Einstein summation for multi-dimensional arrays *

Krister Åhlander

Abstract

One of the most common data abstractions, at least in scientific computing, is the multi-dimensional array. A numerical algorithm may sometimes conveniently be expressed as a generalized matrix multiplication, which computes a multi-dimensional array from two other multi-dimensional arrays. By adopting index notation with the Einstein summation convention, an elegant tool for expressing generalized matrix multiplications is obtained. Index notation is the succinct and compact notation primarily used in tensor calculus.

In this paper, we develop computer support for index notation as a domain specific language. Grammar and semantics are proposed, yielding an unambiguous interpretation algorithm. An object-oriented implementation of a C++ library that supports index notation is described.

A key advantage with computer support of index notation is that the notational gap between a mathematical index notation algorithm and its implementation in a computer language is avoided. This facilitates program construction as well as program understanding. Program examples that demonstrate the close resemblance between code and the original mathematical formulation are presented.

Key words: Index notation, mathematical software, domain specific language, tensor calculus

1 Introduction

Multi-dimensional arrays are used extensively in scientific computing. In software that simulates partial differential equations, multi-dimensional arrays are used for example for the following tasks:

1. Representing the computational domain as a discrete grid. In much the same way as each point in space can be represented by its continuous coordinates, each discretization point in a structured grid can be indexed with one, two, or three indices, depending on the number of space dimensions. Thus, a proper data structure for a structured grid is an array with up to three indices.

*Research funded via a grant from the Norwegian research council. Email: krister@ii.uib.no

2. Representing components of physical quantities. A velocity vector, for instance, can be represented as a one-dimensional array, with one, two or three components, depending on the spatial dimension. Physical quantities which require more than one index also occur frequently in applications, for instance the stress tensor in elasticity theory. These quantities may also vary over the computational domain, which requires the data structures to be arrays of arrays, one array referring to the physical quantity and one array referring to the discretization. See for instance [1] for a discussion of these abstractions.

3. Expressing linear systems of equations as matrix vector multiplications, usually denoted $Ax = b$. A is here a matrix, representable as an array with two indices, and the unknown x and the right hand side b are vectors, which can be represented as one-dimensional arrays.

Since arrays are so frequently used in scientific computing, software support for arrays is very well developed in programming languages for scientific computing, for instance in Fortran 90 and subsequent versions. Also in C++, several array libraries exist, for instance A++/P++ [2], Pooma [3], and Blitz++ [4]. These libraries allows the programmer, in the same way as Fortran 90 does, to refer to whole (sub)arrays and not necessarily to individual components. The first advantage is that the notation becomes more compact, since one no longer has to loop explicitly over individual indices. Still, we believe that the biggest advantage is not the compactness, but the increased resemblance with the language of the problem domain. This becomes even more explicit when we raise the abstraction level to matrix algebra. Languages such as Matlab support matrix multiplication, which allows algorithms to be coded in a way that resembles the original matrix algebra formulation to a very high degree.

However, even if the last decade's development of appropriate software abstractions for scientific computing has been significant, it is often the case that the most natural abstractions from the problem domain are not utilized. Instead it is often necessary to restate the problem, very early in the implementation process, to suit the abstractions in the implementation domain. For example, in order to formulate a linear system of equations from a discretized partial differential equation, multi-dimensional quantities that actually belong to categories 1 and 2 above are often transformed into two-dimensional matrices and one-dimensional vectors of category 3. Sometimes, the reformulation can be avoided with index notation as used in tensor calculus, because index notation offers a powerful way of expressing generalized matrix multiplications [5].

Index notation is a convenient way to express and manipulate multi-indexed quantities, including tensors. In its general form, these quantities are indexed with both upper and lower indices[1]. The compactness of index notation is due to two conventions, the range convention and the Einstein summation convention.

[1]There is an underlying physical/mathematical motivation to use both upper and lower indices for tensors, namely to distinguish so called covariant vector components from contravariant vector components. See e.g. [6] for more details.

Briefly speaking, the range convention implies that "free" indices shall loop over all values in a clearly understood range. The summation convention was introduced by Einstein in 1916, see [7]. It implies that whenever an index in a term is repeated, once as a superindex and once as a subindex, a summation over that index is understood. With these conventions, matrix vector multiplication is written

$$b^i = A^i_j c^j. \tag{1}$$

Here, the "matrix" A has one upper and one lower index, and the "column vectors" x and b use a superindex. The summation convention implies summation over j, whereas i varies over its index range, in accordance with the range convention. Compared with matrix algebra, a small advantage with index notation is that the multiplication order is not critical, as it is for matrix algebra. However, the real power of index notation is released when the number of indices increase. Consider, for instance, transformation of a tensor E with two upper and two lower indices, given a transformation matrix Λ with inverse $\bar{\Lambda}$:

$$\hat{E}^{ij}_{kl} = \bar{\Lambda}^i_m \bar{\Lambda}^j_n \Lambda^p_k \Lambda^q_l E^{mn}_{pq}. \tag{2}$$

With some practice of using index notation, it is easy to interpret this expression. The indices i, j, k and l are free indices, whereas m, n, p and q are summation indices. It is cumbersome to express this transformation with matrix algebra, let alone the difficulties of representing E at all!

Equations such as (2) are frequent in tensor calculus, the mathematics used for instance for relativity and differential geometry. We stress that the task of supporting index notation is motivated not only by its abilities to support generalized matrix multiplications, but it is also highly motivated by its enormous usefulness for tensor calculus, see for instance [8, 6].

In this paper, we define a domain specific language for index notation, and we report on its implementation as a C++ library. In order to lay a firm foundation for this task, we have studied possible index notation ambiguities and limitations. There actually exist a few different interpretations of index notation. In practice, the expressions used in tensor calculus are often quite simple, and the differences between different interpretations do not become visible. It is, however, utterly important that the notation is uniform and unambiguous, in order to support the notation in a computer language. To this end, we develop grammar and semantics for index notation. We find that expressions written in this language obey usual associative and commutative rules for summation and multiplication. Index notation support has also been investigated in [9], where a preprocessor for a less general version of index notation, intended for computer graphics, is described.

The remainder of this paper is outlined as follows. In Section 2, we present some challenges for the notation, situations where different index notation descriptions may yield different interpretations. In Section 3, we develop an index notation grammar and describe its semantics. In Section 4, we outline our object-oriented design and implementation, and we provide some code samples. In the concluding remarks, we also point at some future work.

2 Challenges for index notation

In the same manner as above, index notation and the Einstein summation convention are often described rather intuitively in the literature, see for instance [10, p. 338] or [6, p. 56]. This is usually good enough for humans, but in order to develop software support for the notation, we need to study the notation more carefully. In this subsection we illustrate some cases where different authors may interpret index notation differently, and we motivate our choiceof interpretation. It is to a high extent based on Papastavridis [8], the most comprehensive description of index notation that we have encountered. We also compare with Barr [9]. Barr states simple and concise rules for index notation, but does not distinguish between upper and lower indices. Also, as discussed below, we believe his rules to be somewhat more restrictive than necessary.

The following equations represent a number of situations where different descriptions of index notations may yield different interpretations.

$$a_i = b_j \tag{3}$$
$$a_i = b^i + c_i \tag{4}$$
$$a_i = b^i_i \tag{5}$$
$$a^i_i = b^i_i \tag{6}$$
$$a = b^i c_i d_i e^i \tag{7}$$
$$a_i = b_i + 4. \tag{8}$$

Equation (3) is probably a typo. A human can easily detect this, perhaps replacing the equation with $a_i = b_i$, making more sense. Software for index notation must of course be employed with error detection mechanisms.

Equation (4) might be a typo as well. In a physics context, it might be strange to add two tensors where the indices are placed differently. Barr does not address this issue, since he does not distinguish upper and lower indices, but Papastavridis' range convention allows it. This is also our choice.

Equation (5) could, according to some definitions, be interpreted as $a_i = \sum_j b^j_j$. However, this is probably not the intention. It is more likely that one wants to treat the diagonal elements of the “matrix” $b^i_j, \forall i, j$ as the “vector” $b^i_i, \forall i$. In order to explicitly suppress summation, many authors provide additional notation. One common alternative is textual information in the margin, which is not well suited for software support. Another possible solution is a “no-sum” operator, suggested by Barr [9]. Yet an alternative, advocated for instance by Papastavridis [8], is to suppress summation by enclosing a repeated index which is not a summation index with parentheses. The latter alternatives are possible to support with software, but is it really needed? The range convention that Papastavridis describes does actually, for equation (5), suppress the summation without need of any additional information. This is because i is considered a free index,in the term a_i. We believe that, as long as we deal with assignments, additional “no sum” notation is not needed. This is demonstrated by equation (6). In standard treatments, it would be interpreted as

$\sum_i a_i^i = \sum_i b_i^i$. But if we regard the statement as an assignment, a summation on the left hand side makes no sense. It is reasonable to modify the conventions so that summation is always suppressed on the left hand side. The expression then reads $\forall i : a_i^i = b_i^i$.

Summation convention definitions do sometimes state that summation is understood for indices repeated *exactly* twice, e.g. [9, 7, 6]. In practice, this is the kind of summation that normally arise in tensor calculus. Barr argues that this restriction is needed to maintain associativity, since generally $(\sum_i b^i c_i)(\sum_i d_i e^i) \neq (\sum_i b^i d_i)(\sum_i c_i e^i)$. Therefore, he rules out equation (7). But Bolton, [11], provides examples where summation is understood over an index repeated thrice. Papastavridis also acknowledges that summation over indices repeated more than twice does arise, but he keeps the summation sign for these cases, writing equation (7) as $a = \sum_i b^i c_i d_i e^i$ [8, p. 11]. Summation over multiply repeated indices is the interpretation we support, and we claim that no associativity problems occur.

Our last example, equation (8), is simple for humans to interpret. But suppose that one, as Barr does, imposes a restriction that addition is allowed only between terms whose indices match exactly. Even though the restriction seems plausible, it actually invalidates the assignment. To address this complication, Barr introduces the notion of an "indexed constant", and requires equation (8) to be written $a_i = b_i + 4_i$. Our interpretation of index notation, as described in the next section, allows terms to be added as long as one without ambiguity can bind all indices. Not only is equation (8) valid, but also more complicated expressions such as $a_i = b_i + c_j^j + d_{ij}^j + 4$. We also interpret $a = b^j(c_j + d_j)$, which has the same effect as $a = b^j c_j + b^j d_j$.

3 Formalizing the index notation

In order to develop software for the index notation, it is important that the notation is concise, unambiguous, and not overly restrictive. In this section, we suggest a grammar for index notation, and we provide an interpretation of this grammar. The formal specifications developed in this section also provides algorithms which are a good basis for the implementation. It is also noted that tensor expressions interpreted in this way obey the usual mathematical rules for summation and multiplication.

The grammar definitions below provide a syntax for the expressions we can formulate. Here and subsequently, we will use the term "tensor" for quantities that actually are multi-indexed arrays[2].

```
V ::=  a | b | c ...          // tensor variables
I ::=  i | j | k ...          // index variables
L ::=  ∅ | L I                // list
T ::=  V(L,L)                 // tensor expression atom
```

[2]Strictly speaking, a multi-indexed array is not a tensor, but every tensor may be represented in a chosen basis as an array with two multi-indices.

```
X ::=  T | XX | X+X | (X)       // tensor expression
A ::=  T=X                      // assignment
```

We assume the usual precedence for addition, multiplication and expressions in parentheses. An empty list is denoted $\emptyset$.

To formulate interpretation algorithms, we will frequently use sets and pairs of sets. Therefore, we introduce some auxiliary definitions:

- The space of all sets of the index variables: $\mathcal{S} = \mathcal{P}(\{[\![\mathtt{i}, \mathtt{j}, \mathtt{k}, \ldots]\!]\})$

  For the cartesian product space $\mathcal{S} \times \mathcal{S}$, we define union and intersection as

$$\begin{aligned} (u_1, l_1) \cup (u_2, l_2) &= (u_1 \cup u_2, l_1 \cup l_2) \\ (u_1, l_1) \cap (u_2, l_2) &= (u_1 \cap u_2, l_1 \cap l_2). \end{aligned}$$

  Reduction operators $\sqcup, \sqcap : \mathcal{S} \times \mathcal{S} \to \mathcal{S}$ are defined as

$$\sqcup(u, l) = u \cup l, \ \sqcap(u, l) = u \cap l.$$

- We denote the mathematical domain as "math". To avoid drowning in details, we omit a rigorous description of this domain. For example, we do not formalize how an index i belongs to a certain range, but we assume the range to be clearly understood. Thus, the "math" expression $\forall s : A$ where $s \in \mathcal{S}$ is a shorthand for the following: For all (integer) values of all indices $i \in s$, execute statement A. Similarly, summation over $s \in \mathcal{S}$, $\sum_s x$, means that all indices in the set s shall take all values in their respective range, and the resulting values x which depends on the values of the indices in s shall be summed together. Summation over an empty set is defined to yield the expression summed over, that is $\sum_{\emptyset} x = x$.

We are now ready to define semantics and auxiliary functions:

- The λ function generates a set of indices from a list of indices: $\lambda : \mathtt{L} \to \mathcal{S}$.

$$\begin{aligned} \lambda[\![\emptyset]\!] &= \emptyset \\ \lambda[\![\mathtt{L\ I}]\!] &= \lambda[\![\mathtt{L}]\!] \cup [\![\mathtt{I}]\!]. \end{aligned}$$

- $[\![\mathtt{v(u,l)}]\!] : \mathtt{T} \to$ "math" indicates a tensor component v_l^u for the tensor variable $v = [\![\mathtt{v}]\!]$, according to the values of the indices in the upper multi-index $u = [\![\mathtt{u}]\!]$ and the lower multi-index $l = [\![\mathtt{l}]\!]$. An attempt to evaluate this expression when the indices in $[\![\mathtt{u}]\!]$ and $[\![\mathtt{l}]\!]$ are not bound is an error (cf. Definition 1 below).

- The ϕ function generates a pair of sets from an expression. $\phi : \mathtt{X} \to \mathcal{S} \times \mathcal{S}$.

$$\begin{aligned} \phi[\![\mathtt{v(u,l)}]\!] &= (\lambda[\![\mathtt{u}]\!], \lambda[\![\mathtt{l}]\!]) \\ \phi[\![\mathtt{ab}]\!] &= \phi[\![\mathtt{a}]\!] \cup \phi[\![\mathtt{b}]\!] \\ \phi[\![\mathtt{a+b}]\!] &= \phi[\![\mathtt{a}]\!] \cap \phi[\![\mathtt{b}]\!] \\ \phi[\![\mathtt{(a)}]\!] &= \phi[\![\mathtt{a}]\!]. \end{aligned}$$

- The ε function generates a summation set from an expression. $\varepsilon : \mathtt{X} \to \mathcal{S}$.

$$\varepsilon[\![x]\!] = \sqcap\phi[\![x]\!].$$

- The E function evaluates an expression, given a set of bound variables. $\mathsf{E} : \mathtt{X}, \mathcal{S} \to$ "math".

$$\begin{aligned}
\mathsf{E}([\![\mathtt{v(u,l)}]\!], s) &= \sum_{\varepsilon[\![\mathtt{v(u,l)}]\!]\setminus s} [\![\mathtt{v(u,l)}]\!] \\
\mathsf{E}([\![\mathsf{ab}]\!], s) &= \sum_{\varepsilon[\![\mathsf{ab}]\!]\setminus s} \mathsf{E}([\![\mathsf{a}]\!], s \cup \varepsilon[\![\mathsf{ab}]\!])\mathsf{E}([\![\mathsf{b}]\!], s \cup \varepsilon[\![\mathsf{ab}]\!]) \\
\mathsf{E}([\![\mathsf{a}+\mathsf{b}]\!], s) &= \sum_{\varepsilon[\![\mathsf{a}+\mathsf{b}]\!]\setminus s} (\mathsf{E}([\![\mathsf{a}]\!], s \cup \varepsilon[\![\mathsf{a}+\mathsf{b}]\!]) + \mathsf{E}([\![\mathsf{b}]\!], s \cup \varepsilon[\![\mathsf{a}+\mathsf{b}]\!]) \\
\mathsf{E}([\![(\mathsf{a})]\!], s) &= \mathsf{E}([\![\mathsf{a}]\!], s).
\end{aligned}$$

- The `sem` function generates a mathematical expression from an assignment: $\mathtt{sem} : \mathtt{A} \to$ "math".

$$\mathtt{sem}[\![\mathtt{v(u,l)} = \mathtt{x}]\!] \;=\; \forall(\sqcup\phi[\![\mathtt{v(u,l)}]\!]) :\; v_l^u = \mathsf{E}(x, \sqcup\phi[\![\mathtt{v(u,l)}]\!])).$$

Finally, we note that an expression is valid only if its interpretation binds all index variables when a specific component is referred to. This rules out assignments such as (3).

Definition 1 *An invalid expression is an expression where the described algorithm leads to an attempt to evaluate* $\mathsf{E}([\![\mathtt{v(u,l)}]\!], s)$ *when* $\sqcup\phi[\![\mathtt{v(u,l)}]\!] \notin (\varepsilon[\![\mathtt{v(u,l)}]\!] \cup s)$*.*

It is interesting to study the implications of the grammar and its semantics. In [12], this is done more rigorously. Here, we just note that the familiar mathematical relations for addition and multiplication holds:

$$\begin{aligned}
\mathsf{E}([\![\mathsf{a}+\mathsf{b}]\!], s) &= \mathsf{E}([\![\mathsf{b}+\mathsf{a}]\!], s) && (9)\\
\mathsf{E}([\![\mathsf{ab}]\!], s) &= \mathsf{E}([\![\mathsf{ba}]\!], s) && (10)\\
\mathsf{E}([\![(\mathsf{a}+\mathsf{b})+\mathsf{c}]\!], s) &= \mathsf{E}([\![\mathsf{a}+(\mathsf{b}+\mathsf{c})]\!], s) && (11)\\
\mathsf{E}([\![(\mathsf{ab})\mathsf{c}]\!], s) &= \mathsf{E}([\![\mathsf{a}(\mathsf{bc})]\!], s) && (12)\\
\mathsf{E}([\![\mathsf{a}(\mathsf{b}+\mathsf{c})]\!], s) &= \mathsf{E}([\![\mathsf{ab}+\mathsf{ac}]\!], s). && (13)
\end{aligned}$$

These are the properties of a commutative ring.

4 Programming with index notation

In the previous section, we developed an index notation grammar with semantics to be used as a domain specific language. It may be supported in various ways.

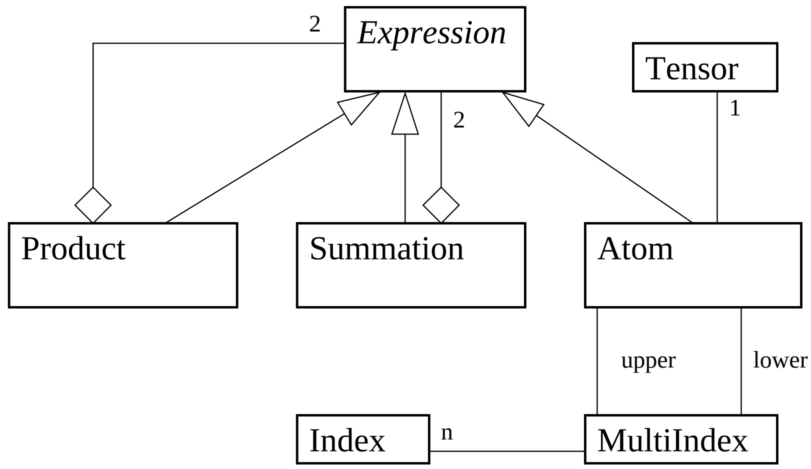

Figure 1: The key structure in the design is an expression tree, as outlined in this class diagram. `Product`, `Summation`, and `Atom` inherits `Expression`. `Expression` is a recursive aggregate [13] because `Product` and `Summation` are binary operators, having two `Expression` as operands. Each `Atom` in the expression tree is connected to one `Tensor` and two `Multi-indices`, one for the upper indices and one for the lower.

We have chosen to implement support for index notation as a C++ class library. This allows users to program directly with index notation, avoiding a notational gap between index notation and standard array or matrix packages. In this section, we sketch our object-oriented design and implementation. We also give program examples, demonstrating the usefulness of the notation.

4.1 An object-oriented implementation

It is natural to use object-orientation in order to implement software support for index notation, since we can introduce new data types as classes. We find that C++ is suitable for our purposes. A crucial point is that C++ allows operator overloading, necessary in order to obtain code that resembles the original notation. Even if the present version does not focus on efficiency, it is also possible to utilize C++ to achieve high performance, see for instance [14].

Inspecting our previous sections, a number of class abstractions are easily found, for instance index, multi-index, tensor, tensor expression, tensor summation, tensor product. Figure 1 illustrates the relation between these abstractions, using UML [15]. The classes are summarized below.

An `Index` represents an index with an explicit identity. An `Index` can be either in the state *free* or *bound*. A loop over a free `index` binds the index and loops over all the values in a specific range. For example, we may declare[3]

```
EinIndex I(0,2);
```

This declaration yields an index `I` ranging from 0 to 2. A loop over a bound index executes only over its current value.

A `Multi Index` contains a list of `Indices`. In order to facilitate the construction of `Multi Indices`, we have overloaded `operator|`, to construct `Multi Indices` from `Indices`. Thus, given three indices `I`, `J` and `K`, we can declare

```
EinMultiIndex M = I|J|K|I;
```

as a `Multi Index` that holds three `Indices`, one `Index` repeated twice. A loop over a `Multi Index` yields a nested loop over its `Indices`.

A `Tensor` is a tensor variable. Given `Indices I`, `J` and `K`, we may declare a tensor with two upper indices and one lower index as

```
EinTensor T(I|J,K);
```

Note that we do not overload the usual arithmetic operators to take `Tensor` arguments, because the grammar does not provide arithmetics for tensor variables. Instead, the arithmetic operators operate on tensor expressions.

An `Expression` is the base class for tensor expressions. Since all tensor expressions have the same properties, inheritance is natural. The `E` function of Section 3 is declared as `eval()` in the base class, deferring its implementation to subclasses.

`Product` and `Summation` have two operands which are `Expressions`. These objects are created via arithmetic operators that take `Expressions` as arguments, thus building expression trees. We have also implemented inheritors for `Subtraction` and `Unary Minus`. The expression tree is automatically built with the correct operator precedence [16, Section r.5].

`Atoms` are the leaves of the expression tree. `Atoms` are generated by `Tensor::operator()`, taking two multi-indices as arguments. For example, `T(I|I,K)` would represent T_k^{ii}, assuming proper declarations. For tensors

[3] In our implementation, we prefix all classes with `Ein`, honoring Einstein as the summation convention inventor.

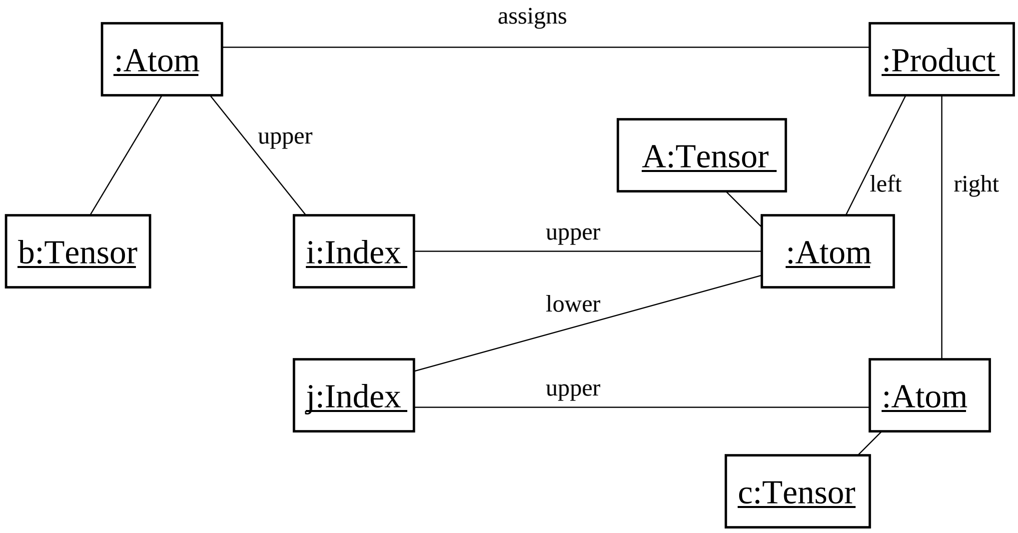

Figure 2: This instance diagram shows objects and necessary relations between them, representing equation (1).

who only have upper or only lower indices, we also provide a version of `Tensor::operator()` with only one multi-index argument. For example, if `B` is declared to have one upper index and no lower indices, `b(I)` represents b^i. In accordance with the grammar, `operator=` is implemented for `Atoms`.

The interpretation of index notation expressions is mainly done in the assignment operator of `Atoms` and in the `eval()` method of `Expressions`, as illustrated in Figure 2. The instance diagram shows relevant associations between objects that carry out a multiplication (1). Written in code, the assignment would be

```
b(I) = A(I,J)*c(J);
```

Here, `b`, `A` and `c` are `Tensors` and `I` and `J` are `Indices`. The `Atom` object associated with `b` loops over `I`, assigning the components of `b` the real value computed by `Product.eval()`. This value is for each value of `I` computed as a sum over `J`. Each term in this sum is computed as the product of the real values returned by the operands' `eval()` method.

4.2 Application examples

We have developed various test programs to confirm that the index notation is interpreted as intended, both with respect to the challenging equations (3) to (8) of Section 2 and with respect to properties (9) to (13) of Section 3. We have also developed a few example programs demonstrating the usefulness of index notation, for instance coordinate transformations (2), computation of cross products, and numerical quadrature for multi-dimensional domains, discretized with structured grids. In this presentation, we show a routine that given three linearly independent vectors constructs an orthonormal base, i.e., a base where the basis vectors are mutually orthogonal and of length one.

We use the well-known Gram-Schmidt algorithm, which transforms the vectors a, b, c into an orthonormal base with respect to a metric g_{ij} as follows:

$$
\begin{aligned}
a^k &= \frac{1}{\sqrt{g_{ij}a^i a^j}}\, a^k \\
b^k &= b^k - g_{ij}a^i b^j a^k \\
b^k &= \frac{1}{\sqrt{g_{ij}b^i b^j}}\, b^k \\
c^k &= c^k - g_{ij}a^i c^j a^k - g_{ij}b^i c^j b^k \\
c^k &= \frac{1}{\sqrt{g_{ij}c^i c^j}}\, c^k .
\end{aligned}
$$

Note that the algorithm actually is not supported by the grammar. The complication is that the expressions under the root signs are understood to be evaluated independently. In our implementation, this is taken care of by the type converting system. The following code is therefore very close to the original mathematical notation:

```
void GramSchmidt(const EinTensor &G, // Metric is (0,2) tensor
                 EinTensor &a,       // (1,0) tensor assumed
                 EinTensor &b,       //        "
                 EinTensor &c)       //        "
{
  EinIndex I(0,2), J(0,2), K(0,2);

  a(K) = 1/sqrt(G(I|J)*a(I)*a(J))*a(K);

  b(K) = b(K) - (G(I|J)*b(I)*a(J))*a(K);
  b(K) = 1/sqrt(G(I|J)*b(I)*b(J))*b(K);

  c(K) = c(K) - (G(I|J)*c(I)*a(J))*a(K)
    - (G(I|J)*c(I)*b(J))*b(K);
  c(K) = 1/sqrt(G(I|J)*c(I)*c(J))*c(K);
}
```

5 Concluding remarks

We have developed software that supports index notation, well-known from tensor calculus. We emhasize that it can also be used for manipulating multi-indexed arrays in general. Index notation has long been a useful tool for many mathematical and physical disciplines, for instance differential geometry and relativity. Since many areas of physics and engineering use index notation and not matrix algebra as the main mathematical tool, it is important to develop class libraries that support the notation actually used. This achieves more compact code, and it avoids a notational gap between the mathematical formulas and the program code.

To support index notation as a domain specific language, we need unambiguous rules for its interpretation. Therefore we propose grammar and semantics for index notation. Our rules support the range convention, which implies that free indices implicitly loop over their range, as well as the Einstein summation convention, which implies that summation is understood for a term with a repeated index. We also find that usual mathematical rules for summation and multiplication are supported. Compared with Barr [9], our rules are more general, mainly because we distinguish between upper and lower indices, we allow indices to be repeated more than twice, and we allow summation between terms even if the indices do not match exactly, as long as the expression can be interpreted unambiguously. Our interpretation is mainly based on Papastavridis comprehensive description of index notation [8], but we have modified the rules to suit the imperative context of assignments. By only treating Einstein summation on the right hand side of an assignment, we believe that there is no reason to introduce an explicit "no sum" notation.

We have implemented support for index notation as a C++ class library. With these classes, a mathematical algorithm written with index notation may readily be converted into code. When executed, summation according to the Einstein summation convention as well as loops over free indices are carried out automatically. Various examples demonstrate the close resemblance between mathematical index notation and the corresponding code.

This paper represents a first step to support index notation as a domain specific language. Regarding future work, many directions are interesting. One issue, which we currently are investigating, is the representation of general symmetries and anti-symmetries, present in many tensor applications. By carefully exploiting symmetries, memory requirements can greatly be reduced. A similar issue is the representation of "sparse tensors", which may arise when simulating partial differential equations [5]. Finally, we acknowledge that index notation as used in tensor calculus is also equipped with notation for partial derivatives. Support for this extension of index notation is desirable, but we believe that many other steps remain before this goal is reached.

Acknowledgements

I am very grateful to Magne Haveraaen for his contributions, in particular concerning Section 3. I am also grateful to several friends for encouragement and valuable feedback.

References

[1] K. Åhlander, M. Haveraaen, and H. Munthe-Kaas. On the role of mathematical abstractions for scientific computing. To appear in the proceedings of IFIP's 8th Working Conference on "Software Architectures for Scientific Computing Applications", Ottawa, Ontario, Canada, 2–4 October 2000.

[2] M. Lemke and D. Quinlan. P++, a parallel C++ array class library for architecture-independent development of structured grid applications. *ACM SIGPLAN Notices*, 28(1):21–23, 1993.

[3] J. Reynders et al. Pooma: A framework for scientific simulations on parallel architectures. In G. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 553–594. MIT Press, 1996.

[4] T. Veldhuizen and K. Ponnambalam. Linear algebra with C++ template metaprograms. *Dr. Dobb's Journal*, August 1996.

[5] K. Otto. A tensor framework for preconditioners based on fast transforms. Report in preparation.

[6] B. Schutz. *Geometrical Methods of Mathematical Physics*. Cambridge University Press, 1980.

[7] Encyclopaedia of mathematics. Entry on Einstein rule.

[8] J.G. Papastavridis. *Tensor calculus and analytical dynamics*. Library of engineering mathematics. CRC Press LLC, 1999.

[9] A. Barr. The Einstein summation notation, introduction to Cartesian tensors and extensions to the notation. Available on `http://www.gg.caltech.edu/~cs174ta/1998-99/Fall/index.html`, as of 11/10 2000.

[10] R. Abraham, J.E. Marsden, and T. Ratiu. *Manifolds, tensor analysis, and applications*, volume 75 of *Applied Mathematical Sciences*. Springer-Verlag, 2nd edition, 1988.

[11] E. Bolton. A simple notation for differential vector expressions in orthogonal curvilinear coordinates. *Geophysical Journal International*, 115(3):654–666, dec 1993.

[12] K. Åhlander and M. Haveraaen. Index notation semantics. Report in preparation.

[13] J. Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[14] E. Arge et al. On the numerical efficiency of C++ in scientific computing. In M. Dæhlen and A. Tveito, editors, *Numerical Methods and Software Tools in Industrial Mathematics*, pages 93–119. Birkhäuser, 1997.

[15] J. Rumbaugh, Jacobson I, and Booch G. *The Unified Modeling Language Reference Manual*. Addison Wesley Longman, Reading, Massachusetts, 1999.

[16] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1991.