Tensor Abstraction Programming of Computational Fluid Dynamics Problems*

Phil Grant
Department of Computer Science
University of Wales Swansea

Magne Haveraaen†
Department of Informatics
University of Bergen, Norway

Mike Webster
Department of Computer Science
University of Wales Swansea

November 2, 1998

Abstract

It has long been acknowledged that scientific software is in need of better software engineering practices. Here we contrast the difference between conventional software development of CFD codes and an abstraction method based on tensor 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 segregates these concerns into fully independent software modules. Such considerations are important for the construction of numerical codes for practical problems.

Contents

*This investigation has been carried out with support from the European Union, ESPRIT-IV project 21871 SAGA (Scientific computing and algebraic abstractions).

†This research was mostly carried out during the author's sabbatical at University of Wales Swansea, throughout the academic year 1997/98.

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.

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 constructors used and having no type abstraction facilities. A change of language, e.g., functional or object-oriented does not necessarily involve a basic change in approach to programming numerical codes. However, there are documented 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 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 phases of which can be summarised as:

- model the physical problem;
- formulate an abstract solver algorithm;
- discretise in space/time, to transform the continuous problem to its discrete counterpart, e.g. differential/integral to algebraic, and expressed as arrays over reals;
- refine the solver, utilising properties of the problem (such as symmetries);
- convert/translate to program code.

In this paper we will contrast such a conventional development technique with an approach related more to pure mathematical concepts:

- model the physical problem;
- formulate an abstract solver algorithm;
- refine the solver utilising properties of the problem (such as symmetries) at the tensor level;
- 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 [1], and its software foundation was explored in [2].

1.1 Outline of Paper

This study presents a fully worked example to highlight the different approaches. The initial example is a coating problem for Newtonian flows and is outlined in section 2. We show how the conventional development approach leads to one type of code, whether it is expressed in a classical procedural language such as FORTRAN–77, the favoured language for scientific programming, or with the use of object–oriented languages or even functional languages.

Subsequently, we demonstrate, in more detail, how the tensor abstraction approach yields quite a different type of program. The latter programming style naturally requires the target language to have template classes, a feature lacking from even the more recent versions of FORTRAN, but present in C++, a language which is increasingly being used by practitioners in the field of high performance computing.

The scale of problem complexity is then advanced to address a non-Newtonian flow problem. The implications of shifting from Newtonian to non–Newtonian problem specifically is discussed in the light of the changes implied in the two classes of software. 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).

2 Outline of the Fluid Flow Problem

The computational fluid dynamics problem to be used for illustration is a variation of the wire coating problem ([3]). Two main tooling designs are commonly employed, described as *pressure* and *tube* tooling. 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, minimal residual stressing.

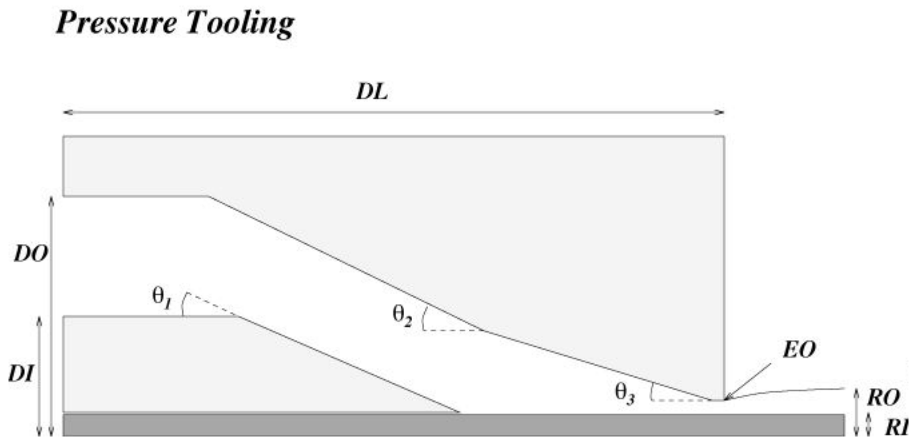


Figure 1: Pressure Tooling

Pressure tooling and Tube tooling are shown schematically in figures 1 and 2. In both cases the liquid polymer is dragged by the wire from a reservoir through the geometry as indicated. With pressure tooling the polymer is guided through a die directly down onto the wire. Commonly the polymer develops a so–called die swell free surface once it exits from the die. The tube tooling process allows the polymer to find a free surface between the one exit and the attachment point on the wire, involving a drawn–down section.

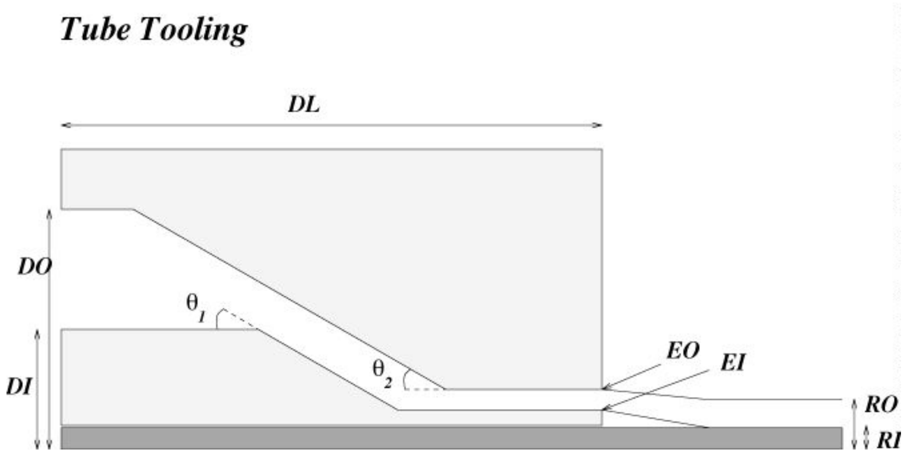


Figure 2: Tube Tooling

We begin with the basic partial differential equations (Navier-Stokes) defining the fluid flow and then outline how this is to be solved using the finite element method. This will involve Jacobi iteration and the Choleski decomposition algebraic solver for symmetric banded matrices.

2.1 The problem description

The specific type of wire coating problem of relevance, whether tube tooling or pressure tooling, is defined via input defining data sets. The governing equations for viscous incompressible non–isothermal flow may be described by the generalised Navier-Stokes equations

$$\rho\frac{\partial \mathbf{u}}{\partial t} + \rho(\mathbf{u}\cdot\nabla)\mathbf{u} - \nabla\cdot(\mu(\nabla\mathbf{u} + (\nabla\mathbf{u})^T)) + \nabla p = \mathbf{f} \tag{1}$$

with the associated incompressibility constraint

$$\nabla\cdot\mathbf{u} = 0. \tag{2}$$

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 a function of $\nabla\mathbf{u}$
- $p(\mathbf{x}, t)$ is the pressure, a scalar field,
- $\_^T$ is the matrix transpose operator,
- $\mathbf{f}$ is an external force acting on the fluid, a vector field, which for the example considered will be assumed to be zero.
- ∇ is the spatial derivative used in several forms:
  1. $(\mathbf{v}\cdot\nabla)\mathbf{u}$, the substantial 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. $\nabla\mathbf{u}$ yields a matrix tensor field when applied to a vector field $\mathbf{u}$,

3. $\nabla \cdot \tau$ yields a vector field when "dotted" with a matrix tensor field τ,
4. ∇p, the gradient, yields a vector field when applied to a scalar field p,
5. $\nabla \cdot \mathbf{u}$, the divergence, yields a scalar field when "dotted" with a vector field $\mathbf{u}$,

Note that scalar and vector fields are defined with a possibly different value at each point within the domain where the fluid is flowing. A real number, on the other hand, remains constant throughout the domain.

2.2 Non–dimensional form

Rather than adopting the above dimensional formulation (1) directly, the solver will be based on *normalised* equations, using a non–dimensional group number, the Reynold number $Re = \rho \frac{u_0 \ell_0}{\mu_0}$ for generality []. We take scaling factors:

- length scale ℓ_0, a real number,
- velocity scale u_0, a real number,
- viscosity scale μ_0, a real number,
- time scale t_0, a real number,

yielding the formulation, for $Re \geq 1$,

$$\frac{\partial \mathbf{u}'}{\partial t'} + (\mathbf{u}' \cdot \nabla')\mathbf{u}' - \frac{1}{Re}\nabla' \cdot (\mu'(\nabla' \mathbf{u}' + (\nabla' \mathbf{u}')^T)) + \nabla' p' = \mathbf{f}'$$
$$\nabla' \cdot \mathbf{u}' = 0$$

or, for $0 < Re \leq 1$,

$$Re\frac{\partial \mathbf{u}'}{\partial t'} + Re(\mathbf{u}' \cdot \nabla')\mathbf{u}' - \nabla' \cdot (\mu'(\nabla' \mathbf{u}' + (\nabla' \mathbf{u}')^T)) + \nabla' p' = \mathbf{f}' \tag{3}$$
$$\nabla' \cdot \mathbf{u}' = 0 \tag{4}$$

where

- $\mathbf{u}' = \mathbf{u}/u_0$ is the fluid's dimensionless velocity, a vector field,
- ∇' is the spatial derivative after normalisation (changed metric from the dimensioned 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,
- $\mathbf{f}' = \mathbf{f}/f_0$ the dimensionless external force, assumed to be zero, scaled by a force factor, and
- $t' = t/t_0$ the dimensionless time, scaled by a time factor.

In addition, the simplification for $Re \leq 1$ also depends on the following relationships between the parameters:

$$u_0 t_0 = l_0; \quad l_0 t_0 f_0 = 1; \quad p_0 t_0 = \mu_0$$

(or with variation for the case $Re \geq 1$ $l_0 t_0 f_0 = Re; p_0 t_0 = \mu_0 Re$).

The prime notation is discarded subsequently for clarity. In the coating problem, the Reynold number tends to be small, of typical value $Re \approx 10^{-4}$, hence form (3) and (4) is assumed.

For non–Newtonian flows, the viscosity itself depends typically on the velocity gradients so that the set of equations given here describes the motion of a *generalised* Newtonian fluid. These equations describe incompressible, isothermal fluid flow and are presented in a coordinate independent form.

For μ constant, the viscous term in a Cartesian co–ordinate system may be transformed to $\mu\nabla^2\mathbf{u}$ utilising the incompressibility constraint (4). Equation (3) can then be simplified to:

$$Re\frac{\partial \mathbf{u}}{\partial t} + Re\mathbf{u}.\nabla\mathbf{u} - \mu\nabla^2\mathbf{u} + \nabla p = \mathbf{f} \tag{5}$$

The particular test problem considered is annular in configuration and hence two dimensional, (see []). The weak formulation presented in section 2.6 is valid for two or three dimensions, and any geometry. Details for the annular co–ordinate configuration are provided in section 5.1

2.3 Time-discretisation and strong form of equations

We derive, the initial equations which lead eventually to the matrix equations (28) – (31) by performing a semi-discretisation on (3) and introducing solution vectors $\mathbf{u}^n$ and p^n at discrete time t_n where the time interval is Δt. We will use the notation $\mathsf{sym}(\tau) = \frac{1}{2}(\tau + \tau^T)$ for the symmetrisation of the matrix tensor field τ, in this description. In the rest of this section we ignore the force term $\mathbf{f}$.

We derive a set of difference equations which we can use 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 $\frac{\partial \mathbf{u}}{\partial t}$ by considering the half interval $(t_n, t_{n+1/2})$:

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

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

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

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

where $0 \le \theta \le 1$. Here we have taken a Crank Nicolson pressure gradient approximation and evaluated s at $\mathbf{u}^{n+1/2}$. However, (7) does not separate $\mathbf{u}^{n+1}$ and p^{n+1}. To this end introduce an intermediate variable $\mathbf{u}^*$, the solution of

$$\frac{Re}{\Delta t}(\mathbf{u}^* - \mathbf{u}^n) = s(\mathbf{u}^{n+1/2}) - \nabla p^n. \tag{8}$$

If $\mathbf{u}^*$ also satisfies:

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

then we see that $\mathbf{u}^{n+1}$ will satisfy (7) as required.

$\mathbf{u}^n$ must also satisfy the incompressibility constraint expressed by equation (4) at all times t_n. If we introduce the following equation:

$$\frac{Re}{\Delta t}\nabla.\mathbf{u}^* = \theta\nabla^2(p^{n+1} - p^n) \tag{10}$$

and apply the divergence operator $\nabla.$ to equation (9), we see that $\mathbf{u}^{n+1}$ will also satisfy (4). This Poisson equation for the pressure difference over a single time step completes the problem specification.

At this point, via discretisation with respect to time, we have derived the four vector equations (6), (8), (10) and (9). These will give rise to the matrix equations (28), (29), (30) and (31) when we discretise with respect to space. This we outline in section 3.

The equations are now re–arranged, yielding the following strong formulation. The algorithm consists of three steps, with the first splits into two substeps. The pressure p^{n+1} and velocity $\mathbf{u^{n+1}}$ at timestep $n+1$ are computed from the pressure p^n and velocity $\mathbf{u}^n$ at timestep n. The substeps 1a and 1b calculate the halfstep $\mathbf{u}^{n+1/2}$ and the auxilliary $\mathbf{u}^*$ approximations to the velocities.

- Step 1a

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

- Step 1b

$$\frac{Re}{\Delta t}(\mathbf{u}^* - \mathbf{u}^n) = \nabla \cdot (2\mu\, \mathsf{sym}(\nabla \mathbf{u}^{n+1/2})) - Re(\mathbf{u}^{n+1/2} \cdot \nabla)\mathbf{u}^{n+1/2} - \nabla p^n \tag{12}$$

- Step 2

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

- Step 3

$$\frac{Re}{\Delta t}(\mathbf{u}^{n+1} - \mathbf{u}^*) = -\theta \nabla(p^{n+1} - p^n) \tag{14}$$

These steps are iterated from an initial guess for the steady state solution. That is starting with some suitably chosen velocity $\mathbf{u}$ and pressure p. The time–stepping procedure is terminated when a steady state solution is achieved, determined say, on an L^2 norm basis for the velocity component

$$\begin{aligned}
\mathbf{d} &= \mathbf{u}^{n+1} - \mathbf{u}^n \\
Lu &= \int_\Omega \mathbf{d} \cdot \mathbf{d} \\
au &= \int_\Omega \mathbf{u}^{n+1} \cdot \mathbf{u}^{n+1} \\
tu &= \begin{cases} \frac{1}{\Delta t}\sqrt{\frac{Lu}{au}} & \text{if } \mathsf{max}(\mathbf{u}^{n+1} \cdot \mathbf{u}^{n+1}) > 1 \\ \frac{1}{\Delta t}\sqrt{Lu} & \text{otherwise} \end{cases}
\end{aligned}$$

and likewise for the pressure component

$$\begin{aligned}
d &= p^{n+1} - p^n \\
Lp &= \int_\Omega d \cdot d \\
ap &= \int_\Omega p^{n+1} \cdot p^{n+1} \\
tp &= \begin{cases} \frac{1}{\Delta t}\sqrt{\frac{Lp}{ap}} & \text{if } \mathsf{max}(p^{n+1} \cdot p^{n+1}) > 1 \\ \frac{1}{\Delta t}\sqrt{Lp} & \text{otherwise} \end{cases}
\end{aligned}$$

where Ω represents the domain of the problem. If both tu and tp are below a certain threshold tol, typically $tol = 10^{-7}$, then a steady state is accepted. With a discretised representation, we may approximate these integrals by summing the discrete values rather than computing the integrals.

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.

$$\begin{aligned}
\mathbf{u}(\mathbf{x}, 0) &= \mathbf{u}_0(\mathbf{x}) \\
p(\mathbf{x}, 0) &= p_0(\mathbf{x}),
\end{aligned} \tag{15}$$

where

$$\nabla \cdot \mathbf{u}_0 = 0, \tag{16}$$

and appropriate boundary conditions on Γ as

$$\begin{aligned} \mathbf{u} &= \mathbf{g}_1(\mathbf{x},t) \quad \text{on } \Gamma_1 \\ \Big(\mathbf{n}\cdot\sigma\Big) &= \mathbf{g}_2(\mathbf{x},t) \quad \text{on } \Gamma_2 \end{aligned} \tag{17}$$

Here $\Gamma\Big(\Gamma_1 \cup \Gamma_2\Big)$ 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 co–ordinate 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 the choice of $\mathbf{u}^*$'s boundary conditions is unrestricted we may take $\mathbf{u}^*$ to equal $\mathbf{u}^{n+1}$ 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 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 step 3. This argument maintains the overall order of the scheme. 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.

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 permits the use of a larger time increment [4, 5]. In step 1a, we replace the term $\mathbf{u}^n$ in the viscous term of the explicit scheme by the average at $t_{n+1/2}$ and t_n. This yields the semi–implicit equivalent form

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

Re–arranging, 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$'$

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

We treat step 1b in a similar manner, where the term $\mathbf{u}^{n+1/2}$ in the viscous term of the explicit scheme is replaced by the average of $\mathbf{u}^n$ and $\mathbf{u}^*$

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

Re–arranging, once again as above, yields the modified step 1b$'$

$$\begin{aligned} \frac{Re}{\Delta t}(\mathbf{u}^* - \mathbf{u}^n) - \nabla\cdot(\mu\,\mathsf{sym}(\nabla(\mathbf{u}^* - \mathbf{u}^n)) &= 2\nabla\cdot(\mu\,\mathsf{sym}(\nabla(\mathbf{u}^n)) \\ &\quad - Re(\mathbf{u}^{n+1/2}\cdot\nabla)\mathbf{u}^{n+1/2} - \nabla p^n \end{aligned} \tag{19}$$

The semi–implicit solver is then described by (18) and (19) with steps 2 and 3 as in section 2.3

2.6 Weak formulation

In this subsection, we present the weak variational formulation of the problem. 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 side of equations (18), (19) and (14) are dotted with the vector shape functions $\mathbf{v}$. These equations now present conservation of energy. The scalar test functions q are multiplied into the algorithmic step 2, equation (13).

The results are integrated over the whole domain Ω. The terms are integrated by parts in order to remove second order derivatives from the integrands and any pressure gradients in the defining velocity equations. Further simplification is achieved using the divergence theorem which gives rise to the integrals over the boundary Γ.

From (18), we have:

$$\begin{aligned}\frac{2Re}{\Delta t}\int_\Omega \Big(\mathbf{u}^{n+1/2} - \mathbf{u}^n\Big) - \nabla\cdot\big(\mu\,\mathsf{sym}(\nabla(\mathbf{u}^{n+1/2}-\mathbf{u}^n))\big) * \mathbf{v}d\Omega \\ = \int_\Omega (2\nabla\cdot(\mu\,\mathsf{sym}(\nabla(\mathbf{u}^n)) - Re(\mathbf{u}^n\cdot\nabla)\mathbf{u}^n - \nabla p^n) * \mathbf{v}d\Omega\end{aligned} \tag{20}$$

With the aid of integration by parts and the divergence theorem we establish the following

$$\int_\Omega \nabla p^n * \mathbf{v}d\Omega = \int_\Gamma p^n\mathbf{v}\cdot\mathbf{n}d\Gamma - \int_\Omega p^n(\nabla\cdot\mathbf{v})d\Omega$$

and

$$\int_\Omega (\nabla\cdot\mathbf{U}) * \mathbf{v}d\Omega = \int_\Gamma (\mathbf{U}\cdot\mathbf{v})\cdot\mathbf{n}d\Gamma - \int_\Omega \mathbf{U} * (\nabla\mathbf{v})d\Omega.$$

Applying these simplifications to (20) yields

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

Step 1a$''$ becomes:

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

From equation (19) we have

$$\begin{aligned}\int_\Omega \left(\frac{Re}{\Delta t}(\mathbf{u}^*-\mathbf{u}^n) - \nabla\cdot(\mu\,\mathsf{sym}(\nabla(\mathbf{u}^*-\mathbf{u}^n))\right) * \mathbf{v}d\Omega \\ = \int_\Omega\Big(2\nabla\cdot(\mu\,\mathsf{sym}(\nabla(\mathbf{u}^n)) - Re(\mathbf{u}^{n+1/2}\cdot\nabla)\mathbf{u}^{n+1/2} - \nabla p^n\Big) * \mathbf{v}d\Omega.\end{aligned} \tag{21}$$

Again, using integration by parts and the divergence theorem on equation (21), we obtain

$$\frac{Re}{\Delta t}\int_\Omega(\mathbf{u}^*-\mathbf{u}^n) * \mathbf{v}d\Omega + \int_\Omega \mu\,\mathsf{sym}(\nabla(\mathbf{u}^*-\mathbf{u}^n)) * \nabla\mathbf{v}d\Omega$$

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

Step 1b$''$ is then:

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

Step 2$''$ is obtained after integrating by parts which reduces second order to first order gradients on pressure:

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

Step 3$''$ results from integrating by parts to avoid pressure gradients:

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

Further simplification can be made by equating the surface integrals 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 equations (17) and for the Coating problem considered it is assumed there are no the traction boundary contributions.

In stage 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 owing to the choice of test functions. The weak formulation can thus be expressed through the following set of equations:

Step 1a:

$$
\begin{aligned}
& \frac{2Re}{\Delta t} \int_\Omega (\mathbf{u}^{n+1/2} - \mathbf{u}^n) * \mathbf{v} d\Omega + \int_\Omega \mu\,\mathsf{sym}(\nabla(\mathbf{u}^{n+1/2} - \mathbf{u}^n)) * \nabla \mathbf{v} d\Omega \\
& = -2 \int_\Omega \mu\,\mathsf{sym}(\nabla(\mathbf{u}^n)) * \nabla \mathbf{v} d\Omega - Re \int_\Omega ((\mathbf{u}^n \cdot \nabla)\mathbf{u}^n) * \mathbf{v} d\Omega + \int_\Omega p^n (\nabla \cdot \mathbf{v}) d\Omega
\end{aligned}
\tag{22}
$$

Step 1b:

$$
\begin{aligned}
& \frac{Re}{\Delta t} \int_\Omega (\mathbf{u}^* - \mathbf{u}^n) \cdot \mathbf{v} d\Omega + \int_\Omega \mu\,\mathsf{sym}(\nabla(\mathbf{u}^* - \mathbf{u}^n)) * \nabla \mathbf{v} d\Omega \\
= & \ -2 \int_\Omega \mu\mathsf{sym}(\nabla(\mathbf{u}^n)) * \nabla \mathbf{v} d\Omega - Re \int_\Omega (\mathbf{u}^{n+1/2} \cdot \nabla)\mathbf{u}^{n+1/2} \cdot \mathbf{v} d\Omega + \int_\Omega p^n \nabla \cdot \mathbf{v} d\Omega
\end{aligned}
\tag{23}
$$

Step 2:

$$
\theta \int_\Omega \nabla(p^{n+1} - p^n) \cdot \nabla q d\Omega = \frac{-Re}{\Delta t} \int_\Omega (\nabla \cdot \mathbf{u}^*) q d\Omega \tag{24}
$$

Step 3:

$$
\frac{Re}{\Delta t} \int_\Omega (\mathbf{u}^{n+1} - \mathbf{u}^*) \cdot \mathbf{v} d\Omega = \theta \int_\Omega (p^{n+1} - p^n) \nabla \cdot \mathbf{v} d\Omega \tag{25}
$$

3 Conventional software development and discretisation

A traditional exposition within the numerical community of such a complex problem as the coating problem would start with the presentation of the fundamentals behind the mathematical solution, passing over the details that are technically involved, but contribute little to the understanding of the problem. For simplicity, a Cartesian coordinate system would be used for this part. Then the exposition would turn to the discretisation technique, in this case the time–stepping scheme and FEM, and use a simplified explanation of the underlying mathematics of the approach. Continuing, the further exposition is now closely linked to its realisation using the FEM technique. Technical details are now supplemented at the discrete level, still working within the framework of the Cartesian coordinate system. As the exposition concludes, it is expanded with various material such as the use of axi–symmetric coordinate system, which has a marked influence on the way the operators are defined, and on the semi–implicit time stepping procedure. This information is often presented independently of each other, where crucial practical detail, such as how to perform the integration of the scalar fields, is often omitted. A student of the problem is intended to redo the development, tracing the additional features through the discretised presentation, finally realising a concrete level algorithm.

In order to numerically solve equations (3) and (4), the domain over which the solution is required is first triangulated into a finite-element mesh. The weak formulation, as presented in subsection (2.6), is used to derive a set of matrix equations relating the values of the velocities and pressure at the mesh nodes. The rest of this section is devoted to establishing these equations using a Cartesian co–ordinate system.

3.1 The Finite Element Method

We must now obtain the fully discretised equations and for this we use the finite element method. The mesh will consist of triangular elements and for each node j we have shape functions $\varphi_j(\mathbf{x})$ and $\psi_j(\mathbf{x})$ which are respectively piecewise quadratic and linear in $\mathbf{x}$. The important property of these functions is that they are 1 at the node j and zero at the other nodes of the mesh. For the quadratic case, the φ_i's are also zero at the mid point nodes of each edge.

We define the finite element interpolants for the velocity and pressure fields $\mathbf{U}(\mathbf{x},t)$ and $P(\mathbf{x},t)$ by:

$$\mathbf{U}(\mathbf{x},t) = \sum_{j=1}^{m_v} \varphi_j(\mathbf{x})\mathbf{U}_j(t); \quad P(\mathbf{x},t) = \sum_{k=1}^{m_p} \psi_k(\mathbf{x})P_k(t) \tag{26}$$

where m_v denotes the total number of nodes including the mid point nodes and m_p the number of nodes at the vertices of the triangles. The interpolants (26) are then substituted in equations (22 – 25) and appropriate test functions $\mathbf{v}$ and q chosen to produce a set of matrix equations.

Taking the first equation (22) we have:

$$\sum_{j=1}^{m_v}\left(\frac{2Re}{\Delta t}\int_\Omega \varphi_j(\mathbf{U}_j^{n+1/2} - \mathbf{U}_j^n)\cdot\mathbf{v}d\Omega + \int_\Omega \mu\,\mathsf{sym}(\nabla((\mathbf{U}_j^{n+1/2} - \mathbf{U}_j^n)\varphi_j)) * \nabla\mathbf{v}d\Omega\right) \tag{27}$$
$$= -\sum_{j=1}^{m_v}\left(2\int_\Omega \mu\,\mathsf{sym}(\nabla\varphi_j\mathbf{U}_j^n) * \nabla\mathbf{v}d\Omega + Re\int_\Omega((\mathbf{U}^n\cdot\nabla)\varphi_j\mathbf{U}_j^n)\cdot\mathbf{v}d\Omega\right) + \sum_{k=1}^{m_p}(\int_\Omega \psi_k(\nabla\cdot\mathbf{v})d\Omega P_k^n)$$

where the superscript n denotes evaluation at time step t_n. It is convenient to represent all the vectors $\mathbf{U}_j$, in $\mathbb{R}^3$, $j = 1\cdots m_v$, as one large 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 different types of integrals which appear equation (27). 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

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

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 can clearly be represented as the matrix–vector product $\mathbf{M}U$ where $\mathbf{M}$ has the block structure

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

where

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

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

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

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

To deal with the second term in (27), after expanding $\mathbf{U}_j$ in terms of the basis vectors, we consider

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

Taking the same set of test functions for $\mathbf{v}$, yields the $3m_v$ terms

$$\sum_{l=1}^{3}\sum_{j=1}^{m_v}\int_\Omega \mu \mathsf{sym}(\nabla(U_{lj}\boldsymbol{\varphi}_j^l)) * \nabla \boldsymbol{\varphi}_i^m d\Omega.$$

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

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

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

The next different type of expression in (27) to consider is

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

(where $\mathbf{U} = \sum_{k=1}^{m_v} \varphi_k \mathbf{U}_k$). With the test functions as before we obtain a $3m_v$ vector which can be written $\mathbf{N}(\mathbf{U})U$ where

$$\mathbf{N}(\mathbf{U})_{ij}^{lm} = \int_\Omega ((\sum_{k=1}^{m_v} \varphi_k \mathbf{U}_k \cdot \nabla)\boldsymbol{\varphi}_j^l) \cdot \boldsymbol{\varphi}_i^m d\Omega.$$

This matrix is called the *convection matrix.*

The final term in (27) has the form

$$\sum_{k=1}^{m_p}(\int_\Omega \psi_k(\nabla\cdot\mathbf{v})d\Omega P_k^n).$$

Substituting the same test functions $\boldsymbol{\varphi}^m$ for $\mathbf{v}$ yields the $3m_v$ terms

$$\sum_{k=1}^{m_p}(\int_\Omega \psi_k(\nabla\cdot\boldsymbol{\varphi}_i^m)d\Omega P_k^n).$$

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

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

and each block is

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

$\mathbf{L}$ is called the *incompressibility matrix.*

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

$$(\frac{2Re}{\Delta t}\mathbf{M}+\mathbf{S})(U^{n+1/2}-U^n)=\mathbf{L}P^n-(2\mathbf{S}+Re\mathbf{N}(U^n))U^n. \tag{28}$$

For step 1b, we see that equation (23) is of a very similar format to (22) and can be represented in a discretised form as:

$$(\frac{Re}{\Delta t}\mathbf{M}+\mathbf{S})(U^*-U^n)=\mathbf{L}P^n-2\mathbf{S}U^n-Re\mathbf{N}(U^{n+1/2})U^{n+1/2}. \tag{29}$$

For discretising step 2, we have, substituting in equation (24)

$$\theta\sum_{k=1}^{m_p}\int_\Omega\nabla\psi_k\cdot\nabla q d\Omega(P_k^{n+1}-P_k^n)=\frac{-Re}{\Delta t}\sum_{l=1}^{3}\sum_{j=1}^{m_v}\int_\Omega(\nabla\cdot\boldsymbol{\varphi}_j^l)qd\Omega U_{lj}^*.$$

Taking as test functions $q=\psi_i$ for $i=1,\cdots,m_p$, we obtain the following set of m_p equations:

$$\theta\sum_{k=1}^{m_p}\int_\Omega\nabla\psi_k\cdot\nabla\psi_i d\Omega(P_k^{n+1}-P_k^n)=\frac{-Re}{\Delta t}\sum_{l=1}^{3}\sum_{j=1}^{m_v}\int_\Omega\frac{\partial\varphi_j}{\partial x_l}\psi_i d\Omega U_{lj}^*.$$

To express the left hand side we introduce the *pressure stiffness matrix* $\mathbf{K}$

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

then step 2 becomes

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

Finally, the type of terms in (25) have already appeared in the other steps and we see that the discretisation of step 3 becomes:

$$\frac{Re}{\Delta t}\mathbf{M}(U^{n+1}-U^*)=\theta\mathbf{L}(P^{n+1}-P^n). \tag{31}$$

Equations (28) — (31) 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 calculated 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.

Equations (28), (29) and (31) are solved using Jacobi iteration which is described in the next section. The pressure equation, (30) is solved using a direct Choleski method, described in section (3.3). The mass matrix $\mathbf{M}$ is general very large. However, it is not necessary to store it explicitly and in practice only the *element* sub–block matrices $\mathbf{M}_e$ need to be constructed. This is also described in the next section.

3.2 Jacobi Iteration

Jacobi iteration is used for solving a matrix equation of the form:

$$\mathbf{MX} = \mathbf{B}$$

where $\mathbf{B}$ is a constant matrix comprising of an equal number of vectorial components to that of the velocity multiple vector.

A sequence of vectors $\mathbf{X}^{(r)}$ is generated by the scheme:

$$\mathbf{DX}^{(r+1)} = (\mathbf{D} - \omega\mathbf{M})\mathbf{X}^{(r)} + \omega\mathbf{B} \tag{32}$$

where $\mathbf{D}$ is a chosen diagonal form and ω is a positive relaxation factor.

If the sequence given by (32) converges to $\mathbf{X}$ then it is a solution to the matrix equation, since taking limits on both sides of (32) yields:

$$\mathbf{DX} = (\mathbf{D} - \omega\mathbf{M})\mathbf{X} + \omega\mathbf{B}$$

and so

$$\mathbf{MX} = \mathbf{B}$$

as required.

This iteration is governed by the iteration matrix $[\mathbf{I} - \omega\mathbf{D}^{-1}\mathbf{M}]$ and provides a method that is symmetrizable and hence ameniable to various iterative acceleration techniques [4, 6]. Such an iteration is convergent provided the spectral radius of the iteration matrix is less than unity. This condition can be satisfied if the matrix $\mathbf{D}$ is selected as the row sum (in modulus) of $\mathbf{M}$, i.e.

$$d_{ii} = \sum_{j=1}^{m_v} |m_{ij}|$$

[7].

For particular forms of $\mathbf{M}$ it is always possible to select ω so that optimal convergence performance can be attained for the equation systems per time-step. The iteration is therefore straightforward to implement and normally results in only a few iterative steps to provide satisfactory convergence.

The vectorial form $\mathbf{D}$, a diagonal matrix, must be stored and inverted and an initial iterative starting vector of $\mathbf{X}^0 = \mathbf{0} = \mathbf{C}^0$ is adopted. The iteration is implemented as:

$$\begin{aligned}
\mathbf{DX}^{(r)} &= \mathbf{C}^{(r)} \\
\mathbf{DX}^{(r+1)} &= \mathbf{C}^{(r)} - \omega\{\mathbf{MX}^{(r)} - \mathbf{B}\} \\
&= \mathbf{C}^{(r)} - \omega\{(\sum_{e=1}^{m_e} \mathbf{L}_e^T\mathbf{M}_e\mathbf{L}_e)\mathbf{X}^{(r)} - \mathbf{B}\} \\
&= \mathbf{C}^{(r)} - \omega\{(\sum_{e=1}^{m_e} \mathbf{L}_e^T\mathbf{M}_e\mathbf{X}_e^{(r)} - \mathbf{B}\}
\end{aligned} \tag{33}$$

which necessitates, per element, only element block matrix element-vector multiplication $\mathbf{M}_e\mathbf{X}_e^{(r)}$, followed by permutation transposition with $\mathbf{L}_e^T$ to derive appropriately ordered r.h.s. iteration vectors. Full system matrices may thus be avoided in this implementation requiring only vectors.

It is worth noting that the computational complexity of (33) is $O(k_1 m_e + k_2 m_v)$ where $k_1 \gg k_2$ are constants and as $m_e \leq k_3 m_v$ and $m_v \leq k_4 m_e$ for constants k_3 and k_4 the complexity is also $\mathrm{O}(m_e)$ or equivalently $\mathrm{O}(m_v)$ [3, 4, 8, 9, 10].

3.3 Choleski Decomposition

The direct method of solution employed for the pressure difference equation (30) is that of Choleski. With a positive definite banded symmetric matrix $\mathbf{A}$ there is a real non-singular matrix $\mathbf{L}$ such that:

$$\mathbf{L}\mathbf{L}^T = \mathbf{A} \text{ where } l_{ij} = 0 \qquad \text{if } (i - j > m_b)$$

and m_b is the band-width.

The elements of $\mathbf{L}$ may be determined in a reduction phase in approximately $\frac{1}{2} m_p (m_b + 1)(m_b + 2)$ multiplications and m_p square roots where m_p is the total number of pressure nodes in the mesh.

The elements of $\mathbf{L}$ are given by

$$\begin{aligned}
l_{pq} &= 0 && \text{if } q < 0 \text{ or } q > p \\
l_{ij} &= (a_{ij} - \sum_{k=i-m_b}^{i-1} l_{ik} l_{jk})/l_{jj} && (j = i - m_b, \ldots, i - 1) \\
l_{ii} &= (a_{ij} - \sum_{k=i-m_b}^{i-1} l_{ik}^2)^{1/2}
\end{aligned}$$

Reduction of this form must be performed *only once* at the outset of the time-stepping procedure. The solution of the set of equations $\mathbf{Ax} = \mathbf{b}$ can then be determined in the forward and backward substitution steps:

$$\mathbf{Ly} = \mathbf{b} \ , \ \mathbf{L}^T\mathbf{x} = \mathbf{y}$$

These two steps must be performed at every time step where a solution to equation (**??**) is required. The operation count of this phase is approximately $2m_p(m_b+1)$ multiplications and with $m_b \ll m_p$ is therefore linear in the length of the pressure difference vector m_p. The elements of $\mathbf{y}$ and $\mathbf{x}$ are given as follows:

$$\begin{aligned}
y_i &= (b_i - \sum_{k=i-m_b}^{i-1} l_{ik} y_k)/l_{ii} && (i = 1, \ldots, m_p) \\
x_i &= (y_i - \sum_{k=i+1}^{i+m_b} l_{ki} x_k)/l_{ii} && (i = m_p, \ldots, 1)
\end{aligned}$$

3.4 Termination for Steady State

The termination limit is based on the L^2 norms of the primitive variables, velocity and pressure. Let $ASOLN(I, J)$ be the value of the primitive variable at node I, for component J at time $n + 1$, and similarly $ASOL(I, J)$ at time n. [1] Then the L^2 norm is given by,

$$L^2_{norm} = \sum_{J=1}^{C} \sum_{I=1}^{N} \Big(ASOLN(I, J) - ASOL(I, J) \Big)^2, \tag{34}$$

where C is the number of components, and N the number of nodes of the primitive variable. Let

$$SOLLSQ = \sum_{J=1}^{C} \sum_{I=1}^{N} \Big(ASOLN(I, J) \Big)^2, \tag{35}$$

and $TOLSOL$ be a solution tolerance being less than a certain tolerance factor, TOL, before at convergence.[2] The maximum value of $TOLSOL$ is found of the separate primitive variables.

[1] J is 1 when the primitive variable is pressure; J=1,2 for the velocity in two-dimensional cases.

[2] Typically $TOL = 1.0 \times 10^{-8}$.

Also let $SOLMAX$ be the maximum absolute magnitude of the primitive variable. Then, for $SOLMAX \leq 1$

$$TOLSOL = \frac{\sqrt{L^2_{norm}}}{\Delta t}, \tag{36}$$

else,

$$TOLSOL = \frac{1}{\Delta t}\sqrt{\frac{L^2_{norm}}{SOLLSQ}}. \tag{37}$$

3.5 A discussion of relevant coding techniques

In the exposition above we have shifted the concepts towards to a very basic data level, where everything is now expressed using vectors and matrices. These may readily be implemented in any programming language using (multi–dimensional) arrays. The form above is imperative in style — *do this, then do that, update the variables and repeat until termination* — and is suitable for 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 [11] of submissions to Collected Algorithms from the ACM).

Several authors have followed this style of software development for Object-Oriented languages such as C++ and more recently Java. The result of this is typically classes representing vector and matrix operations with BLAS like routines. The general impressions is that this does not gain much over FORTRAN from the conceptual point of view, even though the increased emphasis on software engineering principles that these projects have often are beneficial for the quality of the code – especially when compared to *dusty deck* FORTRAN.

A variation of this is the Diffpack software package. 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 and allowing the numerical software developer to focus more clearly on the numerical aspects of task than that experienced by a FORTRAN programmer. Diffpack's approach to numerical software development is basically the conventional one outlined above, making choice of discretisation and coordinate systems explicit in the code.

More applicative implementations using functional languages have also been tried. Also in this case 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 larger readiness for automatic parallelisation (see the papers and approaches in [12]) than in the way the code is structured in the large. But as for object-orientation, people working in this area tend to put more emphasis on good software practices than the classically trained FORTRAN programmer does.

4 Tensor oriented methodology

In the tensor-oriented methodology we write the program code based directly on the concepts present in the abstract algorithm as shown in Section 2.6. For this we need an understanding of these abstract mathematical concepts, and how they can be supported by software. Then implementing the solver becomes straightforward.

In this section we briefly present the notions of tensors and coordinate freeness, scalar and tensor fields, the Sophus library, a finite element method implementation (FEM), and how the solver for the coating problem can be implemented using Sophus.

4.1 Coordinate free mathematics

Plain numerical types, such as the real numbers, are often abstracted as fields or rings. Rings are structures equipped with binary operations $+$ (addition), $-$ (subtraction) and $\cdot$ (multiplication), where addition is associative and commutative, subtraction is the inverse of addition, and multiplication is associative and distributes over addition in the familiar way. The neutral element with respect to addition and multiplication both exist, and are denoted 0 (zero) and 1 (one), respectively. A field is a ring equipped with / (division), the inverse of multiplication, multiplication is commutative, and such that 0 is distinct from 1.

Given a ring R, we may define, for any integer $n \geq 0$, 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). A vector space is a structure equipped with binary operations $+$ (addition) and $-$ (subtraction), where addition is associative and commutative and subtraction is the inverse of addition. Moreover, given a vector, we may scale it by a ring element using a ring-vector multiplication $*$. The ring-vector multiplication distributes over both ring and vector addition (like normal multiplication), and associates with ring multiplication[3]. Thus, given ring elements $a, b \in R$ and vectors $\mathbf{v}, \mathbf{v}' \in V$, we have that

$$\begin{aligned} (a+b) * \mathbf{v} &= (a * \mathbf{v}) + (b * \mathbf{v}), \\ a * (\mathbf{v} + \mathbf{v}') &= (a * \mathbf{v}) + (a * \mathbf{v}'), \\ a * (b * \mathbf{v}) &= (a \cdot b) * \mathbf{v}. \end{aligned}$$

Obviously, every vector space has a zero vector $\mathbf{0}$, and the 0-dimensional vector space has exactly one element, the zero vector. The ring itself satisfies the properties of being a 1-dimensional vector space if we use $\cdot$ as the ring-vector multiplication $*$.

A linear combination of vectors $\mathbf{v}_1, \ldots, \mathbf{v}_k$, for $k \geq 0$, is an expression of the form $a_1 * \mathbf{v}_1 + \ldots + a_k * \mathbf{v}_k$ where $a_1, \ldots, a_k \in R$. The vectors are linearly independent when the linear combination equals $\mathbf{0}$ if only if $a_1 = \ldots = a_k = 0$. Any vector $\mathbf{v}$ from an n-dimensional vector space V may be generated[4] from a collection $B \subseteq V$ of n linearly independent vectors. Such a collection of vectors is called a basis for V. Given a basis $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$, a vector $\mathbf{v} \in V$ can be identified by its coordinates $\bar{v} = (a_1, \ldots, a_n)$, i.e., ring elements $a_1, \ldots, a_n \in R$, such that $\mathbf{v} = a_1 * \mathbf{b}_1 + \ldots + a_n * \mathbf{b}_n$. 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.

Given any two vector spaces V and W over a ring R, then there exists linear mappings $\ell : V \to W$. The linear mappings preserve addition and multiplication, and thus subtraction and zero vectors, from the source vector space V to the target vector space W, i.e., $\ell(a * \mathbf{v}) = a * (\ell(\mathbf{v}))$ and $\ell(\mathbf{v} + \mathbf{v}') = \ell(\mathbf{v}) + \ell(\mathbf{v}')$ for $\mathbf{v}, \mathbf{v}' \in V$ and $a \in R$. But the collection M of linear mappings from V to W satisfies the vector space property, since we may define the ring-mapping multiplication $a * \ell$ by $(a * \ell)(\mathbf{v}) = a * (\ell(\mathbf{v}))$ and the mapping addition $\ell + \ell'$ by $(\ell + \ell')(\mathbf{v}) = \ell(\mathbf{v}) + \ell'(\mathbf{v})$ for linear mappings $\ell, \ell' \in M$ and all $a \in R$ and all $\mathbf{v} \in V$.

A linear map $e : V \to V$ is called an identity map if $e(\mathbf{v}) = \mathbf{v}$ for all $\mathbf{v} \in V$. A linear map $\ell : V \to W$ is an isomorphism iff there exists a linear map $\ell^{-1} : W \to V$ such that $\ell(\ell^{-1}(\mathbf{w})) = \mathbf{w}$ and $\ell^{-1}(\ell(\mathbf{v})) = \mathbf{v}$ for all $\mathbf{v} \in V$ and $\mathbf{w} \in W$. An isomorphism $\ell : V \to V$ will map a basis B for V into another basis B', and the inverse map ℓ^{-1} can be used to take a vector expressed as a linear combination $\bar{v}$ of the basis vectors B to its expression as a linear combination $\bar{v}'$ of B'. Basis changes allow stretching, twisting and interchanging of coordinates.

If $V = R^m$ and $W = R^n$, for $m, n \geq 0$, 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 basis B for an m-dimensional vector space V with the 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.

[3] This last rule is not mentioned in Serge Lang: Algebra

[4] This only holds in general if R is a field

This means that any linear mapping $\ell \in M$ is expressible using a linear combination of the basis vectors $b_i \otimes b'_j$, for $b_i \in B \subseteq V$ and $b'_j \in B' \subseteq W$ (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 $\mathbf{v} \in V^T$ 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 $\_^T : V \to V^T$, 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 $\mathbf{v} \cdot \mathbf{v}' = \mathbf{v}^T(\mathbf{v}')$.

Normally we have to do the 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 $\_^T$, 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 $b \geq 1$, 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 $k \geq 0$. 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 $(\tau_1 \cdot \tau_2) \in T^{(0)}_{R,B}$.

Normally we will distinguish between the tensor spaces formed from covectors and those formed from vectors. 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 $T^{(1)}_{R,B}$ (vectors and covectors) and 1-dimensional tensors $T^{(0)}_{R,B}$ (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 scalarfield. We may consider every ring element $a \in R$ as a scalarfield $a \in F_{X\to R}$ by $a(x) = a$ 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 $k \geq 0$. 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.

If the set X satisfies the properties of being a manifold, i.e., it has the notions of direction and distance, then we may define non-trivial derivation operations on the scalarfield. Typically X will be a b-dimensional vector space with basis vectors $B \in X$, 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 derivations 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 give rise to tensor field integration $\alpha = \int_\Omega \sigma dX$, yielding values in $\alpha \in T^{(k)}_{R,E}$, for any tensor $\sigma \in T^{(k)}_{F_{X\to R},E}$ and subdomain $\Omega \subseteq X$. The resulting tensor is relative to basis E, but the tensor is constant all points in the manifold X.

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

If X is 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 $T^{(1)}_{F_{X\to R},E}$.

We may now treat all symbols in the equations of Section 2.6 as tensors and coordinate free operations on tensors. Let us summarise the operations we need ,after 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 $a * \tau = a \otimes \tau$.
- $\sigma \cdot \tau$ 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 substantial 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 $\sigma \in T^{(k)}_{F_{X\to R},E}$ over a subdomain $\Omega \subseteq X$ yields a tensor in $T^{(k)}_{R,E}$.

There is a very 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 allows us to built advanced tensor operations, such as derivation operations, from scalar field operations, such as the partial derivatives, the latter requires us to implement the advanced tensor operations directly in terms of the (discretised) representation of X. As is evident, the former, which we have chosen, gives a clear separation between discretisation methods for the scalar field and the spatial derivation operations at the tensor level. The latter forces a tensor field implementation for each discretisation.

4.3 A Tensor Based Library for Solving PDEs

The Sophus library provides the abstract mathematical concepts from PDE theory as programming entities. It is based on the notions of manifold, scalar field and tensor field, while the implementations are based on the conventional numerical algorithms and discretisations. Sophus 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 equation solvers such as Choleski decomposition may easily be implemented for the meshes. Sparse meshes with operations that include Jacobi iteration may also be provided. Parallel and sequential implementations of mesh structures 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 distance 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 basic mesh structures for data.
- 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, axi–symmetric 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.

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 (FEM) 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 approximated by a sum $p \approx \sum_{g \in G} P_g \cdot g$ for scalar fields $g \in F_{X \to R}$, 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. Due to numerical properties, 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 $\cup_{e \in \mathcal{E}} e = X$. In a 2-dimensional case the domain may typically be split into triangles. A good choice of shape functions is to use functions which 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 in the domain X, and are non-0 only on those elements adjacent to that point. The shape function g is said to *reside* at the point for which it is 1. The collection of residing points are called the *nodal points* of the shape functions.

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. Since these operations are linear with respect to R, we have that

$$\begin{aligned} \frac{\partial p}{\partial \mathbf{x}} &\approx \sum_{g \in G} P_g \cdot \frac{\partial g}{\partial \mathbf{x}} \\ \int_\Omega p dX &\approx \sum_{e \in \mathcal{E}} \left(\sum_{g \in G} (P_g \cdot \int_e g dX) \right) \end{aligned}$$

Similar properties exist for the other scalar fields operations.

4.5 Developing a solution to 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, $E = \{\mathbf{e}_1, \ldots, \mathbf{e}_3\} \subseteq V_{X \to R}$ are 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 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.

In the development of the solver, the vector test functions $\mathbf{v}$ were dotted into the algorithmic steps 1 and 3. These equations now represent conservation of energy. The scalar test functions q were multiplied into the algorithmic step 2. The results were integrated over the whole subdomain Ω. Then the diffusion terms (step 1) were given a Crank-Nicolson treatment resulting in a semi-implicit time stepping scheme. This allowed a wider window of stability above an explicit implementation. The terms were further integrated by parts in order to remove second order integrals and any pressure gradients in the velocity defining equations.

We are working on a steady state problem with no traction boundary conditions, i.e., where all the boundary conditions have prescribed velocities. Also, the contribution from the boundary integrals is of the same order as the error introduced by the discretisation method/solver algorithm, so we will ignore these terms. It is however important to ensure that the velocities on the boundaries remain the same through all steps of the solver.

4.5.1 The Sophus algorithm

The coordinate free, variational form, of the solver equation steps (22) — (25) for the coating problem need some refinement to serve as an algorithm for a 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 are known, not any information about the direction (steps 1 and 3).
- The symmetrisations of $\nabla \mathbf{u}$ are "dotted" with $\nabla \mathbf{v}$ terms, thus intermixing information about all components of the vector $\mathbf{u}$, so that not even the magnitude of the unknown vectors are explicitly available (step 1).

The solution to these problems is to generate more equations at each step so that we find enough scalars to restore the scalar and vector fields we want to find. 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: scalar field integrated to 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 $(P^{n+1}_g - P^n_g) \in R$ 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

$$\begin{aligned}\forall q \in Q \quad . \quad & \theta \sum_{g\in G}(P^{n+1}_g - P^n_g)(\int_\Omega \nabla g \cdot \nabla q dX) \\ & = \frac{-1}{\Delta t}\int_\Omega (\nabla\cdot \mathbf{u}^*) * q dX.\end{aligned}$$

  This gives us a system of equations with $|Q|$ right hand sides[5] $B_q = \frac{-1}{\Delta t}\int_\Omega(\nabla\cdot\mathbf{u}^*) * q dX$ for $q \in Q$. It has $|G|$ unknowns $(P^{n+1}_g - P^n_g)$ for $g \in G$. The $|Q| \times |G|$ integrals on the left hand 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). 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^n_g is known it is easy to find P^{n+1}_g once the system is solved and the $(P^{n+1}_g - P^n_g)$, 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 loose any accuracy by the approximations of p^{n+1} in the system of equations.

- Step 3: 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 $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3 \in E \subseteq T^{(1)}_{F_{X\to R},E}$ and scalar fields $u_1, u_2, u_3 \in T^{(0)}_{F_{X\to R},E}$. Then the equation becomes

$$\begin{aligned}\forall \mathbf{v} \in V \quad . \quad & \frac{1}{\Delta t}\sum_{i=1}^{3}(\int_\Omega u_i * \mathbf{e}_i \cdot \mathbf{v} dX) \\ & = \theta \int_\Omega (p^{n+1} - p^n) * (\nabla\cdot\mathbf{v}) dX.\end{aligned}$$

  Now we can restore the scalar fields using the technique of shape functions $g \in G'$ as above, giving the following system of equations.

$$\begin{aligned}\forall \mathbf{v} \in V \quad . \quad & \frac{1}{\Delta t}\sum_{g\in G'}\left(\sum_{i=1}^{3}(U^{n+1}_{(g,i)} - U^*_{(g,i)})(\int_\Omega g * \mathbf{e}_i \cdot \mathbf{v} dX)\right) \\ & = \theta \int_\Omega (p^{n+1} - p^n) * (\nabla\cdot\mathbf{v}) dX.\end{aligned}$$

  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.

- Step 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

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

unknowns from the symmetrised term since symmetrisation is linear with respect to scalar multiplication. Then we may reformulate step 1a to

$$\begin{aligned}
&\forall \mathbf{v} \in V. \\
&\sum_{g \in G'} \left(\sum_{i=1}^{3} (U^{n+1/2}_{(g,i)} - U^{n}_{(g,i)}) \left(\frac{2Re}{\Delta t} \int_\Omega g * \mathbf{e}_i \cdot \mathbf{v} dX + \int_\Omega \mu\, \mathsf{sym}(\nabla(g * \mathbf{e}_i)) \cdot \nabla \mathbf{v} dX \right) \right) \\
&= -2 \int_\Omega \mu\, \mathsf{sym}(\nabla \mathbf{u}^n) \cdot (\nabla \mathbf{v}) dX \\
&- Re \int_\Omega ((\mathbf{u}^n \cdot \nabla) \mathbf{u}^n) \cdot \mathbf{v} dX \\
&+ \int_\Omega p^n * (\nabla \cdot \mathbf{v}) dX
\end{aligned}$$

with equation matrix $M' = \frac{2Re}{\Delta t} M + S$, and step 1b to

$$\begin{aligned}
&\forall \mathbf{v} \in V. \\
&\sum_{g \in G'} \left(\sum_{i=1}^{3} (U^{*}_{(g,i)} - U^{n}_{(g,i)}) \left(\frac{Re}{\Delta t} \int_\Omega g * \mathbf{e}_i \cdot \mathbf{v} dX + \int_\Omega \mu\, \mathsf{sym}(\nabla(g * \mathbf{e}_i)) \cdot \nabla \mathbf{v} dX \right) \right) \\
&= -2 \int_\Omega \mu\, \mathsf{sym}(\nabla \mathbf{u}^n) \cdot (\nabla \mathbf{v}) dX \\
&- Re \int_\Omega ((\mathbf{u}^{n+1/2} \cdot \nabla) \mathbf{u}^{n+1/2}) \cdot \mathbf{v} dX \\
&+ \int_\Omega p^n * (\nabla \cdot \mathbf{v}) dX
\end{aligned}$$

with equation matrix $M'' = \frac{Re}{\Delta t} M + S$, where $S_{\mathbf{v},(g,i)} = \int_\Omega \mu\, \mathsf{sym}(\nabla(g * \mathbf{e}_i)) \cdot \nabla \mathbf{v} dX$. This gives us 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.

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 all this information, we end up with 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 use the matrix solvers written for the mesh classes to find the unknowns. The algorithm repeats the following steps until convergence, 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

$$M' @ (U^{n+1/2} - U^n) \quad = \quad \left[\int_\Omega -2\mu\, \mathsf{sym}(\nabla \mathbf{u}^n) \cdot (\nabla \mathbf{v}) - Re((\mathbf{u}^n \cdot \nabla)\mathbf{u}^n) \cdot \mathbf{v} + p^n * (\nabla \cdot \mathbf{v}) dX \right]_{\mathbf{v} \in V}$$

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

$$M'' @ (U^* - U^n) \quad = \quad \left[\int_\Omega -2\mu\, \mathsf{sym}(\nabla \mathbf{u}^n) \cdot (\nabla \mathbf{v}) - Re((\mathbf{u}^{n+1/2} \cdot \nabla)\mathbf{u}^{n+1/2}) \cdot \mathbf{v} + p^n * (\nabla \cdot \mathbf{v}) dX \right]_{\mathbf{v} \in V}$$

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

$$\theta K @(P^{n+1} - P^n) \quad = \quad \left[\int_\Omega \frac{-Re}{\Delta t}(\nabla \cdot \mathbf{u}^*) * q)dX\right]_{q \in Q}$$

Calculate $p1$ from $P^{n+1} - P^n$, P^n and G;
Step 3: solve for $U^{n+1} - U^*$ in

$$\frac{Re}{\Delta t} M @(U^{n+1} - U^*) \quad = \quad \left[\int_\Omega \theta(p^{n+1} - p^n) * (\nabla \cdot \mathbf{v})dX\right]_{\mathbf{v} \in V}$$

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, so we must ensure that these values remain unchanged at every step. The matrix K will normally be a dense matrix 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 matrix. This also has a good effect on M' and M'', and Jacobi iteration will be a useful technique for solving the equations in steps 1 and 3.

For this problem good choices for test and shape functions are

- for $Q = G$: functions which reside at the corners of the elements, and are linear within each element, and
- for $V = G'$: functions which reside at the corners 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 scalarfield,
- in the implementation of the tensors,
- for storing the set of test functions, and
- for data matrices K (dense) and M, M' and M'' (sparse).

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 save storage requirements, improve run-time efficiency and provide parallel execution.

Testing for convergence is performed using the L^2 norm for velocities and pressure, and checking that the change over the last iteration is less than a threshold tol. Typically $tol = 10^{-8}$. The velocity norm is computed by

$$L_u = \begin{cases} \frac{1}{\Delta t}\sqrt{\frac{\int_\Omega (\mathbf{u}^{n+1} - \mathbf{u}^n)\cdot(\mathbf{u}^{n+1} - \mathbf{u}^n)dX}{\int_\Omega \mathbf{u}^{n+1}\cdot \mathbf{u}^{n+1} dX}} & \text{if } \max(\mathbf{u}^{n+1} \cdot \mathbf{u}^{n+1}) > 1 \\ \frac{1}{\Delta t}\sqrt{\int_\Omega (\mathbf{u}^{n+1} - \mathbf{u}^n) \cdot (\mathbf{u}^{n+1} - \mathbf{u}^n)dX} & \text{otherwise} \end{cases}$$

and for pressure

$$L_p = \begin{cases} \frac{1}{\Delta t}\sqrt{\frac{\int_\Omega (p^{n+1} - p^n)\cdot(p^{n+1} - p^n)dX}{\int_\Omega p^{n+1}\cdot p^{n+1} dX}} & \text{if } \max(p^{n+1} \cdot p^{n+1}) > 1 \\ \frac{1}{\Delta t}\sqrt{\int_\Omega (p^{n+1} - p^n) \cdot (p^{n+1} - p^n)dX} & \text{otherwise} \end{cases}$$

Thus the iteration terminates with a result when both $L_u < tol$ and $L_p < tol$.

Coding this algorithm as a computer program is now straight forward if a library like Sophus is available. The PDE solver would be written as a procedure, and based on the detailed exposition above this should 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 bring the solver module code size up to about 1000 lines of code. The code needed for a simple user interface, I/O file handling and configuration of the PDE solver typically lies around a 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++ [13], Eiffel [14], GJ [15] and Java [16], type abstraction and functors in applicative functional languages like standard ML and Haskel, 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, Haskel 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 alegbraic 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 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 C++. 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 upcoming Fortran-2000 supports templates it would seem like a good language for this style of programming.

The tensor oriented package RHALE++ from Sandia National Laboratories [17, 18] 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 quality of a programming style can best be evaluated by checking 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 axi–symmetry 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 rotation angle of an rz-plane. The data fields will vary only along the r- and z-axis, while they will be constant along the θ-axis. Thus all partial derivatives along θ-axis are 0, and there is no need to store information for this axis. A general problem with cylindrical coordinates is singularities for $r = 0$. For the coating problem this may be ignored, as the polymer flow never will reach the z-axis, and, hence, the singular case is outside of our domain Ω.

5 Modifying the problem

The quality of a programming style can best be evaluated by checking 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 axi–symmetry 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 rotation angle of an rz-plane. The data fields will vary only along the r- and z-axis, while they will be constant along the θ-axis. Thus all partial derivatives along θ-axis are 0, and there is no need to store information for this axis. A general problem with cylindrical coordinates is singularities for $r = 0$. For the coating problem this may be ignored, as the polymer flow never will reach the z-axis, and, hence, the singular case is outside of our domain Ω.

5.1 Conventional case

We now describe the formulation using cylindrical co–ordinates, 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 $\mathbf{u} = (u_r, 0, u_z)$. The integration over the domain becomes specific for any particular geometry, so

$$\int_\Omega f(x)g(x)d\Omega \equiv 2\pi \int_r \int_z f(r,z)g(r,z)rdrdz,$$

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,

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

$$\nabla s = \delta_{\mathbf{r}}\frac{\partial s}{\partial r} + \delta_{\mathbf{z}}\frac{\partial s}{\partial z},$$

where $\delta_{\mathbf{r}}$ and $\delta_{\mathbf{z}}$ are unit vectors, and

$$\nabla = \delta_{\mathbf{r}}\frac{\partial}{\partial r} + \delta_\theta \frac{1}{r}\frac{\partial}{\partial \theta} + \delta_{\mathbf{z}}\frac{\partial}{\partial z}. \tag{38}$$

With the above definitions, the operators in the variational formulation of the problem (22) — (25) can be replaced to yield a formulation in cylindrical coordinates. For example the dyad $\nabla\mathbf{u}$ can be evaluated, using equation (38) and properties of unit vectors $\boldsymbol{\delta}_r$, $\boldsymbol{\delta}_\theta$ and $\boldsymbol{\delta}_z$, to be;

$$\nabla\mathbf{u} = \begin{bmatrix} \frac{\partial u_r}{\partial r} & 0 & \frac{\partial u_z}{\partial r} \\ 0 & \frac{u_r}{r} & 0 \\ \frac{\partial u_r}{\partial z} & 0 & \frac{\partial u_z}{\partial z} \end{bmatrix}. \tag{39}$$

By expanding the $\mathbf{U}_j$ in terms of the base vectors $\boldsymbol{\delta}_r$, $\boldsymbol{\delta}_\theta$ and $\boldsymbol{\delta}_z$ and using as test functions $\boldsymbol{\varphi}_i^r = \varphi_i\boldsymbol{\delta}_r$ and $\boldsymbol{\varphi}_i^z = \varphi_i\boldsymbol{\delta}_z$ 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

$$\mathbf{S}_{ij}^{lm} = \int_\Omega \mu \mathsf{sym}(\nabla\boldsymbol{\varphi}_j^l) * \nabla\boldsymbol{\varphi}_j^m d\Omega \tag{40}$$

and $l, m \in \{r, z\}$. Using (39) we have

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

and

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

Substituting in (40) we can express the components of $\mathbf{S}$ as follows:

$$\begin{aligned}
\mathbf{S}_{ij}^{rr} &= 2\pi \int_r \int_z \Big[2\frac{\partial \varphi_i}{\partial r}\frac{\partial \varphi_j}{\partial r} + 2\frac{\varphi_i \varphi_j}{r^2} + \frac{\partial \varphi_i}{\partial z}\frac{\partial \varphi_j}{\partial z} \Big] r dr dz, \\
\mathbf{S}_{ij}^{rz} &= 2\pi \int_r \int_z \Big[\frac{\partial \varphi_i}{\partial r}\frac{\partial \varphi_j}{\partial z} \Big] r dr dz, \\
\mathbf{S}_{ij}^{zz} &= 2\pi \int_r \int_z \Big[2\frac{\partial \varphi_i}{\partial z}\frac{\partial \varphi_j}{\partial z} + \frac{\partial \varphi_i}{\partial r}\frac{\partial \varphi_j}{\partial r} \Big] r dr dz.
\end{aligned}$$

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

$$\mathbf{M}_{ij} = 2\pi \int_r \int_z \Big[\varphi_i \varphi_j \Big] r dr dz, \tag{41}$$

$$\mathbf{N}(V)_{ij} = 2\pi \int_r \int_z \varphi_i \Big[\varphi_l V_r^\ell \frac{\partial \varphi_j}{\partial r} + \varphi_l V_z^\ell \frac{\partial \varphi_j}{\partial z} \Big] r dr dz, \tag{42}$$

$$\mathbf{K}_{kl} = 2\pi \int_r \int_z \nabla \psi_k \cdot \nabla \psi_l r dr dz, \tag{43}$$

$$\mathbf{L}_{li}^1 = 2\pi \int_r \int_z \psi_l \frac{\partial \phi_i}{\partial r} r dr dz, \quad \mathbf{L}_{li}^2 = 2\pi \int_r \int_z \psi_l \frac{\partial \varphi_i}{\partial z} r dr dz. \tag{44}$$

Here V_r^ℓ and V_z^ℓ are the nodal velocity components in radial (r) and axial (z) directions respectively. Repeated indices imply summation taken over i,j and ℓ for all velocity node points, and k,l for all vertex pressure nodal points on triangular meshes.

5.2 Coordinate free case

Here we are taking a closer look at the solver algorithm presented in Section 4.5. There the the solver for the coating problem was presented in a coordinate free notation. This means that the solver is independent of coordinate system, and need not be changed when moving to cylindrical coordinates. However the configuration has to be changed to provide the tensor class with the definition of the appropriate coordinate system. Also, the scalar field implementation should be changed to two-dimensional form and the θ direction should be handled by a constant scalar field. 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 was implemented. Such an implementation may take 1000–4000 lines of code, but not all of this needs to be written from scratch.

6 Discussion

These two different development strategies give vastly different software characteristics. The reference code prepared for SAGA (deliverable D1-III (Coating-F90)) totals about 18 000-19 000 lines of code.

The exposition of the coating program and its detailed pseudocode is well below 1 000 lines of code. An estimate of the size of the coded application program yields approximately 2 000 lines of code. Using lines of code as a measurement of the development costs, which is the basis for cost estimating models like COCOMO, we find an overwhelming reduction in code development costs for the SAGA methodology compared to the traditional methodology.

The cost of implementing a discretisation from scratch, i.e., the manifold with associated point set and scalar field seems to lie around 4 000 lines of code. Adding a new discretisation technique for an existing manifold corresponds to developing 1 000 lines of code. Recall that all scalar fields have the same interface, so that given two different scalar field discretisations, we may interchange them within the same application program without much adaptation taking place.

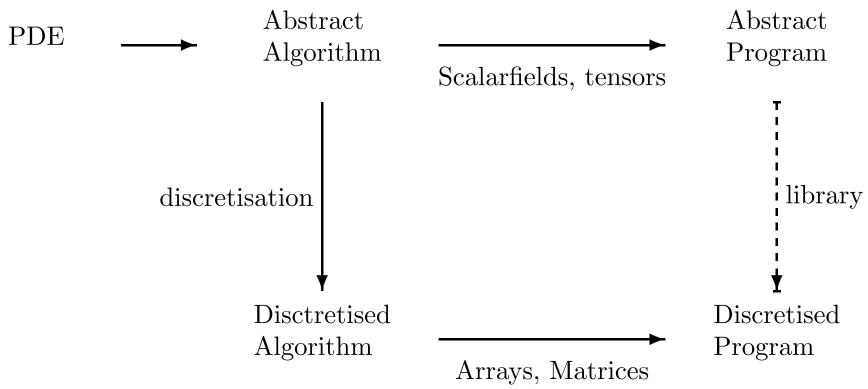
These figures show that the methodologies promoted in SAGA may improve computational modelling productivity dramatically. Also recall, that this includes both parallel and sequential versions of the code.

Does this abstract programming then have a detrimental impact on execution speed of the final code. Since we have not been able to finish the coding of the coating problem in Sophus, a direct comparison with the reference code is not possible. However we have some indications from deliverable D1-VII, which compares C++ with FORTRAN-90. In some tests C++ obtained less than a quarter of the speed of F90, while in other tests F90 was half the speed of sequential C++. The results seem highly dependent on memory layout and other factors which are difficult to control. A tool like CodeBoost, deliverable D1-VI, would be valuable in systemising experiments with various data layouts and other transformations of the code.

Another open question is the effect this way of writing programs has on 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 a “wrong” discretisation technique is being used. Perhaps it will be possible to provide the scalar fields with some “certificate” of their numerical properties at the abstract level.

From a programming theory viewpoint there is a definite need to present the coating problem on the abstract level as far as possible. Only after all the technical details have been exposed at that level, comes the time to introduce the discretisation technique 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.

These two methodological approaches to the presentation of the problem also show up in the way software development is handled.



Software development in the field of computational modelling typically starts with some partial differential equation (PDE). This is then refined into an abstract algorithm, and a mental leap takes place, bringing us down to a discretised version of the algorithm. Further refinement takes place at the discrete level, and the language being used are the elementary data types of arrays and matrices. We then end up with a sequential program, that may be further refined into a parallel program.

The software development methodology we propose for the SAGA project would also start with going from a PDE to an abstract algorithm. But then further refinement would stay on 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 development work.

References

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

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

[3] 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*, 10:327–351, 1990.

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

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

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

[7] D. Ding, P. Townsend, and M.F. Webster. Computer modelling of transient thermal flows of non-Newtonian fluids. *J. Non-Newtonian Fluid Mechanics*, 47:239–265, 1993.

[8] D.M. Hawken, P. Townsend, and M.F. Webster. Use of dynamic data structures in finite element applications. *Int. J. Num. Meth. Eng.*, 33:1795–811, 1992.

[9] A. Baloch, P. Townsend, and M.F. Webster. On two and three dimensional expansion flows. *J. Comput. Fluids*, 24(8):863–82, 1995.

[10] A. Baloch, P. Townsend, and M.F. Webster. On vortex development in viscoelastic expansion and contraction flows. *J. Non–Newt. Fluid Mech.*, 65:133–49, 1996.

[11] T.R. Hopkins. Is the quality of numerical subroutine code improving. In Arge, Bruase, and Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 311–324. Birkhäuser, Boston, 1997.

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

[13] Bjarne Stroustrup. *The* C++ *Programming Language.* Addison-Wesley, 3d edition, 1997.

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

[15] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip 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/.

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

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

[18] 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):655–663, 1993.