Vedlegg til: Tensorfelt, fra Spesifikasjon til Implementasjon.

Victor Madsen

Universitetet i Bergen,
Institutt for Informatikk,
studieretning Databehandling

June 10, 1993

Contents

Chapter 1

Idea Documentation for Class : Standard

1.1 NAME

Name : *Standard*
Version : 1.0
Compiler : AT&T C++ v. 3.01
Description : Specifies the signature and semantics for some operations that all C++ classes must contain.

1.2 PURPOSE

This documentation describes the syntax and semantics for some operations that all C++ classes must contain. This documentation will only describe the signature of the operations, and the semantics the functions in the classes must have.

1.3 DESCRIPTION

All classes in C++ must have manager operations for assignment, copying, test for equality, and default construction. This description contains the signature and semantics for these operations. The C++ compiler will generate default operations with these properties for all classes, but one may choose to overload the default operations with class-specific implementations.

1.4 C++ CLASS SIGNATURE

This class signature is contained in every C++ class.

```
class Standard
{
   public :
       Standard                    (                    )       ;
       Standard                    ( const Standard & )       ;
       ~Standard                   (                    )       ;
       int          operator ==    ( const Standard & ) const ;
       Standard &   operator =     ( const Standard & )       ;
} ;
```

1.5 OPERATIONS

1.5.1 Generators

Signature : Standard ( ) ;
Preconditions : None.
Result : Creates an arbitrary element in the class *Standard.*
Notify : This is the *default constructor* for the class.
Example : Standard r ;

Signature : Standard (const Standard & r) ;
Preconditions : None.
Result : Constructs a copy of r.
Notify : This is the *copy constructor* for the class.
Example : Standard r1 ;
Standard r2 (r1) ;

Signature : S̃tandard ( ) ;
Preconditions : None.
Result : Removes the object from the computer memory. The object may no longer be used, because it is uninitialized.
Notify : To use the object again, one has to initialize it. This function should be used with care, because the object is no longer initialized, and the internal data structure does not satisfy the abstraction function used in the implementation of the class.
Example : Standard r ;
r::S̃tandard () ;

Signature : Standard & operator = (const Standard & r) ;
Preconditions : None.

Result : Assigns the value of r to the calling object.
Notify : This function returns a reference to the calling object.
Example : Standard r1, r2 ;
r1 = r2 ;

1.5.2 Observer functions

Signature : int operator == (const Standard & r) const ;
Preconditions : None.
Result : Returns a non-zero integer if the calling object is equal to r, if else the integer zero is returned.
Notify : This function tests for semantic equality.
Example : Standard r ;
if (! (r == r))
cout << "Something is very wrong !" << endl ;

1.6 FILES AND VERSIONS

Idea documentation : /tmp_mnt/Home/stud2/vmadsen/tensor/dok/standard.ps

1.7 MAINTENANCE AND SUPPORT

This class specification is documented by :
Victor Madsen
Institute of Informatics
University of Bergen
email : vmadsen@ii.uib.no

Chapter 2

Idea Documentation for Class : Ring

2.1 NAME

Name : *Ring*
Version : 1.0
Compiler : AT&T C++ v. 3.01
Description : Specifies the signature and semantics for the mathematical object *Ring*.

2.2 PURPOSE

This documentation describes the syntax and semantics for the mathematical object *Ring* in C++. There may be many C++ classes that have properties equal to a *Ring*, and one may not give one implementation that cover them all. This documentation will only describe the least signature these classes need, and the semantics the functions in the classes must have.

2.3 DESCRIPTION

A *ring* is a triple $\langle K, +, * \rangle$, where K is a set, and $'+'$ and $'*'$ are functions:

$$+ : K * K \rightarrow K$$

$$* : K * K \rightarrow K$$

in such a way that the following axioms are satisfied:

$$
\begin{array}{lllcl}
\mathbf{K1}: & \exists\, 0 \in K,\ \forall a \in K & : & a+0 & = & a \\
\mathbf{K2}: & \forall a,b \in K & : & a+b & = & b+a \\
\mathbf{K3}: & \forall a,b,c \in K & : & a+(b+c) & = & (a+b)+c \\
\mathbf{K4}: & \forall a \in K,\ \exists -a \in K & : & a+-a & = & 0 \\
\mathbf{K5}: & \exists 1 \in K,\ \forall a \in K & : & a*1 & = & a \\
\mathbf{K6}: & \forall a,b,c \in K & : & a*(b*c) & = & (a*b)*c \\
\mathbf{K7}: & \forall a,b,c \in K & : & (b+c)*a & = & (b*a)+(c*a) \\
\mathbf{K8}: & \forall a,b,c \in K & : & a*(b+c) & = & (a*b)+(a*c)
\end{array}
$$

2.4 ALGEBRAIC SPECIFICATION

We may now construct the following *algebraic specification* of a *Ring*:

```
specification RING
parameters
   sort        Ring
   operations
      0      :              -> Ring
      1      :              -> Ring
      - _    : Ring         -> Ring
      _ + _  : Ring * Ring  -> Ring
      _ * _  : Ring * Ring  -> Ring
specifies
   variables a, b, c : Ring
   equations
      a + 0           ==  a                     //  K1
      a + b           ==  b + a                 //  K2
      a + ( b + c )   ==  ( a + b ) + c         //  K3
      a + ( - a )     ==  0                     //  K4
      a * 1           ==  a                     //  K5
      a * ( b * c )   ==  ( a * b ) * c         //  K6
      ( b + c ) * a   ==  ( b * a ) + ( c * a ) //  K7
      a * ( b + c )   ==  ( a * b ) + ( a * c ) //  K8
end specification
```

This algebraic specification is independent of any programming language, it only asserts properties (semantics) about a sort and a set of functions that operates on the sort.

2.5 C++ CLASS SIGNATURE

One may probably implement many different classes in C++ having the properties described above. It is still an advantage to use equal signature for all classes sharing the properties described. The reason is that we then may implement algorithms that are reusable between a wide range of different implementations.

From the given algebraic specification, we can construct a C++ class signature which every class with *Ring* semantics must contain:

```
class Ring
{
   public :

       //   C++ specific functions
       Ring                    (              )       ;
       Ring                    ( const Ring & )       ;
       ~Ring                   (              )       ;
       int      operator ==    ( const Ring & ) const ;
       Ring &   operator =     ( const Ring & )       ;

       //   basic functions
       Ring                    ( const int  & )       ;
       Ring     operator  -    (              ) const ;
       Ring     operator  +    ( const Ring & ) const ;
       Ring     operator  *    ( const Ring & ) const ;

       //   Composed functions
       Ring     operator  -    ( const Ring & ) const ;
       Ring &   operator +=    ( const Ring & )       ;
       Ring &   operator -=    ( const Ring & )       ;
       Ring &   operator *=    ( const Ring & )       ;
} ;
```

We demand that all classes in C++ that is supposed to represent a em Ring, must at least have the above given class signature. Notice that this is the minimum set of functions an implementation must contain, it will cause no problem if there are implemented more member functions in the class. Typically, there must at least be some extra constructors that are dependent of the actual implementation of the *ring*. After this specification, the built in types *int*, *float*, and *double* in C++ are implementations of a *Ring* class. This is because they have the syntax and semantics described above.

Most of the member functions in the class signature have their obvious counterpart in the algebraic specification and C++ semantics (e.g. test for equality, assignment, default constructor, destructor, ...), but the member functions:

```
Ring                    ( const int  & )       ;
Ring     operator  -    ( const Ring & ) const ;
Ring &   operator +=    ( const Ring & )       ;
Ring &   operator -=    ( const Ring & )       ;
Ring &   operator *=    ( const Ring & )       ;
```

are special. Notice that the C++ signature does not contain the constant functions 0 and 1. This is because one in practical use most often need to construct a element that is a sum of several 1's (f. ex. 2, 3, ...). The signature contains because of this a constructor that generates the constants 0 and 1, and also calculates the sum of a number of 1's. The constructor has the following algebraic specification:

```
i : integer

( 0 <= i ) => Ring ( i ) == IF n = 0 THEN 0
                                    ELSE 1 + Ring ( n - 1 ) FI
```

The other operators (-, +=, -=, *=) belongs to the class because of the return-value problem inherent in C++. They have the specification:

```
a, b : Ring

( a - b  )  ==  ( a + ( -b ))
( a += b )  ==  ( a = a + b )
( a -= b )  ==  ( a = a - b )
( a *= b )  ==  ( a = a * b )
```

2.6 OPERATIONS

2.6.1 Generators

Signature : Ring (const int i) ;
Preconditions : 0 <= i.
Result : If i is zero, the element of unity under addition (0) is returned. If i is greater than zero, the sum of i unit elements under multiplication is returned.
Notify : See also the algebraic specification above.
Example : Ring zero (0);
Ring one (1) ;
Ring two (2) ;

Signature : Ring operator - ( ) const ;
Preconditions : None.
Result : Returns an element $e2$ with the property that: $e1 + e2 = Ring(0)$, where $e1$ is the calling object.
Notify : May be regarded as *unary minus* in standard arithmetic notation.
Example : Ring r ;
Ring zero (0) ;
if (! (r + (-r) == zero))
cout << "Something is very wrong !" << endl ;

Signature : Ring operator + (const Ring & r) const ;
Preconditions : None.

Result : This member function with the following properties:

```
a + Ring ( 0 ) == a
a + ( b + c ) == ( a + b ) + c
a + b == b + a
```

where a, b, c are objects of type *Ring*. These properties state that the function is associative and commutative, and has $Ring(0)$ as identity element.

Notify : May be regarded as *addition* in standard arithmetic notation.

Example : Ring r1, r2, r3 ;
r3 = r1 + r2 ;

Signature : Ring operator * (const Ring & r) const ;
Preconditions : None.
Result : This member function has the following properties:

```
a * Ring ( 1 ) == a
a * ( b * c ) == ( a * b ) * c
a * ( b + c ) == ( a * b ) + ( a * c )
( a + b ) * c == ( a * c ) + ( b * c )
```

where a, b, c are objects of type *Ring*.These properties state that the function is associative and distributive (not necessary commutative !), and has $Ring(0)$ as identity element.

Notify : May be regarded as *multiplication* in standard arithmetic notation.

Example : Ring r1, r2, r3 ;
r3 = r1 * r2 ;

2.6.2 Composed functions

Signature : Ring operator - (const Ring & r) const ;
Preconditions : None.
Result : This function is equal to: $a + (-r)$, where a is the calling object.
Example : Ring r1, r2 ;

```
if ( ! ( r1 - r2 == r1 + ( - r2 ) ) )
    cout << "Something is very wrong !" << endl ;
```

Signature : Ring & operator += (const Ring & r) ;
Preconditions : None.
Result : This function is equal to: $a = a + r$, where a is the calling object.
Example : Ring r1, r2 ;
r1 += r2 ;

Signature : Ring & operator -= (const Ring & r) ;
Preconditions : None.
Result : This function is equal to: $a = a - r$, where a is the calling object.
Example : Ring r1, r2 ;
r1 -= r2 ;

Signature : Ring & operator *= (const Ring & r) ;
Preconditions : None.
Result : This function is equal to: $a = a * r$, where a is the calling object.
Example : Ring r1, r2 ;
r1 *= r2 ;

2.7 REFERENCES

A good introduction to algebraic specification may be found in:
David A. Watt : *Programming Language Syntax and Semantics*, Prentice Hall 1991.

Most of the mathematical background will be found in:
Serge Lang: *Algebra*, Addison Wesley 1965.
Abraham, Marsden and Ratiu : *Manifolds, Tensor Analysis, and Applications*, Springer-Verlag 1983.

For a more theoretical approach, see :
M. Haveraaen, V. Madsen, H. Munthe-Kaas : *Algebraic Programming Technology for Partial Differential Equations*, Precedings from "Norsk Informatikk Konferanse 1992".

2.8 FILES AND VERSIONS

Idea documentation : /tmp_mnt/Home/stud2/vmadsen/tensor/dok/ring_idea.ps

2.9 MAINTENANCE AND SUPPORT

This class specification is documented by :
Victor Madsen
Institute of Informatics
University of Bergen
email : vmadsen@ii.uib.no

Chapter 3

Idea Documentation for Class : Field

3.1 NAME

Name	: *Field*
Version	: 1.0
Compiler	: AT&T C++ v. 3.01
Based on	: Idea Documentation for the Class *Ring*.
Description	: Specifies the signature and semantics for the mathematical object *Field*.

3.2 PURPOSE

This documentation describes the syntax and semantics for the mathematical object *Field* in C++. There may be many C++ classes that have properties equal to a *Field*, and one may not give one implementation that cover them all. This documentation will only describe the least signature these classes need, and the semantics the functions in the classes must have.

3.3 DESCRIPTION

A *Field* is a triple $\langle K, +, * \rangle$, where $\langle K, +, * \rangle$ is a *Ring* and in addition we have the axioms:

$$\begin{array}{lllll} \textbf{F1}: & \forall a, b \in K & : & a * b & = & b * a \\ \textbf{K2}: & \forall a \in K - 0,\ \exists a^{-1} \in K & : & a * a^{-1} & = & 1 \end{array}$$

3.4 ALGEBRAIC SPECIFICATION

We may construct the following *algebraic specification* of a *Field*:

```
specification FIELD
parameters
   sort        Field
   operations
      0      :                      ->   Field
      1      :                      ->   Field
      - _    : Field                ->   Field
      _ + _  : Field * Field        ->   Field
      _ * _  : Field * Field        ->   Field
      1 / _  : Field ( #1 != 0 )    ->   Field
specifies
   variables a, b : Field
   equations
      RING ( Field, 0, 1, -, +, * )

      a * b       == b * a
      a * ( 1/a ) == 1
end specification
```

This algebraic specification is independent of any programming language, it only asserts properties (semantics) between a sort (class) and a set of functions that operates on the sort. Note the reuse of the algebraic specification of RING.

3.5 C++ CLASS SIGNATURE

One may probably implement many different classes in C++ that have the properties described above. It is still an advantage to use equal signature for all classes that have the properties described. The reason is that we then may implement algorithms that are reusable between a wide range of different implementations. From the given algebraic specification, we may construct a C++ class signature that every class with *Field* semantics must contain:

```
class Field
{
   public :

      //   C++ specific functions
      Field                  (                )       ;
      Field                  ( const Field & )       ;
      ~Field                 (                )       ;
      int      operator ==   ( const Field & ) const ;
      Field &   operator =   ( const Field & )       ;
```

```
        //   basic functions
        Field                   ( const int   & )       ;
        Field     operator  -   (               ) const ;
        Field     operator  +   ( const Field & ) const ;
        Field     operator  *   ( const Field & ) const ;
        Field     operator  /   ( const Field & ) const ;

        //   Composed functions
        Field     operator  -   ( const Field & ) const ;
        Field &   operator +=   ( const Field & )       ;
        Field &   operator -=   ( const Field & )       ;
        Field &   operator *=   ( const Field & )       ;
        Field &   operator /=   ( const Field & )       ;
} ;
```

We demand that all classes in C++ that is supposed to represent a *Field*, must at least have the above given class signature. Notice that this is the minimum set of functions an implementation must contain, it will cause no problem if there are implemented more member functions in the class. Typically, there must at least be some extra constructors that are dependent of the actual implementation. After this specification, the built in types *float* and *double* in C++ are both implementations of a *Field* class. This is because they both have the syntax and semantics described above. On the other hand, the type *int* is not a *Field* class, even if the type has the signature given above. This is because the division of integers not satisfy the axiom **F2**.

The member functions:

```
Field     operator  /    ( const Field & ) const ;
Field &   operator /=    ( const Field & )       ;
```

have their associated algebraic specification:

```
a, b : Field

( a / b  ) == a * ( 1/b )
( a /= b ) == ( a = a / b )
```

3.6 OPERATIONS

This class specification contains all member functions from the class specification *Ring*. In addition we have the functions:

3.6.1 Generators

Signature : Field operator * (const Field & f) const ;

Preconditions : None.
Result : This function has the specification for a ring, and in addition:
(a * b) == b * a
I.e. the operator is *commutative.*
Example : Field a, b ;
if (! (a * b == b * a))
cout << "Error !" << endl ;

Signature : Field operator / (const Field & f) const ;
Preconditions : f is not equal to $Field(0)$.
Result : This function has the following specification:
(a / b) == a * Field (1) / b
(b * Field (1) / b) == Field (1)
Example : Field one (1), two (2) ;
Field ptFive ;
ptFive = one / two ;

3.6.2 Composed functions

Signature : Field & operator /= (const Field & f) ;
Preconditions : f is not equal to $Field(0)$.
Result : This function is equal to: $a = a/f$, where a is the calling object.
Example : Field one (1), two (2) ;
Field ptFive (1) ;
ptFive /= two ;

3.7 REFERENCES

A good introduction to algebraic specification may be found in:
David A. Watt : *Programming Language Syntax and Semantics*, Prentice Hall 1991.

Most of the mathematical background will be found in:
Serge Lang: *Algebra*, Addison Wesley 1965.
Abraham, Marsden and Ratiu : *Manifolds, Tensor Analysis, and Applications*, Springer-Verlag 1983.

For a more theoretical approach, see :
M. Haveraaen, V. Madsen, H. Munthe-Kaas : *Algebraic Programming Technology for Partial Differential Equations*, Precedings from "Norsk Informatikk Konferanse 1992".

3.8 FILES AND VERSIONS

Idea documentation : /tmp_mnt/Home/stud2/vmadsen/tensor/dok/field_idea.ps
Based on : /tmp_mnt/Home/stud2/vmadsen/tensor/dok/ring_idea.ps

3.9 MAINTENANCE AND SUPPORT

This class specification is documented by :
Victor Madsen
Institute of Informatics
University of Bergen
email : vmadsen@ii.uib.no

Chapter 4

Idea Documentation for Class : diffRing

4.1 NAME

Name	: *diffRing*
Version	: 1.0
Compiler	: AT&T C++ v. 3.01
Based on	: Idea documentation for class *Ring*.
Description	: Specifies the signature and semantics for the mathematical object *differentiable ring*.

4.2 PURPOSE

This documentation describes the syntax and semantics for the mathematical object *differentiable Ring* in C++. There are many classes having properties equal to a differentiable ring, and one may not give one implementation that cover them all. This documentation will only describe the least signature these classes need, and what properties the functions in the classes must have.

4.3 DESCRIPTION

A *differential operator* on a *Ring* R, is a linear map $D_r : R \to R$, which distributes over '+' and '*' like this:

1. $\forall x, y \in R : D_r(x + y) = D_r(x) + D_r(y)$
2. $\forall x, y \in R : D_r(x * y) = D_r(x) * y + x * D_r(y)$

A differential operator D_r is *trivial* if for $\forall x \in R, D_r(x) = 0$.

All rings may be equipped with a trivial differential operator, but there are not all rings that have differential operators that are not trivial. All classes with ring properties that also have one or more nontrivial differential operators are called *differential rings.* Two differential operators, D_1 and D_2, on a differential ring are *lineary independent* if and only if :

$$\forall a, b \in R : a * D_1(c) + b * D_2(c) = 0 \Rightarrow a = b = 0$$

where $c \in R$ and $D_1(c)$ or $D_2(c)$ are nonzero. The number of nontrivial, lineary independent, differential operators on a differential ring is called the *dimension* of the differential ring. The application of the nontrivial, differential operator number i ($0 \leq i < dimension$) on a element $e \in R$ is said to be *the partial derivative* of e in *dimension* i.

4.4 ALGEBRAIC SPECIFICATION

We may give the following *algebraic specification* of a differential ring:

```
specification DIFFERENTIAL-RING
include BOOL, HELTALL
include instantiation ARRAY  by  diffRing
                using rArray for Array
parameters
   sort        diffRing
   operations
      0                    :                              -> diffRing
      _ + _                :  diffRing * diffRing         -> diffRing
      - _                  :  diffRing                    -> diffRing
      _ * _                :  diffRing * diffRing         -> diffRing
      numDim       ( _ )   :  diffRing                    -> Integer
      partialDiff ( _,_ ) :  Integer * diffRing
                              | (0<#1<=numDim(#2))        -> diffRing
uses
   nonZeroDeriv  ( _ )     :  Integer
                              | (0<#1<=numDim(0))         -> diffRing
   linSum        ( _,_,_ ) :  rArray * diffRing* Integer
            | (size(#1)=numDim(#2)/\0<#3<=numDim(0)) -> diffRing
specifies
   variables   a, b : diffRing
               A    : rArray
               i, j : Integer
   equations
      RING  ( ring, 0, 1, +, -, * )

      partialDiff ( i, a + b ) == partialDiff ( i, a ) + partialDiff ( i, b )
      partialDiff ( i, a * b ) == partialDiff ( i, a ) * b +
```

```
                                a * partialDiff ( i, b )

      0          < numDim (b)
      numDim(a) == numDim (b)

      linSum(A,b,1) == A [ 1 ] * partialDiff( 1, b )
      linSum(A,b,i) == A [ i ] * partialDiff( i, b ) + linsum ( A, b, i-1 )

      ( linSum ( A, nonZeroDeriv(i), numDim(b) ) = 0 ) => ( A [ i ] = 0 )

      ! ( partialDiff ( i, nonZeroDeriv (i) ) = 0 )
end specification
```

This specification is independent of any programming language, it only asserts properties (semantics) between a sort and a set of functions that operates on the sort. Notice the reuse of the specification RING.

4.5 C++ CLASS SIGNATURE

One may probably implement many different classes in C++ that have the properties above. It is still an advantage to use the same signature for all classes that have the properties described. The reason is that we then may implement algorithms that are reusable between a wide range of different differentiable rings. Because of this, we demand that all classes in C++ that is supposed to represent a differentiable ring, must at least have the following signature:

```
class diffRing
{
   public :

      //   C++ specific functions
      diffRing                   (                   )       ;
      diffRing                   ( const diffRing & )       ;
      ~diffRing                  (                   )       ;
      int          operator ==   ( const diffRing & ) const ;
      diffRing &   operator =    ( const diffRing & )       ;

      //   Basic functions
      diffRing                   ( const int      & )       ;
      diffRing     operator  -   (                   ) const ;
      diffRing     operator  +   ( const diffRing & ) const ;
      diffRing     operator  *   ( const diffRing & ) const ;
static int         numDimensions (                   )       ;
      diffRing     partialDiff   ( const int        ) const ;

      //   Composed functions
      diffRing     operator  -   ( const diffRing & ) const ;
```

```
        diffRing &   operator +=    ( const diffRing & )       ;
        diffRing &   operator -=    ( const diffRing & )       ;
        diffRing &   operator *=    ( const diffRing & )       ;

} ;
```

Notice that this is the minimum set of functions an implementation must contain, it will cause no problem if there are implemented more member functions in the class. Typically, there must at least be some extra constructors that are dependent of the actual implementation of the differential ring. Observe that the function *numDimensions* is declared as *static*. This implies that all elements in the class must have equal number of dimensions.

4.6 OPERATIONS

This class specification contain all member functions from the class specification *Ring*. We have also have the functions specified below.

4.6.1 Generators

Signature : diffRing partialDiff (const int i) const ;
Preconditions : $0 <= i < diffRing :: numDimensions()$.
Result : Returns the partial derivative of the calling object in dimension nr. i.
Notify : The dimensions are indexed from 0 to $diffRing :: numDimensions() - 1$.
Example : diffRing r1, r2 ;
if (0 < diffRing::numDimensions ())
r2 = r1.partialDiff (0) ;

4.6.2 Observer functions

Signature : static int numDimensions ( ) ;
Preconditions : None.
Result : Returns the number of dimensions of the objects in the class.
Notify : This is a static member function. This implies that all objects in the class have equal number of dimensions.
Example : cout << "The number of dimensions are : "
<< diffRing::numDimensions () << endl ;

4.7 REFERENCES

A good introduction to algebraic specification may be found in:
David A. Watt : *Programming Language Syntax and Semantics*, Prentice Hall 1991.

Most of the mathematical background will be found in:
Serge Lang: *Algebra*, Addison Wesley 1965.
Abraham, Marsden and Ratiu : *Manifolds, Tensor Analysis, and Applications*, Springer-Verlag 1983.

For a more theoretical approach, see :
M. Haveraaen, V. Madsen, H. Munthe-Kaas : *Algebraic Programming Technology for Partial Differential Equations*, Precedings from "Norsk Informatikk Konferanse 1992".
For a less theoretical approach, see :
V. Madsen:*Tensors, from specification to implementation*; Thesis for the master degree, Instititute of Informatics, University of Bergen.

4.8 FILES AND VERSIONS

Idea documentation : /tmp_mnt/Home/stud2/vmadsen/tensor/dok/diffring_idea.ps
Based on : /tmp_mnt/Home/stud2/vmadsen/tensor/dok/ring_idea.ps

4.9 MAINTENANCE AND SUPPORT

This class specification is documented by :
Victor Madsen
Institute of Informatics
University of Bergen
email : vmadsen@ii.uib.no

Chapter 5

Idea Documentation for Class : Module

5.1 NAME

Name	: *Module*
Version	: 1.0
Compiler	: AT&T C++ v. 3.01
Based on	: Idea documentation for class *Ring*.
Description	: Specifies the signature and semantics for the mathematical object *Module over Ring*.

5.2 PURPOSE

This documentation describes the syntax and semantics for the mathematical object *Module over ring* in C++. There may be many C++ classes that have properties equal to a *Module*, and one may not give one implementation that cover them all. This documentation will only describe the least signature these classes need, and the semantics the functions in the classes must have.

5.3 DESCRIPTION

5.3.1 Def. Module over Ring

A set M and a *Ring* $\langle R, +, * \rangle$, together with the functions:

$$+ : M * M \rightarrow M$$

$$* : R * M \rightarrow M$$

is said to be a *module* over the ring R, if the following axioms are satisfied:

M1 :	$\forall a,b,c \in M$	: $a+(b+c)$	=	$(a+b)+c$
M2 :	$\exists 0 \in M,\ \forall a \in M$	: $a+0$	=	a
M3 :	$\forall a \in M\ , \exists -a \in M$	: $a+(-a)$	=	0
M4 :	$\forall c \in R\ , \forall a,b \in M$	: $c*(a+b)$	=	$c*a+c*b$
M5 :	$\forall c,k \in R\ , \forall a \in M$	: $(c+k)*a$	=	$c*a+k*a$
M6 :	$\forall c,k \in R\ , \forall a \in M$	: $c*(k*a)$	=	$(c*k)*a$
M7 :	$\forall a \in M$	: $1*a$	=	a

and is referred to as the module $\langle M, R, +, * \rangle$.

5.3.2 Lineary Independent Elements

Given a module $\langle M, R, +, * \rangle$, and a subset $N \subseteq M$. All elements in the set N are *lineary independent* if all subsets $\{v_1, \ldots, v_k\} \subseteq N (k < \infty)$ and $\forall c^1, \ldots, c^k \in R$ the equation:

$$c^1 * v_1 + \ldots + c^k * v_k = 0$$

is satisfied if and only if $c^i = 0 (0 < i \leq k)$.

5.3.3 Linear combination

Given a module $\langle M, R, +, * \rangle$. A sum:

$$c^1 * v_1 + \ldots + c^k * v_k$$

where $v_i \in M$, and $c^i \in R$ is a *linear combination* of the elements in the set $\{v_1, \ldots, v_k\}$.

Basis for Module

Given a module $\langle M, R, +, * \rangle$. A set $B \subseteq M$ is a *basis* for M if all elements in B are linary independent, and $\forall v \in M$ may be written as a linear combination of the elements in the set B. Not all modules have a basis.

Dimension of a Module

Given a module $\langle M, R, +, * \rangle$ and a basis B for M. The number of elements in the set B is the number of *dimensions* of the module.

If M has a basis, and the basis is a finite set (i.e. dimension $< \infty$), the module is finite dimensional. If else, it is said to be infinite dimensional.

Coordinate representation

Given a finite dimensional module $\langle M, R, +, * \rangle$ with basis $B = \{b_1, \ldots, b_d\}$. Now every element $v \in M$ may be written as a linear combination:

$$v = c^1 * b_1 + \ldots + c^d * b_d, \quad where \ c^i \in R \ , \ 0 < i \leq d$$

The sequence $\langle c^1, \ldots, c^d \rangle$ is said to be the *coordinates* of v with respect to B.

5.4 ALGEBRAIC SPECIFICATION

We may construct the following *algebraic specification* of a finite dimensional *Module over a Ring*:

```
specification MODULE
include BOOL, INTEGER
parameters
   sort          Ring, Module
   operations
      0                 :                                          -> Ring
      1                 :                                          -> Ring
      - _               : Ring                                     -> Ring
      _ + _             : Ring * Ring                              -> Ring
      _ * _             : Ring * Ring                              -> Ring

      0                 :                                          -> Module
      _ + _             : Module * Module                          -> Module
      - _               : Module                                   -> Module
      _ * _             : Ring * Module                            -> Module
      dimension ( _ )   : Module                                   -> Integer
      coordinat ( _,_ ) : Module * Integer |(0<#2<=dimension(#1)) -> Ring
      basis     ( _,_ ) : Module * Integer |(0<#2<=dimension(#1)) -> Module
uses
      linComb ( _,_ )   : Integer * Module |(0<#1<=dimension(#2)) -> Module
specifies
   variables    m, n    : Ring
                a, b, c : Module
                i, j    : Integer
   equations
      RING ( Ring, 0, 1, -, +, * )
      a + 0          ==   a              //   M1
      a + ( - a )    ==   0              //   M2
      a + ( b + c )  ==   ( a + b ) + c  //   M3
      a + b          ==   b + a          //   M4
      m * ( a + b )  ==   m * a + m * b  //   M5
      ( m + n ) * a  ==   m * a + n * a  //   M6

                  ( 0 < dimension ( a ) )          == true
```

```
                dimension ( a )                 == dimension ( b )
                basis ( a, i )                  == basis ( b , i )
                linComb ( dimension ( a ), a )  == a
    ( i  = j ) => coordinat ( basis ( a, i ) , j ) == 1
    ( i <> j ) => coordinat ( basis ( a, i ) , j ) == 0

    linComb ( 1, a ) == coordinat ( a, 1 ) * basis ( a, 1 )
    linComb ( i, a ) == coordinat ( a, i ) * basis ( a, i ) + linComb ( i - 1, e )
end specification
```

This algebraic specification is independent of any programming language, it only asserts properties (semantics) between a sort (class) and a set of functions that operates on the sort. This specification use an array class that is not formally specified, but the semantic is obvious, and the specification of ARRAY is because of this not explicitly shown.

5.5 C++ SIGNATURE

One may probably implement many different classes in C++ that have the properties described above. It is still an advantage to use equal signature for all classes that have the properties described. The reason is that we then may implement algorithms that are reusable between a wide range of different *Modules*.

From the given algebraic specification, we can construct a C++ class signature that every class with *Module over Ring* semantics must contain:

```
class module
{  public:

       //   C++ specific operations
       module                    ( const module &       )       ;
       ~module                   (                      )       ;
       int          operator  == ( const module &       ) const ;
       module  &    operator  =  ( const module &       )       ;

       //   Basic operations
       module                    (                      )       ;
friend module       operator  *  ( const Ring & ,
                                   const module &       )       ;
       module       operator  -  (                      ) const ;
       module       operator  +  ( const module &       ) const ;
       array<Ring>  coordinates  (                      ) const ;
static int          dimension    (                      )       ;
static array<module> basis       (                      )       ;

       //   Composed functions
       module       operator  -  ( const module &       ) const ;
       module &     operator  -= ( const module &       )       ;
```

```
        module &      operator  += ( const module &       )       ;
        module &      operator  *= ( const Ring   &       )       ;
} ;
```

We demand that all classes in C++ that is supposed to represent a em Module, must at least have the above given class signature. We also demand that the class *Ring* must satisfy the requirements given in the idea documentation for a ring. Notice that this is the minimum set of functions an implementation must contain, it will cause no problem if there are implemented more member functions in the class. Typically, there must at least be some extra constructors that are dependent of the actual implementation of the *Module*. Also notice that both *basis* and *dimension* are declared as static functions. This implies that all elements in the class have equal basis and dimension.

5.6 OPERATIONS

5.6.1 Generators

Signature : module ( ) ;
Preconditions : None.
Result : Constructs the zero-element in the class. This element has the property:
m + module () = m
for all elements m in the class. It may be proved that the following theorem is true:
module().coordinates () [i] == Ring (0)
for all $0 \leq i <$ module::dimension ().
Notify : This is the *unit element* under addition.
Example : module m ;

Signature : friend
module operator * (const Ring & r, const module & m) ;
Preconditions : None.
Result : This element has the properties:
m * (a + b) == m * a + m * b
(m + n) * a == m * a + n * a
(m * n) * a == m * (n * a)
for all elements m and n in the class *Ring*, all elements a and b in this class. It may be proved that the following theorem is true:
(m * a).coordinates () [i] == m * a.coordinates () [i]
for all $0 \leq i <$ module::dimension ().
Notify : This is a *friend* function.
Example : Ring r ;

```
module m, n ;
m = r * n ;
```

Signature : module operator - ( ) const ;
Preconditions : None.
Result : Returns an element n with the property that: $m + n = module < Ring > ()$, where m is the calling object. It may be proved that the following theorem is true:

```
( - m ) .coordinates () [ i ] == - m.coordinates () [ i ]
```

for all $0 \leq i <$ module::dimension ().
Notify : May be regarded as *unary minus* in standard arithmetic notation.
Example :

```
module m ;
m = - m ;
```

Signature : module operator + (const module & m) const ;
Preconditions : None.
Result : This member function has the following properties:

$$a + b == b + a$$
$$a + (b + c) == (a + b) + c$$

for all elements a,b and c in this class. It may be proved that the following theorem is true:

```
( a+b).coordinates() [ i ] == a.coordinates() [ i ] + b.coordinates() [ i ]
```

for all $0 \leq i <$ module::dimension ().
Notify : May be regarded as *addition* in standard arithmetic notation.
Example :

```
module m1, m2, m3 ;
m1 = m2 + m3 ;
```

Signature : static array<module> basis ( ) ;
Preconditions : None.
Result : Returns basis elements in the class.
Notify : The function is declared as static function. This implies that all elements in the class have equal basis.
Example :

```
module tmp, res ;
for ( int i = 0 ; i ¡ module::dimension () ; ++ i )
    res += tmp.coordinates () [i] * module::basis ( )[i] ;
if ( ! ( res == tmp ) )
    cout << "Error !" << endl ;
```

5.6.2 Observer functions

Signature : array<Ring> coordinates ( ) const ;

Preconditions : None.
Result : This function returns the coordinate representation of the calling object.
Notify : The length of the array is equal to the number of dimensions in the class.
Example :
```
array<Ring> arr ;
module m ;
arr = m.coordinates () ;
```

Signature : static int dimension ( ) ;
Preconditions : None.
Result : Returns the number of lineary independent basis elements in the class.
Notify : The function is declared as static function. This implies that all elements in the class have equal dimension.
Example :
```
cout << module::dimension () << endl ;
```

5.6.3 Composed functions

Signature : module operator - (const module & m) const ;
Preconditions : None.
Result : Returns the value of the expression $n + (-m)$, where n is the calling object.
Example :
```
module m1, m2 ;
if ( ! ( m1 - m2 == m1 + ( - m2 ) ) )
    cout << "Something is very wrong !" << endl ;
```

Signature : module & operator -= (const module & m) ;
Preconditions : None.
Result : This function is equal to: $n = n - m$, where n is the calling object.
Example :
```
module m1, m2 ;
m1 -= m2 ;
```

Signature : module & operator += (const module & m) ;
Preconditions : None.
Result : This function is equal to: $n = n + m$, where n is the calling object.
Example :
```
module m1, m2 ;
m1 += m2 ;
```

Signature : module & operator *= (const Ring & r) ;
Preconditions : None.
Result : This function is equal to: $m = r * m$, where m is the calling object.
Example : module m ;
Ring r ;
m *= r ;

5.7 REFERENCES

A good introduction to algebraic specification may be found in:
David A. Watt : *Programming Language Syntax and Semantics*, Prentice Hall 1991.

Most of the mathematical background will be found in:
Serge Lang: *Algebra*, Addison Wesley 1965.
Abraham, Marsden and Ratiu : *Manifolds, Tensor Analysis, and Applications*, Springer-Verlag 1983.

For a more theoretical approach, see :
M. Haveraaen, V. Madsen, H. Munthe-Kaas : *Algebraic Programming Technology for Partial Differential Equations*, Precedings from "Norsk Informatikk Konferanse 1992".
For a less theoretical approach, see :
V. Madsen: *Tensors, from specification to implementation*; Thesis for the master degree, Instittute of Informatics, University of Bergen.

5.8 FILES AND VERSIONS

Idea documentation : /tmp_mnt/Home/stud2/vmadsen/tensor/dok/module_idea.ps
Based on : /tmp_mnt/Home/stud2/vmadsen/tensor/dok/ring_idea.ps

5.9 MAINTENANCE AND SUPPORT

This class specification is documented by :
Victor Madsen
Institute of Informatics
University of Bergen
email : vmadsen@ii.uib.no

Chapter 6

Idea Documentation for Class : diffModule

6.1 NAME

Name	: *diffModule*
Version	: 1.0
Compiler	: AT&T C++ v. 3.01
Based on	: Idea documentation for class *diffRing*. Idea documentation for class *module*.
Description	: Specifies the signature and semantics for the mathematical object *differential module*.

6.2 PURPOSE

This documentation describes the syntax and semantics for the mathematical object *differential Module* in C++. There may be many C++ classes that have properties equal to a *differential Module*, and one may not give one implementation that cover them all. This documentation will only describe the least signature these classes need, and the semantics the functions in the classes must have.

6.3 DESCRIPTION

6.3.1 Differential operators on modules

A *differential operator* on a module M over a differential ring R is a linear map $D_m : M \rightarrow M$ with the following properties:

1. $\forall x, y \in M : D_m(x + y) = D_m(x) + D_m(y)$

2. $\forall x \in R, \forall y \in M : D_m(x * y) = D_r(x) * y + x * D_m(y)$

where D_r is a differential operator on R. A differential operator D_m is *trivial* if and only if D_r is a trivial differential operator.

6.3.2 Def. differential module

Let R be a differential ring. We now **define the module $D[R]$ to be the set of all differential operators** on the ring R. The basis elements of $D[R]$ is the set:

$$B = \frac{\partial}{\partial x^i} : R \rightarrow R | 0 < i \leq Dim(R)$$

where *Dim(R)* is the dimension of the differential ring and $\frac{\partial}{\partial x^i}$ is the partial differential operator for dimension number i in R. The application of an element $d \in D[R]$ on an element $e \in R$ is referred to as the *directional derivative* of e in *direction* d. The module $D[R]$ is defined to be the *differential module* over the differential ring R.

We may do this because we know that all partial differential operators on R are lineary independent. This implies that all differential operators on R may be written as a linear combination of the elements in the set B. The module properties of $D[R]$ is given as:

$$\begin{array}{lcll} \forall r \in R & : & 0(r) & == \ 0 \\ \forall D \in D[R], \forall r \in R & : & (-D)(r) & == \ -D(r) \\ \forall D1, D2 \in D[R], \forall r \in R & : & (D1 + D2)(r) & == \ D1(r) + D2(r) \\ \forall D \in D[R], \forall k, r \in R & : & (k * D)(r) & == \ k * D(r) \end{array}$$

It is easy to prove that the set $D[R]$ in fact is the set of differential operators on R. The basis is by def. differential operators, and the element $0 \in D[R]$ is the trivial differential operator for R. To prove that the generator functions (-, + , *) generates only new elements which are also differential operators is trivial.

6.3.3 Lie derivative

Lie derivative of a ring element

.

Let R be a differentiable ring, and let D[R] be the set of all differential operators on R. We define the *Lie derivative* of an element $f \in R$ in direction $w \in D[K]$, $\mathcal{L}_w(f)$, to be:

$$\mathcal{L}_w(f) = w^i * \frac{\partial}{\partial x^i}(f)$$

where w^i is coordinate number i and $\frac{\partial}{\partial x^i}$ is basis element number i of w.

One may show that this definition is in fact independent of a fixed basis B. A equivalent definition of $\mathcal{L}_w(f)$ is the function that satisfies the following axioms:

- $\mathcal{L}_{\frac{d}{dx^i}}(f) == \frac{d}{dx^i}(f)$
- $\mathcal{L}_{v+w}(f) == \mathcal{L}_v(f) + \mathcal{L}_w(f)$
- $\mathcal{L}_{k*w}(f) == k * \mathcal{L}_w(f)$

for $\forall v, w \in D[R]$, and $\forall k, f \in R$.

Lie derivative of a module element

.

Let R be a diferentiable ring, and let D[R] be the set of all differential operators on R. We define the *Lie derivative* of an element $v \in D[R]$ in direction $w \in D[K]$, $\mathcal{L}_w(v)$, to be:

$$\mathcal{L}_v(w) = [v, w]$$

where $[v, w] \in D[R]$ is defined to be the unique element that satisfies the following relation:

$$\mathcal{L}_{[v,w]}(f) = \mathcal{L}_v(\mathcal{L}_w(f)) - \mathcal{L}_w(\mathcal{L}_v(f))$$

for $\forall f \in R$. $[v, w]$ is often referred to as the *Lie-Jacobi bracket.*

One may prove that $\mathcal{L}_w(v)$ may be found like this:

$$\mathcal{L}_w(v) = (\mathcal{L}_w(v^i) - \mathcal{L}_v(w^i)) * \frac{\partial}{\partial x^i}$$

$$\Updownarrow$$

$$\mathcal{L}_w(v) = (w^j * \frac{\partial}{\partial x^j}(v^i) - v^j * \frac{\partial}{\partial x^j}(w^i)) * \frac{\partial}{\partial x^i}$$

6.3.4 Christoffel-symbols

We want to extend the definition of partial derivatives to the elements in the set $D[R]$. We may do this if we know the partial derivatives of the basis elements in $D[R]$, since:

$$\frac{\partial}{\partial x^i}(w) = \frac{\partial}{\partial x^i}(w^j * \frac{\partial}{\partial x^j})$$

$$= \frac{\partial}{\partial x^i}(w^j) * \frac{\partial}{\partial x^j} + w^j * \frac{\partial}{\partial x^i}(\frac{\partial}{\partial x^j})$$

The partial derivative of an element $e \in D[R]$ must also be an element in the set $D[R]$. Because of this we may specify the partial derivatives of the basis as a linear combination:

$$\frac{\partial}{\partial x^i}(\frac{\partial}{\partial x^j}) = \Gamma^q_{i,j} * \frac{\partial}{\partial x^q}$$

The numbers $\Gamma^q_{i,j} \in R$, $0 < q, i, j \leq d$ (where d is the dimension of R), is often referred to as an *euclidian affinity* or *Christoffel symbols*. The Christoffel symbols must be given for each differentable ring R, there are no way to calculate them with the available information in $D[R]$.

6.3.5 Partial derivatives of module elements

Let R be a differential ring, and let $D[R]$ be the differential module over R. Let $\Gamma^q_{i,j}$ be the Christoffel symbols for $D[R]$. Then the partial derivative of an element $w \in D[R]$, $\frac{\partial}{\partial x^i}(w)$ defined to be:

$$\frac{\partial}{\partial x^i}(w) = [\frac{\partial}{\partial x^i}(w^q) + w^j * \Gamma^q_{i,j}] * \frac{\partial}{\partial x^q}$$

6.4 ALGEBRAIC SPECIFICATION

We must give an algebraic specification of the Christoffel symbols over a differential ring before we can specify the differential module formally. This is because we then need the symbols:

```
specification CHRISTOFFEL
include INTEGER, BOOL
parameters
   sort        diffRing, Christoffel
   operations
      0                    :                          -> diffRing
      1                    :                          -> diffRing
      - _                  : diffRing                 -> diffRing
      _ + _                : diffRing * diffRing      -> diffRing
      _ * _                : diffRing * diffRing      -> diffRing
      numDim       ( _ )   : diffRing                 -> Integer
      partialDiff ( _,_ )  : Integer * diffRing
                           | (0<#1<=numDim(#2))       -> diffRing

      spaceDimension( _ ) : Christoffel                          -> Integer
      okIndex ( _,_,_ )   : Integer * Integer * Integer          -> Bool
```

```
      read ( _,_,_,_ )     : Christoffel * Integer * Integer
                             * Integer | okIndex (#2,#3,#4)    -> diffRing
      set  ( _,_,_,_,_ )   : Christoffel * diffRing * Integer
                             * Integer * Integer
                             | okIndex(#3,#4,#5)              -> Christoffel
specifies
   variables   d       : diffRing
               c       : Christoffel
               i, j, k,
               p, q, r : Integer
   equations
      DIFFERENTIAL-RING ( diffRing, 0, 1, - , +, *,
                          numDim, partialDiff )

      spaceDimension ( c ) ==   numDim ( d )
      okIndex  ( i, j, k ) == (  0 < i,j,k  <= spaceDimension(c) )

      read ( set ( c , d, i, j, k ), i, j ,k ) ==  d

      ( p != i /\ q != j /\ r != k ) =>
      read ( set ( c , d, p, q, r ), i, j ,k ) == read ( c, i, j ,k ) )
end specification
```

There is not much to say about the Christoffel symbols, except that they are indexed by three integers in the domain $[1, \ldots, d]$, where d is the number of dimensions of the differential ring.

After this, we may construct the following *algebraic specification* of a *differential Module over a differential Ring*:

```
specification DIFFERENTIAL-MODULE
include INTEGER, BOOL
parameters
   sort         diffRing, Christoffel, diffModule
   operations
      0                     :                        -> diffRing
      1                     :                        -> diffRing
      - _                   : diffRing               -> diffRing
      _ + _                 : diffRing * diffRing    -> diffRing
      _ * _                 : diffRing * diffRing    -> diffRing
      numDim      ( _ )     : diffRing               -> Integer
      partialDiff ( _,_ )   : Integer * diffRing
                              | (0<#1<=numDim(#2)) -> diffRing

      spaceDimension( _ ) : Christoffel                        -> Integer
      okIndex  ( _,_,_ )  : Integer * Integer * Integer        -> Bool
      read ( _,_,_,_ )    : Christoffel * Integer * Integer
                            * Integer | okIndex (#2,#3,#4)     -> diffRing
      set  ( _,_,_,_,_ )  : Christoffel * diffRing * Integer
                            * Integer * Integer
```

```
                          | okIndex(#3,#4,#5)              -> Christoffel

      0                 :                                                -> diffModule
      _ + _             : diffModule * diffModule                        -> diffModule
      - _               : diffModule                                     -> diffModule
      _ * _             : diffRing * diffModule                          -> diffModule
      dimension ( _ )   : diffModule                                     -> Integer
      coordinat ( _,_ ) : diffModule * Integer | (0<#2<=dimension(#1)) -> diffRing
      basis     ( _,_ ) : diffModule * Integer | (0<#2<=dimension(#1)) -> diffModule

      partialDiff ( _,_,_ ) : Integer * diffModule * Christoffel
                              Christoffel | (0<#1<=dimension(#2))      -> diffModule
      LieDiff ( _,_ )       : diffModule * diffRing                    -> diffRing
      LieDiff ( _,_ )       : diffModule * diffModule                  -> diffModule

uses
      Sum : Integer * Integer * Integer * Christoffel  -> diffModule
specifies
   variables   v, w, x : diffModule
               i, j, p : Integer
               c       : Christoffel
               d, k    : diffRing
   equations
      CHRISTOFFEL        ( diffRing, Christoffel,
                           0, 1, - , +, *, numDim, partialDiff,
                           spaceDimension, okIndex, read, set   )

      DIFFERENTIAL-RING  ( diffRing,
                           0, 1, - , +, *, numDim, partialDiff  )

      MODULE             ( diffRing, diffModule,
                           0, 1, - , +, *,
                           0, -, +, *, dimension, coordinat, basis )

      LieDiff (v, d + k ) == LieDiff (v, d ) + LieDiff (v, k )
      LieDiff (v, d * k ) == LieDiff (v, d ) * k +
                             d * LieDiff (v, k )

      LieDiff (v, w + x ) == LieDiff (v, w ) + LieDiff (v, x )
      LieDiff (v, d * w ) == LieDiff (v, d ) * w +
                             d * LieDiff (v, w )

      partialDiff ( i, v + w, c ) == partialDiff ( i, v, c ) + partialDiff ( i, w, c )
      partialDiff ( i, k * w, c ) == partialDiff ( i, k ) * w +
                                     k * partialDiff ( i, w, c )


      dimensjon ( v ) ==  numDim ( d )
      basis ( v, i )  ==  partialDiff ( i, _ )

      LieDiff ( basis ( v, i ), d )    == partialDiff ( i, d )
```

```
      LieDiff ( v + w          , d )    == LieDiff ( v, d ) + LieDiff ( w, d )
      LieDiff ( k * v          , d )    == k * LieDiff ( v, d )

      LieDiff ( LieDiff ( v, w ), d )   == LieDiff ( v, LieDiff ( w, d ) ) -
                                           LieDiff ( w, LieDiff ( v, d ) )

      partialDiff ( i, basis(v,j), c ) == Sum ( numDim ( d ) ,i, j, c )

      Sum ( 1, i, j , c )               == read ( c, 1 , i, j ) * basis (v, 1 )
      Sum ( p, i, j , c )               == read ( c, p , i, j ) * basis (v, p ) +
                                           Sum  (p-1,i,j,c )
end specification
```

Note the recursive use of *Sum* in the specification of 'partialDiff'. This algebraic specification is independent of any programming language, it only asserts properties (semantics) between a sort (class) and a set of functions that operates on the sort. This specification uses an array class that is not formally specified, but the semantic is obvious, and the specification of ARRAY is because of this not explicitly shown.

6.5 C++ SIGNATURE

One may probably implement many different classes in C++ that have the properties described above. It is still an advantage to use equal signature for all classes that have the properties described. The reason is that we then may implement algorithms that are reusable between a wide range of different implementations.

From the given algebraic specification of CHRISTOFFEL , we can construct a C++ class signature that every class with *Christoffel* semantics must contain:

```
class christoffel
{
   public:
       //   C++ specific operations
       christoffel                (                        )       ;
       christoffel                ( const christoffel &    )       ;
       ~christoffel               (                        )       ;
       int          operator ==   ( const christoffel &    ) const ;
       christoffel& operator =    ( const christoffel &    )       ;

       //   basic operations
static int          spaceDimension (                      )       ;
       diffRing &   operator  ()  ( const int,const int,
                                    const int               )       ;
       diffRing     operator  ()  ( const int,const int,
                                    const int               ) const ;
} ;
```

From the given algebraic specification of a DIFFERENTIAL-MODULE, we may construct a C++ class signature that every class with *differential Module over a differential Ring* semantics must contain:

```
class diffModule
{  public:

       //   C++ specific operations
       diffModule                    ( const diffModule &  )       ;
       ~diffModule                   (                     )       ;
       int               operator  == ( const diffModule &  ) const ;
       diffModule  &     operator  =  ( const diffModule &  )       ;

       //   Basic operations
       diffModule                    (                     )       ;
friend diffModule        operator  *  ( const diffRing & ,
                                        const diffModule &  )       ;
       diffModule        operator  -  (                     ) const ;
       diffModule        operator  +  ( const diffModule &  ) const ;
       array<diffRing>   coordinates  (                     ) const ;
static int               dimension    (                     )       ;
static array<diffModule> basis        (                     )       ;

       diffRing          lieDiff      ( const diffRing   &  ) const ;
       diffModule        lieDiff      ( const diffModule &  ) const ;
       diffModule        partialDiff  ( const int           ,
                                        const christoffel &  ) const ;

       //   Composed functions
       diffModule        operator  -  ( const diffModule &  ) const ;
       diffModule &      operator  -= ( const diffModule &  )       ;
       diffModule &      operator  += ( const diffModule &  )       ;
       diffModule &      operator  *= ( const diffRing   &  )       ;
} ;
```

We demand that all classes in C++ that is supposed to represent these classes, must at least have the above given class signature. Notice that this is the minimum set of functions an implementation must contain, it will cause no problem if there are implemented more member functions in the class. Typically, there must at least be some extra constructors that are dependent of the actual implementation of the class specification.

6.6 OPERATIONS Christoffel

6.6.1 Generators

Signature : christoffel ( ) ;
Preconditions : None.
Result : Constructs an object witch contains *diffRing(0)* in every position.

Notify : This is the *default constructor* for the class.
Example : christoffel r ;

Signature : diffRing &
operator () (const int i ,const int j , const int p) ;
Preconditions : $\leq i, j, p <$ *christoffel* :: *spaceDimension*().
Result : Returns a reference to the element $\Gamma^i_{j,p}$, where Γ is the calling object.
Notify : This function may be used to update the calling object.
Example : christoffel c ;
c (0, 0, 0) = diffRing (2) ;

6.6.2 Observers

Signature : diffRing operator () (const int i ,const int j , const int p) const ;
Preconditions : $\leq i, j, p <$ *christoffel* :: *spaceDimension*().
Result : Returns the element $\Gamma^i_{j,p}$, where Γ is the calling object.
Notify : This function may not be used to update the calling object.
Example : christoffel c ;
diffRing r = c (0, 0, 0) ;

Signature : static int spaceDimension ( ) ;
Preconditions : None.
Result : Returns the number of dimensions of the space that the objects exists in.
Notify : This is equal to diffRing::numDimensions ().
Example : cout << christoffel::spaceDimension () << endl ;

6.7 OPERATIONS diffModule

This class specification contains all member functions from the class specification *Module over Ring*. In addition this class contains:

6.7.1 Generators

Signature : diffModule lieDiff (const diffModule & r) const ;
Preconditions : None.
Result : Returns the Lie derivative of r in the direction of the calling object.

Notify : See the above given definition.
Example : diffModule a, b, c ;
a = b.lieDiff (c) ;

Signature : diffModule
partialDiff (const int i , const christoffel & c) const ;
Preconditions : $0 \leq i <$ *diffModule::dimension ()*.
Result : Returns the i'th partial derivative of the calling object with regard to c.
Notify : See the above given definition.
Example : diffModule a, b ;
christoffel c ;
b = a.partialDiff (0, c) ;

6.7.2 Observers

Signature : diffRing lieDiff (const diffRing & r) const ;
Preconditions : None.
Result : Returns the Lie derivative of r in the direction of the calling object.
Notify : See the above given definition.
Example : diffRing r, s ;
diffModule m ;
s = m.lieDiff (r) ;

6.8 REFERENCES

A good introduction to algebraic specification may be found in:
David A. Watt : *Programming Language Syntax and Semantics*, Prentice Hall 1991.

Most of the mathematical background will be found in:
Serge Lang: *Algebra*, Addison Wesley 1965.
Abraham, Marsden and Ratiu : *Manifolds, Tensor Analysis, and Applications*, Springer-Verlag 1983.

For a more theoretical approach, see :
M. Haveraaen, V. Madsen, H. Munthe-Kaas : *Algebraic Programming Technology for Partial Differential Equations*, Precedings from "Norsk Informatikk Konferanse 1992".
For a less theoretical approach, see :
V. Madsen: *Tensors, from specification to implementation*; Thesis for the master degree, Institute of Informatics, University of Bergen.

6.9 FILES AND VERSIONS

Idea documentation : /tmp_mnt/Home/stud2/vmadsen/tensor/dok/diffmodule_idea.ps
Based on : /tmp_mnt/Home/stud2/vmadsen/tensor/dok/diffring_idea.ps
/tmp_mnt/Home/stud2/vmadsen/tensor/dok/module_idea.ps

6.10 MAINTENANCE AND SUPPORT

This class specification is documented by :
Victor Madsen
Institute of Informatics
University of Bergen
email : vmadsen@ii.uib.no

Chapter 7

Idea Documentation for Class : coModule

7.1 NAME

Name	: *coModule*
Version	: 1.0
Compiler	: AT&T C++ v. 3.01
Based on	: Idea documentation for class *Ring*. Idea documentation for class *Module*.
Description	: Specifies the signature and semantics for the mathematical object *comodule*. This object is also often referred to as the *dual module* of a module.

7.2 PURPOSE

This documentation describes the syntax and semantics for the mathematical object *coModule* in C++. There may be many C++ classes that has properties equal to a *coModule*, and one may not give one implementation that cover them all. This documentation will only describe the least signature these classes need, and the semantics the functions in the classes must have.

7.3 DESCRIPTION

7.3.1 Linear Maps

Let M and N be two modules over the ring R. A function:

$$A : E \rightarrow F$$

is a *linear map* if the following axioms are satisfied:

L1 : $\forall x, y \in M : A(x+y) = A(x) + A(y)$

L2 : $\forall c \in R, \forall x \in M : A(c * x) = c * A(x)$

The set of all linear maps from a module M to a module N is denoted *L(M,N)*.

7.3.2 Comodules

Let M be a module over the ring R. The *comodule* of M is the set of all linear maps from M to R, $L(M, R)$, and is denoted M^*.

7.3.3 Theorem

Let M be a module over the ring R, with dimension d. Then M^* also have dimension d. If $B = \{b_1, \ldots, b_d\}$ is a basis for M, and a set $F = \{f^1, \ldots, f^d\} \subseteq M^*\}$ have the property:

$$f^i(b_j) = \delta^i_j = \begin{cases} 0 & \text{if } i <> j \\ 1 & \text{if } i = j \end{cases}$$

then F is a basis for M^*, and F is referred to as the *dual basis* of M.

It may further be proved that the following axioms are true:

L3 : $\forall A, B \in M^*, \forall m, n \in M : (A+B)(m) = A(m) + B(m)$

L4 : $\forall A \in M^*, \forall m \in M \; \forall k \in R : (k * A)(m) = k * A(m)$

and that M^* are also a module over the ring R.

7.3.4 Inner product

Let M be a module over the ring R, with dimension d and basis $B = \{b_1, \ldots, b_d\}$. The application of an element $f \in M^*$ on an element $e \in M$ is often denoted as an *inner product* between e and f, and the result is written as $f(e)$. The result of an inner product may always be expressed as a function of the coordinates of e and f alone. This is because:

$$\begin{aligned} f(e) &= \sum_{i=1}^{d} \sum_{j=1}^{d} f_i * b'^i(e^j * b_j) \\ &= \sum_{i=1}^{d} \sum_{j=1}^{d} f_i * e^j * b'^i(b_j) \\ &= \sum_{i=1}^{d} \sum_{j=1}^{d} f_i * e^j * \delta^i_j \\ &= \sum_{i=1}^{d} f_i * e^i \\ &= < f, e > \end{aligned}$$

where $B' = \{b'^1, \ldots, b'^d\}$ is a basis for M^*.

7.4 ALGEBRAIC SPECIFICATION

We may now construct the following *algebraic specification* of a *comodule* to a *Module over Ring*:

```
specification KOMODULE
include BOOL, INTEGER
parameters
   sort          Ring, Module, coModule
   operations
      0                 :                                                -> Ring
      1                 :                                                -> Ring
      - _               : Ring                                           -> Ring
      _ + _             : Ring * Ring                                    -> Ring
      _ * _             : Ring * Ring                                    -> Ring

      0                 :                                                -> Module
      _ + _             : Module * Module                                -> Module
      - _               : Module                                         -> Module
      _ * _             : Ring * Module                                  -> Module
      dimension ( _ )   : Module                                         -> Integer
      coordinat ( _,_ ) : Module * Integer | (0<#2<=dimension(#1))       -> Ring
      basis     ( _,_ ) : Module * Integer | (0<#2<=dimension(#1))       -> Module

      0                 :                                                -> coModule
      _ + _             : coModule * coModule                            -> coModule
      - _               : coModule                                       -> coModule
      _ * _             : Ring * coModule                                -> coModule
      _ ( _ )           : coModule * Module                              -> Ring
      dimension ( _ )   : coModule                                       -> Integer
      coordinat ( _,_ ) : coModule * Integer | (0<#2<=dimension(#1))     -> Ring
      basis     ( _,_ ) : coModule * Integer | (0<#2<=dimension(#1))     -> coModule
      _ ( _ )           : coModule * Module                              -> Ring
specifies
   variables  m, n : Module
              a, b : coModule
              k    : Ring
              i, j : Integer
   equations
      RING     ( Ring, 0, 1, -, +, *   )
      MODULE   ( Ring, Module,
                 0, 1, -, +, * ,
                 0, +, -, *, dimension, coordinat, basis  )
      MODULE   ( Ring, coModule,
                 0, 1, -, +, * ,
                 0, +, -, *, dimension, coordinat, basis  )

      a ( m + n )       ==   a ( m ) + a ( n )      \\   L1
      a ( k * m )       ==   k * a ( m )            \\   L2
      0 ( m     )       ==   0                      \\   L3
```

```
      ( a + b ) ( m )   ==   a ( m ) + b ( m )     \\   L4
      ( - a   ) ( m )   ==   - ( a ( m ) )         \\   L5
      ( k * a ) ( m )   ==   k * a ( m )           \\   L6

      ( i  = j ) =>  basis ( a, i ) ( basis ( m, j ) ) == 1
      ( i <> j ) =>  basis ( a, i ) ( basis ( m, j ) ) == 0
end specification
```

This algebraic specification is independent of any programming language, it only asserts properties (semantics) between a sort (class) and a set of functions that operates on the sort. Note the reuse of the specifications RING and MODULE.

7.5 C++ SIGNATURE

One may probably implement many different classes in C++ that have the properties described above. It is still an advantage to use equal signature for all classes that have the properties described. The reason is that we then may implement algorithms that are reusable between a wide range of different implementations of the class *coModule*.

From the given algebraic specification, we may construct a C++ class signature that every class with *coModule* semantics must contain:

```
class coModule
{  public:

       //   C++ specific operations
       coModule                    ( const coModule &  )       ;
       ~coModule                   (                   )       ;
       int             operator  == ( const coModule &  ) const ;
       coModule  &     operator  =  ( const coModule &  )       ;

       //   Basic operations
       coModule                    (                   )       ;

friend coModule        operator  *  ( const Ring & ,
                                      const coModule &  )       ;
       coModule        operator  -  (                   ) const ;
       coModule        operator  +  ( const coModule &  ) const ;
       array<Ring>     coordinates  (                   ) const ;
static int             dimension    (                   )       ;
static array<coModule> basis        (                   )       ;

       //   Inner product
       Ring            operator ()  ( const Module &    ) const ;

       //   Composed functions
       coModule        operator  -  ( const coModule &  ) const ;
```

```
        coModule &      operator  -= ( const coModule &  )      ;
        coModule &      operator  += ( const coModule &  )      ;
        coModule &      operator  *= ( const Ring     &  )      ;
} ;
```

We demand that all classes in C++ that is supposed to represent a em coModule, must at least have the above given class signature. Notice that this is the minimum set of functions an implementation must contain, it will cause no problem if there are implemented more member functions in the class. Typically, there must at least be some extra constructors that are dependent of the actual implementation of the *coModule*. The signature contains only one extra function in addition to the signature for a *module*, and it is the overloaded operator (), which represent the inner product between a comodule element and a module element. The composed functions have the usual associated semantics for the operators. Note that the class *Ring* must satisfy the specification for a ring, and the class *Module* must satisfy the specification for a module.

7.6 OPERATIONS

This class specifiaction contain all member functions from the class specification *Module over Ring*. In addition the specification contains the operation:

7.6.1 Observer function

Signature : Ring operator () (const Module & m) const ;
Preconditions : None.
Result : This function satisfies the following properties:

$$a(m+n) == a(m) + a(n)$$
$$a(k*m) == k*a(m)$$
$$(a+b)(m) == a(m) + b(m)$$
$$(k*a)(m) == k*a(m)$$

for all elements m, n of type *Module*, all a, b of type *coModule* and all k of type *Ring*. The result of this function may be expressed as:

$$m\ (\ a\ ) == \sum_{i=0}^{dimension()-1} (m.coordinates()[i]\ *\ a.coordinates()[i])$$

Notify : This function is often referred to as *inner product.*
Example : Ring r ;
Module m ;
coModule coMod ;
r = coMod (m) ;

7.7 REFERENCES

A good introduction to algebraic specification may be found in:
David A. Watt : *Programming Language Syntax and Semantics*, Prentice Hall 1991.

Most of the mathematical background will be found in:
Serge Lang: *Algebra*, Addison Wesley 1965.
Abraham, Marsden and Ratiu : *Manifolds, Tensor Analysis, and Applications*, Springer-Verlag 1983.

For a more theoretical approach, see :
M. Haveraaen, V. Madsen, H. Munthe-Kaas : *Algebraic Programming Technology for Partial Differential Equations*, Precedings from "Norsk Informatikk Konferanse 1992".

7.8 FILES AND VERSIONS

Idea documentation : /tmp_mnt/Home/stud2/vmadsen/tensor/dok/comodul_idea.ps
Based on : /tmp_mnt/Home/stud2/vmadsen/tensor/dok/modul_idea.ps
/tmp_mnt/Home/stud2/vmadsen/tensor/dok/ring_idea.ps

7.9 MAINTENANCE AND SUPPORT

This class specification is documented by :
Victor Madsen
Institute of Informatics
University of Bergen
email : vmadsen@ii.uib.no

Chapter 8

Idea Documentation for Class : diffCoModule

8.1 NAME

Name	: *diffCoModule*
Version	: 1.0
Compiler	: AT&T C++ v. 3.01
Based on	: Idea documentation for class *diffRing*. Idea documentation for class *diffModule*.
Description	: Specifies the signature and semantics for the mathematical object *differential comodule*.

8.2 PURPOSE

This documentation describes the syntax and semantics for the mathematical object *differential comodule* in C++. There may be many C++ classes that has properties equal to a *differential comodule*, and one may not give one implementation that cover them all. This documentation will only describe the least signature these classes need, and the semantics the functions in the classes must have.

8.3 DESCRIPTION

8.3.1 Def. differential operators on comodules

Let R be a ring, and let M be a module over R. A *differential operator* on the dual module M* is a linear map $D_k : M^* \rightarrow M^*$ with the properties:

1. $\forall x, y \in M^* : D_k(x + y) = D_k(x) + D_k(y)$

2. $\forall x \in R, \forall y \in M^* : D_k(x * y) = D_r(x) * y + x * D_k(y)$
3. $\forall y \in M, \forall z \in M^* : D_r(z(y)) = (D_k(z))(y) + z(D_m(y))$

where D_r is a differential operator on the ring R and D_m is a differential operator on the module M. D_k is a trivial operator if and only if D_r is a trivial operator.

8.3.2 Theorem

Let R be a ring, and let M be a module over R with basis $B = \{b_1, \ldots, b_d\}$. Let D_r be a differential operator on R and D_m a differential operator on M. Then there always exists a dual module M* with basis $B' = \{b^1, \ldots, b^d\}$ and a differential operator D_k on M*. We know that D_m may be represented with a set of elements e_i^j:

$$D_m(b_i) = e_i^j * b_j$$

and may by the definition of differential operators on comodules derive:

$$D_k(b^i) = -e_j^i * b^j$$

and:

$$\forall x \in M^* : D_k(x) = [D_r(x_j) - x_i * e_j^i] * b^j$$

where x_j are the coordinates of x.

This is a very important theorem, since it states that if D_r and D_m is given, there exists one fixed corresponding differential operator D_k. The theorem also show how D_k may be found. We may use this theorem to extend the the definition of the *Lie derivative* and *partial derivative* given in *diffModule* to the dual module $M[R]^*$.

8.3.3 Lie derivative

Let R be a differential ring, and let $D[R]$ be the differential module over R. The Lie derivative of an element $m \in D[R]^*$ in direction $w \in D[R]$ is found by calculating the set E of ring elements:

$$E = \{e_i^j | \mathcal{L}_w(\frac{\partial}{\partial x^i}) = e_i^j * \frac{\partial}{\partial x^j}\}$$

where $\frac{\partial}{\partial x^i}$ is basis element number i in $D[R]$. We know from the definition of the lie derivative how to find the derivative of a module element, and get:

$$\begin{array}{lcl} \mathcal{L}_w(\frac{\partial}{\partial x^i}) & = & -\frac{\partial}{\partial x^i}(w^j) * \frac{\partial}{\partial x^j} \\ & \Updownarrow & \\ e_j^i & = & -\frac{\partial}{\partial x^j}(w^i) \end{array}$$

By the previous theorem we may derive:

$$\begin{array}{rcl} D_k(w) & = & [D_r(w_j) - w_i * e^i_j] * dx^j \\ & \Updownarrow & \\ \mathcal{L}_v(w) & = & [\mathcal{L}_v(w_j) - w_i * e^i_j] * dx^j \\ & \Updownarrow & \\ \mathcal{L}_v(w) & = & [v^i * \frac{\partial}{\partial x^i}(w_j) + w_i * \frac{\partial}{\partial x^j}(v^i)] * dx^j \end{array}$$

for $\forall v \in D[R], \forall w \in D[R]^*$, where dx^j is basis element number j in $D[R]^*$.

8.3.4 Partial derivatives of comodule elements

Let R be a differential ring, and let $D[R]$ be the corresponding differential module. Then the set $B = \{\frac{\partial}{\partial x^1}, \ldots, \frac{\partial}{\partial x^d}\}$ is the basis for $D[R]$. Let $\Gamma^q_{i,j}$ be some suitable Christoffel symbols. We know that the partial derivative of an element $w \in D[R]$ is found like this:

$$\frac{\partial}{\partial x^i}(w) = [\frac{\partial}{\partial x^i}(w^q) + w^j * \Gamma^q_{i,j}] * \frac{\partial}{\partial x^q}$$

By the above given theorem we know that we may find the partial derivative of an element $v \in D[R]^*$ like this:

$$\frac{\partial}{\partial x^i}(v) = [\frac{\partial}{\partial x^i}(v_q) - w^j * \Gamma^j_{i,q}] * dx^q$$

8.4 ALGEBRAIC SPECIFICATION

After this, we may construct the following *algebraic specification* of a *differential comodule over a differential Ring*:

```
specification DIFFERENTIAL-COMODULE
include INTEGER,BOOL
parameters
   sort         diffRing, Christoffel, diffModule , diffCoModule
   operations
      0                      :                          -> diffRing
      1                      :                          -> diffRing
      - _                    : diffRing                 -> diffRing
      _ + _                  : diffRing * diffRing      -> diffRing
      _ * _                  : diffRing * diffRing      -> diffRing
      numDim       ( _ )     : diffRing                 -> Integer
      partialDiff ( _,_ )    : Integer * diffRing
                               | (0<#1<=numDim(#2)) -> diffRing

      spaceDimension( _ ) : Christoffel                          -> Integer
      okIndex  ( _,_,_ )  : Integer * Integer * Integer          -> Bool
      read ( _,_,_,_ )    : Christoffel * Integer * Integer
                            * Integer | okIndex (#2,#3,#4)       -> diffRing
```

```
      set  ( _,_,_,_,_ )  : Christoffel * diffRing * Integer
                            * Integer * Integer
                            | okIndex(#3,#4,#5)              -> Christoffel

      0                 :                                                -> diffModule
      _ + _             : diffModule * diffModule                        -> diffModule
      - _               : diffModule                                     -> diffModule
      _ * _             : diffRing * diffModule                          -> diffModule
      dimension ( _ )   : diffModule                                     -> Integer
      coordinat ( _,_ ) : diffModule * Integer | (0<#2<=dimension(#1)) -> diffRing
      basis     ( _,_ ) : diffModule * Integer | (0<#2<=dimension(#1)) -> diffModule

      partialDiff ( _,_,_ ) : Integer * diffModule * Christoffel
                              Christoffel | (0<#1<=dimension(#2))      -> diffModule
      LieDiff ( _,_ )       : diffModule * diffRing                    -> diffRing
      LieDiff ( _,_ )       : diffModule * diffModule                  -> diffModule


      0                     :                                        -> diffCoModule
      _ + _                 : diffCoModule * diffCoModule            -> diffCoModule
      - _                   : diffCoModule                           -> diffCoModule
      _ * _                 : diffRing * diffCoModule                -> diffCoModule
      dimension ( _ )       : diffCoModule                           -> Integer
      coordinat ( _,_ )     : diffCoModule * Integer
                              | (0<#2<=dimension(#1))                -> diffRing
      basis     ( _,_ )     : diffCoModule * Integer
                              | (0<#2<=dimension(#1))                -> diffCoModule
      _ ( _ )               : diffCoModule * diffModule              -> diffRing

      partialDiff ( _,_,_ ) : Integer * diffCoModule *
                              Christoffel | (0<#1<=dimension(#2))    -> diffCoModule
      LieDiff ( _,_ )       : diffModule * diffCoModule              -> diffCoModule
specifies
   variables   c    : Christoffel
               m, n : diffModule
               v, w : diffCoModule
               k    : diffRing
               i    : Integer
   equations
      DIFFERENTIAL-MODULE   ( diffRing, christoffel, diffModule,
                              0, 1, - , +, *, numDim, partialDiff,
                              spaceDimension, okIndex, read, set ,
                              0, -, +, *, dimension, coordinat , basis,
                              partialDiff, LieDiff, LieDiff           )

      COMODULE              ( diffRing, diffModul, diffCoModule,
                              0, 1, - , +, *,
                              0, -, +, *, dimension, coordinat, basis,
                              0, -, +, *, dimension, coordinat, basis, _(_) )

      partialDiff ( i, v + w, c ) == partialDiff ( i, v , c ) +
```

```
                                  partialDiff ( i, w, c )
      partialDiff ( i, k * w, c ) == partialDiff ( i, k ) * w +
                                     k * partialDiff ( i, w, c )
      partialDiff ( i, w ( m ) )  == partialDiff ( i, w, c ) ( m ) +
                                     w ( partialDiff ( i, m , c ) )

      LieDiff ( m, v + w   )      == LieDiff ( m, v ) + LieDiff ( m, w )
      LieDiff ( m, k * w   )      == LieDiff ( m, k ) * w + k * LieDiff ( m, w )
      LieDiff ( m, w ( n ) )      == LieDiff ( m, w ) ( n ) +
                                     w ( LieDiff ( m, n ) )
end specification
```

This algebraic specification is independent of any programming language, it only asserts properties (semantics) between a sort (class) and a set of functions that operates on the sort. This specification uses an array class that is not formally specified, but the semantic is obvious, and the specification of ARRAY is because of this not explicitly shown. Also note that all differential operators on differential comodules is defined uniquely by the corresponding operators on the differentable ring and differentable module.

8.5 C++ SIGNATURE

One can probably implement many different classes in C++ that have the properties described above. It is still an advantage to use equal signature for all classes that have the properties described. The reason is that we then may implement algorithms that are reusable between a wide range of different implementations.

From the given algebraic specification of a DIFFERENTIAL-COMODULE, we may construct a C++ class signature that every class with *differential coModule over a differential Ring* semantics must contain:

```
class diffCoModule
{  public:

       //   C++ specific operations
       diffCoModule                    ( const diffCoModule &  )       ;
       ~diffCoModule                   (                       )       ;
       int                operator  == ( const diffCoModule &  ) const ;
       diffCoModule  &    operator  =  ( const diffCoModule &  )       ;

       //   Basic operations
       diffCoModule                    (                       )       ;
friend diffCoModule       operator  *  ( const diffRing & ,
                                         const diffCoModule &  )       ;
       diffCoModule       operator  -  (                       ) const ;
       diffCoModule       operator  +  ( const diffCoModule &  ) const ;
       array<diffRing>    coordinates  (                       ) const ;
```

```
static int                  dimension    (                          )       ;
static array<diffCoModule> basis        (                          )       ;
       diffRing             operator ()  ( const diffModule   &  ) const ;

       diffCoModule         lieDiff      ( const diffModule   &  ) const ;
       diffCoModule         partialDiff  ( const int          ,
                                           const christoffel  &  ) const ;

       //   Composed functions
       diffCoModule         operator  -  ( const diffCoModule &  ) const ;
       diffCoModule &       operator  -= ( const diffCoModule &  )       ;
       diffCoModule &       operator  += ( const diffCoModule &  )       ;
       diffCoModule &       operator  *= ( const diffRing     &  )       ;
} ;
```

We demand that all classes in C++ that is supposed to represent these classes, must at least have the above given class signature. Notice that this is the minimum set of functions an implementation must contain, it will cause no problem if there are implemented more member functions in the class. Typically, there must at least be some extra constructors that are dependent of the actual implementation of the class specification.

8.6 OPERATIONS

This class specification contain all member functions from the class specification *coModule over Ring*. In addition this class contains:

8.6.1 Generators

Signature : diffCoModule lieDiff (const diffModule & m) const ;
Preconditions : None.
Result : Returns the Lie derivative of the calling object in the direction of m.
Notify : See the above given definition. Also note that this function returns the Lie derivative of the **calling** object.
Example : diffCoModule a, b;
diffModule c ;
a = b.lieDiff (c) ;

Signature : diffCoModule
partialDiff (const int i , const christoffel & c) const ;
Preconditions : $0 \leq i <$ *diffCoModule::dimension ()*.
Result : Returns the i'th partial derivative of the calling object with regard to c.
Notify : See the above given definition.
Example : diffCoModule a, b ;

```
christoffel c ;
b = a.partialDiff ( 0, c ) ;
```

8.7 REFERENCES

A good introduction to algebraic specification may be found in:
David A. Watt : *Programming Language Syntax and Semantics*, Prentice Hall 1991.

Most of the mathematical background will be found in:
Serge Lang: *Algebra*, Addison Wesley 1965.
Abraham, Marsden and Ratiu : *Manifolds, Tensor Analysis, and Applications*, Springer-Verlag 1983.

For a more theoretical approach, see :
M. Haveraaen, V. Madsen, H. Munthe-Kaas : *Algebraic Programming Technology for Partial Differential Equations*, Precedings from "Norsk Informatikk Konferanse 1992".

For a less theoretical approach, see :
V. Madsen: *Tensors, from specification to implementation*; Thesis for the master degree, Institute of Informatics, University of Bergen.

8.8 FILES AND VERSIONS

Idea documentation : /tmp_mnt/Home/stud2/vmadsen/tensor/dok/diffcomodule_idea.ps
Based on : /tmp_mnt/Home/stud2/vmadsen/tensor/dok/comodule_idea.ps
tmp_mnt/Home/stud2/vmadsen/tensor/dok/diffmodule_idea.ps
tmp_mnt/Home/stud2/vmadsen/tensor/dok/diffring_idea.ps

8.9 MAINTENANCE AND SUPPORT

This class specification is documented by :
Victor Madsen
Institute of Informatics
University of Bergen
email : vmadsen@ii.uib.no

Chapter 9

Idea Documentation for Class : tensor

9.1 NAME

Name	: *tensor*
Version	: 1.0
Compiler	: AT&T C++ v. 3.01
Based on	: Idea documentation for class *Ring.* Idea documentation for class *module.* Idea documentation for class *coModule.*
Description	: Specifies the signature and semantics for the mathematical object *tensor.*

9.2 PURPOSE

This documentation describes the syntax and semantics for the mathematical object *tensor* in C++. There may be many C++ classes that has properties equal to a *tensor*, and one may not give one implementation that cover them all. This documentation will only describe the least signature these classes need, and the semantics the functions in the classes must have.

9.3 DESCRIPTION

9.3.1 Cartesian product

Let the sets $E_1, E_2, ..., E_k$ be modules over the ring R, where module E_i has the basis B_i. We define the *cartesian product* F to be:

$$F = E_1 \times E_2 \times \ldots \times E_k$$

where the F consists of the elements $(v_1, \ldots, v_k)$ for all $v_i \in V_i$.

$$F = \{(v_1, \ldots, v_k) | v_i \in V_i \wedge 0 < i \leq k\}$$

F is also a module over R, and the module properties is constructed in the following way:

- $0 = (0_1, \ldots, 0_k)$, where $0_i + v_i = v_i$
- $(v_1, \ldots, v_k) + (w_1, \ldots, w_k) = (v_1 + w_1, \ldots, v_k + w_k)$
- $-(v_1, \ldots, v_k) = (-v_1, \ldots, -v_k)$
- $c * (v_1, \ldots, v_k) = (c * v_1, \ldots, c * v_k)$
- The dimension of F is defined to be : $dim(E_1) * \ldots * dim(E_k)$.
- The basis B is:

$$B = \{(b_{i1}, \ldots, b_{ik}) | b_{ij} \in B_k \wedge 0 < i \leq dimension(E_j)\}$$

- The coordinates of an element $(v_1, \ldots, v_k) \in F$ is the sequence:

$$coord(v_1), \cdots, koord(v_k)$$

  where $coord(v_i)$ are the coordinates of the element v_i in the module E_i.

The dual module F^* to F is the set of all linear maps from F into R. It is easy to prove that F^* is isomorfic with the cartesian product $(E_1^*) \times \ldots \times (E_k^*)$, where E_i^* is the dual module to E_i.

9.3.2 Multilinear maps

Let $E_1, \ldots, E_k, F$ be modules over the ring R. A function:

$$A : E_1 \times \ldots \times E_k \rightarrow F$$

is a *k-multilinear map* if A is a multilinear map in each separate argument. The set of all multilinear maps from $E1, \ldots, E_k$ into F is denoted $L(E_1, \ldots, E_k; F)$. If $E_i = E$ the map is denoted $L^k(E; F)$.

9.3.3 Def. tensor

Let E be a module over the ring R, and let E^* be a dual module to E. We define the set $T_s^r(E)$ to be:

$$T_s^r(E) = L((E^*)^r \times E^s; K)$$

The elements in the set $T_s^r(E)$ is referred to as tensors of *type* (r, s). The elements in $T_s^r(E)$ are *contravariant* in r arguments and *covariant* in s arguments.

9.3.4 Theorem

Let E be a module over the ring R. Then there exists an isomorphism between the ring R and $T_0^0(E)$.

This is because a tensor of type (0,0) is a linear map that takes no arguments , and returns a element in the set R (i.e. is a constant).

Theorem

Let E be a module over the ring R. Then there exists an isomorphism between the dual module E^* and $T_1^0(E)$.

This is because a tensor of type (0,1) is a linear map that takes one argument from the set M and returns a element in the set R. $T_1^0(E)$ is then isomorphic to $L(M, R)$ which is by definition equal to E^*.

9.3.5 Theorem

Let E be a module over the ring R. Then there exists an isomorphism between the dual module E and $T_0^1(E)$.

This is because a tensor of type (1,0) is a linear map that takes one argument from the set M^* and returns a element in the set R. $T_0^1(E)$ is then isomorphic to $L(M^*, R)$ which is equal to E^{**} (the dual module of the dual module to E). One may prove that for finite dimensional modules with a fixed basis, E^{**} is equal to E.

9.3.6 Tensor product

The *tensor product* is a function $\_ * \_ : T_n^m(E) * T_p^o(E) \rightarrow T_{n+p}^{m+o}(E)$, where the coordinates of the tensor product is found by the formula:

$$(t_1 * t_2)^{i_1,\ldots,i_{r_1},m_1,\ldots,m_{r_2}}_{j_1,\ldots,j_{s_1},n_1,\ldots,n_{s_2}} = (t_1)^{i_1,\ldots,i_{r_1}}_{j_1,\ldots,j_{s_1}} * (t_2)^{m_1,\ldots,m_{r_2}}_{n_1,\ldots,n_{s_2}}$$

(there is a implicit summation over all repeated indexes). Note that the *outer product* between two elements in a module E in fact is a special case of a tensor product.

9.3.7 Inner product

Let E be a module over the ring R. The function $\_(\_) : T^{r_1}_{s_1}(E) * T^{r_2}_{s_2}(E) \rightarrow T^{r_1-s_2}_{s_1-r_2}(E)$ is an *inner product* where the coordinates of the result is found by the formula:

$$(t_1(t_2))^{i_1,\ldots,i_m}_{j_1,\ldots,j_n} = (t_1)^{i_1,\ldots,i_m,k_1,\ldots,k_{s_2}}_{j_1,\ldots,j_n,l_1,\ldots,l_{r_2}} * (t_2)^{l_1,\ldots,l_{r_2}}_{k_1,\ldots,k_{s_2}}$$

9.3.8 Contraction

Let E be a module over the ring R. Contraction is a function $k : T^r_s(E) \rightarrow T^{r-1}_{s-1}(E)$, where the coordinates of the result of a contraction of a tensor $t \in T^r_s(E)$, $(0 < r, s)$ is found by the formula:

$$k(t)^{i_1,\ldots,i_{r-1}}_{j_1,\ldots,j_{s-1}} = t^{i_1,\ldots,i_{r-1},m}_{j_1,\ldots,j_{s-1},m}$$

9.3.9 Kronecker tensor

Let E be a module over the ring R.The tensor δ is of type $(1,1)$, and the coordinates of this tensor may be expressed as:

$$\delta^i_j = \begin{cases} 0 & \text{if } i <> j \\ 1 & \text{if } i = j \end{cases}$$

This tensor is known as the *Kronecker tensor.*

9.4 ALGEBRAIC SPECIFICATION

The above given functions may be formalized with the aid of algebraic specification:

```
specification TENSOR
include BOOL, INTEGER
parameters
   sort          Ring, Module, coModule, Tensor
   operations
      0                 :                                              -> Ring
      1                 :                                              -> Ring
      - _               : Ring                                         -> Ring
      _ + _             : Ring * Ring                                  -> Ring
      _ * _             : Ring * Ring                                  -> Ring

      0                 :                                              -> Module
      _ + _             : Module * Module                              -> Module
      - _               : Module                                       -> Module
      _ * _             : Ring * Module                                -> Module
      dimension ( _ )   : Module                                       -> Integer
      coordinat ( _,_ ) : Module * Integer | (0<#2<=dimension(#1))     -> Ring
      basis     ( _,_ ) : Module * Integer | (0<#2<=dimension(#1))     -> Module
```

```
      0                 :                                              -> coModule
      _ + _             : coModule * coModule                          -> coModule
      - _               : coModule                                     -> coModule
      _ * _             : Ring * coModule                              -> coModule
      _ ( _ )           : coModule * Modul                             -> Ring
      dimension ( _ )   : coModule                                     -> Integer
      coordinat ( _,_ ) : coModule * Integer | (0<#2<=dimension(#1)) -> Ring
      basis     ( _,_ ) : coModule * Integer | (0<#2<=dimension(#1)) -> coModule
      _ ( _ )           : coModule * Modul                             -> Ring

      //   observers
      ring         ( _ ) : Tensor | isRing    (#1)                   -> Ring
      module       ( _ ) : Tensor | isModule  (#1)                   -> Module
      comodule     ( _ ) : Tensor | isCoModule(#1)                   -> coModule
      numCoModules ( _ ) : Tensor                                    -> Integer
      numModules   ( _ ) : Tensor                                    -> Integer

      //   generators
      tensor ( _ )       : Ring                                      -> Tensor
      tensor ( _ )       : Module                                    -> Tensor
      tensor ( _ )       : coModule                                  -> Tensor

      0 ( _ )            : Tensor                                    -> Tensor
      _ + _              : Tensor * Tensor                           -> Tensor
      - _                : Tensor                                    -> Tensor
      _ * _              : Ring * Tensor                             -> Tensor
      dimension ( _ )    : Tensor                                    -> Integer
      coordinat ( _,_ )  : Tensor * Integer | (0<#2<=dimension(#1)) -> Ring
      basis     ( _,_ )  : Tensor * Integer | (0<#2<=dimension(#1)) -> Tensor

      _ * _              : Tensor * Tensor                           -> Tensor
      < _,_ >            : Tensor * Tensor  | okInnerProd (#1,#2)   -> Tensor
      contract    ( _ )  : Tensor           | canContract ( #1 )    -> Tensor
      Kronecker          :                                           -> Tensor

      //   Composed functions
      isRing          ( _ )    : Tensor                             -> Bool
      isModule        ( _ )    : Tensor                             -> Bool
      isCoModule      ( _ )    : Tensor                             -> Bool
      equalType       ( _,_ )  : Tensor * Tensor                    -> Bool
      okInnerProd     ( _, _ ) : Tensor * Tensor                    -> Bool
      canContract     ( _ )    : Tensor                             -> Bool
specifies
   variables   a, b, k             : Ring
               m, n                : Module
               v, w                : coModule
               t1, t2, t3, t4, t5  : Tensor
   equations
      KOMODUL    ( Ring, Module, coModule,
                   0, 1, -, +, * ,
```

```
            0, +, -, * ,dimension, coordinat, basis ,
            0, +, -, * ,dimension, coordinat, basis     )
MODUL     ( Ring,
            Tensor | equalType( #1, t1 ),            //   NB !
            0, 1, -, +, * ,
            0 ( t1 ),                                //   NB !
            +, -, *, dimension, coordinat, basis )

ring        ( tensor ( a )  ) == a
module      ( tensor ( m )  ) == m
comodule    ( tensor ( v )  ) == v

numCoModules ( tensor ( a )  ) ==  0
numModules   ( tensor ( a )  ) ==  0
numCoModules ( tensor ( m )  ) ==  1
numModules   ( tensor ( m )  ) ==  0
numCoModules ( tensor ( v )  ) ==  0
numModules   ( tensor ( v )  ) ==  1
numModules   ( Kronecker     ) ==  1
numCoModules ( Kronecker     ) ==  1
numCoModules ( 0 ( t1 )      ) ==  numCoModules ( t       )
numModules   ( 0 ( t1 )      ) ==  numModules   ( t       )
numCoModules (      - t      ) ==  numCoModules ( t       )
numModules   (      - t      ) ==  numModules   ( t       )
numCoModules (  t1 + t2      ) ==  numCoModules ( t1 + t2 )
numModules   (  t1 + t2      ) ==  numModules   ( t1 + t2 )
numCoModules (   a * t1      ) ==  numCoModules ( t1      )
numModules   (   a * t1      ) ==  numModules   ( t1      )
numCoModules ( t1 * t2       ) ==  numCoModules(t1) + numCoModules(t2)
numModules   ( t1 * t2       ) ==  numModules  (t1) + numModules  (t2)
numCoModules ( < t1,t2>      ) ==  numCoModules(t1) - numModules  (t2)
numModules   ( < t1,t2>      ) ==  numModules  (t1) - numCoModules(t2)

numCoModules ( contract ( t1 )    )   == numCoModules ( t ) - 1
numModules   ( contract ( t1 )    )   == numModules   ( t ) - 1

< tensor ( a ), tensor ( b ) >        ==  tensor (   a * b   )
< tensor ( w ), tensor ( m ) >        ==  tensor (   w ( m ) )
< tensor ( m ), tensor ( w ) >        ==  tensor (   w ( m ) )
tensor ( a ) * tensor ( b )           ==  tensor ( a * b )
tensor ( a ) * tensor ( m )           ==  tensor ( a * m )
tensor ( a ) * tensor ( w )           ==  tensor ( a * w )

< < t1,  tensor ( a ) >, t2 >         ==  < t1, tensor ( a ) * t2 >
<   t1 * tensor ( a )  , t2 >         ==  < t1, tensor ( a ) * t2 >
< t1 * tensor ( m ) , tensor ( w ) >  ==  t1 * < tensor ( m ), tensor ( w ) >
< t1 * tensor ( v ) , tensor ( n ) >  ==  t1 * < tensor ( v ), tensor ( n ) >
< t1 * tensor ( m ) , tensor ( n ) >  ==  < t1, tensor ( n ) > * tensor ( m )
< t1 * tensor ( v ) , tensor ( w ) >  ==  < t1, tensor ( w ) > * tensor ( v )
< t1 , t2 * t3 >                      ==  < < t1, t3 > , t2 >
< t1 + t2, t3 >                       ==  < t1, t3 > + < t2, t3 >
```

```
      < t1, t2 + t3 >                      ==  < t1, t2 > + < t1, t3 >
      < a * t1, t2  >                      ==  a * < t1, t2 >
      < t1, a * t2  >                      ==  a * < t1, t2 >
      < t1, ( t2 + t3 ) * t4 >             ==  < t1, t2 * t4 > + < t1, t3 * t4 >
      < t1, t2 * ( t3 + t4 ) >             ==  < t1, t2 * t3 > + < t1, t2 * t4 >
      < t1, ( a * t2 ) * t3  >             ==  a * < t1, t2 * t3 >
      < t1, t2 * ( a * t3 )  >             ==  a * < t1, t2 * t3 >

      contract ( t1 * tensor(m) )          ==  < t1, tensor ( m ) >
      contract ( t1 * tensor(v) )          ==  < t1, tensor ( v ) >
      < Kronecker,( tensor (v)*tensor (m) > ==  tensor ( v ( m ) )


      isRing    ( t1 )         == ( numCoModules(t1)=0 /\ numModules ( t1 )=0)
      isModule  ( t1 )         == ( numCoModules(t1)=1 /\ numModules ( t1 )=0)
      isCoModule( t1 )         == ( numCoModules(t1)=0 /\ numModules ( t1 )=1)
      equalType ( t1, t2 )     == ( numCoModules(t1)   =  numCoModules(t2) /\
                                    numModules  (t1)   =  numModules  (t2)   )
      okInnerProdrod ( t1,t2 ) == ( numModules  (t2)   <= numCoModules(t1) /\
                                  ( numCoModules(t2)   <= numModules  (t1)    )
      canContract ( t1 )       == 0 < numModules(t1)   /\ 0 < numCoModules(t1)
end specification
```

This algebraic specification is independent of any programming language, it only asserts properties (semantics) between a sort (class) and a set of functions that operates on the sort. Note how the module properties of the tensor is specified. Only tensors with equal type belongs to a common module, and we must because of this specify the module properties for all possible types of tensors.

9.5 C++ CLASS SIGNATURE

One may probably implement many different classes in C++ that have the properties described above. It is still an advantage to use equal signature for all classes that have the properties described. The reason is that we then may implement algorithms that are reusable between a wide range of different *tensors*.

From the given algebraic specification, we can construct a C++ class signature that every class with *tensor* semantics must contain:

```
class tensor
{  public:

      //   C++ specific operations
      tensor                    (                  )      ;
      tensor                    ( const tensor   & )      ;
      ~tensor                   (                  )      ;
```

```
      int           operator ==   ( const tensor    & ) const ;
      tensor &      operator =    ( const tensor    & )       ;

      //   basic operations
      tensor                      ( const ring      & )       ;
      tensor                      ( const module    & )       ;
      tensor                      ( const coModule  & )       ;
      operator      ring          (                   ) const ;
      operator      module        (                   ) const ;
      operator      coModule      (                   ) const ;

      int           isRing        (                   ) const ;
      int           isModule      (                   ) const ;
      int           isCoModule    (                   ) const ;
      int           equalType     ( const tensor &    ) const ;
      int           canContract   (                   ) const ;
      int           numCoModules  (                   ) const ;
      int           numModules    (                   ) const ;

      tensor        zero          (                   ) const ;
      tensor        operator   -  (                   ) const ;
      tensor        operator   +  ( const tensor   &  ) const ;

friend tensor       operator *    ( const ring     &,
                                    const tensor   &  )       ;

      array<ring>   coordinates   (                   ) const ;
      int           dimension     (                   ) const ;
      array<tensor> basis         (                   ) const ;

      tensor        operator  *   ( const tensor   &  ) const ;
      tensor        contract      (                   ) const ;
      tensor        operator ()   ( const tensor   &  ) const ;
static tensor       kronecker     (                   )       ;

      //   composed functions
      tensor        operator  -   ( const tensor   &  ) const ;
      tensor &      operator -=   ( const tensor   &  )       ;
      tensor &      operator +=   ( const tensor   &  )       ;
      tensor &      operator *=   ( const ring     &  )       ;
} ;
```

We demand that all classes in C++ that is supposed to represent a em tensor, must (at least) have the above given class signature. Notice that this is the minimum set of functions an implementation must contain, it will cause no problem if there are implemented more member functions in the class. Typically, there must at least be some extra constructors that are dependent of the actual implementation of the *tensor*.

9.6 OPERATIONS

9.6.1 Generators

Signature : tensor ( ) ;
Preconditions : None.
Result : Constructs a tensor of type (0,0) with the property:
Ring (tensor ( )) == Ring (0)
Notify : This is the *default constructor* for the class.
Example : tensor t ;

Signature : tensor (const Ring & *r*) ;
Preconditions : None.
Result : Constructs a tensor of type (0,0) with the property:
Ring (tensor (r)) == r
Notify : tensor(r).dimension() == 1.
Example : Ring r ;
tensor t (r) ;

Signature : tensor (const Module & *m*) ;
Preconditions : None.
Result : Constructs a tensor of type (1,0) with the property:
Module (tensor (m)) == m
Notify : tensor(m).coordinates () == m.coordinates ()
Example : Module m ;
tensor t (m) ;

Signature : tensor (const CoModule & *c*) ;
Preconditions : None.
Result : Constructs a tensor of type (0,1) with the property:
CoModule (tensor (c)) == c
Notify : tensor(c).coordinates () == c.coordinates ()
Example : CoModule c ;
tensor t (m) ;

Signature : tensor zero ( ) const ;
Preconditions : None.
Result : Constructs a tensor with the following property:
t + t.zero () == t
for all tensors *t*.

Notify : t.zero().coordinates() == array<Ring> (t.dimension(), Ring (0))
Example : tensor t ;
cout << t.zero().coordinates () [0] << endl ;

Signature : tensor operator - ( ) const ;
Preconditions : None.
Result : Constructs a tensor with the following property:
$t + (-t) == t.zero ()$
for all tensors t.
Notify : (-t).coordinates() [i] == - (t.coordinates() [i])
Example : tensor t1, t2 ;
t1 = - t2 ;

Signature : tensor operator + (const tensor & t) const ;
Preconditions : The two tensor arguments must be of equal type.
Result : This member function has the following properties:
$a + b == b + a$
$a + (b + c) == (a + b) + c$
for all tensors a, b and c of equal type.
Notify : (t1+t2).coordinates()[i] == (t1.coordinates()[i])+(t2.coordinates()[i])
Example : tensor t1, t2, t3 ;
if (t2.equalType (t3))
t1 = t2 + t3 ;

Signature : friend tensor operator * (const Ring & r , const tensor & t) ;
Preconditions : None.
Result : This member function has the following properties:
m * (a + b) == m * a + m * b
(m + n) * a == m * a + n * a
(m * n) * a == m * (n * a)
for all elements m and n in the class *Ring*, all tensors a and b of equal type..
Notify : (m*a).coordinates()[i] == m*(a.coordinates()[i])
Example : Ring r ;
tensor t ;
t = r * t ;

Signature : array<tensor> basis (const int i) const ;
Preconditions : None.

Result : Returns the basis elements of the object.
Notify : tensors of different type have different basis elements.
Example : tensor t, res ;
for (int i = 0 ; i ¡ t.dimension () ; ++ i)
res += t.basis ( )[i] ;

Signature : tensor operator * (const tensor & t) const ;
Preconditions : None.
Result : Returns the *tensor product* of the calling tensor and t.
Notify : One may prove that the coordinates of the result may be expressed as:

$$(t_1 * t_2)^{i_1,\ldots,i_{r_1},m_1,\ldots,m_{r_2}}_{j_1,\ldots,j_{s_1},n_1,\ldots,n_{s_2}} = (t_1)^{i_1,\ldots,i_{r_1}}_{j_1,\ldots,j_{s_1}} * (t_2)^{m_1,\ldots,m_{r_2}}_{n_1,\ldots,n_{s_2}}$$

(with an implicit summation over all repeated indexes).See also the definition of *tensor product* given above.
Example : Module m ;
CoModule c ;
tensor t = tensor (m) * tensor (c) ;

Signature : tensor contract ( ) const ;
Preconditions : The calling tensor must be of type (r, s), where $0 < r, s$.
Result : Returns the *contraction* of the calling tensor.
Notify : One may prove that the coordinates of the result may be expressed as:

$$k(t)^{i_1,\ldots,i_{r-1}}_{j_1,\ldots,j_{s-1}} = t^{i_1,\ldots,i_{r-1},m}_{j_1,\ldots,j_{s-1},m}$$

(with an implicit summation over all repeated indexes).See also the definition of *contraction* given above.
Example : Module m ;
CoModule c ;
tensor t1 = tensor (m) * tensor (c) ;
tensor t2 = t1.contract () ;
Ring r = Ring (t2) ;

Signature : tensor operator () (const tensor & t) const ;
Preconditions : If the calling tensor is of type $(r1, s1)$, and t is of type $r2, s2)$, then we must have that $s2 \leq r1$ and $r2 \leq s1$.
Result : Calculates the inner product between the calling tensor and t.
Notify : One may prove that the coordinates of the result may be expressed as:

$$i(t_1, t_2)^{i_1,\ldots,i_m}_{j_1,\ldots,j_n} = (t_1)^{i_1,\ldots,i_m,k_1,\ldots,k_{s_2}}_{j_1,\ldots,j_n,l_1,\ldots,l_{r_2}} * t^{l_1,\ldots,l_{r_2}}_{k_1,\ldots,k_{s_2}}$$

(with an implicit summation over all repeated indexes).See also

the definition of *inner product* given above.

Example : tensor t1, t2, t3 ;
t3 = t1 (t2) ;

Signature : static tensor kronecker ( ) ;
Preconditions : None.
Result : Returns a tensor c of type (1, 1), with the following properties:
$c\ (\ v\ ) == v$
$c\ (\ w\ ) == w$
for all tensors v of type (1, 0) and all tensors w of type (0, 1).
Notify : the coordinates of this tensor may be expressed as:

$$\delta^i_j = \begin{cases} 0 & \text{if } i <> j \\ 1 & \text{if } i = j \end{cases}$$

and is known as the *Kronecker tensor.*
Example : tensor t = tensor::kronecker () ;

9.6.2 Observer functions

Signature : operator Ring ( ) const ;
Preconditions : The calling tensor must be of type (0, 0).
Result : Returns the ring element which is isomorphic with the calling object.
Notify : ring (t) == t.coordinates () [0].
Example : tensor t ;
Ring r ;
if (t.isRing ())
r = Ring (t) ;

Signature : operator Module ( ) const ;
Preconditions : The calling tensor must be of type (1, 0).
Result : Returns the module element which is isomorphic with the calling object.
Notify : Module (t).coordinates () == t.coordinates ()
Example : tensor t ;
Module r ;
if (t.isModule ())
r = Module (t) ;

Signature : operator CoModule ( ) const ;

Preconditions : The calling tensor must be of type (0, 1).
Result : Returns the comodule element which is isomorphic with the calling object.
Notify : CoModule (t).coordinates () == t.coordinates ()
Example :
```
tensor t ;
CoModule r ;
if ( t.isCoModule () )
    r = CoModule ( t ) ;
```

Signature : int numCoModules ( ) const ;
Preconditions : None.
Result : Returns the number of comodule arguments the calling tensor may accept.
Notify : The type of a tensor t is equal to (t.numCoModules (), t.numModules ()).
Example :
```
tensor t ;
cout << t.numCoModules () << endl ;
```

Signature : int numModules ( ) const ;
Preconditions : None.
Result : Returns the number of module arguments the calling tensor may accept.
Notify : The type of a tensor t is equal to (t.numCoModules (), t.numModules ()).
Example :
```
tensor t ;
cout << t.numModules () << endl ;
```

Signature : array<Ring> coordinates ( ) const ;
Preconditions : None.
Result : Returns an array with the coordinates of the tensor. It is common to regard a tensor t of type (r, s) as a $(r + s)$ dimensional array indexed like this:
$t^{i_1,\ldots,i_r}_{j_1,\ldots j_s}$
where $0 \leq i_1, \ldots, i_r, j_1, \ldots j_s < d$,where d is equal to *Module::dimension()*. This function will 'unfold' this array into the corresponding one dimensional array.
Notify : t.coordinates().size () == t.dimension().
Example :
```
tensor t ;
array<Ring> arr = t.coordinates () ;
```

9.6.3 Composed functions

Signature : int dimension ( ) const ;
Preconditions : None.
Result : Returns the dimension of the module the calling tensor belongs to.
Notify : A tensor of type (r,s) has dimension d^{r+s}, where d is equal to *Module::dimension()*.
Example : tensor t ;
cout << t.dimension () << endl ;

Signature : int isRing ( ) const ;
Preconditions : None.
Result : Returns a non-zero integer if the calling object is of type (0,0). If else, the integer zero is returned.
Notify : t.isRing () == (t.numCoModules () == 0 && t.numModules () == 0)
Example : tensor t ;
Ring r ;
(if t.isRing ())
r = Ring (t) ;

Signature : int isModule ( ) const ;
Preconditions : None.
Result : Returns a non-zero integer if the calling object is of type (1,0). If else, the integer zero is returned.
Notify : t.isModule () == (t.numCoModules () == 1 &&
t.numModules () == 0).
Example : tensor t ;
Module r ;
(if t.isModule ())
r = Module (t) ;

Signature : int isCoModule ( ) const ;
Preconditions : None.
Result : Returns a non-zero integer if the calling object is of type (0,1). If else, the integer zero is returned.
Notify : t.isCoModule () == (t.numCoModules () == 0 &&
t.numModules () == 1).
Example : tensor t ;
CoModule r ;
(if t.isCoModule ())
r = CoModule (t) ;

Signature : int equalType (const tensor & t) const ;
Preconditions : None.
Result : Returns a non-zero integer if the calling object has equal type as t. If else, the integer zero is returned.
Notify : t.equalType (t1) == (t.numCoModules () == t1.numCoModules () && t.numModules () == t1.numModules ())
Example : tensor t1, t2, t3 ;
if (t2.equalType (t3)) t1 = t2 + t3 ;

Signature : int canContract ( ) const ;
Preconditions : None.
Result : Returns a non-zero integer if the calling object has type (r, s), where $0 < r, s$.
Notify : t.canContract () == (0 < t.numCoModules () && 0 < t.numCoModules ())
Example : tensor t1, t2 ;
if (t2.canContract ()) t1 = t2.contract () ;

Signature : tensor operator - (const tensor & t) const ;
Preconditions : The two argument tensors must be of equal type.
Result : This function is equal to: $t1 + (-t)$, where $t1$ is the calling object.
Example : tensor t1, t2, t3 ;
if (t1.equalType (t2)) t3 = t1 - t2 ;

Signature : tensor & operator -= (const tensor & t) ;
Preconditions : The two argument tensors must be of equal type.
Result : This function is equal to: $t1 = t1 + (-t)$, where $t1$ is the calling object.
Example : tensor t1, t2 ;
if (t1.equalType (t2)) t1 -= t2 ;

Signature : tensor & operator += (const tensor & t) ;
Preconditions : The two argument tensors must be of equal type.
Result : This function is equal to: $t1 = t1 * t$, where $t1$ is the calling object.
Example : tensor t1, t2 ;
if (t1.equalType (t2)) t1 += t2 ;

Signature : tensor & operator *= (const Ring & r) ;
Preconditions : None.
Result : This function is equal to: $t = r * t$, where t is the calling object.

Example : tensor t ;
Ring r ;
t *= r ;

9.7 REFERENCES

A good introduction to algebraic specification may be found in:
David A. Watt : *Programming Language Syntax and Semantics*, Prentice Hall 1991.

Most of the mathematical background will be found in:
Serge Lang: *Algebra*, Addison Wesley 1965.
Abraham, Marsden and Ratiu : *Manifolds, tensor Analysis, and Applications*, Springer-Verlag 1983.

For a more theoretical approach, see :
M. Haveraaen, V. Madsen, H. Munthe-Kaas : *Algebraic Programming Technology for Partial Differential Equations*, Precedings from "Norsk Informatikk Konferanse 1992".

For a less theoretical approach, see :
V. Madsen: *Tensors, from specification to implementation*; Thesis for the master degree, Institute of Informatics, University of Bergen.

9.8 FILES AND VERSIONS

Idea documentation : /tmp_mnt/Home/stud2/vmadsen/tensor/dok/tensor_idea.ps
Based on : /tmp_mnt/Home/stud2/vmadsen/tensor/dok/ring_idea.ps
/tmp_mnt/Home/stud2/vmadsen/tensor/dok/module_idea.ps
/tmp_mnt/Home/stud2/vmadsen/tensor/dok/comodule_idea.ps
/tmp_mnt/Home/stud2/vmadsen/tensor/dok/tensor_idea.ps

9.9 MAINTENANCE AND SUPPORT

This class specification is documented by :
Victor Madsen
Institute of Informatics
University of Bergen
email : vmadsen@ii.uib.no

Chapter 10

Idea Documentation for Class : diffTensor

10.1 NAME

Name : *diffTensor*
Version : 1.0
Compiler : AT&T C++ v. 3.01
Based on : Idea documentation for class *diffRing.*
Idea documentation for class *diffModule.*
Idea documentation for class *diffCoModule.*
Idea documentation for class *tensor.*
Description : Specifies the signature and semantics for the mathematical object *differential tensor.*

10.2 PURPOSE

This documentation describes the syntax and semantics for the mathematical object *differential tensor* in C++. There may be many C++ classes that have properties equal to a *differential tensor*, and one may not give one implementation that cover them all. This documentation will only describe the least signature these classes need, and the semantics the functions in the classes must have.

10.3 DESCRIPTION

10.3.1 Differential operators on tensors

Let R be a differential ring, and let $D[R]$ be a differential module over R. Let D_r be a differential operator on R, and let D_m be a differential operator on $D[R]$. A differential

operator D_t on $T(M)$ is a linear map $D_t : T_s^r(M) \rightarrow T_s^r(M), r, s \geq 0$ with the following property:

$$D_r(t(a_1 * \ldots * a_r * x_1 * \ldots * x_s)) =$$

$$D_t(t)(a_1 * \ldots * a_r * x_1 * \ldots * x_s) +$$

$$\sum_{j=1}^{r} t(a_1 * \ldots * D_k(a_j) * \ldots * a_r * x_1 * \ldots * x_s) +$$

$$\sum_{j=1}^{s} t(a_1 * \ldots * a_r * x_1 * \ldots * D_m(x_j) * \ldots * x_s)$$

where $a_i \in T_1^0(M)$ and $x_i \in T_0^1(M)$. D_t is uniquely defined when D_r and D_m is given. D_t is a trivial differential operator if and only if D_r is trivial.

Note that for all tensors $t \in T_0^0(M)$, D_t is equal to D_r. We may do simular observations for $t \in T_0^1(M)$ and $T_1^0(M)$. Because of this, we may specify the following properties:

- $\forall f, g \in T_s^r(M) : D_t(f + g) = D_t(f) + D_t(g)$
- $\forall f, g \in T_0^0(M) : D_t(f * g) = D_t(f) * g + f * D_t(g)$
- $\forall f \in T_0^0(M), \forall g \in T_s^r(M) : D_t(f * g) = D_t(f) * g + f * D_t(g)$

10.3.2 Theorem

Let t_1 and t_2 be two tensors $t_1 \in T_{s1}^{r1}(M)$ and $t_2 \in T_{s2}^{r2}(M)$. A differential operator D_t on tensors distributes over *tensor product* in like this:

$$D_t(t_1 * t_2) = D_t(t_1) * t_2 + t_1 * D_t(t_2)$$

This property may be used to give an equivalent definition of a differential operator D_t on tensors:

- $\forall f \in T_{s1}^{r1}(M), \forall g \in T_{s2}^{r2}(M) : D_t(t_1 * t_2) = D_t(t_1) * t_2 + t_1 * D_t(t_2)$
- $\forall f, g \in T_s^r(M) : D_t(f + g) = D_t(f) + D_t(g)$
- $\forall f \in T_0^0(M), \forall g \in T_s^r(M) : D_t(f * g) = D_t(f) * g + f * D_t(g)$

10.3.3 Theorem

Let R be a differential ring, and let $D[R]$ be a differential module over R. Let D_r be a differential operator on R, and let D_m be a differential operator on $D[R]$. We may always construct a corresponding differential operator D_t on $T^r_s(R)$. This differential operator is uniquely defined to be:

$$D_t(t) = (D_r(t^{p_1,\ldots,p_r}_{q_1,\ldots,q_s}) + t^{i_1,p_2,\ldots,p_r}_{q_1,\ldots,q_s} * e^{p_1}_{i_1} + \ldots + t^{p_1,\ldots,p_{r-1},i_r}_{q_1,\ldots,q_s} * e^{p_r}_{i_r}$$

$$-t^{p_1,\ldots,p_r}_{j_1,q_2,\ldots,q_s} * e^{j_1}_{q_1} - \ldots - t^{p_1,\ldots,p_r}_{q_1,\ldots,q_{s-1},j_s} * e^{j_s}_{q_s})*$$

$$(b_{p_1},\ldots,b_{p_r},b^{q_1},\ldots * b^{q_s})$$

Note that the construction of a differential operator D_k on the dual module M^* is only a special case of the above given formula.

10.3.4 Lie derivative

By the above theorem we may extend the definition of the *Lie derivative* to become a differential operator on tensors. We have found:

$$e^i_j = -\frac{\partial}{\partial x^j}(w^i)$$

Together with the formula:

$$D_t(t) = (D_r(t^{p_1,\ldots,p_r}_{q_1,\ldots,q_s}) + t^{i_1,p_2,\ldots,p_r}_{q_1,\ldots,q_s} * e^{p_1}_{i_1} + \ldots + t^{p_1,\ldots,p_{r-1},i_r}_{q_1,\ldots,q_s} * e^{p_r}_{i_r}$$

$$-t^{p_1,\ldots,p_r}_{j_1,q_2,\ldots,q_s} * e^{j_1}_{q_1} - \ldots - t^{p_1,\ldots,p_r}_{q_1,\ldots,q_{s-1},j_s} * e^{j_s}_{q_s})*$$

$$(b_{p_1},\ldots,b_{p_r},b^{q_1},\ldots,b^{q_s})$$

we may derive the following explit formula for the Lie derivative of tensors:

$$\mathcal{L}_w(t) = [w^i * \frac{\partial}{\partial x^i}(t^{p_1,\ldots,p_r}_{q_1,\ldots,q_s}) - t^{i_1,p_2,\ldots,p_r}_{q_1,\ldots,q_s} * \frac{\partial}{\partial x^{i_1}}(w^{p_1}) - \ldots - t^{p_1,\ldots,p_{r-1},i_r}_{q_1,\ldots,q_s} * \frac{\partial}{\partial x^{i_r}}(w^{p_r})$$

$$+t^{p_1,\ldots,p_r}_{j_1,q_2,\ldots,q_s} * \frac{\partial}{\partial x^{q_1}}(w^{j_1}) + \ldots + t^{p_1,\ldots,p_r}_{q_1,\ldots,q_{s-1},j_s} * \frac{\partial}{\partial x^{q_s}}(w^{j_s})]*$$

$$(\frac{\partial}{\partial x^{p_1}},\ldots,\frac{\partial}{\partial x^{p_r}},dx^{q_1},\ldots * dx^{q_s})$$

10.3.5 Partial derivatives

Since we have defined the partial derivative operators on a differential ring R and a differential module $D[R]$, we may extend the definition of the partial derivative to $T^r_s(R)$ in the following way:

$$\frac{\partial}{\partial x^m}(t) = (\frac{\partial}{\partial x^m}(t^{p_1,\ldots,p_r}_{q_1,\ldots,q_s}) + t^{i_1,p_2,\ldots,p_r}_{q_1,\ldots,q_s} * \Gamma^{p_1}_{m,i_1} + \ldots + t^{p_1,\ldots,p_{r-1},i_r}_{q_1,\ldots,q_s} * \Gamma^{p_r}_{m,i_r}$$

$$-t^{p_1,\ldots,p_r}_{j_1,q_2,\ldots,q_s} * \Gamma^{j_1}_{m,q_1} - \ldots - t^{p_1,\ldots,p_r}_{q_1,\ldots,q_{s-1},j_s} * \Gamma^{j_s}_{m,q_s})*$$

$$(\frac{\partial}{\partial x^{p_1}}, \ldots, \frac{\partial}{\partial x^{p_r}}, dx^{q_1}, \ldots, dx^{q_s})$$

10.3.6 covariant derivative

Let R be a differential ring, and let $D[R]$ be the differential module over R. Let D_r be a differential operator on R, and let D_m be a differential operator on $D[R]$. Let $\Gamma^q_{i,j}$ be some suitable Christoffel symbols. The *covariant derivative* is defined as a differential operator $covDiff : T^r_s(D[R]) \rightarrow T^r_{s+1}(R)$ with the following property:

$$covDiff(t)(\frac{\partial}{\partial x^i}) = \frac{\partial}{\partial x^i}(t)$$

where $B = \{\frac{\partial}{\partial x^1}, \ldots, \frac{\partial}{\partial x^d}\}$ is the basis for $D[R]$.

By the above given theorem we may derive this explicit formula for the covariant derivative:

$$covDiff(t) = (\frac{\partial}{\partial x^m}(t^{p_1,\ldots,p_r}_{q_1,\ldots,q_s}) + t^{i_1,p_2,\ldots,p_r}_{q_1,\ldots,q_s} * \Gamma^{p_1}_{m,i_1} + \ldots + t^{p_1,\ldots,p_{r-1},i_r}_{q_1,\ldots,q_s} * \Gamma^{p_r}_{m,i_r}$$

$$-t^{p_1,\ldots,p_r}_{j_1,q_2,\ldots,q_s} * \Gamma^{j_1}_{m,q_1} - \ldots - t^{p_1,\ldots,p_r}_{q_1,\ldots,q_{s-1},j_s} * \Gamma^{j_s}_{m,q_s})*$$

$$(\frac{\partial}{\partial x^{p_1}}, \ldots, \frac{\partial}{\partial x^{p_r}}, dx^{q_1}, \ldots, dx^{q_s}, b^m)$$

The covariant derivative of an element $f \in T^0_0(R)$ is often referred to as the *gradient* of f.

10.4 ALGEBRAIC SPECIFICATION

After this, we may construct the following *algebraic specification* of a *differential tensor*:

```
specification DIFFERENTIAL-TENSOR
include INTEGER, BOOL
parameters
   sort      diffRing, Christoffel, diffModule, diffCoModule, diffTensor
```

```
operations
   0                      :                                -> diffRing
   1                      :                                -> diffRing
   - _                    : diffRing                       -> diffRing
   _ + _                  : diffRing * diffRing            -> diffRing
   _ * _                  : diffRing * diffRing            -> diffRing
   numDim      ( _ )      : diffRing                       -> Integer
   partialDiff ( _,_ )    : Integer * diffRing
                            | (0<#1<=numDim(#2))           -> diffRing

   spaceDimension( _ ) : Christoffel                       -> Integer
   okIndex  ( _,_,_ )  : Integer * Integer * Integer       -> Bool
   read ( _,_,_,_ )    : Christoffel * Integer * Integer
                         * Integer | okIndex (#2,#3,#4)    -> diffRing
   set  ( _,_,_,_,_ )  : Christoffel * diffRing * Integer
                         * Integer * Integer
                         | okIndex(#3,#4,#5)              -> Christoffel

   0                :                                                -> diffModule
   _ + _            : diffModule * diffModule                        -> diffModule
   - _              : diffModule                                     -> diffModule
   _ * _            : diffRing * diffModule                          -> diffModule
   dimension ( _ )  : diffModule                                     -> Integer
   coordinat ( _,_ ) : diffModule * Integer | (0<#2<=dimension(#1)) -> diffRing
   basis     ( _,_ ) : diffModule * Integer | (0<#2<=dimension(#1)) -> diffModule

   partialDiff ( _,_,_ ) : Integer * diffModule * Christoffel
                           Christoffel | (0<#1<=dimension(#2))      -> diffModule
   LieDiff ( _,_ )       : diffModule * diffRing                    -> diffRing
   LieDiff ( _,_ )       : diffModule * diffModule                  -> diffModule


   0                      :                                   -> diffCoModule
   _ + _                  : diffCoModule * diffCoModule       -> diffCoModule
   - _                    : diffCoModule                      -> diffCoModule
   _ * _                  : diffRing * diffCoModule           -> diffCoModule
   dimension ( _ )        : diffCoModule                      -> Integer
   coordinat ( _,_ )      : diffCoModule * Integer
                            | (0<#2<=dimension(#1))           -> diffRing
   basis     ( _,_ )      : diffCoModule * Integer
                            | (0<#2<=dimension(#1))           -> diffCoModule
   _ ( _ )                : diffCoModule * diffModule         -> diffRing

   partialDiff ( _,_,_ ) : Integer * diffCoModule *
                           Christoffel | (0<#1<=dimension(#2)) -> diffCoModule
   LieDiff ( _,_ )       : diffModule * diffCoModule          -> diffCoModule


   ring         ( _ ) : diffTensor | isRing    (#1)           -> diffRing
   module       ( _ ) : diffTensor | isModule  (#1)           -> diffModule
   comodule     ( _ ) : diffTensor | isCoModule(#1)           -> diffCoModule
```

```
      numCoModules ( _ ) : diffTensor                                    -> Integer
      numModules   ( _ ) : diffTensor                                    -> Integer

      tensor ( _ )       : diffRing                                      -> diffTensor
      tensor ( _ )       : diffModule                                    -> diffTensor
      tensor ( _ )       : diffCoModule                                  -> diffTensor

      0 ( _ )            : diffTensor                                    -> diffTensor
      _ + _              : diffTensor * diffTensor                       -> diffTensor
      - _                : diffTensor                                    -> diffTensor
      _ * _              : diffRing * diffTensor                         -> diffTensor
      dimension ( _ )    : diffTensor                                    -> Integer
      coordinat ( _,_ )  : diffTensor * Integer | (0<#2<=dimension(#1))  -> diffRing
      basis     ( _,_ )  : diffTensor * Integer | (0<#2<=dimension(#1))  -> diffTensor

      _ * _              : diffTensor * diffTensor                       -> diffTensor
      < _,_ >            : diffTensor * diffTensor  |okInnerProd (#1,#2) -> diffTensor
      contract    ( _ )  : diffTensor           | canContract ( #1 )     -> diffTensor
      Kronecker          :                                               -> diffTensor

      isRing          ( _ )    : diffTensor                              -> Bool
      isModule        ( _ )    : diffTensor                              -> Bool
      isCoModule      ( _ )    : diffTensor                              -> Bool
      equalType       ( _,_ )  : diffTensor * diffTensor                 -> Bool
      okInnerProd     ( _, _ ) : diffTensor * diffTensor                 -> Bool
      canContract     ( _ )    : diffTensor                              -> Bool

      LieDiff ( _,_ )        : diffTensor * diffTensor | isModule( #1 )  -> diffTensor
      partialDiff ( _,_,_ )  : Integer * diffTensor *
                               Christoffel |(0<#1<=numDim(0))            -> diffTensor
      covDiff ( _, _ )       : diffTensor * Christoffel                  -> diffTensor
specifies
   variables   t1, t2   : diffTensor
               a        : diffRing
               v, m     : diffModule
               c        : Christoffel
               i        : Integer
   equations
      DIFFERENTIAL-COMODULE ( diffRing, christoffel, diffModule,
                              0, 1, - , +, *, numDim, partialDiff,
                              spaceDimension, okIndex, read, sett ,
                              0, -, +, *, dimension, coordinat , basis,
                              partialDiff, LieDiff, LieDiff,
                              0, -, +, *, dimension, coordinat , basis, _ (_),
                              partialDiff, LieDiff                       )

      TENSOR                ( diffRing, diffModule, diffCoModule, diffTensor,
                              0, 1, - , +, *,
                              0, -, +, *, dimension, coordinat, basis,
                              0, -, +, *, dimension, coordinat, basis, _(_),
                              ring, module, comodule, numCoModules, numModules,
```

```
                        tensor, tensor, tensor, 0, +, -, *, dimension,
                        coordinat, basis, *, <,>, contract, Kronecker,
                        isRing, isModule, isCoModule,
                        equalType, okInnerProd, mayContract  )

    partialDiff ( i, tensor ( a ), c )  ==  tensor ( partialDiff ( i, a    ) )
    partialDiff ( i, tensor ( m ), c )  ==  tensor ( partialDiff ( i, m, c ) )
    partialDiff ( i, t1 * t2     , c )  ==  partialDiff ( i, t1, c ) * t2 +
                                            t1 * partialDiff ( i, t2, c )

    LieDiff ( tensor ( v ), tensor ( a ) )  == tensor ( LieDiff ( v, a ) )
    LieDiff ( tensor ( v ), tensor ( m ) )  == tensor ( LieDiff ( v, m ) )
    LieDiff ( tensor ( v ), t1 * t2      )  == LieDiff ( tensor ( v ), t1 ) * t2 +
                                               t1 * LieDiff ( tensor ( v ), t2 )

    < CovDiff ( t, c ), tensor ( basis ( v, i ) ) > == partialDiff ( i, t, c )
end specification
```

This algebraic specification is independent of any programming language, it only asserts properties (semantics) between a sort (class) and a set of functions that operates on the sort. This specification use an array class that is not formally specified, but the semantic is obvious, and the specification of ARRAY is because of this not explicitly shown.

10.5 C++ CLASS SIGNATURE

One may probably implement many different classes in C++ that have the properties described above. It is still an advantage to use equal signature for all classes that have the properties described. The reason is that we then may implement algorithms that are reusable between a wide range of different implementations.

From the given algebraic specification of a DIFFERENTIAL-TENSOR, we may construct a C++ class signature that every class with *differential tensor* semantics must contain:

```
class diffTensor
{  public:
        //   C++ specific operations
        diffTensor                      (                          )       ;
        diffTensor                      ( const  diffTensor    & )       ;
        ~diffTensor                     (                          )       ;
        int             operator ==     ( const  diffTensor    & ) const ;
        diffTensor &    operator =      ( const  diffTensor    & )       ;

        //   basic operations
        diffTensor                      ( const diffRing       & )       ;
```

```
        diffTensor                  ( const diffModule     & )       ;
        diffTensor                  ( const diffCoModule   & )       ;
        operator        diffRing    (                        ) const ;
        operator        diffModule  (                        ) const ;
        operator        diffCoModule(                        ) const ;

        int             isRing      (                        ) const ;
        int             isModule    (                        ) const ;
        int             isCoModule  (                        ) const ;
        int             equalType   ( const  diffTensor    & ) const ;
        int             canContract (                        ) const ;
        int             numCoModules(                        ) const ;
        int             numModules  (                        ) const ;

        diffTensor      zero        (                        ) const ;
        diffTensor      operator  - (                        ) const ;
        diffTensor      operator  + ( const  diffTensor    & ) const ;
friend  diffTensor      operator *  ( const  diffRing      &,
                                      const  diffTensor    & )       ;

        array<diffRing>   coordinates(                       ) const ;
        int               dimension (                        ) const ;
        array<diffTensor> basis     (                        ) const ;

        diffTensor      operator  * ( const  diffTensor    & ) const ;
        diffTensor      contract    (                        ) const ;
        diffTensor      operator () ( const  diffTensor    & ) const ;
static  diffTensor      kronecker   (                        )       ;

        //   differential operators
        diffTensor      lieDiff     ( const  diffTensor    & ) const ;
        diffTensor      partialDiff ( const int          ,
                                      const christoffel    & ) const ;
        diffTensor      covDiff     ( const christoffel    & ) const ;

        //   composed functions
        diffTensor      operator  - ( const  diffTensor    & ) const ;
        diffTensor &    operator -= ( const  diffTensor    & )       ;
        diffTensor &    operator += ( const  diffTensor    & )       ;
        diffTensor &    operator *= ( const  diffRing      & )       ;

} ;
```

We demand that all classes in C++ that is supposed to represent these classes, must at least have the above given class signature. Notice that this is the minimum set of functions an implementation must contain, it will cause no problem if there are implemented more member functions in the class. Typically, there must at least be some extra constructors that are dependent of the actual implementation of the class specification.

10.6 OPERATIONS

This class specifiation contain all member functions from the class specification *tensor*. In addition this class contains:

Generators

Signature : DiffTensor lieDiff (const DiffTensor & t) const ;
Preconditions : t is a tensor of type (1,0).
Result : Returns the Lie derivative of the calling tensor in direction t.
Notify : See also the definition above.
Example : DiffModule m ;
DiffTensor t1, t2 ;
t1 = t2.lieDiff (DiffTensor (m)) ;

Signature : DiffTensor partialDiff (const int i, const Christoffel & c) const ;
Preconditions : $0 \leq i <$*diffRing::numDimensions ()*.
Result : Returns the partial derivative of the calling object in dimension i with regard to the christoffel symbol c.
Notify : See also the definition above.
Example : Christoffel c ;
DiffTensor t1, t2 ;
t1 = t2.partialDiff (0, c) ;

Signature : DiffTensor covDiff (const Christoffel & c) const ;
Preconditions : None.
Result : Returns the covariant derivative of the calling object with regard to the christoffel symbol c.
Notify : See also the definition above.
Example : Christoffel c ;
DiffTensor t1, t2 ;
t1 = t2.covDiff (c) ;

10.7 REFERENCES

A good introduction to algebraic specification may be found in:
David A. Watt : *Programming Language Syntax and Semantics*, Prentice Hall 1991.

Most of the mathematical background will be found in:
Serge Lang: *Algebra*, Addison Wesley 1965.

Abraham, Marsden and Ratiu : *Manifolds, Tensor Analysis, and Applications*, Springer-Verlag 1983.

For a more theoretical approach, see :
M. Haveraaen, V. Madsen, H. Munthe-Kaas : *Algebraic Programming Technology for Partial Differential Equations*, Precedings from "Norsk Informatikk Konferanse 1992".

For a less theoretical approach, see :
V. Madsen: *Tensors, from specification to implementation*; Thesis for the master degree, Institute of Informatics, University of Bergen.

10.8 FILES AND VERSIONS

Idea documentation : /tmp_mnt/Home/stud2/vmadsen/ tensor/dok/difftensor_idea.ps
Based on : /tmp_mnt/Home/stud2/vmadsen/ tensor/dok/tensor_idea.ps
/tmp_mnt/Home/stud2/vmadsen/ tensor/dok/diffcomodule_idea.ps
/tmp_mnt/Home/stud2/vmadsen/ tensor/dok/diffmodule_idea.ps
/tmp_mnt/Home/stud2/vmadsen/ tensor/dok/diffring_idea.ps

10.9 MAINTENANCE AND SUPPORT

This class specification is documented by :
Victor Madsen
Institute of Informatics
University of Bergen
email : vmadsen@ii.uib.no

Chapter 11

User Documentation Class : array<T>

11.1 NAME

Name	: array<T>
Version	: 1.0
Compiler	: AT&T C++ v. 3.01
Arguments	: *template<class T> class array*
Description	: Implements the class array as a container with elements from the class T. The elements are indexed by integers in the range $[0, \ldots, n-1]$, where n is the number of elements in the array.

11.2 PURPOSE

An array is an indexed collection of elements from a class T. The elements in the array have no name, the only way to access the elements is through indexes. The array has a bounded *size*, where the *size* means how many elements the array contains. Each element in the array have a fixed position, and we refer to the elements in the array by their position. The first position in an array is the position 0, and the last position is $size - 1$, where the *size* is the numbers of elements in the array. This class may create arrays of any size, only limited by the available memory on the computer. We may also concatenate two arrays to get a new array with a new length equal to the sum of the lengths of the two argument arrays.

11.2.1 Limitations

The class has been implemented with a technique called *reference counting*. It means that objects in this class may be copied in constant time (independent of the size of the array). This makes the class efficient for assignments, argument passing, etc.. The drawback with this approach is that indexing in an array that is not declared as constant requires us to do one extra test to ensure that we will not get any side effects to other arrays that share the same memory locations. If there are any chance of side effects, the object will make its own local copy of the data. If efficiency is important, one should declare as many arrays as possible constant. A constant array object knows there are no possibility for side effects, and will because of this not perform the extra test.

11.3 DESCRIPTION

11.3.1 Restrictions

If the symbol _DEBUG is defined there will be done argument testing in all functions in the class (including tests for indexes out of bound). It is recommended that this symbol is always defined. At least one should have this symbol defined while debugging any applications. If the class finds an error in the arguments to a function, or there is not enough available memory, there will be printed an error message to the stream *cout* that reports where the error occurred, and execution stops (by the function call *exit(-1);)*. The symbol _DEBUG is defined like this :

```
#define   _DEBUG
```

The class has the following signature :

```
template <class T> class array
{
   public:
       //   C++ operations
inline array                  (                          )       ;
inline array                  ( const array & elArr      )       ;
inline ~array                 (                          )       ;
       int        operator  == ( const array & elArr      ) const ;
inline array &    operator  =  ( const array & elArr      )       ;

       //   basic operations
inline array                  ( const int     size       )       ;
       array                  ( const int     size  ,
                                const T     & el         )       ;
inline int        size        (                          ) const ;
       array      operator  +  ( const array &            ) const ;
inline T &        operator  [] ( const int                )       ;
inline const T &  operator  [] ( const int                ) const ;
} ;
```

11.3.2 Template arguments

The class uses the following template argument:

```
template <class T> class array
```

Class arguments

Name : T
Description : The class that the elements in the array belongs to.
Requirements : None.

11.3.3 Space analysis

Because this class is implemented with reference counting, the memory requirements will most often depend on if the object is a copy of another object, or have its own memory block. If an array with n objects owns its own memory block, it will require the following number of bytes:

$$2 * S_{int} + 2 * S_{ptr} + n * S_T$$

where :

S_{int} : is the number of bytes required to represent an integer.
$S_{ptr(int)}$: is the number of bytes required to represent a pointer.
S_T : is the number of bytes required to represent an element in the class T.

If an array share memory block with another array, the memory requirements for the copy of the array is:

$$S_{int} + 2 * S_{ptr}$$

Notice that the copy of an array has constant size !

11.3.4 Public members

Generators

Signature : inline array<T> () ;
Preconditions : None.
Result : Creates an array that contains no elements. I.e. an array with $size() == 0$.

Time consumption : Constant time.
Notify : The array contains zero objects.
Error handling : None.
Example : array<int> iArr ;

Signature : inline array<T> (const int *size*) ;
Preconditions : 0 <= *size*.
Result : Creates an array with *size* arbitrary objects of type T.
Time consumption : $O(size * T_T)$ time, where T_T is the time needed to construct a default element of type T.
Notify : The array contains objects generated by T's default constructor.
Error handling : Writes an error message and aborts if $size < 0$ and the symbol _DEBUG is defined, or the computer has not enough available memory.
Example : array<char> chrArr (10) ;

Signature : array<T> (const int *size*, const T & *el*) ;
Preconditions : 0 <= *size*.
Result : Creates an array with *size* objects of type T. Every object in the array will be a copy of *el*.
Time consumption : $O(size * T_T)$ time , where T_T is the time needed to copy an object in the class T.
Notify :
Error handling : Writes an error message and aborts if $size < 0$ and the symbol _DEBUG is defined, or the computer has not enough available memory.
Example : const int elVal = 13 ;
array<int> intArr (10, elVal) ;

Signature : inline array<T> (const array<T> & *elArr*) ;
Preconditions : None.
Result : Constructs a new array that is a copy of *elArr*. The new array will share memory block with the argument.
Time consumption : Constant time.
Notify : Copying is done in constant time.
Error handling :
Example : const array<int> arr1 (10, 10) ;
const array<int> arr2 (arr1) ;

Signature : inline array<T> & operator = (const array<T> & *elArr*) ;
Preconditions : None.
Result : Assigns the array on the left side to *elArr*. It means that the calling object is updated to be a copy of *elArr*. The calling object will also share memory block with *elArr*. If the calling object shared memory block with another object, the other object will now be the owner of the shared block.
Time consumption : If the calling object own its own memory block, it will first free it. This takes $O(size * T_T)$ time , where T_T is the time one needs to delete an object from the class T. If not, the deallocation of the old array takes constant time. The assignment to the new array is done in constant time.
Notify : The assignment takes constant time if the calling object already shares memory with another object.
Error handling :
Example : const array<int> arr1 (10, 10) ;
const array<int> arr2 (20, 20) ;
arr2 = arr1 ;

Signature : inline ãrray () ;
Preconditions : None.
Result : Removes the array object from the computer memory. The object cannot be used any longer, because it is uninitialized. If the calling object shared memory block with another object, the other object will now be the owner of the memory block.
Time consumption : If the calling object owns its own memory block, it will remove the block. This takes $O(size * T_T)$ time , where T_T is the time one needs to delete an object from the class T. If not, the deallocation of the array takes constant time.
Notify : To use the object again, one has to initialize it. This function should be used with care, because the object is no longer initialized, and the internal data structure does not satisfy the abstraction function used in the implementation of the class.
Error handling :
Example : array<int> arr (10, 10) ;
arr::ãrray() ;

Signature : array<T> operator + (const array<T> & *a*) const ;
Preconditions : None.
Result : Creates a new array that contains both the calling array and

the array a. The array a is appended to the result after the calling array.

Time consumption : $O(size * T_T)$, where T_T is the time one needs to copy an element of type T, and $size$ is the sum of the number of elements in the calling array and a.

Notify : The result of the concatenation owns its own memory block.

Error handling : Writes an error message and aborts if the computer not has enough available memory.

Example :
```
const array<int> arr1 ( 10, 10 ) ;
const array<int> arr2 ( 20, 20 ) ;
if ( (arr1 + arr2).size () != arr1.size() + arr2.size() )
    cout << "Something is very wrong !" << endl ;
```

Signature : inline T & operator [] (const int i) ;

Preconditions : $0 <= i < size()$.

Result : Returns a reference to the object in position i in the calling array.

Time consumption : If the calling array shares memory block with another array, the array will allocate a new memory block and copy all the elements from the old block into the new. This takes $O(size * T_T)$ time, where T_T is the time one needs to copy an element of type T, and $size$ is the number of objects in the array.

Notify : Since this function returns a reference to an object in the calling array, the result of this function may be used to **update the element** in position i in the array.

Error handling : If the precondition is not satisfied and the symbol _DEBUG is defined, an error message will be written and execution stops. If the precondition not is satisfied and the symbol _DEBUG is not defined, the behavior is not specified. If the array must allocate a new memory block and there is not enough available memory, an error message will be written and execution stops.

Example :
```
array<int> arr ( 10, 10 ) ;
arr [ 5 ] = 20 ;
```

Observer functions

Signature : int operator == (const array<T> & *elArr*) const ;

Preconditions : None.

Result : Returns a non-zero integer if the calling array is equal

to $elArr$, else zero.

Time consumption : $O(size * T_T)$ time, where T_T is the time one needs to compare two elements of type T, and $size$ is the number of objects in the array.

Notify : Two arrays are equal if they have the same number of elements, and the elements in the same positions are equal.

Error handling :

Example :
```
const array<int> a1 ( 10, 10 ) ;
const array<int> a2 ( 15, 20 ) ;
if ( a1 == a2 )
    cout << "Something is very wrong !" << endl ;
```

Signature : inline int size ( ) const ;

Preconditions : None.

Result : Returns the number of elements in the array.

Time consumption : Constant time.

Notify :

Error handling :

Example :
```
const array<int> a1 ( 10, 10 ) ;
    cout << "The size of a1 is : " << a.size () ;
```

Signature : inline const T & operator [] (const int i) const ;

Preconditions : $0 <= i < size()$.

Result : Returns a copy of the object in position i in the calling array.

Time consumption : Constant time.

Notify : This function is used to access elements in an array without updating the array.

Error handling : If the precondition is not satisfied and the symbol _DEBUG is defined, an error message will be written and execution stops. If the precondition is not satisfied and the symbol _DEBUG is not defined, the behavior is not specified.

Example :
```
const array<int> arr ( 10, 10 ) ;
const int arrEl = arr [ 5 ] ;
```

11.4 ERRORS AND LIMITATIONS

This class has been through a lot of tests, including tests using the program *purify*. This program tests for possible memory leaks and illegal memory accesses. If one chooses to

run an application using this class without having defined the symbol _DEBUG, the class can give no guarantee about the behavior of functions which receives illegal arguments.

11.5 REFERENCES

A good introduction to the use of reference counting may be found in :
"USENIX C++ Conference Proceedings"; Portland, Oregon 1992.
Page 131 to page 150.

11.6 FILES AND VERSIONS

User documentation : /tmp_mnt/Home/stud2/vmadsen/tensor/dok/array_userdoc.tex
Source code : /tmp_mnt/Home/stud2/vmadsen/tensor/src/array.h
Include files : iostream.h
stdlib.h

11.7 MAINTENANCE AND SUPPORT

This class is implemented and documented by :
Victor Madsen
Institute of Computer Science
University of Bergen
email : vmadsen@ii.uib.no

Chapter 12

User Documentation for Class : grid2d

12.1 NAME

Name : *grid2d*
Version : 1.0
Compiler : AT&T C++ v. 3.01
Arguments : template <**class Field, int n, int y , int x**> *class grid2d*
class Field : The class of the nodes in the grid.
int n : Integer that indicates the accuracy when calculating partial derivatives.
int y : The number of nodes in dimension one.
int x : The number of nodes in dimension zero.
Based on : Idea documentation for class *diffRing*.
Idea documentation for class *Field*.
Depends on : /tmp_mnt/Home/stud2/vmadsen/tensor/src/array.h
/tmp_mnt/Home/stud2/vmadsen/tensor/src/matrix.h
Description : Implements a scalar field as a grid with $x * y$ nodes. The implementation has a class signature that is a superset of the signature for *differential rings*.

12.2 PURPOSE

This class implements a grid in two dimensions with $x * y$ nodes. The nodes are elements from the class *Field*. A grid in two dimensions may be used to represent a two-dimensional differential ring. The class has a superset of the operations described in the idea documentation for the class *diffRing*, and it is recommended to also read the idea documentation. This class is implemented in a way that is efficient for calculations on constant grids (grids

that have equal value in every node). This class is because of this suitable for tensor field calculations. The class has also implemented a division operator.

12.3 DESCRIPTION

This class may be used to represent functions:

$$f : Field * Field \rightarrow Field$$

where $Field$ is a field. Usually $Field$ will be one of the built in types $float$ or $double$, but any class with field-properties will work. A two-dimensional grid with $x * y$ nodes is a discretization of the domain $Field * Field$ to be indexed by two integers in the range $[0, \ldots, y-1]$ and $[0, \ldots, x-1]$.

This grid class defines the domain of the functions to be in the interval $[0.0, \ldots, 1.0]$ in both dimensions. This implies that the *distance* between two neighboring nodes in the grid are:

Dimension 0 : $\frac{Field(1)}{Field(x-1)}$

Dimension 1 : $\frac{Field(1)}{Field(y-1)}$

This is no limitation, because one always may scale a wanted function to be in the chosen domain. Notice that we need a coercion from int to $Field$ in order to calculate the distance. This is done by a constructor that every field class has implemented. The distance between two nodes is needed in the evaluation of the partial derivatives. This class uses a finite difference method in the calculation of partial derivatives. It means that the partial derivative in a point (i, j) is calculated from the values of one or more neighbors in the actual dimension. The accuracy of the differentiation is determined by the parameter n to the class. The parameter tells the class how many neighboring nodes in each direction the class must use in the differentiation. This means that the class uses a $2 * n$ point differential operator in each dimension. If f.ex. $n = 1$, the class will use the following differential operator:

$$a.partialDiff(0)(i,j) = \frac{a(i,j+1) - a(i,j-1)}{Field(x-1)}$$

$$a.partialDiff(1)(i,j) = \frac{a(i+1,j) - a(i-1,j)}{Field(y-1)}$$

This differential operator is constructed by the aid of Taylor sequences, and this is the case for all the other possible $2n$ point differential operators in this class. Notice also that because we chose the interval to be $[0.0, \ldots, 1.0]$, it is quite easy to calculate the weights we use in the differential operators. Experiments have shown that these weights

are quite accurate, but when a grid represent waves with high frequencies, we may get diffraction. This happens even when $n = 4$, which usually gives very correct evaluation of the derivatives. To compensate for this, the class has built in a special differential operator which is constructed for the purpose of differentiating waves. This operator is chosen with $n = 7$, i.e. it is a 14 point finite difference operator. This operator is most known as the *Holberg operator*, and uses the following weights:

```
w [ -7 ] =   3.5709981e-3/h
w [ -6 ] =   0.0
w [ -5 ] =  -2.0637069e-2/h
w [ -4 ] =   0.0
w [ -3 ] =   0.10411822  /h
w [ -2 ] =   0.0
w [ -1 ] =  -1.2316663   /h
w [  0 ] =   0.0
w [  1 ] =   1.2316663   /h
w [  2 ] =   0.0
w [  3 ] =  -0.10411822  /h
w [  4 ] =   0.0
w [  5 ] =   2.0637069e-2/h
w [  6 ] =   0.0
w [  7 ] =  -3.5709981e-3/h
```

where $h = delta * 0.9975 * 2$, and *delta* is the distance between two neighboring nodes in the actual dimension. This class uses wraparound when calculating partial derivatives. I.e.:

- $a(-m, j) = a(y - m, j)$
- $a(i, -n) = a(i, x - n)$
- $a(y, j) = a(0, j)$
- $a(i, x) = a(i, 0)$

for all $0 \leq i, m < y$ and $0 \leq j, n < x$, and a is a grid with $y * x$ nodes.

The class has the following signature:

```
template <class Field, int y , int x> class grid2d
{
   public:

inline   grid2d                    (                    )       ;
inline   grid2d                    ( const grid2d   &   )       ;
inline   ~grid2d                   (                    )       ;
inline   int           operator ==  ( const grid2d   &   ) const ;
inline   grid2d &      operator =   ( const grid2d   &   )       ;
```

```
inline   grid2d                    ( const Field    &   )       ;
inline   grid2d                    ( const int          )       ;
         grid2d       operator -   (                    ) const ;
         grid2d       operator +   ( const grid2d   &   ) const ;
         grid2d       operator *   ( const grid2d   &   ) const ;
         grid2d       operator /   ( const grid2d   &   ) const ;
static
inline   int          numDimensions (                   )       ;
         grid2d       partialDiff  ( const int dim      ) const ;

inline   Field &      operator  ()  ( const int,const int)       ;
inline   Field        operator  ()  ( const int,const int) const ;

         grid2d       operator -   ( const grid2d   &   ) const ;
         grid2d &     operator +=  ( const grid2d   &   )       ;
         grid2d &     operator -=  ( const grid2d   &   )       ;
         grid2d &     operator *=  ( const grid2d   &   )       ;
         grid2d &     operator /=  ( const grid2d   &   )       ;
} ;
```

12.3.1 Time and Space Analysis

A technique used in the implementation is to observe if the objects in this class have equal value in each node (i.e. is a **constant grid**). If a grid is constant, we do not need to store $x * y$ values. Instead we may set a constant flag, and store only the value common for all nodes in the grid. We may then take benefit of this information in the member functions. We do f.ex. know that the partial derivatives of a constant grid returns a constant grid with zero in all nodes! It means we then may differentiate in constant time (independent of x and y). We also know that a zero grid multiplied with any other grid returns a new zero grid. If we have a constant grid with value one, we may also implement a more efficient multiplication. This is because $1 * a = a$. If we know that one of the arguments are equal to one, we only need to return the other argument without any calculation. We may find several of these special cases. They are listed in the figures below. Let e and f be constant grids not equal to zero or one, and let X and Y be **variable grids** (i.e. not constant grids). The constants 0 and 1 represents here their associated constant grids:

$+$	0	1	f	Y
0	0	1	f	Y
1	1	$1+1$	$1+f$	$1+Y$
e	e	$e+1$	$e+f$	$e+Y$
X	X	$X+1$	$X+f$	$X+Y$

-	0	1	f	Y
0	0	-1	$-f$	$-Y$
1	1	0	$1-f$	$1-Y$
e	e	$e-1$	$e-f$	$e-Y$
X	X	$X-1$	$X-f$	$X-Y$

*	0	1	f	Y
0	0	0	0	0
1	0	1	f	Y
e	0	e	$e*f$	$e*Y$
X	0	X	$X*f$	$X*Y$

/	0	1	f	Y
0	Err	0	0	0
1	Err	1	$1/f$	$1/Y$
e	Err	e	e/f	e/Y
X	Err	X	X/f	X/Y

Of these figures we may read when we need to actually perform any calculations. An expression in the form 'e *op* f' may be calculated with only one binary operation. Expressions like 'e *op* Y', 'X *op* f', and 'X *op* Y' may be calculated with $x * y$ binary operations. Still, one do not need to access any container structure to get the values of the nodes in e and f, since they are constant and is stored the value is stored in a variable that is not a container class. The net effects of this approach are that we save a lot of valuable computer memory, and we can often calculate the partial derivatives and binary operations very fast. The only possible drawback is that the implementation is a bit more complex.

In the case where we do have a variable grid, we have to store all $x * y$ elements. These elements are stored in an array. The array is implemented with the aid of reference counting , and this automatic gives us the benefits from the reference counting technique. Whenever this class performs a copy of a variable grid, it is a reference counting copy.

There may be applications where we know that all involved grids will be variable grids. In this case a grid implementation without the implementation details mentioned over will probably be more efficient. But there are situations where we know that we have a lot of constant grids. This is in particular true when we want to do tensor field calculations. There are several examples of tensors where most of the coordinates are zero, and in this case this grid class will be very suitable.

12.3.2 Template arguments

The class need the following arguments:

class Field : This class must satisfy the requirements given in the idea documentation for a field class.
int n : As mentioned above, this parameter specifies the number of nodes in each direction that a grid will use when calculating the partial derivatives of a grid.
int y : This integer specifies the number of nodes in dimension one.
int x : This integer specifies the number of nodes in dimension zero.

12.3.3 Operations

Generators

Signature	: grid2d< $Field, n, y, x$ > (const $Field$ & f) ;
Preconditions	: None.
Result	: Creates a constant grid with the value f in every node.
Time consumption	: $O(t)$ time, where t is the time one need to construct a copy of f.
Notify	: This function does **not** allocate an array with $x * y$ elements. Instead, a flag is set and f is stored.
Error handling	: None.
Example	: grid2d< $float, 2, 100, 100$ > ptFive (0.5) ;

Signature	: grid2d< $Field, n, y, x$ > ( ) ;
Preconditions	: None.
Result	: Creates a constant grid with the value $Field(0)$ in every node.
Time consumption	: $O(t)$ time, where t is the time one need to construct the value $Field(0)$.
Notify	: This function does **not** allocate an array with $x * y$ elements. Instead, a flag is set and f is stored.
Error handling	: None.
Example	: grid2d< $float, 2, 100, 100$ > g ;

Signature	: grid2d< $Field, n, y, x$ > (const int i) ;
Preconditions	: $0 \leq i$.
Result	: Creates a constant grid with the value $Field(i)$ in every node.
Time consumption	: $O(t)$ time, where t is the time one need to construct the value $Field(i)$.
Notify	: This function does **not** allocate an array with $x * y$ elements. Instead, a flag is set and f is stored.
Error handling	: If the precondition is not met, an error message is written to *cout*, and execution stops.
Example	: grid2d< $float, 2, 100, 100$ > $zero$ (0) ;

grid2d< $float, 2, 100, 100$ > two (2) ;

Signature : grid2d< $Field, n, y, x$ > (const grid2d< $Field, n, y, x$ > & g) ;
Preconditions : None.
Result : Constructs a copy of r.
Time consumption : Constant time.
Notify : This is the *copy constructor* for the class.
Error handling : None.
Example : grid2d< $float, 2, 100, 100$ > g1 ;
grid2d< $float, 2, 100, 100$ > g2 (g1) ;

Signature : $\tilde{\text{g}}$rid2d< $Field, n, y, x$ > ( ) ;
Preconditions : None.
Result : Removes the object from the computer memory. The object can not longer be used, because it is uninitialized.
Time consumption : If the grid is constant, the object is deleted in $O(t)$ time. If else, it is deleted in $O(x * y * t)$ time (where t is the time one need to delete one element in the class *Field.*
Notify : To use the object again, one has to initialize it. This function should be used with care, because the object is no longer initialized, and the internal data structure does not satisfy the abstraction function used in the implementation of the class.
Error handling : None.
Example : grid2d< $float, 2, 100, 100$ > r ;
r::$\tilde{\text{d}}$iffRing() ;

Signature : grid2d< $Field, n, y, x$ > &
operator = (const grid2d< $Field, n, y, x$ > & g) ;
Preconditions : None.
Result : Assigns the value of g to the calling object.
Time consumption : If the calling object not is constant , it will first delete the old nodes. This takes $O(x * y * t)$ time (where t is the time one need to delete one element in the class *Field.* If not, the deallocation of the old grid takes $O(t)$ time. The assignment to the new grid takes constant time.
Notify : This function returns a reference to the calling object.
Error handling : None.
Example : g1 = g2 ;

Signature : grid2d< $Field, n, y, x$ > operator - ( ) const ;
Preconditions : None.
Result : Returns a grid g with the values $g(i,j) = -f(i,j)$ in the nodes (where f is the calling object).
Time consumption : If the calling object is constant, the operation uses $O(t)$ time. If else,the operation uses $O(x * y * t)$ time, where t is the time one need to negate one element in the class *Field*.
Notify : If the calling object is a constant grid, then the result will be a constant grid.
Error handling : None.
Example : grid2d< $float, 2, 100, 100$ > g1, g2 ;
g1 = - g2 ;

Signature : grid2d< $Field, n, y, x$ >
operator + (const grid2d< $Field, n, y, x$ > & g) const ;
Preconditions : None.
Result : Returns a grid h with the values $h(i,j) = f(i,j) + g(i,j)$ in the nodes (where f is the calling object).
Time consumption : See the corresponding table in the time-space analysis.
Notify : See the corresponding table in the time-space analysis.
Error handling : If there are not enough available computer memory, an error message is written to *cout*, and execution stops.
Example : grid2d< $float, 2, 100, 100$ > g1, g2, g3 ;
g1 = g2 + g3 ;

Signature : grid2d< $Field, n, y, x$ >
operator * (const grid2d< $Field, n, y, x$ > & g) const ;
Preconditions : None.
Result : Returns a grid h with the values $h(i,j) = f(i,j) * g(i,j)$ in the nodes (where f is the calling object).
Time consumption : See the corresponding table in the time-space analysis.
Notify : See the corresponding table in the time-space analysis.
Error handling : If there are not enough available computer memory, an error message is written to *cout*, and execution stops.
Example : grid2d< $float, 2, 100, 100$ > g1, g2, g3 ;
g1 = g2 * g3 ;

Signature : grid2d< $Field, n, y, x$ >
operator / (const grid2d< $Field, n, y, x$ > & g) const ;
Preconditions : g must not contain the value $Field(0)$ in any nodes.

Result : Returns a grid h with the values $h(i,j) = f(i,j)/g(i,j)$ in the nodes (where f is the calling object).
Time consumption : See the corresponding table in the time-space analysis.
Notify : See the corresponding table in the time-space analysis.
Error handling : If there are not enough available computer memory, an error message is written to *cout*, and execution stops.
What happens if the precondition is not met is compiler dependent.
Example : grid2d< $float, 2, 100, 100$ > g1, g2, g3 (2) ;
g1 = g2 / g3 ;

Signature : *Field* & operator () (const int i,const int j) ;
Preconditions : $0 \leq i < y$ and $0 \leq j < x$.
Result : Returns a reference to the element in node (i,j) in the calling object. If the grid is constant, it is expanded to contain $x * y$ elements. This is done because one else may get unwanted side effects. The object is because of this not any longer a constant grid.
Time consumption : If the calling object is a constant grid, this operation is done in $O(x * y * t)$ time. If else, the operation is done in $O(t)$ time, where t is the time one need to construct a new Field-element.
Notify : Since this function returns a reference to an object in the calling grid, the result of this function may be used to **update the element** in position (i,j) in the grid.
Error handling : If the precondition is not met, an error message is written to *cout*, and execution stops.
Example : grid2d< $float, 2, 100, 100$ > g ;
g (99,99) = 3.14 ;

Signature : grid2d< $Field, n, y, x$ > partialDiff (const int dim) const ;
Preconditions : $0 \leq dim < 2$.
Result : Returns the partial derivative of the calling object in dimension dim.
Time consumption : If the calling object is a constant grid, this operation is done in $O(1)$ time, where t is the time one need to constuct a zero element in the field class. If else, the operation is done in $O(n * y * x * t)$ time, where t is the time one need to invert a field element.
Notify : If the calling object is a constant grid, the calculation is done in constant time.
Error handling : If the precondition is not met, an error message is written to *cout*, and execution stops.
Example : grid2d< $float, 2, 100, 100$ > g1, g2 ;

```
g1 = g2.partialDiff ( 0 ) + g2.partialDiff ( 1 ) ;
```

Observer functions

Signature : int operator == (const grid2d< $Field, n, y, x$ > & g) const ;
Preconditions : None.
Result : Returns a non-zero integer if the calling object is equal to r, if else the integer zero is returned.
Time consumption : If both arguments are constant grids, the test is done in $O(t)$ time, if else the test is done in $O(x * y * t)$ time (where t is the time one need to compare two elements in the field class.
Notify : This function tests for semantic equality.
Error handling : None.
Example :

```
grid2d< float, 2, 100, 100 > r ;
if ( ! ( r == r ) )
    cout << "Something is very wrong !" << endl ;
```

Signature : static int numDimensions ( ) ;
Preconditions : None.
Result : Returns the number 2, because this is the number of dimensions of all elements in the class.
Time consumption : Constant time.
Notify : This is a static member function. This implies that all objects in the class has the same number of dimensions.
Error handling : None.
Example :

```
cout << "The number of dimensions are : "
     << grid2d< float, 2, 100, 100 >::numDimensions () << endl ;
```

Signature : Field operator () (const int i ,const int j) const ;
Preconditions : $0 \leq i < y$ and $0 \leq j < x$.
Result : Returns the element in node (i, j) in the calling grid.
Time consumption : The operation is done in constant time.
Notify : This operation may not be used to update the grid.
Error handling : If the precondition is not met, an error message is written to *cout*, and execution stops.
Example :

```
const grid2d< float, 2, 100, 100 > g ;
float f ;
f = g ( 99,99 ) ;
```

Composed functions

Signature : grid2d< $Field, n, y, x >$
operator - (const grid2d< $Field, n, y, x >$ & g) const ;
Preconditions : None.
Result : Returns a grid h with the values $h(i,j) = f(i,j) - g(i,j)$ in the nodes (where f is the calling object).
Time consumption : See the corresponding table in the time-space analysis.
Notify : This function is implemented more efficient than the operation $f + (-g)$).
Error handling : None.
Example : grid2d< $float, 2, 100, 100 >$ g1, g2, g3 ;
g1 = g2 - g3 ;

Signature : grid2d< $Field, n, y, x >$ &
operator += (const grid2d< $Field, n, y, x >$ & g) ;
Preconditions : None.
Result : Updates the grid f to $f(i,j) = f(i,j) + g(i,j)$ in all the nodes (where f is the calling object).
Time consumption : This operation is more efficient than $a = a + b$, since this is an update function.
Notify : $f+ = g$ is implemented more efficient than the operation $f = f + g$.
Error handling : None.
Example : grid2d< $float, 2, 100, 100 >$ g1, g2 ;
g1 += g2 ;

Signature : grid2d< $Field, n, y, x >$ &
operator -= (const grid2d< $Field, n, y, x >$ & g) ;
Preconditions : None.
Result : Updates the grid f to $f(i,j) = f(i,j) - g(i,j)$ in all the nodes (where f is the calling object).
Time consumption : This operation is more efficient than $a = a - b$, since this is an update function.
Notify : $f- = g$ is implemented more efficient than the operation $f = f - g$.
Error handling : None.
Example : grid2d< $float, 2, 100, 100 >$ g1, g2 ;
g1 -= g2 ;

Signature : grid2d< $Field, n, y, x >$ &
operator *= (const grid2d< $Field, n, y, x >$ & g) ;
Preconditions : None.

Result : Updates the grid f to $f(i,j) = f(i,j) * g(i,j)$ in all the nodes (where f is the calling object).
Time consumption : This operation is more efficient than $a = a * b$, since this is an update function.
Notify : $f* = g$ is implemented more efficient than the operation $f = f * g$.
Error handling : None.
Example : grid2d< $float, 2, 100, 100$ > g1, g2 ;
g1 *= g2 ;

Signature : grid2d< $Field, n, y, x$ > &
operator /= (const grid2d< $Field, n, y, x$ > & g) ;
Preconditions : g must not contain the value $Field(0)$ in any nodes.
Result : Updates the grid f to $f(i,j) = f(i,j)/g(i,j)$ in all the nodes (where f is the calling object).
Time consumption : This operation is more efficient than $a = a/b$, since this is an update function.
Notify : $f/ = g$ is implemented more efficient than the operation $f = f/g$.
Error handling : What happens if the precondition is not met is compiler dependent.
Example : grid2d< $float, 2, 100, 100$ > g1, g2 (2) ;
g1 /= g2 ;

12.4 ERRORS AND LIMITATIONS

This class has been tested a lot, and realistic program examples have been written using this class. Most of the operations may hardly be implemented more efficient (because of the simplicity of some of the algorithms), but the operation *partialDiff* may perhaps be implemented a bit more efficient with some tuning. Because this class is implemented with a reference counting array class, the advantages and drawbacks of this technique also are valid for this class implementation (very effective copying, but not as efficient access as possible to nodes in a grid).

12.5 REFERENCES

For more information about the differential operators, contact Hans Munthe-Kaas.

An introduction to *differentiable rings* is given in:
/tmp_mnt/Home/stud2/vmadsen/tensor/dok/user_diffring.ps

A good introduction to the use of reference counting may be found in :
"USENIX C++ Conference Proceedings"; Portland, Oregon 1992.
Page 131 to page 150.

12.6 FILES AND VERSIONS

User documentation : /tmp_mnt/Home/stud2/vmadsen/tensor/dok/user_grid.tex
Source code : /tmp_mnt/Home/stud2/vmadsen/tensor/src/grid2d.h
Program example : /tmp_mnt/Home/stud2/vmadsen/tensor/eks1/seis2d.C
Include files : /tmp_mnt/Home/stud2/vmadsen/tensor/src/array.h
/tmp_mnt/Home/stud2/vmadsen/tensor/src/matrix.h
<iostream.h>
<stdlib.h>
<math.h>

12.7 MAINTENANCE AND SUPPORT

This class specification is documented by :
Victor Madsen
Institute of Computer Science
University of Bergen
email : vmadsen@ii.uib.no

Chapter 13

User Documentation for Class : tensor

13.1 NAME

Name	: *tensor*
Version	: 1.0
Compiler	: AT&T C++ v. 3.01
Based on	: Idea documentation for class *diffTensor*.
Arguments	: *template <class diffRing> class tensor*.
Depends on	: /tmp_mnt/Home/stud2/vmadsen/tensor/src/array.h /tmp_mnt/Home/stud2/vmadsen/tensor/src/matrix.h /tmp_mnt/Home/stud2/vmadsen/tensor/src/index.h /tmp_mnt/Home/stud2/vmadsen/tensor/src/christof.h
Description	: Implementation of a *differential tensor*.

13.2 PURPOSE

This class is an implementation based on the idea documentation for a differential tensor. This implementation is a template class which may be instantiated with a differential ring class. The template argument must because of this satisfy the specification for a differential ring.

13.3 DESCRIPTION

This class implementation complies with the specification for a differential tensor, and the idea documentation for differential tensors contains a complete description of the theory and use of this class. This class implementation does also contain some additional

operations which may be useful for tensor field calculations.

13.4 C++ CLASS SIGNATURE

This template class has the following class signature:

```
template <class diffRing> class tensor
{
   public:
inline   tensor                          (                          )       ;
inline   tensor                          ( const tensor  &          )       ;
inline   ~tensor                         (                          )       ;
inline   int               operator ==   ( const tensor  &          ) const ;
inline   tensor  &         operator =    ( const tensor  &          )       ;

         /*-----------------------------------------------------------*/
         /*                standard tensor operations                 */
         /*-----------------------------------------------------------*/
         //    Module operations
         tensor            zero          (                          ) const ;
         tensor            operator   -  (                          ) const ;
         tensor            operator   +  ( const tensor     &       ) const ;
friend   tensor<diffRing>  operator   *  ( const diffRing   &,
                                          const tensor<diffRing> &  )       ;
inline   int               dimension     (                          ) const ;
inline   array<diffRing>   coordinates   (                          ) const ;
         array<tensor>     basis         (                          ) const ;

         //   tensor operations
inline   tensor                          ( const diffRing &         )       ;
inline   operator          diffRing      (                          ) const ;
inline   int               numCoModules  (                          ) const ;
inline   int               numModules    (                          ) const ;
         tensor            contract      (                          ) const ;
         tensor            operator  *   ( const tensor  &          ) const ;
         tensor            operator  ()  ( const tensor  &          ) const ;
inline   static tensor     kronecker     (                          )       ;

         //   Differential operators
         tensor            lieDiff       ( const tensor  &          ) const ;
         tensor            partialDiff   ( const int          ,
                                 const christoffel<diffRing> &      ) const ;
         tensor            covDiff       (
                                 const christoffel<diffRing> &      ) const ;

         //   Composed functions
inline   int         isRing                (                          ) const ;
inline   int         isModule              (                          ) const ;
inline   int         isCoModule            (                          ) const ;
inline   int         equalType             ( const tensor     &       ) const ;
```

```
inline   int        canContract          (                    ) const ;
         tensor     operator   -         ( const tensor     & ) const ;
         tensor &   operator  -=         ( const tensor     & )       ;
         tensor &   operator  +=         ( const tensor     & )       ;
         tensor &   operator  *=         ( const diffRing   & )       ;

         /*-------------------------------------------------------*/
         /*                   Extra operations                    */
         /*-------------------------------------------------------*/

         //   generators
inline   tensor                     ( const int, const int,
                                      const array<diffRing>  & )       ;
inline   tensor     swapUpperIndex  ( const int , const int    ) const ;
inline   tensor     swapLowerIndex  ( const int , const int    ) const ;
static   tensor     identityTensor  ( const int , const int    )       ;
inline   tensor     invert          (                          ) const ;

         //   observers
inline   int                    canInvert          (           ) const ;
         int                    canMakeChristoffel (           ) const ;
         christoffel<diffRing>  makeChristoffel    (           ) const ;

         //   composed functions
inline   tensor                     ( const int, const int     )       ;
         tensor     gradient        (                          ) const ;
         tensor     divergence (const christoffel<diffRing> &  ) const ;
} ;
```

13.5 TEMPLATE ARGUMENTS

The class uses the following template argument:

```
template <class diffRing> class tensor
```

Class arguments

Name : *diffRing*
Description : This class must satisfy the specification for a differential ring.
Requirements : None.

13.6 OPERATIONS

13.6.1 Generators

Signature : tensor<diffRing> ( ) ;

Preconditions : None.
Result : Constructs a tensor of type (0,0) with the property:
diffRing (tensor ( )) == diffRing (0)
Notify : This is the *default constructor* for the class.
Example : tensor<diffRing> t ;

Signature : tensor<diffRing> (const tensor<diffRing> & *t*) ;
Preconditions : None.
Result : Constructs a copy of *t*.
Notify : This is the *copy constructor* for the class.
Example : tensor<diffRing> t1 ;
tensor<diffRing> t2 (t1) ;

Signature : $\tilde{\text{t}}$ensor<diffRing> ( ) ;
Preconditions : None.
Result : Removes the object from the computer memory. The object can not longer be used, because it is uninitialized.
Notify : To use the object again, one has to initialize it. This function should be used with care, because the object is no longer initialized, and the internal data structure does not satisfy the abstraction function used in the implementation of the class.
Example : tensor<diffRing> t ;
t::$\tilde{\text{t}}$ensor<diffRing>() ;

Signature : tensor<diffRing> & operator = (const tensor<diffRing> & *t*) ;
Preconditions : None.
Result : Assigns the value of *t* to the calling object.
Notify : This function returns a reference to the calling object.
Example : tensor<diffRing> t1, t2 ;
t1 = t2 ;

Signature : tensor<diffRing> (const diffRing & *r*) ;
Preconditions : None.
Result : Constructs a tensor of type (0,0) with the property:
diffRing (tensor (r)) == r
Notify : tensor(r).dimension() == 1.
Example : diffRing r ;
tensor<diffRing> t (r) ;

Signature : tensor<diffRing> zero ( ) const ;
Preconditions : None.
Result : Constructs a tensor with the following property:
t + t.zero () == t
for all tensors *t*.
Notify : t.zero().coordinates() == array<diffRing> (t.dimension(), diffRing (0))
Example : tensor<diffRing> t ;
cout << t.zero().coordinates () [0] << endl ;

Signature : tensor<diffRing> operator - ( ) const ;
Preconditions : None.
Result : Constructs a tensor with the following property:
t + (-t) == t.zero ()
for all tensors *t*.
Notify : (-t).coordinates() [i] == - (t.coordinates() [i])
Example : tensor<diffRing> t1, t2 ;
t1 = - t2 ;

Signature : tensor<diffRing> operator + (const tensor<diffRing> & *t*) const ;
Preconditions : The two tensor arguments must be of equal type.
Result : This member function has the following properties:
$a + b == b + a$
$a + (b + c) == (a + b) + c$
for all tensors a, b and c of equal type.
Notify : (t1+t2).coordinates()[i] == (t1.coordinates()[i])+(t2.coordinates()[i])
Example : tensor<diffRing> t1, t2, t3 ;
if (t2.equalType (t3))
t1 = t2 + t3 ;

Signature : friend tensor<diffRing>
operator * (const diffRing & *r* , const tensor<diffRing> & *t*) ;
Preconditions : None.
Result : This member function has the following properties:
m * (a + b) == m * a + m * b
(m + n) * a == m * a + n * a
(m * n) * a == m * (n * a)
for all elements m and n in the class *diffRing*, all tensors a and b of equal type..
Notify : (m*a).coordinates()[i] == m*(a.coordinates()[i])
Example : diffRing r ;

```
tensor<diffRing> t ;
t = r * t ;
```

Signature : array<tensor<diffRing> > basis (const int i) const ;
Preconditions : None.
Result : Returns the basis elements of the object.
Notify : tensors of different type have different basis elements.
Example :
```
tensor<diffRing> t, res ;
for ( int i = 0 ; i < t.dimension () ; ++ i )
    res += t.basis ( )[i] ;
```

Signature : tensor<diffRing> operator * (const tensor<diffRing> & t) const ;
Preconditions : None.
Result : Returns the *tensor product* of the calling tensor and t.
Notify : One may prove that the coordinates of the result may be expressed as:

$$(t_1 * t_2)^{i_1,\ldots,i_{r_1},m_1,\ldots,m_{r_2}}_{j_1,\ldots,j_{s_1},n_1,\ldots,n_{s_2}} = (t_1)^{i_1,\ldots,i_{r_1}}_{j_1,\ldots,j_{s_1}} * (t_2)^{m_1,\ldots,m_{r_2}}_{n_1,\ldots,n_{s_2}}$$

(with an implicit summation over all repeated indexes).See also the definition of *tensor product* given in the idea documentation.
Example :
```
tensor<diffRing> m,c ;
tensor<diffRing> t = m * c ;
```

Signature : tensor<diffRing> contract ( ) const ;
Preconditions : The calling tensor must be of type (r, s), where $0 < r, s$.
Result : Returns the *contraction* of the calling tensor.
Notify : One may prove that the coordinates of the result may be expressed as:

$$k(t)^{i_1,\ldots,i_{r-1}}_{j_1,\ldots,j_{s-1}} = t^{i_1,\ldots,i_{r-1},m}_{j_1,\ldots,j_{s-1},m}$$

(with an implicit summation over all repeated indexes).See also the definition of *contraction* given in the idea documentation.
Example :
```
tensor<diffRing> t = tensor<diffRing>::kronecker () ;
diffRing r = diffRing ( t.contract () ) ;
```

Signature : tensorr<diffRing> operator () (const tensorr<diffRing> & t) const ;
Preconditions : If the calling tensor is of type $(r1, s1)$, and t is of type $r2, s2)$, then we must have that $s2 \leq r1$ and $r2 \leq s1$.
Result : Calculates the inner product between the calling tensor and t.
Notify : One may prove that the coordinates of the result may be expressed as:

$$i(t_1, t_2)^{i_1,\ldots,i_m}_{j_1,\ldots,j_n} = (t_1)^{i_1,\ldots,i_m,k_1,\ldots,k_{s_2}}_{j_1,\ldots,j_n,l_1,\ldots,l_{r_2}} * t^{l_1,\ldots,l_{r_2}}_{k_1,\ldots,k_{s_2}}$$

(with an implicit summation over all repeated indexes).See also the definition of *inner product* given in the idea documentation.

Example : tensor<diffRing> t1, t2, t3 ;
t3 = t1 (t2) ;

Signature : static tensor<diffRing> kronecker ( ) ;
Preconditions : None.
Result : Returns a tensor c of type (1, 1), with the following properties:

$c\ (\ v\ ) == v$
$c\ (\ w\ ) == w$

for all tensors v of type (1, 0) and all tensors w of type (0, 1).
Notify : the coordinates of this tensor may be expressed as:

$$\delta^i_j = \begin{cases} 0 & \text{if } i <> j \\ 1 & \text{if } i = j \end{cases}$$

and is known as the *Kronecker tensor.*
Example : tensor<diffRing> t = tensorr<diffRing>::kronecker () ;

Signature : tensor<diffRing> lieDiff (const tensor<diffRing> & t) const ;
Preconditions : t is a tensor of type (1,0).
Result : Returns the Lie derivative of the calling tensor in direction t.
Notify : See also the definition in the idea documentation.
Example : tensor<diffRing> t1, t2, t3 ;
if (t3.isModule ())
t1 = t2.lieDiff (t3) ;

Signature : tensor<diffRing>
partialDiff (const int i, const christoffel<diffRing> & c) const ;
Preconditions : $0 \leq i <$ *diffRing::numDimensions ()*.
Result : Returns the partial derivative of the calling object in dimension i with regard to the christoffel symbol c.
Notify : See also the definition in the idea documentation.
Example : christoffel<diffRing> c ;
tensor<diffRing> t1, t2 ;
t1 = t2.partialDiff (0, c) ;

Signature : tensor<diffRing> covDiff (const christoffel<diffRing> & c) const ;
Preconditions : None.
Result : Returns the covariant derivative of the calling object with regard to

the christoffel symbol c.

Notify : See also the definition in the idea documentation.

Example :
```
christoffel<diffRing> c ;
tensor<diffRing> t1, t2 ;
t1 = t2.covDiff ( c ) ;
```

13.6.2 Observer functions

Signature : int operator == (const tensor<diffRing> & t) const ;

Preconditions : None.

Result : Returns a non-zero integer if the calling object is equal to m, if else the integer zero is returned.

Notify : Two tensors are equal if they are of equal type and they have equal coordinates.

Example :
```
tensor<diffRing> t ;
if ( ! ( t == t ) )
    cout << "Something is very wrong !" << endl ;
```

Signature : operator diffRing ( ) const ;

Preconditions : The calling tensor must be of type (0, 0).

Result : Returns the ring element which is isomorphic with the calling object.

Notify : diffRing (t) == t.coordinates () [0].

Example :
```
tensor<diffRing> t ;
diffRing r ;
if ( t.isRing () )
    r = diffRing ( t ) ;
```

Signature : int numCoModules ( ) const ;

Preconditions : None.

Result : Returns the number of comodule arguments the calling tensor may accept.

Notify : The type of a tensor t is equal to (t.numCoModules (), t.numModules ()).

Example :
```
tensor<diffRing> t ;
cout << t.numCoModules () << endl ;
```

Signature : int numModules ( ) const ;

Preconditions : None.

Result : Returns the number of module arguments the calling tensor may accept.
Notify : The type of a tensor t is equal to (t.numCoModules (), t.numModules ()).
Example : tensor<diffRing> t ;
cout << t.numModules () << endl ;

Signature : array<diffRing> coordinates ( ) const ;
Preconditions : None.
Result : Returns an array with the coordinates of the tensor. It is common to regard a tensor t of type (r, s) as a $(r + s)$ dimensional array indexed like this:
$t^{i_1,\ldots,i_r}_{j_1,\ldots,j_s}$
where $0 \leq i_1, \ldots, i_r, j_1, \ldots j_s < d$,where d is equal to *diffRing::numDimensions()*. This function will 'unfold' this array into the corresponding one dimensional array.
Notify : t.coordinates().size () == t.dimension().
Example : tensor<diffRing> t ;
array<diffRing> arr = t.coordinates () ;

13.6.3 Composed functions

Signature : int dimension ( ) const ;
Preconditions : None.
Result : Returns the dimension of the module the calling tensor belongs to.
Notify : A tensor of type (r,s) has dimension d^{r+s}, where d is equal to *diffRing::dimension()*.
Example : tensor<diffRing> t ;
cout << t.dimension () << endl ;

Signature : int isRing ( ) const ;
Preconditions : None.
Result : Returns a non-zero integer if the calling object is of type (0,0). If else, the integer zero is returned.
Notify : t.isRing () == (t.numCoModules () == 0 && t.numModules () == 0)
Example : tensor<diffRing> t ;
diffRing r ;
if (t.isRing ())
r = diffRing (t) ;

Signature : int isModule ( ) const ;
Preconditions : None.
Result : Returns a non-zero integer if the calling object is of type (1,0). If else, the integer zero is returned.
Notify : t.isModule () == (t.numCoModules () == 1 &&
t.numModules () == 0).
Example :
```
tensor<diffRing> t ;
if ( t.isModule () )
    cout << "OK !" << endl ;
```

Signature : int isCoModule ( ) const ;
Preconditions : None.
Result : Returns a non-zero integer if the calling object is of type (0,1). If else, the integer zero is returned.
Notify : t.isCoModule () == (t.numCoModules () == 0 &&
t.numModules () == 1).
Example :
```
tensor<diffRing> t ;
if ( t.isCoModule () )
    cout << "OK !" << endl ;
```

Signature : int equalType (const tensor<diffRing> & t) const ;
Preconditions : None.
Result : Returns a non-zero integer if the calling object has equal type as t. If else, the integer zero is returned.
Notify : t.equalType (t1) == (t.numCoModules () ==
t1.numCoModules () && t.numModules () == t1.numModules ())
Example :
```
tensor<diffRing> t1, t2, t3 ;
if ( t2.equalType ( t3 ) ) t1 = t2 + t3 ;
```

Signature : int canContract ( ) const ;
Preconditions : None.
Result : Returns a non-zero integer if the calling object has type (r, s), where $0 < r, s$.
Notify : t.canContract () == (0 < t.numCoModules () &&
0 < t.numCoModules ())
Example :
```
tensor t1, t2 ;
if ( t2.canContract () ) t1 = t2.contract () ;
```

Signature : tensor<diffRing> operator - (const tensor<diffRing> & t) const ;

Preconditions : The two argument tensors must be of equal type.
Result : This function is equal to: $t1 + (-t)$, where $t1$ is the calling object.
Example :
```
tensor<diffRing> t1, t2, t3 ;
if ( t1.equalType ( t2 ) ) t3 = t1 - t2 ;
```

Signature : tensor<diffRing> & operator -= (const tensor<diffRing> & t) ;
Preconditions : The two argument tensors must be of equal type.
Result : This function is equal to: $t1 = t1 + (-t)$, where $t1$ is the calling object.
Example :
```
tensor<diffRing> t1, t2 ;
if ( t1.equalType ( t2 ) ) t1 -= t2 ;
```

Signature : tensor<diffRing> & operator += (const tensor<diffRing> & t) ;
Preconditions : The two argument tensors must be of equal type.
Result : This function is equal to: $t1 = t1 * t$, where $t1$ is the calling object.
Example :
```
tensor<diffRing> t1, t2 ;
if ( t1.equalType ( t2 ) ) t1 += t2 ;
```

Signature : tensor<diffRing> & operator *= (const diffRing & r) ;
Preconditions : None.
Result : This function is equal to: $t = r * t$, where t is the calling object.
Example :
```
tensor<diffRing> t ;
diffRing r ;
t *= r ;
```

13.7 EXTRA OPERATIONS

13.7.1 Generators

Signature : tensor<diffRing> (const int r, const int s , const array<diffRing> & a) ;
Preconditions : $0 \leq r, s$ and $a.size() = d^{r+s}$, where d = *diffRing::numDimensions ()*. dimensions of *diffRing*.
Result : Constructs a tensor with the coordinates a.
Notify : tensor<diffRing> (r,s,a).coordinates () == a.
Example :
```
const int d == diffRing::numDimensions () ;
array<diffRing> a ( d * d, diffRing ( 1 ) ) ;
tensor<diffRing> t1 ( 2, 0, a ) ;
tensor<diffRing> t2 ( 1, 1, a ) ;
tensor<diffRing> t3 ( 0, 2, a ) ;
```

Signature : static tensor<diffRing> identityTensor (const int r, const int s) ;
Preconditions : $0 \leq r, s$.
Result : Constructs a tensor of type (r, s), with all coordinates equal to *diffRing (0)*, except the coordinates which is indexed with only equal indexes, which contains the value *diffRing (1)*.
Notify : The coordinates of the constructed tensor have the values:

$$t^{i_1,\ldots,i_r}_{j_1,\ldots,j_s} = \begin{cases} \textit{diffRing (1)} & \text{if } i_1 = \ldots = i_r = j_1 = \ldots = j_s \\ \textit{diffRing (0)} & \text{if else} \end{cases}$$

Note the close relation to the *Kronecker tensor*.
Example : tensor<diffRing> g = tensor<diffRing>::identityTensor(0,2) ;

Signature : tensor<diffRing> swapUpperIndex (const int p, const int q) const ;
Preconditions : $0 \leq p < q <$ *numCoModules ()*.
Result : Changes the position of the coordinates of the calling tensor like this:

$$t^{i_1,\ldots,i_p,\ldots i_q,\ldots,i_r}_{j_1,\ldots,j_s} = t^{i_1,\ldots,i_{p-1},i_q,i_{p+1},\ldots,i_{q-1},i_p,i_{q+1},\ldots,i_r}_{j_1,\ldots,j_s}$$

Notify : This function is often useful before a contraction.
Example : tensor<diffRing> t (4, 0) ;
t = t.swapUpperIndex (2,3) ;

Signature : tensor<diffRing> swapLowerIndex (const int s1, const int s2) const ;
Preconditions : $0 \leq p < q <$ *numModules ()*.
Result : Changes the position of the coordinates of the calling tensor like this:

$$t^{i_1,\ldots,i_r}_{j_1,\ldots,j_p,\ldots,j_q,\ldots,j_s} = t^{i_1,\ldots,i_r}_{j_1,\ldots,j_{p-1},j_q,j_{p+1},\ldots,j_{q-1},j_p,j_{q+1},\ldots,j_s}$$

Notify : This function is often useful before a contraction.
Example : tensor<diffRing> t (0, 4) ;
t = t.swapLowerIndex (2,3) ;

Signature : tensor<diffRing> invert ( ) const ;
Preconditions : The class *diffRing* must satisfy the specification for a field class, and *canInvert () != 0* .
Result : This function will create a matrix of the coordinates of the calling tensor t, invert the matrix, and construct a new tensor wich is the 'inverted' tensor t^{-1}. If t is of type (r, s), then t^{-1} will be of type (s, r). The inverted tensor will have the following property:

$$(t^{-1})_{ij} * t_{jp} = \delta_{ip}$$

Notify : This function is only implemented for tensors with two indexes.
Example :
```
tensor<diffRing> g ( 0, 2 ) ;
if ( g.canInvert () )
    tensor<diffRing> gInv = g.invert () ;
```

13.7.2 Observer functions

Signature : int canInvert ( ) const ;
Preconditions : The class *diffRing* must satisfy the specification for a field class.
Result : Returns a non-zero integer if there exist an inverse of the calling tensor. If else, the integer zero is returned.
Notify : This function is only implemented for tensors with two indexes.
Example :
```
tensor<diffRing> g ( 0, 2 ) ;
if ( g.canInvert () )
    tensor<diffRing> gInv = g.invert () ;
```

Signature : int canMakeChristoffel ( ) const ;
Preconditions : The class *diffRing* must satisfy the specification for a field class.
Result : Returns a non-zero integer if the calling tensor is of type $(0, 2)$ and may be inverted. If else, the integer zero is returned.
Notify : Is most often used in combination with the operation *makeChristoffel*.
Example :
```
tensor<diffRing> g ( 0, 2 ) ;
christoffel<diffRing> c ;
if ( g.canMakeChristoffel() )
    c = g.makeChristoffel () ;
```

Signature : christoffel<diffRing> makeChristoffel ( ) const ;
Preconditions : The class *diffRing* must satisfy the specification for a field class,

and *canMakeChristoffel () != 0* .

Result : Constructs a christoffel symbol c by the following formula:

$$c(n,i,j) = \frac{1}{2} * g^{nk} * [\frac{\partial}{\partial x^j}(g_{ki}) + \frac{\partial}{\partial x^i}(g_{jk}) - \frac{\partial}{\partial x^k}(g_{ij})]$$

where g_{ij} are the coordinates of the calling tensor and g^{ij} are the coordinates of the inverse of the calling tensor.

Notify : If the calling tensor is the *metric tensor*, then the result is valid christoffel symbols for the module generated by *diffRing*.

Example :
```
tensor<diffRing> g ( 0, 2 ) ;
christoffel<diffRing> c ;
if ( g.canMakeChristoffel() )
    c = g.makeChristoffel () ;
```

13.7.3 Composed functions

Signature : tensor<diffRing> (const int r, const int s) ;
Preconditions : $0 \leq r, s$
Result : Constructs a zero-tensor of type (r, s).
Notify : tensor<diffRing> (r,s).coordinates () == array<diffRing> (d^{r+s}, diffRing (0)).
Example : tensor<diffRing> t (2, 0) ;

Signature : tensor<diffRing> gradient ( ) const ;
Preconditions : The tensor must be of type $(0,0)$.
Result : Calculates the gradient of the calling tensor. This function is equal to *covDiff(c)*, for arbitrary christoffel symbols c.
Notify : Tensors of type $(0,0)$ does not need any cristoffel symbols when calculating the covariant derivatives.
Example :
```
tensor<diffRing> t ( 0, 0 ) ;
tensor<diffRing> t2 = t.gradient () ;
```

Signature : tensor<diffRing> divergence (const christoffel<diffRing> & c) const ;
Preconditions : The calling tensor must be of type $(r+1, s)$, where $0 \leq r, s$.
Result : This function is equal to : covDiff(c).contract ().
Notify : This function is implemented more efficient than simply composing the operations *covDiff* and *contract*.

Example : tensor<diffRing> t (2,3) ;
christoffel<diffRing> c ;
tensor<diffRing> t2 = t.divergence (c) ;

13.8 REFERENCES

A good introduction to algebraic specification may be found in:
David A. Watt : *Programming Language Syntax and Semantics*, Prentice Hall 1991.

Most of the mathematical background will be found in:
Serge Lang: *Algebra*, Addison Wesley 1965.
Abraham, Marsden and Ratiu : *Manifolds, Tensor Analysis, and Applications*, Springer-Verlag 1983.

For a more theoretical approach, see :
M. Haveraaen, V. Madsen, H. Munthe-Kaas : *Algebraic Programming Technology for Partial Differential Equations*, Precedings from "Norsk Informatikk Konferanse 1992".

For a less theoretical approach, see :
V. Madsen:*Tensors, from specification to implementation*; Thesis for the master degree, Institute of Informatics, University of Bergen.

13.9 FILES AND VERSIONS

Source code : /tmp_mnt/Home/stud2/vmadsen/tensor/src/tensor.h
User documentation : /tmp_mnt/Home/stud2/vmadsen/tensor/dok/tensor_user.ps
Based on : /tmp_mnt/Home/stud2/vmadsen/tensor/dok/difftensor_idea.ps
/tmp_mnt/Home/stud2/vmadsen/tensor/dok/diffring_idea.ps
/tmp_mnt/Home/stud2/vmadsen/tensor/dok/field_idea.ps
Example program : /tmp_mnt/Home/stud2/vmadsen/tensor/eks1/seis2d.C
Include files : /tmp_mnt/Home/stud2/vmadsen/tensor/src/array.h
/tmp_mnt/Home/stud2/vmadsen/tensor/src/matrix.h
/tmp_mnt/Home/stud2/vmadsen/tensor/src/index.h
/tmp_mnt/Home/stud2/vmadsen/tensor/src/christof.h
iostream.h
stdlib.h

13.10 MAINTENANCE AND SUPPORT

This class specification is documented by :
Victor Madsen
Institute of Informatics
University of Bergen
email : vmadsen@ii.uib.no

Chapter 14

Kildekode seismikkeksempel

```
//
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                 Parameters to batch job :                          //
//                                                                    //
////////////////////////////////////////////////////////////////////////
                                                                      //
const int    SIZE             =   400 ;                              //
const int    N                =     7 ;                              //
                                                                      //
const char * DATAFILE         =                                      //
"/tmp_mnt/Home/stud2/vmadsen/tensor/eks1/seis.dat" ;                  //
                                                                      //
////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////
//                                                                    //
//            Phycical constants are set to :                         //
//                                                                    //
////////////////////////////////////////////////////////////////////////
                                                  //                  //
const float  DENSITY_OVER   =    2.0000 ;         //   kkg/m^3        //
const float  DENSITY_BELOW  =    2.2000 ;         //   kkg/m^3        //
const float  VS_OVER        =    1.1547 ;         //   km/s           //
const float  VS_BELOW       =    1.5011 ;         //   km/s           //
const float  VP_OVER        =    2.0000 ;         //   km/s           //
const float  VP_BELOW       =    2.6000 ;         //   km/s           //
const float  RANGE          =    3.0000 ;         //   km             //
                                                  //                  //
////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////
//                                                                    //
//              Include files and typedef's :                         //
```

```
//                                                                    //
////////////////////////////////////////////////////////////////////////
                                                                      //
extern "C" {                                                          //
#include    <stdio.h>                                                 //
} ;                                                                   //
#include    <iostream.h>                                              //
#include    <math.h>                                                  //
#include    "/tmp_mnt/Home/stud2/vmadsen/tensor/src/grid2d.h"         //
#include    "/tmp_mnt/Home/stud2/vmadsen/tensor/src/tensor.h"         //
#include    "/tmp_mnt/Home/stud2/vmadsen/tensor/src/christof.h"       //
                                                                      //
////////////////////////////////////////////////////////////////////////
                                                                      //
typedef     float                           Ring         ;           //
typedef     grid2d<Ring,N,SIZE,SIZE>        diffRing     ;           //
typedef     tensor<diffRing>                Tensor       ;           //
typedef     christoffel<diffRing>           Christoffel  ;           //
                                                                      //
////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////
//                                                                    //
//                       Prototypes :                                 //
//                                                                    //
////////////////////////////////////////////////////////////////////////
                                                                      //
int         isBelowSection ( const int         i        ,           //
                             const int         j          ) ;       //
                                                                      //
void        initVs         ( diffRing        & vs         ) ;       //
                                                                      //
void        initVp         ( diffRing        & vp         ) ;       //
                                                                      //
void        initDensity    ( diffRing        & density    ) ;       //
                                                                      //
void        initMetric     ( Tensor          & g          ) ;       //
                                                                      //
void        initHook       ( Tensor          & hook     ,           //
                             const diffRing    & vs       ,           //
                             const diffRing    & vp       ,           //
                             const diffRing    & density  ,           //
                             const Tensor      & g          ) ;       //
                                                                      //
diffRing    defVal         ( const Ring        t          ) ;       //
                                                                      //
Tensor      timestep       ( const int         numSteps ,            //
                             const Ring        deltaT   ,            //
                             const Tensor      & g        ,          //
```

```
                              const Tensor    & Hook     ,          //
                              const diffRing  & density    ) ;      //
                                                                    //
void        postProcess    ( const Tensor    & g        ,          //
                              const Tensor    & res        ) ;      //
                                                                    //
void        saveDiffRing   ( const diffRing  & f          ) ;      //
                                                                    //
//////////////////////////////////////////////////////////////////////


int main ( void ) {

   int        numSteps    ;
   Ring       deltaT      ;
   diffRing   invDens     ,
              density     ,
              vs          ,
              vp          ;
   Tensor     hook        ,
              res         ,
              g           ;


   deltaT  = RANGE / ( sqrt ( 2 ) * VP_BELOW * SIZE ) ;
   deltaT *= 0.5                                      ;

   cout << "Delta T er : " << deltaT << endl ;

   numSteps = 450 ;

   initVs         ( vs             ) ;
   initVp         ( vp             ) ;
   initDensity    ( density        ) ;
   initMetric     ( g              ) ;
   initHook       ( hook        ,
                    vs          ,
                    vp          ,
                    density     ,
                    g             ) ;

   invDens = diffRing ( 1 ) / density       ;

   res = timestep ( numSteps    ,
                    deltaT      ,
                    g           ,
                    hook        ,
                    invDens       ) ;

   postProcess    ( g   , res     ) ;
```

```
   return 0 ;

}


/*-------------------------------------------------------------*/

int isBelowSection ( const int i, const int j )
{
   int res = 0 ;

   if ( ( 5 * ( SIZE / 8 ) ) < i )
      res = 1 ;

   return res ;
}

/*-------------------------------------------------------------*/

void initVs ( diffRing & vs )
{
   vs = diffRing ( ( Ring ) VS_OVER ) ;

   for ( int i = 0 ; i < SIZE ; ++ i ) {
      for ( int j = 0 ; j < SIZE ; ++ j ) {
         if ( isBelowSection ( i, j ) )
            vs ( i, j ) =  ( Ring ) VS_BELOW  ;
      }
   }
}

/*-------------------------------------------------------------*/

void initVp ( diffRing & vp )
{
   vp = diffRing ( ( Ring ) VP_OVER ) ;

   for ( int i = 0 ; i < SIZE ; ++ i ) {
      for ( int j = 0 ; j < SIZE ; ++ j ) {
         if ( isBelowSection ( i, j ) )
            vp ( i, j ) = ( Ring ) VP_BELOW ;
      }
   }
}

/*-------------------------------------------------------------*/


void initDensity ( diffRing & density )
{
   density = diffRing ( ( Ring ) DENSITY_OVER ) ;
```

```
   for ( int i = 0 ; i < SIZE ; ++ i ) {
      for ( int j = 0 ; j < SIZE ; ++ j ) {
         if ( isBelowSection ( i, j ) )
            density ( i, j ) = ( Ring ) DENSITY_BELOW  ;
      }
   }
}


/*-------------------------------------------------------------*/

void initMetric ( Tensor & g  )
{
   const diffRing scale = diffRing ( RANGE ) ;

   g  = Tensor::identityTensor ( 0, 2 ) ;
   g *= scale * scale                   ;
}

/*-------------------------------------------------------------*/

void initHook (       Tensor    & hook    ,
                const diffRing  & vs      ,
                const diffRing  & vp      ,
                const diffRing  & density ,
                const Tensor    & g         )
{

   const diffRing   my      ( vs * vs * density           ) ;
   const diffRing   lambda  ( vp * vp * density - my - my ) ;
   const Tensor     gInv    ( g.invert ()                 ) ;
   const diffRing   ptFive  ( diffRing(1) / diffRing(2)   ) ;

   hook  = my * gInv * gInv             ;
   hook  = hook.swapUpperIndex ( 1, 2 ) ;
   hook += ptFive * lambda *gInv * gInv ;
}

/*-------------------------------------------------------------*/

//   This variable is only needed in the function 'defVal',
//   but is declared global of efficiency reasons.

diffRing   expF ( 0 ) ;

/*-------------------------------------------------------------*/

diffRing defVal ( const Ring t  )
{
   Ring             arg    ,
```

```
                    val    ,
                    td     ,
                    time   ;

   const int  center_x  =  SIZE / 2 ;
   const int  center_y  =  SIZE / 2 ;
   Ring       freq                  ;

   freq  =  SIZE * VS_OVER / ( 4 * RANGE ) ;
   freq *= 0.7                             ;

   td    = sqrt ( 6.0 / PI ) / freq        ;
   time  = t - td                          ;
   arg   = PI * freq * ( time )            ;
   arg  *= arg                             ;
   val   = exp ( - arg ) * ( 1 - 2 * arg ) ;

   expF ( center_y    , center_x     ) = val ;
   expF ( center_y    , center_x + 1 ) = val ;
   expF ( center_y + 1, center_x     ) = val ;
   expF ( center_y + 1, center_x + 1 ) = val ;

   return expF ;
}

/*-------------------------------------------------------------*/

Tensor timestep   ( const int         numSteps    ,
                    const Ring        deltaT      ,
                    const Tensor    & g           ,
                    const Tensor    & hook        ,
                    const diffRing  & invDens       )
{
   Tensor             newU1                              ,
                      accl                               ,
                      strain                             ,
                      stress                             ,
                      u0          (  1, 0               ) ,
                      u1          (  1, 0               ) ;
   const diffRing     two_deltaT ( deltaT * deltaT      ) ;
   const Tensor       g_dual     ( g.invert ()          ) ;
   const Christoffel  c          ( g.makeChristoffel () ) ;


   for ( register int i = 0 ; i < numSteps ; ++ i ) {

      strain   = g.lieDiff ( u1 )         ;

      stress   = g_dual                   ;
      stress  *= defVal ( deltaT * i )    ;
      stress  += hook ( strain )          ;
```

```
      accl     = stress.divergence ( c )  ;
      accl    *= invDens                  ;

      newU1    = accl                     ;
      newU1   *= two_deltaT               ;
      newU1   += u1                       ;
      newU1   += u1                       ;
      newU1   -= u0                       ;

      u0       = u1                       ;
      u1       = newU1                    ;
   }

   return u1 ;
}

/*-------------------------------------------------------------*/

void postProcess ( const Tensor & g, const Tensor & t )
{
   diffRing  res ( t ( g ( t ) ) ) ;

   for ( register int i = 0 ; i < SIZE ; ++ i )
      for ( register int j = 0 ; j < SIZE ; ++ j )
         res ( i, j ) = sqrt ( res ( i, j ) ) ;

   saveDiffRing ( res ) ;
}

/*-------------------------------------------------------------*/

void saveDiffRing ( const diffRing & f )
{
   FILE * fptr = fopen ( DATAFILE , "w" ) ;

   if ( ! fptr )
      cout << "Error opening data-file .. " << endl ;
   else {
      for ( register int i = 0 ; i < SIZE ; ++ i ) {
         for ( register int j = 0 ; j < SIZE ; ++ j ) {
            fprintf ( fptr, "%g ", f ( i, j ) ) ;
         }
         fprintf ( fptr, "\n" ) ;
      }
      fclose ( fptr ) ;
   }
   return ;
}

/*-------------------------------------------------------------*/
```

```
//
```

Chapter 15

Kildekode : Klassebibliotek

Dette kapittelet inneholder kildekoden til følgende klasser:

- *array.h*
- *christof.h*
- *grid2d.h*
- *index.h*
- *matrix.h*
- *tensor.h*

15.1 array.h

```
//
//------------------------------------------------------------------------
//
//   Name       : array.h
//
//   Written by : Victor Madsen
//
//   Date       : 02.04.1992
//
//   Purpose    :
//
//     Implementation of an C-like array type with templates. This class is
//     implemented with reference-counting, which makes this class specially
//     good where it is done a lot of assignments of whole arrays. Inlining is
//     used whenever possible. Notice it is allowed with arrays of zero
//     elements. If the symbol _DEBUG is defined, it is done checks on the
//     parameters to the functions.
//
//------------------------------------------------------------------------


#ifndef _ARRAY_H
#define _ARRAY_H
```

```
#include <iostream.h>
#include <stdlib.h>

/*-------------------------------------------------------------*/
/*                                                             */
/*                       DECLARATION                           */
/*                                                             */
/*-------------------------------------------------------------*/


template <class T> class array
{
   public:

       /*-----------------------------------------------------*/
       /*              Basic operations                       */
       /*-----------------------------------------------------*/

inline array               (                         )       ;
inline array               ( const int     size      )       ;
       array               ( const int     size  ,
                             const T     & el        )       ;
inline array               ( const array & elArr     )       ;
inline ~array              (                         )       ;

       int     operator  == ( const array & elArr    ) const ;
inline array & operator  =  ( const array & elArr    )       ;

       /*-----------------------------------------------------*/
       /*            Characteristic operations                */
       /*-----------------------------------------------------*/

inline int        size         (                     ) const ;
       array      operator  +  ( const array &       ) const ;
inline T &        operator  [] ( const int           )       ;
inline const T &  operator  [] ( const int           ) const ;


   private:

       /*-----------------------------------------------------*/
       /*              Data representation                    */
       /*-----------------------------------------------------*/

       void   makeCopy  () ;

       int    arrSize ;
       int *  refCnt  ;
       T   *  data    ;

} ;


/*-------------------------------------------------------------------------*/
/*                                                                         */
/*                        SPECIFICATION                                    */
/*                                                                         */
/*-------------------------------------------------------------------------*/

/*-------------------------------------------------------------------------*/

template <class T> inline
array<T>::array ()
: arrSize ( 0 ),
  data    ( 0 ),
```

```
  refCnt  ( 0 )
{}

/*-------------------------------------------------------------------------*/

template <class T> inline
array<T>::array ( const int size )
: arrSize ( size                        ),
  data    ( 0 < size ? new T [ size ] : 0  ),
  refCnt  ( 0 < size ? new int        : 0  )
{

#ifdef _DEBUG
   if ( size < 0 ) {
      cout <<  "Error in array<T>::array ( const int size ).  size < 0 !" << endl ;
      exit (-1) ;
   }
#endif

   if ( 0 < size )
   {
      if ( ! data || ! refCnt )
      {
         cout <<  "Error in class array<T>.  Memory exhausted !" << endl ;
         exit (-1) ;
      }
      ( * refCnt ) = 1 ;
   }
}

/*-------------------------------------------------------------------------*/

template <class T> inline
array<T>::array   ( const array<T> & a )
: arrSize ( a.arrSize ),
  data    ( a.data    ),
  refCnt  ( a.refCnt  )
{
   if ( refCnt )
      ++ ( * refCnt ) ;
}

/*-------------------------------------------------------------------------*/

template <class T>
array<T>::array ( const int size, const T & el )
: arrSize ( size                          ),
  data    ( 0 < size ? new T [ size ] : 0  ),
  refCnt  ( 0 < size ? new int        : 0  )
{

#ifdef _DEBUG
   if ( size < 0 ) {
      cout <<  "Error in class array<T>::array ( const int, const T & ).  size < 0 !"
           << endl ;
      exit (-1) ;
   }
#endif

   if ( 0 < size )
   {
      if ( ! data || ! refCnt )
      {
         cout <<  "Error in class array<T>.  Memory exhausted !" << endl ;
         exit (-1) ;
```

```
      }

      ( * refCnt ) = 1 ;

      for ( register int i = 0 ; i < size ; i ++ )
    data[i] = el ;
   }
}

/*-------------------------------------------------------------------------*/

template <class T> inline
array<T>::~array  ()
{
   if ( arrSize ) {
      if ( ( * refCnt ) == 1 ) {
         delete [] data   ;
         delete    refCnt ;
      }
      else
         -- ( * refCnt ) ;
   }
}

/*-------------------------------------------------------------------------*/

template <class T>
int array<T>::operator  == ( const array<T> & a ) const
{
   register int res = 1 ;

   if ( arrSize != a.arrSize )
      res = 0 ;
   else
   {
      register int i = 0 ;

      while ( res && ( i < arrSize ) ) {
         if ( ! ( data[i] == a.data[i] ) )
            res = 0 ;
         ++ i ;
      }
   }

   return res ;
}

/*-------------------------------------------------------------------------*/

template <class T> inline
array<T> & array<T>::operator  =  ( const array<T> & a )
{
   if ( this != & a ) {

      if ( arrSize )       //   sletter gammel struktur
      {
         if ( ( * refCnt ) == 1 ) {
            delete [] data   ;
            delete    refCnt ;
         }
         else
            -- ( * refCnt ) ;
      }

      data    = a.data    ;
```

```
      refCnt  = a.refCnt  ;
      arrSize = a.arrSize ;

      if ( refCnt )
         ++ ( * refCnt ) ;
   }

   return *this ;
}

/*-------------------------------------------------------------------------*/

template <class T> inline
int array<T>::size () const
{
   return arrSize ;
}

/*-------------------------------------------------------------------------*/

template <class T>
array<T> array<T>::operator +  ( const array<T> & a ) const
{
   array<T>      tmpArray ( arrSize + a.arrSize ) ;

   for ( register int i = 0 ; i < arrSize ; ++ i )
      tmpArray.data [ i ] = data [ i ] ;

   for ( i = 0 ; i < a.arrSize ; ++ i )
      tmpArray.data [ i + arrSize ] = a.data [ i ] ;

   return tmpArray ;
}

/*-------------------------------------------------------------------------*/

template <class T> inline
T &  array<T>::operator  [] ( const int i )
{
#ifdef _DEBUG
   if ( i < 0 || arrSize <= i ) {
      cout << "Index out of bound in : array<T>::operator  [] ( const int i ) !"
           << endl ;
      exit ( -1 ) ;
   }
#endif

   if ( ( * refCnt ) != 1 )
      makeCopy () ;

   return  data[ i ] ;
}

/*-------------------------------------------------------------------------*/

template <class T> inline
const T & array<T>::operator  [] ( const int i ) const
{
#ifdef _DEBUG
   if ( i < 0 || arrSize <= i ) {
      cout << "Index out of bound in : array<T>::operator [] ( const int ) const !"
           << endl ;
      exit ( -1 ) ;
   }
#endif
```

```
   return data [ i ] ;
}

/*-------------------------------------------------------------------------*/

template <class T>
void   array<T>::makeCopy ()
{
   T * tmpArr = data ;

   -- ( * refCnt )            ;

   data   = new T [ arrSize ] ;
   refCnt = new  int          ;
   if ( ! data || ! refCnt )
   {
      cout <<  "Error in class array<T>.  Memory exhausted !" << endl ;
      exit (-1) ;
   }
   ( * refCnt ) = 1 ;

   for ( register int i = 0 ; i < arrSize ; ++ i )
      data [ i ] = tmpArr [ i ] ;
}

/*-------------------------------------------------------------------------*/

#endif
//
```

15.2 christof.h

```
//

//-------------------------------------------------------------------------
//
//   Name       : christof.h
//
//   Written by : Victor Madsen
//
//   Dato       : 03.04.1993
//
//   Purpose    :
//
//      This is a class that contains information about the deformation of a
//      ( not necessecearily metric ) space with d dimensions.  This class is
//      needed when one want to find the partial derivatives of a differential
//      module.  One may regard this class as an array in three
//      dimensions, with upper limit d in each dimension.
//
//-------------------------------------------------------------------------

#ifndef _CHRISTOFFEL_H
#define _CHRISTOFFEL_H


#include <iostream.h>
#include "/tmp_mnt/Home/stud2/vmadsen/tensor/src/array.h"



/*-------------------------------------------------------------------------*/
/*                                                                         */
/*                          DECLARATION                                    */
/*                                                                         */
/*-------------------------------------------------------------------------*/


template <class diffRing> class christoffel
{
   public:

        /*-----------------------------------------------------------------*/
        /*                        Basic operations                         */
        /*-----------------------------------------------------------------*/

inline   christoffel                    (                          )       ;
inline   christoffel                    ( const array<diffRing> & )       ;
inline   christoffel                    ( const christoffel     & )       ;
inline   ~christoffel                   (                          )       ;

inline   int             operator ==    ( const christoffel &      ) const ;
inline   christoffel  &  operator =     ( const christoffel &      )       ;


       /*-----------------------------------------------------------------*/
       /*            Characteristic christoffel operations                */
       /*-----------------------------------------------------------------*/

static
inline   int              spaceDimension (                          )       ;
inline   array<diffRing>  elements       (                          ) const ;
inline   diffRing        & operator  ()   ( const int, const int,
                                            const int                )       ;
inline   diffRing         operator  ()   ( const int, const int,
```

```
                                        const int             ) const ;

   private:

       /*-------------------------------------------------------------------*/
       /*                      Data representation                          */
       /*-------------------------------------------------------------------*/

      array<diffRing> c ;

} ;


/*-------------------------------------------------------------------------*/
/*                                                                         */
/*                          SPECIFICATION                                  */
/*                                                                         */
/*-------------------------------------------------------------------------*/

/*-------------------------------------------------------------------------*/

template <class diffRing> inline
christoffel<diffRing>::christoffel  (  )
: c ( diffRing::numDimensions () *
      diffRing::numDimensions () *
      diffRing::numDimensions ()  ,
      diffRing ( 0 )               )
{}

/*-------------------------------------------------------------------------*/

template <class diffRing> inline
christoffel<diffRing>::christoffel  ( const christoffel<diffRing> &  v )
: c ( v.c )
{}

/*-------------------------------------------------------------------------*/

template <class diffRing> inline
christoffel<diffRing>::christoffel ( const array<diffRing> & f )
: c   ( f )
{
   const int dim = diffRing::numDimensions () ;

   if ( dim * dim * dim != f.size () ) {
      cout << "Error in arg. to : christoffel ( const"
           << " array<diffRing> & )!" ;
      exit ( -1 ) ;
   }
}

/*-------------------------------------------------------------------------*/

template <class diffRing> inline
christoffel<diffRing>::~christoffel ()
{}

/*-------------------------------------------------------------------------*/

template <class diffRing>  inline
int   christoffel<diffRing>::operator  ==
                                ( const christoffel<diffRing> & v ) const
{
   int res = 0 ;
```

```
   if ( c == v.c )
      res = 1 ;

   return res ;
}

/*-------------------------------------------------------------------------*/

template <class diffRing>  inline
christoffel<diffRing> &  christoffel<diffRing>::operator  =  (
   const christoffel<diffRing> & v )
{
   c = v.c ;

   return ( *this ) ;
}

/*-------------------------------------------------------------------------*/

template <class diffRing>  inline
int      christoffel<diffRing>::spaceDimension ()
{
   return diffRing::numDimensions () ;
}

/*-------------------------------------------------------------------------*/

template <class diffRing> inline
array<diffRing>  christoffel<diffRing>::elements () const
{
   return c ;
}

/*-------------------------------------------------------------------------*/

template <class diffRing> inline
diffRing &  christoffel<diffRing>::operator () ( const int i ,
                                                 const int j ,
                                                 const int p   )
{
   const int dim = diffRing::numDimensions () ;

   if ( i < 0 || j < 0 || p < 0 || dim <= i || dim <= j || dim <= p )
   {
      cout << "Error in arg. to : diffRing &  christoffel<diffRing>"
           << "::operator () ( ... ) !" << endl ;
      exit ( -1 ) ;
   }

   return c [ ( i * dim + j ) * dim + p ] ;
}

/*-------------------------------------------------------------------------*/

template <class diffRing> inline
diffRing  christoffel<diffRing>::operator () ( const int i ,
                                               const int j ,
                                               const int p   ) const
{

   const int dim = diffRing::numDimensions () ;

   if ( i < 0 || j < 0 || p < 0 || dim <= i || dim <= j || dim <= p )
   {
      cout << "Error in arg. to : diffRing &  christoffel<diffRing>"
```

```
            << "::operator () ( ... ) !" << endl ;
        exit ( -i ) ;
    }

    return c [ ( i * dim + j ) * dim + p ] ;
}

/*-------------------------------------------------------------------------*/

#endif
//
```

15.3 grid2d.h

```
//

//-------------------------------------------------------------------------
//
//   Name       : grid2d.h
//
//   Written by : Victor Madsen
//
//   Date       : 03.04.1993
//
//   Purpose    :
//
//      Implements the class 'grid2d' as a two-dimensional array with
//      differentable ring properties.  It means that this class contains
//      all functions that a differential ring need.  This requires that
//      the template class argument must have field properties ( as
//      described in the file field_idea.ps ).  Calculation of the partial
//      deratives, do demand that the template type argument has the
//      division operators :
//
//         Ring     operator /    ( const Ring   &   ) const ;
//         Ring &   operator /=   ( const Ring   &   )       ;
//
//      In other words, the template argument type must be a field-type.
//
//
//      Notice that this class may create grids of all sizes in two
//      dimensions.
//
//      The first integer specifies the numer of nodes in each direction that
//      will be used when differentiating. The total numer of nodes
//      are then 2n + 1 ( including the current node ).
//      The second integer specifies the number of nodes in dimension one, and
//      the third integer gives the number of nodes in dimension zero. When
//      a particular node, the first integer specifies position in the first
//      dimension, and the last index the position in dimension zero.
//
//      Also note that this class uses wraparound when differentiating.
//
//      This implementation is optimized for binary operations and differentiation
//      for constant fields ( fields with same value in every point ).
//
//      Experiments have shown that if the differentable ring have wave-patterns,
//      we will get diffraction if 1 < n.  This is because of our numerical
//      method for calculation of differentiation weights.  Diffraction means
//      that waves with different frequencies will propagate with different
//      speeds.  A choice of n = 1 will in most cases be acceptable, if x and y
//      or the frequence are large. This will give some numerical errors.  To
//      compensate for this, with n = 7, a different numerical method that supports
//      differentiation of waves is used.
//
//-------------------------------------------------------------------------


#ifndef _GRID2D_H
#define _GRID2D_H


#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include "/tmp_mnt/Home/stud2/vmadsen/tensor/src/array.h"
#include "/tmp_mnt/Home/stud2/vmadsen/tensor/src/matrix.h"
```

```
/*-------------------------------------------------------------------------*/
/*                                                                         */
/*                          DECLARATION                                    */
/*                                                                         */
/*-------------------------------------------------------------------------*/


template <class Field, int n, int y , int x> class grid2d
{
   public:

        /*-----------------------------------------------------*/
        /*                Basic operations                     */
        /*-----------------------------------------------------*/

inline  grid2d                  ( const Field    &   )       ;

inline  grid2d                  (                    )       ;
inline  grid2d                  ( const int          )       ;
inline  grid2d                  ( const grid2d   &   )       ;
inline  ~grid2d                 (                    )       ;

inline  int       operator  ==  ( const grid2d   &   ) const ;
inline  grid2d &  operator  =   ( const grid2d   &   )       ;

        /*-----------------------------------------------------*/
        /*                  diffField operations               */
        /*-----------------------------------------------------*/

        grid2d    operator -    (                    ) const ;
        grid2d    operator -    ( const grid2d   &   ) const ;
        grid2d    operator +    ( const grid2d   &   ) const ;
        grid2d    operator *    ( const grid2d   &   ) const ;

        grid2d &  operator +=   ( const grid2d   &   )       ;
        grid2d &  operator -=   ( const grid2d   &   )       ;
        grid2d &  operator *=   ( const grid2d   &   )       ;

static
inline  int       numDimensions (                    )       ;
        grid2d    partialDiff   ( const int dim      ) const ;


        /*-----------------------------------------------------*/
        /*               Division operators                    */
        /*-----------------------------------------------------*/

        grid2d     operator /    ( const grid2d   &   ) const ;
        grid2d &   operator /=   ( const grid2d   &   )       ;

        /*-----------------------------------------------------*/
        /*               Grid2d operations                     */
        /*-----------------------------------------------------*/

inline  int          numElements    (                   ) const ;
inline  int          numDimElements ( const int         ) const ;
inline  Field  &     operator  ()   ( const int,const int)       ;
inline  Field        operator  ()   ( const int,const int) const ;

   private:

      /*-------------------------------------------------------------*/
```

```
      /*                  Implementation details                          */
      /*------------------------------------------------------------------*/

static   array<Field> calculateWeights ( const int ) ;

static   const array<Field>  weight_x   ;
static   const array<Field>  weight_y   ;
         array<Field>        arr        ;
         int                 isConstant ;
         Field               val        ;

} ;


/*-------------------------------------------------------------------------*/
/*                                                                         */
/*                         SPECIFICATION                                   */
/*                                                                         */
/*-------------------------------------------------------------------------*/


/*-------------------------------------------------------------------------*/

template <class Field, int n, int y, int x> inline
grid2d<Field,n,y,x>::grid2d ()
: arr        ( 0 ) ,
  isConstant ( 1 ) ,
  val        ( 0 )
{}

/*-------------------------------------------------------------------------*/

template <class Field, int n, int y, int x> inline
grid2d<Field,n,y,x>::grid2d ( const Field & el )
: arr        ( 0  ) ,
  isConstant ( 1  ) ,
  val        ( el )
{}

/*-------------------------------------------------------------------------*/

template <class Field, int n, int y, int x> inline
grid2d<Field,n,y,x>::grid2d ( const int el )
: arr        ( 0  ) ,
  isConstant ( 1  ) ,
  val        ( el )
{}

/*-------------------------------------------------------------------------*/

template <class Field, int n, int y, int x> inline
grid2d<Field,n,y,x>::grid2d   ( const grid2d<Field,n,y,x> & a )
: arr        ( a.arr        ) ,
  isConstant ( a.isConstant ) ,
  val        ( a.val        )
{}

/*-------------------------------------------------------------------------*/

template <class Field, int n, int y, int x> inline
grid2d<Field,n,y,x>::~grid2d  ()
{}

/*-------------------------------------------------------------------------*/
```

```
template <class Field, int n, int y, int x> inline
int grid2d<Field,n,y,x>::operator  == ( const grid2d<Field,n,y,x> & a ) const
{
   int res = 0 ;

   if ( isConstant && a.isConstant )
      res = ( val == a.val ) ;

   if ( isConstant && ! a.isConstant )
      res = ( array<Field> ( x*y, val ) == a.arr ) ;

   if ( ! isConstant && a.isConstant )
      res = ( arr == array<Field> ( x*y, a.val )  ) ;

   if ( ! isConstant && ! a.isConstant )
      res = ( arr == a.arr ) ;

   return res ;
}

/*-------------------------------------------------------------------------*/

template <class Field, int n, int y, int x> inline
grid2d<Field,n,y,x> & grid2d<Field,n,y,x>::operator  =
      ( const grid2d<Field,n,y,x> & a )
{
   arr        = a.arr        ;
   val        = a.val        ;
   isConstant = a.isConstant ;

   return *this ;
}

/*-------------------------------------------------------------------------*/

template <class Field, int n, int y, int x>
grid2d<Field,n,y,x>  grid2d<Field,n,y,x>::operator -   ( ) const
{

   grid2d<Field,n,y,x>   tmp ( * this ) ;

   if ( isConstant )

      tmp.val = - val ;

   else {

      const register int max = x * y ;

      for ( register int i = 0 ; i < max ; ++ i )
         tmp.arr [ i ]  =  - arr [ i ] ;
   }

   return tmp ;
}

/*-------------------------------------------------------------------------*/

template <class Field, int n, int y, int x>
grid2d<Field,n,y,x>  grid2d<Field,n,y,x>::operator +
       ( const grid2d<Field,n,y,x>  & g ) const
{

   if ( g.isConstant && ( g.val == Field ( 0 ) ) )
   {
```

```
      return ( * this ) ;
   }

   if ( isConstant && ( val == Field ( 0 ) ) )
   {
      return g ;
   }

   if ( isConstant && g.isConstant )
   {
      return grid2d<Field,n,y,x> ( val + g.val ) ;
   }

   if ( isConstant && ! g.isConstant )
   {
      grid2d<Field,n,y,x>   tmp ( g     ) ;
      const register int    max = x * y   ;

      for ( register int i = 0 ; i < max ; ++ i )
         tmp.arr [ i ]  += val ;

      return tmp ;
   }

   if ( ! isConstant && g.isConstant )
   {
      grid2d<Field,n,y,x>   tmp ( * this ) ;
      const register int    max = x * y    ;

      for ( register int i = 0 ; i < max ; ++ i )
         tmp.arr [ i ]  +=  g.val ;

      return tmp ;
   }

   if ( ! isConstant && ! g.isConstant )
   {
      grid2d<Field,n,y,x>   tmp ( * this ) ;
      const register int    max = x * y    ;

      for ( register int i = 0 ; i < max ; ++ i )
         tmp.arr [ i ]  +=  g.arr [ i ] ;

      return tmp ;
   }

   return ( * this ) ; //   Dummy statement, will never happen
}

/*-------------------------------------------------------------------------*/

template <class Field, int n, int y, int x>
grid2d<Field,n,y,x>  grid2d<Field,n,y,x>::operator *
      ( const grid2d<Field,n,y,x>  & g ) const
{

   if ( isConstant && g.isConstant )
   {
      return grid2d<Field,n,y,x> ( val * g.val ) ;
   }

   if ( isConstant && ! g.isConstant )
   {

      if ( val == Field ( 0 ) )
```

```
      {
 return ( * this ) ;
      }

      if ( val == Field ( 1 ) )
      {
 return g ;
      }

      grid2d<Field,n,y,x> tmp ( g     ) ;
      const register int  max = x * y   ;

      for ( register int i = 0 ; i < max ; ++ i )
         tmp.arr [ i ]  *= val ;

      return tmp ;
   }

   if ( ! isConstant && g.isConstant )
   {
      if ( g.val == Field ( 0 ) )
      {
         return g ;
      }

      if ( g.val == Field ( 1 ) )
      {
         return ( *this ) ;
      }

      grid2d<Field,n,y,x> tmp ( * this ) ;
      const register int  max = x * y    ;

      for ( register int i = 0 ; i < max ; ++ i )
         tmp.arr [ i ]  *= g.val ;

      return tmp ;
   }

   if ( ! isConstant && ! g.isConstant )
   {
      grid2d<Field,n,y,x> tmp ( * this ) ;
      const register int  max = x * y    ;

      for ( register int i = 0 ; i < max ; ++ i )
         tmp.arr [ i ]  *= g.arr [ i ] ;

      return tmp ;
   }

   return ( * this ) ;
}

/*--------------------------------------------------------------------------*/

template <class Field, int n, int y, int x>
grid2d<Field,n,y,x>  grid2d<Field,n,y,x>::operator -
       ( const grid2d<Field,n,y,x>  & g ) const
{

   if ( g.isConstant && ( g.val == Field ( 0 ) ) )
   {
      return ( * this ) ;
   }
```

```
   if ( isConstant && g.isConstant )
   {
      return grid2d<Field,n,y,x> ( val - g.val ) ;
   }

   if ( isConstant && ! g.isConstant )
   {
      grid2d<Field,n,y,x>   tmp           ;
      const register int    max = x * y   ;

      tmp.isConstant = 0                  ;
      tmp.arr        = array<Field> ( max ) ;

      for ( register int i = 0 ; i < max ; ++ i )
      {
         tmp.arr [ i ]   = val         ;
         tmp.arr [ i ]  -= g.arr [ i ] ;
      }

      return tmp ;
   }

   if ( ! isConstant && g.isConstant )
   {
      grid2d<Field,n,y,x>   tmp ( * this ) ;
      const register int    max = x * y    ;

      for ( register int i = 0 ; i < max ; ++ i )
         tmp.arr [ i ]  -=  g.val ;

      return tmp ;
   }

   if ( ! isConstant && ! g.isConstant )
   {
      grid2d<Field,n,y,x>   tmp ( * this ) ;
      const register int    max = x * y    ;

      for ( register int i = 0 ; i < max ; ++ i )
         tmp.arr [ i ]  -=  g.arr [ i ] ;

      return tmp ;
   }

   return ( * this ) ; //   Dummy statement, will never happen
}

/*-------------------------------------------------------------------------*/

template <class Field, int n, int y, int x>
grid2d<Field,n,y,x>  grid2d<Field,n,y,x>::operator /
      ( const grid2d<Field,n,y,x>  & g ) const
{

   if ( isConstant && g.isConstant )
   {
      return grid2d<Field,n,y,x> ( val / g.val ) ;
   }

   if ( isConstant && ! g.isConstant )
   {

      if ( val == Field ( 0 ) )
      {
         return ( * this ) ;
```

```
      }

      grid2d<Field,n,y,x> tmp           ;
      const register int  max = x * y   ;

      tmp.isConstant = 0                ;
      tmp.arr        = array<Field> ( max ) ;

      for ( register int i = 0 ; i < max ; ++ i )
      {
         tmp.arr [ i ]   = val          ;
         tmp.arr [ i ]  /= g.arr [ i ] ;
      }

      return tmp ;
   }

   if ( ! isConstant && g.isConstant )
   {
      if ( g.val == Field ( 1 ) )
      {
         return ( *this ) ;
      }

      grid2d<Field,n,y,x> tmp ( * this ) ;
      const register int  max = x * y   ;

      for ( register int i = 0 ; i < max ; ++ i )
         tmp.arr [ i ]  /= g.val ;

      return tmp ;
   }


   if ( ! isConstant && ! g.isConstant )
   {
      grid2d<Field,n,y,x> tmp ( * this ) ;
      const register int  max = x * y   ;

      for ( register int i = 0 ; i < max ; ++ i )
         tmp.arr [ i ]  /= g.arr [ i ] ;

      return tmp ;
   }

   return ( * this ) ;
}

/*-------------------------------------------------------------------------*/

template <class Field, int n, int y, int x>
grid2d<Field,n,y,x> & grid2d<Field,n,y,x>::operator +=
       ( const grid2d<Field,n,y,x>  & g )
{

   if ( g.isConstant && ( g.val == Field ( 0 ) ) )
   {
      return ( * this ) ;
   }

   if ( isConstant && g.isConstant )
   {
      val += g.val ;
      return ( * this ) ;
   }
```

```
   if ( isConstant && ! g.isConstant )
   {
      const register int    max = x * y    ;

      isConstant = 0                       ;
      arr        = array<Field> ( max ) ;

      for ( register int i = 0 ; i < max ; ++ i )
      {
         arr [ i ]   = val         ;
         arr [ i ]  += g.arr [ i ] ;
      }

      return ( * this ) ;
   }

   if ( ! isConstant && g.isConstant )
   {
      const register int    max = x * y    ;

      for ( register int i = 0 ; i < max ; ++ i )
         arr [ i ]  +=  g.val ;

      return ( * this ) ;
   }

   if ( ! isConstant && ! g.isConstant )
   {
      const register int    max = x * y    ;

      for ( register int i = 0 ; i < max ; ++ i )
         arr [ i ]  +=  g.arr [ i ] ;

      return ( * this ) ;
   }

   return ( * this ) ; //   Dummy statement, will never happen
}

/*-------------------------------------------------------------------------*/

template <class Field, int n, int y, int x>
grid2d<Field,n,y,x> & grid2d<Field,n,y,x>::operator *=
       ( const grid2d<Field,n,y,x>  & g )
{

   if ( isConstant && g.isConstant )
   {
      val *= g.val ;
      return ( * this ) ;
   }

   if ( isConstant && ! g.isConstant )
   {

      if ( val == Field ( 0 ) )
      {
         return ( * this ) ;
      }

      if ( val == Field ( 1 ) )
      {
         * this = g ;
         return ( * this ) ;
```

```
      }

      const register int  max = x * y   ;

      isConstant = 0     ;
      arr        = g.arr ;

      for ( register int i = 0 ; i < max ; ++ i )
         arr [ i ]  *= val ;

      return ( * this ) ;
   }

   if ( ! isConstant && g.isConstant )
   {
      if ( g.val == Field ( 0 ) )
      {
         * this = g ;
         return ( * this ) ;
      }

      if ( g.val == Field ( 1 ) )
      {
         return ( *this ) ;
      }

      const register int  max = x * y    ;

      for ( register int i = 0 ; i < max ; ++ i )
         arr [ i ]  *= g.val ;

      return ( * this ) ;
   }

   if ( ! isConstant && ! g.isConstant )
   {
      const register int  max = x * y    ;

      for ( register int i = 0 ; i < max ; ++ i )
         arr [ i ]  *= g.arr [ i ] ;

      return ( * this ) ;
   }

   return ( * this ) ;

}

/*-------------------------------------------------------------------------*/

template <class Field, int n, int y, int x>
grid2d<Field,n,y,x> & grid2d<Field,n,y,x>::operator -=
       ( const grid2d<Field,n,y,x>  & g )
{

   if ( g.isConstant && ( g.val == Field ( 0 ) ) )
   {
      return ( * this ) ;
   }

   if ( isConstant && g.isConstant )
   {
      val -= g.val ;
      return ( * this ) ;
   }
```

```
   if ( isConstant && ! g.isConstant )
   {
      const register int    max = x * y   ;

      isConstant = 0                    ;
      arr        = array<Field> ( max ) ;

      for ( register int i = 0 ; i < max ; ++ i )
      {
         arr [ i ]   = val         ;
         arr [ i ]  -= g.arr [ i ] ;
      }

      return ( * this ) ;
   }

   if ( ! isConstant && g.isConstant )
   {
      const register int    max = x * y    ;

      for ( register int i = 0 ; i < max ; ++ i )
         arr [ i ]  -=  g.val ;

      return ( * this ) ;
   }

   if ( ! isConstant && ! g.isConstant )
   {
      const register int    max = x * y    ;

      for ( register int i = 0 ; i < max ; ++ i )
         arr [ i ]  -=  g.arr [ i ] ;

      return ( * this ) ;
   }

   return ( * this ) ; //   Dummy statement, will never happen
}

/*-------------------------------------------------------------------------*/

template <class Field, int n, int y, int x>
grid2d<Field,n,y,x> & grid2d<Field,n,y,x>::operator /=
       ( const grid2d<Field,n,y,x>  & g )
{

   if ( isConstant && g.isConstant )
   {
      val /= g.val ;
      return ( * this ) ;
   }

   if ( isConstant && ! g.isConstant )
   {

      if ( val == Field ( 0 ) )
      {
         return ( * this ) ;
      }

      const register int  max = x * y   ;

      isConstant = 0                    ;
      arr        = array<Field> ( max ) ;
```

```
      for ( register int i = 0 ; i < max ; ++ i )
      {
         arr [ i ]   = val         ;
         arr [ i ]  /= g.arr [ i ] ;
      }

      return ( * this ) ;
   }

   if ( ! isConstant && g.isConstant )
   {
      if ( g.val == Field ( 1 ) )
      {
         return ( *this ) ;
      }

      const register int  max = x * y    ;

      for ( register int i = 0 ; i < max ; ++ i )
         arr [ i ]  /= g.val ;

      return ( * this ) ;
   }

   if ( ! isConstant && ! g.isConstant )
   {
      const register int  max = x * y    ;

      for ( register int i = 0 ; i < max ; ++ i )
         arr [ i ]  /= g.arr [ i ] ;

      return ( * this ) ;
   }

   return ( * this ) ;
}

/*--------------------------------------------------------------------------*/

template <class Field, int n, int y, int x> inline
int grid2d<Field,n,y,x>::numDimensions     ( )
{
   return 2 ;
}

/*--------------------------------------------------------------------------*/

template <class Field, int n, int y, int x> inline
int grid2d<Field,n,y,x>::numElements     ( ) const
{
   return ( y * x ) ;
}

/*--------------------------------------------------------------------------*/

template <class Field, int n, int y, int x> inline
int grid2d<Field,n,y,x>::numDimElements     ( const int dim ) const
{
   if ( dim < 0 || 1 < dim ) {
      cout << "Error in arg. to grid2d<Field,n,y,x>::numDimElements "
           << " ( const int dim ) !" << endl ;
      exit ( -1 ) ;
   }
```

```
   return ( dim == 0 ? x : y )  ;
}

/*------------------------------------------------------------------------*/

template <class Field, int n, int y, int x> inline
Field & grid2d<Field,n,y,x>::operator ()    ( const int i, const int j )
{
   if ( i < 0 || j < 0 || y <= i || x <= j ) {
     cout << "Error in arg. to grid2d<Field,n,y,x>::operator () ( ... ) !" ;
     exit ( -1 ) ;
   }

   if ( isConstant )
   {
      isConstant = 0                          ;
      arr        = array<Field> ( y * x, val ) ;
   }

   return arr [ i * x + j ] ;
}

/*------------------------------------------------------------------------*/

template <class Field, int n, int y, int x>
Field grid2d<Field,n,y,x>::operator ()    ( const int i, const int j ) const
{
   if ( i < 0 || j < 0 || y <= i || x <= j ) {
     cout << "Error in arg. to grid2d<Field,n,y,x>::operator () ( ... ) const !" ;
     exit ( -1 ) ;
   }
   if ( isConstant )
      return val ;
   else
      return ( arr [ i * x + j ] ) ;
}

/*------------------------------------------------------------------------*/

template <class Field, int n, int y, int x>
grid2d<Field,n,y,x> grid2d<Field,n,y,x>::partialDiff  ( const int dim ) const
{
   if ( dim < 0 || 1 < dim ) {
      cout << "Error in arg. to grid2d<Field,n,y,x>::partialDiff "
           << " ( const int dim ) const !" << endl ;
      exit ( -1 ) ;
   }

   grid2d<Field,n,y,x>  res  ( 0 ) ;


   if ( ! isConstant && ( dim == 0 ) ) {

      register int         i          ;
      register int         i_x        ;
      register int         ii         ;
      register int         j          ;
      register int         k          ;
      register Field       pkt        ;

      res.isConstant = 0                       ;
      res.arr        = array<Field> ( x * y ) ;


      for ( i = 0 ; i < y ; ++ i ) {
```

```
      // left edge

      i_x = i * x ;

      for ( j = 0 ; j < n ; ++ j ) {

         pkt = Field ( 0 ) ;

         for ( k = - n ; k <= n ; ++ k ) {

            ii = j + k ;
            if ( ii < 0 )
               ii = ii + x ;

            pkt += arr [ i_x + ii ] * weight_x [ k + n ] ;
         }

         res.arr [ i_x + j ]  = pkt ;
      }

      // middle x

      for ( j = n ; j < x - n ; ++ j ) {

         pkt = Field ( 0 ) ;

         for ( k = - n ; k <= n ; ++ k ) {

            pkt += arr [ i_x + ( j + k ) ] * weight_x [ k + n ] ;
         }

         res.arr [ i_x + j ] = pkt ;

      }

      //   right edge

      for ( j = x - n ; j < x ; ++ j ) {

         pkt = Field ( 0 ) ;

         for ( k = - n ; k <= n ; ++ k ) {

            ii = j + k ;
            if ( x <= ii )
               ii = ii - x ;

            pkt += arr [ i_x + ii ] * weight_x [ k + n ] ;
         }

         res.arr [ i_x + j ] = pkt ;

      }
   }
}


if ( ! isConstant && ( dim == 1 ) )
{

   register int         i          ;
   register int         ii         ;
   register int         j          ;
   register int         k          ;
```

```
      register Field       pkt        ;

      res.isConstant = 0                      ;
      res.arr        = array<Field> ( x * y ) ;

      for ( j = 0 ; j < x ; ++ j ) {

         // upper edge

         for ( i = 0 ; i < n ; ++ i ) {

            pkt = Field ( 0 ) ;

            for ( k = - n ; k <= n ; ++ k ) {

               ii = i + k ;
               if ( ii < 0 )
                  ii = ii + y ;

               pkt += arr [ ii * x + j ] * weight_y [ k + n ] ;

            }

            res.arr [ i * x + j ] = pkt ;
         }

         // middle y

         for ( i = n ; i < y - n ; ++ i ) {

            pkt = Field ( 0 ) ;

            for ( k = - n ; k <= n ; ++ k ) {

               pkt += arr [ ( i + k ) * x + j ] * weight_y [ k + n ] ;

            }

            res.arr [ i * x + j ] = pkt ;

         }

         // bottom edge

         for ( i = y - n ; i < y ; ++ i ) {

            pkt = Field ( 0 ) ;

            for ( k = - n ; k <= n ; ++ k ) {

               ii = i + k ;
               if ( y <= ii )
                  ii = ii - y ;

               pkt += arr [ ii * x + j ] * weight_y [ k + n ] ;

            }

            res.arr [ i * x + j ] = pkt ;
         }
      }
   }

   return res ;
}
```

```
/*-------------------------------------------------------------------------*/

template <class Field, int n, int y, int x>
const array<Field> grid2d<Field,n,y,x>::weight_x =
grid2d<Field,n,y,x>::calculateWeights ( 0 ) ;

/*-------------------------------------------------------------------------*/

template <class Field, int n, int y, int x>
const array<Field> grid2d<Field,n,y,x>::weight_y =
grid2d<Field,n,y,x>::calculateWeights ( 1 ) ;

/*-------------------------------------------------------------------------*/

template <class Field, int n, int y, int x>
array<Field> grid2d<Field,n,y,x>::calculateWeights ( const int dim )
{
   const int    max   = 2 * n + 1          ;
   array<Field> w     ( max, Field ( 0 ) ) ;
   const Field  delta ( dim == 0 ? ( Field ) 1.0 / ( Field ) ( x - 1 ) :
      ( Field ) 1.0 / ( Field ) ( y - 1 )    ) ;


   if ( n == 7 )
   {
      const Field h =  delta * 0.9975 * 2 ;

      w [  0 ] = ( Field )  3.5709981e-3/h ;    //    -7 ;
      w [  1 ] = ( Field )  0.0            ;    //    -6 ;
      w [  2 ] = ( Field ) -2.0637069e-2/h ;    //    -5 ;
      w [  3 ] = ( Field )  0.0            ;    //    -4 ;
      w [  4 ] = ( Field )  0.10411822/h   ;    //    -3 ;
      w [  5 ] = ( Field )  0.0            ;    //    -2 ;
      w [  6 ] = ( Field ) -1.2316663/h    ;    //    -1 ;
      w [  7 ] = ( Field )  0.0            ;    //     0 ;
      w [  8 ] = ( Field )  1.2316663/h    ;    //     1 ;
      w [  9 ] = ( Field )  0.0            ;    //     2 ;
      w [ 10 ] = ( Field ) -0.10411822/h   ;    //     3 ;
      w [ 11 ] = ( Field )  0.0            ;    //     4 ;
      w [ 12 ] = ( Field )  2.0637069e-2/h ;    //     5 ;
      w [ 13 ] = ( Field )  0.0            ;    //     6 ;
      w [ 14 ] = ( Field ) -3.5709981e-3/h ;    //     7 ;

   }
   else
   {
      array<Field> A     ( max * max        ) ;
      array<Field> A_inv ( max * max        ) ;
      array<Field> b     ( max, Field ( 0 ) ) ;
      Field        res                       ;
      Field        tmp                       ;
      register int i                         ;
      register int j                         ;
      register int k                         ;
      register int i_max                     ;



      b [ 1 ] = ( Field ) 1.0 ;

      for ( i = 0 ; i < max ; ++ i ) {

         i_max = i * max ;
```

```
         for ( j = 0 ; j < max ; ++ j ) {

            res = ( Field ) 1.0          ;
            tmp = delta * Field ( j - n ) ;

            for ( k = 0 ; k < i ; ++ k ) {
               res *= tmp ;
            }
            A [ i_max + j ] = res ;
         }
      }


      A_inv = matrix<Field>::invert ( max, A ) ;

      for ( i = 0 ; i < max ; ++ i ) {

         i_max = i * max ;

         for ( j = 0 ; j < max ; ++ j ) {
            w [ i ] += A_inv [ i_max + j ] * b [ j ] ;
         }
      }
   }

   return w ;
}

/*-------------------------------------------------------------------------*/

#endif
//
```

15.4 index.h

```
//

//-------------------------------------------------------------------------
//
//   Name       : index.h
//
//   Written by : Victor Madsen
//
//   Date       : 03.04.1993
//
//   Purpose    :
//
//      Implements an index as a sequence of integers, where each element of
//      an index may give a unique position in a one-dimensional array. This
//      class is suitable if one want to implement a multidimensional array,
//      and needs a way to position a component in the array. If the symbol
//      _DEBUG is defined, it is done checks on the parameters to the functions.
//
//-------------------------------------------------------------------------

#ifndef _INDEX_H
#define _INDEX_H


#include <iostream.h>
#include <stdlib.h>
#include "/tmp_mnt/Home/stud2/vmadsen/tensor/src/array.h"


/*-------------------------------------------------------------------------*/
/*                                                                         */
/*                              DECLARATION                                */
/*                                                                         */
/*-------------------------------------------------------------------------*/


class index
{
   public:

        /*-----------------------------------------------------*/
        /*              Basic operations                       */
        /*-----------------------------------------------------*/

inline   index                 (                       )       ;
inline   index                 ( const index & i       )       ;
inline   index                 ( const int , const int )       ;
inline   ~index                (                       )       ;

inline   index & operator  =   ( const index &         )       ;
inline   int     operator  ==  ( const index &         ) const ;

 /*-----------------------------------------------------*/
        /*            Characteristic operations              */
        /*-----------------------------------------------------*/

inline   int     overflow      (                       ) const ;
inline   int     pos           (                       ) const ;
inline   int     size          (                       ) const ;
inline   int     upperLimit    (                       ) const ;
         void    reset         (                       )       ;

inline   void    operator  ++  (                       )       ;
```

```
inline   index   operator  +    ( const index &         ) const ;
inline   void    set            ( const int, const int  )       ;
inline   int     read           ( const int             ) const ;

   private:

        /*-------------------------------------------------*/
        /*                Data representation              */
        /* ------------------------------------------------*/

        int  incrementRest     ( const int start       )       ;
        int  calculateRestPos  ( const int, const int  ) const ;

        array<int>  iArr       ;
        int         upperBound ;
        int         overfl     ;

} ;

/*-------------------------------------------------------------------------*/
/*                                                                         */
/*                            SPECIFICATION                                */
/*                                                                         */
/*-------------------------------------------------------------------------*/


/*-------------------------------------------------------------------------*/

inline
index::index ()
: iArr       ( 0 ) ,
  upperBound ( 0 ) ,
  overfl     ( 0 )
{}

/*-------------------------------------------------------------------------*/

inline
index::index ( const index & i )
: iArr       ( i.iArr       ) ,
  upperBound ( i.upperBound ) ,
  overfl     ( i.overfl     )
{}

/*-------------------------------------------------------------------------*/

inline
index::index ( const int numEl  , const int upper )

#ifdef _DEBUG

{
   if ( numEl < 0 || upper < 1 ) {
      cout << "Error in arg. to index::index ( const int, const int) !"
           << endl ;
      exit ( -1 ) ;
   }

  iArr       = array<int> ( numEl, 0     ) ;
  upperBound = upper                       ;
  overfl     = 0                           ;
}

#else
```

```
: iArr       ( numEl, 0     ) ,
  upperBound ( upper        ) ,
  overfl     ( 0            )
{}

#endif

/*-------------------------------------------------------------------------*/

inline
index::~index ()
{}

/*-------------------------------------------------------------------------*/

inline
index & index::operator  =  ( const index & i )
{
   iArr       = i.iArr       ;
   upperBound = i.upperBound ;
   overfl     = i.overfl     ;

   return ( *this ) ;
}

/*-------------------------------------------------------------------------*/

inline
int index::operator  ==  ( const index & i ) const
{
   int res = 0 ;

   if ( iArr       == i.iArr       &&
        upperBound == i.upperBound &&
        overfl     == i.overfl         )
      res = 1 ;

   return res ;
}

/*-------------------------------------------------------------------------*/

inline
index index::operator  + ( const index & i ) const
{
#ifdef _DEBUG
   if ( upperBound != i.upperBound ) {
      cout << "Error in arg. to index index::operator  + ( ... ) !" << endl ;
      exit ( -1 ) ;
   }
#endif

   index tmp ;

   tmp.overfl     = overfl     + i.overfl ;
   tmp.iArr       = iArr       + i.iArr   ;
   tmp.upperBound = upperBound             ;

   return tmp ;
}

/*-------------------------------------------------------------------------*/

inline
void index::operator  ++ ()
```

```
{
#ifdef _DEBUG
      if ( overfl ) {
         cout << "Error in index::operator  ++ (). Index has overflow !" ;
         cout << endl ;
         exit ( -1 ) ;
      }
#endif

   int   done  = 0               ;
   int   start = iArr.size() - 1 ;

   if (  0 <= start  )
      if ( iArr[start] + 1 < upperBound ) {
         ++ iArr [start] ;
         done = 1        ;
      }
      else {
         iArr[ start ] = 0 ;
         if ( 1 <= start )
            done = incrementRest ( -- start ) ;
      }

   if ( ! done )
      overfl = 1 ;

}

/*-------------------------------------------------------------------------*/

inline
void index::set ( const int i, const int val )
{
#ifdef _DEBUG
      if ( i < 0 || iArr.size () <= i ) {
         cout <<"Error in arg. to index::set ( ... ) !" << endl;
         exit ( -1 ) ;
      }
      if ( upperBound <= val ) {
         cout <<"Error in arg. to index::set ( ... ) !" << endl;
         exit ( -1 ) ;
      }
#endif

   iArr [i] = val ;
}

/*-------------------------------------------------------------------------*/

inline
int   index::read ( const int i ) const
{
#ifdef _DEBUG
      if ( i < 0 || iArr.size () <= i ) {
         cout <<"Error in arg. to index::read ( ... ) !" << endl;
         exit ( -1 ) ;
      }
#endif

   return iArr[i] ;
}

/*-------------------------------------------------------------------------*/

inline
```

```
int index::overflow     () const
{
   return overfl  ;
}

/*-------------------------------------------------------------------------*/

void  index::reset ()
{
   const int max = iArr.size () ;

   for ( register int i = 0 ; i < max ; ++ i )
      iArr [ i ] = 0 ;

   overfl = 0 ;
}

/*-------------------------------------------------------------------------*/

inline
int  index::pos () const
{
   const int max = iArr.size ()  ;
   int res       = max == 0 ? 0  : iArr[0] ;

   if ( 1 < max )
      res = calculateRestPos ( max, res ) ;

   return res ;
}

/*-------------------------------------------------------------------------*/

inline
int  index::size () const
{
   return iArr.size () ;
}

/*-------------------------------------------------------------------------*/

inline
int  index::upperLimit () const
{
   return upperBound ;
}

/*-------------------------------------------------------------------------*/

int  index::incrementRest ( const int start )
{
   int   done  = 0  ;

   for ( register int i = start ; ( ! done ) && 0 <= i ; -- i )
      if (iArr [ i ] + 1 < upperBound ) {
         ++ iArr [ i ]   ;
         done = 1 ;
      }
      else
         iArr [ i ] = 0 ;

   return done ;
}

/*-------------------------------------------------------------------------*/
```

```
int  index::calculateRestPos ( const int max, const int partial ) const
{
   int res = partial ;

   for ( register int i = 1 ; i < max ; ++ i ) {
      res *= upperBound ;
      res += iArr [ i ] ;
   }

   return res ;
}

/*-------------------------------------------------------------------------*/

#endif
//
```

15.5 matrix.h

```
//

//-------------------------------------------------------------------------
//
//   Name       : matrix.h
//
//   Written by : Victor Madsen
//
//   Date       : 16.02.1992
//
//   Purpose    :
//
//     This file contains some template functions that finds the inverse
//     and determinant of matrices represented by arrays.  The template
//     class argument Field must have the properties described in the
//     file field_idea.ps.  This is just a helping class for the classes
//     grid2d and tensor, and is not intended for other purposes !!!!!
//
//-------------------------------------------------------------------------

#ifndef _MATRIX_H
#define _MATRIX_H

#include "/tmp_mnt/Home/stud2/vmadsen/tensor/src/array.h"

/*-----------------------------------------------------------------------*/


template <class Field> class matrix
{
   public :

static    array<Field>   invert         ( const int            d   ,
                                          const array<Field> & arr  ) ;

static   Field           coFactor       ( const int            dim,
                                          const array<Field> & A   ,
                                          const int            i   ,
                                          const int            j    ) ;

static   array<Field>   identity        ( const int dim             ) ;

static   Field           det            ( const array<Field> & A ,
                                          const int dim             ) ;


   private :

static   array<Field> extractRowColumn  ( const int dim           ,
                                          const array<Field> & A   ,
                                          const int i              ,
                           const int j                ) ;

} ;

/*-----------------------------------------------------------------------*/

template <class Field>
array<Field> matrix<Field>::identity ( const int dim  )
{
   if ( dim <= 0 ) {
      cout << "Error in arg. to  matrix<Field>::identity ( ... ) !" << endl ;
      exit ( -1 ) ;
```

```
   }

   array<Field> tmpArr ( dim * dim, Field ( 0 ) ) ;
   const Field  one    ( 1                      ) ;

   for ( register int i = 0 ; i < dim ; ++ i )
      tmpArr [ i * dim + i ] = one ;

   return tmpArr ;
}

/*-------------------------------------------------------------------------*/

template <class Field>
Field matrix<Field>::det ( const array<Field> & A ,const int dim  )
{
   if ( dim <= 0 ) {
      cout << "Error in arg. to  matrix<Field>::det ( ... ) !" << endl ;
      exit ( -1 ) ;
   }
   if ( dim * dim != A.size () ) {
      cout << "Error in arg. to  matrix<Field>::det ( ... ) !" << endl ;
      exit ( -1 ) ;
   }

   //   Det ( A ) = Field ( 0 )
   //   For j = 0 to dim - 1 do
   //      Det ( A ) += A ( 0, j ) * koFaktor ( A, 0, j ) ;
   //   Endfor


   Field     res  ( 0 ) ;

   if ( dim == 2 ) {            //   res = A[0,0] * A[1,1] - A[0,1] * A[1,0]

      res  = A [ 0 ] ;

      res *= A [ 3 ] ;

      Field tmp ( A [ 2 ] ) ;

      tmp *= A [ 1 ] ;

      res -= tmp     ;
   }
   else

      if ( dim == 1 )

         res = A [ 0 ] ;

      else {                                              // dim > 2

         for ( register int j = 0 ; j < dim ; ++ j ) {

            res += A [ j ] * coFactor ( dim, A, 0, j ) ;

         }
      }

   return res ;
}

/*-------------------------------------------------------------------------*/
```

```
template <class Field>
array<Field> matrix<Field>::extractRowColumn ( const int dim          ,
                                               const array<Field> & A ,
                                               const int i            ,
                                 const int j                  )
{
   if ( dim <= 0 ) {
      cout << "Error in arg. to  matrix<Field>::extractRowColumn ( ... ) !" << endl ;
      exit ( -1 ) ;
   }
   if ( dim * dim != A.size () ) {
      cout << "Error in arg. to  matrix<Field>::extractRowColumn ( ... ) !" << endl ;
      exit ( -1 ) ;
   }
   if ( i < 0 || j < 0 || dim <= i || dim <= j ) {
      cout << "Error in arg. to  matrix<Field>::extractRowColumn ( ... ) !" << endl ;
      exit ( -1 ) ;
   }


   const int    max        = dim - 1     ;
   array<Field> M          ( max * max ) ;
   int          ii_offset  = 0           ;
   int          jj_offset  = 0           ;


   for ( register int ii = 0 ; ii < max ; ++ ii ) {

      if ( ii == i )
         ii_offset = 1 ;

      jj_offset = 0 ;

      for ( register int jj = 0 ; jj < max ; ++ jj ) {

         if ( jj == j )
            jj_offset = 1 ;

         M [ ii * max + jj ] = A [ ( ii + ii_offset ) * dim + jj + jj_offset  ] ;
      }
   }
   return M ;
}

/*-------------------------------------------------------------------------*/

template <class Field>
Field  matrix<Field>::coFactor ( const int           dim,
                                 const array<Field> & A   ,
                                 const int           i   ,
                                 const int           j    )
{
   if ( dim <= 0 ) {
      cout << "Error in arg. to  matrix<Field>::coFactor ( ... ) !" << endl ;
      exit ( -1 ) ;
   }
   if ( dim * dim != A.size () ) {
      cout << "Error in arg. to  matrix<Field>::coFactor ( ... ) !" << endl ;
      exit ( -1 ) ;
   }
   if ( i < 0 || j < 0 || dim <= i || dim <= j ) {
      cout << "Error in arg. to  matrix<Field>::coFactor ( ... ) !" << endl ;
      exit ( -1 ) ;
   }
```

```
   const array<Field> tmp ( matrix<Field>::extractRowColumn ( dim, A, i, j ) ) ;

   Field res = matrix<Field>::det ( tmp , dim - 1 ) ;

   if ( ( i + j ) % 2 )
      res = - res ;

   return res ;
}

/*-------------------------------------------------------------------------*/

template <class Field>
array<Field>  matrix<Field>::invert ( const int d, const array<Field> & arr )
{
   if ( d <= 0 ) {
      cout << "Error in arg. to  matrix<Field>::invert ( ... ) !" << endl ;
      exit ( -1 ) ;
   }
   if ( d * d != arr.size () ) {
      cout << "Error in arg. to  matrix<Field>::invert ( ... ) !" << endl ;
      exit ( -1 ) ;
   }

   //   X * A = I
   //
   //   X ( i, j ) = koFaktor ( A, j, i ) / Det ( A )
   //

   const Field    det_A ( matrix<Field>::det ( arr, d ) ) ;
   array<Field>   res   ( d * d                         ) ;


   if ( det_A == Field ( 0 ) )  {
      cout << "Error in invert ( const int d, const array<Field> "
           << "& arr ). The matrix can not be inverted !" << endl ;
      exit ( -1 ) ;
   }


   if ( d == 1 ) {
      res [ 0 ]  = Field ( 1 ) ;
      res [ 0 ] /= arr [ 0 ]   ;
   }
   else

      for ( register int i = 0 ; i < d ; ++ i ) {

         for ( register int j = 0 ; j < d ; ++ j ) {

            res [ i * d + j ] = matrix<Field>::coFactor ( d, arr, j, i ) ;

            res [ i * d + j ] /= det_A ;

         }
      }

   return res ;

}

/*-------------------------------------------------------------------------*/

#endif
//
```

15.6 tensor.h

```
//

//-------------------------------------------------------------------------
//
//   Name       : tensor.h
//
//   Written by : Victor Madsen
//
//   Date       : 03.04.1993
//
//   Purpose    :
//
//      Creates a class with properties equal to the mathemathical object
//      differential tensor ( over a differential ring ). This template
//      class takes an class argument diffRing, with properties as described
//      in the file diffring_idea.ps.
//
//-------------------------------------------------------------------------


#ifndef _TENSOR_H
#define _TENSOR_H



#include <iostream.h>
#include <stdlib.h>
#include "/tmp_mnt/Home/stud2/vmadsen/tensor/src/array.h"
#include "/tmp_mnt/Home/stud2/vmadsen/tensor/src/matrix.h"
#include "/tmp_mnt/Home/stud2/vmadsen/tensor/src/index.h"
#include "/tmp_mnt/Home/stud2/vmadsen/tensor/src/christof.h"



/*-------------------------------------------------------------------------*/
/*                                                                         */
/*                          DECLARATION                                    */
/*                                                                         */
/*-------------------------------------------------------------------------*/


template <class diffRing> class tensor
{
   public:
         /*-------------------------------------------------------------*/
         /*                    C++ operations                           */
         /*-------------------------------------------------------------*/

inline   tensor                          (                    )       ;
inline   tensor                          ( const tensor  &    )       ;
inline   ~tensor                         (                    )       ;
inline   int               operator ==   ( const tensor  &    ) const ;
inline   tensor  &         operator =    ( const tensor  &    )       ;


         /*-------------------------------------------------------------*/
         /*                    Module operations                        */
         /*-------------------------------------------------------------*/

         tensor            zero          (                    ) const ;
         tensor            operator   -  (                    ) const ;
         tensor            operator   +  ( const tensor     &  ) const ;
friend   tensor<diffRing>  operator   *  ( const diffRing   &,
```

```
                                  const tensor<diffRing> &  )       ;
inline   int                dimension     (                   ) const ;
inline   array<diffRing>    coordinates   (                   ) const ;
         array<tensor>      basis         (                   ) const ;

         /*-------------------------------------------------------------*/
         /*                 pure tensor operations                      */
         /*-------------------------------------------------------------*/

inline   tensor                           ( const diffRing &    )       ;
inline   operator           diffRing      (                     ) const ;
inline   int                numCoModules  (                     ) const ;
inline   int                numModules    (                     ) const ;
         tensor             contract      (                     ) const ;
         tensor             operator  *   ( const tensor  &     ) const ;
         tensor             operator  ()  ( const tensor  &     ) const ;
inline   static tensor      kronecker     (                     )       ;

         /*-------------------------------------------------------------*/
         /*             Differentiation operators                       */
         /*-------------------------------------------------------------*/

 tensor            lieDiff       ( const tensor  &     ) const ;
 tensor            partialDiff   ( const int           ,
                                   const christoffel<diffRing> & ) const ;
 tensor            covDiff       (
                                   const christoffel<diffRing> & ) const ;

         /*-------------------------------------------------------------*/
         /*                   Composed functions                        */
         /*-------------------------------------------------------------*/

inline   int         isRing                (                     ) const ;
inline   int         isModule              (                     ) const ;
inline   int         isCoModule            (                     ) const ;
inline   int         equalType             ( const tensor     &  ) const ;
inline   int         canContract           (                     ) const ;
         tensor      operator   -          ( const tensor     &  ) const ;
         tensor &    operator  -=          ( const tensor     &  )       ;
         tensor &    operator  +=          ( const tensor     &  )       ;
         tensor &    operator  *=          ( const diffRing   &  )       ;
         tensor      gradient              (                     ) const ;
         tensor      divergence (const christoffel<diffRing> &  ) const ;

         /*-------------------------------------------------------------*/
         /*                 Extra helping functions                     */
         /*-------------------------------------------------------------*/

inline   tensor                     ( const int, const int      )       ;
inline   tensor                     ( const int, const int,
                                      const array<diffRing>  &  )       ;
inline   tensor      swapUpperIndex ( const int , const int     ) const ;
inline   tensor      swapLowerIndex ( const int , const int     ) const ;
static   tensor      identityTensor ( const int , const int     )       ;

         /*-------------------------------------------------------------*/
         /*            These functions requires that diffRing           */
         /*            is also a field class !!!!                       */
         /*-------------------------------------------------------------*/

inline   int                    canInvert         (           ) const ;
         int                    canMakeChristoffel (          ) const ;
inline   tensor                 invert            (           ) const ;
         christoffel<diffRing>  makeChristoffel   (           ) const ;
```

```
      /*----------------------------------------------------------------*/
      /*                 Implementation details                         */
      /*----------------------------------------------------------------*/

   private:

      static int      i_pow ( const int p , const int s )       ;
      tensor          swap  ( const int , const int     ) const ;

      array<diffRing> arr   ;
      int             r     ,
                      s     ,
                      d     ;
} ;


/*-------------------------------------------------------------------------*/
/*                                                                         */
/*                          IMPLEMENTATION                                 */
/*                                                                         */
/*-------------------------------------------------------------------------*/

/*-------------------------------------------------------------------------*/

template <class diffRing> inline
tensor<diffRing>::tensor  (  )
: d   ( diffRing::numDimensions () ),
  r   ( 0                          ),
  s   ( 0                          ),
  arr ( 1, diffRing ( 0 )          )
{}

/*-------------------------------------------------------------------------*/

template <class diffRing> inline
tensor<diffRing>::tensor ( const diffRing & el )
: d   ( diffRing::numDimensions () ),
  r   ( 0                          ),
  s   ( 0                          ),
  arr ( 1, el                      )
{}

/*-------------------------------------------------------------------------*/

template <class diffRing> inline
tensor<diffRing>::tensor  ( const tensor<diffRing> &  v )
: d   ( v.d   ),
  r   ( v.r   ),
  s   ( v.s   ),
  arr ( v.arr )
{}

/*-------------------------------------------------------------------------*/

template < class diffRing > inline
tensor<diffRing>::tensor ( const int numCoModules ,
                           const int numModules     )
: d   ( diffRing::numDimensions ()           ),
  r   ( numCoModules                         ),
  s   ( numModules                           ),
  arr ( i_pow ( diffRing::numDimensions (),
                numCoModules + numModules  ),
                diffRing ( 0 )               )
{
```

```
   if ( numCoModules < 0 || numModules < 0 ) {
      cout << "Error in arg. to tensor<diffRing>::tensor ( const int,"
           << " const int ) !"
           << endl;
      exit ( -1 ) ;
   }
}

/*--------------------------------------------------------------------------*/

template < class diffRing > inline
tensor<diffRing>::tensor (  const   int               numCoModules  ,
    const   int               numModules    ,
    const   array<diffRing> & mArr           )
: d   ( diffRing::numDimensions () ),
  r   ( numCoModules               ),
  s   ( numModules                 ),
  arr ( mArr                       )
{
   if ( numCoModules < 0 || numModules < 0 ||
        mArr.size () != i_pow ( diffRing::numDimensions (),
                                numCoModules + numModules   ) ) {
      cout << "Error in arg. to tensor<diffRing>::tensor ( const int,"
           << " const int, const   array<diffRing> & ) !"
           << endl;
      exit ( -1 ) ;
   }

}

/*--------------------------------------------------------------------------*/

template <class diffRing> inline
tensor<diffRing>::~tensor ()
{}

/*--------------------------------------------------------------------------*/

template <class diffRing>  inline
int   tensor<diffRing>::operator  == ( const tensor<diffRing> & v ) const
{
   int res = 0 ;

   if ( equalType ( v ) && arr == v.arr )
      res = 1 ;

   return res ;
}

/*--------------------------------------------------------------------------*/

template <class diffRing>  inline
tensor<diffRing> &  tensor<diffRing>::operator  =
   ( const tensor<diffRing> & v )
{
   d   = v.d   ;
   r   = v.r   ;
   s   = v.s   ;
   arr = v.arr ;

   return ( *this ) ;
}

/*--------------------------------------------------------------------------*/
```

```
template < class diffRing > inline
tensor<diffRing>::operator diffRing    ( ) const
{
   if ( ! isRing () ) {
      cout << "Error in tensor<diffRing>::operator diffRing ! The tensor has "
           << "wrong type !" << endl ;
      exit ( -1 ) ;
   }

   return arr [ 0 ] ;
}

/*-------------------------------------------------------------------------*/

template <class diffRing>  inline
int tensor<diffRing>::equalType (const tensor<diffRing> & v ) const
{
   int res = 0 ;

   if ( d == v.d && r == v.r && s == v.s )
      res = 1 ;

   return res ;
}

/*-------------------------------------------------------------------------*/

template <class diffRing>  inline
int tensor<diffRing>::isRing (   ) const
{
   return ( r + s == 0 ? 1 : 0 ) ;
}

/*-------------------------------------------------------------------------*/

template <class diffRing>  inline
int tensor<diffRing>::isModule ( ) const
{
   int res = 0 ;

   if (  r == 1 && s == 0 )
      res = 1 ;

   return res ;
}

/*-------------------------------------------------------------------------*/

template <class diffRing>  inline
int tensor<diffRing>::isCoModule ( ) const
{
   int res = 0 ;

   if (  r == 0 && s == 1 )
      res = 1 ;

   return res ;
}

/*-------------------------------------------------------------------------*/

template <class diffRing>  inline
int tensor<diffRing>::canContract ( ) const
{
   int res = 0 ;
```

```
   if ( 0 < r && 0 < s )
      res = 1 ;

   return res ;
}

/*--------------------------------------------------------------------------*/

template <class diffRing>  inline
int tensor<diffRing>::numCoModules ( ) const
{
   return ( r ) ;
}

/*--------------------------------------------------------------------------*/


template <class diffRing>  inline
int tensor<diffRing>::numModules ( ) const
{
   return ( s ) ;
}

/*--------------------------------------------------------------------------*/

template <class diffRing>
tensor<diffRing>  tensor<diffRing>::zero ( ) const
{
   return tensor<diffRing> ( r, s ) ;
}

/*--------------------------------------------------------------------------*/

template <class diffRing>
tensor<diffRing>  tensor<diffRing>::operator  -  () const
{
   tensor<diffRing>  res ( * this      ) ;
   const int         max = arr.size ()   ;

   for ( register int i = 0 ; i < max ; ++ i )
      res.arr[ i ] = - arr [ i ] ;

   return res ;
}

/*--------------------------------------------------------------------------*/

template <class diffRing>
tensor<diffRing>  tensor<diffRing>::operator  +
     ( const tensor<diffRing> & v ) const
{
   if ( ! equalType ( v ) ) {
      cout << "Error, binary operation on not compatible tensors !" << endl ;
      exit ( -1 ) ;
   }

   tensor<diffRing> res ( * this      ) ;
   const int        max = arr.size ()   ;

   for ( register int i = 0 ; i < max ; ++ i )
      res.arr[ i ] += v.arr [ i ] ;

   return res ;
}
```

```
/*-------------------------------------------------------------------------*/

template <class diffRing>
tensor<diffRing> operator * ( const diffRing & f, const tensor<diffRing> & v )
{
   tensor<diffRing> res ( v             ) ;
   const int        max = v.arr.size ()   ;

   for ( register int i = 0 ; i < max ; ++ i )
      res.arr[ i ] *= f ;

   return res ;
}

/*-------------------------------------------------------------------------*/

template <class diffRing>
tensor<diffRing>  tensor<diffRing>::operator  -
     ( const tensor<diffRing> & v ) const
{
   if ( ! equalType ( v ) ) {
      cout << "Error, binary operation on not compatible tensors !" << endl ;
      exit ( -1 ) ;
   }

   tensor<diffRing> res ( * this      ) ;
   const int        max = arr.size ()   ;

   for ( register int i = 0 ; i < max ; ++ i )
      res.arr[ i ] -= v.arr [ i ] ;

   return res ;
}

/*-------------------------------------------------------------------------*/

template <class diffRing>
tensor<diffRing> & tensor<diffRing>::operator += ( const tensor<diffRing> & v )
{
   if ( ! equalType ( v ) ) {
      cout << "Error, binary operation on not compatible tensors !" << endl ;
      exit ( -1 ) ;
   }

   const int max = arr.size ()   ;

   for ( register int i = 0 ; i < max ; ++ i )
      arr[ i ] += v.arr [ i ] ;

   return *this ;
}

/*-------------------------------------------------------------------------*/

template <class diffRing>
tensor<diffRing> & tensor<diffRing>::operator -= ( const tensor<diffRing> & v )
{
   if ( ! equalType ( v ) ) {
      cout << "Error, binary operation on not compatible tensors !" << endl ;
      exit ( -1 ) ;
   }

   const int max = arr.size ()   ;
```

```
   for ( register int i = 0 ; i < max ; ++ i )
      arr[ i ] -= v.arr [ i ] ;

   return *this ;
}

/*--------------------------------------------------------------------------*/

template <class diffRing>
tensor<diffRing> &  tensor<diffRing>::operator  *= ( const diffRing & f )
{
   const int         max = arr.size ()   ;

   for ( register int i = 0 ; i < max ; ++ i )
      arr[ i ] *= f ;

   return *this ;
}

/*--------------------------------------------------------------------------*/

template <class diffRing>  inline
int  tensor<diffRing>::dimension () const
{
   return arr.size () ;
}

/*--------------------------------------------------------------------------*/

template <class diffRing>  inline
array<diffRing>   tensor<diffRing>::coordinates () const
{
   return arr ;
}

/*--------------------------------------------------------------------------*/

template <class diffRing>
array<tensor<diffRing> >  tensor<diffRing>::basis () const
{
   const int              max   = arr.size () ;
   const diffRing         one   ( 1         ) ;

   array<tensor<diffRing > > res ( max, zero () ) ;

   for ( int i = 0 ; i < max ; ++ i ) {
      ( res [ i ] ).arr [ i ] = one ;
   }

   return res ;
}

/*--------------------------------------------------------------------------*/

template < class diffRing >
tensor<diffRing> tensor<diffRing>::kronecker (  )
{
   const int dim = diffRing::numDimensions () ;
   return tensor<diffRing> ( 1, 1, matrix<diffRing>::identity ( dim ) ) ;
}

/*--------------------------------------------------------------------------*/

template <class diffRing>
tensor<diffRing> tensor<diffRing>::operator  *
```

```
    ( const tensor<diffRing>  & t ) const
{
   // A, B, C, D are indexes
   // Loop over A, B, C, D and calculate :
   //    resTensor ( A + C, B + D ) = T1 ( A, B ) * T2 ( C, D ) ;

   register int     i                                    ;
   int              resPos                               ;
   int              arg1pos                              ;
   int              arg2pos                              ;
   index            A          ( r                , d ) ;
   index            B          ( s                , d ) ;
   index            C          ( t.r              , d ) ;
   index            D          ( t.s              , d ) ;
   index            resIndx    ( r + t.r + s + t.s, d ) ;
   index            arg1Indx   ( r + s            , d ) ;
   index            arg2Indx   ( t.r + t.s        , d ) ;
   tensor<diffRing> res        ( r + t.r, s + t.s     ) ;


   while ( ! A.overflow () ) {

      B.reset () ;
      for ( i = 0 ; i < r ; ++ i ) {
         arg1Indx.set ( i, A.read ( i ) ) ;
         resIndx.set  ( i, A.read ( i ) ) ;
      }

      while ( ! B.overflow () ) {

         C.reset () ;
         for ( i = 0 ; i < s ; ++ i ) {
            resIndx.set  ( i + r + t.r, B.read ( i ) ) ;
            arg1Indx.set ( i + r, B.read ( i ) ) ;
         }

         arg1pos = arg1Indx.pos () ;

         while ( ! C.overflow () ) {

            D.reset () ;
            for ( i = 0 ; i < t.r ; ++ i ) {
               resIndx.set ( i + r, C.read ( i ) ) ;
               arg2Indx.set ( i, C.read ( i ) ) ;
            }

            while ( ! D.overflow () ) {

               for ( i = 0 ; i < t.s ; ++ i ) {
                  resIndx.set  ( i + r + t.r + s, D.read ( i ) ) ;
                  arg2Indx.set ( i + t.r, D.read ( i ) ) ;
               }

               arg2pos = arg2Indx.pos () ;
               resPos  = resIndx.pos  () ;

               res.arr [ resPos ]  = arr   [ arg1pos ] ;
               res.arr [ resPos ] *= t.arr [ arg2pos ] ;

               ++ D ;
            }
            ++ C ;
         }
         ++ B ;
      }
```

```
      ++ A ;
   }

   return res ;
}

/*-------------------------------------------------------------------------*/

template <class diffRing>
tensor<diffRing> tensor<diffRing>::contract () const
{
   // A, B, C are indexes, C is an index with only one position.
   // Loop over A, B, C and calculate :
   //    resTensor ( A , B  ) = T ( A + C, B + C ) ;


   if ( ! canContract () ) {
      cout << "Error in tensor<diffRing> tensor<diffRing>::contract ()," ;
      cout << " There are no indexes to contract !" << endl ;
      exit ( -1 ) ;
   }


   int             resPos                    ;
   register int    i                         ;
   register int    val1                      ;
   index           A       ( r - 1     , d  ) ;
   index           B       ( s - 1     , d  ) ;
   index           C       ( 1         , d  ) ;
   index           argIndx ( r + s     , d  ) ;
   index           resIndx ( r + s - 2 , d  ) ;
   tensor<diffRing> res    ( r - 1,s - 1    ) ;


   while ( ! A.overflow () ) {

      B.reset () ;

      val1 = r - 1 ;
      for ( i = 0 ; i < val1 ; ++ i ) {
         resIndx.set ( i, A.read ( i ) ) ;
         argIndx.set ( i, A.read ( i ) ) ;
      }

      while ( ! B.overflow () ) {

         C.reset ()               ;

         val1 = s - 1 ;
         for ( i = 0 ; i < val1 ; ++ i ) {
            resIndx.set ( i + r - 1, B.read ( i ) ) ;
            argIndx.set ( i + r    , B.read ( i ) ) ;
         }
         resPos = resIndx.pos () ;

         while ( ! C.overflow () ) {

            argIndx.set ( r - 1    , C.read ( 0 ) ) ;
            argIndx.set ( r + s - 1, C.read ( 0 ) ) ;

            res.arr  [ resPos ] += arr [ argIndx.pos () ] ;

            ++ C ;
         }
         ++ B ;
```

```
      }
      ++ A ;
   }

   return res ;
}

/*-------------------------------------------------------------------------*/

template <class diffRing>
tensor<diffRing>   tensor<diffRing>::operator ()
     ( const tensor<diffRing> & t ) const
{

   //  For all M and N ,loop over  A and B and calculate :
   //      res ( M , N ) = T1 ( M + A, N + B ) * T2 ( B, A )


   if ( r < t.s || s < t.r ) {
      cout << "Error in tensor<diffRing> tensor<diffRing>::operator() ( ... )," ;
      cout << " The argument has wrong type !" << endl ;
      exit ( -1 ) ;
   }

   register int     i                                ;
   register int     val1                             ;
   register int     val2                             ;
   int              resPos                           ;
   index            M        ( r - t.s          , d ) ;
   index            N        ( s - t.r          , d ) ;
   index            A        ( t.s              , d ) ;
   index            B        ( t.r              , d ) ;
   index            resIndx  ( r - t.s + s - t.r, d ) ;
   index            arg1Indx ( r + s            , d ) ;
   index            arg2Indx ( t.r + t.s        , d ) ;
   tensor<diffRing> res      ( r -t.s, s - t.r      ) ;


   while ( ! M.overflow () ) {

      N.reset () ;
      val1 = r - t.s ;
      for ( i = 0 ; i < val1 ; ++ i ) {
         resIndx.set  ( i, M.read ( i ) ) ;
         arg1Indx.set ( i, M.read ( i ) ) ;
      }

      while ( ! N.overflow () ) {

         A.reset () ;
         val1 = s - t.r ;
         val2 = r - t.s ;
         for ( i = 0 ; i < val1 ; ++ i ) {
            resIndx.set  ( i + val2, N.read ( i ) ) ;
            arg1Indx.set ( i + r   , N.read ( i ) ) ;
         }
         resPos = resIndx.pos () ;

         while ( ! A.overflow () ) {

            B.reset () ;
            val1 = r - t.s ;
            for ( i = 0 ; i < t.s ; ++ i ) {
               arg1Indx.set ( i + val1, A.read ( i ) ) ;
               arg2Indx.set ( i + t.r , A.read ( i ) ) ;
```

```
            }

            while ( ! B.overflow () ) {

               val1 = r + s - t.r ;
               for ( i = 0 ; i < t.r ; ++ i ) {
                  arg1Indx.set ( i + val1, B.read ( i ) ) ;
                  arg2Indx.set ( i       , B.read ( i ) ) ;
               }

               res.arr[ resPos ] +=  arr   [ arg1Indx.pos () ] *
                                    t.arr [ arg2Indx.pos () ] ;

               ++ B ;
            }
            ++ A ;
         }
         ++ N ;
      }
      ++ M ;
   }

   return res ;
}

/*-------------------------------------------------------------------------*/

template < class diffRing > inline
tensor<diffRing> tensor<diffRing>::swapUpperIndex
                                      ( const int i1 , const int i2 ) const
{
   if ( i1 == i2 || i1 < 0  || i2 < 0 || r <= i1  || r <= i2 ) {
      cout << "Error in arg. to tensor<diffRing>::swapUpperIndex !" <<endl;
      exit ( -1 ) ;
   }

   return swap ( i1, i2 ) ;
}

/*-------------------------------------------------------------------------*/

template < class diffRing > inline
tensor<diffRing> tensor<diffRing>::swapLowerIndex
                                    ( const int i1 , const int i2  ) const
{
   if ( i1 == i2 || i1 < 0 || i2 < 0 || s <= i1  || s < i2 ) {
      cout << "Error in arg. to tensor<diffRing>::swapLowerIndex !" <<endl;
      exit ( -1 ) ;
   }

   return swap ( i1 + r, i2 + r ) ;
}

/*-------------------------------------------------------------------------*/

template < class diffRing >
tensor<diffRing> tensor<diffRing>::swap ( const int i1 , const int i2  ) const
{
   int i ;
   const int        iMax   = i1 < i2 ? i2 : i1 ;
   const int        iMin   = i1 < i2 ? i1 : i2 ;
   index            resI   ( r + s , d       ) ;
   index            tmpI   ( r + s , d       ) ;
   array<diffRing>  resArr ( arr.size ()     ) ;
```

```
   while ( ! resI.overflow () ) {

      tmpI     = resI                        ;
      i        = tmpI.read ( iMin )          ;
      tmpI.set ( iMin, tmpI.read ( iMax ) ) ;
      tmpI.set ( iMax, i                   ) ;

      resArr [ resI.pos () ] = arr [ tmpI.pos () ] ;

      ++ resI ;
   }

   return tensor<diffRing> ( r, s, resArr ) ;
}

/*--------------------------------------------------------------------------*/

template <class diffRing>
tensor<diffRing> tensor<diffRing>::lieDiff ( const tensor<diffRing>& v ) const
{

   //   res [mn..;pq..] = v[i]*arg[mn..;pq..].partialDiff(i)
   //                     - arg[in..;pq..] * v[m].partialDiff(i)
   //                     - arg[mi..;pq..] * v[n].partialDiff(i)
   //                     ...
   //                     + arg[mn..;iq..] * v[i].partialDiff(p)
   //                     + arg[mn..;pi..] * v[i].partialDiff(q)
   //                     ...


   if ( ! v.isModule () ) {
      cout << "Error in arg. to tensor<diffRing>::lieDiff ( const tensor ...  ) !"
           << endl ;
      exit( -1 ) ;
   }


   index                 resI    ( r + s, d                      ) ;
   tensor<diffRing>      res     ( r, s, array<diffRing>
                                   ( arr.size () , diffRing ( 0 ))) ;


   while ( ! resI.overflow () ) {

      for ( register int i = 0 ; i < d ; ++ i ) {

         res.arr [ resI.pos () ] += v.arr [ i ] *
                                    arr [ resI.pos () ].partialDiff ( i ) ;
      }
      ++ resI ;
   }


   if ( 0 < r + s ) {

      array<diffRing> diffVec    ( d * d , diffRing ( 0 )  ) ;
      int             pos                                    ;
      int             argRank    = r + s                     ;
      index           argI       ( resI                   ) ;

      for ( register int i = 0 ; i < d ; ++ i )
      {
         const int i_d = i * d ;

         for ( register int k = 0 ; k < d ; ++ k )
```

```
            diffVec [ i_d + k ] = v.arr [ i ].partialDiff ( k ) ;
      }


      for ( register int n = 0 ; n < r ; ++ n ) {

         resI.reset () ;

         while ( ! resI.overflow () ) {

            for ( register int k = 0 ; k < argRank ; ++ k )
               argI.set ( k, resI.read ( k ) ) ;

            pos   = resI.read ( n ) * d ;

            for ( register int i = 0 ; i < d ; ++ i ) {

               argI.set ( n, i ) ;

               res.arr [ resI.pos () ] -= arr     [ argI.pos() ] *
                                          diffVec [ pos + i    ] ;
            }
            ++ resI ;
         }
      }


      for ( register int m = 0 ; m < s ; ++ m ) {

         resI.reset () ;

         while ( ! resI.overflow () ) {

            for ( register int k = 0 ; k < argRank ; ++ k )
               argI.set ( k, resI.read ( k ) ) ;

            pos =  resI.read ( r + m ) ;

            for ( register int i = 0 ; i < d ; ++ i ) {

               argI.set ( r + m, i ) ;

               res.arr [ resI.pos () ] += arr     [ argI.pos()  ] *
                                          diffVec [ i * d + pos ] ;
            }
            ++ resI ;
         }
      }
   }

   return res ;
}

/*--------------------------------------------------------------------------*/

template < class diffRing >
tensor<diffRing>  tensor<diffRing>::partialDiff
                     ( const int dim, const christoffel<diffRing> & c ) const
{
   //
   //   res [ pq..ij..] = A [pq..ij..].partialDiff ( k ) - c[mki]*A[pq..mj..]
   //                                                    - c[mkj]*A[pq..im..]
   //                                                     ...
   //                                                    + c[pkm]*A[mq..ij..]
   //                                                    + c[qkm]*A[pm..ij..]
```

```
   //                                                    ...

   if ( dim < 0 || d <= dim ) {
      cout << "Error in arg. to tensor<diffRing>::partialDiff ( const tensor ...  ) !"
           << endl ;
      exit( -1 ) ;
   }

   index             resI    ( r + s    , d ) ;
   index             argI    ( r + s    , d ) ;
   const int         argRank = r + s                ;
   const int         max     = arr.size ()          ;
   array<diffRing>   res     ( max , diffRing ( 0 ) ) ;


   for ( register int i = 0 ; i < max ; ++ i ) {
      res [ i ] = (arr [ i ]).partialDiff ( dim ) ;
   }

   for ( i = 0 ; i < s ; ++ i ) {

      resI.reset () ;

      while ( ! resI.overflow () ) {

         argI     = resI         ;

         for ( register int m = 0 ; m < d ; ++ m ) {

            argI.set ( r + i, m ) ;

            res  [ resI.pos () ] -= c   (m, dim, resI.read(r + i) ) *
                                    arr [ argI.pos ()             ] ;
         }
         ++ resI ;
      }
   }


   for ( i = 0 ; i < r ; ++ i ) {

      resI.reset () ;

      while ( ! resI.overflow () ) {

         argI = resI ;

         for ( register int m = 0 ; m < d ; ++ m ) {

            argI.set ( i, m ) ;

            res  [ resI.pos () ] +=  c   ( resI.read(i), dim , m ) *
                                     arr [ argI.pos ()           ] ;
         }
         ++ resI ;
      }
   }

   return tensor<diffRing> (  r , s , res ) ;
}


/*-------------------------------------------------------------------------*/

template < class diffRing >
```

```
tensor<diffRing>  tensor<diffRing>::covDiff
   ( const christoffel<diffRing> & c ) const
{
   //
   //   res [ pq..ij..;k] = arg[pq..ij..].partialDiff ( k ) - c[mki]*A[pq..mj..]
   //                                                       - c[mkj]*A[pq..im..]
   //                                                       ...
   //                                                       + c[pkm]*A[mq..ij..]
   //                                                       + c[qkm]*A[pm..ij..]
   //                                                       ...


   index             resI    ( r + s + 1, d ) ;
   index             argI    ( r + s    , d ) ;
   const int         argRank = r + s                      ;
   const int         max     = arr.size ()                ;
   array<diffRing>   res     ( max * d, diffRing ( 0 )      ) ;

   {
      register int i_d     = 0 ;

      for ( register int i = 0 ; i < max ; ++ i ) {

         for( register int k = 0 ; k < d ; ++ k ) {
            res [ i_d + k ] = (arr [ i ]).partialDiff ( k ) ;
         }

         i_d += d ;
      }
   }



   for ( register int i = 0 ; i < s ; ++ i ) {

      resI.reset () ;

      while ( ! resI.overflow () ) {

         for ( register int k = 0 ; k < argRank ; ++ k )
            argI.set ( k, resI.read ( k ) ) ;

         for ( register int m = 0 ; m < d ; ++ m ) {

            argI.set ( r + i, m ) ;

            res  [ resI.pos () ] -=
                     c   ( m , resI.read(argRank-1),resI.read(r + i) ) *
                     arr [ argI.pos ()                                ] ;
         }
         ++ resI ;
      }
   }


   for ( register int j = 0 ; j < r ; ++ j ) {

      resI.reset () ;

      while ( ! resI.overflow () ) {

         for ( register int k = 0 ; k < argRank ; ++ k )
            argI.set ( k, resI.read ( k ) ) ;

         for ( register int m = 0 ; m < d ; ++ m ) {
```

```
            argI.set ( j, m ) ;

            res  [ resI.pos () ] +=
                        c   ( resI.read(j), resI.read(argRank-1), m ) *
                        arr [ argI.pos ()                          ] ;
         }
         ++ resI ;
      }
   }

   return tensor<diffRing> (  r , s + 1 , res ) ;
}

/*-------------------------------------------------------------------------*/

template < class diffRing >
tensor<diffRing>  tensor<diffRing>::identityTensor ( const int r, const int s )
{
   if ( r < 0 || s < 0 ) {
      cout << "Error in arg. to tensor<diffRing>::identityTensor ( ... ) !"
           << endl ;
      exit( -1 ) ;
   }

   tensor<diffRing> res       ( r, s         ) ;
   index            resI      ( r + s, res.d ) ;
   const diffRing   one       ( 1            ) ;
   int              allEqual                  ;
   int              indx                      ;


   if ( r + s == 0 )
         res.arr [ 0 ] = one ;
   else

      while ( ! resI.overflow () ) {

         allEqual = 1               ;
         indx     = resI.read ( 0 ) ;

         for ( register int i = 1 ; i < ( r + s ) && allEqual ; ++ i ) {
            if ( indx != resI.read ( i ) )
               allEqual = 0 ;
         }

         if ( allEqual )
            res.arr [ resI.pos () ] = one ;

         ++ resI ;
      }

   return res ;
}

/*-------------------------------------------------------------------------*/

template <class diffRing>
int tensor<diffRing>::i_pow ( const int p , const int s )
{
   int res = 1 ;
   for ( register int i = 0 ; i < s ; ++i )
       res = res * p ;
   return res ;
}
```

```
/*--------------------------------------------------------------------------*/

template <class diffRing>
int  tensor<diffRing>::canInvert ( ) const
{
   int res = 1 ;

   if ( r + s  != 2 )
      res = 0 ;

   if ( res == 1 )
      if ( diffRing ( 0 ) ==
           matrix<diffRing>::det ( arr, i_pow ( d, ( r + s ) / 2 ) ) )
         res = 0 ;

   return res ;
}

/*--------------------------------------------------------------------------*/

template <class diffRing>
tensor<diffRing>  tensor<diffRing>::invert ( ) const
{
   if ( ! canInvert () ) {
      cout << "Error in arg. to tensor<diffRing>::invert ( ... ) !"
           << " The tensor can not be inverted !" << endl ;
      exit( -1 ) ;
   }

   return tensor<diffRing>
        ( s, r, matrix<diffRing>::invert ( i_pow(d,( r + s )/2) ,arr ) )  ;
}

/*--------------------------------------------------------------------------*/

template <class diffRing>
int  tensor<diffRing>::canMakeChristoffel ( ) const
{
   int res = 1 ;

   if ( numCoModules () != 0 || numModules () != 2 || ! canInvert () )
      res = 0 ;

   return res ;
}

/*--------------------------------------------------------------------------*/

template <class diffRing>
christoffel<diffRing> tensor<diffRing>::makeChristoffel () const
{
   //   christ [ n, j, i ] = ( 1 / 2 ) * g_inv [ n, k ] *
   //                        ( g [ k, i ].partialDiff ( j ) +
   //                          g [ j, k ].partialDiff ( i ) -
   //                          g [ i, j ].partialDiff ( k )    ) ;

   if ( ! canMakeChristoffel () ) {
      cout << "Error in arg. to tensor<diffRing>::makeChristoffel ( ... ) !"
           << " The tensor can not make Christoffel !" << endl ;
      exit( -1 ) ;
   }

        array<diffRing>   res      ( d * d * d, diffRing ( 0 )       ) ;
   const tensor<diffRing>  g_inv    ( ( * this ).invert ()            ) ;
```

```
        array<diffRing>   gDiff    ( d * d * d                        ) ;
   const diffRing         halv     ( diffRing ( 1 ) / diffRing ( 2 ) ) ;
        diffRing          resPart  ( diffRing ( 0 )                   ) ;


   for ( register int i = 0 ; i < d ; ++ i ) {

      for ( register int j = 0 ; j < d ; ++ j ) {

         for ( register int k = 0 ; k < d ; ++ k ) {

            gDiff [ ( i * d + j ) * d + k ] = arr [ i * d + j ].partialDiff ( k ) ;

         }
      }
   }

   for ( register int n = 0 ; n < d ; ++ n ) {

      for ( register int i = 0 ; i < d ; ++ i ) {

         for ( register int j = 0 ; i < d ; ++ i ) {

            for ( register int k = 0 ; k < d ; ++ k ) {

               resPart           = gDiff [ ( k * d + i ) * d + j ] ;
               resPart          += gDiff [ ( j * d + k ) * d + i ] ;
               resPart          -= gDiff [ ( i * d + j ) * d + k ] ;
               resPart          *= g_inv.arr [ n * d + k ]         ;
               res [ ( n * d + j) * d + i ]  += resPart ;

            }

            res [ ( n * d + j ) * d + i ] *= halv ;
         }
      }
   }

   return christoffel<diffRing> ( res ) ;
}

/*--------------------------------------------------------------------------*/

template < class diffRing >
tensor<diffRing>  tensor<diffRing>::gradient  ( ) const
{
   if ( ( r + s ) != 0 ) {
      cout << "Error in arg. to tensor<diffRing>::gradient  ( ) const !"
           << endl ;
      exit ( -1 ) ;
   }

   tensor<diffRing> res ( 1, 0 ) ;

   for ( register int i = 0 ; i < d ; ++ i )
      res.arr [ i ] = arr [ 0 ].partialDiff ( i ) ;

   return res ;
}

/*--------------------------------------------------------------------------*/

template < class diffRing >
tensor<diffRing>  tensor<diffRing>::divergence
                                  ( const christoffel<diffRing> & c ) const
```

```
{
   tensor<diffRing>   res     ( r - 1, s     ) ;
   index              resIndx ( r + s - 1, d ) ;
   index              argIndx ( r + s    , d ) ;
   const int          r_1     = r - 1          ;
   register int       j                        ;

   for ( register int i = 0 ; i < d ; ++ i ) {

      const tensor<diffRing> arg ( partialDiff ( i , c ) ) ;

      resIndx.reset () ;

      while ( ! resIndx.overflow () ) {

         for ( j = 0 ; j < r_1 ; ++ j )
            argIndx.set ( j, resIndx.read ( j ) ) ;

         argIndx.set ( r_1, i ) ;

         for ( j = 0 ; j < s ; ++ j )
            argIndx.set ( j + r, resIndx.read ( j + r_1 ) ) ;

         res.arr [ resIndx.pos () ] += arg.arr [ argIndx.pos () ] ;

         ++ resIndx ;
      }
   }

   return res ;
}

/*----------------------------------------------------------------------------*/


#endif
//
```