ISSN 0333-3590

Coordinate Free Numerics

Part 1: How to avoid index wrestling in tensor computations[1]

Hans Munthe-Kaas, hans@ii.uib.no
Magne Haveraaen, magne@ii.uib.no
Department of Informatics, University of Bergen,
Høyteknologisenteret, N-5020 Bergen.

REPORT NO 101 MARCH 1995

Abstract

The theory of partial differential equations have diverged in two different directions in our century; the pure mathematical presentation based on coordinate free formulations and commutative diagrams, and the applied mathematical presentation which has its origin in hand calculations. Developments in theoretical computer science indicate that much can be gained by narrowing the gap between the different approaches. The SOPHUS package [10], is a system for tensor field computations that is almost entirely depending on coordinate free formulations. In a sequence of papers, we will show the usefulness of this approach. In the present paper, we will show that commutative diagrams can be almost immediately translated into practical computer code, which automatically and invisibly performs the necessary index manipulations in tensor calculus in an optimal manner.

1 Introduction

The coordinate free language is a systematic way to treat differential equations and other objects without relying on a concrete representation in terms of a coordinate system. This language is centered around Category Theory (or Abstract Nonsense), which was invented by Eilenberg and Mac Lane [7] in 1945. The basic idea is to avoid unnecessary details that rely upon particular representations, and to concentrate on what are the important properties of objects, which is specifyed by the way they interact with the 'rest of the world'. During the second half of this century, coordinate free formulations in general and category theory in particular have become fundamental tools in pure mathematics. Current research topics in areas such as differential geometry and algebraic topology cannot be formulated (and much less solved) without this language.

In pure computer science, category theory also gains importance. The reason for this is the need for structuring software. Programming is 'what' + 'how', and category theory is a fundamental tool for understanding the 'what' part of mathematical structures. Modern programming languages are also well suited for expressing and taking benefit from abstract formulations.

In computational sciences the situation is different. Coordinate free formulations are apparently totally absent. In [10] we discuss some reasons for this. The obvious reason is that it is impossible to do computations without a coordinate system, thus coordinate free numerics seems to be a contradiction in terms! We will in a series of papers show that this is a major misconception. In the present paper we will present a technique for avoiding complicated index manipulations in tensor computations. The technique is impossible to derive and explain without the category

[1] Dedicated to Syvert P. Nørsett on the occasion of his 50'th birthday

theoretical language, but astonishingly efficient and simple to implement once it is understood! The paper is self contained w.r.t. category theory, and we hope that also readers without any previous knowledge of this subject are able to follow the general ideas.

The background for this paper is the SOPHUS project, named after the Norwegian mathematician Sophus Lie, 1842–1899. This is a joint project between Magne Haveraaen and the author, where we develop a system for doing tensor field computations on parallel computers. Coordinate free formulations are present at almost all levels of this code. This has lead to major simplifications, both in terms of portability between parallel architectures and flexibility towards changes in numerical schemes, and also simplifications in the formulation of central algorithms such as e.g. grid staggering and multigrid techniques.

2 A brief outline of SOPHUS

This section gives an explanation of why modules over a ring is so important in SOPHUS. If you do not like the talk about modules and rings, you can substitute 'module' with 'vector space', 'ring' with 'reals', and jump to the next section.

A module over a ring is *essentially* the same as a vector space over a field. The ring possess all the properties of a field except one; the property that all non–zeros have a multiplicative inverse. This difference leads to differences in modules versus vector spaces. Whereas a vector space always has a basis, the same is not always true for a module. Modules with a basis are called *free modules*.

The SOPHUS system is built around the concept of modules over a ring. At the bottom is the set of all differentiable functions from a smooth manifold to the reals, $\mathcal{F}(\mathcal{M})$. This forms a *commutative ring with unity*, i.e. we can add and multiply functions in $\mathcal{F}(\mathcal{M})$, there exist an identity function $f(x) \equiv 1$ and a zero function $f(x) \equiv 0$ and multiplication is commutative $f \cdot g = g \cdot f$ for all $f, g \in \mathcal{F}(\mathcal{M})$. At the next level, we have the set of all smooth tangent vector fields on the manifold, $\Xi(\mathcal{M})$. This forms a module over $\mathcal{F}(\mathcal{M})$, since the addition of two vector fields forms a vector field, and multiplication of a vector field with a function also forms a vector field.

In the familiar coordinate dependent formulations of PDEs, partial derivation w.r.t a particular coordinate system is omnipresent. In coordinate free formulations, explicit use of partial derivatives is forbidden. The basic result to achieve this is the fact that $\Xi(\mathcal{M})$ is naturally isomorphic to the set of all derivations on $\mathcal{F}(\mathcal{M})$, the isomorphism is via the Lie derivative. Thus if we can express vector operations without explicitly mentioning coordinates, we may also introduce coordinate free derivations. In SOPHUS, we first introduce partial derivations on $\mathcal{F}(\mathcal{M})$, then construct $\Xi(\mathcal{M})$ from these, and thereafter hide all explicit use of partial derivations.

By applying tensorial constructions on $\Xi(\mathcal{M})$ (as an $\mathcal{F}(\mathcal{M})$ module), we can construct higher order tensor fields on the manifold. These are also $\mathcal{F}(\mathcal{M})$ modules. By recursive techniques, we may lift the Lie derivative (and other derivatives) to all these fields.

In differential geometry, vector and tensor fields are introduced in a less algebraic and more geometric way, i.e. as continuous sections of vector– and tensor–bundles over the manifold. It is a basic result in algebraic topology that the algebraic approach and the geometric approach are equivalent, e.g. from $\Xi(\mathcal{M})$ as an $\mathcal{F}(\mathcal{M})$ module, we may form the tangent bundle and vice versa. Thus the module approach contain all the geometric information.

It poses some problems that not all modules are free. E.g. the reason why there exist no smooth global coordinate systems on the sphere is because $\Xi(\mathcal{M})$ is not free. These problems can be solved by the fact that vector bundles on a manifold are locally free, and that any module is the homomorphic image of a free module. This is an algebraic version of the patching technique used to construct manifolds from local coordinate maps. We may even benefit from introducing local coordinates when $\Xi(\mathcal{M})$ is free, as a way of introducing e.g. local grid refinements and domain decomposition. These aspects will be covered in forthcoming papers.

3 The problems with classical tensor notation

Let R denote the reals, and U, V, W real vector spaces, let $\times$ denote the cartesian product and $\otimes$ the tensor product. Let $[U \to V]$ denote the set of all linear maps between the spaces and $[U \times V \to W]$ the set of all bilinear maps from $U \times V$ to W. Let $U^* = [U \to R]$ be the dual space. Classical tensor notation is built around the following isomorphisms:

$$
\begin{aligned}
[U \times V \to W] &\simeq [U \otimes V \to W] \quad \text{(The left part is bilinear, the right part linear).} \\
[U \times V] &\simeq [V \times U] \\
[U \to V] &\simeq [U \otimes V^* \to R] \\
U^{**} &\simeq U \\
(U \otimes V)^* &\simeq U^* \otimes V^*
\end{aligned}
$$

If only one vector space U is involved (as is typical in differential geometry), all spaces formed by linear maps, duals, cartesian– and tensor products can be rewritten to the form: $[U^* \otimes U^* \otimes \ldots \otimes U \otimes U \to R] \simeq U \otimes U \otimes \ldots \otimes U^* \otimes U^*$. This leads to a notation such as e.g. the Riemann curvature tensor: $R^i{}_{j,k,l} \in [U^* \otimes U \otimes U \otimes U \to R]$ and Hookes law in elasticity:

$$\sigma^{i,j} = \Lambda^{i,j,k,l} \cdot \epsilon_{k,l} \ ,$$

where we use the Einstein summation convention; sum over repeated indexes. Such notation is necessary for hand calculations, since all indexes are simple integers, and all components are reals. To compensate for the introduction of these isomorphisms, the same object must be used in many different contexts. E.g. 'the' metric tensor g is commonly used as:

$$
\begin{aligned}
y^i &= g^{i,j} x_j \quad g \in [U^* \to U] \\
r &= g^{i,j} y_i x_j \quad g \in [U \times U \to R] \\
r &= g^{i,j} \epsilon_{i,j} \quad g \in [U \otimes U \to R] \\
\epsilon_k^i &= g^{i,j} \epsilon_{k,j} \quad g \in [U \otimes U \to U \otimes U^*]
\end{aligned}
$$

The only advantage of this kind of notation is that it is convenient for hand calculations. The disadvantages in computer implementations are many:

- It hides intention. What do we *really* want to do? How do we specify in a consistent way what we want to do?
- How can we guarantee that a user writes coordinate independent expressions?
- How can we specify and take advantage of symmetries? E.g. Hookes tensor $\Lambda^{i,j,k,l}$ possess several symmetries:
$$\Lambda^{i,j,k,l} = \Lambda^{j,i,k,l} = \Lambda^{i,j,l,k} = \Lambda^{k,l,i,j}$$
How can we automatically detect that only 21 of its 81 components are different? What is the minimum number of flops needed to compute Hookes law, given that both σ and ϵ are symmetric?
- Given a particular use of a tensor, how can we store the components in a systematic way and retrieve the components as fast as possible in the order needed for a particular use?
- How can we introduce mappings and tensor products between different spaces (of possibly different dimensionalities)?
- How can we debug the program and guarantee that it computes correctly?

It is impossible to simultaneously solve all these problems satisfactory within the classical framework. It is therefore a great pleasure to note that they are all solved when we base our implementation on the right category theoretical formulation, and that this approach is actually relatively simple to implement once the right formulation is found[2].

Here is an outline of our solution:

- A data set always carry (type) information showing exactly how it is going to be used as a mapping. E.g. the use of the metric tensor g in the above examples are represented by four different data sets.
- Explicit conversions are used to take a data set from one form to another.
- The type conversion is specified in the language of category theory via commutative diagrams.
- The diagram is described in program code, specifying how to do the data transform on modules.
- The program is pulled down from the 'category of modules' to a 'category of indexes'.
- This lowered program is doing 'index wrestling' (it computes where to fetch data and where to store). There is an immediate correspondence between the original diagram and this index wrestler.
- The conversion of a data set from one form to another involves no numerical computing nor any actual data movements, thus it is computationally almost free.
- For users who don't know how to draw commutative diagrams, it is trivial to put functions on top that emulate the classical tensor notation.

We cannot formalize this approach without introducing some central concepts from category theory. The following section contains the concepts needed to understand the solution.

4 Basic category theory

Good references for applications of category theory in computer science are [3, 14]. The use of categories in mathematics is described in [4, 8, 14].

4.1 Basic definitions

Definition 1 *A category $\mathcal{C}$ consists of the following:*

1. *A class $obj(\mathcal{C})$, whose elements are the* objects *of the category.*
2. *for every pair of objects $A, B \in obj(\mathcal{C})$ a class of arrows from A to B. A particular arrow f from A to B is written $f : A \to B$, and the collection of all arrows in $\mathcal{C}$ is written $arr(\mathcal{C})$.*
3. *An associative composition rule $\circ$, such that for every pair of arrows $f : A \to B$ and $g : B \to C$, the composition is $g \circ f : A \to C$.*
4. *For every object $A \in obj(\mathcal{C})$ an identity arrow $Id_A : A \to A$, such that for every arrow $f : A \to B$, $f \circ Id_A = f$ and $Id_B \circ f = f$.*

Some objects in a category have a special importance:

Definition 2 *An object $A \in obj(\mathcal{C})$ is called:*

[2] That did, however, take a long time, since the literature on this subject is not oriented towards computations.

1. *an* initial object *if there for all $B \in obj(\mathcal{C})$ exist a unique arrow $f : A \to B$.*

2. *an* terminal object *if there for all $B \in obj(\mathcal{C})$ exist a unique arrow $f : B \to A$.*

3. *a* zero object *if it is both initial and terminal.*

These objects are denoted as I, T and 0.

Example 1 *The category* **Set** *has as objects sets and as arrows (set)-functions. The initial object in* **Set** *is the empty set, and any set with only one element is a terminal object.*

Example 2 *The category* **Mod***R has as objects modules over the ring R, and as arrows linear maps between modules.* **Mod***R has a null object 0, being the module with the 0–vector as the only element. The composition:*

$$A \longrightarrow 0 \longrightarrow B$$

is called the 0–arrow, and is written

$$A \xrightarrow{0} B$$

For any pair of parallel arrows $f, g : A \to B$ there exist a sum arrow *$f + g : A \to B$, which represent the sum of f and g as linear operators.*

The goal in category theory is to characterize mathematical objects and constructions entirely via the way the object interacts with other objects (arrows in and out). The concept of elements of an object of a category is not defined (e.g. an element of a set, or a vector in a module). In the case of **Set**, there is a 1–1 correspondence between a set element $a' \in A$ and arrows of the type $a : T \to A$, where T is a terminal object. This function is always returning $a' \in A$. Generally in category theory, arrows from a terminal object can be thought of as functions returning a constant, regardless of its input, and are called *constant arrows*.

In the case of **Mod**R , the use of constant arrows to define vectors is less useful, since the only constant arrows are the 0–arrows which represent the 0–vector. Instead, since R is a module over itself, there are arrows of the type $a : R \to A$. Each vector $a' \in A$ can be associated with a unique arrow of this kind, via $a' = a(1)$. The elements of R are thus arrows $r : R \to R$. Note that the composition $a \circ r : R \to A$ represent the scalar–vector product $a' \cdot r$. Similarly, the application of a linear map on a vector is also accomplished by the composition of two arrows. The situation is not much different from the component wise linear algebra notation, where both vector–scalar–, matrix–vector– and matrix–matrix–products are denoted by the same symbol '$\cdot$'.

For the rest of the paper, the following interpretation of **Mod**R is useful:

- The objects $A, B \in$ obj(**Mod**R) carry type information only.

- Instead of writing $v \in A$, we write $v : R \to A$. An arrow of this kind should be considered as a data set.

- Arrows $f : A \to B$ represent linear maps. Such functions should be considered as programs taking inn data $v : R \to A$ and returning data $w : R \to B$. If we want to speak of all such arrows as a data set, we will write $f : R \to [A \to B]$. This notation is formalized in the next section.

The same interpretation holds in **Set** if we replace R with T.

4.2 Cartesian Closed Categories

In category theory the usual way to express algebraic equations and specifications is via commutative diagrams. It is often surprising to see that relatively simple diagrams is sufficient to characterize mathematical objects uniquely (i.e. up to an isomorphism). An example of a categorical characterization is the categorical definition of Cartesian products. Our set–theoretic intuitive notion of a cartesian product $A \times B$ is a set consisting of all pairs $\langle a, b\rangle$ such that $a \in A$ and $b \in B$. The categorical definition is:

Definition 3 *Given $A, B \in obj(\mathcal{C})$. The* cartesian product *is an object $A \times B \in obj(\mathcal{C})$ and two arrows $\pi_1 : A \times B \to A$, $\pi_2 : A \times B \to B$ such that for all $U \in obj(\mathcal{C})$ and all arrows $f : U \to A$ and $g : U \to B$ there exist a unique ($\exists!$) arrow $h : U \to A \times B$ such that the diagram*

$$\begin{tikzcd}
 & U \arrow[dl, "f"'] \arrow[d, "\exists! h"] \arrow[dr, "g"] & \\
A & A \times B \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B
\end{tikzcd}$$

commutes. The arrow h is called the product of f and g and is written $h = \langle f, g\rangle$. The arrows π_1 and π_2 are called the projections of the product.

Similarly, we define the *k–fold cartesian product*, $k \geq 0$, as an object $\prod_{i=1^k} A_i = A_1 \times A_2 \times \cdots \times A_k$ with k projections $p_i : \prod_{i=1^k} A_i \to A_i$. The number k is called the *rank* of the product. If T is a cartesian product with rank 0, then T has no projections, and for any other object U there is a unique arrow $\langle\rangle : U \to T$. I.e. T is a terminal object.

Note that if A_i are objects in **Set** and $f_i : T \to A_i$ represent elements $f_i' \in A_i$, then

$$\langle f_1, \ldots, f_k\rangle : T \to \prod_{i=1}^{k} A_i$$

represents the k–tuple $\langle f_1', \ldots, f_k'\rangle \in \prod_i = 1^k A_i$.

In many categories the collection of all arrows between any two objects $A, B \in \mathrm{obj}(\mathcal{C})$ can be turned into an object $[A \to B] \in \mathrm{obj}(\mathcal{C})$. E.g. in **Mod**R , the set of all linear maps between two R–modules A and B is itself a R–module. Such an object is called an *exponential object*, and is denoted $[A \to B]$. Let $\mathcal{C}$ be a category with a terminal object T and $f : A \to B$ an arrow of $\mathcal{C}$. The elements of $[A \to B]$ can be applied as arrows by using an evaluation arrow $\mathrm{eval} : [A \to B] \times A \to B$. The element in $[A \to B]$ which represent f is given as a constant arrow $\lambda f : T \to [A \to B]$. Let $t : A \to T$. Then λf and eval are defined by the following commutative diagram:

$$\begin{tikzcd}
A \arrow[d, "\langle \lambda f \circ t, \mathrm{Id}_A\rangle"'] \arrow[dr, "f"] & \\
{[A \to B] \times A} \arrow[r, "\mathrm{eval}"'] & B
\end{tikzcd}$$

An arrow $f : \prod_{i=1}^{k} A_i \to B$ can be thought of as a function taking k arguments $\langle a_1, \ldots, a_k\rangle$, where $a_i : T \to A_i$. It is frequently the case that we know the value of one of its arguments, say a_r, and we want to find the partially evaluated function, (or curry'ed function) where a_r is fixed and all other inputs are free. Given f, we define the arrow $\lambda_r f : A_r \to \left[\prod_{\substack{i=1\\ i\neq r}}^{k} A_i \to B\right]$ such that the following diagram commutes:

$$
\begin{tikzcd}
\prod\limits_{i=1}^{k} A_i \arrow[d, "{\langle \lambda_r f \circ \pi_r, \langle \pi_1, .., \pi_{r-1}, \pi_{r+1}, .., \pi_k \rangle \rangle}"'] \arrow[drr, "f"] & & \\
{[\prod\limits_{\substack{i=1\\ i\neq r}}^{k} A_i \to B] \times \prod\limits_{\substack{i=1\\ i\neq r}}^{k} A_i} \arrow[rr, "\text{eval}"'] & & B
\end{tikzcd}
$$

Definition 4 *A* Cartesian Closed Category *(CCC) is a category $\mathcal{C}$ such that*

1. *There exist a terminal object $T \in obj(\mathcal{C})$.*
2. *For all objects $A, B \in obj(\mathcal{C})$ there exist an exponential object $[A \to B]$.*
3. *For all collections of k objects $A_1, \ldots, A_k \in obj(\mathcal{C})$ there exist a cartesian product object $\prod_{i=1}^{k} A_i$.*
4. *For all arrows $f : \prod_{i=1}^{k} A_i \to B$ and all $r \in \{1, 2, \ldots, k\}$, there exist an arrow*

$$\lambda_r f : A_i \to \left[\prod_{\substack{i=1\\ i\neq r}}^{k} A_i \to B\right]$$

   defined by the diagram above.

In a CCC we may similarly for any subset $\mathcal{S} \subset \{1, 2, \ldots, k\}$ and any arrow $f : \prod_{i=1^k} A_i \to B$ construct an arrow $\lambda_{\mathcal{S}} f : \prod_{i\in\mathcal{S}} A_i \to [\prod_{i\notin\mathcal{S}} A_i \to B]$.

CCCs are important in computer science, since some functional programming languages like e.g. ML is a CCC. It is essentially the same as a 'typed λ–calculus'.

4.3 Functors

In this section we will introduce some concepts concepts needed to discuss mappings between different categories.

Definition 5 *A functor is a mapping $F : \mathcal{C} \to \mathcal{D}$ from a category $\mathcal{C}$ to a category $\mathcal{D}$, such that objects in $\mathcal{C}$ maps to objects in $\mathcal{D}$ and arrows maps to arrows, and such that compositions and identity arrows are preserved:*

$$
\begin{aligned}
F(Id_A) &= Id_{F(A)} \quad \textit{for all } A \in obj(\mathcal{C}) \\
F(f \circ g) &= F(f) \circ F(g) \quad \textit{for all composable arrows } f \textit{ and } g \textit{ in } arr(\mathcal{C})
\end{aligned}
$$

Example 3 *The category* **Cat** *has as objects categories, and as arrows* functors.

Example 4 *Let U :* **Mod**R $\to$ **Set** *be functor which maps modules to their underlying sets and linear maps to the corresponding set functions. U is called the* underlying set functor. *It is also called a* forgetful functor, *since it forgets that modules have an additional structure besides being sets, i.e. it forgets the $+$ and $\cdot$ rules in* **Mod**R .

Definition 6 *Given two categories $\mathcal{C}$ and $\mathcal{D}$ and two functors $E, F : \mathcal{C} \to \mathcal{D}$. A* natural transformation *from E to F, written $\eta : E \Rightarrow F$, is a family of arrows $A.\eta \in arr(\mathcal{D})$ such that:*

1. *For all objects $A \in \mathcal{C}$ we have an arrow $A.\eta : E(A) \to F(A)$.*
2. *For all arrows $f \in arr(\mathcal{C})$, such that $f : A \to B$, the diagram below commutes:*

$$\begin{tikzcd}
E(A) \arrow[r, "A.\eta"] \arrow[d, "E(f)"'] \arrow[dr, phantom, "\mathcal{D}"] & F(A) \arrow[d, "F(f)"] \\
E(B) \arrow[r, "\eta.B"] & F(B)
\end{tikzcd}$$

Definition 7 *Given two categories $\mathcal{A}$ and $\mathcal{B}$ and a functor $U : \mathcal{B} \to \mathcal{A}$. Suppose that there exist a functor $F : \mathcal{A} \to \mathcal{B}$ and a natural transformation $\eta : Id_{\mathcal{A}} \Rightarrow F{\circ}U$, such that for any object $A \in obj(\mathcal{A})$, any object $B \in \mathcal{B}$ and any arrow $f \in arr(\mathcal{A})$, $f : A \to U(B)$ there exist a unique arrow $g \in arr(\mathcal{B})$, $g : F(A) \to B$ such that $f = U(g){\circ}A.\eta$. Then we say that F is the left adjoint of U, or that U is the right adjoint of F, or that (F, η, U) is an adjunction. We call η the* unit *of the adjunction.*

$$\begin{tikzcd}
 & \mathcal{A} \arrow[rr, dashed, bend left=10, "F"] & & \mathcal{B} \arrow[ll, dashed, bend left=10, "U"] \\
A \arrow[r, "A.\eta"] \arrow[dr, "f"'] & U{\circ}F(A) \arrow[d, "U(g)"] & & F(A) \arrow[d, "\exists! g"] \\
 & U(B) & & B
\end{tikzcd}$$

If U has a left adjoint F, then F and η are (essentially) uniquely defined.

Recall that we defined a free module to be a module with a basis. Here is the categorical definition:

Definition 8 *Let U : $\mathbf{Mod}R \to \mathbf{Set}$ be the underlying set functor. Then the left adjoint F is called the* free functor*. Objects of the type $F(A) \in \mathbf{Mod}R$ are called* free modules*.*

The informal interpretation of this definition is as follows: Suppose A is the finite set $\{1, 2, \ldots, n\}$. Then $F(A)$ should be thought of as R^n, and also $U{\circ}F(A)$ as R^n, but interpreted as a set. We should think of $A.\eta$ as a function taking i to e_i, the i'th basis vector in R^n, and the definition says that whenever we specify a linear map on the basis vectors, it may be uniquely extended to a linear map on the whole space.

Now we can also give a formal definition of tensor product spaces.

Definition 9 *The category of multilinear maps, $\mathbf{Mlin}R$, is the category where the objects are cartesian products of modules $A_1{\times}\cdots{\times}A_k$ and the arrows are multilinear maps: $f : A_1{\times}\cdots{\times}A_k \to B$.*

Since all modules are the rank–1 cartesian product of itself, and any linear map is also multilinear, we may define the functor I : $\mathbf{Mod}\mathrm{R} \to \mathbf{Mlin}\mathrm{R}$ as the inclusion of $\mathbf{Mod}\mathrm{R}$ in $\mathbf{Mlin}\mathrm{R}$.

Definition 10 *Let T be the left adjoint of I and τ the unit of the adjunction. If $A = A_1{\times}\cdots{\times}A_k \in obj(\mathbf{Mlin}R)$ then $T(A) = A_1{\otimes}\cdots{\otimes}A_k \in obj(\mathbf{Mod}R)$ is called a* tensor product space*.*

If $u_i : R \to A_i$, then

$$u_1 \otimes \cdots \otimes u_k = T(\langle \mathrm{I}\,(u_1), \ldots, \mathrm{I}\,(u_k)\rangle) : R \to A_1 \otimes \cdots \otimes A_k$$

is the tensor product of the vectors u_i.

The formal treatment of symmetries is very similar. There are two basic functors for creating symmetric spaces: The wedge functor $\wedge$ and the symmetric functor S. A multilinear function $f : A \times A \times \cdots \times A \to B$ is called *symmetric* if it has the property that

$$f(u_1, \ldots, u_i, \ldots, u_j, \ldots u_k) = f(u_1, \ldots, u_j, \ldots, u_i, \ldots u_k) \text{ for all pair of indexes } i \text{ and } j$$

and it is called *alternating* if

$$f(u_1, \ldots, u_i, \ldots, u_j, \ldots u_k) = -f(u_1, \ldots, u_j, \ldots, u_i, \ldots u_k) \text{ for all pair of indexes } i \text{ and } j.$$

They are defined as the left adjoints of the functor including **Mod**R in the categories of symmetric and alternating multilinear functions.

Example 5 *Hookes tensor Λ can be defined as an arrow $\Lambda : R \to (U \otimes_s U) \otimes_s (U \otimes_s U)$.*

Wedge products are very important in Cartans theory of exterior derivation and differential forms, which is the differential geometric treatment of Stokes and Greens type theorems etc.

5 Programming with diagrams

5.1 The Tuple category

To express the opetations needed in typical diagram constructions, we have formalized a category called **Tuple**. This category is capturing the essential properties of **Mlin**R , and is in addition extended with the operations of a CCC. The formal definition is as follows:

Definition 11 *Definition of the category* **Tuple***:*

- **Objects:** *All k–fold cartesian product of modules, $k \geq 0$. T denotes the empty tuple, which is terminal. All objects in* **Mod***R are injected as rank–1 objects, and for these we will use the same symbol as in* **Mod***R . We have a mechanism for computing type information, e.g. from $[A \otimes B \to C]$, we can find A, B and C. The particular mecanism for this will not be discussed, since it has no theoretical importance.*
- **Arrows:**
  - *All linear and multilinear maps.*
  - *For every set of arrows $\{f_i : B \to A_i\}_{i=1}^k$ an arrow $\langle f_1, f_2, \ldots, f_k\rangle : B \to \prod_{i=1}^k A_i$.*
  - *For every object $A = \prod_{i=1}^k A_i$ a set of projections $A.\pi_i : A \to A_i$.*
  - *For every arrow $f : A = \prod_{i=1}^k A_i \to B$ and every subset $\mathcal{S} \subset \{1, 2, \ldots, k\}$ an arrow $\lambda_{\mathcal{S}} f : \prod_{i \in \mathcal{S}} A_i \to [\prod_{i \notin \mathcal{S}} A_i \to B]$.*
  - *For every object $[A \to B] \times B$ an arrow $eval : [A \to B] \times B \to B$.*
  - *For every object A an arrow $A.\tau : A \to$ $Tensor(A)$, the unit of adjunction of the tensor product functor (see below).*
  - *For every object $A = \prod_{i=1}^k A_i$, where $A_1 = \cdots = A_k$ an arrow $A.\sigma : A \to$ $Symm(A)$ and an arrow $A.\omega : A \to$ $Wedge(A)$, the unit of adjunction for the symmetric functor and the wedge functor.*
- **Functors:**

– *Between different categories:*

  ∗ *The inclusion functor:*
  $$I\ : \mathbf{Mod}R\ \rightarrow \mathbf{Tuple}$$

  ∗ *The left adjoint of I :*
  $$T : \mathbf{Tuple} \rightarrow \mathbf{Mod}R$$

  ∗ *The left adjoint of I within the subcategory of symmetric multilinear maps:*
  $$S : \mathbf{Tuple} \rightarrow \mathbf{Mod}R$$

  ∗ *The left adjoint of I within the subcategory of alternating multilinear maps:*
  $$\wedge : \mathbf{Tuple} \rightarrow \mathbf{Mod}R$$

– *Internal functors: For convenience we introduce the internal functors*

$$\begin{aligned} Tensor &= I \circ T : \mathbf{Tuple} \rightarrow \mathbf{Tuple} \\ Symm &= I \circ S : \mathbf{Tuple} \rightarrow \mathbf{Tuple} \\ Wedge &= I \circ \wedge : \mathbf{Tuple} \rightarrow \mathbf{Tuple} \end{aligned}$$

- **Note:** *In* **Tuple** *there is no way of adding arrows, or multiplying arrows with ring elements. To accomplish this, we must move to* **Mod**R .

The following examples are more ment as an illustration of the technique, rather than as practical useful procedures. The program codes are simplified to enhance clarity. We use comments to indicate the type of the arrows created.

Example 6 *Grouping components. We want to compute the isomorphism $t : U{\otimes}V{\otimes}W \rightarrow (U{\otimes}V){\otimes}W$. It is specified by the following diagram:*

$$\begin{tikzcd} U \times V \times W \arrow[r, "\tau"] \arrow[d, "\langle \tau\circ\langle \pi_1, \pi_2\rangle, \pi_3\rangle"'] & U{\otimes}V{\otimes}W \arrow[d, "!t"] \\ U{\otimes}V \times W \arrow[r, "\tau"'] & (U{\otimes}V){\otimes}W \end{tikzcd}$$

And it is programmed by the following code:

```
function group(T)
   // input:  T = U⊗V⊗W :  type information
   // output:  t : U⊗V⊗W → (U⊗V)⊗W  :  isomorphism
   //
   // from T compute the following type info:
   A = U×V×W;  B = U×V;
   t =  Tensor(⟨B.τ∘⟨A.π1, A.π2⟩, A.π3⟩)  ;
return(t);
```

Example 7 *The ungroup map is the inverse of the above $t : (U\otimes V)\otimes W \to U\otimes V\otimes W$. The diagram for this operation is more difficult to draw. It looks like:*

$$
\begin{tikzcd}
U\times V\times W \arrow[r, equal] \arrow[ddrr, "A.\tau"'] & U\times V\times W \arrow[rr, "\langle \tau\circ\langle \pi_1, \pi_2\rangle, \pi_3\rangle"] \arrow[dr, "\langle \lambda_{\{1,2\}}(A.\tau)\circ\langle \pi_1, \pi_2\rangle, \pi_3\rangle"] & & (U\otimes V)\times W \arrow[r, "\tau"] \arrow[dl, "\langle !t_3, \pi_2\rangle"'] & (U\otimes V)\otimes W \arrow[ddll, "!t"] \\
& & {[W\to U\otimes V\otimes W]\times W} \arrow[d, "eval"] & & \\
& & U\otimes V\otimes W & &
\end{tikzcd}
$$

The idea is to work from left to right by completing the arrows indicated by the exclamation marks.

```
function ungroup(T)
   // input:  T = (U⊗V)⊗W  :  type information
   // output:  t : (U⊗V)⊗W → U⊗V⊗W  :  isomorphism
   //
   // from T compute the following type info:
   A = U×V×W;  B = (U⊗V)×W;
   t2 = λ_{1,2}(A.τ);  //  :   U×V → [W → U⊗V⊗W]
   t3 =  Tensor(t2);  //  :   U⊗V → [W → U⊗V⊗W]
   t4 = eval∘⟨t3, B.π2⟩  //  :   (U⊗V)×W → U⊗V⊗W
   t =  Tensor(t4);  //  :   (U⊗V)⊗W → U⊗V⊗W
return(t);
```

Example 8 *This third example is a mapping*

$$ip : [U1\otimes U2 \to W]\times U1 \to [U2 \to W] \ .$$

This is a partial evaluation, where we put an argument into the first 'slot' of a tensor. It is called an interior product, and is programmed as:

```
function intProd(T)
   // Input:
   //      T = A×B, where
   //           A = [U1⊗U2 → W]
   //           B = U1
   // Output:
   //      ip ∈ arr(Tuple) such that
   //      ip : A×B → [U2 → W]
   //
   // from T compute:   D = [U1×U2 → W]×U1×U2;
   t = λ_{1,2}(D.eval);  //  t : [U1×U2 → W]×U1 → [U2 → W]
   t2 =  Tensor(t);  //  t2 : [U1⊗U2 → W]⊗U1 → [U2 → W]
   // t2 is tensorized ‘too much’.  We must ‘untensorize’ the second product.
   ip = t2∘T.τ;
return(ip);
```

Note 1 *This way of writing programs is totally coordinate independent, and is not depending on any particular property of the modules. There are only a few underlying routines that must* *be implemented with coordinates. It is therefore easy to debug the program. We are guarenteed that all expressions written by the users of the system are independent of coordinates, and since everything is strongly typed we can detect errors that would otherwise pass unnoticed.*

5.2 Implementation

So far we have said nothing about how to implement the operations in the **Tuple** category. One possibility is to build a parse structure, and to compute expressions when they are complete (i.e. when we have a complete arrow from T to some object A. This approach is not particularly simple. A special problem is posed by the Tensor functor, since τ is not invertible. The computation of this functor *must* in one way or another depend on some coordinate representation.

$$\begin{tikzcd} A \times B \arrow[r, "\tau"] \arrow[dr, "f"'] & A \otimes B \arrow[d, "!g = \text{Tensor}(f)"] \\ & C \end{tikzcd}$$

In the case where all the involved modules are finite dimensional free modules, there exist a dramatically much simpler implementation. We construct a simple category of indexes **Index**, (which consist of tuples of integers) and a forgetful functor U : **Tuple** $\to$ **Index**. The left adjoint of U is a free functor F : **Index** $\to$ **Tuple**. Our goal is to do this in a way such that **Index** can be equipped with the same functions as **Tuple** and such that:

$$\begin{array}{rcl}
F(\langle f, g\rangle) & = & \langle F(f), F(g)\rangle \\
F(A.\pi_i) & = & F(A).\pi_i \\
F(\lambda_i f) & = & \lambda_i F(f) \\
F(A.\text{eval}) & = & F(A).\text{eval} \\
F(A.\tau) & = & F(A).\tau \\
F(A.\sigma) & = & F(A).\sigma \\
F(A.\omega) & = & F(A).\omega \\
F \circ \text{Tensor} & = & \text{Tensor} \circ F \\
F \circ \text{Symm} & = & \text{Symm} \circ F \\
F \circ \text{Wedge} & = & \text{Wedge} \circ F
\end{array}$$

To achieve this, U cannot be as forgetful as the underlying set functor described in section 4.3. To preserve eval it must not forget 0, and to preserve wedge products it must not forget the difference between $+1$ and -1. The solution is as follows:

Definition 12 *Definition of* **Index** *category.*

1. *Basic objects (which correspond to all rank–1 objects in* **Tuple***) are non–negative integers. The integer corresponds to the dimension of the space.*

2. *Basic arrows between two objects $A = m$ and $B = n$, $f : A \to B$ consist of all set functions $f : \{-m, \ldots, -1, 0, 1, \ldots, m\} \to \{-n, \ldots, -1, 0, 1, \ldots, n\}$ subject to the condition that $f(-i) = -f(i)$. An arrow $f : \langle\rangle \to i$ corresponds to the i'th basis vector, and $-f$ to the negative of this vector.*

3. *The set of objects and arrows are extended with cartesian products in the standard way.*

Now, we use the equations above to introduce all the functions in **Index**. The only thing we have freedom to choose is the forgetful functor U. This corresponds to choosing appropriate basis vectors for the derived spaces in **Mod**R such as e.g. $A \otimes B$ and $[A \to B]$. The solution to this corresponds to the standard choices of basis vectors in tensor calculus. E.g. if A has basis vectors $e_i = F(i)$ then $[A \to R]$ is equipped by the dual basis $e^j = F(j)$ such that $eval \circ \langle e_i, e^j \rangle = \delta_{i,j}$. If this condition is not met, the eval function cannot be pulled down to **Index**. Similarly, for tensor products $A \otimes B$ we must choose as basis the vectors that are injected from τ, i.e. the vectors $(A \times B).\tau \circ \langle e_i, e_j \rangle$. In **Index** we have some freedom to choose the function τ. This just corresponds to a particular ordering of the basisvectors in $A \otimes B$.

In the case of symmetries, the arrows σ and ω of **Index** must belong to the subcategory of symmetric (resp. alternating) functions in **Index**. E.g. if A is an n–dimensional space, then $A \otimes_s A$ is $n(n+1)/2$ dimensional, and the arrow $\sigma \in \mathrm{arr}(\mathbf{Index})$, $\sigma : A \times A \to A \otimes_s A$ returns a number in this range. It can be chosen freely as long as $\sigma \circ \langle i, j \rangle = \sigma \circ \langle j, i \rangle$.

When this construction is finished, we can pull down all the programs of the previous section to the category **Index**, and we have created our 'index–wrestler'. A result t returned by such an index wrestler is a function computing addresses. From a set of fetch addresses, it computes the store addresses. To construct the corresponding arrow in **Tuple**, we apply the free functor $F(t)$. This is just implemented as a function taking instructions from t about where to fetch and where to store numbers!

Note that the data of symmetric tensors is not duplicated. I.e. t knows about all the symmetries, and about how to store the data in a compact form.

References

[1] Abraham R., Marsden J.E.: *Foundations of Mechanics*, Addison–Wesley, 1978.

[2] Abraham R., Marsden J.E., Ratiu T.: *Manifolds, Tensor Analysis, and Applications*, Springer–Verlag 1980.

[3] Barr M., Wells C.: *Category Theory for Computing Science*, Prentice Hall 1989.

[4] Borceux F.: *Handbook of Categorical Algebra*, Cambridge univesity press, 1994.

[5] Bott R., Tu L.W.: *Differential Forms in Algebraic Topology*, Springer GTM no. 82, 1982.

[6] Brendon G. E.: *Sheaf Theory*, McGraw–Hill, 1967.

[7] Eilenberg S., Mac Lane S.: *General Theory of Natural Equivalenecs*, Transactions of the AMS 58, 1945, 231–294

[8] Mac Lane S.: *Categories for the Working Mathematician*, Springer–Verlag, 1971.

[9] Misner C., Thorne K., Wheeler J.A.: *Gravitation*, W.H. Freeman, 1973.

[10] Haveraaen M., Madsen V., Munthe–Kaas H.: *Algebraic Programming Technology for Partial Differential Equations*, Proceedings of Norsk Informatikk Konferanse (NIK), Tapir Trondheim Norway, 1992.

[11] Husemoller D.: *Fiber Bundles*, Springer GTM no. 20, 1975.

[12] Marsden J.E., Hughes T.J.R.: *Mathematical Foundations of Elasticity*, Prentice–Hall, 1983.

[13] Roman S.: *Advanced Linear Algebra*, Springer GTM no. 135, 1992.

[14] Walters R.F.C.: *Categories and Computer Science*, Cambridge University Press, 1991.

[15] Warner F.W.: *Foundations of Differential Manifolds and Lie Groups*, Springer GTM no. 94, 1983.