An implicit finite volume method for arbitrary transport equations

D. J. E. Harvie[1]

(Received 30 January 2011; revised 15 February 2012)

Abstract

A finite volume framework is described for solving multiphysics transport problems. The method operates in a unique way: the transport equations and associated boundary conditions are input by the user using pseudo-mathematical expressions. A Perl program parses these equations and, via the computer algebra system Maxima, 'metaprograms' a Fortran code that solves the problem on an unstructured mesh using the Newton–Raphson method. The strength of the technique is that a fully implicit numerical formulation is generated and modified easily, for an arbitrary set of equations. The implemented algorithm ('arb') is available for download and licensed under the GNU General Public License.

http://journal.austms.org.au/ojs/index.php/ANZIAMJ/article/view/3949 gives this article, © Austral. Mathematical Soc. 2012. Published March 27, 2012. ISSN 1446-8735. (Print two pages per sheet of paper.) Copies of this article must not be made otherwise available on the internet; instead link directly to this URL for this article.

Contents

1 Introduction **C1127**

2 Finite volume discretisation **C1130**

3 Pseudo-mathematical expression language **C1134**

4 Newton–Raphson solution procedure **C1136**

5 Example nonlinear diffusion equation results **C1140**

6 Conclusion **C1143**

1 Introduction

Transport equations are partial differential equations (PDEs) that describe the movement of physical quantities such as mass, momentum and thermal energy. In the engineering and science fields, the transport equations that govern a process often have a 'multiphysics' nature, meaning that:

- Several quantities are interdependent, obeying coupled transport equations that must be solved concurrently (for example, exothermic reactions occurring in a flowing gas containing a number of chemical species);
- The transport equations contain varying and/or nonlinear coefficients, often in the diffusion or source terms (for example, non-Newtonian fluid flow or systems that involve temperature dependent material properties); or
- The simulation domain is split between a number of coupled regions, each of which may have different dimensions or different interdependent quantities to be found.

Due to these complexities, and also because most practically relevant geometries tend to be complex, numerical methods are usually required to solve multiphysics transport problems.

The three most popular numerical techniques used to solve transport equations are the Finite Difference Method (FDM), Finite Element Method (FEM) and Finite Volume Method (FVM). Under the FDM, each derivative within each PDE is replaced by its differenced equivalent. Numerically stored values approximate the real solution at particular points. While the FDM is simple to implement on structured meshes, the procedure is less straightforward on unstructured meshes, and unstructured meshes most easily represent typical practical (that is, complex) geometries.

Both finite element and finite volume methods are readily applied to unstructured meshes. Within the FEM unknown variables are approximated in each element by basis functions that have unknown coefficients. The coefficients are chosen so that within each element, an integral error between the approximating function and the PDE solution is minimised. The FVM is specific to the solution of transport equations. Under the FVM each transport equation is integrated over the volume of each cell (the FVM equivalent of the FEM's element) and, using the Gauss divergence theorem, is converted into a relationship between the fluxes occurring over the faces of the cell and integral quantities associated with the cell. Numerically stored values approximate solution variables averaged over their corresponding mesh element (either a face or a cell).

FEMs and FVMs have different strengths. The principal strength of the FEM is that it can be highly accurate for smoothly varying functions, and as a result it has been extensively employed in areas such as structural analysis where physical quantities such as stresses and (small) deformations vary continuously. The principal strength of the FVM is that the total amount of each transported quantity within the domain is conserved (for example, mass, momentum, thermal energy). This strength has meant that the FVM has (arguably) become the most popular method for solving complex transport problems (for

example, multiphase, varying physical properties, non-Newtonian) as it can be usefully applied when the physical system involves shocks and/or large property variations.

While the use of the FVM in the present multiphysics context would appear to be attractive, a complication is that unlike the FEM, each equation discretised by the FVM depends on variables that are not local to the corresponding mesh cell. An implication of this is that FVM discretisation methods are difficult to apply generally (unlike FEMs), instead tending to be specific to the particular equation set being solved (for example, the SIMPLE type methods for the solution of the Navier–Stokes equations). This lack of flexibility has prevented the FVM from being utilised as a true multiphysics simulation 'workbench', with most commercial and open source multiphysics software currently available being FEM based (for example, COMSOL [6] and Elmer [4]).

This article resolves this inflexibility by presenting a mathematical and computational framework that automates the generation of finite volume algorithms for solving multiphysics transport equations. The method is based on a metaprogramming algorithm that takes pseudo-mathematical expressions entered by the user and discretises these to produce compilable Fortran code. We illustrate the method via a three dimensional, steady state, nonlinear diffusion equation solved on an unstructured mesh.

Section 2 discretises the illustrative diffusion problem using the finite volume method, producing a set of nonlinear equations that can be solved computationally. This process serves to detail the relationship between the analytical PDE and discretised FVM system, and introduces the different variables (type and centring) employed in the final code. Section 3 overviews the pseudo-mathematical finite volume language, demonstrating in particular how the generic diffusion PDE is represented using this syntax. Section 4 shows how the resulting set of nonlinear equations is solved implicitly via a Newton–Raphson method, and outlines what constraints this solution method places on the original system of equations. Finally, Section 5 presents results for the demonstration diffusion problem.

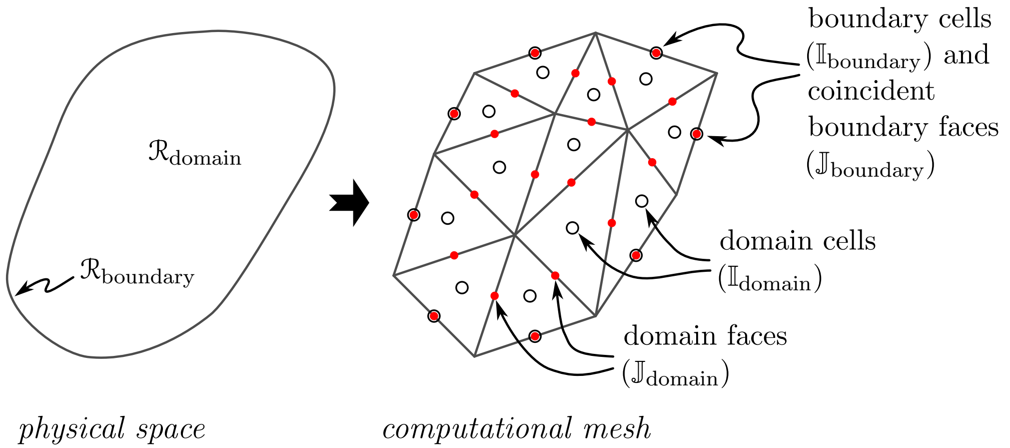

Figure 1: A two dimensional schematic showing the relationship between the physical space and computational mesh.

2 Finite volume discretisation

The example problem used to illustrate the method consists of a PDE for the unknown variable $\phi(\mathbf{x})$,

$$\boldsymbol{\nabla} \cdot \Gamma(\phi)\boldsymbol{\nabla}\phi + \Lambda(\phi) = 0\,, \tag{1}$$

applied over the domain region $\mathcal{R}_{\text{domain}}$. Dirichlet boundary conditions,

$$\phi = \phi_{\text{boundary}}(\mathbf{x}), \tag{2}$$

are applied along the domain's boundary, $\mathcal{R}_{\text{boundary}}$. Although this problem is chosen purely to demonstrate various aspects of the computational framework, it could represent a heat conduction problem (say), with ϕ representing temperature, $\Gamma(\phi)$ and $\Lambda(\phi)$ representing a temperature dependent thermal diffusivity and heat generation rate, respectively, and fixed temperatures applied around the surface of the object.

The first step is to discretise the solution region. As shown in Figure 1 an unstructured mesh is used that is composed of cells and faces. A face separates two cells and has a dimension that is one less than the dimension of $\mathcal{R}_{\text{domain}}$. The indices of the cells and faces are represented by the sets $\mathbb{I}$ and $\mathbb{J}$, respectively. The set of all cells is composed of domain cells ($\mathbb{I}_{\text{domain}}$) which have the same dimension as the region $\mathcal{R}_{\text{domain}}$, and boundary cells ($\mathbb{I}_{\text{boundary}}$), which have the same dimension as $\mathcal{R}_{\text{boundary}}$. The set of all faces is similarly composed of faces that separate only domain cells ($\mathbb{J}_{\text{domain}}$) and faces that are on the region boundary ($\mathbb{J}_{\text{boundary}}$). Note that each boundary cell has the same geometry and is coincident with a boundary face. The boundary cells are defined to aid in equation discretisation.

FVM concepts are now applied to discretise each equation. For the PDE, Equation (1) is integrated over each domain cell yielding

$$\mathring{E}_i^{(1)} = \frac{1}{|V_i|}\int_{V_i} \boldsymbol{\nabla}\cdot\Gamma(\phi)\boldsymbol{\nabla}\phi\,dV + \frac{1}{|V_i|}\int_{V_i}\Lambda(\phi)\,dV\,, \quad i\in\mathbb{I}_{\text{domain}}\,. \tag{3}$$

Application of the Gauss divergence theorem then gives, for $i\in\mathbb{I}_{\text{domain}}$,

$$\mathring{E}_i^{(1)} = \frac{1}{V_i}\sum_{j\in\mathbb{J}_{\text{cellfaces},i}} \frac{\mathbf{N}_{i,j}\cdot\boldsymbol{n}_j}{|S_j|}\int_{S_j}\Gamma(\phi)\,dS\int_{S_j}\boldsymbol{n}_j\cdot\boldsymbol{\nabla}\phi\,dS + \frac{1}{|V_i|}\int_{V_i}\Lambda(\phi)\,dV\,, \tag{4}$$

where the conventional FVM distributive approximation

$$\int_{S_j}\Gamma(\phi)\mathbf{N}_{i,j}\cdot\boldsymbol{\nabla}\phi\,dS \approx \frac{1}{|S_j|}\int_{S_j}\Gamma(\phi)\,dS\int_{S_j}\mathbf{N}_{i,j}\cdot\boldsymbol{\nabla}\phi\,dS \tag{5}$$

is used [3]. In these equations $\mathbb{J}_{\text{cellfaces},i}$ is the set of all faces that surround cell i, $|V_i|$ is the volume of cell V_i, $|S_j|$ is the area of face S_j, $\mathbf{N}_{i,j}$ is a unit normal vector at face j that is directed outward from cell i, and $\boldsymbol{n}_j$ is a unit normal that defines a unique orientation for face j.

The variable $\mathring{E}_i^{(1)}$, defined by Equation (3), is a 'code' variable as it is employed in the final algorithm. Each code variable has both a type and a centring.

This code variable is an *equation* variable, meaning that when the discretised system is solved, its value will be zero. The code variable is also *cell* centred, meaning that it has a value for each cell within a certain set of cells (in this case, those within $\mathbb{I}_{\text{domain}}$). Regarding notation: a cell centred code variable has an open circle above it, while a face centred code variable has a disc above it.

Returning to the discretisation we recognise that, under the FVM, calculated variables represent values averaged over their corresponding mesh element (either a face or a cell). Hence, a cell centred *unknown* code variable is used to numerically represent ϕ,

$$\mathring{U}_i^{(1)} = \frac{1}{|V_i|} \int_{V_i} \phi \, dV \,, \quad i \in \mathbb{I} \,. \tag{6}$$

Similarly, Λ is represented by the cell centred *derived* code variable

$$\mathring{D}_i^{(1)} = \Lambda(\mathring{U}_i^{(1)}) \approx \frac{1}{|V_i|} \int_{V_i} \Lambda(\phi) \, dV \,, \quad i \in \mathbb{I}_{\text{domain}} \,, \tag{7}$$

where $\Lambda(\phi)$ is a user supplied function.

Equations (6) and (7) introduce two new code variable types: *unknown* variables are those that we are solving for. The system is solved when the current *unknown* variable values result in all *equation* variables equalling zero. *Derived* code variables are introduced for convenience. These variables are functions of *unknown* variables and/or other *derived* variables and are entered as expressions by the user. Each *equation* and *derived* variable is ultimately only a function of the *unknown* variables, and hence will have a unique value for a given set of *unknown* values.

The two surface integrals in Equation (4) represent face centred quantities, and hence are represented via face centred code variables. The discretised Γ is represented by the face centred *derived* variable,

$$\dot{D}_j^{(2)} = \Gamma(\dot{L}_j^{(1)}) \approx \frac{1}{|S_j|} \int_{S_j} \Gamma(\phi) \, dS \,, \quad j \in \mathbb{J} \,, \tag{8}$$

where $\Gamma(\phi)$ is another user supplied function, and $\dot{L}_j^{(1)}$ is a *local* variable representing the face averaged value for ϕ, defined as

$$\dot{L}_j^{(1)} = \sum_{i \in \mathbb{I}_{\text{facecells},j}} \mathring{k}_{j,i}^{(0)} \mathring{U}_i^{(1)} \approx \frac{1}{|S_j|} \int_{S_j} \phi \, dS \,, \quad j \in \mathbb{J} \,. \tag{9}$$

The averaging kernel coefficients $\mathring{k}_{j,i}^{(0)}$ appearing in Equation (9) are calculated at the start of a simulation using the Moving Least Squares (MLS) method [1], with $\mathbb{I}_{\text{facecells},j}$ being the set of all cells local to the face j.

Local variables, such as $\dot{L}_j^{(1)}$ defined in Equation (9), are similar to *derived* variables in that they can be functions of *unknown*, *derived* and other *local* variables. The differences are that they

1. are generated by the language parsing algorithm rather than being entered directly by the user,
2. may contain at most one sum of other code variables, and
3. are evaluated individually only when they are needed, rather than being evaluated as a set and stored in memory.

The second surface integral in (4) represents the gradient of ϕ in a direction normal to face j. This term is represented by a second face centred *local* variable

$$\dot{L}_j^{(2)} = \sum_{i \in \mathbb{I}_{\text{facecells},j}} \mathring{k}_{j,i}^{(1)} \mathring{U}_i^{(1)} \approx \frac{1}{|S_j|} \int_{S_j} \mathbf{n}_j \cdot \boldsymbol{\nabla} \phi \, dS \,, \quad j \in \mathbb{J} \,, \tag{10}$$

where the first derivative kernel coefficients $\mathring{k}_{j,i}^{(1)}$ are again pre-calculated using the MLS method. Finally, by defining a cell centred third *local* variable as

$$\mathring{L}_i^{(3)} = \sum_{j \in \mathbb{J}_{\text{cellfaces},i}} \dot{d}_{i,j} \dot{D}_j^{(2)} \dot{D}_j^{(3)} , \quad i \in \mathbb{I}_{\text{domain}} \,, \tag{11}$$

where

$$\dot{d}_{i,j} = \frac{(\mathbf{N}_{i,j} \cdot \mathbf{n}_j)|S_j|}{|V_i|} \,, \tag{12}$$

Equations (6–11) can be used to rewrite the original PDE (4) as

$$\mathring{E}_i^{(1)} = \mathring{L}_i^{(3)} + \mathring{D}_i^{(1)}, \quad i \in \mathbb{I}_{\text{domain}}. \tag{13}$$

Equation (13) is the equation that is used in the final code to define the *equation* variable, $\mathring{E}_i^{(1)}$.

The problem's boundary conditions are treated similarly. Integrating Equation (2) over each boundary cell yields

$$\mathring{E}_i^{(2)} = \mathring{U}_i^{(1)} - \mathring{D}_i^{(3)}, \quad i \in \mathbb{I}_{\text{boundary}}, \tag{14}$$

where Equation (6) is used to represent ϕ and a third *derived* variable is defined via

$$\mathring{D}_i^{(3)} = \phi_{\text{boundary}}(x_i) \approx \frac{1}{|V_i|}\int_{V_i} \phi_{\text{boundary}}(x)\, dV, \quad i \in \mathbb{I}_{\text{boundary}}. \tag{15}$$

Here x_i is the centroid location of cell i. The function $\phi_{\text{boundary}}(x)$ is supplied by the user. The boundary condition is satisfied when the cell centred *equation* variable $\mathring{E}_i^{(2)}$ is zero in each boundary cell within the set $\mathbb{I}_{\text{boundary}}$.

3 Pseudo-mathematical expression language

Using the diffusion example we showed how a PDE can be discretised into a set of nonlinear equations that involves four code variable types; *unknown*, *derived*, *local* and *equation*. Equations for the *derived* and *equation* variables are specific to each problem, and so need to be entered by the user. These equations are specified using a pseudo-mathematical language that employs a number of finite volume specific operators. Each operator acts on an expression of one centring, producing a variable of (possibly) another centring.

To illustrate, the following code defines the PDE and boundary condition *equation* variables for the example diffusion problem:

```
CELL_EQUATION <phi domain equation> "celldiv(<Gamma>*
   facegrad(<phi>))-<Lambda>" ON <domain>
CELL_EQUATION <phi boundary equation> "<phi>-<phi
   boundary>" ON <boundary cells>
```

The expression for each code variable is contained in the double quotations. Comparing the mathematical equations with the finite volume expressions (that is, Equations (1) and (2) with the above) illustrates the key operators. For example, the relationship between the mathematical, FVM meaning and FVM discretisation for the operator `celldiv` is

$$\texttt{celldiv} : \boldsymbol{\nabla}\cdot \rightarrow \frac{1}{|V_i|}\int_{V_i} \boldsymbol{\nabla}\cdot\, dV \rightarrow \sum_{j\in\mathbb{J}_{\text{cellfaces},i}} \dot{d}_{i,j}\,. \tag{16}$$

Hence this operator acts on face centred quantities and produces a cell centred result; namely the divergence of a vector field (as represented by face centred vector components). Similarly, `faceave` produces a face centred average from surrounding cell centred data,

$$\texttt{faceave} : \text{N/A} \rightarrow \frac{1}{|S_j|}\int_{S_j} dS \rightarrow \sum_{i\in\mathbb{I}_{\text{facecells},j}} \mathring{k}^{(0)}_{j,i}\,, \tag{17}$$

whereas `facegrad` produces a face centred gradient taken normal to that face, also from cell centred data,

$$\texttt{facegrad} : \boldsymbol{n}_j\cdot\boldsymbol{\nabla} \rightarrow \frac{1}{|S_j|}\int_{S_j} \boldsymbol{n}_j\cdot\boldsymbol{\nabla}\, dS \rightarrow \sum_{i\in\mathbb{I}_{\text{facecells},j}} \mathring{k}^{(1)}_{j,i}\,. \tag{18}$$

Other operators allow cell and face centred gradient evaluation (in various directions), face-to-cell averaging, conditional statements, region identification, advection averaging (high and low order upwinding), sums and products [7].

Computationally the pseudo-mathematical expressions are converted into Fortran source code via a ‘metaprogramming’ algorithm written in Perl.

Specifically, this algorithm generates the Fortran code by recursively parsing each expression,

- searching for any operators, and if found, creating new *local* variables to represent the appropriate discretised operation;
- checking the centring of all code variables used, and if not consistent with the context of the expression, averaging the variable to the appropriate location; and
- using the computer algebra system Maxima [10] to simplify each mathematical expression, calculate any required partial derivatives (discussed later) and output the equivalent Fortran code.

In effect this metaprogramming algorithm works through the discretisation methodology described in the previous section, for any equation that can be expressed using the mathematical capabilities of Maxima and the finite volume operators described above.

4 Newton–Raphson solution procedure

This section details the method for solving the discretised equations. To aid our description, the code variables defined for the example diffusion problem are combined sequentially into four vectors, one for each variable type. Specifically,

$$\mathbf{U} = \left(\mathring{U}_i^{(1)} \mid i \in \mathbb{I}\right), \tag{19}$$

$$\mathbf{D} = \left(\mathring{D}_i^{(1)} \mid i \in \mathbb{I}_{\text{domain}} \quad , \quad \dot{D}_j^{(2)} \mid j \in \mathbb{J} \quad , \quad \mathring{D}_i^{(3)} \mid i \in \mathbb{I}_{\text{boundary}}\right), \tag{20}$$

$$\mathbf{L} = \left(\dot{L}_j^{(1)} \mid j \in \mathbb{J} \quad , \quad \dot{L}_j^{(2)} \mid j \in \mathbb{J} \quad , \quad \mathring{L}_i^{(3)} \mid i \in \mathbb{I}_{\text{domain}}\right), \tag{21}$$

$$\mathbf{E} = \left(\mathring{E}_i^{(1)} \mid i \in \mathbb{I}_{\text{domain}} \quad , \quad \mathring{E}_i^{(2)} \mid i \in \mathbb{I}_{\text{boundary}}\right). \tag{22}$$

The problem to solve is then written as

$$\mathsf{E}\left(\mathsf{L},\mathsf{D},\mathsf{U}\right)=\mathsf{0}\,,\quad \text{or}\quad \mathsf{E}\left(\mathsf{U}\right)=\mathsf{0}\,, \tag{23}$$

noting that L and D can be expressed as functions of only U.

A backstepped Newton–Raphson method [9] is employed to solve Equation (23). That is, if U^n is an estimate of the solution, then a better estimate is

$$\mathsf{U}^{n+1}=\mathsf{U}^{n}-\lambda\left[\left.\frac{d\mathsf{E}}{d\mathsf{U}}\right|_{\mathsf{U}^n}\right]^{-1}\cdot\mathsf{E}(\mathsf{U}^n) \tag{24}$$

where $\lambda\leqslant 1$ is a backstepping parameter, generally chosen to be a maximum such that $\left\|\mathsf{E}'(\mathsf{U}^{n+1})\right\|_2<(1-2\alpha\lambda)\left\|\mathsf{E}'(\mathsf{U}^{n})\right\|_2$. Here α is a small positive number [9, 8] and the prime indicates that each equation is individually normalised using an order of magnitude estimate (discussed below). Equation (24) is applied sequentially until $\left\|\mathsf{E}'(\mathsf{U}^{n+1})\right\|_2<E_{tol}$, where E_{tol} is a requested tolerance parameter (typically 10^{-12}).

The matrix $d\mathsf{E}/d\mathsf{U}|_{\mathsf{U}^n}$ used in equation (24) is the Jacobian of E, evaluated at U^n. Certain properties of this Jacobian must be satisfied for it to be invertible, and hence allow a solution to the overall Equation (23). We now draw a link between these properties and the discretised system, in the process specifying several conditions on the underlying equations that must be satisfied for a solution to be possible.

1. For a matrix to be invertible, it must be square. Hence, the structure of the Jacobian shows that the total number of *equation* variables defined by the user must equal the number of *unknown* variables. In the example diffusion problem of Section 2 for example, there was one *unknown* variable and one *equation* variable associated with each cell, thus an equal number ($|\mathbb{I}|$) of each. While the total number of each of these code variables must be equal, they need not have a one-to-one correspondence with each of the cells contained within the mesh. Indeed, *equation* and *unknown* variables may be associated with faces rather than cells, or conversely have no centring at all.

2. For the Jacobian to be invertible, it must have a rank equal to its order (that is, $\operatorname{rank}(d\mathbf{E}/d\mathbf{U}) = |\mathbb{I}|$). Equivalently, each row within the Jacobian must be linearly independent.

   The first implication of this linear independence is that each row of the Jacobian must have at least one nonzero element. In terms of the underlying mathematical problem this means that each *equation* variable must have at least one nonzero partial derivative with respect to an *unknown* variable throughout the entire solution process.

   The second implication is as follows. If the rows of the Jacobian were not linearly independent, then for a particular j there would be a solution to

$$\left.\frac{dE_j}{d\mathbf{U}}\right|_{\mathbf{U}^n} = \sum_{i\in\mathbb{I},\, i\neq j} a_i \left.\frac{dE_i}{d\mathbf{U}}\right|_{\mathbf{U}^n} \tag{25}$$

   such that at least one a_i is nonzero. Now, consider an example system where one of the equations can be expressed as a (nonlinear) function of only the other equations, that is

$$E_j = f(E_i : i \in \mathbb{I} \mid i \neq j). \tag{26}$$

   Differentiating using the chain rule gives

$$\frac{dE_j}{d\mathbf{U}} = \sum_{i\in\mathbb{I},\, i\neq j} \frac{\partial f}{\partial E_i}\frac{dE_i}{d\mathbf{U}}\,. \tag{27}$$

   Identifying the $\partial f/\partial E_i$ scalars from this equation with the a_i constants from Equation (25) shows that if Equation (26) is true for any $j \in \mathbb{I}$ then the rows of the Jacobian will not be linearly independent (for any $\mathbf{U}^n$), and the Jacobian will not be invertible.

   In terms of the FVM discretisation, this means that each *equation* variable defined by the user, when expressed solely in terms of the *unknown* variables, must be nonlinearly independent (that is, not able to be expressed solely as a function of the other *equation* variables). For

example, it would not be possible to specify via equations that all of the components of a vector *as well as* the magnitude of that vector be zero at a given location, as this set of equations is nonlinearly dependent.

3. If the *unknown* variable u_i has a typical order of magnitude of $\mathcal{O}(u_i)$, then the representation of this variable using floating point arithmetic will be limited to an accuracy of $\varepsilon(u_i) = E_{\text{mach}}\mathcal{O}(u_i)$, where E_{mach} is the 'machine precision' being employed. Equivalently, defining the two vectors $\varepsilon(\mathbf{u})$ and $\mathcal{O}(\mathbf{u})$,

$$\varepsilon(\mathbf{u}) = E_{\text{mach}}\mathcal{O}(\mathbf{u}). \tag{28}$$

The precision to which $\mathbf{u}$ can be represented has implications for the minimum possible $\left\|\mathbf{E}(\mathbf{u}^{n+1})\right\|_2$ that can be achieved. Specifically, an order of magnitude analysis on Equation (24) shows that the accuracy of the next, 'improved' estimate for $\mathbf{E}$ will be at best

$$\varepsilon\left(\mathbf{E}(\mathbf{u}^{n+1})\right) = \left.\frac{d\mathbf{E}}{d\mathbf{u}}\right|_{\mathbf{u}^n} \cdot \varepsilon(\mathbf{u}) = E_{\text{mach}} \left.\frac{d\mathbf{E}}{d\mathbf{u}}\right|_{\mathbf{u}^n} \cdot \mathcal{O}(\mathbf{u}), \tag{29}$$

where we used $\mathcal{O}(\lambda) = 1$. Thus, in general, successive Newton–Raphson iterations will not be able to decrease $\mathbf{E}(\mathbf{u}^{n+1})$ below this value.

In practice we use this result to calculate an order of magnitude estimate for each *equation* variable,

$$\mathcal{O}(\mathbf{E}) = \max\left[\mathbf{E}(\mathbf{u}^0), \left.\frac{d\mathbf{E}}{d\mathbf{u}}\right|_{\mathbf{u}^0} \cdot \mathcal{O}(\mathbf{u})\right], \tag{30}$$

and use these estimates to normalise $\mathbf{E}$ prior to calculating the residual norms $\left\|\mathbf{E}'\right\|_2$. As indicated, each *equation* magnitude estimate is based on initial values for both $\mathbf{E}$ and $d\mathbf{E}/d\mathbf{u}$, and user supplied order estimates for $\mathbf{u}$.

To apply Equation (24) numerically, both $\mathbf{E}$ and $d\mathbf{E}/d\mathbf{u}$ need to be calculated using the current best *unknown* estimates $(\mathbf{u}^n)$. To calculate $\mathbf{E}$, each *derived*

variable is first evaluated in the order of its definition. This ensures that each evaluation depends on only *unknown* and other *derived* variables that have already been evaluated, and is hence explicit. The *equation* variables are then evaluated. Note that *local* variables are not stored but rather calculated as they are needed using a recursively called subroutine. Hence, *local* variables do not need to be defined or calculated in any particular order.

The Jacobian matrix is evaluated using the chain rule from current values of $\mathbf{U}$, $\mathbf{D}$ and $\mathbf{L}$, and calculated concurrently with $\mathbf{E}$. Analytical expressions for the partial derivatives required for the Jacobian are found during the metaprogramming step (discussed previously) and hence are explicitly included within the Fortran executable.

Code variables are stored in the Fortran executable using a declared type that contains not only the current value for the variable, but also an allocatable float and integer array within which are stored any nonzero partial derivatives the variable currently has, and with which *unknown* variable that derivative corresponds to. Packaged sparse linear solvers are used to numerically solve Equation (24) [11, 2].

5 Example nonlinear diffusion equation results

As an example, the following system of equations was solved

$$\begin{aligned} \text{Equation:} \quad & \nabla \cdot \Gamma(\phi)\nabla\phi - \Lambda(\phi) = 0 \\ \text{Derived functions:} \quad & \Gamma(\phi) = \phi^2 \\ & \Lambda(\phi) = 2\phi^3 \left(\frac{1}{(1+x)^2} + \frac{1}{(1+y)^2} + \frac{1}{(1+z)^2} \right) \\ \text{Boundary Conditions:} \quad & \phi = (1+x)(1+y)(1+z) \\ \text{Solution:} \quad & \phi = (1+x)(1+y)(1+z) \end{aligned} \tag{31}$$

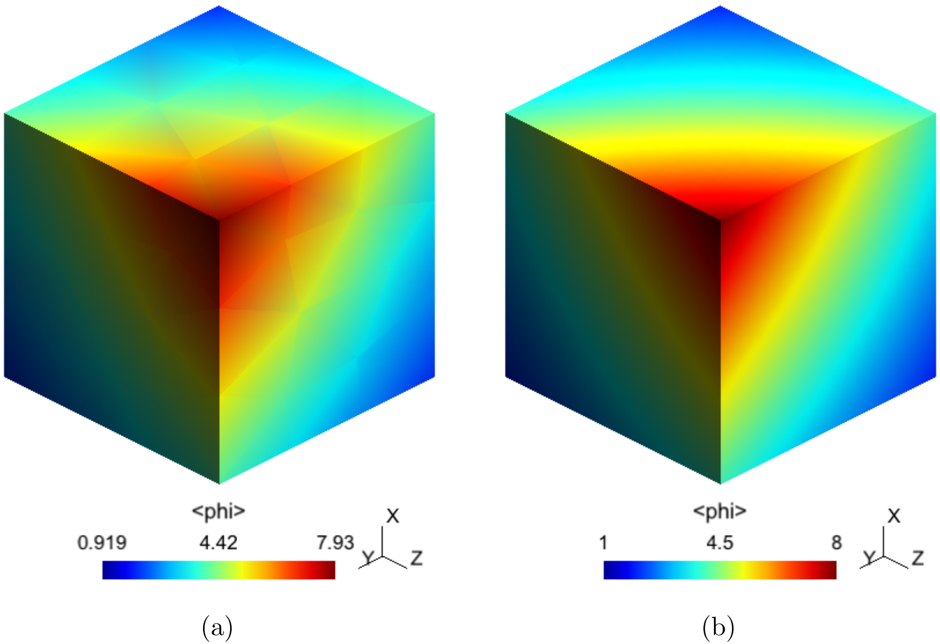

Figure 2: Frames (a) and (b) show ϕ for the example problem specified by Equations (31), computed using mesh sizes of $|\mathbb{I}| = 380$ and 612968 cells, respectively.

over a three dimension region ($0 \leqslant x, y, z \leqslant 1$), and the results compared against the analytical solution.[1] The unstructured tetrahedron meshes were created with the program Gmsh [5]. Figures 2a and 2b show example ϕ fields computed using representative coarse and fine meshes. Figure 3 shows various norms for the relative difference between the numerically generated and analytical ϕ solutions, plotted against average cell size, which for this three dimensional problem is approximately $|\mathbb{I}|^{-1/3}$. The error approaches zero as

[1] http://www.chemeng.unimelb.edu.au/people/staff/daltonh/downloads/arb/code/latest/examples/manual/cube_laplacian_dhctac10_2012 provide the input files for this problem.

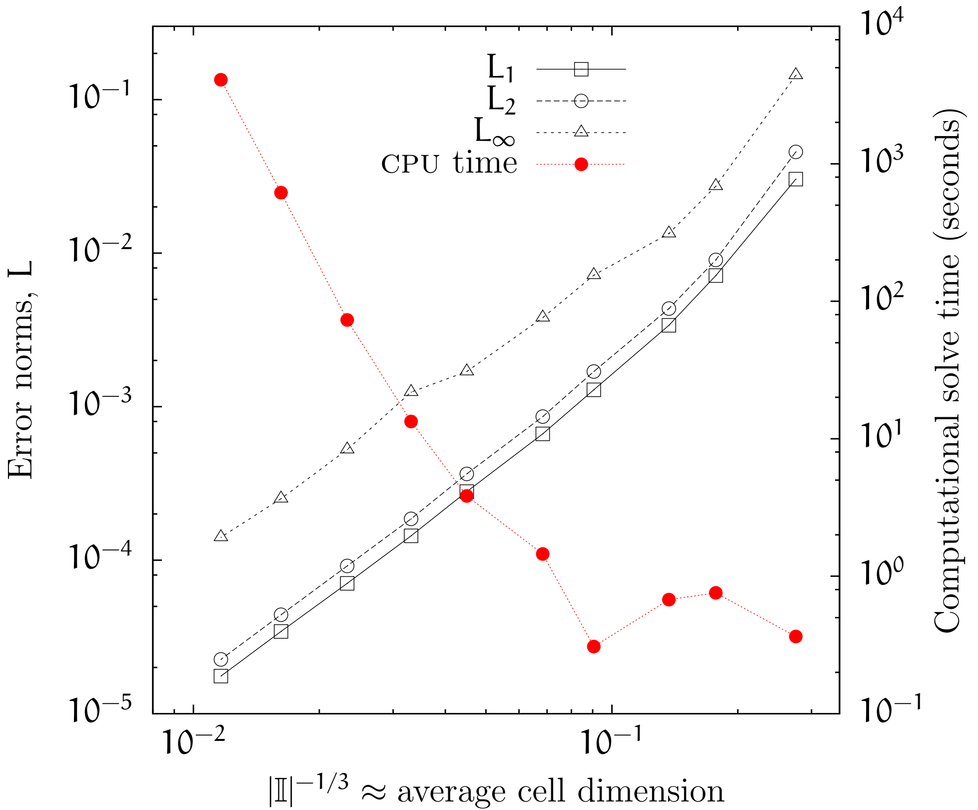

Figure 3: Various error norms for the numerical solution, and total computational time taken, both as a function of average cell dimension, for the computations of Figure 2.

the cell size is decreased, indicating that the discretisation method for this set of equations is consistent. Also shown in Figure 3 is the computational time required for each simulation. Once the mesh is sufficiently fine, the error norms shown decrease by a factor of approximately 2.0 with increasing mesh refinement, with computational time increasing by a corresponding factor of approximately 5.5.

6 Conclusion

A mathematical and computational framework has been described that generates finite volume code for solving multiphysics transport problems. Within the framework, equations are specified using pseudo-mathematical expressions that combine the mathematical capabilities of the symbolic algebra package Maxima with a number of defined finite volume operators. Hence, while only a single nonlinear diffusion equation has been discussed in this work, quite arbitrary systems of equations can be solved within this framework. In future work we will apply the method to a greater range of physical systems.

References

[1] Luis Cueto-Felgueroso, Ignasi Colominas, Xesus Nogueira, Fermin Navarrina, and Manuel Casteleiro. Finite volume solvers and moving least-squares approximations for the compressible Navier–Stokes equations on unstructured grids. *Computer Methods in Applied Mechanics and Engineering*, 196(45–48):4712–4736, 2007. doi:10.1016/j.cma.2007.06.003. C1133

[2] Timothy A. Davis. Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, 30(2):196–199, June 2004. ISSN 0098-3500. doi:10.1145/992200.992206. http://www.cise.ufl.edu/research/sparse/umfpack/. C1140

[3] J. H. Ferziger and M. Perić. *Computational Methods for Fluid Dynamics*. Springer-Verlag, 3rd edition, 2002. C1131

[4] CSC IT Center for Science. Elmer: Open source finite element software for multiphysical problems. http://www.csc.fi/english/pages/elmer/. Accessed 31/1/11. C1129

[5] Christophe Geuzaine and Jean-Francois Remacle. Gmsh: A 3-d finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009. ISSN 1097-0207. doi:10.1002/nme.2579. C1141

[6] The COMSOL Group. Comsol multiphysics. http://www.comsol.com/products/multiphysics/. Accessed 31/1/11. C1129

[7] Dalton Harvie. arb manual: version 0.25, 2011. http://www.chemeng.unimelb.edu.au/people/staff/daltonh/downloads/arb/code/manual.pdf. C1135

[8] J. E. Dennis Jr. and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, 1983. ISBN 0-13-627216-9. C1137

[9] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in FORTRAN: The art of scientific computing*. Cambridge University Press, Second edition, 1992. C1137

[10] William Schelter, Maxima Users, and Developers Group. Maxima: A computer algebra system, 2011. http://maxima.sourceforge.net/. C1136

[11] Olaf Schenk and Klaus Gartner. Solving unsymmetric sparse systems of linear equations with pardiso. *Future Generation Computer Systems*, 20(3):475–487, 2004. ISSN 0167-739X. doi:10.1016/j.future.2003.07.011. http://www.pardiso-project.org/. Selected numerical algorithms. C1140

Author address

1. **D. J. E. Harvie**, Department of Chemical and Biomolecular Engineering, University of Melbourne, Parkville, Victoria 3010, Australia.
mailto:daltonh@unimelb.edu.au