A

A new approach for developing discrete adjoint models

Patrick E. Farrell, Simon W. Funke and David A. Ham, Imperial College London

Adjoint techniques provide powerful tools for computational scientists, with important applications across the whole of science and industry. However, these techniques are not widely used, mainly due to the difficulty of implementing adjoint models. The main tool used in the development of adjoint models has heretofore been algorithmic differentiation (AD) tools, which are based upon the abstraction that a model is a sequence of elemental instructions. In this paper, we investigate a new abstraction: that a model is a sequence of linear solves. Following this viewpoint, we describe the implementation of an open-source library (`libadjoint`) designed to assist in developing discrete adjoint models. The library implements the core algorithms for implementing such models, including assembly of the adjoint equations, managing the storage of forward and adjoint variables, and checkpointing algorithms to balance storage and recomputation costs. The library is applicable to any discretisation or equation, and is explicitly designed to be bolted-on to an existing forward model. This approach is compared to the alternative of writing the discrete adjoint model by hand, and is found to have several major advantages. We demonstrate the utility of the approach by adjoining models which are very difficult to differentiate with the AD approach.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: Algorithm Design and Analysis; G.1.8 [**Numerical Analysis**]: Partial Differential Equations; G.1.6 [**Numerical Analysis**]: Optimization; I.6.5 [**Simulation and Modelling**]: Model Development; J.2 [**Computer Applications**]: Physical Sciences and Engineering; J.6 [**Computer Applications**]: Computer-Aided Engineering; D.2 [**Software**]: Software Engineering

General Terms: Design, Algorithms

Additional Key Words and Phrases: Adjoints, automatic differentiation, algorithmic differentiation, data assimilation, gradient evaluation, inverse problems, optimisation

ACM Reference Format:
Farrell, P. E., Funke, S. W., Ham, D. A. 2011. A new approach for developing discrete adjoint models. ACM Trans. Math. Softw. V, N, Article A (January YYYY), 29 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

1.1. The need for derivatives

In the past 50 years, the computational solution of partial differential equations (PDEs) has established itself as an essential tool across the whole of the quantitative sciences. Not only can computer models simulate the results of experiments too expensive or too impractical to replicate in a laboratory, they are also crucial tools in the solution of important problems, such as optimisation problems and inverse problems

This work is supported by EPSRC grant EP/I00405X/1, NERC grant NE/I001360/1, the Grantham Institute for Climate Change and a Fujitsu CASE studentship. The authors would like to thank C. J. Cotter for assistance in understanding the shallow water model presented in section 8.2. The manuscript was greatly improved after useful discussions between P. E. Farrell and P. Heimbach.
Author's address: Farrell, Funke and Ham: Department of Earth Science and Engineering, Imperial College London. Funke and Ham: Grantham Institute for Climate Change, Imperial College London.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

subject to partial differential equation constraints. While these techniques are employed today in certain fields, the ability to solve such problems across a wider range of applications would have a massive impact across science, engineering and industry. In Griewank [2008], the author states that "we may juxtapose the mere *simulation* of a physical or social system by the repeated running of an appropriate computer model for various input data with its *optimization* by a systematic adjustment of certain decision variables and model parameters. The transition from the former computational paradigm to the latter may be viewed as a central characteristic of present-day scientific computing."

While it is possible to solve optimisation/inverse problems using only the computer model itself, the most powerful algorithms for solving these problems rely on the availability of derivative information. This leads to the need to differentiate computer models with respect to their inputs. If derivative information is not available from a given model, then gradient-based optimisation methods may not be used, and so the model is impractical for use in whole classes of important applications. For an excellent introduction to control and optimisation with computer models, see Gunzburger [2003]; for a rigorous mathematical treatment, see Hinze et al. [2009].

1.2. Algorithmic differentiation

The main tool used in the differentiation of models has heretofore been algorithmic differentiation tools, also known as automatic differentiation or AD tools [Rall and Corliss 1996; Griewank 2003; 2008]. The fundamental abstraction of algorithmic differentiation is that a model is a sequence of elemental instructions (such as additions, subtractions, $\cos$, $\sin$, etc.). Each elemental instruction may be differentiated individually, and these derivatives are then composed using the chain rule. A typical source-to-source AD tool takes in the source code for a given function f and returns source code that computes the action of the Jacobian ∇f on a given vector (the so-called *forward* mode), or the action of the transpose of the Jacobian on a given vector (the so-called *reverse* or *adjoint* mode). Griewank [2008] gives an authoritative survey of the field. The forward mode computes how a perturbation in one input affects all outputs, whereas the adjoint model computes how one output is affected by a perturbation in any of the inputs [Heimbach et al. 2010]. In the typical case of optimisation, there is only one functional (output) to be optimised, but many possible inputs; therefore, attention is now restricted to adjoint calculations.

Algorithmic differentiation tools have successfully differentiated complex models such as the MITgcm general circulation model [Heimbach et al. 2005], the FLUENT CFD code [Bischof et al. 2007], the CICE sea-ice model [Kim et al. 2006], and the WRF weather forecasting model [Xiao et al. 2008]. However, applying algorithmic differentiation to large complex models is a nontrivial endeavour; this is the reason why several authors prefer alternative terms to automatic differentiation. For example, Griewank [2003] states that "on the kind of real-life model indicated above, nothing can be achieved in an entirely automatic fashion. Therefore, the author much prefers the term algorithmic differentiation". Heimbach et al. [2010] states that "work is thus required initially to make the model amenable to efficient adjoint code generation for a given AD tool. This part of the adjoint code generation is not automatic (we sometimes refer to it as semi-automatic) and can be substantial for legacy code, in particular if the code is badly modularized and contains many irreducible control flows." (Of course, once that initial investment is made, the process becomes routine and automatic.) In discussing the possibility of automatically differentiating the NEMO ocean model, Vidard et al. [2008] states that "Even for this simplified configuration, however, substantial human intervention and additional work was required to obtain a useable product from the raw [AD]-generated code ... [The] memory management and CPU

performance of the raw code were rather poor. . . . From that experience it has been decided to go toward the hand-coding approach."

The efficient application of AD requires significant expertise and intimate familiarity with both the AD (algorithmic differentiation) tool and the model concerned. Firstly, the code must be modified to remove the use of language features that the tool does not support. Often, an AD tool only supports a subset of the programming language, requiring the model developer to rewrite portions of the code and forego the use of advanced language features. The model must be annotated with tool-specific directives to supply enough information to the AD tool so that it can generate an efficient adjoint; this requires intimate knowledge of how the AD tool works to do correctly. Models employing external numerical libraries or multiple programming languages need manual intervention to organise the differentiation. Furthermore, many AD tools are proprietary, sometimes limiting the availability of the adjoint models they create. These issues, while superable, motivate the search for new approaches: anything which reduces the time, money and expertise required to differentiate models will have a large impact in transferring these techniques to real-world applications.

1.3. An example: MITgcm and Fluidity

To consider the current domain of applicability of AD, we shall compare and contrast two large, complex models: MITgcm, a leading ocean/climate model [Marshall et al. 1997], and Fluidity, a computational/geophysical fluid dynamics framework currently under development [Piggott et al. 2008].

MITgcm is the flagship application for both the TAF [Giering and Kaminski 2003] and OpenAD [Utke et al. 2008] AD tools. Its adjoint has had a large scientific impact: it has been used in oceanographic and glaciological parameter estimation, sensitivity studies, and large scale data assimilation. Several features of MITgcm make it amenable for the use of AD. Firstly, it is written in Fortran 77, which is fairly straightforward to parse. The numerics are mostly explicit; the implicit step is self-adjoint [Heimbach et al. 2005], which means that no derivative code needs to be generated for the linear solve. The model has no hard dependencies on any external libraries; all of the core numerical calculations are performed within the model itself. The model has been co-developed with TAF, and so the model developers write the model within the language supported by the AD tool in mind; the MITgcm group is a centre of expertise in algorithmic differentiation. All of these factors combine to make the use of AD tractable.

By contrast, Fluidity enjoys none of these factors. The model is written in modern Fortran, and makes extensive use of advanced language features (dynamic memory allocation, pointers, derived data types, function overloading, the C binding support of Fortran 2003) that make life very difficult for the parser of an AD tool (indeed, for the parser of a compiler; the model developers have reported tens of compiler bugs to the Sun, Intel and gfortran developer teams). The model is semi-implicit, which necessitates the solution of non-self-adjoint linear systems; for this, it has a hard dependency on PETSc [Balay et al. 1997], for which no differentiated version is available. Indeed, in some configurations Fluidity depends on the matrix-free solvers available in PETSc [Davies et al. 2011]; an AD tool capable of differentiating through this process would have to be extremely sophisticated. The model has undergone years of development with no consideration for the constraints of an AD tool. Some parts of the model are written in C or C++. The model embeds the Python interpreter [van Rossum et al. 2008] for dynamic runtime functionality; for example, users can specify initial conditions, boundary conditions, source terms, and diagnostics entirely in Python. In fact, some of the model itself is implemented in Python. All of these factors combine to make applying currently available AD tools to the whole model intractable.

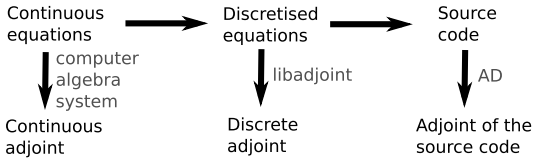


Fig. 1: The adjoint system can be derived after any stage of the forward system development. `libadjoint` facilitates the development of the discrete adjoint.

If the application of an AD tool is tractable, and gives acceptably efficient results, then that is almost certainly the best approach to differentiating the model, as it allows the adjoint to be effortlessly maintained. However, the question arises: what strategy should we adopt to differentiate Fluidity, and models like it, where the application of AD is not tractable?

1.4. An alternative: assembling the discrete adjoint equations

While AD has become the central tool in supplying derivatives of models, it is not the only possible approach. As observed in [Giering and Kaminski 1998]: "Suppose we want to simulate a dynamical system numerically. The development of a numerical simulation program is usually done in three steps. First, the analytical differential equations are formulated. Then a discretization scheme is chosen, and the discrete difference equations are constructed. The last step is to implement an algorithm that solves the discrete equations in a programming language. The construction of the adjoint model code may be implemented after any of these three steps", see figure 1.

The first choice of Giering's taxonomy consists of formulating the continuous adjoint equations and discretising them, possibly in a different manner to the discretisation of the forward system. This approach is often disfavoured, as the gradients produced are in general not consistent with the discrete forward system, complicating the optimisation loop [Gunzburger 2003, §4.1.2]. Furthermore, the discretisation of continuous adjoint equations can involve a significant development effort, comparable to developing the original forward model.

The third choice consists of applying algorithmic differentiation to the whole computational model, as discussed above.

The second choice of Giering's taxonomy is to assemble the discrete adjoint equations and use these to compute the necessary gradients [Gunzburger 2003]. Such an approach requires only the selective application of an AD tool to the nonlinear assembly operators [Giles et al. 2005], not the whole model. This is much more manageable than applying the AD tool to the whole model: they are typically small, self-contained, can be re-written to suit an AD tool if necessary, and the AD tool only needs to be re-applied if these sections are changed, which may be rare. Furthermore, some authors report that this discrete adjoint approach yields faster adjoint models than applying AD to the whole model: Coleman et al. [1998] reports that "performance gains of several orders of magnitude can sometimes be achieved by using AD in a selective manner", while Marta et al. [2007] reports that "while [the discrete adjoint approach] does not constitute a fully automatic way of obtaining sensitivities like pure AD, it is much faster in terms of execution time and drastically reduces the memory requirements". Müller and Cusdin [2005] show that unoptimised reverse-mode AD can be much slower than hand-written adjoints, but manual intervention to remove difficult constructs may dramatically improve the output of the AD tool. This intervention is practical when the code to be differentiated is small, but impractical if the entire model must be differentiated. However, despite the speed advantage of the discrete adjoint approach, the connecting and arrangement of these differentiated routines to compute the discrete adjoint

equations correctly is still very complex for large models, especially models which perform temporal integration, offer multiple discretisations, or optional packages. Expert knowledge is required to determine what operators should be differentiated when, and to determine when each forward variable is no longer necessary through the adjoint temporal calculation. Manually implementing essential checkpointing strategies such as the algorithms of Griewank and Walther [2000] and Wang et al. [2009] is invasive, difficult and error-prone; many hand-written adjoints do not implement a checkpointing strategy, as it is too complicated to do by hand. Heimbach et al. [2010] states: "The burden of developing "by hand" an adjoint model in general matches that of the forward model development. The substantial extra investment often prevents serious attempts at making available adjoint components of sophisticated models." As it stands, this approach is more complex and time-consuming than applying algorithmic differentiation to the whole model, which is why many model developers prefer the AD approach.

We propose a new abstraction for developing discrete adjoint models. Where algorithmic differentiation treats the model as *a sequence of elemental instructions*, we instead treat the model as *a sequence of linear solves*. As explained later, this abstraction is applicable to any computational model: it applies to any type of discretisation, linear or nonlinear models, and steady or time-dependent calculations (both implicit and explicit). We have written an open-source library, `libadjoint`, which implements this abstraction. The model developer *annotates* each linear solve the model performs to record details such as for what variable the equation is solving, what operators feature, and what dependencies these operators have. This builds a "tape" analogous to the concept of a tape in reverse-mode AD; but here, the units on the tape are much larger (the assembly and solution of one linear system could involve billions or trillions of elemental operations). Furthermore, the model developer supplies callbacks to `libadjoint`, so that the library may assemble or use the operators as necessary. With this information supplied, `libadjoint` can symbolically manipulate the forward annotation to automatically assemble the discrete adjoint equations, thus yielding the necessary model derivatives.

This approach has several advantages. It requires the differentiation only of much smaller pieces of code, and is agnostic about how that differentiation is achieved. One may use an AD tool; since it must be applied only to small pieces of code, the use of algorithmic differentiation is much more tractable and straightforward. An alternative approach to differentiation could be to use a finite difference approximation to the derivatives, like the independent set perturbation method of Fang et al. [2010], or to use the complex-step derivative approximation [Martins et al. 2003]. The process of annotating the forward model is very systematic; the library provides powerful debugging features that make the process straightforward. As `libadjoint` derives the adjoint equations, the expertise required to develop an adjoint model is greatly diminished. The model developer is freed from concerns such as managing the lifecycle of forward and adjoint variables: `libadjoint` can compute when each variable is no longer necessary and deallocate it appropriately. Furthermore, the model developer is freed from implementing checkpointing strategies; `libadjoint` has enough information about the forward model to re-play the forward solve when necessary, and thus the checkpointing strategy can be implemented entirely within the library itself.

Firstly, we discuss how to cast computational models into a standard canonical form.

2. REPRESENTING THE SYSTEM OF FORWARD DISCRETE EQUATIONS

As mentioned above, `libadjoint` views the computational model as a sequence of linear solves. That means that the model is cast in the form

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

where u is the vector of all unknowns in space and time, $b(u)$ is the source term for all time levels, and $A(u)$ the entire discretisation matrix. Before we derive the adjoint equations for this possibly unfamiliar system, we give several examples of how to represent common problems in this form.

2.1. Steady diffusion equation

Suppose the model approximately solves the steady-state diffusion equation

$$-\nabla^2 u = f, \tag{2}$$

subject to appropriate boundary conditions. Discretising with the Galerkin finite element method (or any other approach) results in a linear system

$$Du = f, \tag{3}$$

where D is the discretised diffusion operator, and u and f are vectors of coefficients for the solution and source term respectively. We may trivially identify $A \equiv D$ and $b \equiv f$ to cast equation (3) in the form of equation (1).

2.2. Time-dependent diffusion equation

Now suppose the model approximately solves the time-dependent diffusion equation

$$\frac{\partial u}{\partial t} - \nabla^2 u = f, \tag{4}$$

subject to some suitable boundary conditions and supplied initial condition $u(t=0) \equiv g$. Discretising with the Galerkin finite element method in space as above and the forward Euler method in time yields an iteration

$$\begin{aligned} u_0 &\leftarrow g \\ Mu_{n+1} &\leftarrow (M - \Delta t D)u_n + \Delta t f_n, \end{aligned} \tag{5}$$

where subscripts denote time levels, Δt is the timestep, M is the mass matrix and D is the discretised diffusion operator as before. At first glance, equation (5) does not appear to be in the same form as equation (1). However, if we rewrite the equations as

$$\begin{pmatrix} I & & & \\ (\Delta t D - M) & M & & \\ & (\Delta t D - M) & M & \\ & & \ddots & \ddots \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \end{pmatrix} = \begin{pmatrix} g \\ \Delta t f_0 \\ \Delta t f_1 \\ \vdots \end{pmatrix}, \tag{6}$$

then we recover the form of equation (1). In general, for time-dependent simulations, u is a block-structured vector containing all the values of the unknowns at all the time levels, A is a matrix with a lower-triangular block structure containing all of the operators featuring in the iteration, and b is a block-structured vector containing all of the right-hand side terms for all of the equations solved in the iteration.

The lower-triangular form of the matrix encodes the forward temporal flow of information in the equations: the solutions at later time levels depend on the solutions at earlier time levels, but not vice-versa. This is why the system usually solved by a computational model is herein referred to as the forward system. As we will see, the discrete adjoint equations involve taking the transpose of the forward system, and so the adjoint system is upper-triangular: therefore the temporal flow of information in the adjoint system is reversed, and the solutions at earlier time levels depend on the solutions at later time levels.

Note that writing a time-dependent model in this format does not imply that the whole of A is assembled at once. Typically, the model will assemble one row to compute

one variable, forget as much as possible, and timestep forward. However, writing it in this format is a useful abstraction, as it allows us to derive the discrete adjoint equations in a general manner. For time-dependent problems, `libadjoint` does not demand the assembly of the whole of A or A^* at any time.

2.3. Time-dependent Burgers' equation

Now suppose the model approximately solves the time-dependent viscous Burgers' equation

$$\frac{\partial u}{\partial t} + u \cdot \nabla u - \nabla^2 u = f, \tag{7}$$

subject to some suitable boundary conditions and supplied initial condition $u(t=0) = g$. Discretising with the Galerkin finite element method in space and the forward Euler method in time as above, and applying two Picard iterations per timestep to deal with the nonlinear advective term yields the iteration

$$\begin{aligned} u_0 &\leftarrow g \\ M u^0_{n+1} &\leftarrow (M - \Delta t V(u_n) - \Delta t D) u_n + \Delta t f_n \\ M u_{n+1} &\leftarrow (M - \Delta t V\left(\frac{1}{2}u_n + \frac{1}{2}u^0_{n+1}\right) - \Delta t D) u_n + \Delta t f_n, \end{aligned} \tag{8}$$

where $V(u)$ is the advection matrix assembled at a given velocity u, and the superscript $u^{(k)}_{n+1}$ denotes the k^{th} intermediate guess for u_{n+1}. For brevity, define

$$T(\cdot) \equiv \Delta t V(\cdot) + \Delta t D - M. \tag{9}$$

As before, equation (8) can be cast into the form of equation (1) by writing the iteration as

$$\begin{pmatrix} I & & & \\ T(u_0) & M & & \\ T\left(\frac{1}{2}u_0 + \frac{1}{2}u^0_1\right) & M & & \\ & \ddots & \ddots & \end{pmatrix} \begin{pmatrix} u_0 \\ u^0_1 \\ u_1 \\ \vdots \end{pmatrix} = \begin{pmatrix} g \\ \Delta t f_0 \\ \Delta t f_1 \\ \vdots \end{pmatrix}. \tag{10}$$

In general, any PDE solver may be cast in this form. This is useful because if we can derive the discrete adjoint equations for the abstract system of equation (1), then a general strategy for assembling discrete adjoint equations may be formulated.

2.4. Contrasting AD and the discrete adjoint equation approaches

To illustrate the contrast between the approach of applying AD to the whole model and using AD to assemble the discrete adjoint equations, we provide a cartoon of what pieces of the code must be differentiated for both approaches. Let J be the functional of interest, i.e. J maps the solution vector u to the value of interest $J(u) \in \mathbb{R}$.

For the approach of applying AD to the whole model, the boxed component of equation (11) shows the parts of the model which must be differentiated with an AD tool:

$$\boxed{\begin{aligned} u_0 &\leftarrow g \\ M u^0_{n+1} &\leftarrow (M - \Delta t V(u_n) - \Delta t D) u_n + \Delta t f_n \\ M u_{n+1} &\leftarrow (M - \Delta t V\left(\frac{1}{2}u_n + \frac{1}{2}u^0_{n+1}\right) - \Delta t D) u_n + \Delta t f_n \\ j &\leftarrow J(u). \end{aligned}} \tag{11}$$

That is, all of the numerical instructions must be differentiated.

By contrast, in the discrete adjoint approach, only the nonlinear assembly routines and the functional need be differentiated:

$$\begin{aligned} u_0 &\leftarrow g \\ Mu^0_{n+1} &\leftarrow (M - \Delta t\, V(u_n) - \Delta t D)u_n + \Delta t f_n \\ Mu_{n+1} &\leftarrow (M - \Delta t\, V\left(\tfrac{1}{2}u_n + \tfrac{1}{2}u^0_{n+1}\right) - \Delta t D)u_n + \Delta t f_n \\ j &\leftarrow J(u). \end{aligned} \tag{12}$$

If applying AD is difficult, this reduction is a significant advantage. Of course, the price paid is the complexity of assembling the discrete adjoint equations; this is the task `libadjoint` is designed to assist.

3. THE DISCRETE ADJOINT EQUATIONS

We now proceed to derive the general discrete adjoint equations associated with equation (1). Let $J(u)$ be a real-valued functional of interest for which we wish to derive the adjoint equations. Following Gunzburger [2003], we form the Lagrangian

$$L(u, \lambda) = J(u) - \langle \lambda, A(u)u - b(u) \rangle, \tag{13}$$

where λ is the adjoint variable corresponding to u, and $\langle \cdot \rangle$ is the usual inner product on $\mathbb{R}^n$ or $\mathbb{C}^n$. In order to derive the adjoint equation, we take the Gâteaux derivative of L with respect to u in an arbitrary direction $\tilde{u}$, and equate it with zero:

$$\frac{\partial L}{\partial u} = \lim_{\epsilon \to 0} \frac{L(u + \epsilon\tilde{u}, \lambda) - L(u, \lambda)}{\epsilon} = 0. \tag{14}$$

Applying Taylor's theorem to the J, A and b terms and taking the ϵ-limit [Gunzburger 2003, §2.2] yields:

$$\begin{aligned} \frac{\partial L}{\partial u} &= \left\langle \frac{\partial J}{\partial u}, \tilde{u} \right\rangle - \left\langle \lambda, \left(A(u) + \left[\frac{\partial A}{\partial u} u \right] - \frac{\partial b}{\partial u} \right) \tilde{u} \right\rangle \\ &= \left\langle \frac{\partial J}{\partial u} - \left(A(u) + \left[\frac{\partial A}{\partial u} u \right] - \frac{\partial b}{\partial u} \right)^* \lambda, \tilde{u} \right\rangle. \end{aligned} \tag{15}$$

Here, A^* denotes the Hermitian of A (transpose the matrix and take the complex conjugate). Since the perturbation $\tilde{u}$ is arbitrary, the left-hand entry of the inner product must be identically zero, and so we derive the general discrete adjoint equation for λ:

$$(A + G - R)^* \lambda = \frac{\partial J}{\partial u}. \tag{16}$$

where

$$G \equiv \frac{\partial A}{\partial u} u, \tag{17}$$

and

$$R \equiv \frac{\partial b}{\partial u}. \tag{18}$$

Let us examine each term in the adjoint equation in turn.

$\partial J / \partial u$ acts as the source term for the equation, indicating that the adjoint solution λ is specific to a given functional. Typically, J is a straightforward function of u, and so computing its derivative can usually be done by hand, or with a computer algebra

system such as SAGE [Stein and Joyner 2005; Stein et al. 2011]. In the examples presented later, the model user writes a small amount of Python code for the evaluation of J, and an object-overloading automatic differentiation tool [Lebigot 2011] automatically computes the necessary derivatives.

A^* is the Hermitian of the forward model's discretisation matrix. As explained in section 2.2, this implies that the temporal propagation of information in the adjoint equation is reversed: the adjoint solutions at earlier time levels depend on the solutions at later time levels, as A^* is upper-triangular. If we are to design a library which can automatically assemble equation (16), then this shows us that we must describe the block structure of A to that library, so that it can transpose that block structure.

R^* is the Hermitian of the Jacobian of the right-hand side. If the right-hand side b does not depend on u, then $R \equiv 0$. This term is sparse, and the sparsity is given by considering which blocks of b depend on which blocks of u. Again, because a right-hand side term appearing in the forward equation cannot depend on a variable computed in the future, the Jacobian must be lower-triangular, and so its Hermitian is upper-triangular, just as with the other terms on the left-hand side. In the context of designing `libadjoint`, this implies that the model developer must express the dependencies of each block of b on the blocks of u, so that the library knows to assemble the appropriate derivatives when assembling each adjoint equation.

G^* arises because of the nonlinear dependency of the operator A. If A has no dependency on u, then $G \equiv 0$. A is a rank-2 matrix, so differentiating it with respect to u yields a rank-3 tensor and the following contraction with u reduces the rank again to two. Written more precisely in tensor index notation,

$$G_{ik} \equiv \left(\frac{\partial A}{\partial u} \right)_{ijk} u_j, \tag{19}$$

i.e. the derivative $\partial A / \partial u$ is contracted with u over the middle index. For clarity, it may be helpful to reduce the rank of the objects considered by imagining we wish to compute G column-by-column. Equivalently, each column g_l of G is given by

$$g_l \equiv \frac{\partial A}{\partial u_l} u, \tag{20}$$

where the operation here is just normal matrix multiplication: as u_l is a scalar, taking the derivative with respect to u_l does not increase rank. In the context of designing `libadjoint`, this implies that the model developer must express the dependencies of each block of A, so that the library knows to assemble the appropriate derivatives when assembling each adjoint equation.

3.1. A worked example: the adjoint Burgers' equation

Consider again the discrete viscous Burgers system as shown in equation (10). For this example, the calculation of A^* is trivial, $R \equiv 0$ as b does not depend on u, and $\partial J / \partial u$ depends on the specific functional of interest. Therefore, we confine our attention to the calculation of G. (Once G is calculated, the calculation of G^* is trivial.)

From the column-wise definition of G (equation (20)), the block-structure of G can be derived:

$$G_{ij} \neq 0 \iff \text{row } i \text{ of } A \text{ depends on variable } j. \tag{21}$$

Therefore, the block-sparsity pattern of the G matrix for equation (10) is given by

$$\begin{pmatrix} 0 & & & & & \\ X & 0 & & & & \\ X & X & 0 & & & \\ & & X & 0 & & \\ & & X & X & 0 & \\ & & & \ddots & \ddots & \ddots \end{pmatrix}, \tag{22}$$

where X denotes a non-zero block. As an example, let us calculate the top-left nonzero block, G_{21}. G_{21} records the dependency of row 2 of A on variable 1, i.e. the dependency of the equation for u_1^0 on u_0. Differentiating every block of row 2 with respect to u_0 and contracting with the vector u of all time levels yields

$$\begin{aligned} G_{21} &= \begin{pmatrix} \Delta t \frac{\partial V(u_0)}{\partial u_0} & 0 & 0 & \dots \end{pmatrix} \cdot \begin{pmatrix} u_0 \\ u_1^0 \\ u_1 \\ \vdots \end{pmatrix} \\ &= \Delta t \frac{\partial V(u_0)}{\partial u_0} u_0. \end{aligned} \tag{23}$$

For this simple example, the derivation of G is not particularly complicated. But from the perspective of an adjoint model developer, it becomes very difficult as the model becomes more complex. As a simple example, suppose the model is extended so that the temporal discretisation is configurable at runtime. If backward Euler or Crank-Nicolson were used, the G-matrix would look very different; and so the adjoint assembly would have to take this into account, and thus this information must be made available somehow. Or suppose the model is extended so that it may solve the advection-diffusion equation for an arbitrary number of passive tracers. Each of these equations introduce rows in A which have nonlinear dependencies on the advecting velocity; and so each of these equations will contribute G^* terms which must be assembled when solving for the corresponding adjoint advecting velocity. Again, the adjoint assembly must have intimate knowledge of exactly the structure of the equations solved, and so this information must be recorded so that the adjoint assembly can execute. Even when this information is readily available, the derivation of the structure of the G-matrix is laborious and easy to get wrong.

The sensitive dependence of the discrete adjoint equations on precise details of the discretisation of the forward equations presents a software engineering problem for model developers. One strategy would be to duplicate the logic that decides upon the sequence and manner of the discretisation from the user input; however, that solution imposes a large maintenance burden on the developer, for now two separate implementations of that logic must be kept consistent. Instead, we suggest that a "tape" of model execution be recorded in a systematic and rigorous way, analogous to the tape concept of AD. With `libadjoint`, the model developer annotates each solve to record what it is solving for, what blocks feature in the equation, and upon what these blocks depend. With this information, the library can derive the structure of the G-matrix via symbolic manipulation, and correctly assemble the adjoint equation.

Having derived the discrete adjoint equations for the standard canonical form, we now examine how this is achieved in code.

4. A CODE EXAMPLE

Suppose we have a pre-existing code that solves the Burgers' equation example (equation (10)). Of course, it will not assemble the whole of A at once, pass it over to the linear solver, and finish. Instead, the model will solve one equation, forget any variables no longer necessary for future calculations, solve the next equation, and so on. Suppose we wish to annotate the model to also solve the corresponding discrete adjoint equations. Ideally, any changes to be made should be minimally invasive, easy to code, and robust to changes in the forward discretisation. `libadjoint` has been designed with such considerations in mind.

As discussed in section 3, `libadjoint` must offer calls to enable the model developer:

— to express the block structure of an equation solved in the forward model,
— to express any nonlinear dependencies of each block,
— to record the value of variables.

The model developer must also supply callbacks:

— to assemble each block on the diagonal of A (and its Hermitian),
— to compute the action of each off-diagonal block of A on a given vector (and its Hermitian),
— to compute the functional derivative source term of the adjoint equation,
— to compute the Hermitian action of the Jacobian of nonlinear terms on a given vector.

Once these requirements are satisfied, then the library may assemble the adjoint system and return it to the model developer to be solved. The non-Hermitian version of first two callbacks are typically straightforward, as this capability must already exist in the forward model, and it is merely a matter of modularising and interfacing it. The creating of the latter two and the Hermitian version of the first two callbacks may be greatly facilitated by applying algorithmic differentiation (to be discussed in section 4.2).

We now sketch the annotation of the Burgers' equation model to also solve the discrete adjoint problem. In an actual complex model, the annotation would be spread around in the code, so that each step or solve would record itself as it happens. For clarity, error checking and memory deallocation have been left out from this sketch. While the example is presented in Fortran, the library is accessible from C and Fortran. Object-oriented C++ and Python interfaces are planned for a future release.

```
! u_0, u_n^0, u_n, u_{n-1}
type(adj_variable) :: u0, un_guess, un_final, u_prev
! blocks that appear in A
type(adj_block) :: I, M, T
! blocks that have dependencies also have an associated adj_nonlinear_block
type(adj_nonlinear_block) :: V

! the fundamental object of libadjoint that records the model execution
type(adj_adjointer) :: adjointer
type(adj_equation) :: equation

integer :: ierr

! Initialisation
ierr = adj_create_adjointer(adjointer)

! Annotate the initial condition
```

```
ierr = adj_create_block(name="IdentityOperator", block=I)
ierr = adj_create_variable(name="Velocity", timestep=0,
                           iteration=1, variable=u0)

! Compare the following line to row 1 of equation 10
ierr = adj_create_equation(variable=u0, blocks=(/I/),
                           targets=(/u0/), equation=equation)
ierr = adj_register_equation(adjointer, equation)

! Set up the mass matrix block
ierr = adj_create_block(name="MassMatrix", block=M)

! Set u_prev to the initial condition.
u_prev = u0

! Now enter the time loop
do timestep=1,no_timesteps
  ! Record the solve for u_n^0
  ierr = adj_create_variable(name="Velocity", timestep=timestep,
                             iteration=0, variable=un_guess)

  ! Annotate the nonlinear dependency of the advection term on u_prev
  ierr = adj_create_nonlinear_block(name="AdvectionOperator",
                                    depends=(/u_prev/), nblock=V)
  ierr = adj_create_block(name="TimesteppingOperator", nblock=V, block=T)

  ! Compare the following line to row 2 of equation 10
  ierr = adj_create_equation(variable=un_guess, blocks=(/T, M/),
                             targets=(/u_prev, un_guess/), equation=equation)
  ierr = adj_register_equation(adjointer, equation)

  ! Now record the solve for u_n
  ierr = adj_create_variable(name="Velocity", timestep=timestep,
                             iteration=1, variable=un_final)

  ! Annotate the nonlinear dependency of the advection term:
  ! now it depends on both un_guess and on u_prev
  ierr = adj_create_nonlinear_block(name="AdvectionOperator",
                                    depends=(/u_prev, un_guess/), nblock=V)
  ierr = adj_create_block(name="TimesteppingOperator", nblock=V, block=T)

  ! Compare the following line to row 3 of equation 10
  ierr = adj_create_equation(variable=un_final, blocks=(/T, M/),
                             targets=(/u_prev, un_final/), equation=equation)
  ierr = adj_register_equation(adjointer, equation)

  ! Next u_prev is current u_final
  u_prev = u_final
end do
```

As can be seen, the annotation is fairly straight-forward and self-explanatory. The structure of the annotation simply mimics the structure of A.

With the information presented above, libadjoint can derive the structure of the discrete adjoint system; it cannot, however, actually assemble the equations. As it stands, the operators are mere abstract handles, known only by name. In order for assembly to be possible, further information must be available: the operators must be backed up with callbacks, so that libadjoint can apply them when necessary.

```
ierr = adj_register_operator_callback(adjointer, ADJ_BLOCK_ASSEMBLY_CB,
                                      "IdentityOperator",
                                      c_funloc(identity_assembly))
ierr = adj_register_operator_callback(adjointer, ADJ_BLOCK_ASSEMBLY_CB,
                                      "MassMatrix",
                                      c_funloc(mass_assembly))
ierr = adj_register_operator_callback(adjointer, ADJ_BLOCK_ACTION_CB,
                                      "TimesteppingOperator",
                                      c_funloc(timestepping_action))
ierr = adj_register_functional_derivative_callback(adjointer,
                                      "MyFunctional",
                                      c_funloc(functional_derivative))
```

For more details of the callback interface, see the libadjoint manual [Farrell and Funke 2011]. With this information, the library can now assemble the adjoint equations by calling adj_get_adjoint_equation:

```
type(adj_matrix) :: adjoint_lhs
type(adj_vector) :: adjoint_rhs, adjoint_soln
type(adj_variable) :: adjoint_var
integer :: nequations
integer :: equation
integer :: ierr

ierr = adj_equation_count(adjointer, nequations)
do equation=nequations-1,0,-1
  ! Have libadjoint assemble the adjoint equation:
  ierr = adj_get_adjoint_equation(adjointer, equation, "MyFunctional",
                                  lhs=adjoint_lhs, rhs=adjoint_rhs,
                                  variable=adjoint_var)
  ! Now solve it:
  adjoint_soln = solve(adjoint_lhs, adjoint_rhs)
  ! And supply the solution to libadjoint again:
  ierr = adj_record_variable(adjointer, adjoint_var, adjoint_soln)
end do
```

This adjoint main loop is intended to give the idea; a more realistic example is given in the manual [Farrell and Funke 2011, §6.2].

4.1. Vectors and matrices

libadjoint needs to manipulate vectors and matrices: the entire purpose of the output function adj_get_adjoint_equation is to assemble a left-hand-side matrix and a right-hand-side vector. Therefore, libadjoint needs to have classes representing the concepts of vectors and matrices.

However, one of the design goals of libadjoint is to be applicable to many different models, using very different data structures. How, then, should the libadjoint vector and matrix classes work?

One approach is for `libadjoint` to define its own vector and matrix classes, to which the model developer must convert the model data structures. However, writing yet another linear algebra API is highly undesirable. A second approach would be to standardise on some common linear algebra API such as PETSc [Balay et al. 1997; Balay et al. 2010] or Trilinos [Heroux et al. 2003]. However, this would introduce a large hard dependency on such a library to any project which uses `libadjoint`; the aim for `libadjoint` is to be small, lightweight, portable, and easy to install. Furthermore, it would entail having the same data around in memory in two different formats (the format supported by `libadjoint`, and the model's own), which would be very inefficient.

`libadjoint` settles on a third approach, inspired by the callback structure of the Zoltan graph partitioning library [Devine et al. 2002]. The `libadjoint` vector and matrix classes are merely thin wrappers around the user's own data structures; at their core, they are nothing but a pointer. While this approach has the great advantage that it can be used with any and all pre-existing data structures, it means that `libadjoint` has a problem: how can it manipulate these objects, when it knows nothing about them? The answer is for the model developer to supply *data callbacks* to `libadjoint`, to give `libadjoint` the power to manipulate these objects as necessary.

This approach entails some extra work on the part of the model developer; however, this extra work is once-off for each sort of data structure employed, and can easily be shared between model developers by incorporating the various sets of data callbacks in `libadjoint` itself. Already, `libadjoint` includes data callbacks for interfacing with the PETSc `Vec` and `Mat` types, and more will be incorporated into the library as they are developed.

4.2. Interacting with algorithmic differentiation

`libadjoint` requires certain derivatives and Hermitian actions in order to assemble the adjoint equation, for which algorithmic differentiation tools can be used to generate the corresponding routines A detailed introduction to the application of AD tools can be found in Rall and Corliss [1996].

The derivative of the functional, $\partial J/\partial u$, is always necessary for assembling the adjoint equations, for it acts as the source term. Code for computing the derivative may be produced by applying an algorithmic differentiation tool in reverse mode to code which computes $J(u)$.

The derivative of the source term for the forward equation, $\partial b/\partial u$, is necessary if the right-hand side of the forward equation itself depends on u. Again, the application of an algorithmic differentiation tool will produce the necessary subroutines.

If the simulation is nonlinear, then the nonlinear assembly routines must be differentiated with respect to their arguments. In this case, the application of algorithmic differentiation is slightly more complex, and so it is explained here. If $V(u_1, u_2, \dots)$ is the block of A with a nonlinear dependency, then the form of the derivatives necessary for assembling the adjoint equation is

$$o = \left[\left(\frac{\partial V}{\partial u_k} \right) c \right]^* \lambda, \tag{24}$$

where u_k is the variable with respect to which V must be differentiated, c is a given contraction vector, and λ is a given adjoint variable. (Compare this to equation (23): there, $c \equiv u_0$, and λ is the adjoint variable corresponding to u_1^0.) In order to produce the code to compute this, write a function f which assembles the nonlinear operator and applies it to a contraction vector:

$$f(u_1, u_2, \dots, c) = V(u_1, \dots)c, \tag{25}$$

and apply an AD tool in reverse mode to differentiate it with respect to u_k. This yields a new function

$$\bar{u}_k = g(u_1, u_2, \ldots, c, \bar{f}), \tag{26}$$

which computes the derivative of f with respect to u_k in direction $\bar{f}$, i.e.

$$g = \left(\frac{\partial f}{\partial u_k}\right)^* \bar{f} = \left[\left(\frac{\partial V}{\partial u_k}\right) c\right]^* \bar{f}. \tag{27}$$

Hence, o can be obtained by evaluating $\bar{u}_k$ at λ:

$$o = g(u_1, u_2, \ldots, c, \lambda). \tag{28}$$

In a similar way, the Hermitian action of blocks can be obtained. For that, write a function f that computes

$$f(u_1, u_2, ..., c) = V(u_1, \ldots)c. \tag{29}$$

The Jacobian of this expression with respect to c is V; therefore, differentiating it in reverse mode with respect to c yields a new function

$$\bar{c} = g(u_1, u_2, \ldots, c, \bar{f}), \tag{30}$$

which computes the action of the Hermitian of V.

One of the advantages of the `libadjoint` approach is that it minimises the dependence on any particular AD tool. Applying an AD tool to a whole codebase often involves rewriting some parts of it to remove language features unsupported by the tool, and the manual insertion of tool-specific directives to guide the differentiation. When the developers of a model make a significant investment of time and money in the use of one AD tool, this creates a hard dependency on that tool: continual re-application of the AD tool is necessary as the model changes, and it would be difficult to swap that tool for an alternative. By contrast, with the `libadjoint` approach, only small parts of the model must be differentiated, and so the investment in any particular tool is light.

5. DERIVING THE DISCRETE ADJOINT EQUATIONS

In this section, we describe the algorithm at the core of `libadjoint`, the derivation of the discrete adjoint equations.

Just as the forward matrix A is assembled row-by-row, the adjoint matrix $(A + G - R)^*$ is also assembled row-by-row. The following algorithms assemble the left-hand side and right-hand side of the adjoint system for λ_k, the adjoint variable at the k-th equation.

ALGORITHM 1: Assembly of the left-hand side of the linear system for λ_k

Data: the `adjointer` object; the index k of the adjoint equation to assemble
Result: the left-hand side matrix of the adjoint system `lhs`
$G^*_{kk} \leftarrow 0$
$R^*_{kk} \leftarrow 0$
if *row k of A has a dependency on u_k* **then**
  for *all blocks V_{kl} that depend on u_k* **do**
    $G^*_{kk} \leftarrow G^*_{kk} + \left(\frac{\partial V_{kl}}{\partial u_k} u_l\right)^*$
  end
end
if *b_k has a dependency on u_k* **then**
  $R^*_{kk} \leftarrow \left(\frac{\partial b_k}{\partial u_k}\right)^*$
end
`lhs` $\leftarrow A^*_{kk} + G^*_{kk} - R^*_{kk}$

The first contribution to the left-hand side, A^*_{kk}, is always present, and is the Hermitian of the operator on the left-hand side of the equation that was solved for u_k. The second contribution, G^*_{kk}, is only present if row k of the matrix A has a dependency on the variable u_k, i.e. if the equation for u_k depends on itself. Similarly, the R^*_{kk} contribution is only present of the right-hand side of the equation for u_k depends on u_k.

ALGORITHM 2: Assembly of the right-hand side of the linear system for λ_k

Data: the `adjointer` object; the index k of the adjoint equation to assemble
Result: the right-hand side vector of the adjoint system `rhs`
`rhs` $\leftarrow \frac{\partial J}{\partial u_k}$
for *all blocks* $A_{lk} \neq 0$, $l \neq k$ **do**
  `rhs` $\leftarrow$ `rhs` $- A^*_{lk}\lambda_l$
end
for *all equations* l *that depend on* u_k, $l \neq k$ **do**
  for *all blocks* A_{lj} *that depend on* u_k **do**
    `rhs` $\leftarrow$ `rhs` $- \left(\frac{\partial A_{lj}}{\partial u_k} u_j\right)^* \lambda_l$
  end
  for *all* l *such that* b_l *depends on* u_k, $l \neq k$ **do**
    `rhs` $\leftarrow$ `rhs` $+ \frac{\partial b_l}{u_k}^* \lambda_l$
  end
end

When assembling the linear system to be solved for λ_k, we need to consider row k of $(A + G - R)^*$. Equivalently, one can consider column k of $(A + G - R)$.

First, consider the A^* contributions to the right-hand side. Column k of A consists of all blocks in all equations that target variable u_k; when transposed, a block in equation l that targets variable u_k will multiply λ_l.

Next, consider the G^* contributions. Column k of A has a nonzero entry in row l if equation l has a dependency on variable u_k. So every equation l that depends on u_k will contribute a term $G^*_{lk}\lambda_l$ to the right-hand side of the equation for λ_k. To compute G_{lk}, we take the derivative of equation l with respect to variable u_k and contract it with u, as in the example of equation (23).

Finally, consider the R^* contributions. Column k of R consists of the derivatives of the right-hand side b with respect to u_k; therefore, each right-hand side component b_l that depends on u_k will contribute a term $(\partial b_l/\partial u_k)^*\lambda_k$.

Of course, users of `libadjoint` do not need to concern themselves with the details of the algorithms given above. However, they may be of use to discrete adjoint model developers who choose not to use `libadjoint`.

6. FEATURES OF `LIBADJOINT`

6.1. Adjoint model development and debugging

`libadjoint` offers powerful debugging features to rapidly identify developer mistakes at any stage of the adjoint development process.

Firstly, `libadjoint` can check the consistency of the original forward model and the annotation supplied to `libadjoint`. The normal use of `libadjoint` is to rewind that annotation, assembling each adjoint equation in turn; however, with the callbacks and information that the model developer has supplied, the library can also re-play the annotation, to re-run the forward model. (Of course, in order for this to be possible, the model developer must supply an additional callback which computes the source term b.) When debugging the annotation, the model developer can include calls which record the value of every variable computed through the original forward model run.

Then, when `libadjoint` re-plays the annotation, it can automatically compare the recomputed value against the original value, and issue an error if they differ. In this way, the annotation becomes very systematic, as the library can identify the exact equation on which the model and annotation differ.

Secondly, `libadjoint` can check the consistency of the Hermitian and non-Hermitian case for each block. In order to check this, the library employs the identity

$$\langle y, Vx \rangle = \langle V^* y, x \rangle, \tag{31}$$

for any suitable vectors x, y and block V. When instructed by the model developer, `libadjoint` computes both sides of this identity for a prescribed number of random vectors x, y, and checks that both sides are equal to machine precision. This check is particularly useful for operators V which are never explicitly represented as matrices in the model code.

Thirdly, `libadjoint` can check the consistency between the action of a nonlinear operator and its derivative. Let V be the nonlinear operator in question, and let u be its dependency. Let $f(u) = V(u) \cdot c$ for some fixed c. In order to check the consistency of the derivative, the library employs the fact that

$$||f(u + \delta u) - f(u)|| \tag{32}$$

converges to zero at $O(||\delta x||)$ as $||\delta x|| \to 0$, while

$$||f(u + \delta u) - f(u) - \nabla f(u) \cdot \delta u|| \tag{33}$$

converges to zero at $O(||\delta x||^2)$ as $||\delta x|| \to 0$. This check is particularly useful when the nonlinear assembly routine had to be modified so that an algorithmic differentiation tool could be applied, as it asserts the correspondence between the original unmodified code and the derivative of the modified model code.

Finally, `libadjoint` can output its current state to a HTML file, which can be viewed by any web browser. This visualisation contains the annotated forward and adjoint system with details about each block, the registered callbacks and the state of all variables (recorded or not and if yes, where, e.g. disk or memory). This functionality greatly facilitates the model annotation process, since it allows the developer to directly compare the model written in the form of equation (1), with `libadjoint`'s annotation.

These features combine to make annotating the model and developing callbacks as straightforward as possible. Each of these debugging features gives useful feedback to the model developer, pinpointing where any error in the input to the library is located. When these debugging tests pass, the adjoint is almost certainly being assembled correctly.

6.2. Checkpointing

The adjoint operator in equation (16) consists of the three terms $(A + G - R)^*$. For a nonlinear system, A will have dependencies on the forward solution and hence A^* will as well. Similarly, G and R may have complex dependencies on the forward solution. As a consequence, different parts of the forward solution must be available at different timesteps when assembling the adjoint of the linearised forward system. While the straightforward approach of storing the whole forward solution is unproblematic for steady state problems, it can become prohibitively expensive for time-dependent simulations, since the storage requirement increases proportionally to the number of timesteps. For example, a simulation with 10^7 unknowns and 10^5 timesteps would require over 7 terabytes. It is therefore apparent that the available storage would greatly constrain the number timesteps for which a real-world adjoint simulation could be run.

One way to circumvent this problem is to use temporal interpolation: the complete time interval $[t, T]$ is divided into subintervals $t = t_1 < t_2 < \cdots < t_n = T$, where each t_i

corresponds to a timestep and n is much smaller than the number of timesteps. Now instead of storing the variables at every timestep, only the variables at the selected timesteps are kept. When the adjoint assembly requires an intermediate variable, a temporal interpolation scheme is used to approximate its value. Although the generality of this approach is questioned in Gunzburger [2003, §5.1], it is sometimes used in hand-written codes; the adjoint of the NEMO ocean model is one such example [Vidard et al. 2008, §1.3.1]. Their experience shows that the necessary code manipulation "requires deep knowledge of the original program and of the underlying equations ... [and] introduces approximation errors into the computed derivatives, whose mathematical behavior is unclear" [Tber et al. 2007, §4.3].

The problems associated with interpolation may be circumvented using checkpoints, from which the forward simulation can be restarted. In the simplest checkpointing strategy, the complete time interval is split into a small number of equidistant subintervals and a checkpoint is stored at the beginning of each interval. When an intermediate value is required for the adjoint assembly, the forward simulation is restarted from the closest preceding checkpoint and run up to the point at which the required value is recomputed. An extension known as multi-level checkpointing applies this idea recursively: the time interval of the partial forward integration is again split into several subintervals on which checkpoints are stored, and so on. This multi-level checkpointing is successfully used in the MITgcm ocean model [Heimbach et al. 2005].

Griewank [1992] proposed a strategy related to the multi-level approach in which the checkpoint distribution is based on a binomial interval splitting. By reusing the available checkpoint slots, a logarithmic growth of temporal and computational complexity is achieved, which is proven to be optimal [Grimm et al. 1996]. Walther and Griewank [2003] showed that this approach is indeed advantageous over the multi-level checkpointing strategy and published a library named `revolve` that facilitates its implementation [Griewank and Walther 2000].

Both multi-level checkpointing and `revolve` allow the adjoint model user to balance the storage and computational complexity to their needs. However, this flexibility comes with additional development effort: the control flow switches context between the forward and adjoint main loops, making the implementation of checkpointing more difficult than the temporal interpolation.

While this can involve extensive development effort for a hand-written adjoint, the abstraction of AD is powerful enough to automatically generate the adjoint control flow according to the checkpointing strategy. Since the tape can be used to restart the simulation from a checkpoint, the checkpointing logic can be implemented entirely within the AD tool. However, because the abstraction of AD considers the model as a sequence of elemental instructions, it does not natively know about the concept of timestepping, which is required for efficient checkpointing. This information must therefore be provided by the model developer, often by explicitly adding AD-specific directives to the forward code. AD tools that offer checkpointing include TAF [Giering and Kaminski 2002] and ADOL-C [Kowarz and Walther 2006].

The abstraction of `libadjoint` is similar to AD but its level is much higher. As with AD, it is sufficiently powerful to execute the required operations for checkpointing entirely within the library. In particular, the annotation and the supplied callbacks allow `libadjoint` to restart the simulation from a checkpoint. But in contrast to AD, `libadjoint`'s annotation also provides the required timestep information. This makes checkpointing with `libadjoint` available for almost no extra model development effort: the only required callbacks are for handling checkpoints and a callback to solve linear systems. The checkpointing strategy used in `libadjoint` is the optimal checkpointing algorithm of Griewank [1992], as implemented in the `revolve` library [Griewank and Walther 2000].

Checkpointing is a crucial feature for an efficient, time-dependent adjoint model. That it is possible to implement the optimal checkpointing strategy entirely within the library is a significant advantage over hand-written adjoints, where its implementation can be a prohibitively difficult task.

7. DISCUSSION

7.1. Maintainability

Developing an adjoint model is a significant investment; maintaining the adjoint as the forward model changes is even more so. An adjoint model left unattended will soon lose consistency with the forward model, and will become useless as a tool for doing science. In this regard, AD has a major advantage: as the AD tool can continually be re-applied to the changing forward code base, the adjoint is always consistent with the forward model (barring bugs in the AD tool, of course, or the introduction of non-differentiable features in the model), and so the maintenance burden is approximately zero. By contrast, if the adjoint model is hand-written, then every change to the forward model must be matched by a corresponding change to the adjoint model, leading to a very high maintenance burden. Heimbach et al. [2010] states: "the work of keeping the [hand-written] adjoint model up-to-date with its forward parent model matches the work of forward model development."

The `libadjoint` approach compares favourably to writing the discrete adjoint model by hand. Firstly, the annotation itself changes only rarely; even if the discretisation implemented is tweaked, the pattern of the equations (i.e., the target and dependencies of the operators) changes only if a major new discretisation is implemented. If the callbacks supplied to `libadjoint` use the code in the forward model itself for computing the action/assembly of the operators, then the change will also be propagated to the adjoint model, and the forward and adjoint models will still be consistent. If the implementation of a nonlinear operator changes, then one can re-apply an AD tool to recompute its derivative code. So while the maintenance burden may be higher than in the pure AD case, we expect that the maintenance load of an adjoint model developed with `libadjoint` will be manageable.

One mechanism for reducing the maintenance load further is to automate the process of verifying the consistency between the forward model and the adjoint model. For example, the MITgcm project runs a nightly suite of forward and adjoint test cases so that any problems are detected as soon as possible. All of the debugging features described in section 6.1 can be automated so that any change that induces an inconsistency is not accepted into the model source code. For more details of the strategy implemented for automated model verification, see Farrell et al. [2011].

7.2. Advantages and disadvantages

Using `libadjoint` to assemble the discrete adjoint equations has a certain fixed overhead in terms of development effort: the model must be annotated and callbacks must be interfaced, regardless of how complex or AD-able the model is. Therefore, if the model has been written in such a way that an AD tool is immediately applicable, taking the pure AD approach will almost certainly be faster and easier to develop. As suggested by the examples of MITgcm and Fluidity in section 1.3, `libadjoint` is not intended for this class of models. `libadjoint` does not replace AD tools: it extends their domain of applicability to models for which AD is currently impractical. However, for the class of models where an AD tool is not immediately applicable, the `libadjoint` approach compares very favourably to writing the discrete adjoint equations by hand.

Firstly, the expertise required to implement the adjoint at all is greatly reduced: the model developer must describe the forward model in some detail, but the derivation

and assembly of the adjoint equations is performed by the library. `libadjoint` manages the lifecycles of the forward and adjoint variables, so that the model developer does not need to implement the complicated algorithm for deciding when a variable may be deallocated. These features combine to make the development of adjoint models more accessible to research groups and industrial teams that currently lack the necessary expertise, and so the many applications of adjoints can be deployed more widely.

Secondly, the process is made much more systematic: by adopting its high-level abstraction, `libadjoint` can pinpoint exactly where the model developer has made a mistake in the process of adjoint model development. By contrast, if the discrete adjoint equations were assembled by hand and found to be incorrect, then the model developer is faced with the very difficult task of finding exactly where the bugs lie. With the `libadjoint` approach, the adjoint model development is incrementally verifiable.

Thirdly, the adjoint model developed has a much lower maintenance burden than the corresponding hand-written adjoint. Many changes that would require developer intervention to retain consistency no longer do so. The model annotation changes rarely. Furthermore, the debugging features of section 6.1 can be automated so that the model developers are notified when a change has lost the consistency of the adjoint model.

Fourthly, by adopting its high-level abstraction, the library is able to provide powerful features that would be prohibitively complex to implement by hand. As described in section 6.2, `libadjoint` can implement an optimal checkpointing scheme with almost no extra effort from the model developer. As noted previously, many hand-written adjoint models do not implement a checkpointing scheme due to the complexity of its implementation, and so their adjoint is severely restricted in the science it can do because of the prohibitive storage costs. The fact that it is possible to use an optimal checkpointing scheme entirely within `libadjoint` strongly suggests that the abstraction on which the library is based is a useful one.

Based on these points, we suggest that `libadjoint` is a significant advance over writing the discrete adjoint model by hand.

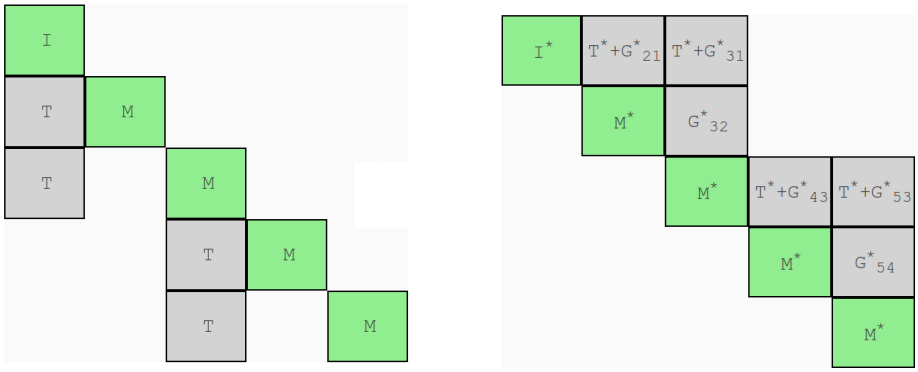
8. EXAMPLES

8.1. Burgers' equation

As part of its developer training documentation, Fluidity includes a small example solver which solves the Burgers' equation (equation (8)). This model was adjoined as a proof-of-concept for `libadjoint`. For the reasons explained in section 1.3, we are unaware of any current free AD tool that is applicable to this model.

Firstly, work was undertaken to implement the data callbacks, to allow `libadjoint` to manipulate Fluidity's scalar, vector and tensor field classes. Next, a system was developed for model users to express their functional of interest. One approach would be to hard-code each functional in Fortran, and select between them with `#ifdef` switches or runtime options, but this would limit users to selecting between the small pool of implemented functionals. Fluidity embeds a Python interpreter [van Rossum et al. 2008] to provide dynamic programming facilities to model users; this was extended to provide a Python interface with which users could code their own functionals of interest in a flexible and user-friendly manner. While this sacrifices some efficiency, functionals are generally quite cheap to compute, and their cost is usually dominated by the cost of the forward/reverse PDE solve; if a computationally expensive functional were desired, the functional could easily be implemented in Fortran.

Then, the model was annotated with an annotation similar to that presented in section 4. The library calls for the annotation are distributed through the model, close to the code which they describe: the annotation of each nonlinear iteration happens within the nonlinear iteration loop, the timestep annotation happens within the



(a) The forward matrix A (compare to equation (10)).

(b) The adjoint matrix $(A+G)^*$. The sparsity pattern of G was derived in equation (22).

Fig. 2: The HTML output of `libadjoint` on the Burgers' equation model. The green background denotes the diagonal blocks; and the grey background marks off-diagonal, nonzero blocks.

timestep loop, etc. At this point, the annotation was visualised in HTML and inspected to ensure it matched the expectations of the model developers, see figure 2.

Once the annotation was complete, the callbacks were interfaced and registered. Matrices that do not change throughout the simulation, such as the mass and diffusion matrices, were cached; the advection matrix must be reassembled at the arguments supplied. Fortunately, the existing forward model was well-written, and the functionality to assemble the advection matrix was available as a subroutine call. At this point, the annotation and callbacks were debugged by comparing the original model run against the forward replay mode of `libadjoint` (section 6.1).

Next, the necessary derivatives were supplied to `libadjoint`. The differentiation of the functional was achieved with `uncertainties`, an object-overloading automatic differentiation tool for Python [Lebigot 2011]. Differentiating the nonlinear advection term was more challenging, as the subroutine made extensive use of derived types which are poorly supported by free AD tools. To that end, the advection assembly subroutine was re-written in Fortran 77, and code for its derivative was generated with TAPENADE, a source-to-source AD tool [Hascoët and Pascual 2004]. At this point, the consistency between the original Fortran 90 advection routine and the derivative of the rewritten Fortran 77 advection routine was asserted with the derivative test described in section 6.1. This test found several inconsistencies between the original and rewritten advection routines, which were rapidly eliminated. Again, it is to be emphasised that no freely-available AD tool is capable of differentiating the entire model, and so the pure AD approach is impractical. However, it was entirely practical to reimplement a Fortran 77 version of just the nonlinear advection operator, as the amount of code to convert to an AD-differentiable form was very small.

With the derivatives supplied, `libadjoint` successfully assembled each adjoint equation in turn, beginning at the end of simulation time and propagating backwards. The gradient of the functional with respect to various parameters (initial conditions, source terms) was computed with

$$\frac{dJ}{dm} = -\langle \lambda, \frac{\partial F}{\partial m} \rangle + \frac{\partial J}{\partial m}, \tag{34}$$

Number of mesh elements	$2 \cdot 10^4$	$4 \cdot 10^4$	$8 \cdot 10^4$	$16 \cdot 10^4$
Runtime of forward solve (s)	17	33	65	129
Runtime of adjoint solve (s)	17	33	66	134

Table I: Runtime comparison of the forward and adjoint Burgers' equation model with varying mesh resolution. The timestep is fixed to $\Delta t = 1/32$ s.

where m is the vector of control parameters and $F \equiv A(u)u - b(u)$ [Gunzburger 2003, equation 2.34]. The correctness of the adjoint solution was verified with a derivative test analogous to that of equation (33), by examining the order of convergence of $J(m + \delta m) - J(m) - dJ/dm \cdot \delta m)$. As expected, the Taylor remainder converged at second-order for a wide variety of simulations, functionals and choice of control parameters, giving high confidence in the correctness of the adjoint equations assembled by `libadjoint`.

To benchmark the efficiency of the implementation, the adjoint model was applied to a steady-state 1D problem with known analytical solution $u = sin(x) + cos(x)$ in the domain interval $[-10, 10]$. A pseudo time-stepping approach with a 1/32 s timestep size was applied for 1 s, which is sufficient to converge to the steady-state solution. The PDE was discretised in space using the finite element method with piecewise linear basis functions on a uniform mesh, and in time using the Crank-Nicolson scheme [Crank and Nicolson 1947]. The runtime benchmarks were performed on one 2.13 GHz Intel Xeon CPU core. To obtain comparable timings for the forward and adjoint model without incorporating model specific overhead, the benchmark includes only the nonlinear assembly, solver times and `libadjoint` related function calls. Specifically, the assembly of the linear operators is excluded, since they are assembled once in the initialisation step and then reused in both the forward and adjoint main loop. Moreover, the runtime of the embedded Python interpreter and I/O are excluded. The averaged results of four runs are given in table I. They show that the adjoint model is almost as fast as the original forward code, even though the G^* matrix contributions have to be computed for the adjoint run, which are not present in the forward run. It may be concluded that, for this example at least, the adjoint implementation produced by `libadjoint` is very efficient compared with the forward model.

8.2. Shallow water model

Fluidity also includes a shallow water model which is significantly more complex than the Burgers' equation solver presented in the previous section. The shallow water model uses the advanced $\mathrm{P1_{DG}}$-P2 discretisation, which is both LBB-stable and can represent geostrophic balance exactly; the model equations and discretisation are fully described in Cotter et al. [2009]. Unlike the Burgers' equation model, the shallow water model has more than one prognostic variable. The model is also made more complex by the fact that it solves on arbitrary one- or two-dimensional manifolds embedded in three-dimensional space, following the strategy of Bernard et al. [2009].

As the shallow water model is also part of Fluidity, it was possible to use the data callbacks for Fluidity's datatypes and the Python interface for implementing functionals from the previous work on the Burgers' equation model. The forward model was annotated, and the callbacks implemented. For this work, we chose to adjoint the linear shallow water model, as the implementation of the nonlinear term on arbitrary manifolds is still under development. Since almost all of the operators were cached by the forward model anyhow, the callbacks merely used these cached matrices. The only matrices not cached were the projection operators associated with the capacity to solve on arbitrary manifolds; the code for these projection operators was modularised to isolate the functionality in subroutines that the callbacks could use.

The debugging features of section 6.1 were applied to detect potential implementation errors. The annotation of the forward model was verified by performing the forward replay test. Once this worked, the adjoint equations were assembled. An initial application of the Hermitian consistency check (equation 31) indicated that the manifold projection operators were not self-adjoint. The required transposed projections were implemented by hand after which the Hermitian consistency check passed. Finally, the correctness of the adjoint solution was verified using the derivative test of equation (33). As expected, the Taylor remainder converged at second-order for a wide variety of simulations with different functionals and choice of control parameters.

Despite the increased code complexity of the forward model, the process of adjoining the shallow water model was largely similar to that of the Burgers' equation, and no more difficult. Using `libadjoint` has a fixed overhead of annotation, but the fact that the process of developing the adjoint model is systematic and incrementally verifiable means that the difficulty of adjoining models scales well with model complexity.

8.3. Application of the shallow water adjoint model to an idealised data assimilation problem

Once the shallow water adjoint model is implemented and verified, it can be used in a wide range of applications: data and parameter estimation, sensitivity and stability analysis, design optimisation, and error estimation. A short introduction can be found in Errico [1997, §5], Giles and Pierce [2000] and Moore et al. [2004].

To demonstrate the functionality and efficiency of the shallow water adjoint model described in the previous section, it was used to solve an idealised data assimilation problem. In data assimilation, some of the parameters specifying the problem are not exactly known; however, observations of the solution (possibly at different time levels if the problem is time-dependent) are available. The goal of data assimilation is to find a better estimate of the unknown input parameters for which the solution best "fits" the observations, usually in a least-squares sense. These kind of problems arise in many fields of geosciences, in particular in weather prediction; a detailed discussion is beyond the scope of this paper, but can be found in Park and Xu [2009].

Here, we restrict ourself to a simple shallow water setup in a two-dimensional, doubly-periodic domain $\Omega = [0,1]^2$. In the considered problem, the initial value (at time $t = 0$) for the layer thickness η is unknown, while that of the velocity u is known. Given the observation of the entire η field at time $t = 1/2$ and $t = 1$, we seek the initial condition for η that recovers these observations. With the misfit functional defined as:

$$J(\eta) := ||\eta(t = 1/2) - \eta^{obs}_{t=1/2}||^2_{L_2} + ||\eta(t = 1) - \eta^{obs}_{t=1}||^2_{L_2}, \tag{35}$$

this problem can be formulated mathematically as an optimisation problem:

$$\min_{\eta_{est}} J(\eta_{est}) \quad \text{subject to} \tag{36a}$$

$$\frac{\partial u}{\partial t} + g\nabla\eta = 0 \tag{36b}$$

$$\frac{\partial \eta}{\partial t} + h\nabla \cdot u = 0 \tag{36c}$$

$$u(t = 0) = u_0 \tag{36d}$$

$$\eta(t = 0) = \eta_{est} \tag{36e}$$

$$u(x = 0) = u(x = 1), \quad u(y = 0) = u(y = 1) \tag{36f}$$

$$\eta(x = 0) = \eta(x = 1), \quad \eta(y = 0) = \eta(y = 1). \tag{36g}$$

The minimum value of (36a) is zero which is achieved if and only if the observations $\eta^{obs}_{t=1/2}$ at time $t = 1/2$ and $\eta^{obs}_{t=1}$ at time $t = 1$ are exactly recovered. The constraints (36b)

and (36c) are the momentum and pressure equation of the linear shallow water equations. The parameters g and h are the gravity constant and the mean layer thickness respectively. Equations (36d) and (36e) enforce the initial condition: u_0 is the known velocity initial condition and η_{est} is the estimate of the layer thickness initial condition. Finally, the equations (36f) and (36g) enforce the periodic boundary conditions.

This optimisation problem can be solved iteratively as follows. Firstly, use the estimate η_{est} to solve the shallow water model and its adjoint with respect to the misfit functional J. Then, equation (34) yields the derivative $dJ/d\eta$ at the current point in parameter space. Finally, a gradient based optimisation algorithm is applied to obtain a better estimate for η_{est}, with which the procedure is restarted. For the following benchmarks, the L-BFGS-B algorithm [Byrd et al. 1995] of SciPy [Jones et al. 2001] was used as optimisation algorithm (no bounds or memory limit were specified, in which case the L-BFGS-B algorithm is equivalent to BFGS). Starting with an initial estimate $\eta_{est} \equiv 0$, the optimisation loop was repeated until either the gradient norm of J was less than 10^{-12} or $(J^k - J^{k+1})/\max(|J^k|, |J^{k+1}|, 1) \leq f \cdot \epsilon$, where the superscript denotes the iteration of the optimisation loop and ϵ the machine precision.

In order to verify the implementation of the solution procedure, its order of convergence was checked against theoretical results. For this, an analytical solution for problem (36) is required. By choosing $g = h = 1$, a periodic solution for equations (36b) and (36c) is given by:

$$u = \begin{pmatrix} \sin(2\pi(t+x)) \\ \sin(2\pi(t+y)) \end{pmatrix}$$
$$\eta = -\sin(2\pi(t+y)) - \sin(2\pi(t+x)).$$

From this, the initial condition and observations are derived:

$$u_0 = u(t=0)$$
$$\eta^{obs}_{t=1/2} = \eta(t=1/2)$$
$$\eta^{obs}_{t=1} = \eta(t=1).$$

With a correct implementation, we expect the final η_{est} in the optimisation loop to converge to $\eta(t=0)$ at the same order of convergence as the method used in the forward discretisation.

The forward and adjoint shallow water problems were solved with the model described in section 8.2. The forward model uses Crank-Nicolson in time [Crank and Nicolson 1947] and the $\mathrm{P1_{DG}}$-P2 finite element in space [Cotter et al. 2009]. For the interpolated initial conditions employed here, $\mathrm{P1_{DG}}$-P2 converges at second order in space [Cotter and Ham 2011] while Crank-Nicolson achieves second order in time. However, any error in the implementation of the discretisation is likely to break either the spatial or temporal order of convergence. Problem (36) was therefore run with four different mesh resolutions and a very small timestep to derive the spatial order of convergence. The results are shown in table II: as expected, second order convergence is observed for both the control variable η_{est} and the final state variables (to observe second order convergence for η_{est} with the highest resolution, it was necessary to perform 9 additional optimisation iterations after reaching the stopping criteria of the optimisation loop). The same technique was used to check the temporal order of convergence. The results in table III show again second order convergence, giving high confidence in the correctness of the data assimilation implementation.

To benchmark the efficiency of the implementation, the average time required to solve the forward and adjoint system was recorded. The problem was computed four times and the results averaged. Table IV shows almost identical timings for both

Mesh element size	0.3	$0.3/2$	$0.3/4$	$0.3/8$
L-BFGS-B iterations	24	27	29	$31 + 9$
Rate of convergence for η_{est}		2.0	2.3	2.1
Rate of convergence for $\eta(t = 1)$		1.6	2.2	2.2
Rate of convergence for $u(t = 1)$		2.3	2.2	2.3

Table II: Spatial order convergence of the data assimilation problem. The timestep size is $5 \cdot 10^{-5}$ s. The expected order is 2.

Timestep size (s)	$1/4$	$1/8$	$1/16$	$1/32$
L-BFGS-B iterations	20	20	30	28
Rate of convergence for η_{est}		1.9	1.9	2.0
Rate of convergence for $\eta(t = 1)$		2.5	2.0	1.9
Rate of convergence for $u(t = 1)$		1.9	1.9	2.0

Table III: Temporal order convergence of the data assimilation problem. The mesh element size is $0.3/16$. The expected order is 2.

Mesh element size	$0.3/4$	$0.3/8$	$0.3/16$
Timestep size (s)	$1/32$	$1/64$	$1/128$
Runtime of forward solve (s)	2.4	29	464
Runtime of adjoint solve (s)	2.5	29	474
`libadjoint` annotation time (s)	0.03	0.07	0.2

Table IV: Averaged runtimes of the forward/adjoint solve and the total execution time of the `libadjoint` annotation in the data assimilation problem.

solves, which can be interpreted as follows: consider the shallow water model cast in the form given in equation 1. The linearity of the problem yields a forward operator A which is independent of the solution vector. With equation (17) we obtain that $G \equiv 0$. Similarly, the right hand side b of equation 1 does not depend on the solution, yielding $R \equiv 0$ (see equation (18)). Hence the adjoint operator is just the transpose of the forward operator A, see equation (16). An efficient adjoint implementation is therefore expected to be approximately as fast as the forward equivalent. It is remarkable that even though the application of `libadjoint` has been performed without any optimisation, the benchmark timings suggest that the efficiency of the resulting adjoint model is very close to this best case assumption. Finally, the execution time of `libadjoint` to annotate the forward model was measured and was found to be less than 2% of the forward running time, see table IV.

9. SUMMARY AND OUTLOOK

A new approach to implementing discrete adjoint models has been presented. Whereas algorithmic differentiation tools are based on the abstraction that the model is *a sequence of elemental instructions*, we instead treat the model as *a sequence of linear solves*: the entire approach flows from exploring this higher-level abstraction. We have written an open-source library, `libadjoint`, which implements this abstraction. The model developer annotates the forward model using library calls with a description of the equations being solved, and supplies callbacks to assemble the featured operators; with this information, `libadjoint` can symbolically manipulate the recorded forward equations, to deduce and assemble the discrete adjoint equations. This approach is intended to extend the domain of applicability of adjoint techniques to models for which the pure AD approach is impractical.

The `libadjoint` approach has several major advantages over hand-writing the discrete adjoint code: it requires significantly less expertise in adjoint theory, the process of adjoint model development becomes systematic and incrementally verifiable, the maintenance burden of the adjoint model is considerably reduced, and it is possible to offer complex features such as checkpointing schemes for balancing storage and recomputation costs entirely within the library itself. Therefore, we suggest it represents a superior alternative to manually implementing the discrete adjoint equations.

We intend to extend `libadjoint` further in future projects. While this paper has focussed entirely on implementing adjoint models, there is no reason why this approach will not also apply to implementing tangent linear models. In fact, once the necessary functionality is implemented in `libadjoint`, models that use the library for their adjoint will get a tangent linear model for almost no extra work. As documented in Moore et al. [2004], the combination of an adjoint and tangent linear model offers extremely powerful tools to computational scientists.

The `libadjoint` abstraction can also be pushed in interesting directions where it is difficult to extend AD. As computers become more powerful, there is an increasing trend towards more realistic multiphysics simulations which couple several individual models together; the climate models employed in the IPCC Coupled Model Intercomparison Project offer an excellent and important example [Randall et al. 2007]. In order to maintain modularity and flexibility of the individual components (for example, the separate ocean, atmosphere, ice and land models) are maintained separately, and combined through a coupler which organises data transfer between and execution of components (weak coupling). The pure AD approach soon runs into great difficulty with such a system; while coupled models have been differentiated with AD, they have been restricted to tightly-coupled models, where the models are embedded into the same codebase [Galanti et al. 2002; Kauker et al. 2009; Heimbach et al. 2010]. By contrast, we hope that the semantic annotation approach of `libadjoint` will extend naturally to weakly coupled models, provided some interaction with the coupling software is performed.

Adjoint models have already made a large impact across a number of scientific and industrial domains, and their impact can only be strengthened by facilitating the process of adjoint model development. By making adjoint model development cheaper and easier, we hope that `libadjoint` can further popularise these powerful techniques.

REFERENCES

BALAY, S., BROWN, J., BUSCHELMAN, K., EIJKHOUT, V., GROPP, W. D., KAUSHIK, D., KNEPLEY, M. G., MCINNES, L. C., SMITH, B. F., AND ZHANG, H. 2010. PETSc users manual. Tech. Rep. ANL-95/11, Argonne National Laboratory. Revision 3.1.

BALAY, S., GROPP, W. D., MCINNES, L. C., AND SMITH, B. F. 1997. Efficient management of parallelism in object oriented numerical software libraries. In *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, 163–202.

BERNARD, P. E., REMACLE, J. F., COMBLEN, R., LEGAT, V., AND HILLEWAERT, K. 2009. High-order discontinuous Galerkin schemes on general 2D manifolds applied to the shallow water equations. *Journal of Computational Physics 228,* 17, 6514–6535.

BISCHOF, C. H., BÜCKER, H. M., RASCH, A., SLUSANSCHI, E., AND LANG, B. 2007. Automatic differentiation of the general-purpose computational fluid dynamics package FLUENT. *Journal of Fluids Engineering 129,* 5, 652–658.

BYRD, R. H., LU, P., NOCEDAL, J., AND ZHU, C. 1995. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific and Statistical Computing 16,* 5, 1190–1208.

COLEMAN, T. F., SANTOSA, F., AND VERMA, A. 1998. Semi-automatic differentiation. In *Computational methods for optimal design and control: proceedings of the AFOSR Workshop on Optimal Design and Control*. Vol. 24. Birkhäuser, Arlington, VA, 113.

COTTER, C. AND HAM, D. 2011. Numerical wave propagation for the triangular p1dg-p2 finite element pair. *Journal of Computational Physics 230,* 8, 2806 – 2820.

COTTER, C. J., HAM, D. A., AND PAIN, C. C. 2009. A mixed discontinuous/continuous finite element pair for shallow-water ocean modelling. *Ocean Modelling 26,* 1-2, 86–90.

CRANK, J. AND NICOLSON, P. 1947. A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type. In *Mathematical Proceedings of the Cambridge Philosophical Society*. Vol. 43. 50–67.

DAVIES, D. R., WILSON, C. R., AND KRAMER, S. C. 2011. Fluidity: A fully unstructured anisotropic adaptive mesh computational modeling framework for geodynamics. *Geochemistry Geophysics Geosystems 12,* 6.

DEVINE, K., BOMAN, E., HEAPHY, R., HENDRICKSON, B., AND VAUGHAN, C. 2002. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering 4,* 2, 90–97.

ERRICO, R. M. 1997. What is an adjoint model? *Bulletin of the American Meteorological Society 78,* 11, 2577–2591.

FANG, F., PAIN, C. C., NAVON, I. M., GORMAN, G. J., PIGGOTT, M. D., AND ALLISON, P. A. 2010. The independent set perturbation adjoint method: a new method of differentiating mesh based fluids models. *International Journal for Numerical Methods in Fluids*.

FARRELL, P. E. AND FUNKE, S. W. 2011. *libadjoint: a library for developing adjoint models (Version 0.8)*. http://launchpad.net/libadjoint.

FARRELL, P. E., PIGGOTT, M. D., GORMAN, G. J., HAM, D. A., WILSON, C. R., AND BOND, T. M. 2011. Automated continuous verification for numerical simulation. *Geoscientific Model Development 4,* 2, 435–449.

GALANTI, E., TZIPERMAN, E., HARRISON, M., ROSATI, A., GIERING, R., AND SIRKES, Z. 2002. The equatorial thermocline outcropping – a seasonal control on the tropical pacific oceanâ??atmosphere instability strength. *Journal of Climate 15,* 19, 2721–2739.

GIERING, R. AND KAMINSKI, T. 1998. Recipes for adjoint code construction. *ACM Transactions on Mathematical Software 24,* 4, 437–474.

GIERING, R. AND KAMINSKI, T. 2002. Recomputations in reverse mode AD. In *Automatic Differentiation of Algorithms: From Simulation to Optimization*, G. Corliss, A. Griewank, C. Faurè, L. Hascoët, and U. Naumann, Eds. Automatic Differentiation of Algorithms: From Simulation to Optimization. Springer Verlag, Heidelberg, Chapter 33, 283–291.

GIERING, R. AND KAMINSKI, T. 2003. Applying TAF to generate efficient derivative code of Fortran 77-95 programs. *Proceedings in Applied Mathematics and Mechanics 2,* 1, 54–57.

GILES, M., GHATE, D., AND DUTA, M. 2005. Using automatic differentiation for adjoint CFD code development. In *Post-SAROD Indo-French Workshop on Recent Developments in Tools for Aerodynamics & Multidisciplinary Optimization*. Bangalore.

GILES, M. B. AND PIERCE, N. A. 2000. An introduction to the adjoint approach to design. *Flow, Turbulence and Combustion 65,* 3-4, 393–415.

GRIEWANK, A. 1992. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software 1,* 1, 35–54.

GRIEWANK, A. 2003. A mathematical view of automatic differentiation. *Acta Numerica 12*, 321–398.

GRIEWANK, A. 2008. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. Frontiers in Applied Mathematics. SIAM.

GRIEWANK, A. AND WALTHER, A. 2000. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software 26,* 1, 19–45.

GRIMM, J., POTTIER, L., AND ROSTAING-SCHMIDT, N. 1996. Optimal time and minimum space-time product for reversing a certain class of programs. In *Computational Differentiation: Techniques, Applications, and Tools*, M. Berz, C. H. Bischof, G. F. Corliss, and A. Griewank, Eds. SIAM, Philadelphia, PA, 95–106.

GUNZBURGER, M. D. 2003. *Perspectives in Flow Control and Optimization*. Advances in Design and Control. Society for Industrial Mathematics.

HASCOËT, L. AND PASCUAL, V. 2004. Tapenade 2.1 user's guide. Tech. Rep. RT-0300, INRIA Sophia Antipolis, Sophia Antipolis, FR 06902.

HEIMBACH, P., HILL, C., AND GIERING, R. 2005. An efficient exact adjoint of the parallel MIT General Circulation Model, generated via automatic differentiation. *Future Generation Computer Systems 21,* 8, 1356–1371.

HEIMBACH, P., MENEMENLIS, D., LOSCH, M., CAMPIN, J. M., AND HILL, C. 2010. On the formulation of sea-ice models. Part 2: Lessons from multi-year adjoint sea-ice export sensitivities through the Canadian Arctic Archipelago. *Ocean Modelling 33,* 1-2, 145–158.

HEROUX, M., BARTLETT, R., HOWLE, V., HOEKSTRA, R., HU, J., KOLDA, T., LEHOUCQ, R., LONG, K., PAWLOWSKI, R., PHIPPS, E., SALINGER, A., THORNQUIST, H., TUMINARO, R., WILLENBRING, J., AND WILLIAMS, A. 2003. An overview of Trilinos. Tech. Rep. SAND2003-2927, Sandia National Laboratories.

HINZE, M., PINNAU, R., ULBRICH, M., AND ULBRICH, S. 2009. *Optimization with PDE constraints*. Mathematical Modelling: Theory and Applications Series, vol. 23. Springer.

JONES, E., OLIPHANT, T., PETERSON, P., ET AL. 2001. SciPy: Open source scientific tools for Python.

KAUKER, F., KAMINSKI, T., KARCHER, M., GIERING, R., GERDES, R., AND VOSSBECK, M. 2009. Adjoint analysis of the 2007 all time Arctic sea-ice minimum. *Geophysical Research Letters 36,* 3, L03707.

KIM, J., HUNKE, E., AND LIPSCOMB, W. 2006. A sensitivity-enhanced simulation approach for Community Climate System Model. In *Computational Science – ICCS 2006*, V. Alexandrov, G. van Albada, P. Sloot, and J. Dongarra, Eds. Lecture Notes in Computer Science Series, vol. 3994/2006. 533–540.

KOWARZ, A. AND WALTHER, A. 2006. Optimal checkpointing for time-stepping procedures in ADOL-C. In *Computational Science–ICCS 2006*, V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds. Springer, Reading, UK, 541–549.

LEBIGOT, E. 2011. *Uncertainties: a Python module for calculations with uncertainties (Version 1.5.4)*.

MARSHALL, J., ADCROFT, A., HILL, C., PERELMAN, L., AND HEISEY, C. 1997. A finite-volume, incompressible navier stokes model for studies of the ocean on parallel computers. *Journal of Geophysical Research 102,* C3, 5753–5766.

MARTA, A. C., MADER, C. A., MARTINS, J. R. R. A., VAN DER WEIDE, E., AND ALONSO, J. J. 2007. A methodology for the development of discrete adjoint solvers using automatic differentiation tools. *International Journal of Computational Fluid Dynamics 21,* 9, 307–327.

MARTINS, J. R. R. A., STURDZA, P., AND ALONSO, J. J. 2003. The complex-step derivative approximation. *ACM Transactions on Mathematical Software 29,* 3, 245–262.

MOORE, A. M., ARANGO, H. G., LORENZO, E. D., CORNUELLE, B. D., MILLER, A. J., AND NEILSON, D. J. 2004. A comprehensive ocean prediction and analysis system based on the tangent linear and adjoint of a regional ocean model. *Ocean Modelling 7,* 1-2, 227–258.

MÜLLER, J.-D. AND CUSDIN, P. 2005. On the performance of discrete adjoint cfd codes using automatic differentiation. *International Journal for Numerical Methods in Fluids 47,* 8-9, 939–945.

PARK, S. K. AND XU, L. 2009. *Data Assimilation for Atmospheric, Oceanic and Hydrologic Applications*. Springer Verlag.

PIGGOTT, M. D., GORMAN, G. J., PAIN, C. C., ALLISON, P. A., CANDY, A. S., MARTIN, B. T., AND WELLS, M. R. 2008. A new computational framework for multi-scale ocean modelling based on adapting unstructured meshes. *International Journal for Numerical Methods in Fluids 56,* 8, 1003–1015.

RALL, L. B. AND CORLISS, G. F. 1996. An introduction to automatic differentiation. In *Computational Differentiation: Techniques, Applications, and Tools*, M. Berz, C. H. Bischof, G. F. Corliss, and A. Griewank, Eds. SIAM, Philadelphia, PA, 1–17.

RANDALL, D. A., WOOD, R. A., BONY, S., COLMAN, R., FICHEFET, T., FYFE, J., KATTSOV, V., PITMAN, A., SHUKLA, J., SRINIVASAN, J., STOUFFER, R., SUMI, A., AND TAYLOR, K. 2007. Climate models and their evaluation. In *Climate Change 2007: The Physical Science Basis*, S. Solomon, D. Qin, M. Manning, Z. Chen, M. Marquis, K. Averyt, M.Tignor, and H. Miller, Eds. Cambridge University Press, 589–662.

STEIN, W. A. ET AL. 2011. *Sage Mathematics Software (Version 4.5.3)*.

STEIN, W. A. AND JOYNER, D. 2005. SAGE: System for Algebra and Geometry Experimentation. *ACM SIGSAM Bulletin 39,* 2, 61–64.

TBER, M. H., HASCOËT, L., VIDARD, A., AND DAUVERGNE, B. 2007. Building the tangent and adjoint codes of the ocean general circulation model OPA with the automatic differentiation tool TAPENADE. Tech. Rep. RR-6372, INRIA, Sophia Antipolis.

UTKE, J., NAUMANN, U., FAGAN, M., TALLENT, N., STROUT, M., HEIMBACH, P., HILL, C., AND WUNSCH, C. 2008. OpenAD/F: A modular open-source tool for automatic differentiation of Fortran codes. *ACM Transactions on Mathematical Software 34,* 4, 1–36.

VAN ROSSUM, G., DRAKE, F. L., ET AL. 2008. Python reference manual.

VIDARD, A., VIGILANT, F., DELTEL, C., AND BENSHILA, R. 2008. *NEMO tangent & adjoint models (NemoTam): reference manual & user's guide*. ANR-08-COSI-016.

WALTHER, A. AND GRIEWANK, A. 2003. Advantages of binomial checkpointing for memory-reduced adjoint calculations. In *Proceedings of ENUMATH 2003*, M. Feistauer, V. Dolejí, P. Knobloch, and K. Najzar, Eds. Springer, Prague, Czech Republic, 834–843.

WANG, Q., MOIN, P., AND IACCARINO, G. 2009. Minimal repetition dynamic checkpointing algorithm for unsteady adjoint calculation. *SIAM Journal on Scientific Computing 31,* 4, 2549–2567.

XIAO, Q., KUO, Y. H., MA, Z., HUANG, W., HUANG, X. Y., ZHANG, X., BARKER, D. M., MICHALAKES, J., AND DUDHIA, J. 2008. Application of an adiabatic WRF adjoint to the investigation of the May 2004 McMurdo, Antarctica, severe wind event. *Monthly Weather Review 136,* 10, 3696–3713.

Received August 2011; revised Month Year; accepted Month Year