Rapid development and adjoining of transient finite element models

Rapid development and adjoining of transient finite element models

J. R. Maddison[a,c,*], P. E. Farrell[c,d]

[a] *School of Mathematics and Maxwell Institute for Mathematical Sciences, University of Edinburgh, Edinburgh EH9 3JZ, United Kingdom*
[b] *Atmospheric, Oceanic and Planetary Physics, Department of Physics, University of Oxford, Oxford OX1 3PU, United Kingdom*
[c] *Mathematical Institute, University of Oxford, Oxford OX2 6GG, United Kingdom*
[d] *Center for Biomedical Computing, Simula Research Laboratory, 1325 Lysaker, Norway*

Abstract

Recent advances in high level finite element systems have allowed for the symbolic representation of discretisations and their efficient automated implementation as model source code. This allows for the extremely compact implementation of complex non-linear models in a handful of lines of high level code. In this work we extend the high level finite element FEniCS system to introduce an abstract representation of the temporal discretisation: this enables the similarly rapid development of transient finite element models. Efficiency is achieved via aggressive optimisations that exploit the temporal structure, such as automated pre-assembly and caching of forms, and the robust re-use of matrix factorisations and preconditioner data. The resulting models are as fast or faster than hand-optimised finite element codes. The high level representation of the system remains extremely compact and easily manipulated. This structure is exploited to derive the associated discrete adjoint model automatically, with the adjoint model inheriting the performance advantages of the forward model. Combined, this provides a system for the rapid development of efficient transient models, together with their discrete adjoints.

Keywords: automated code generation, finite element method, discrete adjoint, Navier-Stokes, barotropic vorticity, FEniCS

1. Introduction

1.1. Automated code generation in computational science

Automated code generation is a crucial tool in computational science, as it allows scientists and engineers to express the structure of an algorithm in notation close to its

*Corresponding author
Email addresses: j.r.maddison@ed.ac.uk (J. R. Maddison), patrick.farrell@maths.ox.ac.uk (P. E. Farrell)
Telephone: +44 131 6505036 (J. R. Maddison)

mathematical formulation. It raises the level of abstraction at which computers may be used, shields programmers from low level details of how a solution is to be obtained on a particular machine, and allows for the solution of problems which would otherwise be too costly or too complex to program. For example, in the preliminary design document for the FORTRAN programming language, Backus [1] writes:

> FORTRAN will comprise a large set of programs to enable the IBM 704 to accept a concise formulation of a problem in terms of a mathematical notation and to produce automatically a high speed 704 program for the solution of the problem ... [S]uch a system will make experimental investigation of various mathematical models and numerical methods more feasible and convenient both in human and economic terms.

In the decades since this same motivation of reflecting mathematical structure in code led to the development of other environments in which higher level problems may be expressed, such as the wildly successful MATLAB environment for numerical linear algebra [2].

The FEniCS project [3] aims to develop a software environment for the automated solution of partial differential equations (PDEs) via the finite element method. In particular, it allows the user to specify a variational form representation of the model equations in the Unified Form Language (UFL, [4, 5]), which closely mimics the mathematical notation in which finite element discretisations may be written. The UFL representation of a problem is automatically compiled by a dedicated form compiler [6] into efficient C++ code, much as FORTRAN code is compiled by a dedicated compiler into efficient machine code.

The UFL abstraction for the spatial discretisation of PDEs has a number of important advantages. As the mathematical structure of the PDE is available for analysis, important equation-specific optimisations may be performed that low level compilers cannot automate, or that would be too laborious to implement by hand [7, 8]. UFL is remarkably compact: a finite element discretisation that might take thousands or tens of thousands of lines of FORTRAN or C++ code to implement can be cleanly expressed in just a handful of lines of UFL. For example, even the complicated elliptic relaxation turbulence model [9] can be represented in eleven lines of UFL code (ignoring boundary conditions) [10]. As UFL expresses *what* problem is to be solved, without specifying *how* it is to be solved, the system is free to adapt the implementation to the hardware, including GPUs [11, 12, 13]. Lastly, the ability to analyse the mathematical structure of the equations vastly simplifies the task of algorithmic differentiation, and allows for the fully automated derivation of the adjoint model associated with a given forward model [14]; these adjoint models can in turn be used to automatically solve PDE-constrained optimisation problems [15] and conduct generalised stability analyses [16]. Adjoint models will be discussed further in section 1.2.

However, UFL lacks a native representation for specifying time-dependent problems: in general the user must perform the temporal discretisation by hand and implement it as a sequence of spatial problems. With this manual approach the FEniCS system is

unable to automatically perform optimisations that exploit the temporal structure of the discretisation, such as the pre-assembly and caching of terms that occur repeatedly, and the reuse of preconditioners or factorisations in the linear solvers. These optimisations are crucial for efficient implementations of timestepping models, but the user must add them by hand. Some limited support for special types of time dependent problems within the FEniCS system is in development – here we address the general case.

Aside from the unnecessary labour, the absence of temporal abstraction has a major disadvantage: *the implementation of the temporal optimisations breaks the spatial abstraction.* The model can no longer be cleanly expressed as a list of variational problems to be solved; the user must manually pre-assemble certain terms outside of the time loop, assemble other terms inside the time loop, and express the problem at the level of matrices and vectors. Firstly, this loss of structure makes the code significantly harder to read, understand, debug and modify. The expression of the problem and guidance for how it should be solved have been irretrievably interwoven. Secondly, it damages performance portability: for example, on a GPU it may be preferable to recompute terms, rather than cache them, but the user has irrevocably committed to one strategy or the other. Thirdly, it significantly hampers the automated derivation of adjoint models, as the variational structure of the time-dependent problem must be pieced together from the lower-level implementation.

In this paper we introduce an abstraction for expressing time-dependent models, and apply this methodology in combination with the FEniCS system. This temporal abstraction allows for the high level expression of both the spatial and temporal structure of the problem, and resolves all the aforementioned disadvantages. The user can specify the spatial discretisation in the native UFL format, while applying the temporal optimisations necessary for efficiency. In particular, the task of adjoint derivation of timestepping models is greatly simplified, and the entire suite of temporal optimisations may be applied to the adjoint model.

For all its advantages, high-level systems that rely on automated code generation also come with drawbacks. Such systems are more complex, with more layers of abstraction, and can thus be less robust and harder to debug. There is a trade-off between the flexibility and generality of lower-level approaches, and the rapid speed and ease of development associated with higher-level approaches; such systems usually lack escape hatches to implement algorithms not expressible in the high-level abstraction. However, the advantages of using high-level systems often greatly outweigh the disadvantages, especially in the typical case where the main constraint on the development of scientific software is the availability of programmer time.

1.2. Adjoint models

The topic of adjoints is fundamental to the theory of differential and integral equations [17, pg. 147], and connects to almost everything in the computational mathematics of PDEs. To mention some examples of relevance to the computational physicist, adjoints are central to *a priori* and *a posteriori* error estimation [18, 19, 20, 21, 17], sensitivity studies [22, 23], PDE-constrained optimisation [24, 25, 26, 27, 28], non-normal stability analysis [29, 30, 31], and PDE-constrained Bayesian inference [32].

A forward model maps some inputs (initial conditions, boundary conditions, physical properties, etc.) to outputs (solution fields, and functionals of those solution fields), generally in a non-linear manner. Its linearisation (the so-called *tangent linear model*) linearly propagates the effect of a single perturbation in an input to the resulting perturbations in all outputs. This can be used to compute directional derivatives – a generalisation of the gradient, defining the derivative of a functional with respect to arbitrary perturbations in input parameter functions (see Appendix B). The Hermitian transpose of the linearisation (the so-called *adjoint model*) linearly propagates causality in the transpose sense, from a single perturbation in an output back to the perturbations in the many inputs that caused it. Intuitively, the tangent linear model computes the many effects of a single cause, while the adjoint model computes the many causes of a single effect.

Developing adjoint models is widely considered to be very difficult, especially for time-dependent models. This has been a major impediment to the widespread application of the advanced techniques that depend on adjoints. For example, Giles and Pierce [33] state:

> Considering the importance of design to aeronautical engineering, and indeed to all of engineering, it is perhaps surprising that the development of adjoint CFD codes has not been more rapid ... [I]t seems likely that part of the reason is its complexity.

This difficulty is one of the main motivations for the field of algorithmic differentiation (AD; see, e.g., [34]).

A core aim of AD is to enable the automated derivation of adjoint models from the forward model source code, for example by means of a source-to-source tool such as TAPENADE [35], TAF [36], or OpenAD [37]. These tools proceed by considering each elementary operation performed by the forward model, differentiating each in turn, and composing the result with the chain rule. In this context, an "elementary operation" refers to the mathematical functions available natively in C or FORTRAN, such as a single addition, multiplication, or trigonometric function evaluation. This approach has the significant benefit that, given a forward model which can be processed by the AD tool, an adjoint model can (in principle) be generated automatically, without the need to differentiate and adjoin the entire model by hand. AD has seen a diverse range of successful practical applications, including for example in engineering design [38, 39] and large-scale ocean state estimation [40, 41]. However, the black-box line-by-line application of AD without any consideration of the mathematical structure of the problem can be too inefficient for practical use [42]. In addition the implementation of an adjoint model is significantly hampered by the need to access forward model data in the *reverse* order to which it was computed, necessitating the use of data checkpointing and recovery strategies.

In recent work, an alternative higher level approach has been proposed [14], imple-

mented in the dolfin-adjoint library[1]. Instead of interpreting the elementary instructions appearing in low level source code such as C or FORTRAN, the elementary instructions are reinterpreted at the highest possible level of mathematical abstraction; in particular, in the finite element case, this means interpreting the program as a sequence of variational problems. This allows for the exploitation of the mathematical structure of the model, which has several key advantages: the adjoint models are derived with the minimum of user intervention, are measured to closely approximate optimal theoretical efficiency[2], parallelise naturally, and make use of the optimal checkpointing strategy of Griewank and Walther [43].

However for time dependent problems the variational structure of a finite element model may be obscured. If a structured discretisation is applied in time then the time dependent model consists of a very large number of individual spatially discrete variational problems. Temporal optimisations may be applied which exploit the resulting temporal structure, but the optimisations easily obscure the high level representation of the model equations. For example, it is often far more efficient to cache matrices, rather than re-assemble the variational forms used to describe them; thus, the equations must be written in terms of linear algebra operations, rather than in variational form. The use of general linear algebra operations allows for a more efficient model implementation, but requires a very general AD tool if an adjoint model is to be derived. Restriction to variational forms prohibits the use of specific optimisation strategies, but can be adjoined via direct symbolic manipulation of equations.

This issue strongly motivates the development of an abstraction for time dependent problems which retains the variational structure of the spatially discrete model equations, includes a description of the time discretisation, and permits the application of aggressive temporal optimisations. Such an approach can ensure an efficient model implementation while retaining a high level structure which can be manipulated and adjoined via the methodology of Farrell et al. [14].

Section 2 discusses the structure of a transient model and its discrete adjoint, and uses this discussion to motivate the elements required in order to provide a high level representation of model timestepping. Section 3 describes the implementation of a tool enabling such a high level representation, building on top of the FEniCS system. Section 4 provides two more complex examples, and the paper concludes in section 5.

2. Structure of timestepping models

In this section the high level structure of a generic timestepping model is outlined. In particular a timestepping model is broken into three key stages: initialisation, timestepping, and finalisation. Each of these stages can be further sub-divided into discrete equations. Crucially, the timestepping stage contains a regular repeating structure, which

[1] `http://dolfin-adjoint.org/`

[2] This is an idealised estimate of how fast the adjoint model can be, by counting how many linear systems the forward and adjoint models must assemble and solve.

has a compact high level representation and is amenable to optimisation strategies.

Section 2.1 discusses the general structure of a timestepping model, and section 2.2 discusses the structure of the associated discrete adjoint model. Section 2.3 uses these discussions to identify the functionality required for a high level representation of a timestepping model.

2.1. Block structure of a timestepping model

Let x be the entire solution of a time dependent numerical model. This solution vector contains the values of all fields at all times. Then the discrete numerical model may in general be written as:

$$A(x)\,x = b(x)\,, \tag{1}$$

where the left-hand-side matrix $A(x)$ and right-hand-side vector $b(x)$ are, if the model is non-linear, dependent upon the solution vector x. Note that equation (1) is a purely conceptual representation — typically, the one-shot construction of the entire system (1) is impractical. Instead, the causal structure of the time-dependent model is exploited: later values depend on earlier ones but, for an initial value problem, not vice versa. Hence for a time-dependent initial value problem the solution vector x may always be defined so that the matrix $A(x)$ is block lower-triangular. It is assumed throughout the remainder of this section that such a definition for x is used.

The entire system (1) may now be divided into a number of coupled sub-systems, corresponding to the division of x into sub-vectors, with the corresponding division of $A(x)$ into distinct matrix blocks and $b(x)$ into corresponding sub-vectors. In a numerical model these sub-systems are solved in sequence via forward substitution. The entire model system may, for example, be divided into sub-systems corresponding to individual discrete equations solved at individual time levels. The system may, alternatively, be divided into a very large number of sub-systems corresponding to the individual elementary instructions appearing in model source code. The fundamental approach used in this article is that the model system should be divided into the largest sub-systems which can be conveniently defined and manipulated. Automated code generation allows a numerical model to be described and implemented via the specification of discrete model equations, and dolfin-adjoint manipulates such a description of a finite element model to yield the associated discrete adjoint — this corresponds to the former division of the model system into relatively large sub-systems corresponding to model equations. On the other hand, implementation in a lower level language such as C or FORTRAN requires a numerical model to be described and implemented via the specification of individual elementary instructions, and algorithmic differentiation tools manipulate such an implementation to yield the associated discrete adjoint — this corresponds to the latter division of a model into a very large number of small sub-systems.

We initially consider the division of the model system into even larger sub-systems, corresponding to three distinct stages: initialisation, timestepping, and finalisation. It is assumed that every timestep has an identical form: the same equations are solved, with different inputs (and possibly different parameter values) at every timestep. The

timestepping stage therefore has a regular repeating structure, with the structure potentially repeated a very large number of times [44]. The timestepping stage is thus divided into distinct timesteps. Each timestep is then further divided into two distinct stages: the timestep variable *solve* stage, in which new field values are computed using old field values, and the timestep variable *cycle* stage, in which old field values are replaced using the new field values. The timestep solve stage comprises the discrete model equations, while the timestep cycle stage consists of simple assignments. In principle the introduction of a timestep cycle stage is not required. However, this reflects the way in which timestepping models are often written – in a timestep old field data are used to compute new field data, and then the old field data are updated ahead of the next timestep.

Let x^n be a sub-vector of x corresponding to data associated with timestep n, with x^0 corresponding to the initial condition and x^N corresponding to the final solution. Let $x^{N,\times}$ correspond to the solution after the finalisation stage. The existence of a finalisation stage allows for the one-time calculation of final model diagnostics. Then the initialisation stage takes the form:

$$A_D^0\left(x^0\right)x^0 = b^0\left(x^0\right), \tag{2}$$

for some matrix $A_D^0\left(x^0\right)$ and vector $b^0\left(x^0\right)$ (which may depend upon the solution vector x^0 if the initialisation equation is non-linear). The timestep variable solve stage takes the form:

$$M_D\left(x^n, x^{n+1,+}\right)x^{n+1,+} = -M_O\left(x^n, x^{n+1,+}\right)x^n + c\left(x^n, x^{n+1,+}\right), \tag{3}$$

where $x^{n+1,+}$ are the newly computed field values, $M_D\left(x^n, x^{n+1,+}\right)$ and $M_O\left(x^n, x^{n+1,+}\right)$ are some matrices, and $c\left(x^n, x^{n+1,+}\right)$ is some vector (noting that the matrices and this vector may again depend upon the solution vectors in a non-linear model). The timestep variable cycle stage takes the form of a simple assignment:

$$x^{n+1} = x^{n+1,+}. \tag{4}$$

Hence, in this notation, x^n corresponds to the *old* field values, used to compute the *new* field values $x^{n+1,+}$. The finalisation stage takes the form:

$$A_D^N\left(x^N, x^{N,\times}\right)x^{N,\times} = -A_O^N\left(x^N, x^{N,\times}\right)x^N + b^N\left(x^N, x^{N,\times}\right), \tag{5}$$

where $A_D^N\left(x^N, x^{N,\times}\right)$ and $A_O^N\left(x^N, x^{N,\times}\right)$ are some matrices and $b^N\left(x^N, x^{N,\times}\right)$ is some

vector. The model system therefore takes the form of (1), with:

$$A = \begin{pmatrix} A_D^0 & & & & & & & & & \\ M_O & M_D & & & & & & & & \\ & -I & I & & & & & & & \\ & & M_O & M_D & & & & & & \\ & & & -I & I & & & & & \\ & & & & & \ddots & & & & \\ & & & & & & M_O & M_D & & \\ & & & & & & & -I & I & \\ & & & & & & & & A_O^N & A_D^N \end{pmatrix},$$

$$x = \begin{pmatrix} x^0 \\ x^{1,+} \\ x^1 \\ x^{2,+} \\ x^2 \\ \vdots \\ x^{N,+} \\ x^N \\ x^{N,\times} \end{pmatrix}, \qquad b = \begin{pmatrix} b^0 \\ c \\ 0 \\ c \\ 0 \\ \vdots \\ c \\ 0 \\ b^N \end{pmatrix}, \tag{6}$$

where I is an identity matrix, and where explicit dependencies have been dropped for clarity.

Hence, with this sub-division of a timestepping model, the matrix A takes a block bi-diagonal lower-triangular form. The initialisation equation (2), timestep solve equation (3), and finalisation equation (5), can now be further subdivided, for example by division into individual discrete model equations. The timestepping abstraction library, described in section 3, provides a means of describing the inherent block structure captured by (6).

2.2. Block structure of an adjoint model

Let the model depend upon a number of parameters, encapsulated by the vector m. Define a functional $J\left(x\left(m\right), m\right)$, depending upon the model solution and the parameters m, where the solution itself depends upon the parameters m. Then the total derivative of the functional with respect to a particular parameter m_i is defined:

$$\frac{\mathrm{d}J}{\mathrm{d}m_i} = \lim_{\delta m_i \to 0} \frac{J\left(x\left(\ldots, m_i + \delta m_i, \ldots\right), \ldots, m_i + \delta m_i, \ldots\right) - J\left(x\left(\ldots, m_i, \ldots\right), \ldots, m_i, \ldots\right)}{\delta m_i}. \tag{7}$$

It follows directly [25] that:

$$\frac{\mathrm{d}J}{\mathrm{d}m_i} = \frac{\partial J}{\partial m_i} - \frac{\partial F}{\partial m_i} \cdot \lambda, \tag{8}$$

where $F(x,m) = A(x,m)\,x - b(x,m)$, and λ is the solution of the adjoint model equation:

$$\left[\frac{\partial F}{\partial x}\right]^* \lambda = \left[\frac{\partial J}{\partial x}\right]^*, \tag{9}$$

where $(\ldots)^*$ denotes the Hermitian transpose. If the adjoint model solution λ is known then one can compute the derivative with respect to many parameters, simultaneously, via (8).

The adjoint model system associated with a timestepping forward model therefore takes the form:

$$\left[\frac{\partial F}{\partial x}\right]^* = \begin{pmatrix} B_D^0 & P_O & & & & & & & \\ & P_D & -I & & & & & & \\ & & I & P_O & & & & & \\ & & & P_D & -I & & & & \\ & & & & I & P_O & & & \\ & & & & & & \ddots & & \\ & & & & & & & I & B_Q^N \\ & & & & & & & & B_D^N \end{pmatrix},$$

$$\lambda = \begin{pmatrix} \lambda^0 \\ \lambda^{1,+} \\ \lambda^1 \\ \lambda^{2,+} \\ \lambda^2 \\ \vdots \\ \lambda^N \\ \lambda^{N,\times} \end{pmatrix}, \qquad \left[\frac{\partial J}{\partial x}\right]^* = \begin{pmatrix} \left[\frac{\partial J}{\partial x^0}\right]^* \\ 0 \\ \left[\frac{\partial J}{\partial x^1}\right]^* \\ 0 \\ \left[\frac{\partial J}{\partial x^2}\right]^* \\ \vdots \\ \left[\frac{\partial J}{\partial x^N}\right]^* \\ \left[\frac{\partial J}{\partial x^{N,\times}}\right]^* \end{pmatrix}, \tag{10}$$

with:

$$B_D^0 = \left[\frac{\partial}{\partial x^0}\left(A_D^0 x_0 - b^0\right)\right]^*, \tag{11a}$$

$$P_O = \left[\frac{\partial}{\partial x^n}\left(M_D x^{n+1,+} + M_O x^n - c\right)\right]^*, \tag{11b}$$

$$P_D = \left[\frac{\partial}{\partial x^{n+1,+}}\left(M_D x^{n+1,+} + M_O x^n - c\right)\right]^*, \tag{11c}$$

$$B_O^N = \left[\frac{\partial}{\partial x^N}\left(A_D^N x^{N,\times} + A_O^N x^N - b^N\right)\right]^*, \tag{11d}$$

$$B_D^N = \left[\frac{\partial}{\partial x^{N,\times}}\left(A_D^N x^{N,\times} + A_O^N x^N - b^N\right)\right]^*, \tag{11e}$$

and where explicit dependencies have again been dropped for clarity.

Hence, with this sub-division of a timestepping model, the adjoint model matrix $[\partial F/\partial x]^*$ takes a block bi-diagonal upper-triangular form. If the forward model initialisation equation (2), timestep solve equation (3), and finalisation equation (5), are now

further subdivided, for example by division into discrete model equations, then the adjoint model blocks (11) can be constructed by differentiating the equations with respect to their dependencies, and then adjoining the resulting matrices.

In particular, if the discrete model equations correspond to finite element discretisations, then the UFL library supplies the symbolic manipulation tools required to construct components of the adjoint model blocks (11). This is precisely the methodology applied for more general finite element models in Farrell et al. [14]. However here, with the specification of a structure for a timestepping model, three key simplifications can be applied. First, only a reduced number of model equations need be considered, and the repeating structure of the timestepping model can then be exploited. Second, no "long range" dependencies are permitted – for example each forward timestep solve equation depends only upon x^n and $x^{n+1,+}$, and not on field data from any earlier timesteps. Third, many adjoint model blocks appearing in the adjoint system matrix $[\partial F/\partial x]^*$ are known to be identity matrices, associated with the timestep variable cycle stage. Combined, these properties significantly simplify the implementation of the methodology of Farrell et al. [14] when applied specifically to a timestepping model.

2.3. Requirements for a high level timestepping abstraction

Two key elements are required in order to construct a high level representation for a timestepping model. First, one must be able to describe the structure of the system represented by (6). In particular one must be able to describe the equations which comprise the initialisation, timestep solve, and finalisation stages, and to describe the sub-vectors x^n and $x^{n,+}$. Second, in order for certain time discretisation specific optimisations to be applied, one must be able to label certain model data, such as parameters or boundary conditions, as time independent. For example an equation term can be identified as time independent, and therefore amenable to caching, only if the parameters on which it depends are themselves known to be time independent. Explicit labelling of time independent data facilitates such analysis.

In order for a discrete adjoint model to be derived and implemented automatically a third key requirement is that equation dependencies should be easily identified, and that the timestepping model must be easily manipulated. It must be possible to perform the necessary differentiation and adjoining operations required to construct the adjoint model system represented by (10). It is also essential that adjoint model dependencies can be easily identified and recorded, so that forward model data can be checkpointed and regenerated as required by an adjoint model calculation.

3. A high level representation for timestepping models

In the previous section the conceptual structure of a timestepping model, and its associated discrete adjoint model, has been outlined. In this section the high level representation and automated implementation of a timestepping finite element model is described. The principles considered are general – one could consider the addition of a timestepping abstraction to a fairly general class of automated code generation tools. Here the specific application to the FEniCS system is considered.

Section 3.1 describes the implementation of a transient finite element model using only native FEniCS functionality, and highlights some issues that arise. Section 3.2 introduces a timestepping abstraction library, which adds a high level representation of a model time discretisation to the FEniCS system, and section 3.3 discusses the automated application of optimisations which exploit the temporal structure. Section 3.4 describes how such a representation may be used to derive an associated discrete adjoint model automatically and section 3.5 describes the automated verification of the derived discrete adjoint model.

3.1. Transient finite element models and DOLFIN

DOLFIN is a front end for the FEniCS system. This library provides the tools required to describe finite element spatial discretisations, and uses other tools in the FEniCS system to implement these discretisations as efficient working model code. However, no means of describing a general time discretisation is provided, and time dependent problems are typically constructed by hand. In this article only the Python interface to DOLFIN is considered. For complete documentation of DOLFIN and the FEniCS system, see Logg et al. [3].

As an exception, some limited support for the solution of time dependent problems within DOLFIN is in development. This adds functionality for special types of time dependent problems, including ODEs and the application of Runge-Kutta discretisations to a single time-dependent PDE (although the latter cannot be used to construct the model shown in this section). Here, instead, a general approach is sought, which can be used to construct complex time dependent models, and allows for the inclusion of multiple coupled time dependent equations which possibly have differing temporal discretisations.

Consider the 1D advection-diffusion problem for a tracer T:

$$\partial_t T + u\partial_x T = \kappa \partial_{xx} T \quad \text{on } x \in (0,1)\,, \tag{12a}$$

$$T = T_0 \quad \text{at } t = 0, \tag{12b}$$

$$T = 1 \quad \text{at } x = 0, \tag{12c}$$

where $u > 0$ and κ is the diffusivity. Let the space $\Omega = (0,1)$ be covered by a set of cells, and equip these cells with degree one Lagrange basis functions ϕ_i. Thus construct a $P1$ continuous Galerkin spatial discretisation with a simple streamline-upwind Petrov-Galerkin (SUPG) method in space [45, 46], and a Crank-Nicolson discretisation in time [47]:

$$\int_0^1 \psi_i T^{\delta,0} dx = \int_0^1 \psi_i T_0 dx, \tag{13a}$$

$$\begin{aligned}\int_0^1 \left(\psi_i + \frac{1}{2}\Delta x \frac{u}{|u|}\partial_x \psi_i\right) \left[\frac{1}{\Delta t}\left(T^{\delta,n+1} - T^{\delta,n}\right) + u\partial_x T^{\delta,n+\frac{1}{2}}\right] dx \\ = \left[\kappa\psi_i \partial_x T^{\delta,n+\frac{1}{2}}\right]_0^1 - \int_0^1 \kappa \partial_x \psi_i \partial_x T^{\delta,n+\frac{1}{2}} dx.\end{aligned} \tag{13b}$$

Here Δx is the cell size, Δt is the timestep size, $T^{\delta,n}$ is the solution at time level n, and $T^{\delta,n+\frac{1}{2}} = \frac{1}{2}\left(T^{\delta,n} + T^{\delta,n+1}\right)$. The test functions are chosen so that $\psi_i \in \{\phi_i : \phi_i|_{x=0} = 0\}$, and the $T^{\delta,n}$ are defined via $T^{\delta,n} = \phi_0 + \sum_i \psi_i \tilde{T}_i^n$, where $\phi_0|_{x=0} = 1$. The Dirichlet boundary condition (12c) is thus applied in the strong sense. A "no boundary condition" outflow boundary condition is applied [48].

Figure 1 shows a complete and functional implementation of this model in DOLFIN using a very high level representation. The discrete initialisation equation (13a) and timestep equation (13b) can be clearly identified. Note also that this model divides into an initialisation stage (consisting of one discrete equation), a timestep solve stage (consisting of one discrete equation) and a timestep variable cycle stage (in which variables corresponding to $T^{\delta,n}$ and $T^{\delta,n+1}$ are swapped). This example illustrates how, when writing a model using DOLFIN, a time discretisation must typically be constructed by hand. Moreover, the implementation fails to exploit the structure inherent in the time dependent problem.

Figure 2 shows a more practical implementation, using only native FEniCS functionality. Here, data which are known to be time independent are computed ahead of time, cached, and reused at every timestep. A linear solver caching option is also enabled. The timestep loop in this latter, optimised, implementation is much faster than the earlier very high level implementation[3]. However this illustrates a key issue with the manual construction of a time discretisation in this fashion — the optimisations required to ensure an efficient implementation of the time discretisation break the high level representation of the spatial discretisation. In this example discrete equations are replaced with lower lever linear algebra operations. For more complex examples, with multiple fields, equations, and with a mixture of time dependent and time independent data, the application of time discretisation specific optimisations leads to additional code complexity, increasing the gap between the mathematical notation and the source code implementation.

3.2. Adding a time discretisation abstraction to DOLFIN

In this section a new Python library is described. This new timestepping abstraction library builds upon the functionality of the DOLFIN library, and makes extensive use of the FEniCS system. In particular all symbolic manipulation, finite element assembly, and interfacing with linear solver libraries is handled by the FEniCS system. The timestepping abstraction library manages, analyses, and optimises individual model equations, and calls the DOLFIN library and other FEniCS tools as required in order to form a functioning timestepping model. The library source code is available as part of the dolfin-adjoint project[4].

[3] In this extremely small example usually negligible overheads have a significant cost, and hence the utility of a direct performance comparison here is limited. Explicit performance numbers are provided for more complex examples in section 4.

[4] `http://dolfin-adjoint.org/`

```
from dolfin import *

# The velocity, diffusivity, and timestep size
u = as_vector([Constant(1.0)])
kappa = Constant(0.02)
dt = Constant(1.0 / 64.0)

# Model mesh and function space
mesh = UnitIntervalMesh(32)
space = FunctionSpace(mesh, family = "Lagrange", degree = 1)
test = TestFunction(space)

# Two functions, representing T on two time levels
T_n, T_np1 = Function(space), Function(space)
# The initial condition
T_0 = interpolate(Expression("x[0] < DOLFIN_EPS ? 1.0 : 0.0"), space)
# Define and solve the initialisation equation
T_bc = DirichletBC(space, 1.0, "on_boundary && x[0] < DOLFIN_EPS")
solve(inner(test, T_n) * dx - inner(test, T_0) * dx == 0,
  T_n, T_bc, solver_parameters = {"linear_solver":"lu"})

# 32 timesteps
for i in range(32):
  # Define and solve the timestep solve equation
  T_h = 0.5 * (T_n + T_np1)
  test_supg = test \
    + 0.5 * CellSize(mesh) * dot(u, grad(test)) / sqrt(dot(u, u))
  solve((1.0 / dt) * inner(test_supg, T_np1 - T_n) * dx \
    + inner(test_supg, dot(u, grad(T_h))) * dx \
    - inner(test, kappa * dot(grad(T_h), FacetNormal(mesh))) * ds
    + inner(grad(test), kappa * grad(T_h)) * dx == 0,
    T_np1, bcs = T_bc, solver_parameters = {"linear_solver":"lu"})
  # Perform the timestep variable cycle
  T_np1, T_n = T_n, T_np1
```

Figure 1: A complete and functional finite element model for the advection-diffusion problem (12) written using DOLFIN, with a discretisation corresponding to (13). The model is integrated from an initial condition corresponding to $\tilde{T}_i^0 = 0$, with $u = 1$, $\kappa = 0.02$, and $\Delta x = 1/64$, for 0.5 units of time with an advective Courant number of 0.5. This implementation applies no time discretisation specific optimisations.

```
from dolfin import *

# The velocity, diffusivity, and timestep size
u = as_vector([Constant(1.0)])
kappa = Constant(0.02)
dt = Constant(1.0 / 64.0)

# Model mesh and function space
mesh = UnitIntervalMesh(32)
space = FunctionSpace(mesh, family = "Lagrange", degree = 1)
test, trial = TestFunction(space), TrialFunction(space)

# Two functions, representing T on two time levels
T_n, T_np1 = Function(space), Function(space)
# The initial condition
T_0 = interpolate(Expression("x[0] < DOLFIN_EPS ? 1.0 : 0.0"), space)
# Define and solve the initialisation equation
T_bc = DirichletBC(space, 1.0, "on_boundary && x[0] < DOLFIN_EPS")
solve(inner(test, T_n) * dx - inner(test, T_0) * dx == 0,
  T_n, T_bc, solver_parameters = {"linear_solver":"lu"})

# Pre-assemble timestep solve matrices
test_supg = test \
  + 0.5 * CellSize(mesh) * dot(u, grad(test)) / sqrt(dot(u, u))
M_dt = assemble((1.0 / dt) * inner(test_supg, trial) * dx)
N = assemble(inner(test_supg, dot(u, grad(trial))) * dx
  - inner(test, kappa * dot(grad(trial), FacetNormal(mesh))) * ds
  + inner(grad(test), kappa * grad(trial)) * dx)
M_lhs = M_dt + 0.5 * N; T_bc.apply(M_lhs)
M_rhs = M_dt - 0.5 * N

# Define an LU solver and enable a caching option
solver = LUSolver()
solver.parameters["reuse_factorization"] = True

# 32 timesteps
for i in range(32):
  # Solve the timestep solve equation
  b = M_rhs * T_n.vector()
  T_bc.apply(b)
  solver.solve(M_lhs, T_np1.vector(), b)
  # Perform the timestep variable cycle
  T_np1, T_n = T_n, T_np1
```

Figure 2: A complete and functional finite element model for the advection-diffusion problem (12) written using DOLFIN, with a discretisation corresponding to (13), with the same parameters as the model in figure 1. This implementation applies time discretisation specific optimisations, with static data cached, and a linear solver optimisation option enabled.

The library provides the functionality outlined in section 2.3. Specifically, time-independent data may be declared:

```
kappa = StaticConstant(0.02)
```

In this example the variable `kappa` is defined to be a time-independent constant value. Constant fields and boundary conditions may be similarly declared. This information can then be used to optimise the implementation of the timestepping model.

Second, time dependent fields may be defined:

```
levels = TimeLevels(levels = [n, n + 1], cycle_map = {n:n + 1})
T = TimeFunction(levels, space, name = "T")
```

In this example a time-dependent field `T` is defined, which exists on a given function space `space` and on the specified time levels. The variable `n` is used as an abstract handle to indicate an arbitrary timestep. In this example, the field `T` is defined to exist upon one past time level `n` and one future time level `n + 1`. The time-dependent field is also provided with information regarding the cycling of data at the end of each timestep – in this case data at time level `n + 1` replaces data at time level `n` during the timestep cycle. The specification of time dependent fields, and the specification of field data cycling at the end of each timestep, is sufficient to define the model time level sub-vectors x^n and $x^{n+1,+}$ (see section 2.1).

Given the definition of parameters and time dependent fields, time discretised equations may be defined. These are handled by registering model equations, for example for the initialisation:

```
system = TimeSystem()
T_bc = StaticDirichletBC(space, 1.0, "on_boundary && x[0] < DOLFIN_EPS")
system.add_solve(inner(test, T[0]) * dx == inner(test, T_ic) * dx,
  T[0], T_bc, solver_parameters = {"linear_solver":"lu"})
```

Here a `TimeSystem` object maintains the record of registered equations, and a single initialisation equation is registered, corresponding exactly to equation (13a). The time level data is referred to symbolically via simple indexing, so that `T[0]` corresponds to $T^{\delta,0}$. The relationship between the model description and the mathematical notation is therefore maintained. Similarly, the timestep solve equation is registered via:

```
T_h = 0.5 * (T[n] + T[n + 1])
test_supg = test \
  + 0.5 * CellSize(mesh) * dot(u, grad(test)) / sqrt(dot(u, u))
system.add_solve((1.0 / dt) * inner(test_supg, T[n + 1] - T[n]) * dx
    + inner(test_supg, dot(u, grad(T_h))) * dx
  == inner(test, kappa * dot(grad(T_h), FacetNormal(mesh))) * ds
    - inner(grad(test), kappa * grad(T_h)) * dx,
  T[n + 1], bcs = T_bc, solver_parameters = {"linear_solver":"lu"})
```

This corresponds exactly to equation (13b). Again, time level data is referred to symbolically via simple indexing, so that `T[n]` corresponds to $T^{\delta,n}$ and `T[n + 1]` corresponds to $T^{\delta,n+1,+}$. The equations thus registered enable the initialisation stage, timestep solve stage, and finalisation stage to be described, thereby providing a representation of the forward model structure (6).

Note that so far equations have only been described. No equation assembly or linear solves have been performed. In the transition from model description to model execution the equations are first analysed and time discretisation specific optimisations are applied. This is achieved via:

```
system = system.assemble()
```

This also initialises the model, solving all initialisation equations. The model is then executed via simple high level instructions, for example:

```
system.timestep(ns = 32)
system.finalise()
```

A complete model, incorporating the code examples discussed in this section, is shown in figure 3. The code is compact and high level. Model equations are described and solved via symbolic representations of the discrete equations, with all time discretisation specific optimisations applied automatically and internally by the library. Specific optimisation strategies which can be applied are detailed in the following section.

3.3. Time discretisation specific optimisations

The `assemble` call, highlighted in the previous section, marks the transition from the description of a timestepping model to the execution of the code itself. Time discretisation specific optimisations are applied at this step. The `TimeSystem` object, used to register the model equations, maintains an internal symbolic representation of the model equations. These may be easily manipulated to identify model dependencies (see Appendix A) and to extract individual equation terms. In particular, the following key optimisations may be applied.

First, if an equation is linear, and if the equation left-hand-side matrix is found to be static (time-independent), the matrix may be assembled and cached (pre-assembly). If the equation boundary conditions are static (and in particular are applied on the same part of the domain boundary for all time) then the matrix, with boundary conditions applied, may be cached. If the equation matrix is non-static or the equation is non-linear then matrix memory and sparsity patterns may be re-used. The matrix may be also be split into a static (and pre-assembled) part, and a non-static part. This latter optimisation results in a trade-off between the cost of adding matrices and the cost of finite element assembly, and hence is disabled by the timestepping library by default. Further more subtle optimisations are also possible – for example, the cost of the matrix addition in the latter optimisation may be reduced by ensuring that the two matrices have matching sparsity patterns.

Second, if a given right-hand-side term is static, then it may be assembled and cached. If a given right-hand-side term may be broken into a matrix multiply with a static matrix (static matrix multiplied by non-static vector), then the matrix may be assembled and cached.

Third, linear solver optimisation options may be enabled automatically. In particular, for equations with static cached matrices, matrix factorisation or preconditioner data may be reused on each timestep. In addition equations with identical left-hand-side matrices may be identified, and a single solver reused for all such equations.

Each of these optimisation strategies may be applied by hand using the native FEniCS system. However this process becomes increasingly cumbersome as more fields and equations are considered, which may contain equations with complex mixtures of static and non-static terms. If equation terms are modified, new parameters are introduced, or existing parameters are modified so that they switch from being static to time dependent, then in a hand optimised code one must propagate all modifications manually. Increasingly subtle optimisations, such as the matching of matrix sparsity patterns, may be applied quite easily by the optimisation routines, but lead to complexity and duplication when applied by hand. The optimisations are required in order to maximise efficiency, but if applied by hand are largely a time consuming and error prone distraction.

3.4. Deriving a discrete adjoint model

The high level representation of a timestepping model provides sufficient information for a discrete adjoint model to be derived automatically, via the methodology of Farrell et al. [14]. The high-level representation enables equation dependencies to be easily identified. Each registered equation is analysed in turn, differentiated with respect to its dependencies, and used to construct associated adjoint model equations. More advanced dependency analysis has not been implemented (for example, one need not adjoin fully diagnostic equations which do not influence a given functional) and this may be required in more advanced configurations. Nevertheless the high level abstraction means that, at least compared to lower level algorithmic differentiation approaches, the total number of dependency relationships which need be considered is considerably reduced. See Appendix A for a more detailed description of the steps involved in the adjoining process.

The timestepping library derives a discrete adjoint model via a single modification to the forward model code. Specifically, in the analysis stage, one requests the derivation of a discrete adjoint model:

```
system = system.assemble(adjoint = True, functional = T[N] * T[N] * dx)
```

In this example a discrete adjoint model is requested, using a functional defined to be equal to the square of the L^2 norm of the final T, $J = \int_0^1 T^{\delta,N} T^{\delta,N} dx$.

Crucially the discrete adjoint model thus derived has a high level symbolic representation, and moreover retains information regarding the temporal structure of the equations. Hence time discretisation specific optimisations, as described in section 3.3, can be applied to the adjoint model. If a forward model equation is modified, then a new consistent discrete adjoint model is derived, and the steps required to optimise the new adjoint code are handled automatically and internally. Since the optimisation strategies require detailed knowledge of the model equations, the manual application of these optimisations to the adjoint model would require intricate manual intervention in the adjoint model implementation.

A total derivative can now be computed upon completion of forward model timestepping via a single call:

```
dJ = system.compute_gradient(T_ic)
```

This method executes the adjoint model and computes the total derivative via equation (8) – this particular example computes the derivative of the functional with respect to the initial condition, T_0. A basic checkpointing algorithm may also be enabled:

```
system = system.assemble(adjoint = True, functional = T[N] * T[N] * dx, \
  disk_period = 20)
```

In this example a forward model checkpoint is saved to disk after every 20 timesteps. The `compute_gradient` method then performs any necessary re-running of the forward model between the checkpoints. The checkpointing scheme currently checkpoints all `TimeFunction` dependencies (which, as noted above, may be sub-optimal), and requires a means of restoring time dependent parameters to be provided by the user. Since the forward model can be advanced with ease more advanced checkpointing algorithms, such as the method described in Griewank and Walther [43], could in principle be used, although such algorithms are not currently implemented.

Figure 3 includes the derivation of a discrete adjoint model associated with the advection-diffusion model (13), with the calculation of a total derivative of a functional. The final T, and the computed derivative, are shown in figure 4. The discrete adjoint model is derived, and a total derivative calculation is performed, via exactly two modifications to the forward model code: the first to request an adjoint model and define a functional, and the second to compute a total derivative.

3.5. Verifying a discrete adjoint model

It is important that, once a discrete adjoint model is derived, the correctness of the adjoint model is asserted. In particular, if the forward model and functional are differentiable, then the total derivative of the functional with respect to model parameters should be computed exactly (except for precision related errors) by equation (8).

The consistency of a computed total derivative can be verified via a Taylor remainder convergence test. Specifically, this test exploits the fact that given a non-linear functional the remainder:

$$\left| J\left(x\left(m+\epsilon\delta m\right), m+\epsilon\delta m\right) - J\left(x\left(m\right), m\right) - \epsilon\frac{\mathrm{d}J}{\mathrm{d}m}\cdot\delta m\right| = \mathcal{O}\left(\epsilon^2\right), \tag{14}$$

converges asymptotically at second order in perturbation amplitude. The Taylor remainder (14) will converge asymptotically at second order only if the derivative $\mathrm{d}J/\mathrm{d}m$ is computed correctly[5]. See Navon et al. [49] for a related verification procedure.

The timestepping library can perform a Taylor remainder convergence test automatically via:

```
orders = system.taylor_test(T_ic, grad = dJ, ntest = 6, fact = 1.0e-4)
```

[5]As always with finite differencing, if the perturbation δm is too large then the asymptotic second order convergence rate will not be observed. If the perturbation δm is too small then the calculation of the Taylor remainder will be polluted by precision errors, and again the asymptotic second order convergence rate will not be observed.

```
from dolfin import *
from timestepping import *

# The velocity, diffusivity, and timestep size
u = as_vector([StaticConstant(1.0)])
kappa = StaticConstant(0.02)
dt = StaticConstant(1.0 / 64.0)

# Model mesh and function space
mesh = UnitIntervalMesh(32)
space = FunctionSpace(mesh, family = "Lagrange", degree = 1)
test = TestFunction(space)

# A time dependent function
levels = TimeLevels(levels = [n, n + 1], cycle_map = {n:n + 1})
T = TimeFunction(levels, space, name = "T")
# The initial condition
T_ic = StaticFunction(space, name = "T_ic")
T_ic.interpolate(Expression("x[0] < DOLFIN_EPS ? 1.0 : 0.0"))

# Register model equations
system = TimeSystem()
# Register the initialisation equation
T_bc = StaticDirichletBC(space, 1.0, "on_boundary && x[0] < DOLFIN_EPS")
system.add_solve(inner(test, T[0]) * dx == inner(test, T_ic) * dx,
  T[0], T_bc, solver_parameters = {"linear_solver":"lu"})
# Register the timestep solve equation
T_h = 0.5 * (T[n] + T[n + 1])
test_supg = test \
  + 0.5 * CellSize(mesh) * dot(u, grad(test)) / sqrt(dot(u, u))
system.add_solve((1.0 / dt) * inner(test_supg, T[n + 1] - T[n]) * dx
    + inner(test_supg, dot(u, grad(T_h))) * dx
  == inner(test, kappa * dot(grad(T_h), FacetNormal(mesh))) * ds
    - inner(grad(test), kappa * grad(T_h)) * dx,
  T[n + 1], bcs = T_bc, solver_parameters = {"linear_solver":"lu"})

# Perform analysis and optimisation, and derive a discrete adjoint model
system = system.assemble(adjoint = True, functional = T[N] * T[N] * dx)
# Perform 32 timesteps
system.timestep(ns = 32)
system.finalise()

# Execute the adjoint model and compute a total derivative
dJ = system.compute_gradient(T_ic, project = True)
```

Figure 3: A complete and functional finite element model for the advection-diffusion problem (12) written using the timestepping abstraction library, with a discretisation corresponding to (13), and with the same parameters as the model in figure 1. This implementation applies time discretisation specific optimisations automatically, and includes the derivation of a discrete adjoint model.

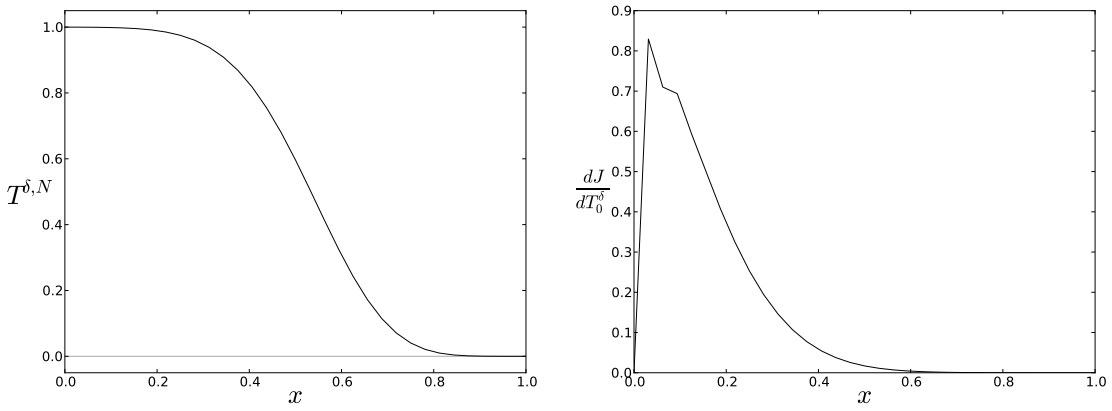

Figure 4: Left: Final solution, $T^{\delta,N}$, for the numerical model shown in figure 3. Right: The total derivative with respect to the initial condition, $\mathrm{d}J/\mathrm{d}T_0^\delta$, where $J(x,m) = \int_0^1 T^{\delta,N} T^{\delta,N} dx$ and T_0^δ is $P1$ on the model mesh. See Appendix B for the definition of $\mathrm{d}J/\mathrm{d}T_0^\delta$.

The `taylor_test` method perturbs a given parameter, reassembles any pre-assembled equations if required, executes the forward model using this perturbed parameter, and computes the Taylor remainder via (14). This is repeated for differing sized perturbations `ntest` times, with $\epsilon = 2^i$ for the ith perturbation. This was used to verify the model shown in figure 3. The results from a Taylor remainder convergence test are shown in figure 5.

4. Examples

In this section two more complex applications of the approach are provided, and the performance of the forward and adjoint models is assessed. Section 4.1 describes the application to the incompressible Navier-Stokes equations, and section 4.2 describes the application to the barotropic vorticity equation.

4.1. Incompressible Navier-Stokes

The incompressible Navier-Stokes equations for a fluid of constant density are:

$$\partial_t u + (u \cdot \nabla) u = -\nabla p + \nu \nabla^2 u, \tag{15a}$$

$$\nabla \cdot u = 0, \tag{15b}$$

where u is the velocity, p is the pressure (divided by the density), and ν is the kinematic viscosity. We consider the solution of these equations for the driven cavity configuration (see, for example, Burggraf [50]), in the unit square $\Omega = (0,1)^2$ and with boundary conditions:

$$\begin{aligned} u_x = u_F(x), u_y = 0 \quad & \text{on } y = 1, \\ u_x = 0, u_y = 0 \quad & \text{on } x = 0, x = 1, y = 0, \end{aligned} \tag{16}$$

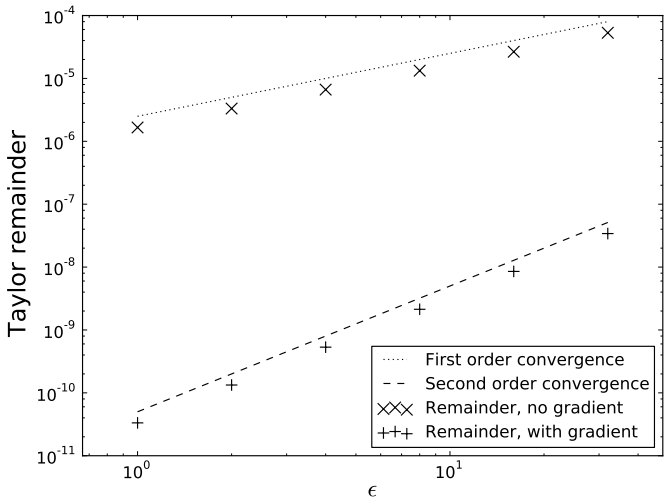

Figure 5: Taylor remainder convergence test for the model shown in figure 3. $\times$ symbols: The Taylor remainder $|J\left(x\left(m+\epsilon\delta m\right),m+\epsilon\delta m\right)-J\left(x\left(m\right),m\right)|$, which makes no use of gradient information, converges asymptotically at first order. $+$ symbols: The Taylor remainder (14), which makes use of the gradient computed using the adjoint model, converges asymptotically at second order as required.

where u_x and u_y are the x- and y-components of u respectively. The Navier-Stokes equations (15) are discretised in space using triangle elements with degree one discontinuous Lagrange basis functions for velocity and degree two continuous Lagrange basis functions for pressure ($P1_{DG}-P2$). This velocity pressure element pair is LBB stable [51]. The momentum advection term is integrated by parts and the cell interface term is treated using averaging of the interfacial fluxes, to ensure that the model is differentiable. Weak no-normal-flux boundary conditions are applied to the advection operator. The viscous term is treated using an interior penalty method [52, 53] with a penalty parameter value of $\eta=10$. Weak Dirichlet boundary conditions, corresponding to (16), are applied to the viscous term. The equations are discretised in time using the pressure projection method described in Ford et al. [54], consisting of a Crank-Nicolson discretisation (implicit midpoint rule) with the non-linear system approximately solved to second order accuracy in time via two Picard iterations. The pressure projection equations are solved with UMFPACK [55] via PETSc [56], and the velocity equations solved via PETSc with Bi-CGSTAB [57] preconditioned with incomplete LU factorisation.

The $P1_{DG}-P2$ velocity-pressure element pair has the important property that the discrete Laplacian matrix formed in the pressure projection step (the $-C^TM^{-1}C$ matrix in the notation of Ford et al. [54]) is identical to the discrete Laplacian formed by multiplying the Laplacian operator by a $P2$ test function and integrating by parts [58]. The former construction requires the separate assembly of divergence and mass matrices, and then the application of linear algebra operations. The latter construction requires the assembly of a single matrix directly from a single bilinear form. Hence the latter construction has a simple high level symbolic representation.

The model was implemented using the timestepping abstraction library. The re-

sulting code is extremely compact, comprising ≈ 260 lines of code if comments and debugging tests are neglected. The primary model development was performed over a few days by a single individual. The discrete adjoint model was derived, and a total derivative calculation performed, via exactly two modifications to the forward model source code, as described in section 3.4.

A quasi-uniform resolution unstructured triangle mesh covering the unit square was generated using Gmsh [59], with a requested element size of 0.01 units. The resulting mesh contained a total of 11671 vertices and 22944 triangle elements. This leads to 68832 degrees of freedom for each component of the velocity field and 46285 degrees of freedom for the pressure field. A solution to the driven-cavity problem was computed on this mesh with $\nu = 0.001$ and $u_F(x) = -1$, corresponding to a Reynolds number of 1000. A timestep size of $\Delta t = 0.01$, corresponding to an advective Courant number of approximately 1, was used. The model was integrated from rest for $T = 300$ units of time, after which the steady state measure defined by Botella and Peyret [60, equation (6)] had a value less than 4×10^{-12}. A $P2$ velocity divergence $D \in P$ was computed via:

$$\int_\Omega \phi D = -\int_\Omega \nabla\phi \cdot u \quad \forall \phi \in P, \tag{17}$$

where P is the $P2$ pressure space, and a $P2$ stream function $\psi \in P^0$ was computed via:

$$\int_\Omega \phi\psi = \int_\Omega \nabla\phi \cdot (\hat{z} \times u) \quad \forall \phi \in P^0, \tag{18}$$

where $P^0 = \{\phi \in P : \phi = 0 \text{ on } \partial\Omega\}$. The stream function was therefore computed using a strong homogeneous Dirichlet boundary condition. The final minimum value of u_x is -1.0018, indicating a small numerical undershoot. The final L^∞ norm of the velocity divergence is[6] 3.4×10^{-11}, indicating that discrete incompressibility is enforced – the error is attributed to accumulated floating point round-off errors. The final stream function is shown in figure 6. The final stream function has a maximum value of 0.11929, which differs from the value quoted in Botella and Peyret [60, table 6] by 0.30%. The final stream function has a minimum value of -0.0017191, which differs from the value quoted in Botella and Peyret [60, table 12] by 0.61%. Hence the model solution closely approximates the benchmark values.

The functional was defined to be the final kinetic energy:

$$J = \frac{1}{2}\int_\Omega u|_{t=T} \cdot u|_{t=T}\,. \tag{19}$$

The adjoint model employed disk checkpointing, and was used to compute the derivative with respect to the boundary condition for u_x at $y = 1$, shown in figure 7. For verification

[6]The L^∞ norm and ranges of $P2$ fields quoted here are computed from nodal value ranges. For a $P2$ field these can differ slightly from the field range, as the basis functions are not bounded between zero and unity.

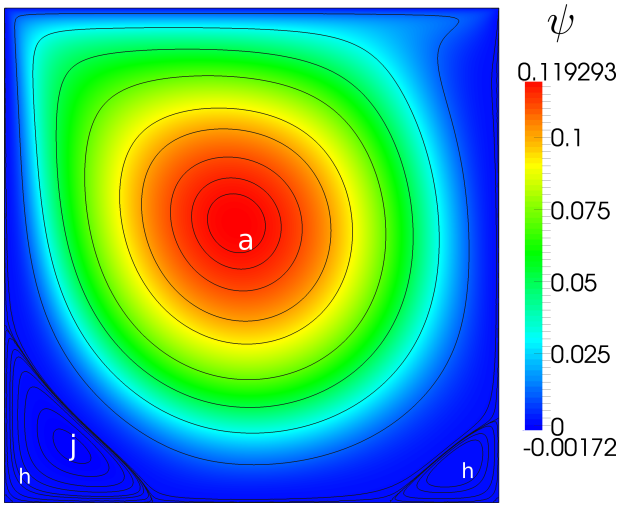

Figure 6: The final stream function for the solution of the incompressible Navier-Stokes equations for the driven cavity configuration at Reynolds number 1000. Contours are as per Botella and Peyret [60, figure 1].

purposes a similar derivative calculation was also performed for a configuration run over only 0.1 units of time, and this shorter calculation was successfully verified via a Taylor remainder convergence test.

The model performance was tested using a configuration run for 1 unit of time, corresponding to a total of 100 timesteps. The performance was tested on a machine with a 2.80 GHz Intel Core i5-2300 processor, with version 1.2.0 of DOLFIN. All tests used identical compiler optimisation flags, had output and diagnostics disabled and, when an adjoint model was derived, stored the entire model trajectory in memory. Practical time-dependent derivative calculations usually require the use of intermittent disk storage, which is not included in this performance analysis. Such checkpointing schemes require some amount of re-execution of the forward model during the adjoint calculation, and also incur additional input/output costs; the amount of re-execution depends on the number of checkpoints available [43]. For each measurement the model was run three times and the average taken. Table 1 shows the measured execution times. The execution times with pre-assembly disabled (but with linear solver optimisation options enabled) are also shown. Note that disabling pre-assembly also disables other optimisations, including some memory reuse and the reuse of sparsity patterns for time dependent matrices.

The use of pre-assembly is found to decrease the forward model run time (with the adjoint model disabled) by 41%. The use of pre-assembly is found to decrease the gradient calculation time (which includes the cost of running the adjoint model) by 36%.

The adjoint efficiency of this model is defined to be the cost of running the forward model and computing the total derivative (excluding the analysis), versus the cost of running the forward model with no adjoint model enabled (excluding the analysis). The measured adjoint efficiency is 2.17, versus an expected optimal efficiency for this model

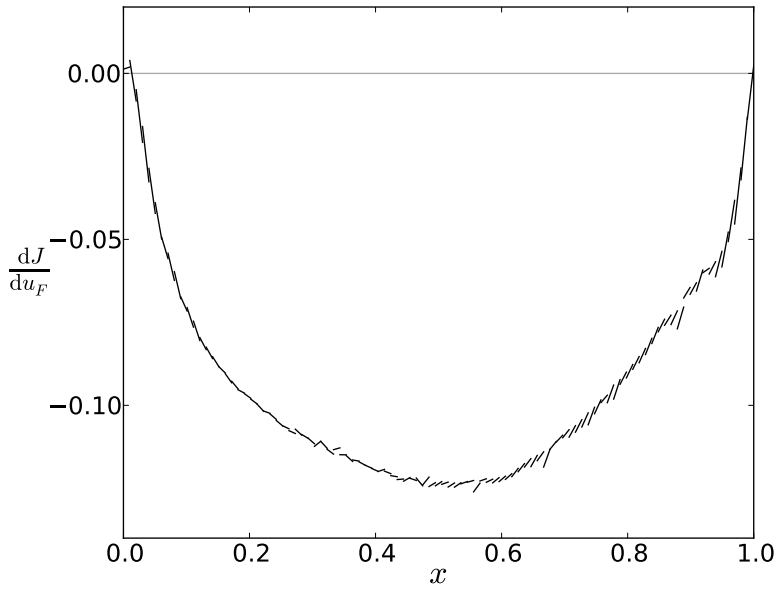

Figure 7: The derivative of the final kinetic energy with respect to the forcing boundary condition for u_x at $y = 1$, for the solution of the incompressible Navier-Stokes equations for the driven cavity configuration at Reynolds number 1000. Here the discrete boundary condition u_F is a $P1_{DG}$ field on the 1D boundary mesh. See Appendix B for the definition of dJ/du_F.

Model configuration	Model component	Time (s)	Normalised time
No adjoint	Analysis	3.83	0.0483
	Forward model	79.32	1.0000
	Total	85.70	1.0804
With adjoint	Analysis	11.18	0.1409
	Forward model	80.21	1.0112
	Gradient calculation	92.07	1.1607
	Total	186.04	2.3454
No pre-assembly, no adjoint	Analysis	0.73	0.0092
	Forward model	135.17	1.7041
	Total	138.45	1.7455
No pre-assembly, with adjoint	Analysis	3.31	0.0417
	Forward model	134.63	1.6973
	Gradient calculation	144.57	1.8226
	Total	285.07	3.5938

Table 1: Model execution times for the Navier-Stokes example. The forward model time includes the cost of field initialisation, timestepping, and any field finalisation. The gradient calculation time is the full cost of computing the total derivative of the functional. The analysis time is the cost of performing optimisations specific to the temporal discretisation and deriving the adjoint model (if these are enabled). The total time is the entire model execution time, excluding only the importing of libraries and some debugging code.

Model configuration	Model component	Time (s)
Timestepping library, upwinding of interfacial advective fluxes	Analysis	3.91
	Forward model	80.82
	Total	87.27
Fluidity	Total	271.82

Table 2: Model execution times comparing solvers for the Navier-Stokes equations. The efficiency of the model generated by the timestepping library compares well to an established CFD model, most likely due to the increased efficiency associated with the FEniCS system and due to the application of time discretisation optimisations.

of 2. While this is slightly sub-optimal, this is nevertheless sufficiently efficient to be of practical use. Note that, if form pre-assembly is disabled, then the adjoint model efficiency is measured to be 2.07. This latter, improved, adjoint efficiency is spurious – it is easier to achieve an improved relative adjoint efficiency if the absolute efficiency of the model is reduced.

For further comparison, the performance of the Navier-Stokes model is compared against an existing finite element code. Fluidity is a multi-purpose finite element CFD code with a diverse range of applications (see, for example, Piggott et al. [61] and Davies et al. [62]). Although a continuous adjoint model has previously been derived by hand [63, 64], this is no longer available. Note that Fluidity supports features which are not currently supported by FEniCS (and hence are not supported by the timestepping library), and Fluidity supports many more features than are considered in this example.

Fluidity was configured to simulate the driven cavity configuration, with the configuration matching, as closely as possible, the model developed here using the timestepping abstraction library. The Fluidity configuration uses upwinding of the interfacial advective fluxes, rather than averaging. The performance of Fluidity was assessed on the same test machine as used for the model written using the timestepping library, with the code compiled using aggressive compiler optimisations and with minimal model output. For each measurement Fluidity was run three times and the average taken. The resulting measured execution times are shown in table 2. The execution times of a model written using the timestepping abstraction library, using upwinding of the interfacial advective fluxes, are provided for reference. Fluidity is found to be 3.11 times slower than this model, indicating that the model generated by the timestepping library has a competitive efficiency.

4.2. Barotropic vorticity

The barotropic vorticity equation on a β-plane for a fluid of constant density and thickness and subject to wind forcing is:

$$\partial_t q + \nabla \cdot (uq) = \nu \nabla^2 (q - \beta y) + \hat{z} \times \nabla \cdot \left(\frac{F_W}{\rho_0 H} \right), \tag{20a}$$

$$q = \nabla^2 \psi + \beta y, \tag{20b}$$

where q is the potential vorticity, ψ is the stream function, $u = \hat{z} \times \nabla\psi$ is the incompressible velocity, F_W is the wind stress, ν is the (Laplacian momentum) viscosity coefficient, y is the meridional coordinate, ρ_0 is the fluid density, and H is the fluid depth. We consider the solution of these equations in a square $\Omega = (0, L)^2$ subject to free-slip boundary conditions:

$$q - \beta y = 0, \psi = 0 \quad \text{on } x = 0, x = L, y = 0, y = L, \tag{21}$$

and subject to a wind forcing of the form:

$$F_x = \begin{cases} +\tau_0 \cos\left(\pi \frac{y - y_m}{L - y_m}\right) & \text{if } y \geq y_M \\ -\tau_0 \cos\left(\pi \frac{y}{y_M}\right) & \text{if } y < y_M \end{cases}, \qquad F_y = 0, \tag{22}$$

where F_x and F_y are the x- and y-components of F, $y_m = (0.2x + 0.4)L$ defines the meridional coordinate of the eastward wind stress maximum, and x is the zonal coordinate. Model parameter values are $\beta = 2 \times 10^{-11}$ m^{-1} s^{-1}, $\nu = 150$ m^2 s^{-1}, $H = 1000$ m, $L = 2000$ km, and $\tau_0/\rho_0 = 4 \times 10^{-5}$ N kg^{-1} m. These parameters correspond to a Munk width of $\delta_M = (\nu/\beta)^{1/3} = 19.6$ km and a Reynolds number relative to the Sverdrup velocity scale of $\mathrm{Re} = \tau_0 / (\rho_0 H \beta \nu) = 13$.

The equations are discretised in space using triangle elements with degree two continuous Lagrange ($P2$) elements for the potential vorticity and stream function, with the boundary conditions (21) applied in the strong sense. The equations are discretised in time using third order Adams-Bashforth, with the model started via a second order explicit Runge-Kutta step and a second order Adams-Bashforth step, thereby ensuring global third order time accuracy. A timestep size of $\Delta t = 30$ minutes was used. A quasi-uniform resolution unstructured triangle mesh covering Ω was generated using Gmsh [59], with a requested element size of 16 km. The resulting mesh contained a total of 17967 vertices and 35436 triangle elements. This leads to 71369 degrees of freedom for the potential vorticity and stream function fields.

The model was implemented using the timestepping abstraction library. The resulting model code is again extremely compact, comprising ≈ 160 lines of code if comments and debugging tests are neglected. Figure 8 shows the model solution after a 100 year (100×365 days) spinup from rest. Once spun up, the model was run for 60 days. The functional was defined to be the integral of the final kinetic energy over the eastern region:

$$J = \frac{1}{2} \int_\Omega MH \, u|_{t=T} \cdot u|_{t=T} \, . \tag{23}$$

where $M = 1$ if $x > 0.9L$ and 0 otherwise. The adjoint model used disk checkpointing. The total derivative of this functional was computed with respect to F_x/ρ_0 over the 60 day integration, and is shown in figure 9. For verification purposes the forward model was also integrated from rest for 14 days, and a total derivative calculation was successfully verified for this configuration via a Taylor remainder convergence test.

The model performance was tested using a configuration run for 14 days from rest. Adjoint model tests computed the total derivative with respect to the initial potential vorticity, and used memory storage rather than disk checkpointing. Other details of

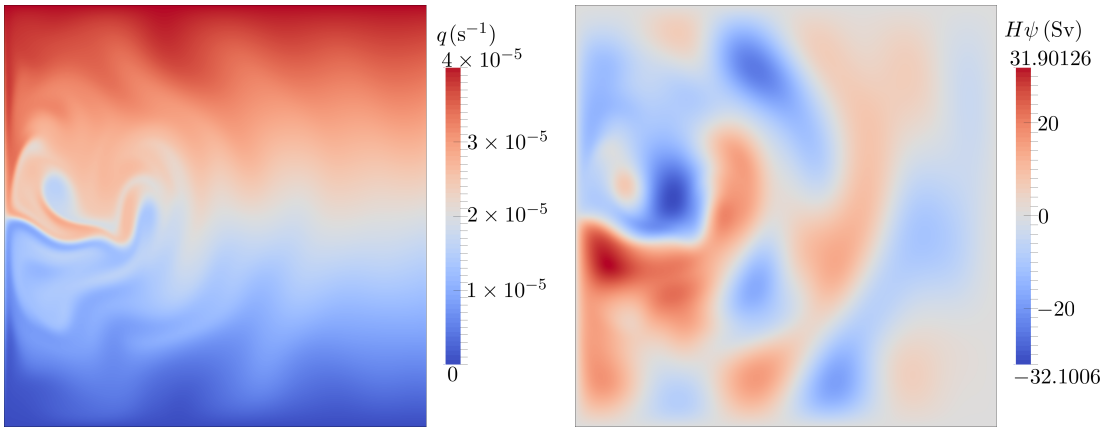

Figure 8: Solution of the barotropic vorticity equation after a spinup of 100 years from rest. Left: Potential vorticity q. Right: Transport stream function $H\psi$ (in Sverdrups).

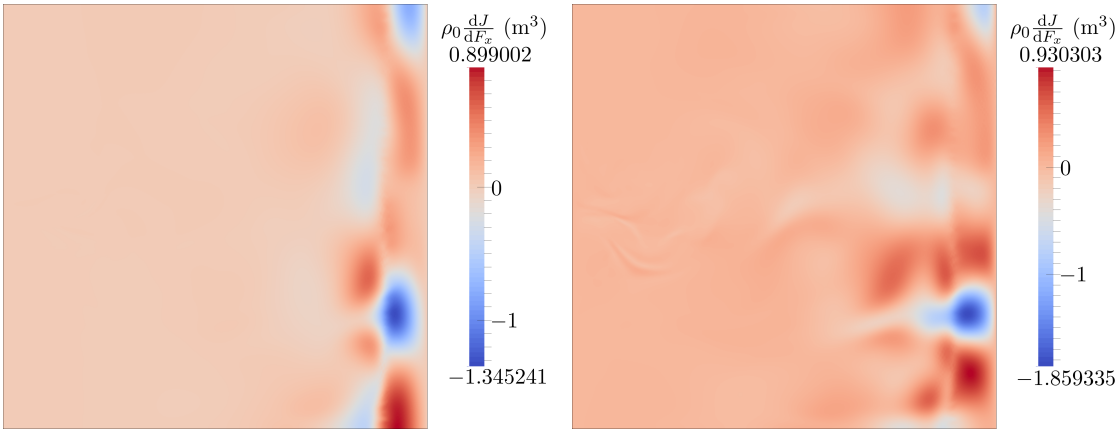

Figure 9: Left: Total derivative of the functional (23) evaluated 60 days after spinup (60 days after figure 8), with respect to F_x/ρ_0 over days 30–60 after spinup. Right: Total derivative of the functional (23) evaluated 60 days after spinup, with respect to F_x/ρ_0 over days 0–60 after spinup. As the length of the derivative calculation is increased the sensitivity propagates away from the eastern region over which the kinetic energy is integrated, indicating that flow near the eastern boundary becomes increasingly sensitive to changes in interior wind forcing as longer time intervals are considered.

Model configuration	Model component	Time (s)	Normalised time
No adjoint	Analysis	1.08	0.0325
	Forward model	33.05	1.0000
	Total	38.94	1.1782
With adjoint	Analysis	2.20	0.0664
	Forward model	34.42	1.0417
	Gradient calculation	42.78	1.2945
	Total	84.14	2.5462
No pre-assembly, no adjoint	Analysis	0.16	0.0049
	Forward model	172.07	5.2068
	Total	176.98	5.3555
No pre-assembly, with adjoint	Analysis	0.42	0.0128
	Forward model	173.52	5.2508
	Gradient calculation	174.41	5.2777
	Total	353.10	10.6848

Table 3: Model execution times for solvers for the barotropic vorticity equation written using the timestepping abstraction library.

the test configuration and test machine are identical to the benchmark described in the Navier-Stokes example in section 4.1. Table 3 shows the resulting measured execution times, including the execution times with pre-assembly disabled. The use of pre-assembly is found to decrease the forward model run time (with the adjoint model disabled) by 81%, and to decrease the gradient calculation time (which includes the cost of running the adjoint model) by 75%. Hence with pre-assembly optimisations the forward model is approximately five times faster, while the adjoint model is approximately four times faster. If both pre-assembly and solver optimisation options are disabled then both the forward and adjoint models are 30 times slower than the optimised versions. This model uses an explicit time discretisation with direct LU solvers [55], and hence time discretisation optimisations are essential.

The adjoint efficiency of this model is defined to be the cost of running the forward model and computing the total derivative (excluding the analysis) versus the cost of running the forward model with no adjoint model enabled (excluding the analysis). The measured adjoint efficiency is 2.34, versus an expected optimal efficiency for this model of 2. This is deemed to be sufficiently efficient for practical use. If pre-assembly and solver optimisation options are disabled then the adjoint efficiency is 2.00007, with (as previously noted) a much higher absolute cost.

5. Conclusions

The development of a numerical model typically consists of the design and selection of continuous model equations, the discretisation of these equations, and then the im-

plementation of the discretised system on a computer. Each of these steps is often the focus of distinct scientific disciplines: for example the physics of a system is embedded in the selection and approximation of continuous equations, and is certainly not related to the details of the source code implementation. Automated code generation enables the logical separation of the discretisation and implementation steps, thereby allowing the model developer to focus on the physics of a problem, or to focus on the selection and testing of discretisations and numerical algorithms.

In this article the principles of automated code generation have been extended to include the model time discretisation. Specifically, a new high level representation for transient finite element models has been presented. This approach builds upon the FEniCS system, and adds an additional library to the FEniCS suite of tools which enables a high level symbolic representation of the model time discretisation. The symbolic representation is used to construct an efficient implementation of the timestepping model, with time discretisation specific optimisations, such as pre-calculation and caching of static data, applied automatically and internally. Moreover the high level representation is amenable to symbolic manipulation. This is exploited to enable the automated derivation of an associated discrete adjoint model, which also benefits from these optimisations. Combined, the approach enables a high level representation of a complex time dependent problem, an efficient implementation of the model, and the automated derivation and efficient implementation of an associated discrete adjoint model.

The library was tested via the implementation of a solver for the incompressible Navier-Stokes equations. This complex model was developed extremely rapidly, and was found to be more than three times faster than the existing CFD code Fluidity. The adjoint model efficiency was measured to be 2.17. The adjoint model, and a total derivative calculation, were derived via only two minor modifications to the forward model code.

The library was further tested via the implementation of a solver for the barotropic vorticity equation. The model used an explicit time discretisation and direct solvers, and pre-assembly optimisations were found to be essential for this configuration. The forward and adjoint models were approximately five and four times faster respectively when pre-assembly optimisations were enabled. It is expected that pre-assembly optimisations are of particular importance for models with explicit time discretisations, and are relatively less important for implicit models with (typically) more expensive linear solves.

The generality of the timestepping abstraction library has been tested via the implementation and adjoining of many differing models with various time discretisations. Other model equations implemented include the non-linear Burgers' equation, advection-diffusion, the multi-layer quasi-geostrophic equations, the Boussinesq Navier-Stokes equations, Stokes' flow, and the Cahn-Hilliard equation. Time discretisations tested include explicit Runge-Kutta schemes, Adams-Bashforth schemes, backward Euler, Crank-Nicolson, leapfrog, and implicit discontinuous Galerkin.

This article has described the primary high-level functionality of the timestepping abstraction library. Lower-level interfaces are available which, for example, can be used to define custom linear or non-linear solvers. One may also define time integrated func-

tionals and define time dependent parameters. The timestepping abstraction library has also been integrated into the more general dolfin-adjoint library, demonstrating that the approach can be extended and used as part of a larger system.

The timestepping abstraction library acts as an intermediate layer, building on the FEniCS system. As a result the library inherits many of the desirable features of the FEniCS system, including efficient finite element assembly, access to efficient linear algebra libraries, and parallelisation methods. For example many models written using the timestepping abstraction library already support MPI parallelism. Should additional parallelisation methods be added to the FEniCS system, such as the exploitation of GPUs, then the timestepping abstraction library will be able to use these methods.

The implementation described in this article has been built upon the FEniCS system, but the general principles are not dependent upon these tools. If an alternative abstract representation and automated implementation of a spatial discretisation is available then a timestepping abstraction can be built onto this framework.

Thus far the only time discretisation specific optimisations that have been considered are optimisations which act to minimise computation time. Moreover at present these optimisations are applied in a "maximally aggressive" fashion. The ability for more detailed configuration is clearly desirable. This could potentially be aided by the heuristic selection of optimisation methods. For some problems it may also be necessary to consider alternative optimisations which reduce memory usage, rather than computation time.

In certain cases the symbolic analysis of the forward and adjoint solves is unable to identify common intermediates that can be re-used between expressions, which may cause the algorithmic complexity of the adjoint assembly to be unnecessarily high (see Griewank [65] for a related discussion). In future work the analysis engine described will be made more sophisticated to identify such cases, and to automatically rewrite them to reuse the available intermediates.

The ultimate objective of the high level representation for model timestepping presented here is to minimise the steps required to create an efficient transient finite element model with an efficient discrete adjoint model available. The particular aim of the authors is to enable the rapid implementation, testing, and optimisation of physical parameterisation schemes. This particular use case involves a modification of the continuous model equations themselves. Hence it is highly desirable to minimise the burden associated with the implementation of the (modified) discretised equations and, for the purpose of gradient-based optimisation, to minimise the burden associated with the development of associated discrete adjoint models. Previous approaches for automated adjoint derivation, such as algorithmic differentiation of low level source code, impose a high development and maintenance burden, which significantly impedes the use of adjoint models. It is hoped that the new approach will reduce the technical barrier to their adoption, facilitating their more widespread use.

Acknowledgements

JRM acknowledges useful discussions with C. J. Cotter, D. A. Ham and M. E. Rognes. Financial support was provided by the UK Natural Environment Research Council (NE/H020454/1), the UK Engineering and Physical Sciences Research Council (EP/I00405X/1, EP/K030930/1, and EP/G036136/1), an EPSRC Pathways to Impact award, and a Center of Excellence grant from the Research Council of Norway to the Center for Biomedical Computing at Simula Research Laboratory.

Appendices

Appendix A. Adjoining a transient finite element model

This appendix provides an example of the derivation of the discrete adjoint of a transient finite element model.

Appendix A.1. Burgers' equation discretisation

Consider the 1D non-linear Burgers' equation:

$$\partial_t u + u\partial_x u = \nu \partial_{xx} u \quad \text{on } x \in (0,1)\,, \tag{A.1a}$$

$$u = u_0 \quad \text{at } t = 0, \tag{A.1b}$$

$$u = 0 \quad \text{at } x = 0, 1, \tag{A.1c}$$

where u is the velocity and ν is the kinematic viscosity. Let the space $(0,1)$ be covered by a set of cells, and equip these cells with degree one Lagrange basis functions ϕ_i. Thus construct a $P1$ continuous Galerkin finite element discretisation with a Crank-Nicolson (implicit midpoint rule) time discretisation:

$$\int_0^1 \psi_i u^{\delta,0} dx = \int_0^1 \psi_i u_0 dx, \tag{A.2a}$$

$$\int_0^1 \psi_i \left[\frac{1}{\Delta t} \left(u^{\delta,n+1} - u^{\delta,n} \right) + u^{\delta,n+\frac{1}{2}} \partial_x u^{\delta,n+\frac{1}{2}} \right] dx = -\int_0^1 \nu \partial_x \psi_i \partial_x u^{\delta,n+\frac{1}{2}} dx. \tag{A.2b}$$

Here Δt is the timestep size, $u^{\delta,n}$ is the solution at time level n, and $u^{\delta,n+\frac{1}{2}} = \frac{1}{2}\left(u^{\delta,n} + u^{\delta,n+1}\right)$. The test functions are chosen so that $\psi_i \in \left\{ \phi_i : \phi_i|_{x=0,1} = 0 \right\}$, and the $u^{\delta,n}$ are defined via $u^{\delta,n} = \sum_i \psi_i \tilde{u}_i^n$. The Dirichlet boundary condition (A.1c) is thus applied in the strong sense. This simple discretisation is not stable, and is here used primarily as a simple non-linear example.

Consistent with the discussion in section 2.1, the discrete model equations can be re-written equivalently as:

$$\int_0^1 \psi_i u^{\delta,0} dx = \int_0^1 \psi_i u_0 dx, \tag{A.3a}$$

$$\int_0^1 \psi_i \left[\frac{1}{\Delta t} \left(u^{\delta,n+1,+} - u^{\delta,n} \right) + u^{\delta,n+\frac{1}{2},+} \partial_x u^{\delta,n+\frac{1}{2},+} \right] dx = - \int_0^1 \nu \partial_x \psi_i \partial_x u^{\delta,n+\frac{1}{2},+} dx, \tag{A.3b}$$

$$u^{\delta,n+1} = u^{\delta,n+1,+}, \tag{A.3c}$$

where $u^{\delta,n,+} = \sum_i \psi_i \tilde{u}_i^{n,+}$ and $u^{\delta,n+\frac{1}{2},+} = \frac{1}{2} \left(u^{\delta,n} + u^{\delta,n+1,+} \right)$. Hence (A.3a) is the initialisation equation, (A.3b) is the timestep solve equation, and (A.3c) is the timestep cycle equation. The three equations (A.3) corresponds to the three unique equations appearing in the forward system (6) for this model.

Appendix A.2. Burgers' equation discrete adjoint

Define a functional:

$$J \left(u^{\delta,N} \right) = \int_0^1 u^{\delta,N} u^{\delta,N} dx. \tag{A.4}$$

The resulting discrete adjoint model is:

$$\tilde{\lambda}_i^N = \int_0^1 2 \psi_i u^{\delta,N} dx, \tag{A.5a}$$

$$\int_0^1 \left[\psi_i \frac{1}{\Delta t} \lambda^{\delta,n+1,+} + \frac{1}{2} \left(\psi_i \partial_x u^{\delta,n+\frac{1}{2},+} + u^{\delta,n+\frac{1}{2},+} \partial_x \psi_i \right) \lambda^{\delta,n+1,+} + \frac{1}{2} \nu \partial_x \psi_i \partial_x \lambda^{\delta,n+1,+} \right] dx - \tilde{\lambda}_i^{n+1} = 0, \tag{A.5b}$$

$$\int_0^1 \left[-\psi_i \frac{1}{\Delta t} \lambda^{\delta,n+1,+} + \frac{1}{2} \left(\psi_i \partial_x u^{\delta,n+\frac{1}{2},+} + u^{\delta,n+\frac{1}{2},+} \partial_x \psi_i \right) \lambda^{\delta,n+1,+} + \frac{1}{2} \nu \partial_x \psi_i \partial_x \lambda^{\delta,n+1,+} \right] dx + \tilde{\lambda}_i^n = 0, \tag{A.5c}$$

where $\lambda^{\delta,n,+} = \sum_i \psi_i \tilde{\lambda}_i^{n,+}$. Equation (A.5b) is the adjoint equation associated with (A.3b), and (A.5c) is the adjoint equation associated with (A.3c). The adjoint model can be integrated backwards in time from the terminal condition (A.5a), via repeated solution of equation (A.5b) for each $\lambda^{\delta,n+1,+}$ and of equation (A.5c) for the $\tilde{\lambda}_i^n$. The three equations (A.5) correspond to the three unique equations appearing in the adjoint system (10) for this model.

Appendix A.3. Automated derivation of the Burgers' equation discrete adjoint

Given an appropriate symbolic representation of the discrete forward model (A.3), and appropriate symbolic manipulation tools, the discrete adjoint model (A.5) can be derived automatically. This is the methodology presented in Farrell et al. [14], using the tools of the FEniCS system, which we sketch here.

In the following `nu` and `dt` are DOLFIN `Constant` objects representing ν and Δt respectively, and `u_n` and `u_np1_p` are DOLFIN `Function` objects representing $u^{\delta,n}$ and

$u^{\delta,n+1,+}$ respectively. `test` represents a general test function, $\sum_i \psi_i \tilde{\psi}_i$ for arbitrary $\tilde{\psi}_i$. Then, using the DOLFIN library, the following yields a symbolic representation of the timestep solve equation (A.3b):

```
u_nph_p = 0.5 * (u_n + u_np1_p)
F = inner(test, (1.0 / dt) * (u_np1_p - u_n)) * dx \
      + inner(test, dot(as_vector([u_nph_p]), grad(u_nph_p))) * dx \
      + inner(grad(test), nu * grad(u_nph_p)) * dx
eq = F == 0
bc = DirichletBC(space, 0.0, "on_boundary")
```

Dependencies of equation (A.3b) can be identified via the UFL function:

```
deps = ufl.algorithms.extract_coefficients(F)
```

The dependencies can then be processed to identify the `Function` dependencies, `u_n` and `u_np1_p`. dolfin-adjoint uses this information to construct a "tape" of model equations and model equation dependencies [14]. The timestepping abstraction library uses this information in the analysis of registered equations (see section 3.2).

The left-hand-side matrix associated with equation (A.5b) can be constructed via:

```
mat_np1_p = assemble(adjoint(derivative(F, u_np1_p)))
hbc = homogenize(bc)
hbc.apply(mat_np1_p)
```

The first term in equation (A.5c) can be constructed via:

```
vec_n = assemble(action(adjoint(derivative(F, u_n)), lam_np1_p))
hbc.apply(vec_n)
```

where `lam_np1_p` represents $\lambda^{\delta,n+1,+}$. The key functions in each case are the `derivative` function, which differentiates a variational form with respect to a discrete field, and the `adjoint` function, which constructs an adjoint form. The `assemble` function assembles the symbolic representation of a form to yield a matrix or vector, generating lower level code to perform the assembly as required.

Appendix B. Riesz representer

Consider a discrete function $m^\delta = \sum_{i=1}^N \phi_i \tilde{m}_i \in V^\delta \subset L^2(\Omega)$, where the ϕ_i form a basis for V^δ. Consider also a functional $J(m^\delta)$ which is assumed to be Fréchet differentiable. Then the Gâteaux derivative of $J(m^\delta)$ in each direction ϕ_i is:

$$\mathrm{d}J(m^\delta; \phi_i) = \frac{\partial J}{\partial \tilde{m}_i}. \tag{B.1}$$

Given some perturbations $\delta\tilde{m}_i$ to the $\tilde{m}_i$, the linear perturbation to $J(m^\delta)$ is:

$$\delta J = \mathrm{d}J(m^\delta; \delta m^\delta) = \sum_{i=1}^N \frac{\partial J}{\partial \tilde{m}_i} \delta \tilde{m}_i, \tag{B.2}$$

where $\delta m^\delta = \sum_{i=1}^N \phi_i \delta \tilde{m}_i \in V^\delta$. Equivalently this may be written:

$$\delta J = \mathrm{d}J(m^\delta; \delta m^\delta) = \int_\Omega \frac{\mathrm{d}J}{\mathrm{d}m^\delta} \delta m^\delta, \tag{B.3}$$

where $\mathrm{d}J/\mathrm{d}m^\delta \in V^\delta$ is the Riesz representer for the Gâteaux derivative, defined via:

$$\mathrm{d}J(m^\delta; \phi_i) = \int_\Omega \frac{\mathrm{d}J}{\mathrm{d}m^\delta} \phi_i \quad \forall i \in \{1, \ldots, N\}\,. \tag{B.4}$$

Conceptually, the $\mathrm{d}J(m^\delta; \phi_i)$ define the linear sensitivity of the functional $J(m^\delta)$ with respect to perturbations to the degrees of freedom $\tilde{m}_i$ of m^δ. $\mathrm{d}J/\mathrm{d}m^\delta$ is a linear sensitivity density function which defines the linear sensitivity of the functional $J(m^\delta)$ with respect to perturbations to m^δ, via the L^2 inner product (B.3).

[1] J. Backus, Preliminary Report: Specifications for the IBM Mathematical FORmula TRANSlating System, FORTRAN, Technical Report, International Business Machines, 1954.

[2] C. B. Moler, Design of an interactive matrix calculator, in: Proceedings of the May 19-22, 1980, national computer conference, AFIPS '80, ACM, New York, NY, 1980, pp. 363–368. doi:10.1145/1500518.1500576.

[3] A. Logg, K.-A. Mardal, G. N. Wells (Eds.), Automated solution of differential equations by the finite element method: The FEniCS book, volume 84 of *Lecture Notes in Computational Science and Engineering*, Springer, 2012.

[4] M. S. Alnæs, UFL: a finite element form language, in: A. Logg, K. A. Mardal, G. N. Wells (Eds.), Automated Solution of Differential Equations by the Finite Element Method, Springer, 2011, pp. 299–334.

[5] M. S. Alnæs, A. Logg, K. B. Øelgaard, M. E. Rognes, G. N. Wells, Unified Form Language: A domain-specific language for weak formulations of partial differential equations, ACM Transactions on Mathematical Software 40 (2014) 9:1–9:37.

[6] R. C. Kirby, A. Logg, A compiler for variational forms, ACM Transactions on Mathematical. Software 32 (2006) 417–444.

[7] K. B. Ølgaard, G. N. Wells, Optimizations for quadrature representations of finite element tensors through automated code generation, ACM Transactions on Mathematical Software 37 (2010) 8:1–8:23.

[8] G. R. Markall, A. Slemmer, D. A. Ham, P. H. J. Kelly, C. D. Cantwell, S. J. Sherwin, Finite element assembly strategies on multi-core and many-core architectures, International Journal for Numerical Methods in Fluids 71 (2012) 80–97.

[9] P. A. Durbin, B. A. Pettersson Reif, Statistical theory and modeling for turbulent flows, Wiley, 2001.

[10] M. Mortensen, H. P. Langtangen, G. N. Wells, A FEniCS-based programming framework for modeling turbulent flow by the Reynolds-averaged Navier–Stokes equations, Advances in Water Resources 34 (2011) 1082–1101.

[11] G. R. Markall, D. A. Ham, P. H. J. Kelly, Towards generating optimised finite element solvers for GPUs from high-level specifications, Procedia Computer Science 1 (2010) 1815–1823.

[12] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, P. H. J. Kelly, Performance analysis of the OP2 framework on many-core architectures, SIGMETRICS Performance Evaluation Review 38 (2011) 9–15.

[13] M. G. Knepley, A. R. Terrel, Finite element integration on GPUs, ACM Transactions on Mathematical Software 39 (2013) 10:1–10:13.

[14] P. E. Farrell, D. A. Ham, S. W. Funke, M. E. Rognes, Automated derivation of the adjoint of high-level transient finite element programs, SIAM Journal on Scientific Computing 35 (2013) C369–C393.

[15] S. W. Funke, P. E. Farrell, A framework for automated PDE-constrained optimisation, Submitted to ACM Transactions on Mathematical Software, 2013. ArXiv:1302.3894 [cs.MS].

[16] P. E. Farrell, C. J. Cotter, S. W. Funke, A framework for the automation of generalised stability theory, SIAM Journal in Scientific Computing 36 (2014) C25–C48.

[17] M. B. Giles, E. Süli, Adjoint methods for pdes: a posteriori error analysis and postprocessing by duality, Acta Numerica (2002) 145–236.

[18] S. Prudhomme, J. T. Oden, On goal-oriented error estimation for elliptic problems: application to the control of pointwise errors, Computer Methods in Applied Mechanics and Engineering 176 (1999) 313–331.

[19] A. Ainsworth, J. T. Oden, A posteriori error estimation in finite element analysis, John Wiley & Sons, 2000.

[20] N. A. Pierce, M. B. Giles, Adjoint recovery of superconvergent functionals from PDE approximations, SIAM Review 42 (2000) 247–264.

[21] R. Becker, R. Rannacher, An optimal control approach to a posteriori error estimation in finite element methods, Acta Numerica 10 (2001) 1–102.

[22] J. Marotzke, R. Giering, K. Q. Zhang, D. Stammer, C. Hill, T. Lee, Construction of the adjoint MIT ocean general circulation model and application to Atlantic heat transport sensitivity, Journal of Geophysical Research: Oceans 104 (1999) 29529–29547.

[23] P. Heimbach, C. Wunsch, R. M. Ponte, G. Forget, C. Hill, J. Utke, Timescales and regions of the sensitivity of Atlantic meridional volume and heat transport: Toward observing system design, Deep Sea Research Part II: Topical Studies in Oceanography 58 (2011) 1858–1879.

[24] J. L. Lions, Optimal control of systems governed by partial differential equations, Springer-Verlag, 1971.

[25] M. D. Gunzburger, Perspectives in flow control and optimization, Advances in design and control, SIAM, 2003.

[26] B. Mohammadi, O. Pironneau, Shape optimization in fluid mechanics, Annual Review of Fluid Mechanics 36 (2004) 255–279.

[27] C. Wunsch, Discrete inverse and state estimation problems with geophysical fluid applications, Cambridge University Press, 2006.

[28] M. Hinze, R. Pinnau, M. Ulbrich, S. Ulbrich, Optimization with PDE constraints, volume 23 of *Mathematical Modelling: Theory and Applications*, Springer, 2009.

[29] B. F. Farrell, P. J. Ioannou, Generalized stability theory. Part I: Autonomous operators, Journal of the Atmospheric Sciences 53 (1996) 2025–2040.

[30] L. N. Trefethen, M. Embree, Spectra and Pseudospectra: The Behavior of Nonnormal Matrices and Operators, Princeton University Press, 2005.

[31] L. Zanna, P. Heimbach, A. M. Moore, E. Tziperman, Upper-ocean singular vectors of the North Atlantic climate with implications for linear predictability and variability, Quarterly Journal of the Royal Meteorological Society 138 (2012) 500–513.

[32] J. Martin, L. Wilcox, C. Burstedde, O. Ghattas, A stochastic Newton MCMC method for large-scale statistical inverse problems with application to seismic inversion, SIAM Journal on Scientific Computing 34 (2012) A1460–A1487.

[33] M. B. Giles, N. A. Pierce, An introduction to the adjoint approach to design, Flow, Turbulence and Combustion 65 (2000) 393–415.

[34] A. Griewank, A. Walther, Evaluating derivatives: principles and techniques of algorithmic differentiation, Frontiers in Applied Mathematics, SIAM, 2008.

[35] L. Hascoët, V. Pascual, TAPENADE 2.1 user's guide, Technical Report RT-0300, INRIA Sophia Antipolis, Sophia Antipolis, FR 06902, 2004. URL: `http://www.inria.fr/rrrt/rt-0300.html`.

[36] R. Giering, T. Kaminski, Applying TAF to generate efficient derivative code of Fortran 77-95 programs, Proceedings in Applied Mathematics and Mechanics 2 (2003) 54–57.

[37] J. Utke, U. Naumann, M. Fagan, N. Tallent, M. Strout, P. Heimbach, C. Hill, C. Wunsch, OpenAD/F: A modular, open-source tool for automatic differentiation of Fortran codes, ACM Transactions on Mathematical Software 34 (2008).

[38] C. A. Mader, J. R. R. A. Martins, A. C. Marta, Towards aircraft design using an automatic discrete adjoint solver, in: 18th AIAA Computational Fluid Dynamics Conference, American Institute of Aeronautics and Astronautics, 2007.

[39] F. Alauzet, O. Pironneau, Continuous and discrete adjoints to the Euler equations for fluids, International Journal for Numerical Methods in Fluids 70 (2012) 135–157.

[40] P. Heimbach, C. Hill, R. Giering, An efficient exact adjoint of the parallel MIT General Circulation Model, generated via automatic differentiation, Future Generation Computer Systems 21 (2005) 1356–1371.

[41] Wunsch, C. and Heimbach, P., Practical global oceanic state estimation, Physica D 230 (2007) 197–208.

[42] J. E. V. Peter, R. P. Dwight, Numerical sensitivity analysis for aerodynamic optimization: a survey of approaches, Computers & Fluids 39 (2010) 373–391.

[43] A. Griewank, A. Walther, Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation, ACM Transactions on Mathematical Software 26 (2000) 19–45.

[44] A. Kowarz, A. Walther, Optimal checkpointing for time-stepping procedures in ADOL-C, in: Proceedings of the 6th international conference on Computational Science - Volume Part IV, ICCS'06, Springer-Verlag, 2006, pp. 541–549.

[45] A. N. Brooks, T. J. R. Hughes, Streamline upwind / Petrov-Galerkin formulations for convection dominated flows with particular emphasis on the incompressible Navier-Stokes equations, Computer Methods in Applied Mechanics and Engineering 32 (1982) 199–259.

[46] O. C. Zienkiewicz, R. L. Taylor, P. Nithiarasu, The finite element method for fluid dynamics, 6 ed., Butterworth-Heinemann, 2005.

[47] J. Crank, P. Nicolson, A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type, Mathematical Proceedings of the Cambridge Philosophical Society 43 (1947) 50–67.

[48] D. F. Griffiths, The 'no boundary condition' outflow boundary condition, International Journal for Numerical Methods in Fluids 24 (1997) 393–411.

[49] I. M. Navon, X. Zou, J. Derber, J. Sela, Variational data assimilation with an adiabatic version of the NMC spectral model (????).

[50] O. R. Burggraf, Analytical and numerical studies of the structure of steady separated flows, Journal of Fluid Mechanics 24 (1966) 113–151.

[51] C. J. Cotter, D. A. Ham, C. C. Pain, S. Reich, LBB stability of a mixed Galerkin finite element pair for fluid flow simulations, Journal of Computational Physics 228 (2009) 336–348.

[52] D. N. Arnold, F. Brezzi, B. Cockburn, L. D. Marini, Unified analysis of discontinuous Galerkin methods for elliptic problems, SIAM Journal on Numerical Analysis 39 (2002) 1749–1779.

[53] K. B. Ølgaard, A. Logg, G. N. Wells, Automated code generation for discontinuous Galerkin methods, SIAM Journal on Scientific Computing 31 (2008) 849–864.

[54] R. Ford, C. C. Pain, M. D. Piggott, A. J. H. Goddard, C. R. E. de Oliveira, A. P. Umpleby, A nonhydrostatic finite-element model for three-dimensional stratified oceanic flows. Part I: Model formulation, Monthly Weather Review 132 (2004) 2816–2831.

[55] T. A. Davis, Algorithm 832: UMFPACK V4.3 – an unsymmetric-pattern multifrontal method, ACM Transactions on Mathematical Software 30 (2004) 196–199.

[56] S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, H. Zhang, PETSc Users Manual, Technical Report ANL-95/11, Argonne National Laboratory, 2011. Revision 3.2.

[57] H. A. van der Vorst, Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems, SIAM Journal on Scientific and Statistical Computing 13 (1992) 631–644.

[58] C. J. Cotter, D. A. Ham, C. C. Pain, A mixed discontinuous / continuous finite element pair for shallow-water ocean modelling, Ocean Modelling 26 (2009) 86–90.

[59] C. Geuzaine, J. F. 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 (2009) 1309–1331.

[60] O. Botella, R. Peyret, Benchmark spectral results on the lid-driven cavity flow, Computers & Fluids 27 (1998) 421–433.

[61] M. D. Piggott, G. J. Gorman, C. C. Pain, P. A. Allison, A. S. Candy, B. T. Martin, M. R. Wells, A new computational framework for multi-scale ocean modelling based on adapting unstructured meshes, International Journal for Numerical Methods in Fluids 56 (2008) 1003–1015.

[62] D. R. Davies, C. R. Wilson, S. C. Kramer, Fluidity: A fully unstructured anisotropic adaptive mesh computational modeling framework for geodynamics, Geochemistry Geophysics Geosystems 12 (2011).

[63] F. Fang, C. C. Pain, M. D. Piggott, G. J. Gorman, A. J. H. Goddard, An adaptive mesh adjoint data assimilation method applied to free surface flows, International Journal for Numerical Methods in Fluids 47 (2005) 995–1001.

[64] F. Fang, M. D. Piggott, C. C. Pain, G. J. Gorman, A. J. H. Goddard, An adaptive mesh adjoint data assimilation method, Ocean Modelling 15 (2006) 39–55.

[65] A. Griewank, On automatic differentiation, in: M. Iri, K. Tanabe (Eds.), Mathematical Programming: Recent Developments and Applications, Kluwer Academic Publishers, 1989, pp. 83–108.