : Korelc
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[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 variable length array data obj

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 .

In[803]:=

```
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 : 913

In[811]:=
!!test.c

```
/*************************************************************
* AceGen    VERSION                                          *
*           Co. J. Korelc  2006           21.8.2006 12:5     *
**************************************************************
User : Korelc
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

SMSD[*exp*,*v*]	partial derivative $\frac{\partial exp}{\partial v}$
SMSD[*exp*,{*v1*,*v2*,...}]	gradient of *exp* $\{\frac{\partial exp}{\partial v_1},\frac{\partial exp}{\partial v_2},...\}$
SMSD[{*exp1*,*exp2*,...},{*v1*,*v2*,...}]	the Jacobian matrix $[\frac{\partial \mathrm{exp_i}}{\partial \mathrm{v_j}}]$
SMSD[*exp*,{{v_{11},v_{12},…},{v_{21},v_{22},…},…}]	differentiation of scalar with respect to matrix $[\frac{\partial \mathrm{exp}}{\partial \mathrm{v_{ij}}}]$
SMSD[*exp*,{*v1*,*v2*,...}, *index*]	create a characteristic expression for an arbitrary element of the gradient $\{\frac{\partial exp}{\partial v_1},\frac{\partial exp}{\partial v_2},...\}$ and return an index data object that represents characteristic element of the gradient with the index *index*
SMSD[*exp*, *array*, *index*]	create a characteristic expression for an arbitrary element of the gradient $\{\frac{\partial exp}{\partial \mathrm{array}}\}$ and return an index data object that represents characteristic element of the gradient with the index *index*

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"−>*admode*	" 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
"Implicit"−>{…,{v,z,$\frac{\partial v}{\partial z}$},...}	{}	during differentiation assume that derivative of auxiliary variable *v* with respect to auxiliary variable *z* is $\frac{\partial v}{\partial z}$
"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)
"Symmetric"−>*truefalse*	False	see example below
"IgnoreNumbers"−>*truefalse*	False	see example below

Options for SMSD.

The derivatives are evaluated by the automatic differentiation technique (see Automatic Differentiation). 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.

Sometimes differentiation with respect to auxiliary variables can lead to incorrect results due to the interaction of automatic differentiation and `expression optimization` (see Automatic Differentiation). In order to prevent this, all the basic independent variables need to have `unique signature`. Functions such as *SMSFreeze*, *SMSReal*, and *SMSFictive* return auxiliary variable with the unique signature.

See also: Automatic Differentiation .

Example: Differentiation with respect to 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(V)* depends on a symmetric matrix of independent variables $V⊨\begin{pmatrix} v_{11} & v_{12} & \ldots \\ v_{12} & v_{22} & \ldots \\ \ldots & \ldots & \ldots \end{pmatrix}$ then we have to possibilities to make proper differentiation:

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

V𝔽⊨SMSFreeze[V];

*δ*exp⊨SMSD[exp(V𝔽),V𝔽};

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

$$\delta\text{exp}⊨ \text{SMSD[exp(V),V,"Symmetric"->True]} \equiv \begin{pmatrix} 1 & \frac{1}{2} & \ldots \\ \frac{1}{2} & 1 & \ldots \\ \ldots & \ldots & \ldots \end{pmatrix} * \begin{pmatrix} \frac{\partial exp}{\partial v_{11}} & \frac{\partial exp}{\partial v_{12}} & \ldots \\ \frac{\partial exp}{\partial v_{12}} & \frac{\partial exp}{\partial v_{22}} & \ldots \\ \ldots & \ldots & \ldots \end{pmatrix}$$

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

$$\text{SMSD}[exp, \begin{pmatrix} v_{11} & v_{12} & \ldots \\ v_{21} & 0 & \ldots \\ \ldots & \ldots & \ldots \end{pmatrix}, \text{"IgnoreNumbers"->True}] \equiv \begin{pmatrix} \frac{\partial exp}{\partial v_{11}} & \frac{\partial exp}{\partial v_{12}} & \ldots \\ \frac{\partial exp}{\partial v_{12}} & 0 & \ldots \\ \ldots & \ldots & \ldots \end{pmatrix}$$

```
In[53]:= << AceGen`;
         SMSInitialize["test"];
         SMSModule["test", Real[a$$, b$$, c$$]];
         {a, b, c} ⊨ SMSReal[{a$$, b$$, c$$}];

In[57]:= x ⊨ {{a, b}, {b, c}};
         f = Det[x];
```

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

```
In[59]:= SMSD[f, x] // MatrixForm
```

Out[59]//MatrixForm=

$$\begin{pmatrix} x_{22} & -2\,x_{21} \\ -2\,x_{21} & x_{11} \end{pmatrix}$$

Various ways how the correct result can be obtained.

```
In[60]:= SMSD[f, x, "Symmetric" → True] // MatrixForm
```

Out[60]//MatrixForm=

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

```
In[61]:= x ⊨ SMSFreeze[x];
         f = Det[x];
         SMSD[f, x] // MatrixForm
```

Out[63]//MatrixForm=

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

Example: Incorrect structure of the program

Differentiation cannot start inside the "If" construct if the variables involved have multiple instances defined on 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[];
```

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[$\{v_1, v_2,\ldots,v_M\}$, $\{z_1,z_2,\ldots,z_N\}$, 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 and SMSReal function or the "Implicit" option of the SMSD function should be used instead whenever possible.

See also: 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*]	start the TRUE branch of the *if* .. *else* .. *endif* construct
SMSElse[]	start the FALSE branch of the *if* .. *else* .. *endif* construct
SMSEndIf[]	end the *if* .. *else* .. *endif* construct
SMSEndIf[*out_var*]	end 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*]	create 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*]	create 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 "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* is a symbol with the value which has to be multi-valued auxiliary variable or a list or collection of lists of symbols.

The *condition* of the "If" construct is a logical statement. 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.

See also: `Auxiliary Variables` , `Signatures of the Expressions`

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.

Example 1: Generic example

Generation of the Fortran subroutine which evaluates the following function

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

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

```
In[821]:=
      << AceGen`;
      SMSInitialize["test", "Language" -> "Fortran"];
      SMSModule["test", Real[x$$, f$$]];
      x ⊨ SMSReal[x$$];
```

Description of the "If" construct.

```
In[825]:=
      SMSIf[x <= 0];
       f ⫤ x^2;
      SMSElse[];
       f ⊣ Sin[x];
      SMSEndIf[f];
```

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

```
In[830]:=
      SMSExport[f, f$$];
      SMSWrite["test"];
```

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

[0] File created : test.f Size : 863
```

This displays the contents of the generated file.

```
In[832]:=
      !!test.f
```

```
!**************************************************************
!* AceGen    VERSION                                          *
!*           Co. J. Korelc  2006            21.8.2006 12:16   *
!**************************************************************
! User : Korelc
! 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 use of the "If" structure

Generation of the Fortran subroutine which evaluates the following function

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

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.

```
In[833]:=
    << 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 auxiliary
          variables in expression are defined out
          of the scope of the current position.
        Module: test Description: Error in user input parameters for function:
          SMSExport
        Input parameter: 2f Current scope:
        Misplaced variables :
         2f ≡ $V[3, 2] Scope: If-False[x ≤ 0]
        Events: 0
        See also: Auxiliary Variables  Troubleshooting

    SMC::Fatal :
     System cannot proceed with the evaluation due to the fatal error in SMSExport .
Out[842]=
    $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 3: Unnecessary dependencies

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

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

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.

In[843]:=
```
<< 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 : 933

Out[856]=
0.441

In[857]:=
```
!! test.c
```

```
/*************************************************************
* AceGen    VERSION                                          *
*            Co. J. Korelc  2006            21.8.2006 12:17   *
*************************************************************
User : Korelc
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];
};
```

```
In[858]:=
    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 : 865

Out[871]=
    0.431

In[872]:=
    !! test.c
```

```c
/*************************************************************
* AceGen    VERSION                                          *
*           Co. J. Korelc  2006            21.8.2006 12:17   *
**************************************************************
User : Korelc
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 .

SMSDo

SMSDo[*v*, *imin*, *imax*]	start the "Do" loop with an auxiliary variable *v* successively taken on the values *imin* through *imax* (in steps of 1)
SMSDo[*v*, *imin*, *imax*, *step*]	start the "Do" loop with an auxiliary variable *v* successively taken on the values *imin* through *imax* in steps of *step*
SMSDo[*v*, *imin*, *imax*, *step*, *init_var*]	start the "Do" loop with an auxiliary variable *v* successively taken on the values *imin* through *imax* in steps of *step* and create fictive instances of the *init _var* auxiliary variables
SMSEndDo[]	end the loop
SMSEndDo[*out_var*]	end the loop and create fictive instances of the *out_var* auxiliary variables

Syntax of the loop construct.

New instances are assigned to the *init_var/out_var* auxiliary variables. The *init_var/out_var* is a symbol with the value which has to be an multi-valued auxiliary variable or a list or collection of lists of symbols. The iteration variable of the "Do" loop is an integer type auxiliary variable *v*. Variable *v* is generated automatically. The values of the *v* are real numbers between 0 and 1. The reason for this is to prevent incorrect simplifications that might arise as a consequence of using the integer random values. The *SMSDo* command returns new instances of the *init_var* auxiliary variables or empty list. The *SMSEndDo* command returns new instances of the *out_var* variables or empty list. The new instances have to be created for all auxiliary variables, that are imported from the outside of the loop and their values are changed inside the loop. The same is valid for variables that are used after the loop and their values have been changed inside the loop.

See also: Auxiliary Variables .

Example 1: Generic example

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

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

```
In[873]:=
      << AceGen`;
      SMSInitialize["test", "Language" -> "Fortran"];
      SMSModule["test", Real[x$$, f$$], Integer[n$$]];
      x ⊨ SMSReal[x$$]; n ⊨ SMSInteger[n$$];
```

Description of the loop.

```
In[877]:=
      f ⊣ 1;
      SMSDo[i, 1, n, 1, f];
       f ⊣ f + x^i;
      SMSEndDo[f];
```

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

In[881]:=

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

```
Method : test 4 formulae, 23 sub-expressions
[0] File created : test.f Size : 869
```

This displays the contents of the generated file.

In[883]:=

```
!!test.f
```

```
!**************************************************************
!* AceGen    VERSION                                          *
!*           Co. J. Korelc  2006            21.8.2006 12:19   *
!**************************************************************
! User : Korelc
! 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: 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 value of the S variable is not random at the beginning of the loop, *AceGen* makes wrong simplification and the resulting code is incorrect.

In[904]:=

```
<< AceGen`;
SMSInitialize["test", "Language" -> "Fortran"];
SMSModule["test", Real[S$$], Integer[n$$]];
n ⊧ SMSInteger[n$$];
S ⫤ 0;
SMSDo[i, 1, n];
 S ⫣ Cos[S];
SMSEndDo[S];
SMSExport[S, S$$];
SMSWrite["test"];
```

Method : **test** 4 formulae, 12 sub-expressions

[0] File created : **test.f** Size : 858

In[914]:=

```
!!test.f
```

```
!**************************************************************
!* AceGen    VERSION                                          *
!*           Co. J. Korelc  2006            21.8.2006 12:20   *
!**************************************************************
! User : Korelc
! 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 new random value to the *S* auxiliary variable prevents wrong simplification and leads to the correct code.

In[915]:=

```
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"];
```

Method : **test** 4 formulae, 15 sub-expressions

[0] File created : **test.f** Size : 865

In[924]:=

```
!!test.f
```

```
!**************************************************************
!* AceGen    VERSION                                          *
!*           Co. J. Korelc  2006            21.8.2006 12:20   *
!**************************************************************
! User : Korelc
! 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 3: How to use variables defined inside the loop outside the loop?

Only the multi-valued variables (introduced by the ⫤ or ⊣ command) can be used outside the loop. The use of the single-valued variables (introduced by the ⊨ or ⊢ command) that are defined within loop outside the loop will result in Variables out of scope error.

Here the variable *X* is defined within the loop and used outside the loop.

Incorrect formulation

```
In[925]:=
<< 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];
Y ⊨ X^2;
```

```
Some of the auxiliary
  variables in expression are defined out
  of the scope of the current position.
Module: test Description: Error in user input parameters for function: SMSR
Input parameter: X^2 Current scope:
Misplaced variables :
 X ≡ $V[4, 1] Scope: Do[i, 1, n, 1]
Events: 0
See also: Auxiliary Variables  Troubleshooting

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

Out[934]=
$Aborted
```

Correct formulation

```
In[935]:=
<< 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, SMSBreak, SMSContinue

SMSReturn[]	≡ SMSVerbatim["C"−>"return;" , "Fortran"−>"return", "*Mathematica*"−>"Return[Null,Module];"] (see *Mathematica* command Return)
SMSBreak[]	≡ SMSVerbatim["C"−>"break;" , "Fortran"−>"exit", "*Mathematica*"−>"Break[];"] (see *Mathematica* command Break)
SMSContinue[]	≡ SMSVerbatim["C"−>"continue;" , "Fortran"−>"cycle", "*Mathematica*"−>"Continue[];"] (see *Mathematica* command Continue)

Utilities

Debugging

SMSSetBreak

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*.

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: `User Interface`, `Interactive Debugging`, `SMSAnalyze` , `AceFEM Structure`

SMSLoadSession

SMSLoadSession[*name*]	reload the data and definitions associated with the *AceGen* session with the session name *name*

In "Debug" mode the system automatically generates file with the name *"sessionname.dbg"* where all the information necessary for the run-time debugging is stored.

SMSAnalyze

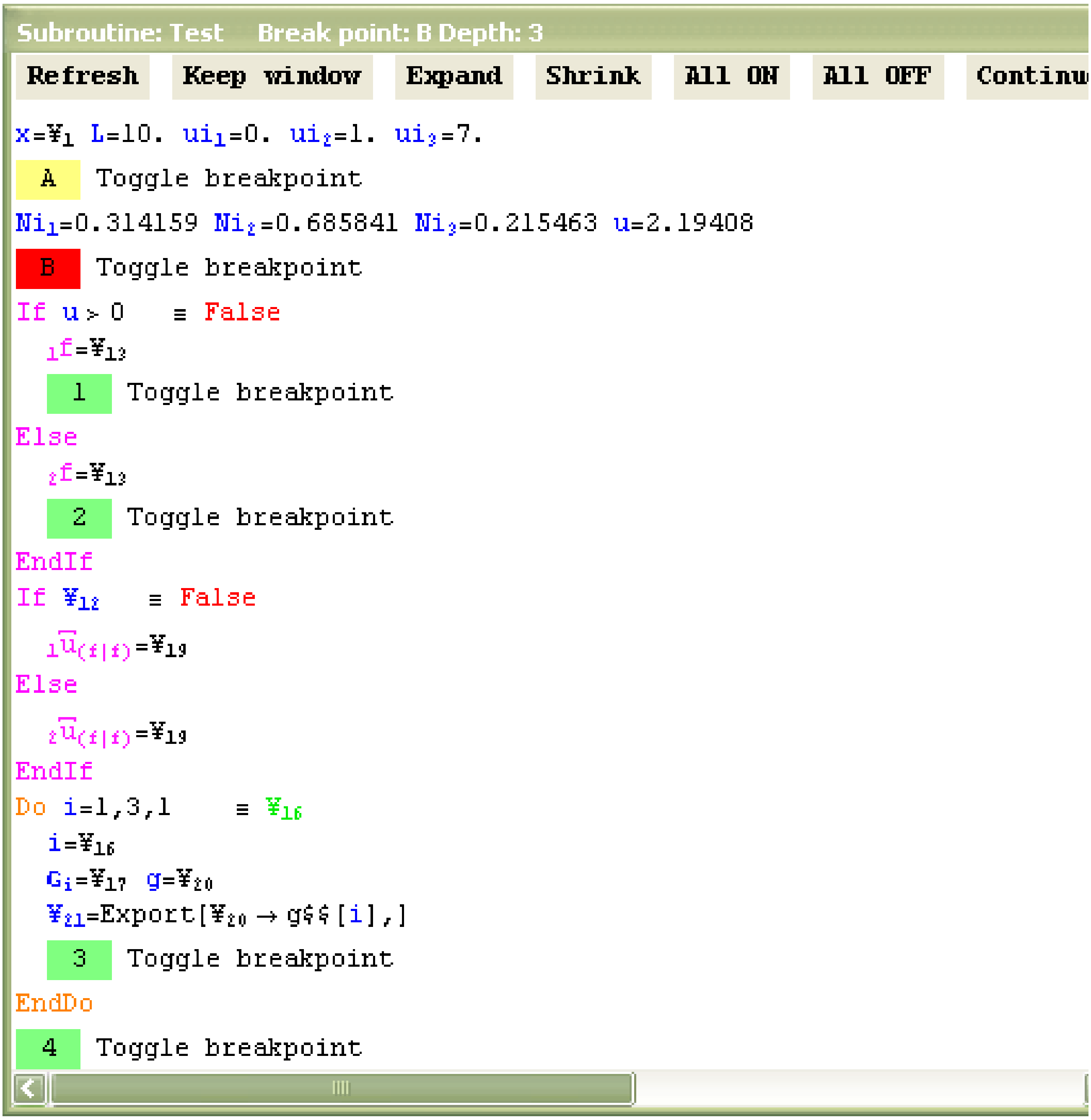
SMSAnalyze[*i_Integer*]	open debug window with the structure of the *i*–*th* generated subroutine
SMSAnalyze[s_*String*]	open debug window with the structure of the generated subroutine with the name *s*
SMSAnalyze[]	open debug windows for all generated subroutines

option name	*default value*	
"Depth"	"Automatic"	"Minimum" ⇒ only auxiliary variables with the names assigned directly by the user are included "Automatic" ⇒ auxiliary variables with the names assigned directly by the user and the control structures are included "Names" ⇒ all auxiliary variables with the names are included "All" ⇒ all auxiliary variables and control structures are included
"Values"	"None"	"None" ⇒ no numeric values are presented "Random" ⇒ signatures associated with the auxiliary variables are included (available only during code generation session) "AceFEM" ⇒ current values of auxiliary variables are included (available only during AceFEM session)

Options for *SMSAnalyze*.

Command opens a separate window where the structure of the program is displayed together with the links to all generated formulae, positions of the break points and the current values of selected auxiliary variables.

See also: `User Interface, SMSLoadSession.`



A ⇒ is the button that represents active user defined break point.

B ⇒ is the button that represents the position in a program where the program has stopped.

1 ⇒ is the button that represents automatically generated inactive break point. The break points are automatically generated at the end of If.. else..endif and Do...enddo structures.

Refresh ⇒ refresh the contents of the debug window.

Keep window ⇒ prevents automatic closing of the debug window

Expand ⇒ increase the extend of the variables that are presented

Shrink ⇒ decrease the extend of the variables that are presented

`All ON` ⇒ enable all breaks points

`All OFF` ⇒ disable all breaks points

`Continue` ⇒ continue to the next break point

SMSClearBreak

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

This command is used to disable break point **at the run time** debugging phase and **not at the code generation** phase. See also: `User Interface`.

SMSActivateBreak

SMSActivateBreak[*breakID*, *opt*]	activate break point with the break identification *breakID* and options *opt*
SMSActivateBreak[*breakID*, *func*]	≡ SMSActivateBreak[i," Function "→ *func*," Window "→False,"Interactive"→False]
SMSActivateBreak[]	enable all break points

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.

This command is used to clear break point **at the run time** debugging phase and **not at the code generation** phase. See also: `User Interface`.

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: `User Interface`, `Interactive Debugging`, `SMSAnalyze`, `AceFEM Structure`

Random Value 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 Optimisation .

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 Optimisation .

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 Optimisation .

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 Optimisation .

SMSMin,SMSMax

SMSMin[*exp1*,*exp2*]	≡ Min[*exp1*,*exp2*]
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: Signatures of the Expressions .

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[i,j]	≡ i^j
SMSPower[i,j," Positive "]	≡ i^j under assumption that i>0

SMSTime

SMSTime[*exp*]	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

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.

See also: Linear Algebra .

SMSLUFactor

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 *SMSLUFactor* performs the factorization of matrix A and returns a new matrix. The matrix generated by the *SMSLUFactor* is a compact way of storing the information contained in the upper and lower triangular matrices of the factorization.

See also: Linear Algebra .

SMSLUSolve

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).

See also: Linear Algebra .

SMSFactorSim

SMSFactorSim[M]	the LU decomposition along with the pivot list of symmetric matrix M

The Gauss elimination procedure is used and simultaneous simplification is performed during the process. The *SMSFactorSim* performs factorization of the matrix A and returns a new matrix. The matrix generated by the *SMSFactorSim* is a compact way of storing the information contained in the upper and lower triangular matrices of the factorization.

See also: Linear Algebra .

SMSInverse

SMSInverse[M]	the inverse of square matrix M

Simultaneous simplification is performed during the process. The Krammer's rule is used and simultaneous simplification is performed during the process. For more than 6 equations is more efficient to use SMSLinearSolve[M,IdentityMatrix[M//Length]] instead.

See also: Linear Algebra .

SMSDet

SMSDet[M]	the determinant of square matrix M

Simultaneous simplification is performed during the process.

See also: Linear Algebra .

SMSKrammer

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

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

See also: Linear Algebra .

Tensor Algebra

SMSCovariantBase

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

In[961]:=

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

Out[965]//MatrixForm=

$$\begin{pmatrix} \mathrm{Cos}[{}_1\phi] & \mathrm{Sin}[{}_1\phi] & 0 \\ -{}_1r\,\mathrm{Sin}[{}_1\phi] & \mathrm{Cos}[{}_1\phi]\,{}_1r & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

SMSCovariantMetric

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

In[966]:=

```
<< AceGen`;
SMSInitialize["test", "Language" -> "Mathematica"];
SMSModule["test"];
{r, ϕ, z} ⊨ Array[SMSFictive[] &, {3}];
SMSCovariantMetric[{r Cos[ϕ], r Sin[ϕ], z}, {r, ϕ, z}] // MatrixForm
```

Out[970]//MatrixForm=

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & {}_1r^2 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

SMSContravariantMetric

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

In[971]:=

```
<< AceGen`;
SMSInitialize["test", "Language" -> "Mathematica"];
SMSModule["test"];
{r, ϕ, z} ⊨ Array[SMSFictive[] &, {3}];
SMSContravariantMetric[{r Cos[ϕ], r Sin[ϕ], z}, {r, ϕ, z}] // MatrixForm
```

Out[975]//MatrixForm=

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{{}_1r^2} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

SMSChristoffell1

SMSChristoffell1[{ϕ_1,ϕ_2,ϕ_3}, {η_1, η_2, η_3}]	the first Christoffell symbol {i,j,k} of transformation from coordinates {η_1, η_2, η_3} to coordinates {ϕ_1, ϕ_2, ϕ_3}

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

Example: Cylindrical coordinates

In[976]:=

```
<< AceGen`;
SMSInitialize["test", "Language" -> "Mathematica"];
SMSModule["test"];
{r, ϕ, z} ⊨ Array[SMSFictive[] &, {3}];
SMSChristoffell1[{r Cos[ϕ], r Sin[ϕ], z}, {r, ϕ, z}] // MatrixForm
```

Out[980]//MatrixForm=

$$\begin{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ {}_1r \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \\ \begin{pmatrix} 0 \\ {}_1r \\ 0 \end{pmatrix} & \begin{pmatrix} -{}_1r \\ 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \\ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \end{pmatrix}$$

SMSChristoffell2

SMSChristoffell1[{ϕ_1,ϕ_2,ϕ_3}, {η_1, η_2, η_3}]	the second Christoffell symbol Γ_{ij}^{k} of transformation from coordinates {η_1, η_2, η_3} to coordinates {ϕ_1, ϕ_2, ϕ_3}

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

Example: Cylindrical coordinates

In[981]:=
```
<< 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
```

Out[985]//MatrixForm=

$$\begin{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ \frac{1}{{}_1r} \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \\ \begin{pmatrix} 0 \\ \frac{1}{{}_1r} \\ 0 \end{pmatrix} & \begin{pmatrix} -{}_1r \\ 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \\ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \end{pmatrix}$$

SMSTensorTransformation

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.

In[986]:=
```
<< AceGen`;
SMSInitialize["test", "Language" -> "Mathematica"];
SMSModule["test"];
{r, ϕ, z} ⊨ Array[SMSFictive[] &, {3}];
SMSTensorTransformation[{r^2, r Sin[ϕ], r z},
 {r Cos[ϕ], r Sin[ϕ], z}, {r, ϕ, z}, {False}]
```

Out[990]=

$$\{\mathrm{Cos}[{}_1\phi]\ {}_1r^2 + {}_1r\ \mathrm{Sin}[{}_1\phi]^2,\ \mathrm{Cos}[{}_1\phi]\ \mathrm{Sin}[{}_1\phi] - {}_1r\ \mathrm{Sin}[{}_1\phi],\ {}_1r\ {}_1z\}$$

SMSDCovariant

SMSDCovariant[*tensor*, *transf*, *coord*, *index_types*]	covariant derivative of arbitrary tensor field *tensor* with indices *index_types* defined in curvilinear coordinates *coord* under transformation *transf*

Transformations *transf* are arbitrary functions while coordinates *coord* have to be proper auxiliary variables with unique signature (see also SMSD). The type of tensor indices is specified by the array *index_types* where *True* means covariant index and *False* contravariant index.

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}.

```
In[991]:=
      << AceGen`;
      SMSInitialize["test", "Language" -> "Mathematica"];
      SMSModule["test"];
      {r, ϕ, z} ⊨ Array[SMSFictive[] &, {3}];
      SMSDCovariant[{r^2, r Sin[ϕ], r z},
        {r Cos[ϕ], r Sin[ϕ], z}, {r, ϕ, z}, {False}] // MatrixForm
```

Out[995]//MatrixForm=

$$\begin{pmatrix} 2\,{}_1r & -{}_1r^2\,\mathrm{Sin}[{}_1\phi] & 0 \\ 2\,\mathrm{Sin}[{}_1\phi] & {}_1r + \mathrm{Cos}[{}_1\phi]\,{}_1r & 0 \\ {}_1z & 0 & {}_1r \end{pmatrix}$$

Mechanics of Solids

SMSLameToHooke, SMSHookeToLame, SMSHookeToBulk, 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 sheer modulus G and bulk modulus κ
SMSBulkToHooke[G,κ]	transform sheer 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!

```
In[4]:=  SMSLameToHooke[λ, μ] // Simplify
```

Out[4]= $\left\{\frac{\mu\,(3\,\lambda+2\,\mu)}{\lambda+\mu}, \frac{\lambda}{2\,(\lambda+\mu)}\right\}$

SMSPlaneStressMatrix, 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!

```
In[5]:= SMSPlaneStressMatrix[e, ν] // MatrixForm
```

Out[5]//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[*matrix*]	create code sequence that calculates the matrix exponent of the third order matrix

All eigenvalues of the matrix have to be real numbers.

option name	*default value*	
"Order"	Infinity	Infinity ⇒ analytical solution _Integer ⇒ order of Taylor series expansion

Options for SMSMatrixExp.

SMSInvariantsI,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

General Numerical Environments

MathLink Environment

SMSInstallMathLink

SMSInstallMathLink[*source*]	compile *source.c* and *source.tm* `MathLink` source files, build the executable program, start the program and install *Mathematica* definitions to call functions in
SMSInstallMathLink[]	create *MathLink* executable from the last generated *AceGen* source code

The SMSInstallMathLink command executes the command line Visual studio C compiler (or MinGW) and linker. For other 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 Mathematica directory ... /AddOns/Applications/AceGen/Include/*MathLink*/).

See also: `SMSInitialize` , `Generation of MathLink code`

option name	*default value*	
"Optimize"	Automatic	use additional compiler optimization
"PauseOnExit"	False	pause before exiting the *MathLink* executable

Options for SMSInstallMathLink.

SMSLinkNoEvaluations

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

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.

Matlab Environment

The AceGen generated M-file functions can be directly imported into Matlab.

See also `Generation of Matlab code`.

Finite Element Environments

FE 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 About AceFEM),

⇒ *FEAP* is the research environment written in FORTRAN (see About FEAP),

⇒ *ELFEN©* is the commercial environment written in FORTRAN (see About ELFEN).

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,

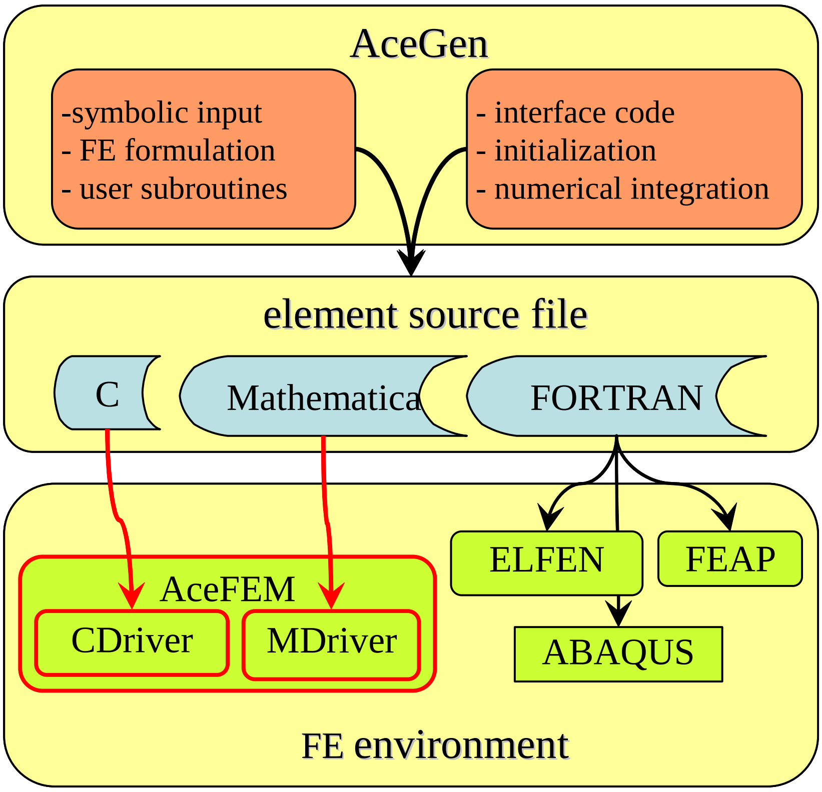
⇒ other researchers are able to repeat the results,

⇒ commercialization of the research is easier,

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

The AceGen system is a growing daily. Please check the www.fgg.uni-lj.si/symech/extensions/ page to see if your environment 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 ../AddOns/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 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 `External Variables`) 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 `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:

$\frac{\partial}{\partial x}\left(k\,\frac{\partial \phi}{\partial x}\right)+\frac{\partial}{\partial y}\left(k\,\frac{\partial \phi}{\partial y}\right)+\frac{\partial}{\partial z}\left(k\,\frac{\partial \phi}{\partial z}\right)+Q=0$ on domain Ω,

$\phi-\overline{\phi}=0$ essential boundary condition on Γ_{ϕ},

$k\,\frac{\partial \phi}{\partial n}-\overline{q}=0$ natural boundary condition on Γ_{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

$\int_{\Omega}[\nabla^T\,\delta\phi\;k\;\nabla\phi-\delta\phi\,Q\,]\,d\Omega-\int_{\Gamma_q}\delta\phi\,\overline{q}\,d\Gamma=0.$

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.

```
In[215]:=
      << AceGen`;
```

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

```
In[216]:=
      SMSInitialize["heatconduction", "Environment" -> "AceFEM"];
```

This initializes constants that are needed for proper symbolic-numeric interface (See Template Constants). Three-dimensional, eight node, hexahedron element with one degree of freedom per node is initialized.

```
In[217]:=
      SMSTemplate["SMSTopology" → "H1", "SMSDOFGlobal" → 1, "SMSSymmetricTangent" → False];
```

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).

```
In[218]:=
      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.

```
In[219]:=
    Xi ⊨ Array[ SMSReal[nd$$[#, "X", 1]] &, 8];
    Yi ⊨ Array[ SMSReal[nd$$[#, "X", 2]] &, 8];
    Zi ⊨ Array[ SMSReal[nd$$[#, "X", 3]] &, 8];
    ϕi ⊨ Array[SMSReal[nd$$[#, "at", 1]] &, 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.

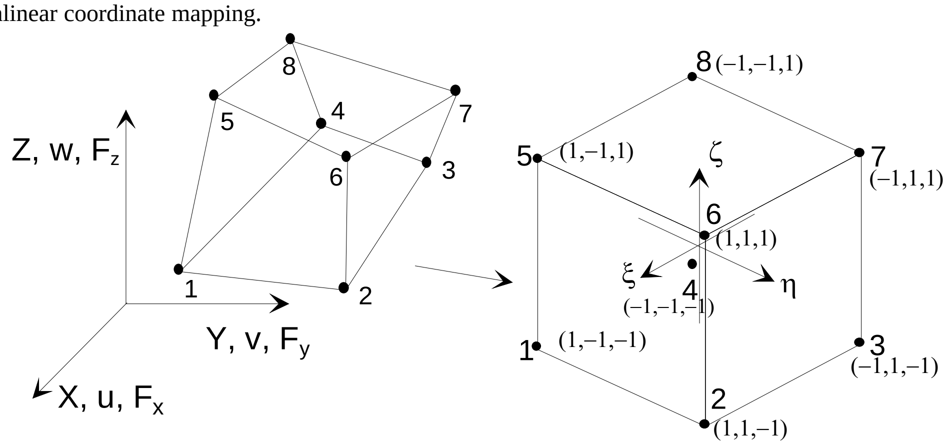
```
In[223]:=
    SMSGroupDataNames = {"Conductivity parameter k0", "Conductivity parameter k1",
       "Conductivity parameter k2", "Heat source"};
    {k0, k1, k2, Q} ⊨
      SMSReal[{es$$["Data", 1], es$$["Data", 2], es$$["Data", 3], es$$["Data", 4]}];
```

Element is numerically integrated by one of the built-in standard numerical integration rules (see Numerical Integration). This starts the loop over the integration points, where ξ, η, ζ are coordinates of the current integration point and *wGauss* is integration point weight.

```
In[225]:=
    SMSDo[IpIndex, 1, SMSInteger[es$$["id", "NoIntPoints"]]];
    {ξ, η, ζ, wGauss} ⊢ Array[SMSReal[es$$["IntPoints", #1, IpIndex]] &, 4];
```

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. The implicit dependencies between the actual and the reference coordinates are given by $\frac{\partial \xi_j}{\partial X_i} = J_m^{-1} \frac{\partial X_i}{\partial \xi_j}$, where J_m is the Jacobean matrix of the nonlinear coordinate mapping.



```
In[227]:=
     Ni ⊨ MapThread[1/8 (1 + ξ #1) (1 + η #2) (1 + ζ #3) &,
        Transpose[{{-1, -1, -1}, {1, -1, -1}, {1, 1, -1}, {-1, 1, -1},
          {-1, -1, 1}, {1, -1, 1}, {1, 1, 1}, {-1, 1, 1}}]];
     X ⊢ SMSFreeze[Ni.Xi];
     Y ⊢ SMSFreeze[Ni.Yi];
     Z ⊢ SMSFreeze[Ni.Zi];
     Jm ⊨ SMSD[{X, Y, Z}, {ξ, η, ζ}];
     SMSDefineDerivative[{ξ, η, ζ}, {X, Y, Z}, SMSInverse[Jm]];
```

The trial function for the temperature distribution within the element is given as linear combination of the shape functions and the nodal temperatures $\phi = N_i.\phi_i$. The ϕ_i are unknown parameters of the variational problem.

```
In[233]:=
     ϕ ⊨ Ni.ϕi;
```

Step 5: Definition of the governing equations

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"]*.

```
In[234]:=
     k ⊨ k0 + k1 ϕ + k2 ϕ^2;
     δϕ ⊨ SMSD[ϕ, ϕi];
     λ ⊨ SMSReal[rdata$$["Multiplier"]];
     Ψi ⊨ Det[Jm] wGauss (k SMSD[δϕ, {X, Y, Z}].SMSD[ϕ, {X, Y, Z}] - δϕ λ Q);
```

Element contribution to global residual vector Ψi is exported into the *p$$* output parameter of the "*Tangent and residual*" subroutine (see SMSStandardModule).

```
In[238]:=
     SMSExport[SMSResidualSign Ψi, 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.

```
In[239]:=
     Kij ⊨ SMSD[Ψi, ϕi];
     SMSExport[Kij, s$$, "AddIn" → True];
```

This is the end of the integration loop.

```
In[241]:=
     SMSEndDo[];
```

Step 7: 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 *heatconduction.c* file with the element source code written in a *C* language.

```
In[242]:=
      SMSWrite[];
```

```
Method : SKR 352 formulae, 5270 sub-expressions
[14] File created : heatconduction.c Size : 17407
```

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 !!!

```
In[71]:= <<AceGen`;
         SMSStandardModule["Tangent and residual"]:=
            SMSModule["RKt",Real[D$$[2],X$$[2,2],U$$[2,2],load$$,K$$[4,4],S$$[2]]];
```

Here the replacement rules are defined that transform standard input/output parameters to user defined input/output parameters.

```
In[73]:= 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.

```
In[74]:= SMSInitialize["userheatconduction", "Environment" -> "User", "Language" -> "C"];
SMSTemplate["SMSTopology" → "H1", "SMSDOFGlobal" → 1, "SMSSymmetricTangent" → False
  , "SMSUserDataRules" → datarules];
SMSStandardModule["Tangent and residual"];
Xi ⊨ Array[ SMSReal[nd$$[#, "X", 1]] &, 8];
Yi ⊨ Array[ SMSReal[nd$$[#, "X", 2]] &, 8];
Zi ⊨ Array[ SMSReal[nd$$[#, "X", 3]] &, 8];
ϕi ⊨ Array[SMSReal[nd$$[#, "at", 1]] &, 8];
SMSGroupDataNames = {"Conductivity parameter k0", "Conductivity parameter k1",
   "Conductivity parameter k2", "Heat source"};
{k0, k1, k2, Q} ⊨
  SMSReal[{es$$["Data", 1], es$$["Data", 2], es$$["Data", 3], es$$["Data", 4]}];
SMSDo[IpIndex, 1, SMSInteger[es$$["id", "NoIntPoints"]]];
 {ξ, η, ζ, wGauss} ⊢ Array[SMSReal[es$$["IntPoints", #1, IpIndex]] &, 4];
 Ni ⊨ MapThread[ 1 / 8 (1 + ξ #1) (1 + η #2) (1 + ζ #3) &,
   Transpose[{{-1, -1, -1}, {1, -1, -1}, {1, 1, -1}, {-1, 1, -1},
     {-1, -1, 1}, {1, -1, 1}, {1, 1, 1}, {-1, 1, 1}}]];
 X ⊢ SMSFreeze[Ni.Xi];
 Y ⊢ SMSFreeze[Ni.Yi];
 Z ⊢ SMSFreeze[Ni.Zi];
 Jm ⊨ SMSD[{X, Y, Z}, {ξ, η, ζ}];
 SMSDefineDerivative[{ξ, η, ζ}, {X, Y, Z}, SMSInverse[Jm]];
 ϕ ⊨ Ni.ϕi;
 k ⊨ k0 + k1 ϕ + k2 ϕ^2;
 λ ⊨ SMSReal[rdata$$["Multiplier"]];
 SMSDo[i, 1, SMSNoAllDOF];
  δϕ ⊨ SMSD[ϕ, ϕi, i];
  Ψi ⊨ Det[Jm] wGauss (k SMSD[δϕ, {X, Y, Z}].SMSD[ϕ, {X, Y, Z}] - δϕ λ Q);
  SMSExport[SMSResidualSign Ψi, p$$[i], "AddIn" → True];
  SMSDo[j, 1, SMSNoAllDOF];
   Kij ⊨ SMSD[Ψi, ϕi, j];
   SMSExport[Kij, s$$[i, j], "AddIn" → True];
  SMSEndDo[];
 SMSEndDo[];
SMSEndDo[];
SMSWrite[];

  Method : RKt 135 formulae, 2609 sub-expressions

  [6] File created : heatconduction.c Size : 6781
```

Reference Guide

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 `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
SMSStandardModule[*code*,*name*]	start the definition of the user subroutine with the default arguments and name *name*
SMSStandardModule[*code*,*name*,*arg*]	start the definition of the user subroutine with the name *name*, and the default set of arguments extended by the set of additional arguments *arg*

Methods for the generation of user subroutines.

codes for the user defined subroutines	*description*	*default 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	"SPP"
"Sensitivity pseudo–load"	standard subroutine that returns the sensitivity pseudo–load vector for the current sensitivity parameter	"SSE"
"Dependent sensitivity"	standard subroutine that resolves sensitivities of the dependent variables defined at the element level	"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	"PAN"
"User n"	*n*–th user defined subroutine	"Usern"

Standard set of user subroutines.

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 (see also the *AceGen* manual for the `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)
nd$$[1,...], nd$$[2,…], …,nd$$[*SMSNoNodes*,…]	nodal data structures for all element nodes (see Nodal Data)
idata$$	integer type environment variables (see Environment Data)
rdata$$	real type environment variables (see 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]	$\{\{x_1^t,y_1^t,z_1^t,x_1^p,y_1^p,z_1^p\},\{x_2^t,y_2^t...\},...\}$
"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 Sensitivity Input Data).

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*
"Tangent and residual"	●	●	
"Postprocessing"	●	●	
"Sensitivity pseudo−load"	●	●	●
"Dependent sensitivity"	●	●	●
"Tangent"			●
"Residuum"			●
"User n"	●		

This creates the element source with the environment dependent supplementary routines and the user defined subroutine "Tangent and residual". The code is created for the 2D, quadrilateral element with 4 nodes, 5 degrees of freedom per node and two material constants. Just to illustrate the procedure the *X* coordinate of the first element node is exported as the first element of the element residual vector p$$. The element is generated for *AceFEM* and *FEAP* environments. The *AceGen* input and the generated codes are presented.

```
In[2]:= << 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 : 3548
```

```
In[8]:= !!test.c
```

```
/*************************************************************
* AceGen    VERSION                                          *
*           Co. J. Korelc  2006            27.10.2006 18:56  *
**************************************************************
User : USER
Evaluation time                 : 0 s     Mode  : Optimal
Number of formulae              : 1       Method: Automatic
Subroutine                      : SKR size :9
Total size of Mathematica  code : 9 subexpressions
Total size of C code            : 225 bytes*/
#include "sms.h"
void SKR(double v[5001],ElementSpec *es,ElementData *ed,NodeSpec **ns,NodeData
**nd,double *rdata,int *idata,double *p,double **s);
FILE *SMTFile;

DLLEXPORT int SMTSetElSpec(ElementSpec *es,int *idata,int ic,int ng)
{ int intc,nd,i;
  static int pn[6]={1, 2, 3, 4, 0, 0};
  static int dof[4]={5, 5, 5, 5};
  static int nsto[4]={0, 0, 0, 0};

  static int ndat[4];
  static char *nid[]={"D","D","D","D"};
  static char *gdcs[]={"Constant 1","Constant 2"};
  static char *gpcs[]={""};
  static char *npcs[]={""};
  static char *sname[]={""};
  static char *idname[]={""};
  static int idindex[1];
  static char *rdname[]={""};
  static char *cswitch[]={""};
  static int iswitch[1]={0};
  static double dswitch[1]={0e0};
  static char *MMAfunc[]={""};
  static char *MMAdesc[]={""};
  static int rdindex[1];
  static int nspecs[4];
  static double pnweights[4]={1e0,1e0,1e0,1e0};
  static double rnodes[12]={-1e0,-1e0,0e0,1e0,-1e0,0e0,
  1e0,1e0,0e0,-1e0,1e0,0e0};
  es->ReferenceNodes=rnodes;
  es->id.NoGroupData=2;
  es->Code="test";es->MainTitle="";es->ProblemType="SLU";
```

```
  es->SubTitle="";
  es->SubSubTitle="";
  es->Bibliography="";

es->id.NoDimensions=2;es->id.NoDOFGlobal=20;es->id.NoDOFCondense=0;es->id.NoNod
es=4;
  es->id.NoSegmentPoints=5;es->Segments=pn;es->PostNodeWeights=pnweights;
  es->id.NoIntSwitch=0;es->IntSwitch=iswitch;
  es->id.NoDoubleSwitch=0;es->DoubleSwitch=dswitch;
  es->id.NoCharSwitch=0;es->CharSwitch=cswitch;
  es->DOFGlobal=dof;es->NodeID=nid;es->id.NoGPostData=0;es->id.NoNPostData=0;

es->id.SymmetricTangent=1;es->id.CreateDummyNodes=0;es->id.PostIterationCall=0;
  es->id.NoMMAFunctions=0;

es->Topology="Q1";es->GroupDataNames=gdcs;es->GPostNames=gpcs;es->NPostNames=np
cs;
  es->MMAFunctions=MMAfunc;es->MMADescriptions=MMAdesc;
  es->AdditionalNodes="{}&";
  es->AdditionalGraphics="{}&";
  es->MMAInitialisation="";
  es->MMANextStep="";
  es->MMAStepBack="";
  es->MMAPreIteration="";

es->IDataNames=idname;es->IDataIndex=idindex;es->RDataNames=rdname;es->RDataInd
ex=rdindex;
  es->id.NoIData=0;es->id.NoRData=0;

es->id.ShapeSensitivity=0;es->id.NoSensNames=0;es->SensitivityNames=sname;es->N
odeSpecs=nspecs;
  es->user.SKR=SKR;;
 if(ng==es->id.NoGroupData){
  es->id.DefaultIntegrationCode=2;
  if(ic==-1){intc=2;} else {intc=ic;};
  es->id.IntCode=intc;
  es->IntPoints=SMTMultiIntPoints(&intc,idata,&es->id.NoIntPoints,
      &es->id.NoIntPointsA,&es->id.NoIntPointsB,&es->id.NoIntPointsC);
  es->id.NoTimeStorage=0;
  es->id.NoElementData=0;
  es->id.NoAdditionalData=0;

nd=es->id.NoDimensions*idata[ID_NoShapeParameters];for(i=0;i<4;i++)ndat[i]=nd;
  es->NoNodeStorage=nsto;es->NoNodeData=ndat;
  return 0;
 }else{
  return 1;
 };
};

/******************* S U B R O U T I N E *********************/
void SKR(double v[5001],ElementSpec *es,ElementData *ed,NodeSpec **ns
     ,NodeData **nd,double *rdata,int *idata,double *p,double **s)
{
p[0]=nd[0]->X[0];
};
```

In[9]:=
```
<< 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 : 5113

```
      subroutine elmt(d,ul,xl,ix,tl,s,p,
     &  ndfe,ndme,nste,isw)
      implicit none
      include 'sms.h'
      integer ix(nen),ndme,ndfe,nste,isw
      double precision  xl(ndfe,nen),d(*),ul(ndfe,nen,*)
      double precision  s(nste,nste),p(nste),tl(nen),sxd(8)
      double precision ul0(ndfe,nen),sg(20),sg0(20)
      character*50 SELEM,datades(2),postdes(0)
      logical DEBUG
      parameter (DEBUG=.false.,
     # SELEM="test")
      integer i,j,jj,ll,ii,k,kk,i1,i2,i3,hlen,icode
      double precision w,v(501),gpost(16,0),npost(4,0)
      integer ipordl(5)
      data (ipordl(i),i=1,5)/1,2,3,4,1/
300   format(i5,20f11.5)
301   format(i5,20f11.5)
1234  format(a4,"[",i3,"]=",f20.10)

      do i=1,ndfe
        do j=1,nen
           ul0(i,j)=ul(i,j,1)-ul(i,j,2)
        enddo
      enddo
      idata(ID_Iteration)=niter+1
      go to(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,
     #      18,19,20,21), isw
c
c.... input record 1 of material properties
1     call dinput(d(1),2)
      write(*,*)  SELEM
      write(iow,*)SELEM
c.....Description of the input data
      datades(1)="Constant 1"
      datades(2)="Constant 2"
      write(iow,"(10x,f15.5,A3,A50)")
     #     (d(i)," = ",datades(i),i=1,2)
c.... number of history variables
      idata(ID_NoSensParameters)=int(d(2))
      if(idata(ID_NoSensParameters).gt.0) then
         call dinput(d(3),idata(ID_NoSensParameters))
         write(iow,*)'Sensitivity parameters:'
         write(iow,"(5(i5,2x,f15.5))")
     #    (i,d(2+i),i=1,idata(ID_NoSensParameters))
      endif
      nsenpa=idata(ID_NoSensParameters)
      mct=0
c.....number of data for TECPLOT
      ntecdata=-3
c.... define node numbering
      inord() = 5
      do ii = 1,5
       ipord(ii,) = ipordl(ii)
      end do
      c.... number of projected quantities
      istv=-7
c.... description of the postprocessing data

      idata(ID_OutputFile)=iow
      return
2     write(*,*)"User switch 2 not implemented"
      return
3     continue
c.... tangent and residuum
      call SKR(v,d,ul,ul0,xl,s,p,hr(nh2),hr(nh1))
      return
4     continue
      goto 8
5     write(*,*)"User switch 5 not implemented"
      return
6     goto 3
7     write(*,*)"User switch 7 not implemented"
      return
c.... postprocessing
8     continue
c.....Description of the post-processing data
      return
9     continue
10    continue
11    continue
12    continue
13    continue
c...... initialize history
14    return
15    continue
16    continue
17    continue
18    continue
19    continue
      write(*,*)"User switch ",isw," not implemented"
      return
c.....sensitivity analysis - external
20    continue
      return
c..... internal sensitivity
21    continue
      return
      End
******************** S U B R O U T I N E ***********
      SUBROUTINE SKR(v,d,ul,ul0,xl,s,p,ht,hp)
      IMPLICIT NONE
      include 'sms.h'
      DOUBLE PRECISION v(501),d(2),ul(5,4),
     & ul0(5,4),xl(2,4),s(20,20),p(20),ht(*),hp(*)
      p(1)=xl(1,1)
      END
```

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 (*SMSTopology*) and the number of nodal degrees of freedom (*SMSDOFGlobal*) 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.

Abbreviation	*Description*	Default value
SMSTopology	element topology (see Element Topology)	?
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	""
SMSNoDimensions	number of spatial dimensions	Automatic
SMSNoNodes	number of nodes	Automatic
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
SMSGroupDataNames	description of the input data values that are common for all elements with the same element specification (e.g material characteristics)	{}
SMSGPostNames	description of the postprocessing quantities defined per material point	{}
SMNPostNames	description of the postprocessing quantities defined per node	{}
SMSNoDOFCondense	number of d.o.f that have to be condensed before the element quantities are assemble (see `Elimination of local unknowns`, `Mixed 3 D Solid FE for AceFEM`)	0
SMSCondensationData	storage scheme for local condensation (see `Elimination of local unknowns`)	
SMSNoTimeStorage	total number of history dependent real type values per element that have to be stored in the memory for transient type of problems	0
SMSNoElementData	total number of arbitrary real values per element	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)	Array[0&, SMSNoNodes]
SMSNoNodeData	total number of arbitrary real values per node (can be different for each node)	Array[idata$$["NoShapeParameters"]* es$$["id", "NoDimensions"]&, SMSNoNodes]

SMSNoDOFGlobal	total number of global d.o.f.	Plus@@SMSDOFGlobal
SMSMaxNoDOFNode	number of d.o.f per node used for dimensioning local arrays	Max[SMSDOFGlobal]
SMSNoAllDOF	number of d.o.f. used for dimensioning local arrays	SMSNoDOFGlobal+ SMSNoDOFCondense
SMSResidualSign	1 ⇒ equations are formed in the form $\boldsymbol{K}\ \boldsymbol{a}+\boldsymbol{\Psi}=\boldsymbol{0}$ −1 ⇒ equations are formed in the form $\boldsymbol{K}\ \boldsymbol{a}=\boldsymbol{\Psi}$	Automatic
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)	Automatic
SMSNodeOrder	ordering of nodes when compared to the standard ordering (used in alternative environments ELFEN, FEAP, etc.)	Automatic
SMSDefaultIntegrationCode	default numerical integration code (see `Numerical Integration`)	Automatic
SMSUserDataRules	user defined replacement rules that transform standard input/output parameters to user defined input/output parameters (see also `User defined environment interface`)	{}

SMSAdditionalNodes	pure function that returns additional nodes in the case of multi−field problems	Hold[{}&]
SMSNodeID	string that is used for identification of the nodes in the case of multi−field problems for all nodes (see Node Identification)	Array["D"&, SMSNoNodes]
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)	Array[1&, SMSNoNodes]
SMSCreateDummyNodes	enable use of dummy nodes	False
SMSAdditionalGraphics	pure function that is called for each element and returns additional graphics primitives per element SMSAdditionalGraphics[{element index, domain index,list of nodes}, True if node marks are required, True if boundary conditions are required, list of node coordinates for all element nodes]	Hold[{}&]
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

SMSMMAInitialisation	*Mathematica*' s code executed after SMTAnalysis command (wrapping the code in Hold prevents evaluation)	Hold[]
SMSMMANextStep	*Mathematica*' s code executed after SMTNextStep command (wrapping the code in Hold prevents evaluation)	Hold[]
SMSMMAStepBack	*Mathematica*' s code executed after SMTStepBack command (wrapping the code in Hold prevents evaluation)	Hold[]
SMSMMAPreIteration	*Mathematica*' s code executed before SMTNextStep command (wrapping the code in Hold prevents evaluation)	Hold[]
SMSIDataNames	list of the names of additional integer type environment data variables (global)	{}
SMSRDataNames	list of the names of additional real type environment data variables (global)	{}
SMSMMAFunctions	list of external function patterns (eg. "adnode[i_,j_]") that are included into the element source code and read from the code at the execution time	{}
SMSMMADescriptions	list of descritions of external functions (the same length as SMSMMAFunctions)	{}
SMSNoAdditionalData	number of additional input data values that are common for all elements with the same element specification (the value can be expression)	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)	{}
SMSCreateDummyNodes	enable use of dummy nodes	False
SMSPostIterationCall	force one additional call of the SKR user subroutines after the convergence of the global solution has been archived in order to improve solution of eventual local equations	False
FEAP$*	FEAP specific template constants described in chapter FEAP (see Specific FEAP Interface Data)	
ELFEN$*	ELFEN specific template constants described in chapter ELFEN (see Specific ELFEN Interface Data)	

Constants defining the general element characteristics .

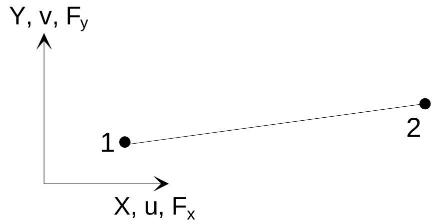
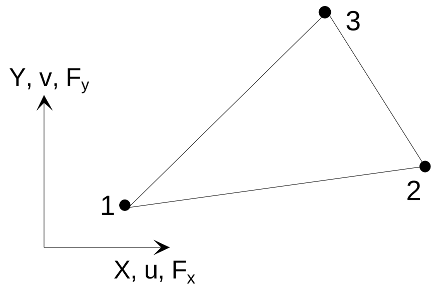
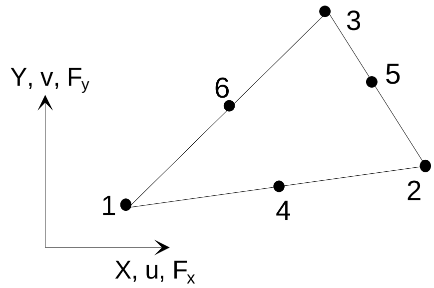
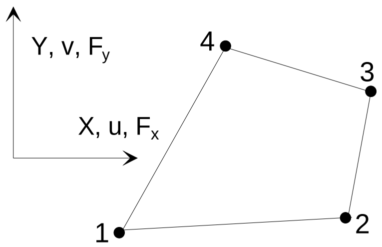
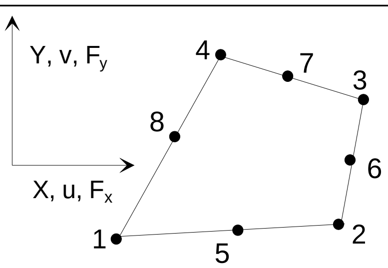
Abbreviation	*Description*	Default value
SMSAddToLibrary	automatically add element to FE library	False
SMSHomePageCode	elements with the same home page code will share the same home page (in principle they should have the same SMSMainTitle and the SMSSubTitle constants)	*element_name*
SMSShortCodes	list of the short codes used also by the element browser	Automatic

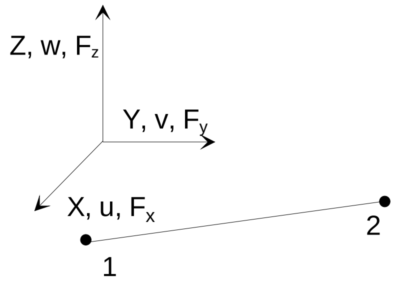
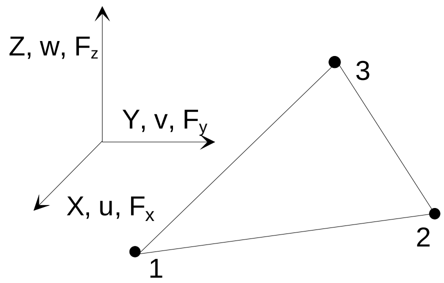
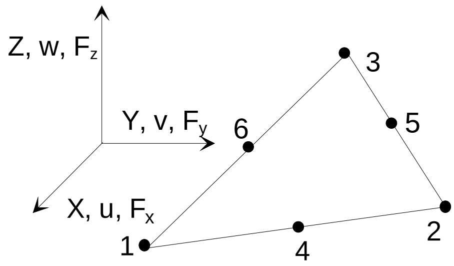
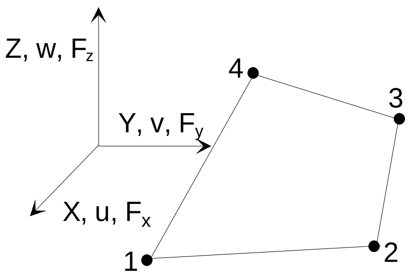
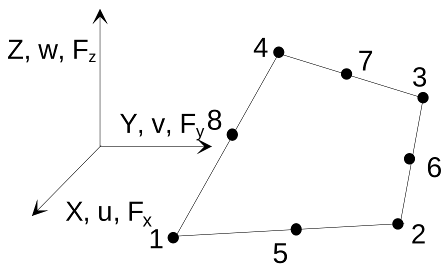
Constants defining creation of the elemnt's home page and library position .

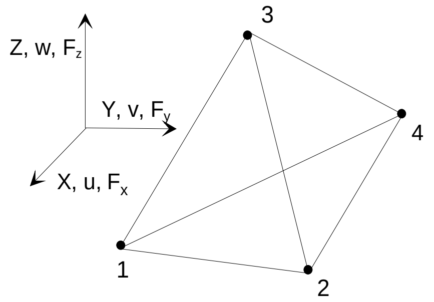
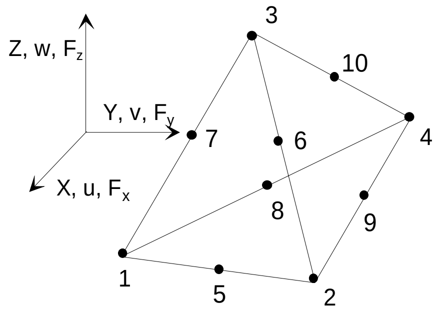
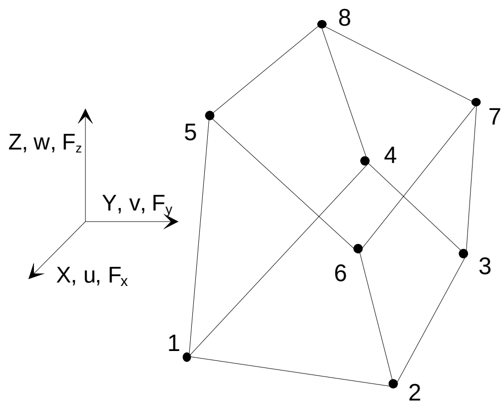
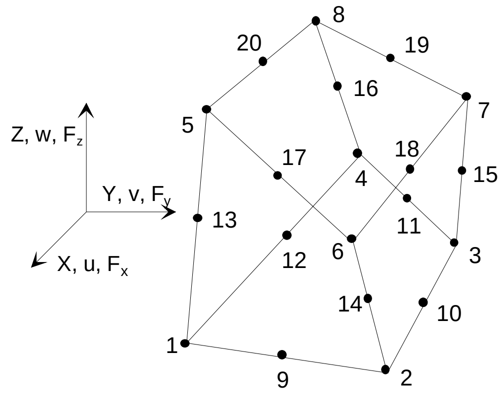
The library of the finite element codes is a part of *AceFEM* package. For details see `FE Library` .

Element Topology

Code	*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

Code	*Description*	*Node numbering*
"L1"	2 D curve with 2 nodes	
"L2"	2 D curve with 3 nodes	1–2–3
"LX"	2 D curve with arbitrary number of nodes	arbitrary
"T1"	2 D Triangle with 3 nodes	
"T2"	2 D Triangle with 6 nodes	
"TX"	2 D Triangle with arbitrary number of nodes	arbitrary
"Q1"	2 D Quadrilateral with 4 nodes	
"Q2"	2 D Quadrilateral with 8 nodes	
"QX"	2 D Quadrilateral with arbitrary number of nodes	arbitrary

Code	*Description*	*Node numbering*
"C1"	3 D curve with 2 nodes	
"C2"	3 D curve with 3 nodes	1–2–3
"CX"	3 D curve with arbitrary number of nodes	arbitrary
"P1"	3 D Triangle with 3 nodes	
"P2"	3 D Triangle with 6 nodes	
"PX"	3 D Triangle with arbitrary number of nodes	arbitrary
"S1"	3 D Quadrilateral with 4 nodes	
"S2"	3 D Quadrilateral with 8 nodes	
"SX"	3 D Quadrilateral with arbitrary number of nodes	arbitrary

<table>
<tr><th>Code</th><th>Description</th><th>Node numbering</th></tr>
<tr><td>"O1"</td><td>3 D Tetrahedron with 4 nodes</td><td>
</td></tr>
<tr><td>"O2"</td><td>3 D Tetrahedron with 10 nodes</td><td>
</td></tr>
<tr><td>"OX"</td><td>3 D Tetrahedron with arbitrary number of nodes</td><td>arbitrary</td></tr>
<tr><td>"H1"</td><td>3 D Hexahedron with 8 nodes</td><td>
</td></tr>
<tr><td>"H2"</td><td>3 D Hexahedron with 20 nodes</td><td>
</td></tr>
<tr><td>"HX"</td><td>3 D Hexahedron with arbitrary number of nodes</td><td>arbitrary</td></tr>
</table>

Default values for the *Topology* constant.

If the element topology is unknown (*SMSTopology="XX"*), then the number of dimensions and the number of nodes have to be specified explicitly. If "default value" is *Automatic*, then the value according to the values of other constants is taken.

The coordinate systems in the figures are only informative (e.g. X,Y can also stand for axisymmetric coordinate system X,Y,ϕ). If the element has one of the standard topologies with the fixed number of nodes described above, then the proper interface for all supported environments is automatically generated. For the nonstandard topology (*"XX"*) or for the variable numbers of nodes (*"LT"*,*"TX"*,…) the proper interface is left to the user.

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. Two or more nodes with the same coordinates and the same node identification are joined together into a single node. 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.. The string type identification is transformed into the integer type identification at run time. Transformation rules are stored in a SMSNodeIDIndex variable.

Node identification	*Description*
"*nid*"	node with node identification "*nid*"
"*nid* –F"	nodes with the switch –F are taken as "fictive" and are ignored by the SMTShowMesh command
"*nid* –C"	nodes with the switch – C are at the beginning by default constrained
"*nid* –D"	nodes with the switch – D or **dummy** nodes are by default constrained and fictive
"*nid* –CF"	any combination of basic switches

Node identifications.

Dummy nodes can only appear as automatically generated additional nodes (see SMSAdditionalNodes). Only one real node is generated for all dummy nodes of particular *nid* type. Dummy node can be changed during the run time into real nodes with the same *nid*.

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. 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 {ξ_i η_i, ζ_i} are the coordinates and w_i is the weight. 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:

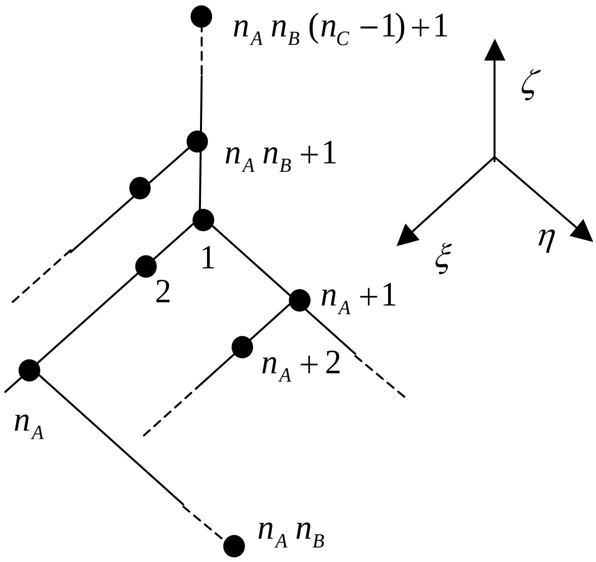
{codeA} ≡ *codeA*

{*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.

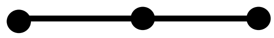
The numbering of the points is depicted below.



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

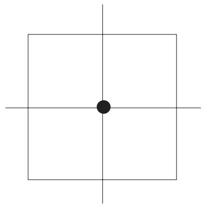
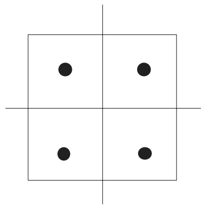
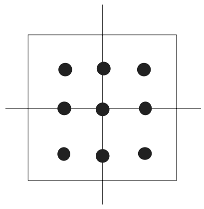
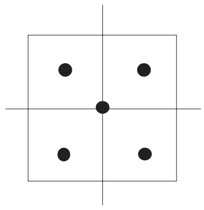
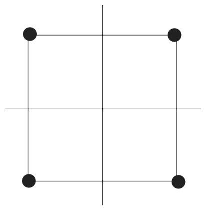
One dimensional

Range: $\{\zeta ,\eta, \zeta\} \in [-1,1] \times [0,0] \times [0,0]$

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	–

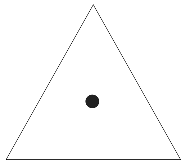
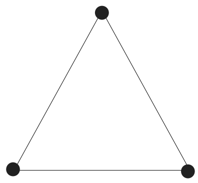
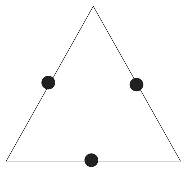
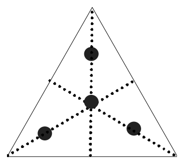
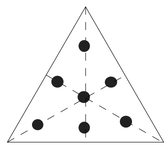
Quadrilateral

$\{\zeta, \eta, \zeta\} \in [-1,1] \times [-1,1] \times [0,0]$

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	

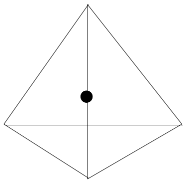
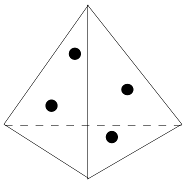
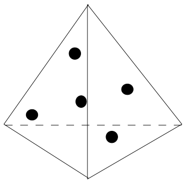
Triangle

$\{\zeta, \eta, \zeta\} \in [0,1] \times [0,1] \times [0,0]$

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	

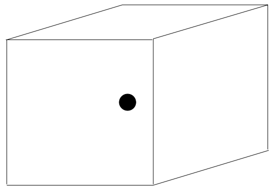
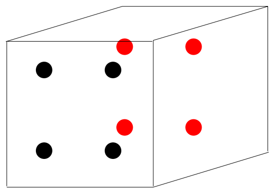
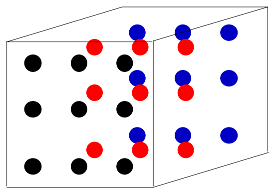
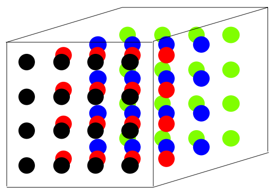
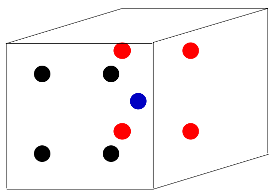
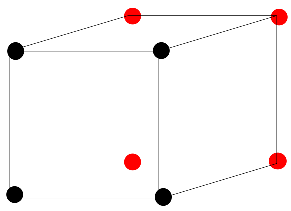
Tetrahedra

$\{\zeta, \eta, \zeta\} \in [0,1] \times [0,1] \times [0,1]$

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

Hexahedra

$\{\zeta ,\eta, \zeta\} \in [-1,1] \times [-1,1] \times [-1,1]$

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	

Example 1

This generates simple loop over all given integration points.

```
SMSDo[IpIndex, 1, SMSInteger[es$$["id", "NoIntPoints"]]];
    {ξ, η, ζ, w} ⊢ Array[SMSReal[es$$["IntPoints", #1, IpIndex]] &, 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:

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

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 build in mechanism ensures automatic condensation of the local tangent matrix before the assembly of the global tangent matrix as follows:

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

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

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

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 has 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$$["ht",1]
$\mathbf{H}_b^n$	SMSCondensationData[[2]]	SMSNoDOFCondense	ed$$["ht", SMSNoDOFCondense+1]
$\mathbf{H}_a^n$	SMSCondensationData[[3]]	SMSNoDOFCondense* SMSNoDOFGlobal	ed$$["ht", 2 SMSNoDOFCondense+1]

Storage scheme for the elimination of the local unknowns.

All three examples below would yield the same storage scheme. See also: `Mixed 3D Solid FE for AceFEM`.

```
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]},
 "SMSNoTimeStorage" → 234]
```

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.

Sse also: `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 material parameter as specified in a description of the Material models
2	shape sensitivity	an index of the current shape parameter
3	implicit sensitivity	it has no meaning for implicit sensitivity

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

One of the input data in the case of shape sensitivity analysis is also the derivation of the nodal coordinates with respect to the shape parameters. The data is by default stored in a data field *nd$$[i,"Data",j,k]* and should be initialized by the user.

This sets a proper dimension for the node data "Data" field. This is also the default value of the *SMSNoNodeData* variable.

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

```
ϕ ⊨ SMSInteger[idata$$["SensIndex"]];
(* index of the current sensitivity parameter*)
ϕt ⊨ SMSInteger[es$$["SensType", ϕ]];
(* type of the parameter 1-material 2-shape 3-implicit*)
ϕti ⊨ SMSInteger[es$$["SensTypeIndex", ϕ]];
(* index of the parameter inside the type group*)
Si ⫤ Array[SMSNoNodes SMSNoDimensions &, 4]; (* sensitivity pseudo-load vector *)

(* material parameters *)
SMSIf[ϕt == 1];
    SMSIf[ϕti == 1]; (* first material parameter *)
        Si ⊣ SMSD[ ..., ϕm1];
    SMSEndIf[];
    SMSIf[ϕti == 2]; (* second material parameter *)
        Si ⊣ SMSD[ ..., ϕm2];
    SMSEndIf[];
     ...
SMSEndIf[Si];

(* shape parameters *)
SMSIf[ϕt == 2];
    (* sensitivity of the coordinates with respect to ϕti-th shape parameter *)
    δxδϕ ⊨ Array[SMSReal[nd$$[#1, "sX", ϕti, #2]] &, {SMSNoNodes, SMSNoDimensions}];
    Si ⊣ ...;
SMSEndIf[Si];

(* implicit dependencies *)
SMSIf[ϕt == 3];
    Si ⊣ ....;
SMSEndIf[Si];

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

Let us suppose that the first shape sensitivity parameter is the X coordinate of the second node and the second Y coordinate of the 50-th node. This sets initial sensitivities of the nodal coordinates for 2D problem in all nodes. The data has to be set before the first sensitivity analysis.

```
SMTNodeData["Data",
  MapIndexed[{If[#2[[1]] == 2, 1, 0], 0, 0, If[#2[[1]] == 50, 1, 0]} &, SMTNodes]];
```

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"*.

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 SMTPost).

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.

The dimension and the contents of the arrays are defined by the two vectors of strings *SMSGPostNames* and *SMSNPost-Names.* They contain the names of the post-processing quantities. Those names are also used in the analysis to identify the specific quantity (see SMTPost). It is the responsibility of the user to keep the names of the post-processing quantities consistent for all used elements.

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[IpIndex, 1, SMSInteger[es$$["id", "NoIntPoints"]]];
  …
  SMSExport[{Sxx, Syy, Sxy, ...}, gpost$$[IpIndex, #1] &];
SMSEndDo[];
…
(* export nodal point postprocessing values for all nodes,
 excluded nodes can be omitted*)
SMSNPostNames = {"DeformedMeshX", "DeformedMeshY", ...};
SMSExport[{{ui[[1]], vi[[1]], ...}, {ui[[2]], vi[[2]], ...}, ...}, npost$$];
```

Data Structures

Environment Data

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 can sometimes fails.

Integer Type Environment 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$$["LastIntCode"]	last integration code for which numerical integration points and weights were calculated	?/R
idata$$["Iteration"]	index of the current iteration within the iterative loop	?/R
idata$$["TotalIteration"]	total number of iterations in session	?/R
idata$$["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)$*\Delta a_{prescribed}$	0/RW
idata$$["ErrorStatus"]	code for the type of the most important error event (see SMTErrorCheck)	0/RW
idata$$["MaterialState"]	number of the "Non−physical material point state" error events detected form the last error check	0/RW
idata$$["NoSensParameters"]	total number of sensitivity parameters (see Subroutine: Sensitivity)	?/R
idata$$["ElementShape"]	number of the "Non−physical element shape" error events detected form the last error check	0/RW
idata$$["SensIndex"]	index of the current sensitivity parameter − globally to the problem (see Subroutine: Sensitivity)	?/R
idata$$["OutputFile"]	output file number or output channel number	?/R

idata$$["MissingSubroutine"]	number of the "Missing user defined subroutine" error events detected form the last error check	0/RW
idata$$["SubDivergence"]	number of the "Divergence of the local sub−iterative process" error events detected form the last error check	0/RW
idata$$["ElementState"]	number of the "Non−physical element state" error events detected form the last error check	0/RW
idata$$["NoNodes"]	total number of nodes	?/R
idata$$["NoElements"]	total number of elements	?/R
idata$$["NoESpec"]	total number of domains	?/R
idata$$["Debug"]	0 ⇒ debug mode is off 1 ⇒ the state of the system is written to output file after each operation	0/RW
idata$$["NoDimensions"]	number of spatial dimensions of the problem (2 or 3)	?/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$$["NoEquations"]	total number of global equations	?/R
idata$$["DiagonalSign"]	number of the "Solver: change of the sign of diagonal" error events detected form the last error check	0/RW
idata$$["Task"]	code of the current task performed	?/R
idata$$["NoSubIterations"]	maximal number of local sub−iterative process iterations performed during the analysis	0/R
idata$$["CurrentElement"]	index of the current element processed	0/R
idata$$["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$$["ExtrapolationType"]	type of extrapolation of integration point values to nodes 0 ⇒ least square extrapolation (Subroutine: Postprocessing) 1⇒ integration point value is multiplied by the user defined weight factors (see Subroutine: Postprocessing)	0/RW
idata$$["TmpContents"]	the meaning of the temporary real type variables stored during the execution of a single analysis into nd$$[i,"tmp", j] data structure 0 ⇒ not used 1 ⇒ residual (reactions) 2 ⇒ used for postprocessing 3 ⇒ initial sensitivity of the nodal coordinates	0
idata$$["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$$["SkipSolver"]	0 ⇒ full Newton−Raphson iteration 1 ⇒ the tangent matrix and the residual vector are assembled but the resulting sistem of equations is not solved	0
idata$$["NoNSpec"]	total number of node specifications	?/R
idata$$["SetSolver"]	1 ⇒ recalculate solver dependent data structures if needed	0
idata$$["NoShapeParameters"]	total number of shape sensitivity parameters	0
idata$$["GeometricTangentMatrix"]	Used for bucklink analysis ($K_0+\lambda\ K_\sigma$) $\{\Psi\}=\{0\}$): 0 ⇒ form full nonlinear matrix 1 ⇒ form K_0 2 ⇒ form K_σ	0
idata$$["DataMemory"]	memory used to store data (bytes)	0
idata$$["SolverMemory"]	memory used by solver (bytes)	0
idata$$["Solver"]	solver identification number	0
idata$$["ErrorElement"]	last element where error event occurred	0
idata$$["SkipTangent"]	1 ⇒ the global tangent matrix is not assemled	0
idata$$["SkipResidual"]	1 ⇒ the global residual vector is not assembled	0
idata$$["SubIterationMode"]	Switch used in the case that alternating solution has been detected by the SMTConvergence function. 0 ⇒ $_{i+1}\mathbf{b}_0^t=\mathbf{b}^p$ ≥1 ⇒ $_{i+1}\mathbf{b}_0^t={}_i\mathbf{b}^t$	0
idata$$["PostIteration"]	is set by the SMTConvergence command to 1 if the switch "PostIteration" has been used	0
idata$$["Solver1"] idata$$["Solver2"] idata$$["Solver3"] idata$$["Solver4"] idata$$["Solver5"]	solver specific parameters	
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	
idata$$["DummyNodes"]	1 ⇒ dummy nodes are supported for the current analysis	0
idata$$["PostIteration"]	1 ⇒ current NR−iteration is a "post−iteration"	0
idata$$["PostIterationCall"]	1 ⇒ additional call of the SKR user subroutines after the convergence of the global solution is enabled	0
idata$$["Step"]	total number of completed solution steps (set by Newton−Raphson iterative procedure)	0
idata$$["DebugElement"]	−1 ⇒ break points (see Interactive Debugging) and control print outs (see SMSPrint) are active for all elements 0 ⇒ break points and control print outs are disabled >0 ⇒ break points and control print outs are active only for the element with the index SMTIData["DebugElement"]	0
idata$$["ZeroPivot"]	index of the equation with the zero pivot (after the decomposition of the global tangent matrix)	0

	(after the decomposition of the global tangent matrix)	
idata$$["GlobalIterationMode"]	Switch used in the case that alternating solution has been detected by the SMTConvergence function. 0 ⇒ no restrictions on global equations ≥1 ⇒ freeze all "If" statements (e.g. nodes in contact, plastic−elastic regime)	0
idata$$["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$$["LineSearchUpdate"]	activate line search procedure (see also idata$$["LineSearchStepLength"])	False

Integer type environment data.

See also Environment Data.

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$$["ResidualError"]	Modified Euklid's norm of the residual vector $\sqrt{\frac{\Psi.\Psi}{\text{NoEquations}}}$	10^{55}
rdata$$["IncrementError"]	Modified Euklid's norm of the last increment of global d.o.f $\sqrt{\frac{\Delta a.\Delta a}{\text{NoEquations}}}$	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.

See also Environment Data.

Node Data Structures

Two types of the node specific data structures are defined. The `node specification data` structure (*ns$$*) defines the major characteristics of the nodes sharing the same node identification (NodeID). Nodal data structure (*nd$$*) 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 `Data Base Manipulations`). 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.

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$$[i,"id", "NoTmpData"]	number of temporary real type variables stored during the execution of a single analysis directive	1
ns$$[i,"id","Constrained"]	1 ⇒ node has initially all d.o.f. constrained	1
ns$$[i,"id","Fictive"]	1 ⇒ node is ignored for the postprocessing of nodes	1
ns$$[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	string

Node specification data structure.

Se also Node Data Structures.

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,"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$$[i,"at", j]	current value of the j–th nodal d.o.f ($\mathbf{a}_i^t$)	NoDOF
nd$$[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,"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,"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$$[i,"tmp", j]	temporary real type variables stored during the execution of a single analysis directive	Max[idata$$["MinNoTmpData"], NoDOF])

Nodal data structure.

Se also Node Data Structures.

For the compatibility with other environments the data stored in the "*tmp*" field should not be addressed directly, but through standard environment independent form. This form is then interpreted by the *AceGen* at the code generation phase.

Default form	*Description*	*AceFEM* interpretation
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", SMSNoDimensions*(j−1)+k]
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]

Interpreted nodal data values.

Element Data Structures

Two types of the element specific data structures are defined. The domain specification data structure 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 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.

Domain Specification Data

Default form	Description	Type
es$$["Code"]	element code according to the general classification	string
es$$["user", *i*]	the *i*–th user defined element subroutines (interpretation depends on the FE environment)	link
es$$["id","SpecIndex"]	global index of the domain specification structure	integer
es$$["id","NoDimensions"]	number of spatial dimensions (1/2/3)	integer
es$$["id","NoDOFGlobal"]	number of global d.o.f per element	integer
es$$["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$$["id","NoNodes"]	number of nodes per element	integer
es$$["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$$["id","NoSegmentPoints"]	the length of the *es$$["Segments"]* field	integer
es$$["id","IntCode"]	integration code according to the general classification (see Numerical Integration)	integer
es$$["id","NoTimeStorage"]	number of transient variables (variable lenght)	integer expression
es$$["id","NoElementData"]	number of arbitrary real values per element (variable lenght)	integer expression
es$$["id","NoIntPoints"]	total number of integration points for numerical integration (see Numerical Integration)	integer
es$$["id","NoGPostData"]	number of post–processing quantities per material point (see SMTElementPostData)	integer
es$$["id","NoNPostData"]	number of post–processing quantities per node (see SMTElementPostData)	integer

Default form	Description	Type
es$$["id", "SymmetricTangent"]	1 ⇒ element tangent matrix is symmetric 0 ⇒ element tangent matrix is unsymmetrical	integer
es$$["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$$["id","NoSensNames"]	number of quantities for which parameter sensitivity pseudo−load code is derived	integer
es$$["id","ShapeSensitivity"]	1 ⇒ shape sensitivity pseudo−load code is present 0 ⇒ shape sensitivity is not enabled	integer
es$$["id", "NoIData"]	number of additional integer type environment data variables	integer
es$$["id", "NoRData"]	number of additional real type environment data variables	integer
es$$["id", "DefaultIntegrationCode"]	default numerical integration code (*see* *Numerical integration*). Value is initialized by template constant SMSDefaultIntegrationCode (see Template Constants).	integer
es$$["id", "NoMMAFunctions"]	number of external function patterns that are included into the source code	0
es$$["id", "NoAdditionalData"]	number of additional input data values that are common for all elements in domain (e.g flow curve points) and are provided by the user is input data (variable lenght)	integer expression
es$$["id","NoCharSwitch"]	number of character type user defined constants	0
es$$["id","NoIntSwitch"]	number of integer type user defined constants	0
es$$["id","NoDoubleSwitch"]	number of double type user defined constants	0
es$$["id", "CreateDummyNodes"]	enable use of dummy nodes	False
es$$["id","PostIterationCall"]	force an additional call of the SKR user subroutines after the convergence of the global solution is achieved	False
es$$["Topology"]	element topology code (see Template Constants)	string
es$$["GroupDataNames", *i*]	description of the *i*−th input data value that is common for all elements with the same specification	NoGroupData strings

es$$["GPostNames", *i*]	description of the *i*−th post−processing quantities evaluated at each material point (see SMTElementPostData)	NoGPostData strings
es$$["NPostNames", *i*]	description of the *i*−th post−processing quantities evaluated at each nodal point (see SMTElementPostData)	NoNPostData strings
es$$["Segments", *i*]	sequence of element node indices that defines the segments on the surface or outline of the element (e.g. for *"Q1"* topology {1,2,3,4,0})	NoSegmentPoints integer numbers
es$$["DOFGlobal", *i*]	number of d.o.f for the *i*−th node (each node can have different number of d.o.f)	NoNodes integer numbers
es$$["SensType", *i*]	type of the *i*−th sensitivity parameter (see Subroutine: Sensitivity)	NoSensParameters integer numbers
es$$["SensTypeIndex", *i*]	index of the *i*−th parameter defined locally in a type group (see Subroutine: Sensitivity)	NoSensParameters integer numbers
es$$["Data",j]	data common for all the elements within a particular domain (fixed length)	NoGroupData real numbers
es$$["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
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$$["AdditionalData",*i*]	additional data common for all the elements within a particular domain (variable length)	NoAdditionalData real numbers
es$$["NoNodeStorage", *i*]	number of history dependent real type values for the *i*−th node	NoNodes integer numbers
es$$["NoNodeData", *i*]	number of arbitrary real values for the *i*−th node	NoNodes integer numbers
es$$["NodeSpec",*i*]	node specification index for the *i*−th node	NoNodes integer numbers
es$$["AdditionalNodes"]	pure 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$$["NodeID",*i*]	integer number that is used for identification of the nodes in the case of multi–field problems for all nodes	NoNodes∗ integer numbers
es$$["AdditionalGraphics"]	pure function that is called for each element and returns additional graphics primitives per element (see SMSAdditionalGraphics)	string
es$$["SensitivityNames",i]	description of the quantities for which parameter sensitivity pseudo–load code is derived	NoSensNames∗ string
es$$["MainTitle"]	description of the element	string
es$$["SubTitle"]	description of the element	string
es$$["SubSubTitle"]	detailed description of the element	string
es$$["Bibliography"]	reference	string
es$$["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$$["MMAPreIteration"]	*Mathematica*' s code executed before SMTNextStep command	string
es$$["MMAFunctions"]	list of external function patterns (eg. "adnode[i_,j_]") that are included into the source code	NoMMAFunctions ∗string
es$$["MMADescriptions"]	list of descriptions of external functions	NoMMAFunctions ∗string
es$$["IDataNames"]	additional integer type environment data variables (global)	NoIData∗string
es$$["RDataNames"]	additional real type environment data variables (global)	NoRData∗string
es$$["IDataIndex"]	index to additional integer type environment data variable	NoIData∗integer
es$$["RDataIndex"]	index to additional real type environment data variable	NoRData∗integer
es$$["CharSwitch"]	character type user defined constants (local)	NoCharSwitch∗ word
es$$["IntSwitch"]	integer type user defined constants (local)	NoIntegerSwitch∗ integer
es$$["DoubleSwitch"]	double type user defined constants (local)	NoDoubleSwitch∗ doube

Domain specification data structure.

Se also Element Data Structures.

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.

Se also Element Data Structures.

Problem Solving Environments

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`)

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 Mathematica directory ... /AddOns/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" (see `Mixed 3D Solid FE for FEAP`).

How to set paths to FEAP's Visual Studio project is described in the Install.txt file available at www.fgg.uni-lj.si/symech/user/install.txt.

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.

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 global sensitivity problem for all parameters
sens,solv,n	solve global sensitivity problem for the *n*–th sensitivity parameter
sens,solv,n,m	solve global sensitivity problem for parameters *n* to *m*
sens,inte	solve element dependent sensitivity problem for all parameters
sens,inte,n	solve element dependent sensitivity problem for the *n*–th sensitivity parameter
sens,inte,n,m	solve element dependent sensitivity problem for parameters *n* to *m*
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.

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 (`SMSNoDOFCondense`=0).

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 ../AddOns/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 (see `Mixed 3D Solid FE for ELFEN`).

How to set paths to ELFEN's Visual Studio project is 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.

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.

```
In[216]:=
        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*
mswitch	dimensions of the integer switch data array	integer *mswitch*
switch	integer type switches	integer *switch* (*mswitch*)
meuvbl	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*
morder	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

Other environments

The AceGen system is a growing daily. Please check the www.fgg.uni-lj.si/symech/extensions/ page to see if your environment is already supported or www.fgg.uni-lj.si/consulting/ to order creation of the interface for your specific environment.

Interactions: Templates-AceGen-AceFEM

Interactions: Glossary

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–global	f&	pure function

Interactions: 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$$["id","NoDOFGlobal"] es$$["DOFGlobal",*i*] nd$$[n,"id","NoDOF"]	SMTDomainData["dID","NoDOFGlobal"] SMTDomainData["dID","DOFGlobal"] SMTNodeData[m,"NoDOF"]
"SMSNoDOFGlobal"->N	es$$["id","NoDOFGlobal"]	SMTDomainData["dID","NoDOFGlobal"]
"SMSNoAllDOF"->N	es$$["id","NoAllDOF"]	SMTDomainData["dID","NoAllDOF"]
"SMSMaxNoDOFNode"->N	es$$["id","MaxNoDOFNode"]	SMTDomainData["dID","MaxNoDOFNode"]
"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.

Interactions: 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*] nd$$[n,"Data",*i*]	SMTDomainData["dID","NoNodeData"] SMTNodeData[n,"Data",i] SMTNodeData[n,"Data",i]
"SMSIDataNames"−>{"K" …}	es$$["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"]

Interactions: 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"]

Interactions: 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"]

Interactions: *Mathematica*

Template Constant	*AceGen external variables*	*AceFEM data*
"SMSMMAFunctions"->{f…} "SMSMMADescriptions"->{"ab" …}	es$$["id","NoMMAFunctions"] es$$["MMAFunctions",*i*] es$$["MMADescriptions",*i*]	SMTDomainData["dID","NoMMAFunctions"] SMTDomainData["dID","MMAFunctions"] SMTDomainData["dID","MMADescriptions"]
"SMSMMAInitialisation"->"ab"	es$$["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"]

Interactions: 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"]

Interactions: 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.

AceGen Examples

About AceGen Examples

The presented examples are meant to illustrate the general symbolic approach to computational problems 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/ .

Solution to the System of Nonlinear Equations

Description

Generate and verify the *MathLink* program that returns solution to the system of nonlinear equations:

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

where x and y are unknowns and a is parameter.

Solution

Here the appropriate *MathLink* module is created.

In[1]:=

```
In[2]:= << 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$$];
        {x, y} ⫤ {x0, y0};
        SMSDo[i, 1, nmax, 1, {x, y}];
            Φ ⊨ {a x y + x^3, a - x y^2};
            Kt ⊨ SMSD[Φ, {x, y}];
            {Δx, Δy} ⊨ SMSLinearSolve[Kt, -Φ];
            {x, y} ⊣ {x, y} + {Δx, Δy};
            SMSIf[SMSSqrt[{Δx, Δy}.{Δx, Δy}] < ϵ];
                SMSExport[{x, y}, {x$$, y$$}];
                SMSBreak[];
            SMSEndIf[];
            SMSIf[i == nmax];
                SMSPrint["'no convergion'"];
                SMSReturn[];
            SMSEndIf[];
        SMSEndDo[];
        SMSWrite[];
```

```
Solution of 2 linear equations.

Method : test 15 formulae, 147 sub-expressions

[1] File created : test.c Size : 2305
```

Here the *MathLink* program test.exe is build from the generated source code and installed so that functions defined in the source code can be called directly from *Mathematica*. (see also `SMSInstallMathLink`)

```
In[23]:= SMSInstallMathLink[]

Out[23]= {SMSSetLinkOption[test, {i_Integer, j_Integer}], SMSLinkNoEvaluations[test],
          test[x_?NumberQ, y_?NumberQ, a_?NumberQ, tol_?NumberQ, n_?NumberQ]}
```

■ Verification

For the verification of the generated code the solution calculated by the build in function is compared with the solution calculated by the generated code.

```
In[24]:= test[1.9,-1.2,3.,0.0001,10]

Out[24]= {1.93318, -1.24573}

In[25]:= x =.; y =.; a = 3.;
         Solve[{a x y + x^3 == 0, a - x y^2 == 0}, {x, y}]

Out[26]= {{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

In the section `Description of Introductory Example` 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	56.0	80	136	964
A.MMA.Newton	5×5×5	55.9	8.6	2588.3	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	387.5	810	10	1685
B.MMA.Newton	11×11×11	57.5	0.3	4.2	810	16	4
B.AceGen.Gradient	11×11×11	57.5	1.4	28.16	810	4	1598
B.AceGen.Newton	11×11×11	57.5	4.0	1.98	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	25230	139	2

The case A with the trial polynomial interpolation represents the situation where the merit function is complicated and the number of parameters is small. The case B with the finite difference interpolation represents the situation where the merit function is simple and the number of parameters is large.

REMMARK: The presented example is meant to illustrate the general symbolic approach to minimization of complicated merit functions and is not the state of the art solution of thermal conduction problem.

A. Trial Lagrange polynomial interpolation

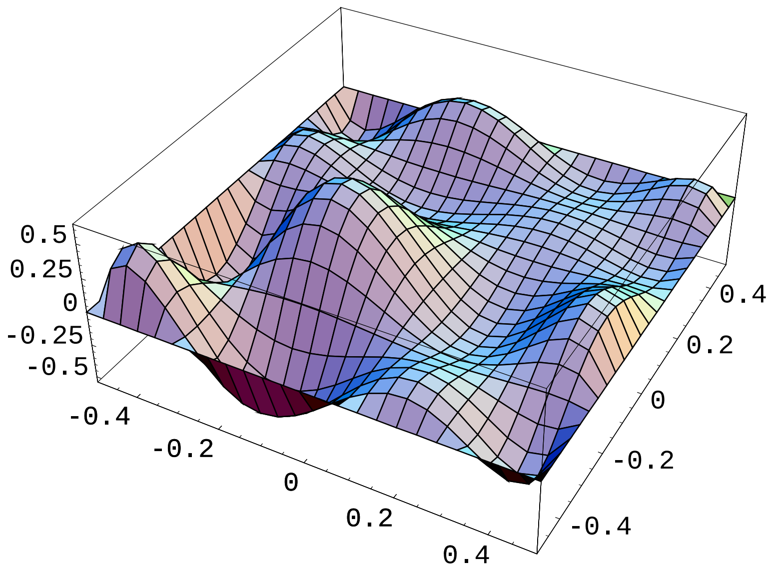
Definitions

A trial function for temperature ϕ is constructed as a fifth order Lagrange polynomial in x y and z direction. The chosen trial function is constructed in a way that satisfies boundary conditions.

```
In[27]:= << AceGen`;
         Clear[x, y, z, a];
         kcond = 0.58; Q = 500;
         order = 5;
         nterm = (order - 1) (order - 1) (order)

Out[31]= 80
```

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

```
In[32]:= 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[a, nterm];
         poly = Flatten[Outer[Times, xp, yp, zp]] // Chop;
         φ = poly.φi;

In[39]:= poly[[28]]
         Plot3D[poly[[28]] /. z → 0.5, {x, -0.5, 0.5}, {y, -0.5, 0.5}, PlotRange → All];

Out[39]= (0.3 + x) (0.5 + x) (12.5 + (-62.5 + (156.25 - 260.417 (-0.3 + x)) (-0.1 + x)) (0.1 + x))
          (0.3 + y) (0.5 + y) (12.5 + (-62.5 + (156.25 - 260.417 (-0.3 + y)) (-0.1 + y)) (0.1 + y))
          (20.8333 + (-104.167 + 260.417 (-0.8 + z)) (-0.6 + 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].

```
In[41]:= ngp = 6;
         << NumericalMath`GaussianQuadrature`;
         g1 = GaussianQuadratureWeights[ngp, -0.5, 0.5];
         g2 = GaussianQuadratureWeights[ngp, -0.5, 0.5];
         g3 = GaussianQuadratureWeights[ngp, 0, 1];
         gp = {g1[[#1[[1]], 1]], g2[[#1[[2]], 1]], g3[[#1[[3]], 1]],
              g1[[#1[[1]], 2]] * g2[[#1[[2]], 2]] * g3[[#1[[3]], 2]]} &/@
            Flatten[Array[{#3, #2, #1} &, {ngp, ngp, ngp}], 2];
```

Direct use of Mathematica

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

```
In[60]:= start = SessionTime[];
         Δϕ = {D[ϕ, x], D[ϕ, y], D[ϕ, z]};
         Π = 1 / 2 kcond Δϕ.Δϕ - ϕ Q;
         Πi = Total[Map[(#[[4]] Π /. {x → #[[1]], y → #[[2]], z → #[[3]]}) &, gp]];
         derivation = SessionTime[] - start

Out[64]= 8.6624560
```

G. Gradient based optimization

```
In[271]:=
      start = SessionTime[]; ii = 0;
      sol = FindMinimum[Πi, Array[{a[#] , 0.} &, nterm],
         Method → "Gradient", EvaluationMonitor :→ (ii++;)];
      {ii, ϕ /. sol[[2]] /. {x → 0, y → 0, z → 0.5}}
      SessionTime[] - start

Out[273]=
      {946, 55.8724}

Out[274]=
      66.8160768
```

N. Newton method based optimization

```
In[275]:=
      start = SessionTime[]; ii = 0;
      sol = FindMinimum[Πi, Array[{a[#] , 0.} &, nterm],
         Method → "Newton", EvaluationMonitor :→ (ii++;)];
      {ii, ϕ /. sol[[2]] /. {x → 0, y → 0, z → 0.5}}
      SessionTime[] - start

Out[30]= {3, 55.8724}

Out[31]= 2588.3418528
```

AceGen code generation

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

```
In[47]:= 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)
          )
```

```
In[49]:= SMSModule["FThermal", Real[a$$[nterm], f$$], "Input" → a$$, "Output" → f$$];
         SMSExport[0, f$$];
         SMSDo[i, 1, gp // Length];
             Π ⊨ Πf[i];
             SMSExport[ Π, f$$, "AddIn" → True];
         SMSEndDo[];
```

```
In[55]:= 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[];
```

```
In[64]:= derivation = SessionTime[] - start

Out[64]= 5.7182224
```

```
In[65]:= 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[];

In[81]:= SMSWrite[];

           Method : FThermal 170 formulae, 6007 sub-expressions

           Method : GThermal 169 formulae, 6115 sub-expressions

           Method : HThermal 87 formulae, 4588 sub-expressions

           [10] File created : Thermal.c Size : 134388

In[82]:= SMSInstallMathLink["Optimize" → False]
         derivation = SessionTime[] - start

Out[82]= {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} &)]}

Out[83]= 12.2275824
```

AceGen Solution

G. Gradient based optimization

```
In[84]:= start = SessionTime[]; ii = 0;
         sol = FindMinimum[FThermal[ϕi], {ϕi, Table[0, {nterm}]},
            Method → "Gradient", Gradient → GThermal[ϕi], EvaluationMonitor :→ (ii++;)];
         {ii, ϕ /. MapThread[Rule, List @@ sol[[2, 1]]] /. {x → 0, y → 0, z → 0.5},
          SessionTime[] - start}

Out[86]= {962, 55.8724, 3.1144784}
```

N. Newton method based optimization

```
In[87]:= start = SessionTime[]; ii = 0;
         sol = FindMinimum[FThermal[ϕi], {ϕi, Table[0, {nterm}]},
            Method → {"Newton", Hessian → HThermal[ϕi]},
            Gradient → GThermal[ϕi], EvaluationMonitor :→ (ii++;)];
         {ii, ϕ /. MapThread[Rule, List @@ sol[[2, 1]]] /. {x → 0, y → 0, z → 0.5},
          SessionTime[] - start}

Out[89]= {4, 55.8724, 1.3519440}
```

B) Finite difference interpolation

Definitions

The central difference approximation of derivatives is used for the points inside the cube and backward or forward difference for the points on the boundary.

```
In[90]:= << AceGen`;
         Clear[a, i, j, k];
         nx = ny = nz = 11;
         dlx = 1. / (nx - 1);
         dly = 1. / (ny - 1);
         dlz = 1. / (nz - 1);
         bound = {0};
         nboun = 1;
         kcond = 0.58; Q = 500;

In[99]:= 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, a[i, j, k - 1]]
               , ++nterm; AppendTo[dofs, a[i, j, k - 1] → nterm]; nterm
               , a[i, j, k - 1] /. dofs
              ]
              , True,
              If[FreeQ[dofs, a[i, j, k]]
               , ++nterm; AppendTo[dofs, a[i, j, k] → nterm]; nterm
               , a[i, j, k] /. dofs
              ]
             ],
             {i, 1, nx + 2}, {j, 1, ny + 2}, {k, 1, nz + 2}] /. b[i_] :→ nterm + i;
         ϕi = Array[a, nterm];
         nterm

Out[102]=
      810
```

Direct use of Mathematica

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

```
In[121]:=
      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]], a[#]] &,
           Extract[index, {{i, j, k}, {i - 1, j, k}, {i + 1, j, k}, {i, j - 1, k},
             {i, j + 1, k}, {i, j, k - 1}, {i, j, k + 1}}]];
         grad = {(aijk[[3]] - aijk[[2]])/(2 dlxt), (aijk[[5]] - aijk[[4]])/(2 dlyt), (aijk[[7]] - aijk[[6]])/(2 dlzt)};

         vol (1 / 2 kcond grad.grad - Q aijk[[1]])
         , {i, 2, nx + 1}, {j, 2, ny + 1}, {k, 2, nz + 1}
        ];
      derivation = SessionTime[] - start

Out[123]=
      0.1502160
```

G. Gradient based optimization

```
In[124]:=
      start = SessionTime[]; ii = 0;
      sol = FindMinimum[Πi, Array[{a[#] , 0.} &, nterm],
         Method → "Gradient", EvaluationMonitor :→ (ii++;)];
      {ii, a[index[[(nx + 3) / 2, (ny + 3) / 2, (nz + 3) / 2]]] /. sol[[2]], SessionTime[] - start}

      FindMinimum::cvmit : Failed to converge to the
         requested accuracy or precision within 100 iterations. More…

Out[19]= {1685, 57.5034, 387.5973376}
```

N. Newton method based optimization

```
In[17]:= start = SessionTime[]; ii = 0;
         sol = FindMinimum[Πi, Array[{a[#] , 0.} &, nterm],
            Method → "Newton", EvaluationMonitor :→ (ii++;)];
         {ii, a[index[[(nx + 3) / 2, (ny + 3) / 2, (nz + 3) / 2]]] /. sol[[2]], SessionTime[] - start}

Out[19]= {4, 57.5034, 3.7654144}
```

AceGen code generation

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

```
In[103]:=
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]])
 )

In[105]:=
SMSModule["FThermal",
  Integer[ndof$$, nt$$[3], index$$["*"]], Real[a$$["*"], mc$$["*"], f$$],
  "Input" → {ndof$$, nt$$, index$$, a$$, mc$$}, "Output" → f$$];
SMSExport[0, f$$];
{nxp, nyp, nzp} ⊨ SMSInteger[Array[nt$$, 3]];
SMSDo[i, 2, nxp + 1];
    SMSDo[j, 2, nyp + 1];
        SMSDo[k, 2, nzp + 1];
            Π ⊨ Πf[i, j, k];
            SMSExport[Π, f$$, "AddIn" → True];
        SMSEndDo[];
    SMSEndDo[];
SMSEndDo[];
```

In[116]:=
```
SMSModule["GThermal", Integer[ndof$$, nt$$[3], index$$["*"]],
  Real[a$$["*"], mc$$["*"], g$$[ndof$$]],
  "Input" → {ndof$$, nt$$, index$$, a$$, mc$$}, "Output" → g$$];
ndof ⊨ SMSInteger[ndof$$];
{nxp, nyp, nzp} ⊨ SMSInteger[Array[nt$$, 3]];
SMSDo[i, 1, ndof];
   SMSExport[0, g$$[i]];
SMSEndDo[];

SMSDo[i, 2, nxp + 1];
   SMSDo[j, 2, nyp + 1];
      SMSDo[k, 2, nzp + 1];
         Π ⊨ Πf[i, j, k];
         SMSDo[i1, 1, indexp // Length];
            dof ⊨ SMSPart[indexp, i1];
            SMSIf[dof <= ndof];
               gi ⊨ SMSD[Π, aijk, i1];
               SMSExport[gi, g$$[dof], "AddIn" → True];
            SMSEndIf[];
         SMSEndDo[];
      SMSEndDo[];
   SMSEndDo[];
SMSEndDo[];
```

In[136]:=
```
derivation = SessionTime[] - start
```

Out[136]=
```
1.8827072
```

In[137]:=
```
SMSModule["HThermal", Integer[ndof$$, nt$$[3], index$$["*"]],
  Real[a$$["*"], mc$$["*"], h$$[ndof$$, ndof$$]],
  "Input" → {ndof$$, nt$$, index$$, a$$, mc$$}, "Output" → h$$];
ndof ⊨ SMSInteger[ndof$$];
{nxp, nyp, nzp} ⊨ SMSInteger[Array[nt$$, 3]];
SMSDo[i, 1, ndof];
    SMSDo[j, 1, ndof];
        SMSExport[0, h$$[i, j]];
    SMSEndDo[];
SMSEndDo[];

SMSDo[i, 2, nxp + 1];
    SMSDo[j, 2, nyp + 1];
        SMSDo[k, 2, nzp + 1];
            Π ⊨ Πf[i, j, k];
            SMSDo[i1, 1, indexp // Length];
                dofi ⊨ SMSPart[indexp, i1];
                SMSIf[dofi <= ndof];
                    gi ⊨ SMSD[Π, aijk, i1];
                    SMSDo[j1, 1, indexp // Length];
                        dofj ⊨ SMSPart[indexp, j1];
                        SMSIf[dofj <= ndof];
                            hij ⊨ SMSD[gi, aijk, j1];
                            SMSExport[hij, h$$[dofi, dofj], "AddIn" → True];
                        SMSEndIf[];
                    SMSEndDo[];
                SMSEndIf[];
            SMSEndDo[];
        SMSEndDo[];
    SMSEndDo[];
SMSEndDo[];
```

In[165]:=
```
SMSWrite[];
```

Method : **FThermal** 32 formulae, 471 sub-expressions

Method : **GThermal** 43 formulae, 562 sub-expressions

Method : **HThermal** 38 formulae, 559 sub-expressions

[2] File created : **Thermal.c** Size : 11915

```
In[166]:=
    SMSInstallMathLink["Optimize" → True]
    derivation = SessionTime[] - start

Out[166]=
    {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] &)]}

Out[167]=
    5.6681504
```

AceGen Solution

G. Gradient based optimization

```
In[168]:=
    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, a[index[[(nx + 3) / 2, (ny + 3) / 2, (nz + 3) / 2]]] /.
      MapThread[Rule, List @@ sol[[2, 1]]], SessionTime[] - start}

    FindMinimum::cvmit : Failed to converge to the
       requested accuracy or precision within 100 iterations. More…

Out[170]=
    {1601, 57.5034, 28.2906800}
```

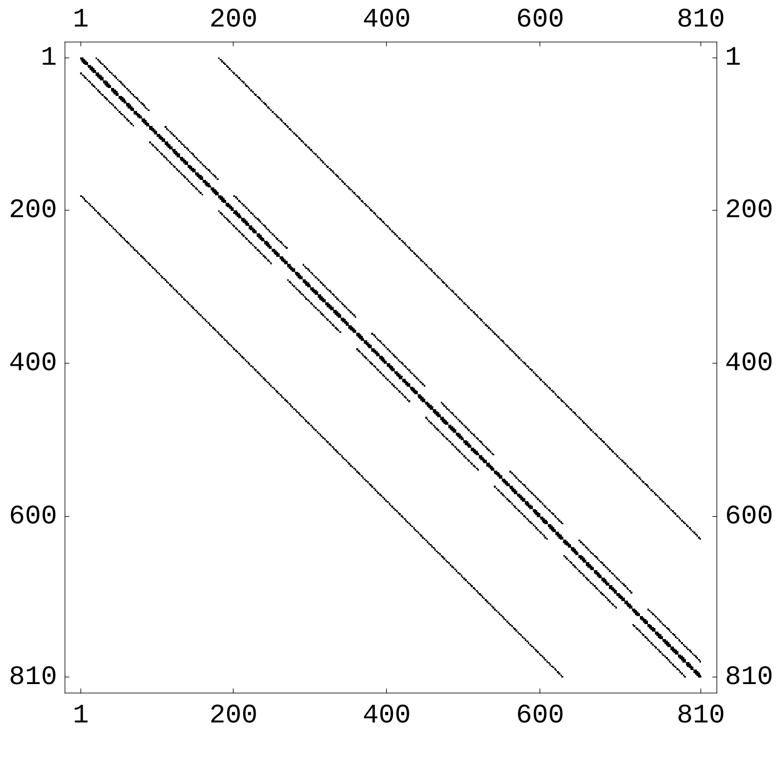
N. Newton method based optimization

In[171]:=
```
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, a[index[[(nx + 3) / 2, (ny + 3) / 2, (nz + 3) / 2]]] /.
  MapThread[Rule, List @@ sol[[2, 1]]], SessionTime[] - start}
```

Out[173]=
```
{4, 57.5034, 2.0229088}
```

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

In[177]:=
```
MatrixPlot[
 HThermal[nterm, {nx, ny, nz}, indexb, Join[0 ϕi, bound], {dlx, dly, dlz, kcond, Q}]]
```



Out[177]=
```
- Graphics -
```

C) Finite element method

First the finite element mesh 30×30×30 is used to obtain convergence solution at the central point of the cube. The procedure to generate heat-conduction element that is used in this example is explained in *AceGen* manual section `Description of FE Characteristic Steps`.

In[243]:=
```
<< AceFEM`;
start = SessionTime[];
SMTInputData[];
k = 0.58; Q = 500;
nn = 30;
SMTAddDomain["cube", "heatconduction", {k, 0, 0, Q}];
SMTAddEssentialBoundary[
  {"X" == -0.5 || "X" == 0.5 || "Y" == -0.5 || "Y" == 0.5 || "Z" == 0. &, 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["Solver" → 5];
```

In[252]:=
```
SMTNextStep[0, 1];
SMTNewtonIteration[];
```

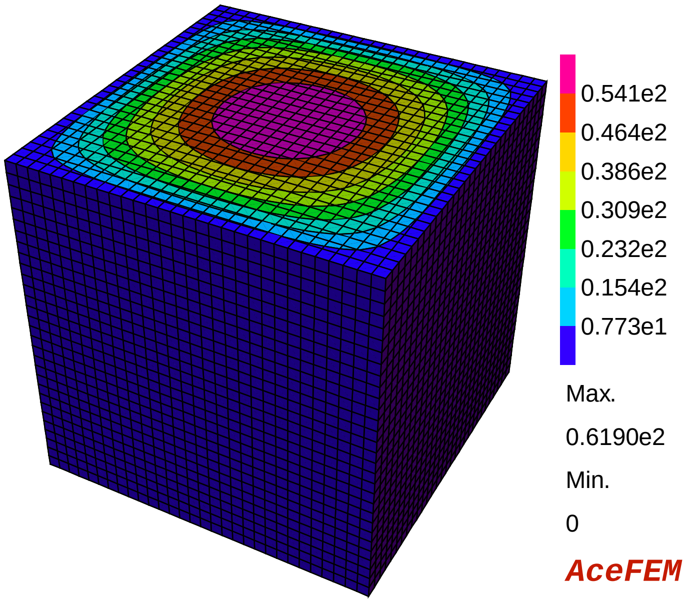
In[254]:=
```
SMTPointValues[{0, 0, 0.5}, SMTPost[1]]
SessionTime[] - start
```

Out[254]=
```
55.8765
```

Out[255]=
```
19.5180656
```

In[256]:=

```
SMTShowMesh["Mesh" → True, "Elements" → True, "Field" → SMTPost[1], "Contour" → True];
```



Mixed 3D Solid FE for AceFEM

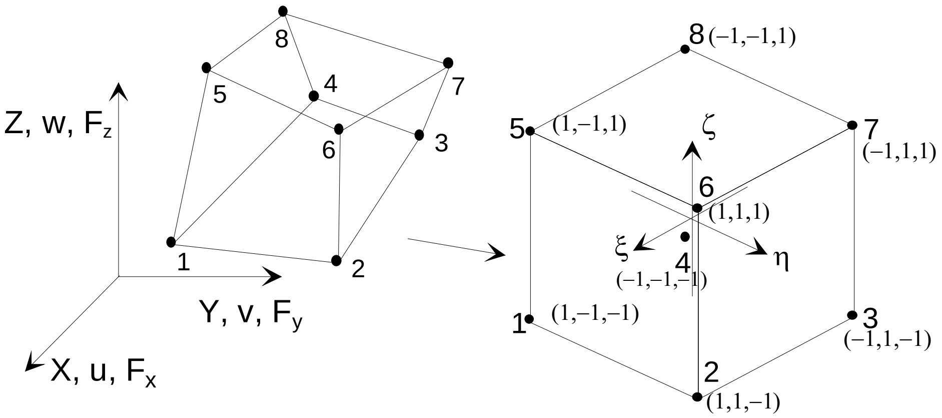
■ Description

Generate the three-dimensional, eight node finite element for the analysis of hyperelastic solid problems. The element has the following characteristics:

⇒ hexahedral topology,

⇒ 8 nodes,

⇒ isoparametric mapping from the reference to the actual frame,



⇒ global unknowns are displacements of the nodes,

$u = u_i\, N_i,\ \ v = v_i\, N_i,\ \ w = w_i\, N_i$

⇒ enhanced strain formulation to improve shear and volumetric locking response,

$$\Delta \boldsymbol{u} = \begin{pmatrix} u_{,X} & u_{,Y} & u_{,Z} \\ v_{,X} & v_{,Y} & v_{,Z} \\ w_{,X} & w_{,Y} & w_{,Z} \end{pmatrix}$$

$$\boldsymbol{D} = \Delta \boldsymbol{u} + \frac{\mathrm{Det}[\boldsymbol{J}_0]}{\mathrm{Det}[\boldsymbol{J}]} \left(\begin{array}{|c|c|c|} \hline \xi\,\alpha_1 & \eta\,\alpha_2 & \zeta\,\alpha_3 \\ \hline \xi\,\alpha_4 & \eta\,\alpha_5 & \zeta\,\alpha_6 \\ \hline \xi\,\alpha_7 & \eta\,\alpha_8 & \zeta\,\alpha_9 \\ \hline \end{array} \right) . \boldsymbol{J}_0{}^{-1}$$

where $\boldsymbol{\alpha} = \{\alpha_1, \alpha_2, \ldots, \alpha_9\}$ are internal degrees of freedom eliminated at the element level.

⇒ the classical hyperelastic Neo-Hooke's potential energy,

$$\Pi = \int_{\Omega_0} \left(\frac{\lambda}{2} (\det \boldsymbol{F} - 1)^2 + \mu \left(\frac{\mathrm{Tr}[\boldsymbol{C}]-2}{2} - \mathrm{Log}[\det \boldsymbol{F}] \right) - \boldsymbol{u}.\boldsymbol{Q} \right) d\Omega_0,$$

where $\boldsymbol{C} = \boldsymbol{F}^T \boldsymbol{F}$ is right Cauchy-Green tensor, $\boldsymbol{F} = \boldsymbol{I} + \boldsymbol{D}$ is enhanced deformation gradient.

■ Solution

In[257]:=
```
<< "AceGen`";
SMSInitialize["Hypersolid", "Environment" → "AceFEM"];
SMSTemplate["SMSTopology" → "H1", "SMSNoDOFCondense" → 9]
```

In[260]:=
```
SMSStandardModule["Tangent and residual"];
SMSDo[IpIndex, 1, SMSInteger[es$$["id", "NoIntPoints"]]];
```

In[262]:=
b:7.3
```
{Xi, Yi, Zi} ⊨ Array[SMSReal[nd$$[#2, "X", #1]] &, {3, 8}];
{ut, vt, wt} ⊨ Array[SMSReal[nd$$[#2, "at", #1]] &, {3, 8}];
SMSGroupDataNames = {"Elastic modulus", "Poisson ratio"};
{Em, ν} ⊨ SMSReal[Array[es$$["Data", #1] &, 2]];
{ξ, η, ζ, wGauss} ⊢ Array[SMSReal[es$$["IntPoints", #1, IpIndex]] &, 4];
{ξi, ηi, ζi} = {{-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 ⊨ MapThread[1 / 8 (1 + ξ #1) (1 + η #2) (1 + ζ #3) &, {ξi, ηi, ζi}];
{X, Y, Z} ⊨ SMSFreeze[{Ni.Xi, Ni.Yi, Ni.Zi}];
Jm ⊨ SMSD[{X, Y, Z}, {ξ, η, ζ}]; Jd ⊨ Det[Jm];
{u, v, w} ⊨ {Ni.ut, Ni.vt, Ni.wt};
Δu ⊨
 SMSD[{u, v, w}, {X, Y, Z}, "Implicit" → {{{ξ, η, ζ}, {X, Y, Z}, SMSInverse[Jm]}}];
Jm0 ⊨ SMSReplaceAll[Jm, {ξ → 0, η → 0, ζ → 0}];
α ⊨ SMSReal[Array[ed$$["ht", #] &, 9]];
H0 = {{ξ α[[1]], η α[[2]], ζ α[[3]]}, {ξ α[[4]], η α[[5]], ζ α[[6]]}, {ξ α[[7]], η α[[8]], ζ α[[9]]}};
H = Det[Jm0]/Jd H0.SMSInverse[Jm0];
𝔻 ⊨ Δu + H;
𝔽 ⊨ IdentityMatrix[3] + 𝔻;
ℂ ⊨ Transpose[𝔽].𝔽; 𝕁 ⊨ Det[𝔽];
{λ, μ} ⊨ SMSHookeToLame[Em, ν];
Π ⊨ 1/2 λ (𝕁 - 1)^2 + μ (1/2 (Tr[ℂ] - 2) - Log[𝕁]);
𝕒 = Flatten[{Transpose[{ut, vt, wt}], α}];
```

In[283]:=

```
SMSDo[i, 1, SMSNoAllDOF];
    Ψi ⊨ Jd wGauss SMSD[Π, a, i];
    SMSExport[SMSResidualSign Ψi, p$$[i], "AddIn" → True];
    SMSDo[j, i, SMSNoAllDOF];
        Kij ⊨ SMSD[Ψi, a, j];
        SMSExport[Kij, s$$[i, j], "AddIn" → True];
    SMSEndDo[];
SMSEndDo[];
SMSEndDo[];
```

In[292]:=

```
SMSWrite[];
```

```
Elimination of local unknowns requires additional
  memory. Corresponding constants are set to:
SMSCondensationData={ed$$[ht, 1], ed$$[ht, 10], ed$$[ht, 19]}
SMSNoTimeStorage=234

Method : SKR 416 formulae, 8026 sub-expressions

[22] File created : Hypersolid.c Size : 36933
```

Test example

You need to install AceFEM package in order to run the example.

In[293]:=

```
<< AceFEM`;
SMTInputData[];
SMTAddDomain["A", "Hypersolid", {1000., .3}];
SMTAddEssentialBoundary[{ "X" == 0 &, 0, 0, 0}, { "X" == 10 &, , , -1}];
SMTMesh["A", "H1", {15, 6, 6}, {{{{0, 0, 0}, {10, 0, 0}}, {{0, 2, 0}, {10, 2, 0}}},
   {{{0, 0, 3}, {10, 0, 2}}, {{0, 2, 3}, {10, 2, 2}}}}];
SMTAnalysis[];
```

In[299]:=
```
SMTNextStep[1, 1];
While[
  While[step = SMTConvergence[10^-8, 10, {"Adaptive", 8, .001, 1, 5}],
   SMTNewtonIteration[];];
  SMTStatusReport[];
  If[step⟦4⟧ === "MinBound", Print["Error: Δλ < Δλmin"]];
  step[[3]]
  , If[step[[1]], SMTStepBack[];];
  SMTNextStep[1, step[[2]]]
 ];
```

T/ΔT=1./1. λ/Δλ=1./1. ‖Δa‖/‖Ψ‖=1.52943×10^{-13} /2.07938×10^{-11} Iter/Total=5/5 Status=0/{Convergence}

T/ΔT=2./1. λ/Δλ=2./1. ‖Δa‖/‖Ψ‖=4.39114×10^{-11} /5.04687×10^{-10} Iter/Total=5/10 Status=0/{Convergence}

T/ΔT=3./1. λ/Δλ=3./1. ‖Δa‖/‖Ψ‖=6.5379×10^{-11} /7.42827×10^{-10} Iter/Total=5/15 Status=0/{Convergence}

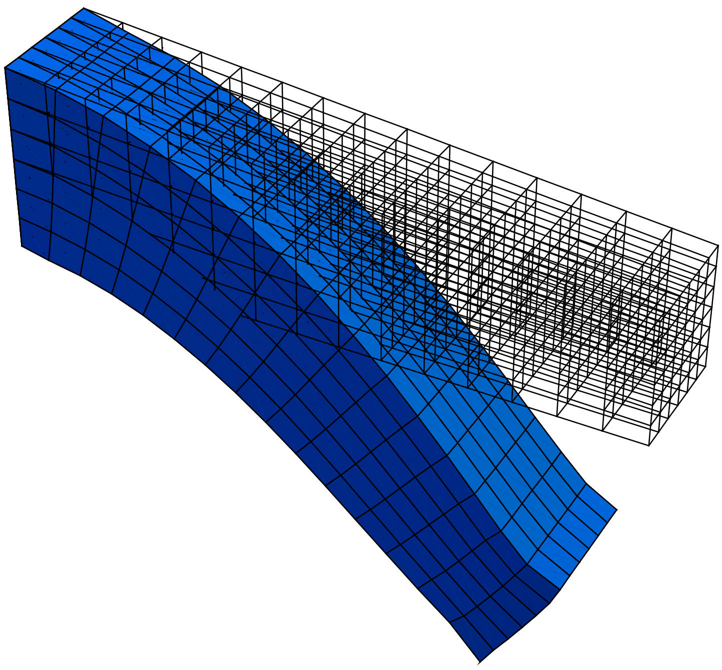
T/ΔT=4./1. λ/Δλ=4./1. ‖Δa‖/‖Ψ‖=1.49454×10^{-11} /2.18178×10^{-10} Iter/Total=5/20 Status=0/{Convergence}

T/ΔT=5./1. λ/Δλ=5./1. ‖Δa‖/‖Ψ‖=4.13173×10^{-12} /6.68365×10^{-11} Iter/Total=5/25 Status=0/{Convergence}

In[301]:=
```
SMTNodeData["X" == 10 && "Y" == 1 && "Z" == 1 &, "at"]
```

Out[301]=
{{-1.69498, -3.82896×10^{-17}, -5.}}

In[302]:=
```
Show[SMTShowMesh["DeformedMesh" → False, "Show" → False, "Elements" -> False],
  SMTShowMesh["DeformedMesh" → True, "Show" → False]];
```



Mixed 3D Solid FE for FEAP

Regenerate the three-dimensional, eight node finite element from chapter `Mixed 3D Solid FE for AceFEM` for *FEAP* environment.

■ Generation of element source code for *FEAP* environment

```
In[303]:=
    << "AceGen`";
    SMSInitialize["Hypersolid", "Environment" → "FEAP"];
    SMSTemplate["SMSTopology" → "H1", "SMSNoDOFCondense" → 9]

In[306]:=
    SMSStandardModule["Tangent and residual"];
    SMSDo[IpIndex, 1, SMSInteger[es$$["id", "NoIntPoints"]]];

In[308]:=
b:7.4
        {Xi, Yi, Zi} ⊧ Array[SMSReal[nd$$[#2, "X", #1]] &, {3, 8}];
        {ut, vt, wt} ⊧ Array[SMSReal[nd$$[#2, "at", #1]] &, {3, 8}];
        SMSGroupDataNames = {"Elastic modulus", "Poisson ratio"};
        {Em, ν} ⊧ SMSReal[Array[es$$["Data", #1] &, 2]];
        {ξ, η, ζ, wGauss} ⊢ Array[SMSReal[es$$["IntPoints", #1, IpIndex]] &, 4];
        {ξi, ηi, ζi} = {{-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 ⊧ MapThread[1 / 8 (1 + ξ #1) (1 + η #2) (1 + ζ #3) &, {ξi, ηi, ζi}];
        {X, Y, Z} ⊧ SMSFreeze[{Ni.Xi, Ni.Yi, Ni.Zi}];
        Jm ⊧ SMSD[{X, Y, Z}, {ξ, η, ζ}]; Jd ⊧ Det[Jm];
        {u, v, w} ⊧ {Ni.ut, Ni.vt, Ni.wt};
        Δu ⊧
      SMSD[{u, v, w}, {X, Y, Z}, "Implicit" → {{{ξ, η, ζ}, {X, Y, Z}, SMSInverse[Jm]}}];
        Jm0 ⊧ SMSReplaceAll[Jm, {ξ → 0, η → 0, ζ → 0}];
        α ⊧ SMSReal[Array[ed$$["ht", #] &, 9]];
        H0 = {{ξ α[[1]], η α[[2]], ζ α[[3]]}, {ξ α[[4]], η α[[5]], ζ α[[6]]}, {ξ α[[7]], η α[[8]], ζ α[[9]]}};
        H = Det[Jm0]/Jd H0.SMSInverse[Jm0];
        𝔻 ⊧ Δu + H;
        𝔽 ⊧ IdentityMatrix[3] + 𝔻;
        ℂ ⊧ Transpose[𝔽].𝔽; 𝕁 ⊧ Det[𝔽];
        {λ, μ} ⊧ SMSHookeToLame[Em, ν];
        Π ⊧ 1/2 λ (𝕁 - 1)^2 + μ (1/2 (Tr[ℂ] - 2) - Log[𝕁]);
        𝕒 = Flatten[{Transpose[{ut, vt, wt}], α}];

In[329]:=
        SMSDo[i, 1, SMSNoAllDOF];
            Ψi ⊧ Jd wGauss SMSD[Π, 𝕒, i];
            SMSExport[SMSResidualSign Ψi, p$$[i], "AddIn" → True];
            SMSDo[j, i, SMSNoAllDOF];
                Kij ⊧ SMSD[Ψi, 𝕒, j];
                SMSExport[Kij, s$$[i, j], "AddIn" → True];
            SMSEndDo[];
        SMSEndDo[];
    SMSEndDo[];
```

```
In[338]:=
SMSWrite[];
   Elimination of local unknowns requires additional
     memory. Corresponding constants are set to:
   SMSCondensationData={ed$$[ht, 1], ed$$[ht, 10], ed$$[ht, 19]}
   SMSNoTimeStorage=234
   Method : SKR10 357 formulae, 7985 sub-expressions
   [21] File created : Hypersolid.f Size : 45572
```

Test example: FEAP

Here is the FEAP input data file for the test example from the chapter `Mixed 3D Solid FE for AceFEM`. You need to install FEAP environment in order to run the example.

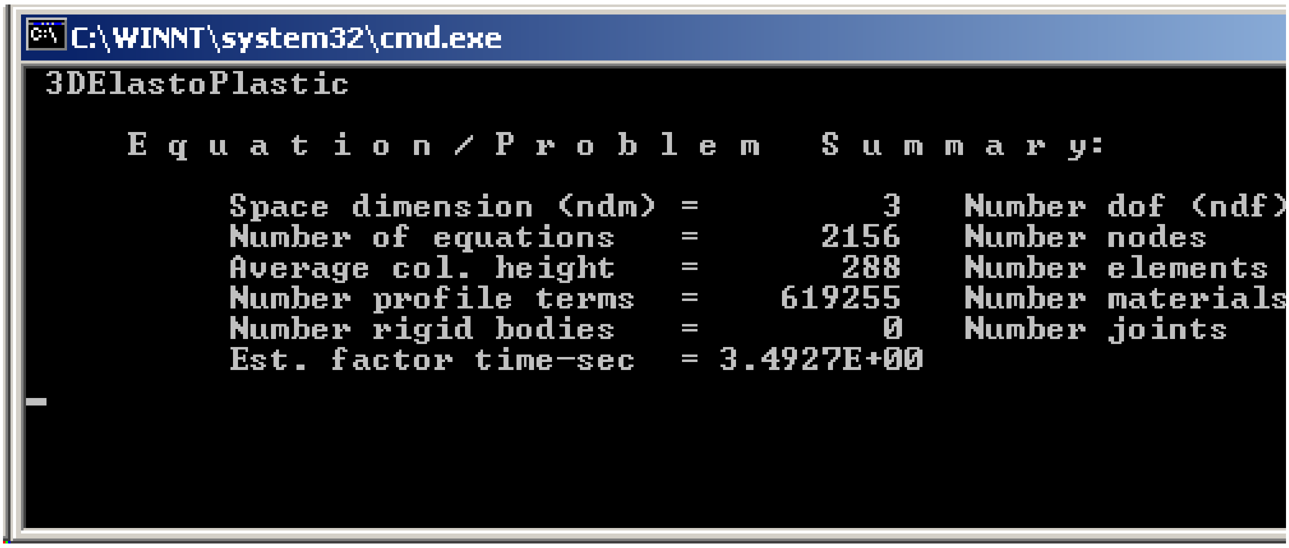
```
feap
0,0,0,3,3,8

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 Install.txt 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.

In[339]:=
```
SMSFEAPMake["Hypersolid"]
```

In[340]:=
```
SMSFEAPRun["feap.inp", "feap.out"]
```

Out[340]=
```
SMSFEAPRun[feap.inp, feap.out]
```



In[341]:=
```
ReadList["feap.out", "Record"][[-4]]
```

Out[341]=
```
  340  1.0000000000000E+01  1.0000000000000E+00  1.0000000000000
E+00 -1.6949762249587E+00 -2.9410643151519E-16 -5.0000000000000E+00
```

3D Solid FE for ELFEN

Regenerate the three-dimensional, eight node finite element from chapter `Mixed 3D Solid FE for AceFEM` 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* as described in About ELFEN section.

In[342]:=

```
<< "AceGen`";
SMSInitialize["Hypersolid", "Environment" → "ELFEN"];
SMSTemplate["SMSTopology" → "H1"]
```

```
Default value for ELFEN$ElementModel is set to:
 D3 ≡  three dimensional solid elements
```

In[345]:=
b:7.5

```
SMSStandardModule["Tangent and residual"];
SMSDo[IpIndex, 1, SMSInteger[es$$["id", "NoIntPoints"]]];
    {Xi, Yi, Zi} ⊨ Array[SMSReal[nd$$[#2, "X", #1]] &, {3, 8}];
    {ut, vt, wt} ⊨ Array[SMSReal[nd$$[#2, "at", #1]] &, {3, 8}];
    SMSGroupDataNames = {"Elastic modulus", "Poisson ratio"};
    {Em, ν} ⊨ SMSReal[Array[es$$["Data", #1] &, 2]];
    {ξ, η, ζ, wGauss} ⊢ Array[SMSReal[es$$["IntPoints", #1, IpIndex]] &, 4];
    {ξi, ηi, ζi} = {{-1, 1, 1, -1, -1, 1, 1, -1},
   {-1, -1, 1, 1, -1, -1, 1, 1}, {-1, -1, -1, -1, 1, 1, 1, 1}};
    Ni ⊨ MapThread[1 / 8 (1 + ξ #1) (1 + η #2) (1 + ζ #3) &, {ξi, ηi, ζi}];
    {X, Y, Z} ⊨ SMSFreeze[{Ni.Xi, Ni.Yi, Ni.Zi}];
    Jm ⊨ SMSD[{X, Y, Z}, {ξ, η, ζ}]; Jd ⊨ Det[Jm];
    {u, v, w} ⊨ {Ni.ut, Ni.vt, Ni.wt};
    Δu ⊨
  SMSD[{u, v, w}, {X, Y, Z}, "Implicit" → {{{ξ, η, ζ}, {X, Y, Z}, SMSInverse[Jm]}}];
    Jm0 ⊨ SMSReplaceAll[Jm, {ξ → 0, η → 0, ζ → 0}];
    𝔽 ⊨ IdentityMatrix[3] + Δu;
    ℂ ⊨ Transpose[𝔽].𝔽; 𝕁 ⊨ Det[𝔽];
    {λ, μ} ⊨ SMSHookeToLame[Em, ν];
    Π ⊨ 1/2 λ (𝕁 - 1)^2 + μ (1/2 (Tr[ℂ] - 2) - Log[𝕁]);
    𝕒 = Flatten[Transpose[{ut, vt, wt}]];
    SMSIf[idata$$["SkipTangent"] == 1];
        SMSDo[i, 1, SMSNoAllDOF];
            Ψi ⊨ Jd wGauss SMSD[Π, 𝕒, i];
            SMSExport[SMSResidualSign Ψi, p$$[i], "AddIn" → True];
        SMSEndDo[];
    SMSElse[];
        SMSDo[i, 1, SMSNoAllDOF];
            Ψi ⊨ Jd wGauss SMSD[Π, 𝕒, i];
            SMSDo[j, i, SMSNoAllDOF];
                Kij ⊨ SMSD[Ψi, 𝕒, j];
                SMSExport[Kij, s$$[i, j], "AddIn" → True];
            SMSEndDo[];
        SMSEndDo[];
    SMSEndIf[];
SMSEndDo[];
```

In[379]:=

```
SMSWrite[];
```

Method : **SKR2999** 258 formulae, 5393 sub-expressions

[17] File created : **Hypersolid.f** Size : 32207

■ Test example: ELFEN

Here is the generated element compiled and linked into the ELFEN's Visual Studio project. See Install.txt 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 ../AddOns/Applications/AceGen/Include/ELFEN/ directory.

In[380]:=

```
SMSELFENMake["Hypersolid"]
```

```
In[381]:=
     SMSELFENRun["ELFEN.dat"]
Out[381]=
     1
```

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 larger and twice slover tnan 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 can produce formulas that are equivalent but not the same (e.q. formulas $\text{Sin}[x]^2 + \text{Cos}[x]^2$ and 1 are equivalent, but not the same).
- The expression optimization procedure can recognize various relations between expressions, however that is no assurance that relations will be in fact recognized.Thus, the users input must not rely on expression optimization as such and it must produce the same result with or without expression optimization (see `Automatic Differentiation` `Expression Optimization`, `Signatures of the Expressions`).
- Check the information given at www.fgg.uni-lj.si/symech/FAQ/.

Message: Variables out of scope

See extensive documentation and examples in Auxiliary Variables, SMSIf, SMSDo, SMSFictive and additional examples below.

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

Erroneous input

```
In[15]:= << AceGen`;
SMSInitialize["test", "Language" -> "C"];
SMSModule["test", Real[x$$, f$$]];
x ⊧ SMSReal[x$$];
SMSIf[x <= 0];
 f ⫤ x^2;
SMSElse[];
 f ⊣ Sin[x];
SMSEndIf[];
SMSExport[f, f$$];
```

```
Some of the auxiliary
  variables in expression are defined out
  of the scope of the current position.
Module: test Description: Error in user input parameters for function:
  SMSExport
Input parameter: {2f} Current scope: {}
Misplaced variables :
 2f ≡ $V[3, 2] Scope: If-False[x ≤ 0]
Events: 0
See also: Auxiliary Variables  Troubleshooting

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

Out[24]= $Aborted
```

Corrected input

```
In[35]:= << AceGen`;
SMSInitialize["test", "Language" -> "Fortran"];
SMSModule["test", Real[x$$, f$$]];
x ⊧ SMSReal[x$$];
SMSIf[x <= 0];
 f ⫤ x^2;
SMSElse[];
 f ⊣ Sin[x];
SMSEndIf[f];
SMSExport[f, f$$];
```

Symbol is defined in other branch of "If" construct

Erroneous input

```
In[45]:= << 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;
```

```
Some of the auxiliary
  variables in expression are defined out
  of the scope of the current position.
Module: test Description: Error in user input parameters for function: SMSR
Input parameter: 2 2f Current scope: {If-False[x ≤ 0]}
Misplaced variables :
 2f ≡ $V[2, 2] Scope: If-True[x ≤ 0]
Events: 0
See also: Auxiliary Variables  Troubleshooting
```

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

```
Out[53]= $Aborted
```

Corrected input

```
In[63]:= 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

```
In[72]:=  a < b < c

Out[72]=  a < b < c

In[73]:=  FullForm[a < b < c]

  Out[73]//FullForm=
      Less[a, b, c]

In[74]:=  CForm[a < b < c]

  Out[74]//CForm=
      Less(a,b,c)
```

There exist no standard C equivalent for Less so it is left in original form and the resulting code would probably failed to compile correctly.

Corrected input

```
In[75]:=  Unprotect[CForm];
          CForm[Less[a_, b_, c_]] := a < b && b < c;
          Protect[CForm];

In[78]:=  CForm[a < b < c]

Out[78]=  a < b && 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 (`SMSPrint`)

New in version

First release

Conversion from the versions before the first official release is done automatically. The major change is that *Computational Templates* package is now fully incorporated into *AceGen* package. The *Driver* package has been renamed to *AceFEM* and is now completely separated from *AceGen* package. More on conversion can be found at www.fgg.uni-lj.si/symech/PreReleaseVersions.nb.