REPORTS IN INFORMATICS

ISSN 0333-3590

Supporting tensor symmetries in EinSum

Krister Åhlander

REPORT NO 212 **June 2001**

Department of Informatics

UNIVERSITY OF BERGEN

Bergen, Norway

This report has URL `http://www.ii.uib.no/publikasjoner/texrap/ps/2001-212.ps`

Reports in Informatics from Department of Informatics, University of Bergen, Norway, is available at `http://www.ii.uib.no/publikasjoner/texrap/`.

Requests for paper copies of this report can be sent to:

Department of Informatics, University of Bergen, Høyteknologisenteret, P.O. Box 7800, N-5020 Bergen, Norway

Supporting tensor symmetries in EinSum*

Krister Åhlander

Department of Informatics
University of Bergen
N-5020 Bergen
Norway

June, 2001

Abstract

Exploiting symmetries are important in numerical mathematics, both with respect to efficient memory usage and with respect to symmetry exploiting algorithms. In this paper, the symmetries of tensors are in focus. A convenient notation for describing coordinate-free tensor symmetries is established, based on sets of permutations. Completely symmetric and antisymmetric tensors are included as special cases. The extensions to multidimensional arrays with other kinds of symmetries or invariant features are also treated.

The symmetry information is used to represent tensors with symmetries more economically with respect to memory. In addition, three algorithms that exploit symmetries are presented. First, a Frobenius norm computation is derived. Second, a projection to an index space with general symmetries is shown, and proven to be optimal in the Frobenius norm. Third, a symmetry utilizing formula for a dual mapping between completely antisymmetric index spaces is shown.

The implementation of symmetry support in EinSum is discussed. EinSum is a C++ package primarily intended for tensor algebra, capable of supporting the Einstein summation convention. Details on the symmetry part of the implementation are explained. Code for the implementation of the Frobenius norm, the general projection, and the dual mapping is shown, illustrating how symmetry aware software may decrease both the memory usage and the number of arithmetic operations.

1 Introduction

> "...patterns that have a high degree of symmetry are easier to remember than those that lack them. They contain redundant information and can be stored more economically. Symmetry offers efficiency in grasping, storing and recalling information."
>
> *Syed Jan Abas*, [1, p. 34]

There is something inherently appealing with symmetries, demonstrated for instance by the works of M. Escher or the usage of patterns in Islamic art [1]. It is

*Research funded via NFR-project 123585/410. Email: `krister@ii.uib.no`

also evident in nature, where some species are believed to choose breeding partners based on symmetry considerations [13]. Symmetries are actually relevant to an enormously wide range of areas, spanning from literature and dance to physics and mathematics [10, 11]. In this document, we discuss data structures suitable for exploiting symmetries in tensor calculus, with applications for instance in elasticity or relativity theory, and many other branches of physics [14]. To discuss these data structures, it is natural to use concepts from group theory. This is not surprising, since symmetries were one important source of inspiration during the development of group theory [7, p. 46].

In order to support tensor calculus on a computer, tensors are normally represented in component form, quite similar to a multi-dimensional array. In our presentation, we do observations on symmetries for tensors as coordinate-free objects, and show the implications for tensors in component form. We also discuss symmetries for general tables or arrays, not necessarily representing tensors. Thus, we describe an efficient method to store arrays which display a wide variety of symmetries.

As an additional motivation for our interest in symmetries, we present three simple algorithms that utilize symmetries. First, we discuss the computation of the Frobenius norm, which is useful in order to compare tensors in component form. Second, a projection to an arbitrary symmetry is devised, and proven to be optimal in the Frobenius norm. Third, we present a symmetry exploiting version of the Hodge star operator, a mapping between two completely antisymmetric spaces, used in tensor calculus.

When developing software for scientific computing, the importance of appropriate data abstractions in order to handle the complexity is getting more and more recognized [3]. Even in this paper, where the problem domain area is quite focused, this observation is crucial. For instance, we point out the separation of tensors from tensor spaces, and the separation of indices from index spaces. The relation between symmetries in the tensor space and the index space is utilized to devise efficient data structures for tensors in component form. Thus, tensors in component form may be declared as data structures which are indexed according to such an index space.

Symmetry support has been included in the EinSum package [1]. EinSum is a C++ package under development intended for tensor calculus [2]. The mechanisms for supporting symmetries in EinSum is described. In this context, we emphasize the tight coupling between the studied mathematical concepts and the specificied computer abstractions. It makes it possible to write C++ code with high resemblance to mathematical formulas, as illustrated by our code examples.

There are a number of projects and products with similar data structures, for instance Pooma [6], MTL [16], Maple, and Mathematica. In comparison, this paper contains a more comprehensive description of multi-dimensional index spaces with symmetries than we have seen elsewhere. Particularly, we highlight the relation between symmetries in the tensor space and the index space, and we provide a more general approach to index spaces. In addition, EinSum is also capable of supporting index notation, including the Einstein summation convention, which implies summation over repeated indices. Index notation is very common for tensor calculus.

This document is organized as follows. In Section 2 we present the basic concepts and we establish a notation for representing tensor symmetries. In Section 3 we give exemples on symmetry exploiting algorithms. The design and implementation of software for tensor symmetries are discussed in Section 4. Examples in Section 5

[1] See `http://www.ii.uib.no/~krister/EinSum/`.

demonstrate the usage of symmetries in EinSum. In Section 6 we conclude that the introduced concepts and the implemented abstractions are useful in order to utilize symmetries both in order to represent tensors more economically and to devise algorithms that exploit symmetries.

2 Relations between tensor symmetries and index space symmetries

In this section we establish a notation by recapitulating some elementary group concepts and relating them to the symmetry of tensors. For a more comprehensive treatment of groups, consult any elementary algebra book, for instance [7]. For more details on tensors, see for instance [14, 15].

In order to make the presentation clearer, we will simplify the notation and only discuss tensors in orthogonal coordinate systems, where it is not necessary to distinguish contravariant and covariant components, or upper and lower indices [4]. The extension to non-Cartesian tensors is straight-forward, and has been carried out in our implementation. For the current exposition, however, the following definition of a tensor is adequate[2]:

Definition 1 *Let $\mathcal{T}_d^n = \mathcal{V}_d \times \ldots \times \mathcal{V}_d \to \Re$ be a tensor space with dimension d and rank n, where $\mathcal{V}_d$ denotes a vector space of dimension d. A coordinate-free tensor $T \in \mathcal{T}_d^n$ is conceived as a mapping which takes n arguments $v_i \in \mathcal{V}_d$ and produces a real number,*

$$T(v_1, \ldots, v_n) \in \Re.$$

Our interest is in symmetry properties of tensors. The standard way of defining (completely) symmetric and antisymmetric tensors is as follows, where $\mathcal{Z}_n \subset \mathcal{Z}$ denotes the interval $[1, n]$.

Definition 2 *Define a completely symmetric tensor $T \in \mathcal{T}_d^n$ as a tensor whose action upon n arbitrary vector arguments is invariant when any two arguments are interchanged,*

$$T(v_1, \ldots, v_l, \ldots, v_k, \ldots, v_n) = T(v_1, \ldots, v_k, \ldots, v_l, \ldots, v_n), \ k, l \in \mathcal{Z}_n.$$

Similarly, define a completely antisymmetric tensor $T \in \mathcal{T}_d^n$ as a tensor whose action upon n arbitrary vector arguments changes sign when any two arguments are interchanged,

$$T(v_1, \ldots, v_l, \ldots, v_k, \ldots, v_n) = -T(v_1, \ldots, v_k, \ldots, v_l, \ldots, v_n), \ k, l \in \mathcal{Z}_n.$$

Often, the attribute "completely" is omitted, since these symmetries are the most common. Maxwells equations, for example, may be written in coordinate-free form as a relation between completely antisymmetric tensors [15, Section 5.11]. However, other symmetries are also of interest. One example is provided by Hookes tensor $\Lambda \in \mathcal{T}_d^4$ in elasticity, which display the following symmetries:

$$\Lambda(v_2, v_1, v_3, v_4) = \Lambda(v_1, v_2, v_4, v_3) = \Lambda(v_3, v_4, v_1, v_2) = \Lambda(v_1, v_2, v_3, v_4). \quad (1)$$

[2] The definition assumes that the concept of a vector space is understood. For a more comprehensive treatment, consult for instance [12].

So far, we have discussed tensors and symmetries strictly from a coordinate-free perspective, to emphasize that tensor symmetries are inherent in the tensors themselves, and do not belong to their representation in any particular coordinate system with basis vectors $\{e_1, \ldots, e_d\} \in \mathcal{V}_d$. However, in order to represent tensors on a computer, the component form of a tensor is convenient:

Definition 3 *The components of a tensor are real numbers obtained when the tensor acts on n basis vectors $\{e_1, \ldots, e_d\}$:*

$$T_\imath = T_{\imath_1 \ldots \imath_n} = T(e_{\imath_1}, \ldots, e_{\imath_n}).$$

The index $\imath = \imath_1 \ldots \imath_n$ contains n juxtapositioned integers in the range $\mathcal{Z}_d = [1, d]$, that is, $\imath \in \mathcal{Z}_d^n = \mathcal{Z}_d \times \ldots \times \mathcal{Z}_d$. A tensor $\{T_\imath\} \in \mathcal{T}_d^n$ in component form is conceived as a table which maps a multi-index with n indices and produces a real number,

$$\{T_\imath\} : \mathcal{Z}_d^n \to \Re. \tag{2}$$

Subsequently, we will mostly discuss tensors in component form. It facilitates the notation, and as we shortly will show, symmetric properties of tensors are valid in all coordinate systems and therefore it is no severe limitation. Any restriction of the tensor space with respect to symmetries are transferred to the component form as well. For instance, Definitions 2 and 3 immediately yield that the following properties hold: For a completely symmetric tensor in component form,

$$T_{\imath_1 \ldots \imath_l \ldots \imath_k \ldots \imath_n} = T_{\imath_1 \ldots \imath_k \ldots \imath_l \ldots \imath_n},\ k, l \in \mathcal{Z}_n, \tag{3}$$

and, similarly, for a completely antisymmetric tensor in component form,

$$T_{\imath_1 \ldots \imath_l \ldots \imath_k \ldots \imath_n} = -T_{\imath_1 \ldots \imath_k \ldots \imath_l \ldots \imath_n},\ k, l \in \mathcal{Z}_n. \tag{4}$$

In order to generalize these results to arbitrary symmetries we must refine the description of symmetries. It is natural to turn our attention to which *permutations* of the arguments do actually preserve the action of the tensor, and which permutations give the same result but with opposite sign. We use the following notation for permutations.

Definition 4 *Let P_n be the permutation group*[3] *of $\mathcal{Z}_n$, that is, all bijections $\sigma : \mathcal{Z}_n \to \mathcal{Z}_n$. The action of $\sigma \in P_n$ on an index $\imath \in \mathcal{Z}_d^n$ is given by*

$$\sigma(\imath) = \sigma(\imath_1 \ldots \imath_n) = \imath_{\sigma(1)} \ldots \imath_{\sigma(n)}.$$

Note that composition of permutations associates with action on an index: $\sigma(\rho(\imath)) = (\sigma\rho)(\imath)$.

We may now define a restriction of a tensor space with respect to a general tensor symmetry, given two arbitrary sets of permutations.

Definition 5 *Given $S \subseteq P_n$ and $A \subseteq P_n$, define $\mathcal{T}_d^n|_{S,A} \subseteq \mathcal{T}_d^n$ as the subspace of $\mathcal{T}_d^n$ such that, for every $T \in \mathcal{T}_d^n|_{S,A}$ acting on n arguments $v_i \in \mathcal{V}_d$,*

$$\begin{aligned} T(v_{\sigma(1)}, \ldots, v_{\sigma(n)}) &= T(v_1, \ldots, v_n), \forall \sigma \in S \\ T(v_{\alpha(1)}, \ldots, v_{\alpha(n)}) &= -T(v_1, \ldots, v_n), \forall \alpha \in A. \end{aligned}$$

[3] Remark: The usual notation is S_n, S indicating the symmetric group. However, "symmetric" is such an overloaded term, and in this context we choose a less orthodox notation.

The observations in equations (3) and (4) are now readily generalized. Following immediately from Definitions 3 and 5, we obtain:

Corollary 6 *For a tensor* $\{T_\imath\} \in \mathcal{T}_d^n|_{S,A}$ *written in component form,*

$$\begin{aligned} T_{\sigma(\imath)} &= T_\imath, \forall \sigma \in S \\ T_{\alpha(\imath)} &= -T_\imath, \forall \alpha \in A. \end{aligned}$$

Conversely, if these relations hold in all coordinate systems, $T \in \mathcal{T}_d^n|_{S,A}$.

The permutations in S and A are not necessarily all permutations ρ such that $T_{\rho(\imath)} = \pm T_\imath$.

Definition 7 *Given* $\mathcal{T}_d^n|_{S,A}$, *define permutation sets* $\overline{S} \subseteq P_n$ *and* $\overline{A} \subseteq P_n$ *as the sets of all permutations such that,*

$$\begin{aligned} \sigma \in \overline{S} \quad &\Leftrightarrow \quad T(v_{\sigma(1)}, \ldots, v_{\sigma(n)}) = T(v_1, \ldots, v_n), \forall T \in \mathcal{T}_d^n|_{S,A} \\ \alpha \in \overline{A} \quad &\Leftrightarrow \quad T(v_{\alpha(1)}, \ldots, v_{\alpha(n)}) = -T(v_1, \ldots, v_n), \forall T \in \mathcal{T}_d^n|_{S,A}. \end{aligned}$$

We note that $\overline{S}$ and $\overline{A}$ have the following properties.

Proposition 8 *The set* $\overline{S}$ *is a subgroup of* $\overline{S} \cup \overline{A}$, *which is a subgroup of* P_n.

Proof. The identity permutation e is in $\overline{S}$ and $\overline{S} \cup \overline{A}$. For $\sigma, \rho \in \overline{S}$,

$$T_{\sigma(\rho(\imath))} = T_{\rho(\imath)} = T_\imath,$$

all $T_\imath \in \mathcal{T}_d^n|_{S,A}$, so $\sigma\rho \in \overline{S}$. Similarly for $\overline{S} \cup \overline{A}$. □

Moreover, since $\overline{S}$ is a subgroup, $\sigma \in \overline{S} \Rightarrow \sigma^{-1} \in \overline{S}$ immediately follows. We also have:

Proposition 9 $\alpha \in \overline{A} \Rightarrow \alpha^{-1} \in \overline{A}$.

Proof. $T_{\alpha^{-1}\alpha(\imath)} = T_\imath = -T_{\alpha(\imath)}$, all $T \in \mathcal{T}|_{S,A}$, $\alpha \in \overline{A}\, \imath \in \mathcal{Z}_d^n$. Let $\jmath = \alpha(\imath)$ gives $T_{\alpha^{-1}(\jmath)} = -T_\jmath \Rightarrow \alpha^{-1} \in \overline{A}$. □

The relations between S, A and $\overline{S}$, $\overline{A}$ are important. The following result is given without proof.

Proposition 10 $\overline{S} \cup \overline{A}$ *is generated by* $S \cup A$. *If an element in* $\overline{S} \cup \overline{A}$ *can be generated by a product of elements in* $S \cup A$ *such that the product contains an even number of elements in* A, *the element is in* $\overline{S}$. *If an element in* $\overline{S} \cup \overline{A}$ *can be generated by a product of elements in* $S \cup A$ *such that the product contains an odd number of elements in* A, *the element is in* $\overline{A}$.

It is not certain that $\overline{S}$ and $\overline{A}$ are disjunct. However, if $\overline{S}$ and $\overline{A}$ are not disjunct, all components of the tensor are zero:

Proposition 11 *If*

$$\overline{S} \cap \overline{A} \neq \emptyset,$$

then

$$T \equiv 0, \forall T \in \mathcal{T}_d^n|_{S,A}.$$

Proof. Consider $\sigma \in \overline{S} \cap \overline{A}$. Then $T_{\sigma(\imath)} = T_\imath = -T_{\sigma(\imath)}$, all $\imath \in \mathcal{Z}_d^n$. Therefore $T \equiv 0$. □

As long as we are interested only in symmetries related to permutations of the arguments, the notation discussed here covers all symmetry properties a tensor may have. A completely symmetric tensor, for instance, belongs to the space $\mathcal{T}_d^n|_{P_n^{\mathtt{T}},\emptyset}$, where $P_n^{\mathtt{T}}$ is the set of all transpositions, i.e., permutations that swap just two numbers. Transpositions were the permutations used in Definition 2. Clearly, $\overline{P_n^{\mathtt{T}}} = P_n$ because every permutation can be constructed by transpositions. Thus, the space of completely symmetric tensors may also be written $\mathcal{T}_d^n|_{P_n,\emptyset}$ Similarly, the space of completely antisymmetric tensors may be denoted $\mathcal{T}_d^n|_{\emptyset,P_n^{\mathtt{T}}}$ according to definition. It may also be stated $\mathcal{T}_d^n|_{P_n^{\mathtt{E}},P_n^{\mathtt{O}}}$ where $P_n^{\mathtt{E}}$ and $P_n^{\mathtt{O}}$ are the sets of even and odd permutations, respectively. An even/odd permutation is equal to an even/odd number of transpositions. The symmetry of Hookes tensor may be given by $\mathcal{T}_d^4\big|_{P_4^{\mathtt{H}},\emptyset}$ where $P_4^{\mathtt{H}}$ is the set of all permutations generated by the permutations described in equation (1). Finally, we remark that even a tensor space with no (non-trivial) symmetries is symmetric under the identity permutation e. Therefore, it can be denoted $\mathcal{T}_d^n|_{\{e\},\emptyset}$.

We are interested in the properties of an index space $\mathcal{Z}_d^n$ when indexing a tensor in a symmetry restricted tensor space $\mathcal{T}_d^n|_{S,A}$, and we proceed by defining equivalences and orbits in $\mathcal{Z}_d^n$, based on the symmetry sets S and A.

Definition 12 *Let $\jmath \sim \imath$ be an equivalence under $\overline{S} \cup \overline{A}$ if $\sigma(\imath) = \jmath$, some $\sigma \in \overline{S} \cup \overline{A}$.*

This is reflexive, symmetric, and transitive, because $\overline{S} \cup \overline{A}$ is a group.

Definition 13 *Let the orbit $\mathcal{O}_\imath$ of $\imath$ under $\overline{S} \cup \overline{A}$ be $\{\jmath \in \mathcal{Z}_d^n \mid \jmath \sim \imath\}$. Let $|\imath|$ be the size of the orbit $\mathcal{O}_\imath$, that is, the number of elements in the orbit.*

We want to identify one element in the orbit as the canonical element. Assuming that we have an ordering of $\mathcal{Z}_d^n$, so that $\imath < \jmath$ makes sense, we can define the canonical element as follows:

Definition 14 *Let the canonical element $\bar{\imath}$ of an orbit $\mathcal{O}_\imath$ be the smallest element in orbit,*

$$\bar{\imath} = \min\{\forall \jmath \mid \jmath \sim \imath\}.$$

It is of course only necessary to store a tensor value for the canonical index in each orbit. However, if antisymmetries are present, one must keep track of the sign. It is also possible, based only on symmetry considerations, that some indices refer to components which must be zero regardless of coordinate system. We can map each index to a "sign" ($-1, 0$ or 1) as follows:

Definition 15 *Let $M_\imath \subseteq \overline{S} \cup \overline{A}$ be the set of permutations in $\overline{S} \cup \overline{A}$ such that $\mu \in M_\imath \Rightarrow \mu(\imath) = \bar{\imath}$.*

Define the sign of $\imath$ under $\overline{S} \cup \overline{A}$ as

$$s(\imath) = \begin{cases} 0, & \text{if} \quad (M_\imath \cap \overline{S}) \neq \emptyset \neq (M_\imath \cap \overline{A}) \\ 1, & \text{if} \quad (M_\imath \cap \overline{S}) \neq \emptyset = (M_\imath \cap \overline{A}) \\ -1, & \text{if} \quad (M_\imath \cap \overline{S}) = \emptyset \neq (M_\imath \cap \overline{A}). \end{cases}$$

It is not necessary to store the canonical element in those orbits that are known to have zero sign. We may also join all orbits with zero sign into one orbit, which corresponds to an amended equivalence:

Definition 16 *Let $\imath \approx \jmath \Leftrightarrow \imath \sim \jmath$ or $s(\imath) = s(\jmath) = 0$, and define orbits accordingly. The zero orbit is the orbit whose elements have zero sign.*

Note that "$\approx$" is an equivalence because all elements of a given $\sim$-orbit either have zero or non-zero sign.

We have now developed tools such that we can restrict the index space for a tensor in a symmetry restricted tensor space. The above definitions immediately yield the following result.

Proposition 17 *Let $\mathcal{Z}_d^n|_{S,A}$ denote the index space which has a sign function according to Definition 15 and orbits according to Definition 16. A tensor $\{T_\imath\} \in \mathcal{T}_d^n|_{S,A}$ can, in component form, be indexed by indices $\imath \in \mathcal{Z}_d^n|_{S,A}$. For each component $T_\imath$, the relation to the component referred to by the canonical element of $\imath$ is*

$$T_\imath = s(\imath) T_{\bar{\imath}}. \tag{5}$$

Another immediate consequence of the definitions is the following result.

Proposition 18 *A tensor $\{T_\imath\} \in \mathcal{T}_d^n|_{S,A}$ has as many independent components as there are orbits in $\mathcal{Z}_d^n|_{S,A}$, excluding the zero orbit. The set of canonical elements with non-zero sign is denoted $\overline{\mathcal{Z}_d^n|_{S,A}}$.*

We conclude this section with a generalization of the concept of a restricted index space. So far, we have developed theory based on a restricted tensor space $\mathcal{T}_d^n|_{S,A}$, and we have explained how the component form $\{T_\imath\} \in \mathcal{T}_d^n|_{S,A}$ can be represented economically by $\imath \in \overline{\mathcal{Z}_d^n|_{S,A}}$. For more general index spaces, we generalize our definition of an index space which supports symmetries as follows.

Definition 19 *Let $\{T_\imath\} \in \mathcal{T} = \mathcal{T}^{n_1 \times \ldots \times n_d}$ be a table with $N = \prod_i n_i$ elements, indexed by indices $\imath \in \mathcal{Z} = \mathcal{Z}_{n_1} \times \ldots \times \mathcal{Z}_{n_d}$. Assume an equivalence $\sim$ in $\mathcal{Z}$ and a sign function $s : \mathcal{Z} \to \{-1, 0, 1\}$, such that $s(\imath) = s(\jmath) = 0 \Rightarrow \imath \sim \jmath$. The table space $\tilde{\mathcal{T}} \subseteq \mathcal{T}$ is the $\sim$ and s restricted table space where relation (5) holds.*

This definition may be applied to tensor symmetries and to several other data structures common in linear algebra. For example, a lower triangualar matrix belongs to an index space characterized by a sign function where $s(\imath) = 0$ if $\imath_1 < \imath_2$, else $s(\imath) = 1$. A Toeplitz matrix is constant on its diagonals, and it may thus be characterized by an equivalence $\imath \sim \jmath \Leftrightarrow \imath_2 - \imath_1 = \jmath_2 - \jmath_1$. Below, we will see examples on algorithms that utilize symmetries in this general sense.

3 Some symmetry exploiting algorithms

In the previous section, theory for understanding and representing tensor symmetries was developed. We now turn our attention to a few algorithms that exploit symmetries. As a preliminary, we discuss computation of a general Frobenius norm, a useful norm in many linear algebra situations. Next, we generalize symmetrization algorithms and present a projection algorithm for restricting a component form tensor (or array) to an index space with symmetries, and we prove that this is the optimal projection in the Frobenius norm. Finally, we discuss the Hodge star operator, which is a mapping between two completely antisymmetric tensor spaces.

3.1 Frobenius norm

The Frobenius norm is a convenient norm in many linear algebra situations. Generalized to component form tensors it is defined by

$$||T||_F^2 = \sum_{\forall \imath \in \mathcal{Z}} (T_\imath)^2. \tag{6}$$

If the index space exploits symmetries, it is of course enough to sum only over the non-zero canonical elements $\imath \in \overline{\mathcal{Z}}$, and the formula becomes

$$||T||_F^2 = \sum_{\forall \imath \in \overline{\mathcal{Z}}} |\imath|(T_\imath)^2. \tag{7}$$

3.2 General symmetrization algorithm

The symmetrization of a tensor is often defined as

$$S(T)_\imath = \frac{1}{n!} \sum_{\sigma \in P_n} T_{\sigma(\imath)},$$

where the factor $\frac{1}{n!}$ comes from the number of permutations in P_n. Similarly, the antisymmetrization of a tensor is often defined

$$A(T)_\imath = \frac{1}{n!} \sum_{\sigma \in P_n} \operatorname{sgn}(\sigma) T_{\sigma(\imath)},$$

where $\operatorname{sgn}(\sigma)$ depends on if σ is an even or odd permutation. As algorithms, these are inefficient. For instance $S(T)_{111} = \frac{1}{6}(\sum_{i=1}^{6} T_{111})$. Using the terminology of the previous section, we may unite these algorithms as a projection

$$\pi : \mathcal{T}_d^n \rightarrow \mathcal{T}_d^n|_{S,A}$$

as follows:

$$\pi(T)_\imath = \begin{cases} \frac{1}{|\imath|} \sum_{\jmath \sim \imath} s(\jmath) T_\jmath, & \imath \in \overline{\mathcal{Z}} \\ 0, & \imath \notin \overline{\mathcal{Z}} \text{ and } s(\imath) = 0 \\ s(\imath)\pi(T)_{\bar{\imath}}, & \imath \notin \overline{\mathcal{Z}} \text{ and } s(\imath) \neq 0. \end{cases} \tag{8}$$

Note that $\pi(\pi(T)) = \pi(T)$, so π is indeed a projection. We also remark that the sign of an index $s(\imath)$ is a generalization of the the sign of a permutation. For the complete antisymmetric case, the definitions are consistent because $\overline{S}$ consists of all even permutations and $\overline{A}$ consists of all odd permutations. Thus, the algorithm is applicable for both symmetrization and antisymmetrization, plus all other cases where we have an equivalence and a sign defined for the index space, for instance a Toeplitz matrix. For the symmetrization and the Toeplitz matrix it is easy to show that the projection (8) yields the optimal projection in the Frobenius norm. This is also the general case.

Theorem 20 *The projection (8) of a tensor to a restricted space is optimal according to the Frobenius norm:*

$$||\pi(T) - T||_F = \min_{S \in \mathcal{T}_d^n|_{S,A}} ||S - T||_F.$$

Proof. Reorder the sum so that components in same orbit are added together:

$$||S - T||_F^2 = \sum_{\imath \in \mathcal{T}} (S_\imath - T_\imath)^2 = \sum_{\mathcal{O}} \sum_{\imath \in \mathcal{O}} (S_\imath - T_\imath)^2.$$

For $S \in \mathcal{T}_d^n|_{S,A}$,

$$\sum_{\imath \in \mathcal{O}} (S_\imath - T_\imath)^2 = \sum_{\imath \in \mathcal{O}} (S_{\overline{\imath}} - s(\imath) T_\imath)^2,$$

which has minimum for $S_{\overline{\imath}} = \frac{1}{|\overline{\imath}|} \sum_{\imath \in \mathcal{O}} s(\imath) T_\imath$ which is equal to $\pi(T)_{\overline{\imath}}$ for all independent components $\imath \in \overline{\mathcal{Z}_d^n|_{S,A}}$ of $\pi(T)$. □

3.3 Hodge star

The Hodge star operator provides a mapping $* : \mathcal{A}_d^n :\rightarrow \mathcal{A}_d^m$, for $d = m + n$ and where $\mathcal{A}_d^n = \mathcal{T}_d^n|_{P_n^{\mathrm{E}}, P_n^{\mathrm{O}}}$ denotes a completely antisymmetric tensor space, and computed as follows (Cf. the definition of $*$ in [15]):

$$*T_{\imath_1 \ldots \imath_m} = \frac{1}{n!} \epsilon_{\jmath_1 \ldots \jmath_n \imath_1 \ldots \imath_m} T_{\jmath_1 \ldots \jmath_n}, \tag{9}$$

where $\epsilon \in \mathcal{A}_d^d$ is the Levi–Civita symbol. It may be regarded as a completely antisymmetric tensor[4] whose only independent element has value 1. A summation over repeated indices are understood according to the Einstein summation convention. That is, we sum over $\jmath$, and the equation may be explicitly written as $*T_\imath = \frac{1}{n!} \sum_\jmath \epsilon_{\jmath\imath} T_\jmath$.

As before, this algorithm is not efficient, because the effect is that we divide with $n!$ a value which has been added $n!$ times. Again, the formula can be rewritten so that only canonical elements is used. Using juxtaposition of indices, $\jmath\imath \in \mathcal{Z}_d^d$, and we obtain

$$*T_{\imath_1 \ldots \imath_m} = s(\jmath\imath) T_{\jmath_1 \ldots \jmath_n}, \ \forall \imath \in \overline{\mathcal{A}_m^m}, \forall \jmath \in \overline{\mathcal{A}_n^n} \mid s(\imath\jmath) \neq 0. \tag{10}$$

Note that there is only one combination of canonical indices $\imath$ and $\jmath$ where the juxtapositioned index $\jmath\imath$ has non zero sign.

This formula allows a simple proof of the result (cf. [15, pp. 126–128])

$$**T_\jmath = (-1)^{mn} T_\jmath.$$

To see this, first note that the sign $s(\imath\jmath)s(\jmath\imath)$ depends on whether the permutation ρ which permutes $\imath\jmath$ to $\jmath\imath$ is odd or even. With $\imath\jmath = \rho(\jmath\imath)$ one finds that $s(\imath\jmath)s(\jmath\imath) = \mathrm{sgn}(\rho) = (-1)^{mn}$. The last identity follows because the permutation requires the moving of n indices $\jmath_j$ behind the m indices $\imath_i$, a total of mn transpositions. Now,

$$**T_\jmath = s(\imath\jmath) * T_\imath = s(\imath\jmath)s(\jmath\imath)T_\jmath = (-1)^{mn} T_\jmath.$$

[4] Actually, the Levi–Civita symbol is a table rather than a tensor, since it does not transform as a tensor.

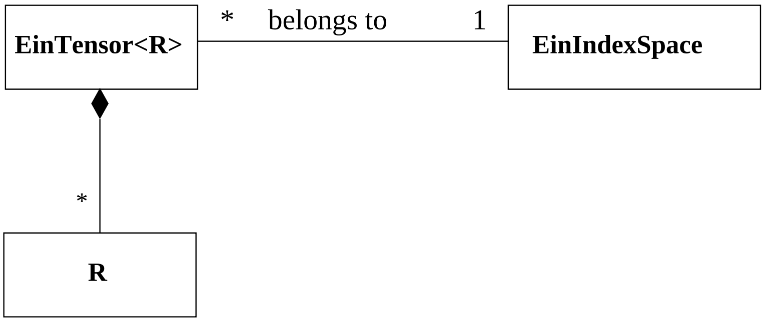

Figure 1: Class structure for tensors represented on component form. `EinTensor` is parametrized over a data type `R`, which is requested to have ring properties, so that the usual mathematical laws holds. Its components are indexed via elements in a specific `EinIndexSpace`.

4 Symmetry support in EinSum

In this section, we relate the symmetry concepts that we have developed to an actual implementation. We describe how symmetries were included in the EinSum project, a C++ package with special support for tensor algebra. The purpose of this section is twofold. The first goal is pragmatic: the implementation of the symmetry aspects of EinSum is described in order to clarify some details regarding the application examples in the next section. The second aim is more philosophical: we emphasize that the results in earlier sections should not be seen just as small intellectual exercises, but they offer valuable input for design. The earlier results are helpful in order to find data structures and operations, to choose appropriate names, and to specify the data structures and the operations.

EinSum is an ongoing project, and the implementation as well as some interfaces are likely to change in future versions. Nevertheless, we believe that the implementation described here and the application examples given in the next section nicely illustrate the close interplay between mathematical theory and the design of expressive computer packages for mathematics.

4.1 EinSum data structures and symmetry facilities

EinSum was developed preliminary to support index notation, including the Einstein summation convention. As explained in conjunction with equation (9), the convention implies a summation over repeated indices[5]. The precise semantics of EinSum index notation and the mechanisms for supporting it are described in [2]. In this paper, we concentrate on data stuctures relevant for our extensions to support symmetries.

[5] Actually, EinSum requires indices to be repeated as both upper and lower indices in order to imply summation. This requirement support general tensors. In this paper, we treat all indices as lower indices, though.

IndexSpace

```
int dim(); // N:o different indices
int size(); // N:o independent indices
int rank(); // N:o of ints in EinMultiInt
EinMultiInt orbit(MultiInt); // Canonical element
Sign sign(MultiInt); // Sign (−1,0,1) of element
int orbitSize(MultiInt); // Size of orbit
IndexSpace unrestricted(); // Returns index space with no symm
begin(); // Starts an iteration over canonical elements
bool end(); // true if iteration finished
operator++(); // iterates
```

Figure 2: Part of the index space interface.

The core data structures of EinSum is depicted in Figure 1. These abstractions are similar to those found for instance in [6, 16]. By separating the index space from the tensor, it is easier to support various indexing strategies. The tensor is responsible for its data and for providing tensor operations, whereas the index space provides various services according to the results in earlier sections. Part of the index space interface is shown in Figure 2. Notice the tight coupling between the earlier sections and this interface! For instance, an index space is requested to provide mappings from `EinMultiInt`, which is an individual element[6] in the index space, to the size of the orbit (cf. Definition 13), to the sign of the element (cf. Definition 15), and to the canonical element (cf. Definition 14). The canonical element depends, of course, on the ordering. The current implementation uses lexicographical ordering, for instance is $\imath = (123) < \jmath = (211)$. The lack of an explicit tensor space abstraction is motivated by the relation between tensor spaces and index spaces established in Proposition 17. In some situations, where coordinate free properties are important, an explicit tensor space abstraction could be advantageous, compare for instance with [9].

The algorithms in Section 3 require iterators over the independent components of an index space. In EinSum, we currently support iteration over the non-zero canonical elements of an index space `IS` using the following notation:

```
// Loop over all elements in IS
for(IS.begin(); !IS.end(); IS++) {
  EinMultiInt mi = IS; // ‘‘Current’’ element
  // ...
}
```

Note that an iteration over all components of an index space—not only the canonical elements—can be obtained using the corresponding unrestricted index space.

[6] In earlier sections, we used the term “index” for elements in an index space. However, the term index is already occupied in EinSum, where `EinIndex` is used to express index notation.

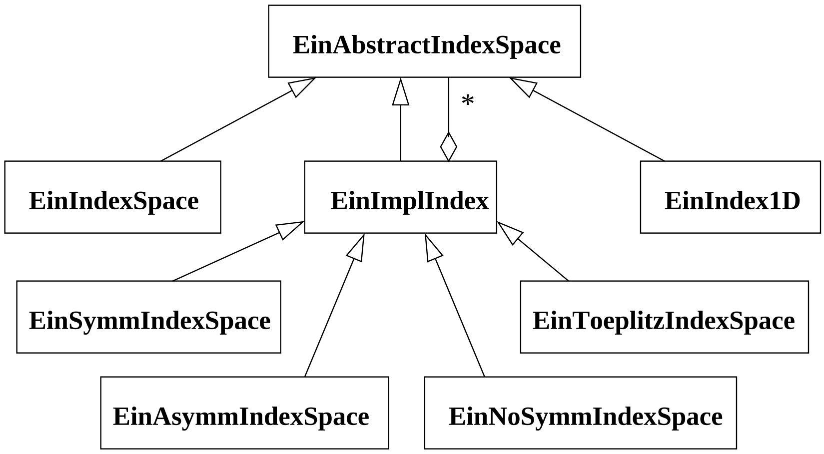

Figure 3: The index space hierarchy.

4.2 The index space hierarchy

Support of index spaces with different symmetry properties may be achieved in various ways. Obviously, Proposition 17 may be used to create an index space with symmetries from two sets of permutations, taking Propositions 8 and 9 into account to verify that the given sets are compatible. In practice, however, some tensor symmetries are more common than other, and we have chosen to support symmetry variation via an index space hierarchy, illustrated in Figure 3. Subclasses of `EinAbstractIndex` provide concrete implementations of index spaces, including various ways of constructing them. In the hierarchy, we use inheritance for subtyping, which implies that specifications valid for the abstract base class also are valid for its subclasses. `EinIndex1d` represents an index space of rank 1. `EinImplIndex` is an implementation class that collects similarities between its subclasses, in order to promote reuse. This recursive aggregate is a variation of the Composite pattern [8]. Subclasses of `EinImplIndex` represent no-symmetry, symmetry, antisymmetry, and Toeplitz symmetry. The subclass `EinIndexSpace` serves a special role. Using the proxy pattern [8], it serves as a proxy for a concrete index space. It allows the use of `EinIndexSpace` instead of a pointer to the abstract base class, which makes the memory management safer and the syntax cleaner.

The recursive aggregate allows the representation of symmetry trees, which may be defined as follows:

Definition 21 *A rank n symmetry tree consists of nodes that are index spaces and contain symmetry information regarding their children. Its n leaves consist of index spaces of rank* 1*, with specified distinct index positions.*

The nodes represent no-symmetry, symmetry, or antisymmetry. For symmetry and antisymmetry nodes, their children must have same structure. A Toeplitz index space requires two children with the same structure. Figure 4 illustrates how symmetry trees can be used for representing several common tensor symmetries. Other kinds of symmetry nodes may also be described, for instance the symmetry of a Toeplitz matrix. The tensor package in Maple (version 6) provides symmetries according to some of these symmetries (a, d, e), but apparently not completely symmetric index spaces nor the symmetries of Hookes tensor.

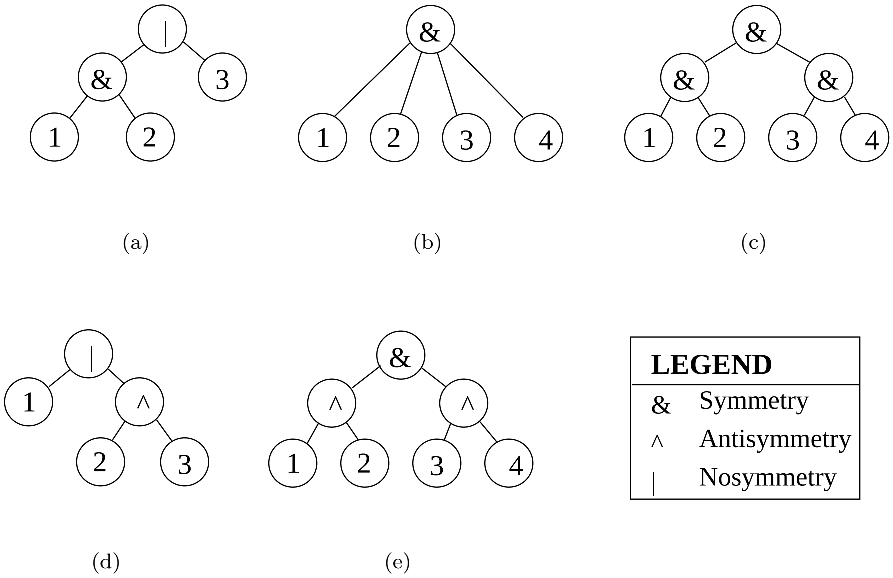

Figure 4: Tensor symmetries represented as symmetry trees. Tree (a) is symmetric in 1 and 2, tree (b) is completely symmetric, tree (c) represents symmetries of Hookes tensor according to equation (1), tree (d) is antisymmetric in 2 and 3, and the final sample tree (e) is symmetric in in 12 and 34, but antisymmetric in 1 and 2, and 3 and 4.

In C++, operator overloading can be used to denote symmetry trees. EinSum overloads `operator|`, `operator&`, and `operator^`, to construct symmetric index trees with these kind of symmetries, with the restriction that the leaves must be ordered. The interpretation relies on the fact that children to symmetric nodes in the tree must have the same structure, and it is therefore possible to distinguish the case when a new child shall be added to a symmetric node, as in `(I&J)&K`, and the case where a new symmetric node shall be created, as in `(I^J)&(K^L)`. Moreover, the precedence of `operator|` is lower than that of `operator&` and `operator^`. The syntax therefore becomes very intuitive, as exemplified in Figure 5. EinSum also provides constructors for completely (anti)symmetric spaces given an index space, etc.

4.3 Handling references which differ by sign

An important aspect of this paper is the ability to recognize the sign of an element, so that equation (5) can be used to store fewer elements. A simple way of utilizing this result in the design is to provide `get` and `set` methods in the tensor interface:

```
R EinTensor::get(const EinMultiInt &mi);
void EinTensor::set(const EinMultiInt &mi, const R& value);
```

These methods would refer to the canonical element, taking the correct sign into accordance. Notice that the setting of an element changes all elements in the orbit. However, even though these specifications are easy to implement, the interface to tensor components become quite obscured. Usually, C++ programmers overload `operator[]` as a subscript operator for arrays, in order to obtain a clear interface.

```
// Declare index objects of dimension dim.
EinIndex1d I(dim), J(dim), K(dim), L(dim);

// Declare an index space array with various symmetries
EinIndexSpace SA[] = {
  (I&J)|K,      // Symm in index 1-2
  I&J&K&L,      // Completely symmetric
  (I&J)&(K&L),  // Symm in index 1-2, and 3-4, and (1,2)-(3,4)
  I|(J^K),      // Antisymm in index 2-3
  (I^J)&(K^L),  // Antisymm in index 1-2, and 3-4, and symm (1,2)-(3,4)
};
```

Figure 5: Construction of index spaces in EinSum. These examples correspond to the symmetry trees of Figure 4.

In our case, the crux is that we can not simply return a reference to the data (the canonical element), but we must also consider the sign according to Definition 15.

This is currently achieved in EinSum by returning an object of an auxiliary class `PlusMinus<R>`, which holds a reference to the correct element and keeps track of its sign[7]. The effect is that if an element with negative sign according to Definition 15 is set to a value v, the canonical element is set to $-v$. For example, suppose `A` is an antisymmetric tensor of rank 2, `mi` represents the index $\imath = (1,3)$ and `mj` represents $\jmath = (3,1)$. In EinSum, an assignment such as

```
A[mj] = 45;
```

would then have the effect that `A[mj]` refers to the canonical element, `A[mi]`, with value -45. This technique makes it possible to treat tensors even in antisymmetric index spaces with a straight-forward syntax.

5 Application examples

In this section, we demonstrate the close resemblance between a mathematical formulation of algorithms and sample implementations using the EinSum package. The algorithms of Section 3 are written using the software that was outlined in Section 4. The code presented illustrates how symmetry support may be implemented and used to construct expressive code.

5.1 Frobenius norm

The symmetry exploiting computation of the Frobenius norm (7) may be formulated using EinSum as follows.

```
R frobenius(const EinTensor<R> &A)
{
  R ret = 0, value;

  EinIndexSpace AI = A.getIndexSpace();

  for(AI.begin(); !AI.end(); AI.next())
```

[7] An alternative solution would be to use a variation of the reference counting idiom [5].

```
    {
      EinMultiInt mi = AI;
      value = A[mi];
      ret += AI.orbitSize(mi) * value * value;
    }
  return sqrt(ret);
}
```

Here, A is a tensor whose elements is of type R, for instance a double precision floating point number, or an array of integers, etc. The loop over the appropriate index space AI is only over the canonical elements of the index space. The index space AI also plays the role of an iterator, and its current value is converted to the object mi.

Of course, a version which does not exploit symmetries may be coded as follows:

```
R frobenius0(const EinTensor<R> &A)
{
  R ret = 0, value;

  EinIndexSpace AI = A.getUnrestrictedIndexSpace();

  for(AI.begin(); !AI.end(); AI.next())
    {
      EinMultiInt mi = AI;
      value = A[mi];
      ret += value * value;
    }
  return sqrt(ret);
}
```

5.2 General symmetry projection

The following algorithm projects a tensor onto an index space with symmetries, according to formula (8). Tensor A is projected upon tensor B.

```
void applyProjection(const EinTensor<R> &A, EinTensor<R> &B)
{
  EinIndexSpace AI = A.getUnrestrictedIndexSpace();
  EinIndexSpace BI = B.getIndexSpace();

  // Zero B
  B(BI) = 0;

  for(AI.begin(); !AI.end(); AI.next() ) {
    EinMultiInt mi=AI;
    if(BI.sign(mi) != 0) {
      B[mi] += A[mi] / BI.orbitSize(mi); // 1
    }
  }
}
```

Notice, specifically, that B[mi] is sign aware. If BI.sign(mi) == -1, the effect of line // 1 is that the value of the canonical element decreases.

5.3 Hodge star

Hodge star is a mapping between completely anti-symmetric spaces. The function below corresponds to formula (10). The computation of `mi` is performed by converting the current value of the index space `AI1|AI2` according to the iterators `AI1` and `AI2`.

```
void HodgeStar(const EinTensor<R> &A,          // Tensor in
               EinTensor<R> &starA)            // Tensor out
{
  EinIndexSpace AI1 = A.getIndexSpace();
  EinIndexSpace AI2 = starA.getIndexSpace();
  EinIndexSpace IS = EinAsymmIndexSet(AI1|AI2);

  for(AI1.begin(); !AI1.end(); AI1++) {
    for(AI2.begin(); !AI2.end(); AI2++) {
      EinMultiInt mi = AI1|AI2;
      if(IS.sign(mi) != 0 )
        starA[AI2] = IS.sign(mi) * A[AI1];
    }
  }
}
```

We remark that it is easy to code the formula (9) as well. If we declare `epsilon` as a tensor over `IS` whose only independent value is 1, and compute `invnfac` as the inverse of $n!$, n being the rank of `AI1`, the following function employs the Einstein summation convention to compute $*A$.

```
void HodgeStar0(const EinTensor<R> &A,          // Tensor in
                EinTensor<R> &starA)            // Tensor out
{
  EinIndexSpace AI1 = A.getIndexSpace();
  EinIndexSpace AI2 = starA.getIndexSpace();
  EinIndexSpace IS = EinAsymmIndexSet(AI1|AI2);

  EinTensor<R> epsilon(IS, IS.rank(), 0);
  epsilon(IS) = 1;

  double invnfac = 1./fac(AI1.rank());
  starA(AI2) = invnfac * epsilon(AI1|AI2) * A(AI1); // 2
}
```

Currently, EinSum utilizes symmetries in the assignment above (line `// 2`), but EinSum is not capable of utilizing symmetries automatically when computing the product. Here, more research is needed, in order to address general cases.

6 Conclusions

We have studied tensor symmetries and related them to index space symmetries, useful for representing tensors in component form. We have shown how sets of permutations may be used to generate tensor spaces with symmetries, and how to define orbits and sign functions in their corresponding index spaces, in order to represent tensors efficiently. Index space symmetries may also be used for representing general arrays with known symmetries, i.e., arrays where the elements are related via some equivalence. One such example is a Toeplitz matrix with constant values on its diagonals. Basic concepts of algebra is useful for describing symmetries, for instance that elements that are “equivalent” belong to the same “orbit”.

Algorithms that exploit symmetries are presented. A trivial example is the computation of the Frobenius norm, where one only needs to use one element in each orbit. Another example is a projection algorithm, which includes (complete) symmetrization and antisymmetrization as general cases. It is proven that this is an optimal projection in the Frobenius norm. Our third sample algorithm is a formula for computing Hodge star, an invertible transform between two completely antisymmetric spaces.

The algebraic concepts and the sample algorithms provide valuable input for software design. First, the mathematical machinery helps us to find appropriate abstractions, to find suitable names for the abstractions, and to specify their interfaces. This is illustrated by our discussion on the implementation of symmetries in EinSum. In this context, we presented symmetry trees as a suitable tool for expressing common tensor symmetries, both as a graphical notation and in actual code. We also discussed the handling of references to data which differ only by sign.

To illustrate that the EinSum implementation is a convenient tool for dealing with tensor symmetries, implementation of the algorithms discussed above were presented. Overall, we believe that the examples demonstrate that the concepts we have developed and the software we have implemented are able to handle general tensor symmetries, an important consideration in many applications. EinSum can therefore serve as a test bed for research on symmetry exploitation. More research is needed, though, for instance with respect to efficiency. One interesting issue is here the possibilities for the computer system to automatically deduce an efficient algorithm, depending on the particular symmetries at hand.

Acknowledgments

I thank Hans Munthe-Kaas for introducing me to equation (1) and inspiring me to study this subject. I thank Malin Ljungberg for many valuable discussions, Uwe Wolter for detailed comments on Section 2, and Magne Haveraaen for continuous support. I also thank Stein Krogstad for many inspiring meetings regarding rectangularly symmetric tables and the orbits of S2, the ball in R3.

References

[1] S. J. Abas and A. S. Salman. *Symmetries of Islamic Geometrical Patterns*. World Scientific Publishing Co. Pte. Ltd., Singapore, 1995.

[2] K. Åhlander. Einstein summation for multi-dimensional arrays. In E Munthe-Kaas et al., editors, *Norsk Informatikk Konferanse – NIK'2000*, pages 67–78. Tapir, Norway, 2000.

[3] K. Åhlander, M. Haveraaen, and H. Munthe-Kaas. On the role of mathematical abstractions for scientific computing. In R. Boisvert and P. Tang, editors, *The Architecture of Scientific Software*, pages 145–158. Kluwer Academic Publishers, Boston, 2001.

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

[5] J. O. Coplien. *Advanced C++ programming styles and idioms.* Addison-Wesley, Reading, Massachusetts, 1992.

[6] J.C. Cummings et al. Rapid application development and enhanced code interoperability using the POOMA framework. In S. L. L. M. E. Henderson and C. R. Anderson, editors, *Object-oriented Methods for Interoperable Scientific and Engineering Computing*, page ch. 29, Philadelphia, 1999. SIAM.

[7] J. Fraleigh. *A first course in abstract algebra.* Addison-Wesley, 5th edition, 1994.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns — Elements of Reusable Object-oriented Software.* Addison-Wesley, Reading, Massachusetts, 1995.

[9] M. S. Gockenbach, M. J. Petro, and W. W. Symes. C++ classes for linking optimization with complex simulations. *ACM Transactions on Mathematical Software*, 25(2):191–212, June 1999.

[10] I. Hargittai, editor. *Symmetry—Unifying Human Understanding*, volume 10 of *International series in Modern applied mathematics and computer science.* Pergamon Press, 1986.

[11] I. Hargittai, editor. *Symmetry 2—Unifying Human Understanding*, volume 18 of *International series in Modern applied mathematics and computer science.* Pergamon Press, 1989.

[12] J. Marsden and T. Hughes. *Mathematical Foundations of Elasticity.* Prentice-Hall, 1983.

[13] A. P. Møller. Female swallow preference for symmetrical male sexual ornaments. *Nature*, 357:238–240, May 1992.

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

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

[16] J. Siek and A. Lumsdaine. The matrix template library: Generic components for high-performance scientific computing. *Computing in Science and Engineering*, pages 70–78, 1999.