MUNTHE-KAAS H., HAVERAAEN M.

Coordinate Free Numerics — Closing the gap between ‘Pure’ and ‘Applied’ mathematics?[1]

The theory of differential equations have diverged in two different directions in our century; the applied mathematical coordinate based presentation which has its origin in hand calculations, and the pure mathematical presentation based on coordinate free formulations, commutative diagrams and global analysis.

Through various examples, we show that much can be gained by employing ‘abstract’ coordinate free formulations also in scientific/numerical computing. The origin of this line of work, was an attempt to design a software system for numerical solution of tensor field equations on parallel computers by using formal methods from pure computer science [2]. It was discovered that the pure abstract mathematical definitions was better suited for software design than the concrete coordinate based definitions, and that modern computer languages enables us to express and program various numerical algorithms directly within a coordinate free language. Later [3] we discovered very useful computational techniques for manipulating tensor indexes which are simple to program within the abstract framework, but cannot be conveniently expressed within a coordinate based formulation. Recently [4], we have shown that the Butcher theory for analyzing Runge–Kutta methods may be replaced by a coordinate free analysis based on Lie series and Lie groups. Through its geometric formulation and structural simplicity, this approach is interesting for investigating geometric ODE integrators (e.g. symplectic ODE methods).

Our thesis is the following: **Modern computer languages enables us to use abstract formulations from pure mathematics in computational mathematics. Computational mathematics will gain from this, and it may eventually lead to a narrowing of the gap between pure and applied mathematics.**

1. ‘Abstract’ vs. ‘Concrete’; ‘What’ vs. ‘How’

A basis for modern algebraic programming technology and Object Oriented software design is the recognition that *“Computer programming is what + how”*. I.e. we should keep in mind the difference between an *abstract specification* of tasks to be done and the actual algorithms/data structures used in a *concrete implementation.* The abstract specification defines the external interfaces of a software module, and the concrete implementation is hidden within the module and not accessible to the rest of the world. The main benefits are modularity (change algorithms/data structures within a given module without doing anything outside), and it is also crucial for successful debugging and verification.

Although this programming model has been very successful in computer science tasks (graphic packages, discrete event simulations, ...), it is not at all evident that the same approach is useful or even possible in numerical computing. At the very hart of the problem is the fact that the applied mathematical language was developed for the use in hand calculations, and is not suited for describing ‘what’. E.g. take the common definition of the laplacian: $\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$. This is really a ‘how’ description; ‘how to compute ∇^2 in a cartesian coordinate system’. Similar problems occur if we, within the classical language, try to answer questions such as ‘what is a tensor’, ‘what is the domain of a differential equation’, ‘what is a Runge–Kutta method’ etc.

In pure mathematics the main goal has been to study structures rather than doing actual computations. Hence one have been forced to develop languages for discussing ‘what’ independently of ‘how’. This is the core of coordinate free formulations, and is also perhaps a reason for the current division between pure and applied math.

2. The SOPHUS system for solving tensor field equations

In the SOPHUS project [2,3], we have developed a software system for doing tensor field computations, which is almost entirely depending on coordinate free formulations. The basis is differential manifolds as definitions of domains, categorical diagrams in multilinear algebra as definitions of tensors and geometrical definitions of differential equations via Lie derivations and covariant derivations. The core of the SOPHUS system is a module implementing real valued functions over a domain. Although this module is implemented by using a given discretization (e.g. finite

[1]This work is sponsored by INTEL corp. through the ‘INTEL University Grants’.

To appear in proc. of ICIAM -95, Zeitschrift fuer Angewandte Mathematik und Mechanik, Akademie Verlag, Berlin, 1996.

differences or spectral methods), it is visible to the rest of the system only through a set of continuous operations; we may add or multiply two functions, take Lie derivatives of functions etc. The rest of the system is independent of the chosen discretization, and most of it is also independent of particular coordinate systems.

This leads to significant gains in terms of software modularity, portability between various parallel computers and flexibility w.r.t. changing discretization techniques and numerical algorithms. The most surprising discovery was perhaps a very useful algorithm for doing manipulations with tensor indexes [3] which we will briefly describe below.

To motivate this algorithm we will look at some examples from componentwise tensor analysis. Let g_{ij} denote the components of the metric tensor. It appears in many different expressions, e.g. inner products $r = y^i g_{ij} x^j$, contractions $r = g_{ij} a^{ij}$ and index lowering $b_i{}^k = g_{ij} a^{jk}$. All these computations are performed by fetching different sets of numbers, multiplying, adding and storing into components of the result. The order of which the numbers should be fetched differ in these examples, and the information about this must be read out of the indexes (sum over repeated indexes). In many cases tensors possess many symmetries, and it is important to only store and compute different components. Consider Hookes law in linear elasticity: $\sigma^{ij} = \Lambda_{ijkl}\epsilon^{kl}$, where σ and ϵ are the symmetric stress and strain tensors, and Λ is Hookes tensor, possessing the symmetries $\Lambda ijkl = \Lambda jikl = \Lambda klij$ which reduce the number of different components from 81 to 21. In the case where a compact storage of σ, η and Λ is used, it is not evident where to fetch and store the components. In designing a general system for tensor computations, we are faced with the problems of how to specify symmetries and various uses of the same object, how to compute fetch and store addresses and how to take advantage of all symmetries, both to reduce storage and computations. Our algorithm solves all these problems in a relatively simple manner, but it cannot be expressed within the componentwise notation.

In the (category theoretical) coordinate free language, tensorial constructions are expressed in terms of commutative diagrams, i.e. diagrams (graphs) where the nodes represent linear spaces and the arrows are multilinear maps. The various uses of the metric tensor in the above examples are represented by *different* arrows, and we can formally express how to obtain one arrow from another. This corresponds to preparing the data set in the metric tensor g for different uses; e.g. for using in an inner product, a contraction or index lowering. The process can be summarized as follows:

1. The tensor operations are expressed in terms of commutative diagrams. The various diagram operations are expressed as computer programs.
2. The diagrams are pulled down with a functor to a category of indexes. In this category the nodes are indexes and the arrows represent programs which compute the index transformations involved in a computation (where to fetch and store components).
3. The arrows are lifted from the index category to the category of linear maps with a free functor. This step corresponds to actually performing the tensor computations, given information about where to fetch and store.

3. Conclusions

We have seen an example of a problem which at best requires a very large and complex program to be handled generally within the classical formulation, but where the solution based on the abstract formulation is surprisingly simple. We believe that modern programming technology gradually will contribute in narrowing the gap between pure and applied mathematics, and that this will influence computational sciences in a positive direction.

4. References

1 Borceux F. : Handbook of Categorical Algebra. Cambridge university press, 1994.

2 Haveraaen, M., Madsen, V., Munthe-Kaas, H.: Algebraic Programming Technology for Partial Differential Equations. Proceedings of Norsk Informatikk Konferanse (NIK), Tapir, Trondheim, Norway 1993.

3 Munthe-Kaas H., Haveraaen M.: Coordinate Free Numerics I : How to avoid index wrestling in tensor computations. Report no. 101, Department of Informatics, University of Bergen, Norway 1995.

4 Munthe-Kaas, H.: Lie–Butcher theory for Runge–Kutta methods. Report no. 105, Department of Informatics, Univ. of Bergen, March 1995.

5 Roman, S.: Advanced Linear Algebra. Springer GTM no. 135, 1992.

Addresses: Dr. H. Munthe-Kaas, M.Haveraaen Dept. of Informatics, Univ. of Bergen, N-5020 Bergen, Norway.
Email: Hans.Munthe-Kaas@ii.uib.no, Magne.Haveraaen@ii.uib.no