On the Role of Mathematical Abstractions for Scientific Computing*

Krister Åhlander Magne Haveraaen Hans Munthe-Kaas

July 12, 2000

Abstract

A distinguished feature of scientific computing is the necessity to design software abstractions for approximations. The approximations are themselves abstractions of mathematical models, which also are abstractions.

In this paper, the relation between different mathematical abstraction levels and scientific computing software is discussed, in particular with respect to the simulation of partial differential equations (PDEs).

It is found that software based on continuous abstractions have more chances of being modular, than software based on discrete approximations of the continuous abstractions. Moreover, it is stated that coordinate-free abstractions are a solid foundation for the simulation of PDEs.

1 Introduction

Applications in the area of scientific computing span a wide range of problem domains, and each domain exhibits its own challenges to be addressed. However, one distinctive feature of scientific computing applications is the need to deal with approximations of continuous abstractions. The obvious example is that the infinite number of reals on any interval must be approximated on a computer, normally by floating point numbers of appropriate precision. But even this basic example shows that scientific computing is a science where nothing is certain, since it is also possible to use, say, integers to represent real values, as is often done in image processing. Another example of a continuous approximation is the abstraction of an angle. In order to become a useful component, this abstraction must recognize that different real numbers, say π and 3π, may represent the same angle, and also that different units may be used. These facts are of a different nature than the fact that the same real number may be approximated

*Draft paper submitted to the IFIP WG 2.5 Working Conference on Software Architecture for Scientific Computing Applications, 2-6 October 2000 in Ottawa, Canada. Research funded via a grant from the Norwegian research council. Email of corresponding author: krister@ii.uib.no

by different discrete representations. The understanding of approximations and the abstraction of approximations is a key aspect for scientific computing, in order to obtain modular software architectures.

In this paper, we promote the philosophy that the design of applications for scientific computing shall be built firstly upon continuous abstractions, where appropriate, and secondly on approximation abstractions. This is nicely illustrated by the angle abstraction having some continuous features, independent of a particular choice of approximation for real numbers. Moreover, the angle abstraction shall be general with respect to choice of units. This situation is similar to the recognition that physical quantities, for instance velocity, shall not depend on certain units or a certain choice of coordinates. Consequently we identify two different mathematical abstraction levels, a continuous abstraction level and a coordinate-free abstraction levels. Both levels can be used as input for software architecture design. The aim of this paper is to discuss the role mathematical abstraction levels have concerning the modularity of the software architecture, particularly with respect to simulations of partial differential equations (PDEs). The problem domain of PDE solvers is an area within the main stream of scientific computing, and it includes the need to evaluate approximate solutions of mathematical models, a demand for high performance, and the necessity to utilize physically motivated simplifications where possible. We find that a design which is based upon continuous abstractions can be modular in many aspects, but we emphasize that a software architecture based on coordinate-free abstractions have additional flexibility.

When designing scientific computing software, there are many different aspects to take into account. Today, there seems to be a trend towards problem solving environments, interactive environments, and distributed applications. Since we find mathematical abstractions, in particular when based on coordinate-free formulations, are beneficial for designing the core of PDE solvers, we believe that it would be of interest to study the role of mathematical abstractions in other problem areas. We think that specifications of mathematical abstractions might have an impact also in problem areas like visualisation and communication between distributed components.

2 Motivation

As a motivating example, we take the wave equation for an elastic medium. It is relevant as an industrial application, for instance when modeling oil reservoirs. It is a standard equation from mathematical physics, and may be found in any text book on the subject, for instance [8]. In its most compact version, the coordinate-free form, it may be stated as:

$$\begin{aligned} \rho \frac{\partial^2 u}{\partial t^2} &= \nabla \cdot \sigma + f(t) \\ \sigma &= \Lambda(e) \\ e &= \mathcal{L}_u g. \end{aligned} \tag{1}$$

Here, ρ is a scalar field, u is the displacement vector to be simulated, f is a time-dependent forcing function, and σ, e and Λ are tensors of order two (σ and e) and four (Λ). A tensor of order n needs n indices. The equations also involve derivation with respect to time, $\frac{\partial}{\partial t}$, and the derivations $\nabla\cdot$ and $\mathcal{L}_u$. In order to find suitable abstractions for design, we may simplify the equations assuming Cartesian coordinates, and the equations may be rewritten in component form as:

$$\begin{aligned} \rho\frac{\partial^2 u_i}{\partial t^2} &= \sum_j \frac{\partial \sigma_{ij}}{\partial x_j} + f_i(t) \\ \sigma_{ij} &= \sum_{kl} \Lambda_{ijkl} e_{kl} \\ e_{ij} &= \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right), \end{aligned} \tag{2}$$

where the component indices i and j and the summation indices k and l vary over the number of space dimensions.

It is not difficult to implement a program simulating a discretized version of these equations in a language that support simple array or matrix abstractions. The continuous abstraction u may for instance be discretized on an equidistant square grid, and the continuous derivatives may be approximated with finite differences. But, some of the questions that we want to pose are: What happens when the mathematical model changes? When the numerical method becomes more sophisticated? When the geometry gets more complicated? In these situations we would probably have little or no use of the previous, quick, implementation of the simple model. If a more modular application is wanted, we argue that the software architecture should be designed according to the levels of the underlying mathematical abstractions.

3 Different mathematical abstraction levels

In order to create modular software, it is important to divide the software into different layers and partitions. For scientific software, that deals with mathematical abstractions, it is natural to divide the software architecture into layers that correspond to the mathematical abstractions. For our problem domain of PDE solvers, the following levels may be identified:

Manifold abstractions The abstraction of the computational domain corresponds to a mathematical manifold. In software, this is normally represented as a `Grid`. Grids may be structured or unstructured, adaptive, curvilinear, or overlapping. It is very common that software projects in this field are very tightly coupled to the choice of manifold abstraction. Consequently, it is often difficult to combine abstractions from different projects. Within the same project, there are many examples of modularity between manifold abstractions. For instance, it is common with a serial and a parallel version of the same manifold abstraction.

Field abstractions On the field abstraction level we find abstractions that are based on equation (2) in our motivating example, for instance scalar fields, vector fields, and partial derivatives with respect to space coordinates. In

software, common names for the key abstraction on this level are `Grid Function` (see, e.g., [2, 11]) or `Field` (see., e.g., [3, 9]).

Coordinate-free abstractions On the coordinate-free abstraction level we find abstractions that are based on equation (1). Here, we also take into account the dependence of the coordinate system. In current software projects, apart from Sophus [5, 6], this level seems to be less developed.

Other levels of abstraction We may also identify higher levels of abstraction, for instance for time discretisations, and for the equations themselves. In [1], a framework design which addresses the modularity of these abstractions is proposed. These issues have also, to some extent, been treated with respect to ordinary differential equations, see e.g., [4].

In the remainder of this paper we will concentrate on a the field and the coordinate-free abstraction levels.

4 Field studies

The field concept is central for PDE codes. To illustrate the importance of continous abstractions, we discuss various issues regarding the representations of fields.

4.1 Scalar fields

Consider a scalar field $u = u(x_1, x_2), x \in \mathcal{M} \subset \Re^2$. Depending on context, the following representations may be plausible:

1. Cartesian coordinates $u = u(x, y)$ may be good for simple geometries.
2. Cylindrical coordinates $u = u(r, \theta)$ may be used for axi-symmetric geometries.
3. Curvi-linear coordinates where several mappings are used to cover the domain: $u = u(\phi_l(r, s)),\ l = 1 \ldots L$. These concepts can be used for instance when to define differentiable manifolds, see for instance [10].
4. Infinite linear combinations of basis functions: $u = \sum_{l=1}^{\infty} c_l \Psi_l$.

This list is not exhaustive, but it illustrates that the continuous abstraction level leads to different abstractions. Table 1 suggests corresponding discrete data structures. One may observe two mechanisms of discretisation for fields, either by discretising the underlying manifold, i.e. rounding, or by terminating an infinite series, i.e. truncation. In order to find modular software abstractions, it is obviuos that different implementations of the same discrete representation shall be exchangeable. But, already if we study the continuous abstraction level, it is clear that the first three representations can be summarized as $u = u(p), p \in \mathcal{M}$. The manifold abstraction is then recognized as a separate module,

	Continuous	Discrete
1.	$u(\mathbf{x}) = u(x, y)$	$u_{ij} = u(x_i, y_j)$
2.	$u(\mathbf{x}) = u(r, \theta)$	$u_{ij} = u(r_i, \theta_j)$
3.	$u(\mathbf{x}) = u(\phi_l(r, s))$	$u_{ijl} = u(\phi_l(r_i, s_i))$
4.	$u(\mathbf{x}) = \sum_{l=1}^{\infty} c_l \Psi_l(\mathbf{x})$	$u(\mathbf{x}) = \sum_{l=1}^{N} c_l \Psi_l(\mathbf{x})$

Table 1: Different continous abstractions lead to different discrete abstractions.

and the exchangability between different field representations can be increased. The discrete version is then $u = u(p), p \in \mathcal{M}_h$, where $\mathcal{M}_h$ is a discretisation of $\mathcal{M}$. The relation between the fourth example and the three others are not as obviuos, particularly on the discrete level. Here, the emphasis is on the basis functions Ψ_l, and only indirectly on the manifold. Still, the description $u = u(p), p \in \mathcal{M}_h$ is valid. It may therefore be taken as a definition of the scalar field, independent of the representation.

4.2 Tensor fields

Real scalar fields relate a real value to every point on a manifold. Similarly, complex scalar fields relate a complex value to every point on a manifold. Therefore, using a C++ template notation, we may express a scalar field as an abstraction over a type, `Field<T>`. In this section, we will discuss how to extend scalar fields to vector fields, or, more generally, tensor fields.

First, we consider a two-dimensional vector in a point p: $v = (v_1, v_2)$. The vector may be real-valued or complex-valued, and a vector may then be constructed over a type: `Vector<T>`. Note that the discrete components of a vector is intrinsic in the mathematical abstraction, and has nothing to do with any approximation. A generalisation of a vector is a tensor, where several indices are used to index a tensor component, as in equation (2).

Now, consider a vector field on a two-dimensional manifold $\mathcal{M}$,

$$\mathbf{u}(x_1, x_2) = (u_1(x_1, x_2), u_2(x_1, x_2)).$$

It can be regarded as an array with three indices. The first is discrete and picks out the component, and the other two "indices" are the continuous coordinates. A discrete representation, based on for instance Cartesian coordinates, might be a three-dimensional array structure: $u_{ijl} = u_l(x_i, y_j)$. It seems as if we have two equivalent ways to construct vector or tensor fields:

```
Field<Tensor<Scalar>>,                    (3)
```

and

```
Tensor<Field<Scalar>>.                    (4)
```

In other words, we can either construct a `Tensor Field` as a `Field` over a `Tensor` or as a `Tensor` over a `Scalar Field`. If we only base our design on the abstraction levels of field representations, both these choices seem plausible.

But if we also consider derivatives, there are important differences. Differentiation of scalar fields is quite straight forward. For vector fields, it is more complicated, because we can not subtract two vectors in different points from each other. However, if the geometry of the manifold is taken into account, vector differentiation may be formulated in terms of the geometry, and in terms of partial derivatives of the components, the scalar fields. Construct (3) yields therefore an awkward implementation. Construct (4) is more suitable, since implementation of vector differentiation can now be stated in terms of partial derivatives. Tensor and vector differentiation are important tools, in order to formulate PDEs in a coordinate-free form, as equation (1). See [7] for more discussions on these aspects.

The implication for software modularity is best illustrated with an example. In many physical situations, it is essential to utilize symmetries in order to reduce the problem to something simpler. For instance, it may be appropriate to use cylindrical coordinates for the wave equation. In this case, equation (2) must be rewritten into cylindrical coordinates. Equation (1), on the other hand, remains intact. Thus, if the software supports coordinate-free abstractions, a change of coordinates may be done without affecting the module that defines the equations.

5 Concluding remarks

The role of mathematical abstractions have been discussed, and we argue that the correct treatment of these are important in order to design modular software architectures for scientific computing.

Particularly, we investigate the role of mathematical abstractions in the context of simulating partial differential equations, that is PDE solvers. We identify several levels of mathematical abstractions and we point out that the modularity of the software depends upon which levels are chosen as the input for the design. For our problem domain, we argue that coordinate-free abstractions shall be used.

We find that mathematical abstractions are important for designing PDE solvers. It would also be of interest to investigate the role of mathematical abstractions in other contexts. For example, we believe that the usage of mathematical abstractions can be pursued further when designing more user-friendly scientific software, such as problem solving environments, or distributed applications.

References

[1] K. Åhlander. *An Object-Oriented Framework for PDE Solvers.* PhD thesis, Uppsala University, Dept. of Scientific Computing, Uppsala, Sweden, 1999.

[2] D. Brown, W. Henshaw, and D. Quinlan. Overture: An object-oriented framework for solving partial differential equations. In Y. Ishikawa,

R. Oldehoeft, J. Reynders, and M Tholburn, editors, *Scientific Computing in Object-oriented Parallel Environments*, pages 177–184, Berlin, 1997. Springer-Verlag.

[3] Are Magnus Bruaset and Hans Petter Langtangen. A comprehensive set of tools for solving partial differential equations; Diffpack. In Morten Dæhlen and Aslak Tveito, editors, *Numerical Methods and Software Tools in Industrial Mathematics*, pages 61–90. Birkhäuser, Boston, 1997.

[4] K. Engø, A. Marthinsen, and H. Z. Munthe-Kaas. DiffMan — an object oriented MATLAB toolbox for solving differential equations on manifolds. Technical Report No. 164, Dept Comp. Sc., University of Bergen, 1999. See also `http://www.ii.uib.no/diffman`.

[5] Phil Grant, Magne Haveraaen, and Mike Webster. Coordinate free programming of computational fluid dynamics problems. *Scientific Programming*, 2000. Accepted for publication.

[6] Magne Haveraaen. Abstractions for programming parallel machines. *Scientific Programming*, 2000. Accepted for publication.

[7] Magne Haveraaen, Helmer André Friis, and Tor Arne Johansen. Formal software engineering for computational modeling. *Nordic Journal of Computing*, 6(3):241–270, 1999.

[8] Jerrold E. Marsden and Thomas J. R. Hughes. *Mathematical Foundations of Elasticity*. Prentice-Hall, 1983.

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

[10] Bernard Schutz. *Geometrical Methods of Mathematical Physics*. Cambridge University Press, 1980.

[11] M Thuné, E Mossberg, P Olsson, J Rantakokko, K Åhlander, and K Otto. Object-oriented construction of parallel PDE solvers. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 203–226. Birkhäuser, 1997.