Coordinate free programming of computational fluid dynamics problems[1]

Philip W. Grant[a], Magne Haveraaen[b,2] and Michael F. Webster[c]
[a]*Department of Computer Science, University of Wales Swansea, UK*
[b]*Department of Informatics University of Bergen, Norway*
[c]*Department of Computer Science, University of Wales Swansea, UK*

It has long been acknowledged that the development of scientific applications is in need of better software engineering practices. Here we contrast the difference between conventional software development of CFD codes with a method based on coordinate free mathematics. The former approach leads to programs where different aspects, such as the discretisation technique and the coordinate systems, can get entangled with the solver algorithm. The latter approach yields programs that segregate these concerns into fully independent software modules. Such considerations are important for the construction of numerical codes for practical problems. The two approaches are illustrated on the coating problem: the simulation of coating a wire with a polymer.

1. Introduction

Numerical codes for solving numerous important civil and industrial problems have existed for many years. A great number of these exhibit the problem which in the computer science community is referred to as *software rot* – a deterioration of software quality as the programs have been modified over the years. Currently, there is therefore, a considerable amount of research being undertaken on the restructuring and redevelopment of numerical software and a genuine interest in using modern software engineering practices in the process, e.g. [1,2,6–8,33,34].

Most numerical codes have been developed within applied mathematics communities, and the preferred programming language has long been Fortran, in one or other of its many versions. Fortran was traditionally an imperative language with multidimensional arrays being the basic type constructor and was without type abstraction facilities. A change of language, e.g., to functional or object-oriented does not necessarily involve a basic change in the approach to programming numerical codes. However, there are genuine benefits from changing to languages with more powerful software structuring concepts. These may include an easier transition to parallel code, as for Fortran-90, High Performance Fortran [26] or functional languages, or a more user friendly interface to advanced data representations, as documented for object-oriented programming.

It is still the case that numerical software development has stayed within the conventional, applied mathematical framework, the main phases of which can be summarised as:

1. model the physical problem;
2. formulate an abstract solver algorithm;
3. discretise in space/time, to transform the continuous problem to its discrete counterpart;
4. refine the solver, utilising properties of the problem (such as symmetries);
5. convert/translate to program code.

This tends to lead to specialised, monolithic programs that are only usable within the realm for which they were developed. They are monolithic in the sense that the whole program was developed as one unit for one purpose. It is often difficult to extract and reuse parts of the software in programs for solving different, but related, problems.

In this paper we will contrast such a conventional development approach with one which is closer to the underlying pure mathematical concepts. The coordinate free development process consists of the steps:

[1]This investigation has been carried out with support from the European Union, ESPRIT-IV project 21871 SAGA (Scientific computing and algebraic abstractions).
[2]This research was mostly carried out during the author's sabbatical at University of Wales Swansea, throughout the academic year 1997/98.

Scientific Programming 8 (2000) 211–230
ISSN 1058-9244 / $8.00 © 2000, IOS Press. All rights reserved

1. model the physical problem;
2. formulate an abstract solver algorithm;
3. refine the solver utilising properties of the problem (such as symmetries) at the *tensor* level;
4. translate to program code.

Here the discretisations etc. come in the form of libraries which are linked into the code. This technique has been proposed in [24], and its software foundation was explored and further developed in [14,17]. A case study is presented in [16].

In this study a detailed worked example is presented to highlight and contrast the two approaches. This will clarify the concepts and differences in reasoning that are used in these two development techniques. Thus, we utilise two pure-bred approaches, being fully aware that conventional development now normally is combined with and utilises more advanced software development techniques. Few groups have employed the abstraction oriented technique to such a full extent as we explore it here. However, the need for higher levels of abstraction have been indicated in [1,2], where the Compose project is described. Compose makes use of the C++ class library Overture [6] to build a framework for extendable PDE solvers where PDE problems are treated as objects. Also in [33] the MAPS system has been proposed which uses sets and maps on which to base more abstract types such as grids and meshes. These can be seen as abstracting the continuous level. The concepts of coordinate free mathematics, as advocated here, provides abstractions at yet a higher level. Hopefully, this presentation may inspire others to try to advance their software technology from a conventional one to higher levels.

The particular example we have chosen to develop is a coating problem for Newtonian flows. This is outlined in Section 2, where the mathematical development leads to a precise algorithmic formulation at a naturally coordinate free level.

The conventional development process in Section 3 begins by describing briefly the finite element method (FEM) where, for simplicity, a Cartesian coordinate system is assumed. Technical details are then supplemented at the discrete level. Often different considerations are presented in a disjoint fashion, and the reader is expected to merge them in a consistent way. The choice of coordinate system has a marked influence on how the operators are defined, but this is often glossed over at this stage of the development process. This approach tends to lead to one type of code, whether it is expressed in a classical procedural language such as Fortran-77, or expressed using object-oriented or even functional languages.

Subsequently, in Section 4 we demonstrate, in more detail, how the coordinate free approach, using tensor mathematics, yields a quite different type of program. The section introduces the concepts of coordinate free mathematics, such as tensor fields, describes the Sophus software library which supports these concepts, and shows how the FEM would be realised within this framework. These concepts are then used to develop the solver for the coating problem at an abstract level, but with the technical detail needed to develop a proper code. This programming style naturally requires the target language to have template classes, a feature lacking from even the more recent versions of Fortran, but which are present in C++ [28]. This latter language is now increasingly being used by practitioners in the field of high performance computing.

In Section 5 the coating problem is modified by changing from the Cartesian coordinate system to an axi-symmetric system which is ideal for the simulation. The implications of such are compared under both programming approaches. For the conventional development technique, this may imply extensive modification and recoding (if say analytical integration is employed), as the change of coordinate system has a marked influence on the way the operators are defined. In contrast, for the coordinate free approach this only entails a reconfiguration of the solver.

The final section discusses consequences of these findings.

2. Outline of the fluid flow problem

The computational fluid dynamics problem used for illustration is that based on a wire coating flow [18]. Here, it is sufficient to consider essentially a Stokesian setting, that of incompressible flow. This is a basis upon which to develop the present methodology, subsequently to be extended into the non-Newtonian regime. The overall goal, in the solution of such problems, is to predict an optimal process design by which a coating (polymer) of suitable properties may be obtained, e.g. smooth coating with minimal residual stressing. The tooling design employed, is *tube* tooling. This is shown schematically in Fig. 1.

The liquid polymer melt undergoes a pressure-driven annular flow within the tooling die, prior to being dragged by the wire through the geometry, as indicated. This design allows the polymer to find a free surface

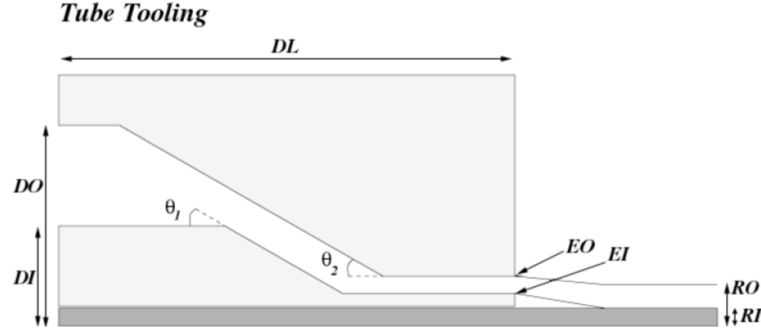

Fig. 1. Tube tooling.

between the die exit and the attachment point on the wire, involving a drawn-down section.

We begin with the basic partial differential equations (Navier-Stokes) to specify the flow problem, and develop a weak, algorithmic form, taking into account a semi-implicit time stepping scheme. This solver will then be further refined, in subsequent sections, to produce executable code using the finite element method for both the conventional and coordinate free development. Iterative and direct algebraic solvers, such as Jacobi iteration and Choleski decomposition, will be used for different stages. The former is employed for inverting Mass-matrix based systems, and the latter for Pressure Poisson Stiffness matrices that are sparse, symmetric and banded.

2.1. The problem description

The specific type of wire coating problem of relevance is defined via input data sets. The governing equations for viscous incompressible isothermal flow may be described, in a coordinate independent form, by the generalised Navier-Stokes equations

$$\rho\frac{\partial \mathbf{u}}{\partial t} + \rho(\mathbf{u}\cdot\nabla)\mathbf{u} - \nabla\cdot(2\mu(\mathsf{sym}(\nabla\mathbf{u}))) + \nabla p = \mathbf{f} \quad (1)$$

with the associated incompressibility constraint

$$\nabla\cdot\mathbf{u} = 0. \quad (2)$$

The parameters and variables with type information are specified as follows:

- ρ is the density, a real number,
- $\mathbf{u}(\mathbf{x},t)$ is the fluid's velocity, a vector field,
- $\frac{\partial \mathbf{u}}{\partial t}$ is the time derivative of the fluid's velocity, a vector field,
- μ is the viscosity, a scalar field and in general a function of $\nabla\mathbf{u}$,
- $p(\mathbf{x},t)$ is the pressure, a scalar field,
- $\mathsf{sym}(-)$ is the symmetrisation operation. For a matrix τ, it can be defined by $\mathsf{sym}(\tau) = \frac{1}{2}(\tau+\tau^T)$, where $_^T$ is the matrix transpose operator,
- $\mathbf{f}$ is an external force acting on the fluid, a vector field,
- ∇ is the spatial derivative, which is used in several forms:
  1. $(\mathbf{v}\cdot\nabla)\mathbf{u}$, the convective derivative of a vector field, yields a scaled derivative of the vector field $\mathbf{u}$ in the direction of the vector field $\mathbf{v}$,
  2. ∇p, the gradient, yields a vector field when applied to a scalar field p,
  3. $\nabla\mathbf{u}$ (gradient) yields a matrix tensor field when applied to a vector field $\mathbf{u}$,
  4. $\nabla\cdot\mathbf{u}$, the divergence, yields a scalar field when "dotted" with a vector field $\mathbf{u}$,
  5. $\nabla\cdot\tau$ (divergence) yields a vector field when "dotted" with a matrix tensor field τ.

2.2. Non-dimensional form

Rather than adopting the above dimensional Eq. (1) directly, the solver will be based on *normalised* equations, using a non–dimensional group number, the Reynold number $Re = \rho\frac{u_0\ell_0}{\mu_0}$. The scaling factors, all real numbers, are: length scale ℓ_0; velocity scale u_0; viscosity scale μ_0; and time scale t_0. We also simplify the presentation, by assuming hereon that $\mathbf{f} = \mathbf{0}$. This yields the formulation, for $0 < Re \leq 1$,

$$Re\frac{\partial \mathbf{u}'}{\partial t'} + Re(\mathbf{u}'\cdot\nabla')\mathbf{u}' - \nabla'\cdot(2\mu'(\mathsf{sym}(\nabla'\mathbf{u}'))) + \nabla' p' = \mathbf{0} \quad (3)$$

$$\nabla'\cdot\mathbf{u}' = 0 \quad (4)$$

where

- $\mathbf{u}' = \mathbf{u}/u_0$ is the fluid's dimensionless velocity,

- ∇' is the spatial derivative after normalisation (using an adjusted metric from its dimensional counterpart),
- $\mu' = \mu/\mu_0$ is the dimensionless viscosity, a scalar field,
- $p' = p/p_0$ is the dimensionless pressure scaled by a pressure factor, a scalar field.

In addition, the simplification also depends on the following relationships between the parameters:

$$u_0 t_0 = \ell_0; \quad p_0 = \mu_0 u_0/\ell_0.$$

This is ensured via choice of u_0, ℓ_0 and μ_0. For clarity, the prime notation is discarded subsequently. In the steady coating problem considered, Reynold numbers tend to be small, of typical value $Re \approx 10^{-4}$. This is due to the large levels of viscosity involved in these polymer melt flows.

The particular test problem considered is annular in configuration and hence two dimensional. The weak formulation presented in Section 2.6, is valid for two or three dimensions, and any geometry. The annular coordinate configuration is detailed in Section 5.1 illustrating the complexity involved in changing coordinate systems.

2.3. *Time-discretisation and strong form of equations*

We now derive the initial equations which lead to the matrix Eqs (26–28) in Section 3. A semi-discretisation of Taylor-Galerkin/pressure correction form is applied to Eq. (3) and solution vectors $\mathbf{u}^n$ and p^n introduced at discrete time t_n for constant time interval Δt.

A set of difference equations is now established which can be used to solve for $\mathbf{u}^{n+1}$ and p^{n+1} in terms of $\mathbf{u}^n$ and p^n. We first take an approximation to $\frac{\partial \mathbf{u}}{\partial t}$ in Eq. (3) by considering the half interval $(t_n, t_{n+1/2})$:

$$\frac{2Re}{\Delta t}(\mathbf{u}^{n+1/2} - \mathbf{u}^n) = s(\mathbf{u}^n) - \nabla p^n \tag{5}$$

where $s(\mathbf{u}) = \nabla \cdot (2\mu \mathsf{sym}(\nabla \mathbf{u})) - Re(\mathbf{u} \cdot \nabla)\mathbf{u}$.

Here, a Taylor-Galerkin approach is adopted of two-step form that addresses the convective aspects of the problem. For a full explanation of this well-established technique see [11,18,30,35].

In order to obtain the solution at time step t_{n+1}, we identify the target equation

$$\begin{aligned} &\frac{Re}{\Delta t}(\mathbf{u}^{n+1} - \mathbf{u}^n) \\ &= s(\mathbf{u}^{n+1/2}) - ((1-\theta)\nabla p^n + \theta \nabla p^{n+1}) \end{aligned} \tag{6}$$

where $0 \leqslant \theta \leqslant 1$. Here, we have taken a θ rule pressure gradient approximation and evaluated s at $\mathbf{u}^{n+1/2}$ (we actually take $\theta = 1/2$, and so, this is a Crank-Nicolson formulation). However, since (6) does not separate $\mathbf{u}^{n+1}$ and p^{n+1} we introduce an intermediate free variable $\mathbf{u}^*$, as the solution of

$$\frac{Re}{\Delta t}(\mathbf{u}^* - \mathbf{u}^n) = s(\mathbf{u}^{n+1/2}) - \nabla p^n. \tag{7}$$

Note that with $\mathbf{u}^*$ also satisfying

$$\frac{Re}{\Delta t}(\mathbf{u}^{n+1} - \mathbf{u}^*) = \theta(\nabla p^n - \nabla p^{n+1}), \tag{8}$$

it follows that $\mathbf{u}^{n+1}$ will satisfy Eq. (6) as required. It is implied that the velocity $\mathbf{u}^n$ at all times t_n, must satisfy the incompressibility constraint expressed by Eq. (4). By applying the divergence operator $\nabla\cdot$ to Eq. (8), we derive the associated auxiliary equation

$$\frac{Re}{\Delta t}\nabla \cdot \mathbf{u}^* = \theta \nabla \cdot \nabla(p^{n+1} - p^n). \tag{9}$$

This Poisson equation for the pressure difference over a single time step completes the problem specification. The four vector Eqs (5), (7), (9) and (8), will subsequently give rise to matrix equations once spatial discretisation has been conducted. This we outline in Section 3.

Rearranging the equations yields the following strong formulation consisting of three steps. The first step splits into two mathematically similar substeps, following a predictor-corrector pattern (two-step Taylor-Galerkin). The substeps 1a and 1b calculate the halfstep $\mathbf{u}^{n+1/2}$ and the auxiliary $\mathbf{u}^*$ approximations to the velocities.

- Step 1a

$$\begin{aligned} &\frac{2Re}{\Delta t}(\mathbf{u}^{n+1/2} - \mathbf{u}^n) \\ &= \nabla \cdot (2\mu\, \mathsf{sym}(\nabla \mathbf{u}^n)) \\ &\quad -Re(\mathbf{u}^n \cdot \nabla)\mathbf{u}^n - \nabla p^n \end{aligned} \tag{10}$$

- Step 1b

$$\begin{aligned} &\frac{Re}{\Delta t}(\mathbf{u}^* - \mathbf{u}^n) \\ &= \nabla \cdot (2\mu\, \mathsf{sym}(\nabla \mathbf{u}^{n+1/2})) \\ &\quad -Re(\mathbf{u}^{n+1/2} \cdot \nabla)\mathbf{u}^{n+1/2} - \nabla p^n \end{aligned} \tag{11}$$

- Step 2

$$\theta \nabla \cdot (\nabla(p^{n+1} - p^n)) = \frac{Re}{\Delta t}(\nabla \cdot \mathbf{u}^*) \tag{12}$$

– Step 3

$$\frac{Re}{\Delta t}(\mathbf{u}^{n+1} - \mathbf{u}^*) = -\theta\nabla(p^{n+1} - p^n) \quad (13)$$

These steps are iterated from an initial guess to produce a steady state solution.

2.4. Initial and boundary conditions

The system of equations of the previous section are solved by imposing appropriate initial conditions on the domain Ω,

$$\mathbf{u}(\mathbf{x}, 0) = \mathbf{u}_0(\mathbf{x})$$
$$p(\mathbf{x}, 0) = p_0(\mathbf{x}),$$

where $\nabla \cdot \mathbf{u}_0 = 0$, and appropriate boundary conditions on Γ as

$$\begin{aligned} \mathbf{u} &= \mathbf{g}_1(\mathbf{x}, t) \text{ on } \Gamma_1 \\ \mathbf{n} \cdot \sigma &= \mathbf{g}_2(\mathbf{x}, t) \text{ on } \Gamma_2. \end{aligned} \quad (14)$$

Here $\Gamma = \Gamma_1 \cup \Gamma_2$ encloses the domain Ω, $\mathbf{n}$ is the unit outer normal on Γ, $\mathbf{g}_1(\mathbf{x}, t)$ represents the velocity vector prescribed on Γ_1, $\mathbf{g}_2(\mathbf{x}, t)$ designates the traction vector prescribed on Γ_2, and σ is the total Cauchy's stress tensor. For an incompressible Newtonian viscous fluid we have in a Cartesian coordinate system

$$\sigma_{ij} = -p\delta_{ij} + \mu_0(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i})$$

where μ_0 is a Newtonian fluid viscosity and δ_{ij} the unit tensor. The coating problem is a steady flow and boundary conditions of type Γ_1 only are assumed. In the most general statement of the problem free surfaces with steady traction boundary conditions would apply.

Since there is no restriction on the choice of boundary conditions for $\mathbf{u}^*$ (a free variable), we may equate $\mathbf{u}^*$ to $\mathbf{u}^{n+1}$ on the complete boundary Γ. The implication from such a choice and step 3 is that $\nabla(p^{n+1} - p^n)$ should vanish on Γ. Also, the boundary conditions at step 3 are clearly prescribed. For steady boundary condition problems of immediate relevance, such assumptions are exact. For transient instances, assuming smoothness in time for pressure, would indicate accuracy to a first order in this variable at least, and hence to a second order in velocity from Eq. (13). This argument maintains the overall order of the scheme [31]. We shall see below, in the variational form of the problem, that natural homogeneous Neumann boundary conditions emerge from step 2 and these are a distinct advantage to this scheme [18,19].

2.5. Semi-implicit form of solver

With a Crank-Nicolson treatment of diffusion terms we derive a semi-implicit time stepping scheme of second order accuracy. This allows a wider window of stability above an explicit implementation and thus permits the use of practical working time steps [18,19,31]. Such advantages continue through to more complex non-Newtonian settings, where inertial influences via Reynolds number are low, in contrast to elastic effects (for further details see [22]). In step 1a, the term $\mathbf{u}^n$ in the viscous term of the explicit scheme is replaced by the average at $t_{n+1/2}$ and t_n. After rearranging, so that $\mathbf{u}^{n+1/2}$ appears in the difference term $\mathbf{u}^{n+1/2} - \mathbf{u}^n$ on the left hand side only, we obtain for the modified step 1a$'$

$$\begin{aligned} &\frac{2Re}{\Delta t}(\mathbf{u}^{n+1/2} - \mathbf{u}^n) \\ &\quad -\nabla \cdot (\mu\, \mathsf{sym}(\nabla(\mathbf{u}^{n+1/2} - \mathbf{u}^n)) \\ &= 2\nabla \cdot (\mu\, \mathsf{sym}(\nabla(\mathbf{u}^n)) \\ &\quad -Re(\mathbf{u}^n \cdot \nabla)\mathbf{u}^n - \nabla p^n. \end{aligned} \quad (15)$$

Step 1b is treated in a similar manner, where $\mathbf{u}^{n+1/2}$ in the viscous term is replaced by the average of $\mathbf{u}^n$ and $\mathbf{u}^*$, yielding step 1b$'$

$$\begin{aligned} &\frac{Re}{\Delta t}(\mathbf{u}^* - \mathbf{u}^n) - \nabla \cdot (\mu\, \mathsf{sym}(\nabla(\mathbf{u}^* - \mathbf{u}^n)) \\ &= 2\nabla \cdot (\mu\, \mathsf{sym}(\nabla(\mathbf{u}^n)) - \nabla p^n \\ &\quad -Re(\mathbf{u}^{n+1/2} \cdot \nabla)\mathbf{u}^{n+1/2}. \end{aligned} \quad (16)$$

The semi-implicit solver is then described by Eqs (15,16,12) and (13).

2.6. Weak formulation

The weak variational formulation of the problem is now derived. We use two sets of test functions: a class of quadratic vector shape functions $\mathbf{v} \in V$ and a class of scalar shape functions $q \in Q$. Both sides of Eqs (13), (15) and (16) are dotted with the vector shape functions $\mathbf{v}$ and the scalar test functions q are multiplied into the algorithmic step 2, Eq. (12).

The results are integrated over the whole domain Ω and simplified, via integration by parts. This reduces second-order derivatives from the integrands to first-order and pressure gradients to order zero in the defining velocity equations. Further simplification is

achieved using the divergence theorem which gives rise to the integrals over the boundary Γ.

From Eq. (15) step 1a$''$ becomes:

$$\frac{2Re}{\Delta t}\int_\Omega(\mathbf{u}^{n+1/2}-\mathbf{u}^n)\cdot\mathbf{v}d\Omega$$
$$+\int_\Omega \mu\,\mathsf{sym}(\nabla(\mathbf{u}^{n+1/2}-\mathbf{u}^n))\cdot\nabla\mathbf{v}d\Omega$$
$$=\int_\Gamma(\mu\,\mathsf{sym}(\nabla(\mathbf{u}^{n+1/2}+\mathbf{u}^n))\cdot\mathbf{v}-p^n\mathbf{v})\cdot\mathbf{n}d\Gamma$$
$$-2\int_\Omega \mu\,(\mathsf{sym}\nabla(\mathbf{u}^n))\cdot\nabla\mathbf{v}d\Omega$$
$$-Re\int_\Omega((\mathbf{u}^n\cdot\nabla)\mathbf{u}^n)\cdot\mathbf{v}d\Omega$$
$$+\int_\Omega p^n(\nabla\cdot\mathbf{v})d\Omega$$

In a similar manner, starting from Eq. (16), we obtain step 1b$''$:

$$\frac{Re}{\Delta t}\int_\Omega(\mathbf{u}^*-\mathbf{u}^n)\cdot\mathbf{v}d\Omega$$
$$+\int_\Omega \mu\,\mathsf{sym}(\nabla(\mathbf{u}^*-\mathbf{u}^n))\cdot\nabla\mathbf{v}d\Omega$$
$$=\int_\Gamma(\mu\,\mathsf{sym}(\nabla(\mathbf{u}^*+\mathbf{u}^n))\cdot\mathbf{v}-p^n\mathbf{v})\cdot\mathbf{n}d\Gamma$$
$$-\int_\Omega 2\mu\mathsf{sym}(\nabla(\mathbf{u}^n))\cdot\nabla\mathbf{v}d\Omega$$
$$+\int_\Omega p^n\nabla\cdot\mathbf{v}d\Omega$$
$$-\int_\Omega Re(\mathbf{u}^{n+1/2}\cdot\nabla)\mathbf{u}^{n+1/2}\cdot\mathbf{v}d\Omega$$

Step 2$''$ becomes:

$$\theta\int_\Omega\nabla(p^{n+1}-p^n)\cdot\nabla q d\Omega$$
$$-\theta\int_\Gamma(\nabla(p^{n+1}-p^n)\cdot\mathbf{n})q d\Gamma$$
$$=\frac{-Re}{\Delta t}\int_\Omega(\nabla\cdot\mathbf{u}^*)q d\Omega$$

and step 3$''$:

$$\frac{Re}{\Delta t}\int_\Omega(\mathbf{u}^{n+1}-\mathbf{u}^*)\cdot\mathbf{v}d\Omega$$
$$=\theta\int_\Omega(p^{n+1}-p^n)\nabla\mathbf{v}d\Omega$$
$$-\theta\int_\Gamma(p^{n+1}-p^n)(\mathbf{v}\cdot\mathbf{n})d\Gamma$$

Finally, all surface integrals are equated to zero. This is in accordance with the details of the boundary conditions as expressed in Section 2.4. The test functions used will vanish on the boundary Γ_1, where the velocities are prescribed, as indicated in Eq. (14) and for the coating problem considered, it is assumed there are no traction boundary contributions.

In step 2 of the algorithm, the surface integral can be removed since $\nabla(p^{n+1}-p^n)$ is taken to vanish over the whole of Γ, as discussed in Section 2.4. In addition, due to imposition of fixed Dirichelet boundary conditions on $\mathbf{u}^{n+1}-\mathbf{u}^*$, we can ignore the associated surface integrals over Γ_2 for the pressure fields again owing to the choice of test functions. The weak formulation can thus be expressed through the following set of equations:

Step 1a:

$$\frac{2Re}{\Delta t}\int_\Omega(\mathbf{u}^{n+1/2}-\mathbf{u}^n)\cdot\mathbf{v}d\Omega$$
$$+\int_\Omega \mu\,\mathsf{sym}(\nabla(\mathbf{u}^{n+1/2}-\mathbf{u}^n))\cdot\nabla\mathbf{v}d\Omega$$
$$=-2\int_\Omega \mu\,\mathsf{sym}(\nabla(\mathbf{u}^n))\cdot\nabla\mathbf{v}d\Omega \tag{17}$$
$$-Re\int_\Omega((\mathbf{u}^n\cdot\nabla)\mathbf{u}^n)\cdot\mathbf{v}d\Omega$$
$$+\int_\Omega p^n(\nabla\cdot\mathbf{v})d\Omega$$

Step 1b:

$$\frac{Re}{\Delta t}\int_\Omega(\mathbf{u}^*-\mathbf{u}^n)\cdot\mathbf{v}d\Omega$$
$$+\int_\Omega \mu\,\mathsf{sym}(\nabla(\mathbf{u}^*-\mathbf{u}^n))\cdot\nabla\mathbf{v}d\Omega$$
$$=-2\int_\Omega \mu\mathsf{sym}(\nabla(\mathbf{u}^n))\cdot\nabla\mathbf{v}d\Omega \tag{18}$$
$$-Re\int_\Omega(\mathbf{u}^{n+1/2}\cdot\nabla)\mathbf{u}^{n+1/2}\cdot\mathbf{v}d\Omega$$
$$+\int_\Omega p^n\nabla\cdot\mathbf{v}d\Omega$$

Step 2:

$$\theta\int_\Omega\nabla(p^{n+1}-p^n)\cdot\nabla q d\Omega = \frac{-Re}{\Delta t}\int_\Omega(\nabla\cdot\mathbf{u}^*)q d\Omega \tag{19}$$

Step 3:

$$\frac{Re}{\Delta t}\int_\Omega(\mathbf{u}^{n+1}-\mathbf{u}^*)\cdot\mathbf{v}d\Omega = \theta\int_\Omega(p^{n+1}-p^n)\nabla\cdot\mathbf{v}d\Omega. \tag{20}$$

This completes the development of the variational formulation of the problem. This will be used as the starting point for the discretisation over space both for the conventional development, in Section 3, and for the coordinate free approach, in Section 4, both leading to implementable simulations.

3. Conventional software development and discretisation

This section describes the conventional approach for numerically solving Eqs (3) and (4). The domain over which the solution is required is first triangulated into a finite-element mesh with nodes at the vertices and mid points of the edges. The weak formulation, as presented in Section (2.6), is used to derive a set of matrix equations relating the values of the velocities and pressure at the mesh nodes. This is presented initially within a Cartesian coordinate framework.

3.1. The finite element method

We now obtain the fully discretised equations using the finite element method (FEM). The total number of nodes and vertex nodes in the mesh are denoted by m_v and m_p respectively. Two sets of spatial shape functions, V and Q are employed. $V = \{\varphi_j | j = 1,\dots,m_v\}$ is a set of piecewise quadratic functions (quadratic on each element) and $Q = \{\psi_j | j = 1,\dots,m_p\}$ is a set of linear functions (linear on each element). A shape function is associated with a node, it is continuous, takes on the value 1 at its node and 0 at the other nodes of its compact support. It has local support, in the sense that the shape function is non-zero only on the elements bordering to its node. A vector field is now approximated by $\mathbf{u}^n \approx \sum_{j=1}^{m_v}\mathbf{U}_j^n\varphi_j$ and a scalar field by $p^n \approx \sum_{j=1}^{m_p}P_j^n\psi_j$, where the superscript n denotes evaluation at time step n, as before.

The finite element interpolants are then substituted in Eqs (17–21) and appropriate test functions $\mathbf{v}$ and q chosen to produce a set of matrix equations.

Taking the first Eq. (17) we have:

$$\begin{aligned}\sum_{j=1}^{m_v}\Big[\frac{2Re}{\Delta t}\int_\Omega\varphi_j(\mathbf{U}_j^{n+1/2}-\mathbf{U}_j^n)\cdot\mathbf{v}d\Omega \\ +\int_\Omega\mu\,\mathsf{sym}(\nabla(\mathbf{U}_j^{n+1/2}-\mathbf{U}_j^n)\varphi_j)\cdot\nabla\mathbf{v}d\Omega\Big] \\ = -\sum_{j=1}^{m_v}\Big[2\int_\Omega\mu\,\mathsf{sym}(\nabla\varphi_j\mathbf{U}_j^n)\cdot\nabla\mathbf{v}d\Omega \\ +Re\int_\Omega((\mathbf{U}^n\cdot\nabla)\varphi_j\mathbf{U}_j^n))\cdot\mathbf{v}d\Omega\Big] \\ +\sum_{k=1}^{m_p}(\int_\Omega\psi_k(\nabla\cdot\mathbf{v})d\Omega P_k^n).\end{aligned} \tag{21}$$

It is convenient to represent all the vectors $\mathbf{U}_j$, in $\mathbf{R}^3$, $j = 1\cdots m_v$, as one combined column vector U of size $3m_v$, thus

$$U = (U_{11}, U_{12}, \cdots, U_{1m_v}, U_{21}, U_{22}, \cdots, U_{2m_v}, U_{31}, U_{32}, \cdots, U_{3m_v})^T$$

where U_{lj} is the x_l-component of $\mathbf{U}_j$ for $l = 1,\cdots,3$.

We consider the various types of integrals which appear in Eq. (21). The first term is of the form:

$$\sum_{j=1}^{m_v}\int_\Omega\phi_j\mathbf{U}_j\cdot\mathbf{v}d\Omega.$$

Expanding $\mathbf{U}_j$ using the standard Euclidean basis vectors $\{\mathbf{e}_1,\mathbf{e}_2,\mathbf{e}_3\}$ leads to

$$\sum_{l=1}^{3}\sum_{j=1}^{m_v}\int_\Omega\phi_jU_{lj}\mathbf{e}_l\cdot\mathbf{v}d\Omega.$$

The Galerkin method is adopted by choosing the test functions to be generated by the shape functions V. Letting $\boldsymbol{\varphi}_j^l = \varphi_j\mathbf{e}_l$ and taking as test functions $\mathbf{v} = \boldsymbol{\varphi}_i^m$ for $m = 1,\cdots,3$ and $i = 1,\cdots,m_v$ yields the following $3m_v$ terms

$$\sum_{l=1}^{3}\sum_{j=1}^{m_v}\int_\Omega\boldsymbol{\varphi}_j^l\cdot\boldsymbol{\varphi}_i^m d\Omega U_{lj}.$$

Taking the above as elements of a $3m_v$ column vector, it clearly can be represented as the matrix–vector product $\mathbf{M}U$ where $\mathbf{M}$ has the block structure

$$\mathbf{M} = \begin{bmatrix}\mathbf{M}^{11} & \mathbf{M}^{12} & \mathbf{M}^{13}\\ \mathbf{M}^{21} & \mathbf{M}^{22} & \mathbf{M}^{23}\\ \mathbf{M}^{31} & \mathbf{M}^{32} & \mathbf{M}^{33}\end{bmatrix}$$

where

$$\mathbf{M}_{ij}^{lm} = \int_\Omega\boldsymbol{\varphi}_j^l\cdot\boldsymbol{\varphi}_i^m d\Omega. \tag{22}$$

Evidently $\mathbf{M}^{lm} = \mathbf{0}$ for $l \neq m$ and $\mathbf{M}^{ll}$ are all identical and given by

$$\mathbf{M}^{ll}_{ij} = \int_\Omega \varphi_i \varphi_j d\Omega.$$

This matrix $\mathbf{M}$ is the *mass matrix*.

To deal with the second term in Eq. (21), after expanding $\mathbf{U}_j$ in terms of the basis vectors, we consider

$$\sum_{l=1}^{3}\sum_{j=1}^{m_v} \int_\Omega \mu\, \mathsf{sym}(\nabla(U_{lj}\boldsymbol{\varphi}^l_j)) \cdot \nabla \mathbf{v} d\Omega.$$

This yields the $3m_v$ terms

$$\sum_{l=1}^{3}\sum_{j=1}^{m_v} \int_\Omega \mu\, \mathsf{sym}(\nabla(U_{lj}\boldsymbol{\varphi}^l_j)) \cdot \nabla \boldsymbol{\varphi}^m_i d\Omega.$$

As a vector this latter term can be expressed as SU and writing $\mathbf{S}$ in block form $(\mathbf{S}^{lm})$, each block is determined by

$$\mathbf{S}^{lm}_{ij} = \int_\Omega \mu\, \mathsf{sym}(\nabla(\boldsymbol{\varphi}^l_j)) \cdot \nabla \boldsymbol{\varphi}^m_i d\Omega. \quad (23)$$

$\mathbf{S}$ is referred to as the *diffusion matrix*. Note that $\mathbf{S}$ is symmetric and in particular $(\mathbf{S}^{ml}) = (\mathbf{S}^{lm})^T$.

The next expression in Eq. (21) to consider is

$$\sum_{l=1}^{3}\sum_{j=1}^{m_v} \int_\Omega ((\mathbf{U}\cdot\nabla)\boldsymbol{\varphi}^l_j U_{lj}) \cdot \mathbf{v} d\Omega$$

(where $\mathbf{U} = \sum_{k=1}^{m_v} \varphi_k \mathbf{U}_k$) producing a $3m_v$ vector which can be written $\mathbf{N}(\mathbf{U})U$ where

$$\mathbf{N}(\mathbf{U})^{lm}_{ij} = \int_\Omega ((\sum_{k=1}^{m_v} \varphi_k \mathbf{U}_k \cdot \nabla)\boldsymbol{\varphi}^l_j) \cdot \boldsymbol{\varphi}^m_i d\Omega. \quad (24)$$

$\mathbf{N}(\mathbf{U})$ is called the *convection matrix*.

The final term in Eq. (21) has the form

$$\sum_{k=1}^{m_p} (\int_\Omega \psi_k (\nabla\cdot\mathbf{v}) d\Omega P^n_k),$$

yielding the $3m_v$ terms

$$\sum_{k=1}^{m_p} (\int_\Omega \psi_k (\nabla\cdot\boldsymbol{\varphi}^m_i) d\Omega P^n_k).$$

Since $\nabla \cdot \boldsymbol{\varphi}^m_j = \frac{\partial \varphi_j}{\partial x_m}$ for $m = 1, \cdots, 3$, the above can be represented as $\mathbf{L}P$, where

$$\mathbf{L} = \begin{bmatrix} \mathbf{L}^1 \\ \mathbf{L}^2 \\ \mathbf{L}^3 \end{bmatrix}$$

and the entries of each block are defined as

$$\mathbf{L}^m_{ik} = \int_\Omega \psi_k \frac{\partial \varphi_i}{\partial x_m} d\Omega \text{ for } m = 1, \cdots, 3. \quad (25)$$

$\mathbf{L}^T$ is then the associated *incompressibility matrix*.

Using the above, the desired matrix equation for a discretised version of step 1a is:

$$\begin{aligned}&(\frac{2Re}{\Delta t}\mathbf{M} + \mathbf{S})(U^{n+1/2} - U^n) \\ &= \mathbf{L}P^n - (2\mathbf{S} + Re\mathbf{N}(U^n))U^n.\end{aligned} \quad (26)$$

Equations (18) and (20) can be discretised in a similar manner yielding the Eqs (27) and (28) for steps 1b and 3.

$$\begin{aligned}&(\frac{Re}{\Delta t}\mathbf{M} + \mathbf{S})(U^* - U^n) \\ &= \mathbf{L}P^n - 2\mathbf{S}U^n - Re\mathbf{N}(U^{n+1/2})U^{n+1/2}.\end{aligned} \quad (27)$$

$$\frac{Re}{\Delta t}\mathbf{M}(U^{n+1} - U^*) = \theta \mathbf{L}(P^{n+1} - P^n). \quad (28)$$

For discretising step 2, we have, substituting in Eq. (19)

$$\begin{aligned}&\theta \sum_{k=1}^{m_p} \int_\Omega \nabla\psi_k \cdot \nabla q d\Omega (P^{n+1}_k - P^n_k) \\ &= \frac{-Re}{\Delta t} \sum_{l=1}^{3}\sum_{j=1}^{m_v} \int_\Omega (\nabla \cdot \boldsymbol{\varphi}^l_j) q d\Omega U^*_{lj}.\end{aligned}$$

Here we take Q as the set of test functions to obtain the following set of m_p equations:

$$\begin{aligned}&\theta \sum_{k=1}^{m_p} \int_\Omega \nabla\psi_k \cdot \nabla \psi_i d\Omega (P^{n+1}_k - P^n_k) \\ &= \frac{-Re}{\Delta t} \sum_{l=1}^{3}\sum_{j=1}^{m_v} \int_\Omega \frac{\partial \varphi_j}{\partial x_l} \psi_i d\Omega U^*_{lj}.\end{aligned}$$

To express the left hand side, we introduce the *pressure stiffness matrix* K

$$(\mathbf{K}_{ij}) = (\int_\Omega \nabla\psi_j \cdot \nabla\psi_i d\Omega) \quad (29)$$

then, step 2 becomes

$$\theta \mathbf{K}(P^{n+1} - P^n) = \frac{-Re}{\Delta t}\mathbf{L}^T U^*. \quad (30)$$

Equations (26,27,30) and (28) then constitute the matrix representation of the problem. It is important to realise that the integrands appearing in the definitions of $\mathbf{M}$, $\mathbf{K}$, and $\mathbf{L}$ can all be evaluated analytically since the functions φ_i and ψ_k are of a simple form, and so the matrix elements are known at the start of the computation.

3.2. Solving the equations

Equations (26)–(28) are solved using Jacobi iteration. Although the mass matrix $\mathbf{M}$ is generally very large, it is not necessary to store such a matrix explicitly, and in practice, only the *element* sub-block matrices $\mathbf{M}_e$ need be constructed. The iteration takes the form:

$$\mathbf{X}^{(r+1)} = \mathbf{X}^{(r)} - \omega\mathbf{D}^{-1}\sum_{e=1}^{m_{el}}\mathbf{L}_e^T\mathbf{M}_e\mathbf{X}_e^{(r)} - \omega\mathbf{D}^{-1}\mathbf{B} \tag{31}$$

where $\mathbf{D}$ is a chosen diagonal form and ω is a positive relaxation factor [19,10].

The direct Choleski method is employed to solve the pressure difference Eq. (30). The decomposition must be performed *only once* at the outset of the time-stepping procedure. This leads to an efficient implementation.

Termination for steady state is determined by using an ℓ^2-norm on the difference between the velocity vectors at consecutive time steps and halting when this is less than some threshold tolerance.

3.3. A discussion of relevant coding techniques

The exposition above is at a very basic data level, expressed in the form of vectors and matrices. The form is imperative in style – *do this, then do that, update the variables and repeat until termination* – this is suitable for implementation in any of a number of languages. The preferred choice among scientific programmers would be a dialect of Fortran, typically Fortran-77, but Fortran-90 is becoming more popular, see the survey [20]. The program developed in this section has been implemented in Fortran-90 and amounts to about $15,000$–$20,000$ lines of code.

Several authors have followed this style of software development for Object-Oriented languages such as C++ and more recently Java. This typically results in classes representing vector and matrix operations with BLAS like routines. The general impression is that there is little gained over the use of Fortran from the conceptual point of view, even though the increased emphasis on software engineering principles that these projects have, often is beneficial for the quality of the code – especially when compared to *dusty deck* Fortran.

A variation on this is the use of the Diffpack software package [7]. It is a support environment for the development of object-oriented numerical software, and has embodied many good software engineering features and practices. The tendency is for Diffpack code to be closer to the numerical algorithm allowing the software developer to focus more clearly on the numerical aspects of the task than that experienced by a Fortran programmer. However, Diffpack's approach to numerical software development is still basically the conventional one outlined above, making the choice of discretisation and coordinate systems explicit in the code.

More applicative implementations using functional languages have also been tried [13]. In these cases also the focus is on basic data structures and operations on them. The benefits of using function calls and recursion rather than variable updates and iteration show more in a greater readiness for automatic parallelisation (see the papers and approaches in [29]) than in the way the code is structured in the large. However, as for object-orientation, developers working in this area tend to put more emphasis on good software practices than does the classically trained Fortran programmer.

4. Coordinate free methodology

In the coordinate free methodology we write the program code based directly on the concepts present in the abstract algorithm as developed in Section 2.6. For this we need an understanding of these abstract mathematical concepts, together with their software realisation and support. Then implementing the numerical solver becomes straightforward.

In this section we first briefly present the notions of tensors and coordinate freeness, introducing the necessary notations and terminology. An outline is then given of the Sophus library and it is indicated how the finite element method (FEM) may be implemented in Sophus. Finally, we show how the solver for the coating problem can be implemented using Sophus.

4.1. Coordinate free mathematics

The concepts from algebra and coordinate free mathematics sketched here are given thorough treatment in the books [21,27].

Plain numerical types, such as the real numbers, are often abstracted as fields or rings. Given a ring R, we may define, for any integer $n \geqslant 0$, n-positional numerical types over R, commonly known as n-dimensional vectors $\mathbf{v} \in R^n$. For any n, the set of n-dimensional vectors $\mathbf{v}$ form a vector space V. (Strictly speaking, this is a *module*, and only a vector space when R is a field. However, to avoid confusion with the usage

of module in computing we shall use the term vector space below).

Given a basis $B = \{\mathbf{b}_1, \dots, \mathbf{b}_n\}$, a vector $\mathbf{v} \in V$ can be identified by its coordinates $\bar{v} = (a_1, \dots, a_n)$, i.e., ring elements $a_1, \dots, a_n \in R$, such that $\mathbf{v} = a_1^* \mathbf{b}_1 + \dots + a_n^* \mathbf{b}_n$. Note that there is a distinction between a vector $\mathbf{v} \in V$ as an n-positional numerical type, and its coordinates $\bar{v}$ with respect to a basis $B \subseteq V$, even though the latter also may be considered an n-positional numerical type. It is only when using the normalised Cartesian coordinate system that the data values $\mathbf{v} = \bar{v}$. The normal representation for vectors is relative to some coordinate system, but then care has to be taken to interpret coordinates $\bar{v}$ only relative to that coordinate system.

The collection M of linear mappings from vector spaces V to W form a vector space. If $V = R^m$ and $W = R^n$, for $m, n \geqslant 0$, we know that the linear mappings from V to W can be represented as n by m matrices, which we also may regard as (nm)-positional numerical types, or, in other words, as (nm)-dimensional vectors. Using the tensor product $\otimes$ we may combine a basis B for an m-dimensional vector space V with a basis B' for an n-dimensional vector space W to form a basis $B \otimes B'$ for the (nm)-dimensional vector space M of linear mappings from V to W. This means that any linear mapping $\ell \in M$ is expressible using a linear combination of the basis vectors $\mathbf{b}_i \otimes \mathbf{b}'_j$, for $\mathbf{b}_i \in B \subseteq V$ and $\mathbf{b}'_j \in B' \subseteq W$ (this does not imply that every vector ℓ can be expressed by $\mathbf{v} \otimes \mathbf{w}$ for $\mathbf{v} \in V$ and $\mathbf{w} \in W$). If $V = R$ then the tensor product reduces to the ring-vector product.

Covectors $\mathbf{v} \in V^*$ are $(n1)$-dimensional vectors which take n-dimensional vectors $\mathbf{v}' \in V$ to 1-dimensional vectors (ring elements $\mathbf{v}(\mathbf{v}') \in R$). There is an operation $_^* : V \to V^*$, dualisation, which takes a vector and returns a covector. The inner product, or the dot product $\mathbf{v} \cdot \mathbf{v}'$ of vectors $\mathbf{v}, \mathbf{v}' \in V$ is defined by $\mathbf{v} \cdot \mathbf{v}' = \mathbf{v}^*(\mathbf{v}')$.

Normally we have to do computations on a vector $\mathbf{v} \in V$ on its coordinate representation $\bar{v}$ relative to a coordinate system $B \subseteq V$. It is important that changing the basis of the vector space V should not change the effect of operations on the vector $\mathbf{v} \in V$. This can be achieved by making the operations, such as the dot product and dualisation $_^*$, aware of which basis is being used. Then appropriate corrective action may be taken in the computations, and we may use operations on vectors independently of coordinate system, thus achieving *coordinate free mathematics*.

Our tool for doing this is the notion of tensors. Given a ring R, an integer $b \geqslant 1$, a b-dimensional vector space V with basis vectors $B \subseteq V$, then tensor spaces $T^{(k)}_{R,B}$ are the n-dimensional vector spaces where $n = b^k$, for some $k \geqslant 0$. It is easy to see that the ring itself is the tensor space $T^{(0)}_{R,B}$ for any b. The tensor product of a (b^{k_1})-dimensional tensor $\tau_1 \in T^{(k_1)}_{R,B}$ and (b^{k_2})-dimensional tensor $\tau_2 \in T^{(k_2)}_{R,B}$ is a $(b^{k_1+k_2})$-dimensional tensor $(\tau_1 \otimes \tau_2) \in T^{(k_1+k_2)}_{R,B}$. The dot product of two tensors $\tau_1, \tau_2 \in T^{(k)}_{R,B}$ is a tensor $(\tau_1 \cdot \tau_2) \in T^{(0)}_{R,B}$.

We need to distinguish between the tensor spaces and also on how they are formed with vector and covector components. This gives us many distinct tensor spaces for any b^k dimensions, $k > 0$, providing a distinct slice for every combination of covectors and vectors in the formation of $T^{(k)}_{R,B}$. In conventional notation these slices are distinguished using *upper* and *lower* indices. Making certain that the covector and vector components match up, we may apply one tensor as a linear mapping to another tensor. Specifically this notion carries down to b-dimensional tensors $T^{(1)}_{R,B}$ (applying covectors to vectors) and 1-dimensional tensors $T^{(0)}_{R,B}$ (multiplication of ring elements).

4.2. Scalar, vector and tensor fields

For a ring or field R, such as the set of real numbers, given a set X, the set $F_{X \to R}$ of functions $f : X \to R$ forms a ring with, for any $f, g \in F_{X \to R}$, multiplication defined by $(f \cdot g)(x) = f(x) \cdot g(x)$ for all $x \in X$, addition by $(f + g)(x) = f(x) + g(x)$ for all $x \in X$, and so forth. The ring $F_{X \to R}$ is called a scalar field. We may consider every ring element $a \in R$ as a scalar field $a \in F_{X \to R}$ by defining $a(x) = a$ for every $x \in X$. Scalar fields $F_{X \to R}$ may be used in the constructions of vector spaces and tensor spaces, yielding the notions of vector fields $V_{X \to R}$, with basis $E \subseteq V_{X \to R}$, and tensor fields $T^{(k)}_{F_{X \to R},E}$, for every $k \geqslant 0$. A scalar field assigns a different value to every point in X. A vector field likewise may change throughout X, such that every point of X is assigned a vector with different direction and magnitude. The basis vectors for a vector field may thus define any kind of curvilinear coordinate system, including Cartesian and cylindrical.

For X a vector space over the ring R, then X will be included in $V_{X \to R}$ in the same way as R is included in $F_{X \to R}$. And, as before, there are inclusions from $F_{X \to R}$ to $T^{(0)}_{F_{X \to R},E}$ and from $V_{X \to R}$ to $T^{(1)}_{F_{X \to R},E}$.

If the set X satisfies the properties of being a manifold, i.e., it has the notions of direction and proximi-

ty, we may define non-trivial derivation operations on the scalar field. Typically X will be a b-dimensional vector space with basis vectors $B \subseteq X$, such as the 3-dimensional space in which most physics takes place. We may then define partial derivatives $\frac{\partial f}{\partial \mathbf{x}}$ of the scalar fields $f \in F_{X\to R}$ along any vector $\mathbf{x} \in X$. The partial derivative will also be a scalar field since it changes throughout X, so $\frac{\partial f}{\partial \mathbf{x}} \in F_{X\to R}$. Knowing the partial derivatives along the basis vectors B is sufficient to compute the derivatives along any vector in X. If X is a b-dimensional vector space this forces the vector field $V_{X\to R}$ to have b dimensions, and the basis E for the vector field will then have b linearly independent b-dimensional vectors. At the level of tensor fields $T^{(k)}_{F_{X\to R},E}$ we may now define many derivation operations, all of which may be computed from the partial derivatives on the scalar field $F_{X\to R}$.

For a scalar field $f \in F_{X\to R}$, an integration $a = \int_\Omega f dX$ of f for a subdomain $\Omega \subseteq X$ with basis vectors $B \subseteq X$ yields a ring element $a \in R$. Scalar field integration gives rise to tensor field integration $\alpha = \int_\Omega \sigma dX$ for $\sigma \in T^{(0)}_{F_{X\to R},E}$ yielding values $\alpha \in R$.

Derivation and integration operations on tensor fields, like the dot product, depend on the choice of basis E for $V_{X\to R}$.

We may now treat all symbols in the equations of Section 2.6 as tensors and coordinate free operations on tensors. Some of the descriptions in Section 2.1 are now explained in the more general tensor setting. Let us summarise the operations we need, after the simplification of the solver, and assuming the appropriate tensor slices:

– $\mathsf{sym}(\sigma)$ is the tensor symmetrisation operator taking a tensor $\sigma \in T^{(k)}_{F_{X\to R},E}$ to a tensor in the symmetrical subspace of $T^{(k)}_{F_{X\to R},E}$.
– $+$, $-$ are the tensor field addition and subtraction operations taking a pair of tensor fields from $T^{(k)}_{F_{X\to R},E}$ to a tensor field in $T^{(k)}_{F_{X\to R},E}$.
– $a^*\tau$ is the scalar-tensor field multiplication taking $a \in T^{(0)}_{F_{X\to R},E}$ and $\tau \in T^{(k)}_{F_{X\to R},E}$ to a tensor field in $T^{(k)}_{F_{X\to R},E}$. Note that $a * \tau = a \otimes \tau$. Normally we write $a\tau$ for $a^*\tau$.
– $\sigma \cdot \tau$ is the tensor field dot product taking tensor fields $\sigma, \tau \in T^{(k)}_{F_{X\to R},E}$ and returning a scalar field in $T^{(0)}_{F_{X\to R},E}$.
– ∇ is the spatial derivative, which is used in several forms:
  1. $(\mathbf{v} \cdot \nabla)\sigma$, the convective derivative, yields a tensor field in $T^{(k)}_{F_{X\to R},E}$ for the derivation of the tensor field $\sigma \in T^{(k)}_{F_{X\to R},E}$ in the direction of the vector field $\mathbf{v} \in T^{(1)}_{F_{X\to R},E}$,
  2. $\nabla\sigma$, the gradient, yields a tensor field in $T^{(k+1)}_{F_{X\to R},E}$ when applied to a tensor field $\sigma \in T^{(k)}_{F_{X\to R},E}$, thus it takes a scalar field to a vector field and a vector field to a matrix field,
  3. $\nabla \cdot \sigma$, the divergence, yields a tensor field in $T^{(k)}_{F_{X\to R},E}$ when applied to a tensor field $\sigma \in T^{(k+1)}_{F_{X\to R},E}$, thus it takes a matrix field to a vector field and a vector field to a scalar field.
– $\int_\Omega \sigma dX$ integration of a tensor field $\sigma \in T^{(0)}_{F_{X\to R},E}$ over a subdomain $\Omega \subseteq X$ yields a ring element in R.

There is an important distinction between tensor fields $T^{(k)}_{F_{X\to R},E}$ based on the tensor construction using scalar fields as the ring, and tensor fields $T : X \to T^{(k)}_{R,B}$ which directly assign a tensor to every point $x \in X$. The former allow us to build advanced tensor operations, such as derivation operations, from scalar field operations, such as the partial derivatives. The latter require us to implement the advanced tensor operations directly in terms of the (discretised) representation of X, see the discussion in [14, Section 4.2]. As is evident, the former, which we have chosen, give a clear separation between discretisation methods for the scalar field and the spatial derivation operations at the tensor level. The latter force a tensor field implementation for each discretisation.

4.3. *A framework for a tensor based library*

The Sophus library framework describes a library architecture for providing the abstract mathematical concepts from PDE theory as programming entities. This means that any piece of a program, or even any module in the library, may be coded using any of the abstractions defined. At compile time, implementations for each of the abstractions will be chosen, such that no circular dependencies on the implementations occur. This means that, e.g., a mesh implementation may build on another mesh implementation, but that the latter mesh cannot be built on the former mesh. The Sophus framework is based on the notions of manifold, scalar field and tensor field, while the implementations are based on the conventional numerical algorithms and discretisations. The Sophus framework is structured around the following concepts:

- Basic n-dimensional mesh structures with a ring R as template argument. These are like rank n arrays (i.e., with n independent indices) with element type R, but with general map operations, i.e., performing an argument function for every element. It also has specific operations like $+$, $-$ and $*$ mapped over all elements (much like Fortran-90 array operators) as well as the ability to add, subtract or multiply all elements of the mesh by a scalar in a single operation. There are also operations for shifting meshes in one or more dimensions. Operations like multidimensional matrix multiplication @ and linear equation solvers such as Choleski decomposition and Jacobi iteration may easily be implemented for the meshes.
  Not all mesh implementations will provide all operations. Some implementations may be more specialised, e.g., assuming a sparse mesh or a mesh with certain symmetries. Other implementations may provide fully general parallel and sequential implementations that can be used interchangeably, allowing easy porting between computer architectures of any program built on top of the mesh abstraction.
- Manifolds X. These are sets with a notion of proximity and direction. They represent the physical space $\Omega \subseteq X$ where the problem to be solved takes place.
- Scalar fields $F_{X\to R}$. They describe the measurable quantities of the physical problem to be solved. As the basic layer of "continuous mathematics" in the library, they provide the partial derivation and integration operations. Also, two scalar fields on the same manifold may be pointwise added, subtracted and multiplied.
  The different discretisation methods, such as the finite difference and finite element methods, provide different designs for the implementation of scalar fields. Scalar fields are typically implemented using the mesh structures with reals for the ring to store the data. Not all mesh operations are relevant in this context, so it is possible to choose mesh implementations that, e.g., do not support equation solvers or matrix multiplication, when configuring implementations for a program.
- Tensors $T^{(k)}_{F_{X\to R},E}$. These provide coordinate free mathematics based on the knowledge of the coordinate system E, whether it is Cartesian, axisymmetric or general curvilinear. The tensor class provides the general differentiation and integration operations, based on the partial derivatives and integrals of the scalar fields. Tensors also provide operations such as componentwise addition, subtraction and multiplication, as well as tensor product, composition and application.
  The implementation is based on the basic mesh structures, with scalar fields as the ring parameter. Thus tensor operations are formed from expressions on scalar fields performed by the mesh classes. Again, many operations of the mesh are not needed, allowing more specialised mesh implementations to be used. For instance, equation solvers are not needed, while matrix multiplication algorithms are important.
- Equation administrators. These are abstractions containing collections of scalar and tensor fields with the purpose of building the matrices and vectors used to describe sets of linear equations, such as those needed for implicit time stepping schemes. These matrices and vectors do not represent coordinate free properties of a physical system, but abstract the important properties of linear equations.
  Equation administrators are also implemented using mesh structures with tensor fields or reals as the ring, as appropriate. Here operations like matrix multiplication and matrix equation solvers are important, and relevant mesh implementations must provide these. Also, additional properties like symmetries and block diagonal structures may be exploited by appropriate mesh implementations.

In general a partial differential equation provides a relationship between spatial derivatives of tensor fields representing physical quantities and their time derivatives. Given constraints in the form of the values of the tensor fields at a specific instance in time together with boundary conditions, the aim of a PDE solver is to show how the physical system will evolve over time, or what state it will converge to if left by itself. Using Sophus, the solvers are formulated on top of the coordinate-free layer, forming an abstract, high level program for the solution of the problem.

4.4. *The finite element method in Sophus*

The finite element method presented in Section 3.1, is now redeveloped in a more abstract manner suitable for implementation in the Sophus approach. In this more general setting, the method is based on the observation that given a manifold X with basis $B \subseteq X$ and a ring R, a scalar field $p \in F_{X\to R}$, may be ap-

proximated by a sum $p \approx \sum_{g\in G} P_g \cdot g$ for scalar fields $g \in G \subseteq F_{X\to R}$, termed *shape functions*, and scalars $P_g \in R$. The larger the set G, the better the approximation, but this will increase computation times since larger data sets will have to be computed. For numerical reasons, different scalar fields should be approximated by different shape function sets, but if these different discretisations are to be used in the same expressions, the different sets of shape functions must be coordinated.

This coordination is achieved by splitting the domain X into a set $\mathcal{E}$ of disjoint elements $e \in \mathcal{E}$, such that $\cup_{e\in\mathcal{E}}\, e = X$ and for each element designate a fixed set of integration points. A scalar field's value at an integration point represents its average value in a subregion of the element. In a 2-dimensional case the domain may typically be split into triangles, the choice adopted in Section 3.1. For the FEM, the shape functions are continuous within elements and have small support, i.e., are non-zero only for a few elements. This is normally restricted further, so that the functions $g \in G$ take their maximum value $1 \in R$ at exactly one point – the nodal point – in the domain X, and are non-zero only on those elements adjacent to that point (referred to as the domain of local compact support for g). Also, a shape function is 0 at the nodal points of all other shape functions within its collection. These are the same constraints as used in the conventional approach presented in Section 3.1.

The shape functions $g \in G$ are normally chosen so that the partial derivatives $\frac{\partial g}{\partial \mathbf{x}} \in F_{X\to R}$ for $\mathbf{x} \in B \subseteq X$ and the integral $\int_e g dX \in R$ on an element $e \subseteq X$ may be computed analytically. In Section 3.1, for example, these were chosen to be either linear or quadratic functions. Since the differentiation and integration operations are linear with respect to R, we have that

$$\frac{\partial p}{\partial \mathbf{x}} \approx \sum_{g\in G} P_g \cdot \frac{\partial g}{\partial \mathbf{x}}$$

$$\int_\Omega p dX \approx \sum_{e\in\mathcal{E}} \left(\sum_{g\in G} (P_g \cdot \int_e g dX) \right)$$

In the variational form, all expressions involving the scalar fields are integrated in each PDE equation. This means we do not need to approximate the scalar field or tensor field expressions as such, but rather their effect on the integrals. The values at the integration points provide such an approximation, so these are the only values we really need to use when computing scalar field expressions. This also implies that elements and integration points are the only coordination needed between scalar fields. Each scalar field may be based on different sets of shape functions for that matter.

4.5. Developing a solver for the coating problem

Recall, that for the coating problem we are working with a subdomain $\Omega \subseteq X$ of the physical 3-dimensional world X. The tensor fields are $T^{(k)}_{F_{X\to R},E}$, where R is the set of reals, $E = \{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\} \subseteq V_{X\to R}$ are the standard basis vectors, and $k = 0$ (scalar fields), $k = 1$ (vector fields) or $k = 2$ (matrix fields). In the variational form of the equations for the solver in Section 2.6 we are using scalar fields $q \in Q \subseteq T^{(0)}_{F_{X\to R},E}$ and vector fields $\mathbf{v} \in V \subseteq T^{(1)}_{F_{X\to R},E}$ as test functions for the integrals.

The coordinate free, variational form, of the solver equation steps Eqs (17)–(21) for the coating problem need some refinement to serve as an algorithm for an actual computation. Looking at the left hand sides of the equations we see there are three problems to address:

- The integrals compound the value of scalar fields into a scalar, so we need to restore the unknown scalar fields on the left hand side from these scalars (steps 1, 2 and 3).
- The vector fields representing the velocity are "dotted" with a vector field, so only information about the magnitude of the left hand side vectors is known, with no information about the direction (steps 1 and 3).
- The symmetrisations of $\nabla\mathbf{u}$ are "dotted" with $\nabla\mathbf{v}$ terms, thus intermixing information concerning all components of the vector $\mathbf{u}$, so that not even the magnitude of the unknown vectors are explicitly available (step 1).

The above is a more abstract view than that indicated in Section 2.6. But, as in the conventional case, the solution to these problems is to generate more equations at each step, by choosing appropriate sets of test functions, so that we get enough scalars to compute the unknown scalar and vector fields. How to achieve this becomes increasingly involved. We relate the techniques to the individual steps of the algorithm, starting with the simplest cases.

- Step 2: a scalar field integrated to a real value. If we approximate the scalar-field $(p^{n+1} - p^n) \in T^{(0)}_{F_{X\to R},E}$ by a sum $\sum_{g\in G}(P^{n+1}_g - P^n_g) \cdot g$ for

scalars $(P_g^{n+1} - P_g^n) \in R$ and shape functions $G \subseteq T^{(0)}_{F_{X\to R},E}$, we may move the scalars out of the derivation and integration expressions. We may then reformulate step 2 to

$$\forall q \in Q \left[\theta \sum_{g\in G} (P_g^{n+1} - P_g^n) \left(\int_\Omega \nabla g \cdot \nabla q dX \right) = \frac{-Re}{\Delta t} \int_\Omega (\nabla \cdot \mathbf{u}^*) q dX \right].$$

This gives us a system of equations with $|Q|$ right hand sides[3]

$$B_q = \frac{-Re}{\Delta t} \int_\Omega (\nabla \cdot \mathbf{u}^*) q dX$$

for each $q \in Q$. It has $|G|$ unknowns $(P_g^{n+1} - P_g^n)$ for $g \in G$. The $|Q| \times |G|$ integrals on the left hand side are independent of the variables of the problem, so we may define a mesh matrix $K_{q,g} = \int_\Omega \nabla g \cdot \nabla q dX$ for $g \in G, q \in Q$ (this is a matrix with data elements from R, and is not a tensor structure but just a mesh-of-R structure). This corresponds to the pressure stiffness matrix $\mathbf{K}$ from Eq. (29). Ensuring that $|Q| = |G|$, the unknowns are uniquely determined, and we may solve the system $\theta K @ (P^{n+1} - P^n) = B$ using a suitable matrix solver.
Assuming that P_g^n is known it is easy to find P_g^{n+1} once the system is solved and the $(P_g^{n+1} - P_g^n)$, for $g \in G$, have been found. Calculating the scalar field $p^{n+1} \in T^{(0)}_{F_{X\to R},E}$ will be simplified if G is taken to be the shape functions used in the discretisation of the scalar fields $p \in F_{X\to R}$ as described in Section 4.4. This also ensures that we will not lose any accuracy by the approximations of p^{n+1} in the system of equations.

– Step 3: a vector field compounded to scalar field and then integrated to a real value. We may represent the unknown vector $(\mathbf{u}^{n+1} - \mathbf{u}^*) \in T^{(1)}_{F_{X\to R},E}$ as the linear combination $u_1^*\mathbf{e}_1 + u_2^*\mathbf{e}_2 + u_3 * \mathbf{e}_3$ of basis vectors $e_1, e_2, e_3 \in E \subseteq T^{(1)}_{F_{X\to R},E}$ and scalar fields $u_1, u_2, u_3 \in T^{(0)}_{F_{X\to R},E}$. Then the equation becomes

$$\forall \mathbf{v} \in V \left[\frac{Re}{\Delta t} \sum_{i=1}^{3} \left(\int_\Omega u_i * \mathbf{e}_i \cdot \mathbf{v} dX \right) = \theta \int_\Omega (p^{n+1} - p^n)(\nabla \cdot \mathbf{v}) dX \right].$$

Now we can restore the scalar fields using the technique of shape functions $g \in G'$ as above, giving the following system of equations for each $\mathbf{v} \in V$

$$\frac{Re}{\Delta t} \sum_{g\in G'} \left(\sum_{i=1}^{3} \left(U_{(g,i)}^{n+1} - U_{(g,i)}^* \right) \left(\int_\Omega g e_i \cdot v dX \right) \right) = \theta \int_\Omega (p^{n+1} - p^n)(\nabla \cdot \mathbf{v}) dX.$$

We choose G' as the shape functions for the discretisation of the scalar field components for $\mathbf{u} \in T^{(2)}_{F_{X\to R},E}$. Then we ensure that $|V| = 3|G'|$ so that we have the same number of equations $|V|$ as unknowns, which is $|G'|$ times the number of dimensions. This time the matrix on the left hand side becomes $M_{\mathbf{v},(g,i)} = \int_\Omega g\mathbf{e}_i \cdot \mathbf{v} dX$, for $g \in G'$, $i \in \{1,2,3\}$ and $\mathbf{v} \in V$. We need to treat (g,i) as one index in order to have M be a normal matrix. This corresponds to the mass matrix $\mathbf{M}$ from Eq. (22).

– Steps 1a and 1b: symmetrisation with vector field compounded to scalar field and then integrated to a real value. Using the same technique as above, we may easily separate the unknowns from the first (explicit) part of the left hand side terms. This also separates the unknowns from the symmetrised term since symmetrisation is linear with respect to scalar multiplication. Then we may reformulate step 1a to, $\forall \mathbf{v} \in V$

$$\sum_{g\in G'} \left(\sum_{i=1}^{3} \left(U_{(g,i)}^{n+1/2} - U_{(g,i)}^n \right) \int_\Omega \left(\frac{2Re}{\Delta t} g\mathbf{e}_i \cdot \mathbf{v} + \mu\, \mathsf{sym}(\nabla(g\mathbf{e}_i)) \cdot \nabla \mathbf{v} \right) dX \right) = -2 \int_\Omega \mu\, \mathsf{sym}(\nabla \mathbf{u}^n) \cdot (\nabla \mathbf{v}) dX - Re \int_\Omega ((\mathbf{u}^n \cdot \nabla)\mathbf{u}^n) \cdot \mathbf{v} dX + \int_\Omega p^n (\nabla \cdot \mathbf{v}) dX$$

with equation matrix $M' = \frac{2Re}{\Delta t} M + S$, and step 1b to, $\forall \mathbf{v} \in V$:

[3]For a set X the notation $|X|$ means the cardinality of X.

$$\sum_{g\in G'}\left(\sum_{i=1}^{3}\left(U^*_{(g,i)}-U^n_{(g,i)}\right)\right.$$

$$\left.\int_\Omega\left(\frac{Re}{\Delta t}g\mathbf{e}_i\cdot\mathbf{v}+\mu\,\mathsf{sym}(\nabla(g\mathbf{e}_i))\cdot\nabla\mathbf{v}\right)dX\right)$$

$$=-2\int_\Omega\mu\,\mathsf{sym}(\nabla\mathbf{u}^n)\cdot(\nabla\mathbf{v})dX$$

$$-Re\int_\Omega((\mathbf{u}^{n+1/2}\cdot\nabla)\mathbf{u}^{n+1/2})\cdot\mathbf{v}dX$$

$$+\int_\Omega p^n(\nabla\cdot\mathbf{v})dX$$

with equation matrix $M''=\frac{Re}{\Delta t}M+S$, where $S_{\mathbf{v},(g,i)}=\int_\Omega\mu\,\mathsf{sym}(\nabla(g\mathbf{e}_i))\cdot\nabla\mathbf{v}dX$. Here S is the diffusion matrix S from Eq. (23). This provides us with augmented matrices compared with step 3, but otherwise with the same number of equations and the same number of unknowns as in step 3 above.

Note that we do not need to build counterparts to the incompressibility matrix $\mathbf{L}$ in Eq. (25) nor the convection matrix $\mathbf{N}$ in Eq. (24). This is possible since those represent right hand sides, and thus are implied by the tensor expressions in this approach.

We will use the Galerkin simplification on the resulting systems of equations by choosing scalar field test functions $Q=G\subseteq T^{(0)}_{F_{X\to R},E}$ and vector field test functions $V=\{g*\mathbf{e}|g\in G',\mathbf{e}\in E\}\subseteq T^{(1)}_{F_{X\to R},E}$. Normally, we will not choose $G=G'$, as there are different numerical constraints on the pressure p and the velocity $\mathbf{u}$.

Combining this information, we arrive at the final coordinate free algorithm for the coating problem. The collections of shape and test functions may be kept in a mesh data structure. We may then use the mesh map operations to generate the right hand sides and left hand side matrices. This eliminates the need for explicit loops for the generation of the equations. Then we employ the matrix solvers written for the mesh classes to find the unknowns. The algorithm repeats the following steps until time-stepping convergence criteria are met, given initial values for U^n and P^n:

`Calculate` $\mathbf{u}^n$ `from` U^n`;`
`Calculate` p^n `from` P^n`;`
`Step 1a: solve for` $U^{n+1/2}-U^n$ `in`

$$M'@(U^{n+1/2}-U^n)$$

$$=\left(\int_\Omega -2\mu\,\mathsf{sym}(\nabla\mathbf{u}^n)\cdot(\nabla\mathbf{v})\right.$$

$$\left.-Re((\mathbf{u}^n\cdot\nabla)\mathbf{u}^n)\cdot\mathbf{v}+p^n(\nabla\cdot\mathbf{v})dX\right)_{\mathbf{v}\in V}$$

`Calculate` $\mathbf{u}^{n+1/2}$ `from` $U^{n+1/2}-U^n$, U^n `and` G'`;`
`Step 1b: solve for` U^*-U^n `in`

$$M''@(U^*-U^n)$$

$$=\left(\int_\Omega -2\mu\,\mathsf{sym}(\nabla\mathbf{u}^n)\cdot(\nabla\mathbf{v})\right.$$

$$-Re((\mathbf{u}^{n+1/2}\cdot\nabla)\mathbf{u}^{n+1/2})\cdot\mathbf{v}$$

$$\left.+p^n(\nabla\cdot\mathbf{v})dX\right)_{\mathbf{v}\in V}$$

`Calculate` $\mathbf{u}^*$ `from` U^*-U^n, U^n `and` G'`;`
`Step 2: solve for` $P^{n+1}-P^n$ `in`

$$\theta K@(P^{n+1}-P^n)$$

$$=\left(\int_\Omega\frac{-Re}{\Delta t}(\nabla\cdot\mathbf{u}^*)qdX\right)_{q\in Q}$$

`Calculate` p^{n+1} `from` $P^{n+1}-P^n$, P^n `and` G`;`
`Step 3: solve for` $U^{n+1}-U^*$ `in`

$$\frac{Re}{\Delta t}M@(U^{n+1}-U^*)$$

$$=\left(\int_\Omega\theta(p^{n+1}-p^n)(\nabla\cdot\mathbf{v})dX\right)_{\mathbf{v}\in V}$$

`Set` U^n `as` $U^{n+1}-U^*$ `plus` U^* `and ensuring boundary condition;`
`Set` P^n `as` $P^{n+1}-P^n$ `plus` P^n `and ensuring boundary condition;`

Recall that the velocity values at the boundaries are prescribed, hence we must ensure that these values remain unchanged at every step. The matrix K will be banded and sparse and the equations in step 2 can be solved using Choleski decomposition. The matrices M, M' and M'' are very large. With a careful choice of elements and using orthogonal basis vectors, i.e., $i\neq j$ implies $\mathbf{e}_i\cdot\mathbf{e}_j=0$, matrix M can be reduced to a banded form. Jacobi iteration will be a useful technique for solving the equations in steps 1 and 3. The element-by-element construct and solve procedure and matrix conditioning provide such a choice.

For this problem appropriate choices for test and shape functions are

- for $Q = G$: functions which reside at the vertices of the elements, and are linear within each element, and
- for $V = G'$: functions which reside at the vertices and mid-points of the edges of the elements, and are quadratic within each element.

Note that the mesh classes are used for many distinct purposes in the solver:

- in the implementation of the scalar field,
- in the implementation of the tensors,
- for storing the set of test functions, and
- for data matrices K and M, M' and M''.

In a configuration of the solver program different implementations of the Mesh class may be chosen for each of these different purposes in order to reduce storage requirements, improve run-time efficiency and provide parallel execution.

Testing for convergence is performed using the L^2 norm for velocities and pressure.

Coding this algorithm as a computer program is now straightforward if a library with the concepts of the Sophus framework is available. We have not fully implemented this application in Sophus yet, but based on experience with this framework [16] the following seems reasonable: The PDE solver would be written as a procedure, and based on the detailed exposition above this should only be about 100–200 lines of code, including test for termination. Additional procedures to input data sets, set up the data and output the results will increase the solver module code size to about 1,000 lines of code. The code needed for a simple user interface, I/O file handling and configuration of the PDE solver typically lies around 1000 lines of code as well. Thus a complete solver and configuration may be written in less than 2000 lines of code.

4.6. *A discussion of relevant coding techniques*

The coding technique we advocate is based on notions of data abstraction and encapsulation. These may take the form of the class construct in object-oriented languages like C++ [28], Eiffel [23], GJ [5] and Java [12], type abstraction and functors in applicative functional languages like standard ML and Haskell, or packages in imperative languages like Ada and Fortran-90. The reuse of modules such as Mesh in both the implementation of scalar fields and the implementation of tensors requires template classes or generic packages, as present in Ada, C++, Eiffel, standard ML, Haskell and GJ.

The structuring mechanism does not force any specific coding practice for implementing the algorithms. Thus both applicative styles, as supported by functional languages, and more conventional styles that modify variables for reuse of storage may be used. The latter encompasses imperative styles, which are typical of Fortran, Ada and C++, and object-oriented styles, which are supported by C++, Eiffel, GJ and Java. Within each group one may favour languages which allow operators and overloading. This supports a more algebraic notation by making it possible to define scalar field and tensor operations with infix syntax and names like $+$, $*$, $-$, $/$ etc. Support for this can be found in diverse languages as C++, standard ML and Fortran-90. Only standard ML allows user defined infix operator names, the other languages only support a limited set of names which is quickly exhausted by the plentitude of tensor level unary and binary operators.

We have developed Sophus using C++ in an imperative, object-oriented manner. This means that the programmer may have full control over creation of temporary variables and reuse of storage by modifying the values of variables. This style tends to favour machine efficiency. The development of the coordinate free algorithm above has an imperative flavour in its sequencing of operations and iteration over the main equations for the PDE solver. Sophus also allows algebraic style expressions by utilising the operator overloading permitted by languages like C++ and Fortran-90. This seems to have a negative effect on execution time efficiency, but provides a greater ease of programming which may improve software development efficiency. Sophus can easily be reimplemented in other languages which support the necessary abstraction mechanism. If the emerging Fortran-2000 supports templates it would seem a suitable language for this style of programming.

The tensor oriented package RHALE++ [8,34] is also implemented in C++ using an object-oriented, imperative style. This package differs from Sophus by implementing vector and tensor fields directly on the manifold, instead of lifting the scalar field. The package Overture [6] does not provide tensor abstractions but provides the scalar field abstractions (the continuous level). Compose [1,2] adds equation handlers on top of these, but the tensor level is still lacking.

5. Modifying the problem

The quality of a software development methodology and programming style can best be evaluated by check-

ing how easy it is to modify and upgrade programs. For the coating problem one such modification is the change of coordinate system, motivated by the annular nature of the problem. Using cylindrical coordinates the data sets may be reduced to two-dimensional scalar fields. Cylindrical coordinates have z-axis at the centre of the wire, r-axis as radial distance from the z-axis, and θ-axis as azimuthal rotation angle of an rz-plane. The data fields will vary only along the r- and z-axis, being constant along the θ-axis. Thus all partial derivatives with respect to θ vanish, and there is no need to store information for this axis.

5.1. Conventional case

We now describe the formulation using cylindrical coordinates, where we take an axi-symmetric geometry which would be a typical situation for the coating problem. A concentricity assumption is adopted for the particular coating problem being considered, so $u = (u_r, 0, u_z)$. The integration over the domain becomes specific for any particular geometry, so

$$\int_\Omega f(\mathbf{x})g(\mathbf{x})d\Omega \equiv 2\pi \int_r \int_z f(r,z)g(r,z) r dz dr,$$

and similarly for the vector and tensor inner products. In cylindrical coordinates, using $\mathbf{v}$ for a vector and s for a scalar, and the particular assumptions of the problem,

$$\nabla \cdot \mathbf{v} = \frac{1}{r}\frac{\partial}{\partial r}(r v_r) + \frac{\partial v_z}{\partial z},$$

$$\nabla s = \boldsymbol{\delta}_\mathrm{r}\frac{\partial s}{\partial r} + \boldsymbol{\delta}_\mathrm{z}\frac{\partial s}{\partial z},$$

where $\boldsymbol{\delta}_\mathrm{r}$ and $\boldsymbol{\delta}_\mathrm{z}$ are unit vectors in the r and z directions, respectively, and

$$\nabla = \boldsymbol{\delta}_\mathrm{r}\frac{\partial}{\partial r} + \boldsymbol{\delta}_\theta \frac{1}{r}\frac{\partial}{\partial \theta} + \boldsymbol{\delta}_\mathrm{z}\frac{\partial}{\partial z}. \qquad (32)$$

With the above definitions, the operators in the variational formulation of the problem Eq. (17)–(20) can be replaced to yield a formulation in cylindrical coordinates. For example the dyad $\nabla\mathbf{u}$ can be evaluated, using Eq. (32) and properties of unit vectors $\boldsymbol{\delta}_\mathrm{r}$, $\boldsymbol{\delta}_\theta$ and $\boldsymbol{\delta}_\mathrm{z}$, to be:

$$\nabla \mathbf{u} = \begin{bmatrix} \frac{\partial u_r}{\partial r} & 0 & \frac{\partial u_z}{\partial r} \\ 0 & \frac{u_r}{r} & 0 \\ \frac{\partial u_r}{\partial z} & 0 & \frac{\partial u_z}{\partial z} \end{bmatrix}. \qquad (33)$$

By expanding the $\mathbf{U}_j$ in terms of the base vectors $\boldsymbol{\delta}_\mathrm{r}$, $\boldsymbol{\delta}_\theta$ and $\boldsymbol{\delta}_\mathrm{z}$ and using as test functions $\boldsymbol{\varphi}_i^r = \varphi_i \boldsymbol{\delta}_\mathrm{r}$ and $\boldsymbol{\varphi}_i^z = \varphi_i \boldsymbol{\delta}_\mathrm{z}$ we arrive at the following expression for the $\mathbf{S}$ matrix:

$$\mathbf{S} = \begin{bmatrix} \mathbf{S}^{rr} & \mathbf{S}^{rz} \\ (\mathbf{S}^{rz})^T & \mathbf{S}^{zz} \end{bmatrix}$$

where

$$\mathbf{S}_{ij}^{lm} = \int_\Omega \mu sym \nabla \boldsymbol{\varphi}_j^l \nabla \boldsymbol{\varphi}_j^m d\Omega \qquad (34)$$

and $l, m \in \{r, z\}$. Using (33) we have

$$sym\nabla\boldsymbol{\varphi}^r = \begin{bmatrix} \frac{2\partial\varphi}{\partial r} & 0 & \frac{\partial\varphi}{\partial z} \\ 0 & \frac{2\varphi}{r} & 0 \\ \frac{\partial\varphi}{\partial z} & 0 & 0 \end{bmatrix}$$

and

$$sym\nabla\boldsymbol{\varphi}^z = \begin{bmatrix} 0 & 0 & \frac{\partial\varphi}{\partial r} \\ 0 & 0 & 0 \\ \frac{\partial\varphi}{\partial r} & 0 & 2\frac{\partial\varphi}{\partial z} \end{bmatrix}.$$

Substituting in Eq. (34) we can express the components of $\mathbf{S}$ as follows:

$$\mathbf{S}_{ij}^{rr} = 2\pi \int_r \int_z \left[2\frac{\partial\varphi_i}{\partial r}\frac{\partial\varphi_j}{\partial r} + 2\frac{\varphi_i\varphi_j}{r^2} + \frac{\partial\varphi_i}{\partial z}\frac{\partial\varphi_j}{\partial z} \right] r dz dr,$$

$$\mathbf{S}_{ij}^{rz} = 2\pi \int_r \int_z \left[\frac{\partial\varphi_i}{\partial r}\frac{\partial\varphi_j}{\partial z} \right] r dr dz,$$

$$\mathbf{S}_{ij}^{zz} = 2\pi \int_r \int_z \left[2\frac{\partial\varphi_i}{\partial z}\frac{\partial\varphi_j}{\partial z} + \frac{\partial\varphi_i}{\partial r}\frac{\partial\varphi_j}{\partial r} \right] r dz dr.$$

In a similar way, the remaining system matrices, defined in axi–symmetric cylindrical coordinates are given by;

$$\mathbf{M}_{ij} = 2\pi \int_r \int_z [\varphi_i \varphi_j [r dz dr, \qquad (35)$$

$$\mathbf{N}(V)_{ij} = 2\pi \int_r \int_z \varphi_i \sum_{\ell=1}^{m_v} \left[\varphi_l V_r^\ell \frac{\partial\varphi_j}{\partial r} + \varphi_l V_z^\ell \frac{\partial\varphi_j}{\partial z} \right] r dz dr, \qquad (36)$$

$$\mathbf{K}_{kl} = 2\pi \int_r \int_z \nabla\psi_k \cdot \nabla\psi_l r dz dr, \qquad (37)$$

$$\mathbf{L}_{li}^1 = 2\pi \int_r \int_z \psi_l \frac{\partial\phi_i}{\partial r} r dz dr, \quad \mathbf{L}_{li}^2 = 2\pi \int_r \int_z \psi_l \frac{\partial\varphi_i}{\partial z} r dz dr. \qquad (38)$$

Here V_r^ℓ and V_z^ℓ are the nodal velocity components in radial (r) and axial (z) directions respectively.

5.2. *Coordinate free case*

The solver for the coating problem was presented in a coordinate free notation in Section 4.5. This means that the solver is independent of coordinate system, and need not be changed when moving to cylindrical coordinates. However, the configuration must be altered to provide the tensor class with the definition of the appropriate coordinate system and the scalar fields must be reduced to 2-dimensional form. These changes will only affect a few lines in the configuration module, assuming that the necessary modules are present in the Sophus library.

The changes in configuration sketched above may not be sufficient to gain optimal speed for the axi-symmetric problem. The reason is that, unless the tensor class has been optimised for axi-symmetry, it will still activate all the computations of a 3-dimensional problem. This can be reduced if a specific axi-symmetric version of the tensor class is implemented. Such an implementation may take 1000–4000 lines of code, but need not be written from scratch.

6. Discussion

In this paper, we have presented two different approaches to developing numerical software. The first, the conventional methodology, is that followed by the majority of the numerical programming and applied mathematical community. The other approach, advocated here, is an abstraction method using coordinate free mathematics.

We can view the two methods as indicated in Fig. 2. Both methods start with the mathematical formulation on the left. The conventional method then drops down to the machine level (left downward arrow and bottom horizontal arrow). In the coordinate free method, all development takes place at the abstract level, and the library modules link down to the machine level (top horizontal arrow and dotted right vertical arrow).

We observe here one symptom of a cultural divide between the field of programming theory and numerical analysis. This divide does not simply depend on the individual problems each community normally addresses, but goes deeper, and depends on the way we reason about problem solving and programming. This can be seen in the different methodological approaches to solving a complex problem like the coating problem and its implementation using the finite element method (FEM), as illustrated in this paper. This indicates that for the abstraction method to be accepted by the numerical community would require new training and instruction. Thus only a gradual transition to coordinate free numerics and other abstraction oriented methodologies is to be expected.

From a programming theory viewpoint, there is a definite need to present the coating problem at the abstract level as far as possible. Only after all the technical details have been exposed at that level, should the discretisation technique be introduced as an orthogonal issue. Here the FEM should be exposed algorithmically, not solely on its mathematical merits as an approximation technique. If this is done properly, the change of coordinate systems, such as switching from Cartesian to axi-symmetric will be orthogonal to both the detailed exposition of the abstract mathematical algorithm and the discretisation technique.

The two methodological approaches to the presentation of the problem as discussed in this paper, are exposed in the manner the software development is handled.

Conventional software development in the field of computational modelling typically commences with some partial differential equation (PDE). This is then refined into an abstract algorithm, and then experience and insight is used to transform to a discretised version of the algorithm. Further refinement takes place at the discrete level, and the language being used allows for the elementary data types of arrays and matrices. We then arrive at a sequential program, that may be further refined into a parallel program.

The software development methodology we propose would also start with the step of converting from a PDE to an abstract algorithm. The departure then lies in the further developments that would stay at a mathematical level, yielding an abstract program. This may then be linked together with a software library such as Sophus, yielding either a sequential or a parallel program without any further significant modification.

The two different development strategies give vastly different software characteristics. First, consider the relative sizes of code produced by the two approaches. The conventional Fortran code for our case study totals approximately 18,000–19,000 lines of code. In contrast, the exposition of the coating program and its detailed pseudocode is well below 1,000 lines of code, with an estimate of the size of the final coded applica-

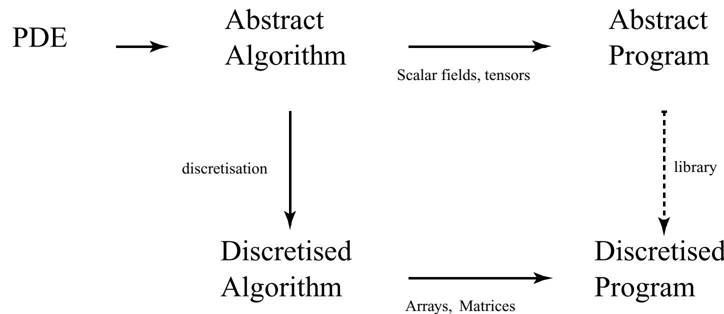

Fig. 2. Coordinate free versus conventional methodology.

tion program being approximately 2,000 lines.[4] Taking lines of code as a measurement of the development costs, which is the basis for cost estimating models like COCOMO [4], we find an overwhelming reduction in code development costs for the coordinate free methodology compared to the traditional methodology.

Second, consider the modifiability and adaptivity of the resulting software. The conventional development produces one application, and reuse of components from this software will be incidental. The Sophus Library framework is designed for reuse and to be incrementally implemented. Basically the solver for a new problem relies on the concepts defined by the Sophus library interface. When configuring the program, relevant modules from the library are reused, but if the library lacks an implementation with certain characteristics, such a code may be developed and integrated into the library. The cost of implementing a discretisation from scratch, i.e., defining the manifold with associated point set and scalar field, we estimate as requiring approximately 4,000 lines of code [16]. Adding a new discretisation technique for an *existing* manifold corresponds to developing only 1,000–2,000 lines of code. It should also be recalled that all implementation of the same abstraction, such as that of scalar fields, have the same interface. So that given two different scalar field discretisations, we may interchange them within the same application program with little adaptation.

The above observations and statistics indicate that the abstraction oriented methodology promoted in this paper may well improve computational modelling productivity dramatically. An added bonus is that such an approach supports easy transition between sequential and parallel versions of the code [15].

Traditionally, applied mathematicians and numerical analysts have been sceptical in adopting programming languages other than Fortran. This is mainly due to a fear of efficiency loss in their codes. This no longer seems the case, as a language such as C++ and the use of abstractions in many cases, has been shown to be comparable in efficiency to Fortran, see [3,25,32]. This is deemed highly encouraging for emerging abstraction oriented implementations of numerical solvers. Unfortunately, the resulting efficiency seems sensitive to memory layout and other factors which are difficult to control. However, a source-level transformation tool, such as CodeBoost [9], can be invaluable in this respect. It makes it possible to systemise experiments with various data layouts and other transformations of the code. It is clear that several pilot implementations, with execution speed comparable to conventional Fortran code, of different problems is needed to convince a larger proportion of the numerical community of the benefits of abstraction oriented methodologies.

A further open question, clearly relevant to the practitioner, is to what extent this manner of writing programs affects numerical error propagation. There is no reason to expect it to be worse than for conventionally developed programs, but the ease with which one may change discretisation technique may lead to situations where an inappropriate discretisation technique is being used. One possibility to prevent this from happening, is to provide the scalar fields with some "certificate" of their numerical properties at the abstract level.

[4]This comparison may seem quite unfair since we are comparing unstructured Fortran code without the use of libraries with estimates of highly structured C++ code using library modules. The comparison is still relevant, as we are comparing the *outcome* of two different development methodologies. We are *not* discussing whether code can be structured in one language or not in another, nor the general availability of libraries and how these may be used in different languages.

References

[1] K. Åhlander, An extendable PDE solver framework, Technical report, Dept. of Scientific Computing, Uppsala University, Uppsala, Sweden, 1997.

[2] K. Åhlander, An extendable PDE solver with reusable components, in: *ASME 2nd International Symposium on Computational Technologies for Fluid/Thermal/Structural/ Chemical Systems with Industrial Applications*, V.V. Kudriavtsev and C.R.K. Satoyuki Kawano, eds, volume 397-1 of *PVP* (*Pressure Vessels and Piping*), American Society of Mechanical Engineers, New York, NY, 1999, pp. 39–46.

[3] E. Arge, A.M. Bruaset, P.B. Calvin, J.F. Kanney, H.P. Langtangen and C.T. Miller, On the numerical efficiency of C++ in scientific computing, in: *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito, eds, Birkhäuser, Boston, 1997, pp. 91–118.

[4] B.W. Boehm, *Software Engineering Economics*, Prentice-Hall, 1981.

[5] G. Bracha, M. Odersky, D. Stoutamire and P. Wadler, Making the future safe for the past: Adding genericity to the Java programming language, in: *Proceedings of the Conference on Object Oriented Programming Systems, Languages, and Applications* (*OOPSLA '98*), 1998, See also URL http://-www.cs.bell-labs.com/˜wadler/pizza/gj/.

[6] D. Brown and W. Henshaw, Overture: An advanced object-oriented software system for moving overlapping grid computations, Technical Report LA-UR-96-2931, Los Alamos National Laboratory, 1996.

[7] A.M. Bruaset and H.P. Langtangen, A comprehensive set of tools for solving partial differential equations; Diffpack, in: *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito, eds, Birkhäuser, Boston, 1997, pp. 61–90.

[8] K.G. Budge, J.S. Peery and A.C. Robinson, High-performance scientific computing using C++, in: *USENIX* C++ *Technical Conference Proceedings*, USENIX Association, August 1992, pp. 131–150.

[9] T.B. Dinesh, M. Haveraaen and J. Heering, An algebraic programming style for numerical software and its optimisation, *Scientific Programming* **8**(4) (2000), 247–259.

[10] D. Ding, P. Townsend and M.F. Webster, Iterative solutions of Taylor-Galerkin augmented mass matrix equations, *Int. J. Num. Meth. Eng.* **35** (1992), 241–253.

[11] J. Donea, A Taylor-Galerkin method for convective transport problems, *Int. J. Numer. Meth. Engng.* **20** (1984), 101–119.

[12] J. Gosling, B. Joy and G. Steele, *The Java Language Specification*, Addison-Wesley, 1996.

[13] P.W. Grant, J.A. Sharp, M.F. Webster and X. Zhang, Experiences of parallelising finite element problems in a functional style, *Software – Practice and Experience* **25**(9) (Sep. 1995), 947–974.

[14] M. Haveraaen, Case study on algebraic software methodologies for scientific computing, *Scientific Programming* **8**(4) (2000), 261–273.

[15] M. Haveraaen, Machine and collection abstractions for user-implemented data-parallel programming, *Scientific Programming* **8**(4) (2000), 231–246.

[16] M. Haveraaen, H.A. Friis and T.A. Johansen, Formal software engineering for computational modeling, *Nordic Journal of Computing* **6**(3) (1999), 241–270.

[17] M. Haveraaen, V. Madsen and H. Munthe-Kaas, Algebraic programming technology for partial differential equations, in: *Proceedings of Norsk Informatikk Konferanse NIK'92*, Tapir, Norway, 1992, pp. 55–68.

[18] D.M. Hawken, H.R. Tamaddon-Jahromi, P. Townsend and M.F. Webster, A Taylor-Galerkin-based algorithm for viscous incompressible flow, *Int. J. Num. Meth. Fluids* **10** (1990), 327–351.

[19] D.M. Hawken, P. Townsend and M.F. Webster, Numerical simulation of viscous flows in channels with a step, *Computers Fluids* **20**(1) (1991), 59–75.

[20] T.R. Hopkins, Is the quality of numerical subroutine code improving? in: *Modern Software Tools for Scientific Computing*, Arge, Bruaset, and Langtangen, eds, Birkhäuser, Boston, 1997, pp. 311–324.

[21] S. Lang, *Algebra*, Addison Wesley, 1995.

[22] H. Matallah, P. Townsend and M.F. Webster, Recovery and stress-splitting schemes for viscoelastic flows, *J. Non-Newtonian Fluid Mechanics* **75** (1998), 139–166.

[23] B. Meyer, *Eiffel: The Language*, Prentice-Hall, 1992.

[24] H. Munthe-Kaas and M. Haveraaen, Coordinate free numerics – closing the gap between 'pure' and 'applied' mathematics? *Zeitschrift für Angewandte Mathematik und Mechanik, ZAMM* **76**(1) (1996), 487–488. Special issue ICIAM/GAMM 95, "Numerical Analysis, Scientific Computing, Computer Science", Proceedings of ICIAM'95, The Third International Congress on Industrial and Applied Mathematics.

[25] A.D. Robison, C++ gets faster for scientific computing, *Computers in Physics* **10**(5) (1996), 458–462.

[26] R.S. Schreiber, An introduction to HPF, in: *The Data Parallel Programming Model*, G.-R. Perrin and A. Darte, eds, volume 1132 of *Lecture Notes in Computer Science*, Springer, 1996, pp. 27–44.

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

[28] B. Stroustrup, *The* C++ *Programming Language*, (3rd ed.), Addison-Wesley, 1997.

[29] B.K. Szymanski, *Functional programming applied to numerical problems*, ACM Press, New York / Addison Wesley, Reading, Mass, 1991.

[30] P. Townsend and M.F. Webster, An algorithm for the three dimensional transient simulation of non-newtonian fluid flow, in: *Transient/Dynamic Analysis and Constitutive Laws for Engineering Materials*, G. Pande and J. Middleton, eds, 1987. T12/1.

[31] J. van Kan, A second-order accurate pressure correction scheme for viscous incompressible flow, *SIAM J. Sci Stat Comput* **6**(9) (1986), 870–891.

[32] T.L. Veldhuizen and M.E. Jernigan, Will C++ be faster than Fortran, in: *Scientific Computing in Object-Oriented Parallel Environments*, volume 1343 of *Lecture Notes in Computer Science*, Y. Ishikawa, R.R. Oldehoeft, J.V. Reynders and M. Tholburn, eds, Springer, 1997, pp. 49–56.

[33] R.D. Williams and S. Karmesin, *MAPS: An efficient parallel language for scientific computing,* Technical report, Caltech, 1992.

[34] M.K.W. Wong, K.G. Budge, J.S. Peery and A.C. Robinson, Object-oriented numerics: A paradigm for numerical object-oriented programming, *Computers in Physics* **7**(5) (1993), 655–663.

[35] O.C. Zienkiewicz, R. Löhner, K. Morgan and J. Peraire, High speed compressible flow and other advection dominated problems of fluid mechanics, in: *Finite Elements in Fluids,* (Vol. 6), (Chapter 2) R.H. Gallagher, G.F. Carey, J.T. Ogden and O.C. Zienkiewicz, eds, Wiley, 1985, pp. 41–88.