Automated calculation of higher order partial differential equation constrained derivative information

AUTOMATED CALCULATION OF HIGHER ORDER PARTIAL DIFFERENTIAL EQUATION CONSTRAINED DERIVATIVE INFORMATION

J. R. MADDISON *, D. N. GOLDBERG †, AND B. D. GODDARD ‡

Abstract. Developments in automated code generation have allowed extremely compact representations of numerical models, and also for associated adjoint models to be derived automatically via high level algorithmic differentiation. In this article these principles are extended to enable the calculation of higher order derivative information. The higher order derivative information is computed through the automated derivation of tangent-linear equations, which are then treated as new forward equations, and from which higher order tangent-linear and adjoint information can be derived. The principal emphasis is on the calculation of partial differential equation constrained Hessian actions, but the approach generalises for derivative information at arbitrary order. The derivative calculations are further combined with an advanced data checkpointing strategy. Applications which make use of partial differential equation constrained Hessian actions are presented.

Key word. FEniCS; tangent-linear; adjoint; second order adjoint; code generation

AMS subject classifications. 49M29, 65M32, 65M60, 68N20

1. Introduction.

In principle a numerical model may be considered a single, possibly highly complex, function mapping from inputs to outputs. This function may typically be broken down into the composition of a possibly very large number of simpler functions. Source-to-source algorithmic differentiation tools,[1] in forward mode, differentiate individual lines of source code appearing in a forward code, and use this to generate associated tangent-linear models (e.g. [6]). A tangent-linear model calculates the derivative of forward model outputs with respect to an input by propagating derivative information forwards, from the input, through the tangent-linear calculation.

An adjoint model instead calculates the derivative of a forward model output with respect to forward equation residuals by propagating information in a reverse sense, from the output, through an adjoint calculation. If a forward variable is computed earlier in the originating forward model, an associated adjoint variable is computed later in an associated adjoint calculation. Source-to-source algorithmic differentiation tools in reverse mode (e.g. [18, 19, 53, 32]) must tackle the additional complexity associated with this reversal of causality. An adjoint model associated with a non-linear forward model, or an adjoint-based calculation of the linear sensitivity of a functional with respect to a control on which the forward depends non-linearly, requires forward solution data. Practical implementations of adjoint models associated with non-linear forward problems must therefore additionally manage the storage, checkpointing, or recalculation of required forward model data (e.g. [28, 48]).

This article describes the calculation of higher order partial differential equation constrained derivative information, through the derivation of higher order tangent-linear equations, and the solution of associated adjoint equations. The principal emphasis is on the calculation of second derivative information, although the metholodology generalises to arbitrary order.

For details on higher order algorithmic differentiation see for example chapter 3 of [45]. See also, for example, [7] and chapter 13 of [30] for Taylor polynomial based methods for computing higher order derivative information.

*School of Mathematics and Maxwell Institute for Mathematical Sciences, The University of Edinburgh, Edinburgh, EH9 3FD, United Kingdom (j.r.maddison@ed.ac.uk).

†School of GeoSciences, The University of Edinburgh, EH8 9XP, United Kingdom.

‡School of Mathematics and Maxwell Institute for Mathematical Sciences, The University of Edinburgh, Edinburgh, EH9 3FD, United Kingdom.

[1] Also commonly referred to in this context as "automatic differentiation" – here, as in [30], the term "algorithmic differentiation" is adopted.

1.1. High level algorithmic differentation. In [16] discrete adjoint models are derived automatically for finite element models written using the FEniCS system, by raising the level at which the forward problem is considered to the level of finite element discretised weak form partial differential equations. This is implemented in the dolfin-adjoint library. The methodology used to derive higher order partial differential equation constrained derivative information, described in this article, is based upon this high level approach.

The key ingredients of the approach are

1. the automatic processing of symbolic representations of discretisised partial differential equations, so as to construct symbolic representations of associated tangent-linear and adjoint information,
2. the implementation of the symbolic representations as lower level code using automated code generation.

Discrete tangent-linear and adjoint models may then be derived and implemented automatically by tackling the problem at the level of discrete equations. In [16] this methodology is applied for finite element models written using the FEniCS automated code generation system [40, 1].

1.2. Escape hatches. A potential shortcoming of the approach described in [16] is that it relies heavily upon the ability to construct appropriate symbolic representations of forward equations. The dolfin-adjoint library specifically processes discretised weak form partial differential equations which are expessed using the Unified Form Language (UFL, [2]). However cases may be encountered that lack such a representation – for example elementary linear algebra operations, or the evaluation of a continuous function at a point. For cases where calculations cannot easily or efficiently be represented as the solution of discretised weak form partial differential equations, escape hatches are required to enable one to supply the relevant derivative information manually. In the version of the dolfin-adjoint library described in [16] this required manual interaction with the lower level libadjoint library [15] underlying dolfin-adjoint. In more recent versions, making use of pyadjoint [43], this can be achieved through the definition of custom `Block` classes.

1.3. Storage and checkpointing. An associated tangent-linear model depends[2] upon forward solution data, but shares the causal structure of the forward code. Hence a tangent-linear model can be solved alongside its associated forward. By contrast, a key difficulty associated with the practical implementation of an adjoint model is that the adjoint model also depends upon forward solution data, but has reverse causal structure to the forward code. Hence while in a forward calculation it may be possible to discard x^n after solving for x^{n+1}, the associated adjoint calculation requires these data to be retained, for example in memory or on disk, or else regenerated through additional forward calculations. More advanced approaches can strategically combine storage with recalculation.

If a specific number of sub-problems are solved and known prior to the forward calculation (e.g. if a known number of timesteps are to be taken) then the approach of [29] (see also [28, 38]) provides (subject to some assumptions) an optimal strategy for the checkpointing and possible recalculation of forward model data. Alternative algorithms can be applied for the case where the number of timesteps is determined dynamically at runtime [35, 54, 51], although such approaches are not considered here – that is, only "offline" strategies are considered.

In [16] the high level algorithmic differentiation approach is combined with the approach of [29], implemented in the revolve library, to yield an optimal data checkpointing strategy for all models which have the required causal structure, and which are written using the FEniCS automated code generation system in a way which is compatible with the dolfin-adjoint library.

[2]Specific limiting cases – such as a fully linear calculation – may have simpler dependency structures than the more general cases considered here.

1.4. Higher order derivative information. Tangent-linear and adjoint models can compute first order partial differential equation constrained derivatives. A partial differential equation constrained second derivative, contracted[3] against a single direction, can be evaluated via the solution of

1. the original forward equations,
2. a set of tangent-linear equations associated with the forward equations,
3. a set of first order adjoint equations,
4. a set of second order adjoint equations [55].

The complexities associated with the derivation of tangent-linear and adjoint models are now compounded. If using algorithmic differentiation, the algorithmic differentiation tool must be capable of processing its own output, so as to generate an associated adjoint model from a tangent-linear model, or to generate an associated tangent-linear model from an adjoint model. Any inefficiencies in the algorithmic differentiation tool are similarly compounded. The data storage problem now becomes more complex. Forward and tangent-linear models have forward causality, while the adjoint and second order adjoint have reverse causality. The tangent-linear and first order adjoint solutions depend upon the forward solution, while the second order adjoint solution depends upon the forward, tangent-linear, and first order adjoint solutions.

While dolfin-adjoint includes functionality for computing second order derivative information through the solution of a second order adjoint, these calculations have not yet been combined with data checkpointing strategies, and have not been generalised beyond second order.

The key step taken in this article is to add to the methodology of [16] the ability of the high level algorithmic differentiation tool to process its own output, through the generation of tangent-linear equations which are treated on an equal footing with their associated forward equations. Tangent-linear and adjoint information is derived for the forward equations and, as the tangent-linear equations are now treated simply as further equations, higher order tangent-linear equations, and adjoint information associated with the tangent-linear equations, can be derived. This allows tangent-linear and adjoint information to be derived to arbitrary order, and for data checkpointing strategies to be used for higher order adjoint calculations.

1.5. Feature summary. This article describes the derivation of arbitrary order partial differential equation constrained derivative information for finite element models written using the FEniCS system. The high level algorithmic differentiation is implemented in the `tlm_adjoint` library.

`tlm_adjoint` is based around an abstract interface for the specification of model equations, following several of the key design principles of the libadjoint library, which itself underlies the version of the dolfin-adjoint library described in [16].[4] As such `tlm_adjoint` shares many of the key benefits of dolfin-adjoint, including

- the ability to re-use the automated code generation system FEniCS itself to generate low-level implementations, derived from higher level symbolic representations, of tangent-linear and adjoint calculations associated with finite element discretisations of partial differential equations,
- MPI parallelism support, principally inherited from the MPI parallelism support of the FEniCS system,
- the ability for specific tangent-linear and adjoint equations associated with non-linear problems to be solved with a single linear solve (see [16], section 6.1),
- automated management of storage and checkpointing, including use of the binomial

[3] Here "contraction" is understood in terms of tensor contractions against the vectors defining the directions.

[4] Note that more recent versions of dolfin-adjoint no longer make use of libadjoint, and are instead based on the pyadjoint library [43].

checkpointing approach of [29], with offline multi-stage checkpointing [50].
`tlm_adjoint` further

- manages the automated derivation of tangent-linear equations to arbitrary order,
- manages higher order derivative calculations, through the solution of arbitrary order adjoint equations,
- manages storage and checkpointing associated with solving these arbitrary order adjoint equations,
- implements the method of [8] and its tangent-linear analogue [21] for the solution of tangent-linear and adjoint equations, of arbitrary order, associated with fixed-point iteration,
- provides a simple "escape hatch" interface, enabling custom equations to be specified,
- implements several automated assembly and solver caching optimisations, similar to those described in [42].

The article proceeds as follows. In section 2 the calculation of higher order partial differential equation constrained derivative information is outlined. The automated derivation of higher order tangent-linear equations and associated adjoint information using the `tlm_adjoint` library is detailed in section 3. Two examples which make use of forward model constrained Hessian information are provided in section 4. Limitations of the approach are discussed in section 5. The paper concludes in section 6.

2. Formulation. This section outlines the calculation of higher order forward model constrained derivative information. The formulation is limited to the consideration of forward models with specific causal structure, such as is encountered in a timestepping solver for discretised time-dependent partial differential equations.

In the following vectors $v \in \mathbb{R}^d$ for some positive integer d are considered to be column vectors. The derivative of a functional $J(v) : \mathbb{R}^d \to \mathbb{R}$ with respect to v is considered to be a row vector with elements

$$\left(\frac{\mathrm{d}J}{\mathrm{d}v}\right)_i = \frac{\partial J}{\partial v_i}, \tag{2.1}$$

where v_i indicates the ith element of v. The derivative of a vector-valued function $F(v) : \mathbb{R}^d \to \mathbb{R}^d$ with respect to v is considered to be a matrix with elements

$$\left(\frac{\mathrm{d}F}{\mathrm{d}v}\right)_{i,j} = \frac{\partial F_i}{\partial v_j}, \tag{2.2}$$

where $F_i(v)$ is the ith element of $F(v)$. Sufficient regularity is assumed throughout.

2.1. Timestepping forward model. Consider a forward variable $x \in \mathbb{R}^{N_x}$ and control parameter $m \in \mathbb{R}^{N_m}$. A residual function $F(x, m) : \mathbb{R}^{N_x} \times \mathbb{R}^{N_m} \to \mathbb{R}^{N_x}$ defines the forward solution as an implicit function of the control parameter, $\hat{x}(m) : \mathcal{M} \to \mathbb{R}^{N_x}$, via

$$F(\hat{x}(m), m) = 0 \quad \forall m \in \mathcal{M}, \tag{2.3}$$

where existence of such an $\hat{x}$ is assumed, and where $\mathcal{M}$ is some appropriate subset of $\mathbb{R}^{N_m}$.

Let the forward variable x be divided into a set of $(N+2)$ blocks $x_n \in \mathbb{R}^{N_{x,n}}$ for $n \in \{0, \dots, N+1\}$, e.g. for $N \geq 1$

$$x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N+1} \end{pmatrix}. \tag{2.4}$$

For simplicity a specific causal structure in the forward model is now assumed, by asserting that the residual function F can be divided into a series of $(N+2)$ blocks F_n which have the form (for $N \geq 1$)

$$F(x,m) = \begin{pmatrix} F_0(x_0,m) \\ F_1(x_0,x_1,m) \\ \vdots \\ F_{N+1}(x_N,x_{N+1},m) \end{pmatrix}. \tag{2.5}$$

That is F_n depends explicitly only on m, x_n, and (for $n \geq 1$) x_{n-1}. Each F_n has codomain $\mathbb{R}^{N_{x,n}}$. For example in a timestepping model F_0 may define the forward model initialisation, F_n for $n \in \{1,\ldots,N\}$ may define N timesteps, and F_{N+1} may define the calculation of final diagnostics (the "initialisation", "timestepping", and "finalisation" stages described in [42]).

2.2. Functional. Given a functional $J(x,m) : \mathbb{R}^{N_x} \times \mathbb{R}^{N_m} \to \mathbb{R}$, a functional depending only upon the control parameter m, $\hat{J}(m) : \mathcal{M} \to \mathbb{R}$, is defined via

$$\hat{J}(m) = J(\hat{x}(m), m) \quad \forall m \in \mathcal{M}. \tag{2.6}$$

Given a value of m, we seek to compute the contraction of the Kth derivative of $\hat{J}$ against K directions $\zeta_i \in \mathbb{R}^{N_m}$ for $i \in \{1,\ldots,K\}$. This can be defined inductively via

$$D_1 = \frac{\mathrm{d}\hat{J}}{\mathrm{d}m}\zeta_1, \tag{2.7a}$$

$$D_k = \frac{\mathrm{d}D_{k-1}}{\mathrm{d}m}\zeta_k \quad \text{for } k \in \{2,\ldots,K\}. \tag{2.7b}$$

Given a value of m, we further seek to compute the contraction of the Kth derivative of $\hat{J}$ against $(K-1)$ directions, which is given by

$$S_K = \frac{\mathrm{d}D_{K-1}}{\mathrm{d}m}. \tag{2.8}$$

The formulation to follow is simplified by asserting that J is equal to a single component of x, and in particular equal to a single component of x_{N+1}. Specifically it is set equal to the Mth component, $x_{N+1,M}$, of x_{N+1},

$$J(x,m) = x_{N+1,M}. \tag{2.9}$$

More complex functionals may be defined by appending additional variables to x, and additional residuals to F. For example, if the functional of interest is a sum over timesteps, then x may include appropriate partial sums. J is then equal to the final partial sum, defined to be an element of x_{N+1}.

2.3. First order derivative, contracted against zero directions. The first order forward model constrained derivative of the functional can be computed using a first order adjoint model (e.g. [31], equation (2.33)),

$$\frac{\partial F_{N+1}}{\partial x_{N+1}}^T \lambda_{1,N+1} = e_{1,N+1}, \tag{2.10a}$$

$$\frac{\partial F_n}{\partial x_n}^T \lambda_{1,n} = -\frac{\partial F_{n+1}}{\partial x_n}^T \lambda_{1,n+1} \quad \forall n \in \{0,\ldots,N\}. \tag{2.10b}$$

Here each $\lambda_{1,n} \in \mathbb{R}^{N_{x,n}}$, and $e_{1,N+1} \in \mathbb{R}^{N_{x,n}}$ is a vector with Mth element equal to one and all other elements equal to zero. These first order adjoint equations can be solved via block backward substitution, consistent with the reverse causal nature of the first order adjoint. The first order forward model constrained derivative (contracted against zero directions) is (e.g. [31], equation (2.34))

$$S_1 = \frac{\mathrm{d}\hat{J}}{\mathrm{d}m} = -\sum_{n=0}^{N+1} \lambda_{1,n}^T \frac{\partial F_n}{\partial m}. \tag{2.11}$$

2.4. First order derivative, contracted against one direction. The first order forward model constrained derivative of the functional, contracted against a single direction, can be computed using a first order tangent-linear model (e.g. [31], equation (2.24)),

$$\frac{\partial F_0}{\partial x_0}\tau_{1,0} = -\frac{\partial F_0}{\partial m}\zeta_1, \tag{2.12a}$$

$$\frac{\partial F_n}{\partial x_n}\tau_{1,n} = -\frac{\partial F_n}{\partial m}\zeta_1 - \frac{\partial F_{n-1}}{\partial x_{n-1}}\tau_{1,n-1} \quad \forall n \in \{1, \ldots, N+1\}. \tag{2.12b}$$

Here each $\tau_{1,n} \in \mathbb{R}^{N_{x,n}}$. These first order tangent-linear equations can be solved via block forward substitution, consistent with the forward causal nature of the first order tangent-linear. The forward model constrained derivative, contracted against a single direction ζ_1, is (e.g. [31], equation (2.21))

$$D_1 = \frac{\mathrm{d}\hat{J}}{\mathrm{d}m}\zeta_1 = \tau_{1,N+1}^T e_{1,N+1}. \tag{2.13}$$

Since both the forward and the first order tangent-linear share a forward causal structure, they can be combined into a single model. Define $X_{1,n} \in \mathbb{R}^{2N_{x,n}}$ with block structure

$$X_{1,n} = \begin{pmatrix} x_n \\ \tau_{1,n} \end{pmatrix} \quad \forall n \in \{0, \ldots, N+1\}, \tag{2.14}$$

and new residual functions $\mathcal{F}_{1,n}$, depending only upon m, $X_{1,n}$, and (for $n \geq 1$) $X_{1,n-1}$, with block structure

$$\mathcal{F}_{1,0}\left(X_{1,0}, m\right) = \begin{pmatrix} F_0\left(x_0, m\right) \\ \frac{\partial F_0}{\partial x_0}\tau_{1,0} + \frac{\partial F_0}{\partial m}\zeta_1 \end{pmatrix}, \tag{2.15a}$$

$$\mathcal{F}_{1,n}\left(X_{1,n-1}, X_{1,n}, m\right) = \begin{pmatrix} F_n\left(x_{n-1}, x_n, m\right) \\ \frac{\partial F_n}{\partial x_n}\tau_{1,n} + \frac{\partial F_n}{\partial m}\zeta_1 + \frac{\partial F_{n-1}}{\partial x_{n-1}}\tau_{1,n-1} \end{pmatrix} \forall n \in \{1, \ldots, N+1\}. \tag{2.15b}$$

The $\mathcal{F}_{1,n}$ define the forward and first order tangent-linear solutions as an implicit function of the control parameter, $\hat{X}_{1,n}\left(m\right) : \mathcal{M} \to \mathbb{R}^{2N_{x,n}}$, via

$$\mathcal{F}_{1,0}\left(\hat{X}_{1,0}\left(m\right), m\right) = 0 \quad \forall m \in \mathcal{M}, \tag{2.16a}$$

$$\mathcal{F}_{1,n}\left(\hat{X}_{1,n-1}\left(m\right), \hat{X}_{1,n}\left(m\right), m\right) = 0 \quad \forall m \in \mathcal{M}, n \in \{1, \ldots, N+1\}. \tag{2.16b}$$

The new combined model shares the causal structure of the originating forward model.

The forward model constrained derivative, contracted against a single direction ζ_1, can now be expressed

$$D_1 = \frac{\mathrm{d}\hat{J}}{\mathrm{d}m}\zeta_1 = \hat{X}_{1,N+1}^T e_{2,N+1}, \tag{2.17}$$

where $e_{2,N+1} \in \mathbb{R}^{2N_x}$ has block structure

$$e_{2,N+1} = \begin{pmatrix} z_{1,N+1} \\ e_{1,N+1} \end{pmatrix}, \tag{2.18}$$

where $z_{1,N+1}$ is a zero vector of length $N_{x,N+1}$.

2.5. Kth order derivative, contracted against K directions. The procedure considered in the preceding subsection can be now applied inductively to any order $K \geq 2$. Consider, for $K \geq 2$,

$$\frac{\partial \mathcal{F}_{K-1,0}}{\partial X_{K-1,0}} \tau_{K,0} = -\frac{\partial \mathcal{F}_{K-1,0}}{\partial m} \zeta_K, \tag{2.19a}$$

$$\frac{\partial \mathcal{F}_{K-1,n}}{\partial X_{K-1,n}} \tau_{K,n} = -\frac{\partial \mathcal{F}_{K-1,n}}{\partial m} \zeta_K - \frac{\partial \mathcal{F}_{K-1,n-1}}{\partial X_{K-1,n-1}} \tau_{K,n-1} \quad \forall n \in \{1, \ldots, N+1\}. \tag{2.19b}$$

Here $\tau_{K,n} \in \mathbb{R}^{2^{K-1} N_{x,n}}$. For $K = 2$ the $\mathcal{F}_{K-1,n}$ and $X_{K-1,n}$ are as defined in the preceding subsection, and otherwise they are defined inductively below. These equations can be solved via block forward substitution, and hence are of forward causal nature.

Define $X_{K,n} \in \mathbb{R}^{2^K N_{x,n}}$, with block structure

$$X_{K,n} = \begin{pmatrix} X_{K-1,n} \\ \tau_{K,n} \end{pmatrix} \quad \forall n \in \{0, \ldots, N+1\}. \tag{2.20}$$

Define new functions $\mathcal{F}_{K,n}$, depending only upon m, $X_{K,n}$, and (for $n \geq 1$) $X_{K,n-1}$, with block structure

$$\mathcal{F}_{K,0}(X_{K,0}, m) = \begin{pmatrix} \mathcal{F}_{K-1,0}(X_{K-1,0}, m) \\ \frac{\partial \mathcal{F}_{K-1,0}}{\partial X_{K-1,0}} \tau_{K,0} + \frac{\partial \mathcal{F}_{K-1,0}}{\partial m} \zeta_K \end{pmatrix}, \tag{2.21a}$$

$$\mathcal{F}_{K,n}(X_{K,n-1}, X_{K,n}, m) = \begin{pmatrix} \mathcal{F}_{K-1,n}(X_{K-1,n-1}, X_{K-1,n}, m) \\ \frac{\partial \mathcal{F}_{K-1,n}}{\partial X_{K-1,n}} \tau_{K,n} + \frac{\partial \mathcal{F}_{K-1,n}}{\partial m} \zeta_K + \frac{\partial \mathcal{F}_{K-1,n-1}}{\partial X_{K-1,n-1}} \tau_{K,n-1} \end{pmatrix} \quad \forall n \in \{1, \ldots, N+1\}. \tag{2.21b}$$

The $\mathcal{F}_{K,n}$ define the solution to the forward and all tangent-linears up to and including order K as an implicit function of the control parameter, $\hat{X}_{K,n}(m) : \mathcal{M} \to \mathbb{R}^{2^K N_{x,n}}$, via

$$\mathcal{F}_{K,0}\left(\hat{X}_{K,0}(m), m\right) = 0 \quad \forall m \in \mathcal{M}, \tag{2.22a}$$

$$\mathcal{F}_{K,n}\left(\hat{X}_{K,n-1}(m), \hat{X}_{K,n}(m), m\right) = 0 \quad \forall m \in \mathcal{M}, n \in \{1, \ldots, N+1\}. \tag{2.22b}$$

The new combined model still shares the causal structure of the originating forward model.

The Kth order forward model constrained derivative, contracted against K directions $\zeta_k \in \mathbb{R}^{N_m}$ for $k \in \{1, \ldots, K\}$, can now be expressed as

$$D_K = \hat{X}_{K,N+1}^T e_{K+1,N+1}, \tag{2.23}$$

where $e_{K+1,N+1} \in \mathbb{R}^{2^K N_x}$ has block structure

$$e_{K+1,N+1} = \begin{pmatrix} z_{K,N+1} \\ e_{K,N+1} \end{pmatrix}, \tag{2.24}$$

where $z_{K,N+1}$ is a zero vector of length $2^{K-1} N_{x,N+1}$.

If two or more directions ζ_i are equal then there is some redundancy in the above, with identical tangent-linear equations defined. These redundant equations can be removed to define a (perhaps significantly) smaller $X_{K,n}$.

2.6. Kth order derivative, contracted against $(K-1)$ directions. Consider, for $K \geq 2$, the adjoint equations

$$\frac{\partial \mathcal{F}_{K-1,N+1}}{\partial X_{K-1,N+1}}^T \lambda_{K,N+1} = e_{K,N+1}, \tag{2.25a}$$

$$\frac{\partial \mathcal{F}_{K-1,n}}{\partial X_{K-1,n}}^T \lambda_{K,n} = -\frac{\partial \mathcal{F}_{K-1,n+1}}{\partial X_{K-1,n}}^T \lambda_{K,n+1} \quad \forall n \in \{0, \dots, N\}, \tag{2.25b}$$

where each $\lambda_{K,n} \in \mathbb{R}^{2^{K-1} N_{x,n}}$. These combine the solution of adjoint equations of order up to and including order K. Since they can be solved via block backward substitution, they are of reverse causal nature.

The Kth order forward model constrained derivative, contracted against $(K-1)$ directions, is

$$S_K = -\sum_{n=0}^{N+1} \lambda_{K,n}^T \frac{\partial \mathcal{F}_{K-1,n}}{\partial m}. \tag{2.26}$$

Note that, for $K = 2$, the above approach forms adjoint equations associated with the forward and first order tangent-linear equations. This contrasts with the generation of tangent-linear equations associated with first order adjoint equations, for example as described in [20, 37]. See chapter 3 of [45] for a relevant discussion.

3. Implementation. The procedure described in the preceding section requires

1. the definition of the forward equations,
2. the derivation of tangent-linear equations associated with forward equations,
3. the derivation of tangent-linear equations associated with tangent-linear equations,
4. the derivation of adjoint equations associated with forward equations,
5. the derivation of adjoint equations associated with tangent-linear equations.

The implementation is simplified by treating forward and tangent-linear equations on an equal footing so that, given a forward equation, associated tangent-linear equations can be derived and then treated as new forward equations. Given the ability to derive tangent-linear equations and adjoint information associated with forward equations, one can then derive adjoint information associated with tangent-linear equations, to arbitrary order.

Specifically an abstraction of a general equation is considered which defines

1. how the forward equation can be solved,
2. how required adjoint information can be computed,
3. how a new tangent-linear equation can be derived.

The first two parts of this definition are consistent with the approach used by the libadjoint library which underlies the version of dolfin-adjoint described in [16]. The key new ingredient is the third, which provides the ability to derive tangent-linear equations, with the tangent-linear equations represented using the same abstraction as the forward equations.

3.1. Representation of forward equations. `tlm_adjoint` is a Python 3 library implementing the principles of dolfin-adjoint as described in [16], and following some of the design principles of the libadjoint library [15], but extending these with the ability to derive tangent-linear equations to arbitrary order. The library was derived out of a custom escape hatch extension to dolfin-adjoint which interfaced directly with libadjoint for the specification of custom equations, but now functions as a standalone library.

The key elements required in the definition of equations are specified in the abstract base class `Equation`. The principles are outlined here via a simple example, considering the forward

equation

$$F(x, y) = x - \alpha y = 0, \tag{3.1}$$

solving for x given y for some $\alpha \in \mathbb{R}$, and where x and y are compatible length vectors. This can be implemented using `tlm_adjoint` via

```
class ScaleSolver(Equation):
  def __init__(self, alpha, y, x):
    Equation.__init__(self, x, [x, y], nl_deps = [], ic_deps = [])
    self.alpha = alpha

  def forward_solve(self, x, deps = None):
    if deps is None:
      y = self.dependencies()[1]
    else:
      y = deps[1]
    function_set_values(x, self.alpha * function_get_values(y))

  def adjoint_jacobian_solve(self, nl_deps, b):
    return b

  def adjoint_derivative_action(self, nl_deps, dep_index, adj_x):
    return ([1.0, -self.alpha][dep_index], adj_x)

  def tangent_linear(self, M, dM, tlm_map):
    x = self.dependencies()[0]
    y = self.dependencies()[1]
    if y in M:
      return ScaleSolver(self.alpha, dM[M.index(y)], tlm_map[x])
    elif function_is_static(y):
      return NullSolver(tlm_map[x])
    else:
      return ScaleSolver(self.alpha, tlm_map[y], tlm_map[x])
```

3.1.1. Definition of dependencies. The constructor calls the base `Equation` class constructor, and specifies that this is an equation solving for `x`, with `x` and `y` as dependencies. Tangent-linear and adjoint equations depend only upon non-linear dependencies of the forward, and these are defined via `nl_deps` – in this linear example there are no non-linear dependencies. If the solution of the equation depends upon the initial value of `x` (for example if it is used as an initial guess for an iterative solver) then this can be specified using the `ic_deps` argument – this information is required for rerunning of the forward.

3.1.2. Forward solution. The method `forward_solve` implements a means of solving the forward equation, solving for `x`. If provided, the input `deps` defines the values of forward equation dependencies, and otherwise these values are defined by `self.dependencies()`. During an adjoint calculation `forward_solve` may be called, perhaps multiple times, in order to regenerate forward solution data from checkpoint data.

3.1.3. Adjoint derivative information. The overridden method `adjoint_derivative_action` computes actions of the adjoint of the derivative of F, with `dep_index` specifying the dependency with respect to which the derivative is taken. For example if `dep_index` equals 1 this computes

$$\frac{\partial F}{\partial y}^T \lambda, \tag{3.2}$$

where λ is defined by `adj_x`. The values of any non-linear dependencies are provided in `nl_deps`.

3.1.4. Solution of adjoint equations. The overridden method `adjoint_jacobian_solve` returns λ where

$$\frac{\partial F}{\partial x}^T \lambda = b, \tag{3.3}$$

and b is defined by `b`. Again the values of non-linear dependencies are provided in `nl_deps`.

3.1.5. Derivation of tangent-linear equations. Tangent-linear information is specified by the `tangent_linear` method, which returns a new `Equation` object suitable for the solution of a tangent-linear equation.

The argument `M` provided to `tangent_linear` defines the control parameter m, and the argument `dM` defines a direction ζ_i. The method may return a new `Equation` object, which in this example solves

$$\frac{\partial F}{\partial x}\tau_x = -\frac{\partial F}{\partial m}\zeta_i, \tag{3.4}$$

for τ_x if m corresponds to y itself, and

$$\frac{\partial F}{\partial x}\tau_x = -\frac{\partial F}{\partial y}\tau_y, \tag{3.5}$$

for τ_x if y is distinct from m. The values of associated tangent-linear variables τ_x and τ_y are stored in the dictionary-like container object `tlm_map`. Note that if y is "static" (see section 3.4) and distinct from m, then it is known that $\tau_y = 0$.

Crucially, since the result returned by the `tangent_linear` method is itself an `Equation` object, adjoint information, and higher order tangent-linear information, can now be derived.

3.2. Processing of equations. By default, when the `solve` method of an `Equation` object is called, the equation is processed by an internal manager. During forward calculations this manager keeps a record of the equations solved, derives tangent-linear equations, and manages the storage and checkpointing of forward model data (see section 3.5).

`tlm_adjoint` includes limited functionality for the overriding or interception of FEniCS functions and methods, and the subsequent automated construction and solution of appropriate `Equation` objects. This automated functionality is less extensive than similar funtionality provided by dolfin-adjoint.

The division of the forward model solution and the forward model residual into logical blocks, as described in section 2.1, is indicated by calling the `new_block()` function at the desired point in the code – for example this may typically be called at the end of forward timesteps.

3.3. Finite element discretisations. Finite element discretised partial differential equations are represented using the `EquationSolver` class, which derives from the abstract base class `Equation`, and provides implementations of each of the `adjoint_derivative_action`, `adjoint_jacobian_solve`, and `tangent_linear` methods. Here this is illustrated using a simple example, where the forward model consists of the Poisson equation in the unit square domain subject to homogeneous Dirichlet boundary conditions.

3.3.1. Second order adjoint calculation. A complete code which computes a forward model constrained Hessian action, integrating with FEnicS 2018.1.0, takes the form

```
from fenics import *
from tlm_adjoint import *

mesh = UnitSquareMesh(10, 10)
space = FunctionSpace(mesh, "Lagrange", 1)
test = TestFunction(space)
trial = TrialFunction(space)

F = Function(space, name = "F", static = True)
F.interpolate(Expression("sin(pi * x[0]) * sin(pi * x[1])", degree = 1))

zeta = Function(space, name = "zeta", static = True)
zeta.assign(Constant(1.0))
```

```
add_tlm(F, zeta)

Psi = Function(space, name = "Psi")
eq = EquationSolver(inner(grad(test), grad(trial)) * dx == -inner(test, F) * dx, Psi,
  DirichletBC(space, 0.0, "on_boundary", static = True, homogeneous = True))
eq.solve()

J = Functional()
J.assign(inner(Psi, Psi) * dx)

stop_manager()
ddJ = compute_gradient(J.tlm(F, zeta), F)
```

Prior to the solving of forward equations the automated derivation of tangent-linear equations is requested via

```
zeta = Function(space, name = "zeta", static = True)
zeta.assign(Constant(1.0))
add_tlm(F, zeta)
```

which requests the automated derivation and solution of tangent-linear equations associated with derivatives with respect to the control parameter represented by `F` in the direction represented by `zeta`. When forward equations are processed by the internal manager, associated tangent-linear equations are derived, solved, and themselves processed by the internal manager. The `EquationSolver.tangent_linear` method generates new `EquationSolver` objects associated with finite element discretised tangent-linear equations as required.

The finite element discretised equation is defined by constructing an `EquationSolver`, and calling its `solve` method. Internally this calls the `forward_solve` method associated with the equation, and further ensures that the equation is processed by the internal equation manager.

The functional is intitialised and evaluated via

```
J = Functional()
J.assign(inner(Psi, Psi) * dx)
```

Note that internally `tlm_adjoint` treats this latter assignment as a new equation, which is processed by the internal equation manager. Further terms may be added to a functional, for example representing the sum of terms over different timesteps in a time-dependent calculation, using the `Functional.addto` method, which is again internally treated as new equations which are processed by the internal equation manager.

After conclusion of the forward calculation higher order derivative information is computed via

```
ddJ = compute_gradient(J.tlm(F, zeta), F)
```

3.3.2. Higher order adjoint calculations. Higher order derivative information can be computed via the addition of multiple tangent-linear models. For example

```
zeta_1 = Function(space, name = "zeta_1", static = True)
zeta_1.assign(Constant(1.0))
zeta_2 = Function(space, name = "zeta_2", static = True)
zeta_2.interpolate(Expression("x[0]", degree = 1))
add_tlm(F, zeta_2)
add_tlm(F, zeta_1)
```

After the first `add_tlm` call, `tlm_adjoint` derives and solves tangent-linear equations – in this case tangent-linear equations associated with derivatives with respect to the function represented by `F` in the direction represented by `zeta_2`. After the second `add_tlm` call, `tlm_adjoint` derives and solves further tangent-linear equations – tangent-linear equations associated with derivatives with respect to the function represented by `F` in the direction represented by `zeta_1`. Crucially in this second case this is applied to *both* the forward equations *and* the tangent-linear equations requested through the first `add_tlm` call – that is, second order tangent-linear equations are derived.

After conclusion of the forward model a third order adjoint calculation can be performed via

```
dddJ = compute_gradient(J.tlm(F, zeta_2).tlm(F, zeta_1), F)
```

This principle generalises to arbitrary order.

The case where `zeta_1` and `zeta_2` correspond to equal directions (see e.g. [30], chapter 13) can be handled by replacing the two `add_tlm` calls with

```
add_tlm(F, zeta_1, max_depth = 2)
```

This usage avoids the redundant solution of identical tangent-linear equations.

3.4. Time loop optimisation. In [42] finite element models for time-dependent problems were optimised by exploiting the availability of information about the model time discretisation. The `EquationSolver` class in `tlm_adjoint` can apply a number of such optimisations automatically. This is facilitated by the appropriate declaration of "static" data, that are known to be fixed for the duration of the model (e.g. as for the function represented by `F` in the preceding example). As described in [42] this allows the equation to be defined using the high-level syntax provided by the Unified Form Language, while allowing static data to be identified and cached automatically and without manual intervention.

Optimisations applied automatically by explicitly constructed `EquationSolver` objects include the following. For linear equations for which the associated left-hand-side matrix is static, the matrix and associated linear solver data are cached. A right-hand-side of a linear equation is broken into the sum of static terms, terms which can be represented as the action of a static matrix, and remaining non-static terms, with the relevant static data cached. For non-linear forward equations no such caching is applied.

Further optimisations are applied in the calculation of adjoint data. In the solution of adjoint equations, if the associated adjoint Jacobian matrix is static, the matrix and associated linear solver data are cached. If an adjoint derivative action can be represented as the action of a static matrix, then the relevant matrix is cached. For non-static cases steps are taken to reduce costs associated with the symbolic manipulation of UFL expressions.

Since tangent-linear equations derived from `EquationSolver` objects are themselves `EquationSolver` objects, caching can be applied automatically in the solution of associated tangent-linear equations, as well as in the calculation of adjoint information associated with these tangent-linear equations, to arbitrary order.

3.5. Storage and checkpointing. The binomial checkpointing strategy of [29] may be applied automatically in adjoint calculations with `tlm_adjoint`. The only required modification is the specification of configuration information prior to the solution of forward equations, for example

```
configure_checkpointing("multistage", {"blocks":100,
                                       "snaps_on_disk":2,
                                       "snaps_in_ram":3})
```

Here this configures offline multi-stage checkpointing [50], storing up to 2 checkpoints on disk and up to 3 checkpoints in memory. The maximum permitted step size when determining the placement of a subsequent checkpoint is used (following the maximum permitted path in Fig. 4 of [29]). Multi-stage checkpointing as described in [50] is implemented through a brute force evaluation of costs (with reads and writes given equal weight).

Data corresponding to the degrees of freedom of discrete functions are stored in a checkpoint. All data associated with discrete function spaces are fully stored in memory. By default disk checkpoints are stored using the HDF5 library [52] using h5py (https://www.h5py.org/), with MPI parallelisation achieved using the MPI functionality supplied with h5py.

The version of dolfin-adjoint described in [16] also supports checkpointing using the approach of [29], and the syntax for the configuration of the checkpointing strategy described here mirrors

the syntax used in the configuration of checkpointing in dolfin-adjoint. However, and crucially, since here tangent-linear equations are treated on an equal footing to forward equations, this allows `tlm_adjoint` to apply offline multi-stage checkpointing in higher order adjoint calculations.

Note that it is not assumed that each forward model block consists of precisely the same set of equations. As forward equations are processed dynamically at runtime it is not known what data consitute a checkpoint at the point in the code execution at which a checkpoint should be stored. This is resolved in `tlm_adjoint` by deferring the storage of data associated with a checkpoint until all equations depending on data to be stored in the checkpoint have been solved.

4. Examples.

4.1. Optimality constrained derivatives. Forward model constrained Hessian information can be applied to compute higher order constrained derivatives. Consider, for example, the introduction of a second parameter $p \in \mathbb{R}^{N_p}$. One can seek to compute the derivative of a functional K with respect to p, subject to the constraint that the forward model constrained derivative of a (possibly different) second functional J with respect to the control parameter m is zero [39, 11]. p may, for example, represent input data used in the optimisation procedure.

4.1.1. Formulation. The details of the calculation of what is here termed an "optimality constrained derivative" are described in [11] (see also [39, 3, 9]). Here the key steps are outlined.

The forward model residual is now considered a three argument function, $F(x, m, p) : \mathbb{R}^{N_x} \times \mathbb{R}^{N_m} \times \mathbb{R}^{N_p} \to \mathbb{R}^{N_x}$. The forward model solution is defined as an implicit function of the parameters m and p, $\hat{x} : \mathcal{M} \times \mathcal{P}_1 \to \mathbb{R}^{N_x}$, via

$$F(\hat{x}(m, p), m, p) = 0 \quad \forall m \in \mathcal{M}, p \in \mathcal{P}_1, \tag{4.1}$$

where existence of such an $\hat{x}$ is assumed, and where $\mathcal{M}$ and $\mathcal{P}_1$ are some appropriate subsets of $\mathbb{R}^{N_m}$ and $\mathbb{R}^{N_p}$ respectively. Given a functional $J(x, m, p) : \mathbb{R}^{N_x} \times \mathbb{R}^{N_m} \times \mathbb{R}^{N_p} \to \mathbb{R}$, this allows the definition of a functional depending only upon the parameters m and p, $\hat{J}(m, p) : \mathcal{M} \times \mathcal{P}_1 \to \mathbb{R}$, where

$$\hat{J}(m, p) = J(\hat{x}(m, p), m, p) \quad \forall m \in \mathcal{M}, p \in \mathcal{P}_1. \tag{4.2}$$

Consider the case where, given p, a forward model constrained optimisation problem is solved so that

$$\frac{\partial \hat{J}}{\partial m} = 0. \tag{4.3}$$

The solution of the optimisation problem allows the implicit definition of the control parameter as a function of p, $\tilde{m}(p) : \mathcal{P}_2 \to \mathcal{M}$, via

$$\left. \frac{\partial \hat{J}}{\partial m} \right|_{\tilde{m}(p), p} = 0 \quad \forall p \in \mathcal{P}_2, \tag{4.4}$$

where existence of such an $\tilde{m}$ is assumed, and where $\mathcal{P}_2$ is some appropriate subset of $\mathcal{P}_1$.

Now given a second functional $K(x, m, p)$, define

$$\hat{K}(m, p) = K(\hat{x}(m, p), m, p), \tag{4.5a}$$

$$\tilde{K}(p) = \hat{K}(\tilde{m}(p), p). \tag{4.5b}$$

Differentiating the latter yields

$$\frac{\mathrm{d}\tilde{K}}{\mathrm{d}p} = \frac{\partial \hat{K}}{\partial m} \frac{\mathrm{d}\tilde{m}}{\mathrm{d}p} + \frac{\partial \hat{K}}{\partial p}. \tag{4.6}$$

Differentiating (4.4) with respect to p and substituting leads to the result

$$\frac{\mathrm{d}\tilde{K}}{\mathrm{d}p}^T = -H_{p,m}H_{m,m}^{-1}\frac{\partial \hat{K}}{\partial m}^T + \frac{\partial \hat{K}}{\partial p}^T, \tag{4.7}$$

with

$$H_{m,m} = \frac{\partial}{\partial m}\left(\frac{\partial \hat{J}}{\partial m}^T\right), \tag{4.8a}$$

$$H_{p,m} = \frac{\partial}{\partial m}\left(\frac{\partial \hat{J}}{\partial p}^T\right). \tag{4.8b}$$

Equation (4.7), for a case where K is independent of p, is as in equation (5) of [11].

4.1.2. Motivating example. Many continuum models can be derived from microscopic dynamics through various coarse-graining techniques. Typically, this involves the use of (possibly unconstrained) approximations, the effects of which may be manifested as unknown parameters in the resulting models. Such parameters must be determined from (microscopic) numerical or physical experiments, which can be very costly. This situation becomes even worse in more complicated examples, such as colloidal dynamics modelled by extensions of Dynamical Density Functional Theory (see [23]). Here, the parameters are functions of space or time. See, e.g. [22] and references therein, where one requires knowledge of a diffusion coefficient that depends on the distance from a wall.

It is clear that there will be some uncertainty in the values of these parameters, irrespective of how they are obtained. Two natural questions arise: (i) how sensitive are the results of the forward model to changes in these parameters? (ii) in which areas of space/time is it important to have accurate values of these parameters? For (i), ideally one would like to be able to show that, within the expected uncertainty of the inputs, the output of the model is relatively stable. This is especially important when the parameters have intrinsic uncertainty, for example being derived from stochastic simulations or interpolation schemes. For (ii), the importance is related to the cost of accurately obtaining the parameters. For example, approximations to a space-dependent diffusion coefficient could be obtained from expensive microscopic simulations on small regions; one would like to know where to focus this effort to maximize accuracy and minimize cost.

4.1.3. Configuration. Consider a discretisation of the advection-diffusion equation

$$\begin{aligned}\int_\Omega \phi\frac{T_{n+1}-T_n}{\Delta t} + \int_\Omega \phi\nabla^\perp\psi\cdot\nabla\left(\frac{T_n+T_{n+1}}{2}\right)\\ + \int_\Omega \nabla\phi\cdot\kappa\nabla\left(\frac{T_n+T_{n+1}}{2}\right) = 0 \quad \forall\phi\in V_0, n\in\{0,\dots,N-1\},\end{aligned} \tag{4.9}$$

where $T_n \in V$ is the discrete solution at $t = n\Delta t$, with $N\Delta t = \tau$ and N a positive integer. Here $V = \left\{\xi \in H^1(\Omega;\mathbb{R}) : \xi - \mathcal{T}_1 \in V_0\right\}$, where $V_0 \subseteq H^1(\Omega;\mathbb{R})$ is a finite element discrete function space consisting of functions which vanish at $x = 0$. Ψ is a discrete approximation for a stream function. $\mathcal{T}_1$ is in a discrete function space, and is defined so that is takes the value T_D at $x = 0$, where T_D is a discrete approximation for a Dirichlet boundary condition applied on the $x = 0$ boundary.

For the calculations described here the solutions T_n are represented using $P1$ finite elements on a triangle mesh generated using Gmsh 3.0.6 [17] with a requested mesh size of 0.02. T_D is in a $P1$ function space on the inflow boundary at $x = 0$ – that is, T_D is a piecewise linear and

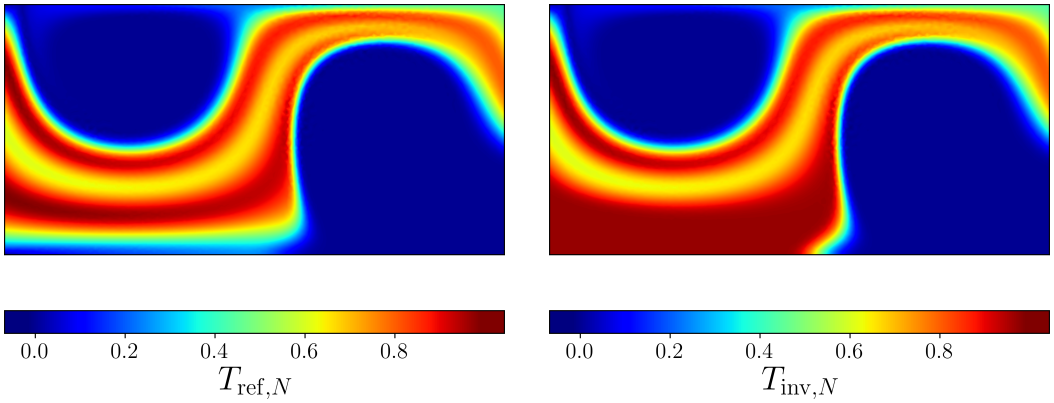

FIG. 4.1. *Left: Reference solution for the advection-diffusion model at $t = \tau = 1$. Right: Solution at $t = \tau = 1$, obtained by finding a critical point of the functional (4.14), subject to the constraint that the forward model is solved with $\kappa = 10^{-3}$.*

continuous function defined on the one-dimensional mesh associated with the inflow boundary. κ is treated as a possibly spatially varying function in a $P0$ function space – that is, it is constant within elements of the interior triangle mesh, but may have jump discontinuities at element boundaries. Ψ is defined to be a $P1$ function, defined via interpolation at the mesh vertices of

$$\Psi(x, y) = (1 - e^y) \sin(\pi x) \sin(\pi y) - y, \tag{4.10}$$

leading to an inflow at $x = 0$, an outflow at $x = 2$, and no-normal flow on other boundaries. The timestep size is $\Delta t = 5 \times 10^{-3}$, and the system is integrated to $t = \tau = 1$. The model is implemented using FEniCS 2018.1.0. Linear systems are solved using UMFPACK 5.7.1 [13] using PETSc 3.9.2 [5, 12, 4].

A reference calculation is initially performed with T_D defined via interpolation at the inflow boundary vertices of

$$T_D(y) = \sin(\pi y) + 0.4 \sin(3\pi y), \tag{4.11}$$

and with a spatially constant diffusivity $\kappa = 10^{-3}$. This generates a reference state $T_{\mathrm{ref,N}}$ at the end of the simulation at $t = \tau = 1$, shown on the left of figure 4.1.

4.1.4. Differentiating with respect to a Dirichlet boundary condition. The inflow boundary condition T_D is to be treated in the following as a control parameter with respect to which derivatives are to be taken. This requires the ability to compute derivative information associated with an essential Dirichlet boundary condition. First, equation (4.9) is re-written

$$\begin{aligned} &\int_\Omega \phi \frac{T_{0,n+1} + \mathcal{T}_1 - T_n}{\Delta t} + \int_\Omega \phi \nabla^\perp \psi \cdot \nabla \left(\frac{T_n + T_{0,n+1} + \mathcal{T}_1}{2} \right) \\ &+ \int_\Omega \nabla \phi \cdot \kappa \nabla \left(\frac{T_n + T_{0,n+1} + \mathcal{T}_1}{2} \right) = 0 \quad \forall \phi \in V_0, n \in \{0, \ldots, N-1\}, \end{aligned} \tag{4.12}$$

where now each $T_{0,n} \in V_0$, and hence satisfies a homogeneous Dirichlet boundary condition at $x = 0$. $\mathcal{T}_1 \in V$ is equal to T_D on the boundary at $x = 0$. The `tlm_adjoint` "escape hatch" provided by the ability to define custom equations is utilised, deriving a new `InflowBCSolver` class from the abstract `Equation` base class, associated with the equation

$$\mathcal{T}_1 = \begin{cases} T_D & \text{if } x = 0 \\ 0 & \text{at mesh vertices with } x > 0 \end{cases}. \tag{4.13}$$

A related approach for differentiating with respect to essential Dirichlet boundary conditions is described in sections 2.4.3 and 4.4 of [43].

4.1.5. Inverse problem. An objective functional is defined

$$J = \int_{\partial\Omega_{\text{outflow}}} \left(T_N - T_{\text{ref},N}\right)^2 + 10^{-15} \int_{\partial\Omega_{\text{inflow}}} \left(\frac{\mathrm{d}T_D}{\mathrm{d}y}\right)^2, \tag{4.14}$$

where $T_{\text{ref},N}$ is the value of the reference solution at the end of the calculation, and where the integrals are taken over the outflow boundary at $x = 2$ and the inflow boundary at $x = 0$ respectively. A critical point of this functional is obtained by finding a point at which the derivative of J with respect to the inflow boundary condition T_D vanishes, where the derivative is subject to the constraint that the forward model is solved, and where $\kappa = 10^{-3}$. That is, here p consists of the degrees of freedom associated with κ, m consists of the degrees of freedom associated with T_D, and we seek the m such that $\partial\hat{J}/\partial m = 0$, given that $\kappa = 10^{-3}$.

The resulting optimisation problem is solved using Newton's method, via construction of the full dense Hessian $\partial/\partial m \left(\partial\hat{J}/\partial m^T\right)$. For this problem m has length 51 and the optimisation converges in a single Newton step, making the construction of the full dense Hessian tractable. More advanced applications may require more advanced methods for the calculation of such inverse Hessian actions [9]. An inverted state $T_{\text{inv,N}}$ at the end of the simulation at $t = \tau = 1$ is thus obtained, shown on the right of figure 4.1.

For this example the full forward trajectory may be stored in memory. In the evaluation of a forward model constrained Hessian action

$$\frac{\partial}{\partial m}\left(\frac{\partial\hat{J}}{\partial m}\zeta\right), \tag{4.15}$$

the forward and first order adjoint solution are independent of the direction ζ. In this case `tlm_adjoint` provides a means of computing forward model constrained Hessian actions without re-solving the forward. The first order adjoint solution, and data involved in the derivation of tangent-linear and first and second order adjoint equations, are not cached.

4.1.6. Derivatives. A second functional K is defined to be the L^2 norm of the solution at $t = \tau = 1$,

$$K = \int_\Omega T_N^2. \tag{4.16}$$

The forward model constrained derivative of K with respect to the diffusivity κ, subject to the constraint that the forward model is solved with T_D defined via interpolation at boundary vertices of (4.11) and with $\kappa = 10^{-3}$, is shown on the left of figure 4.2. This is a value of the forward model constrained derivative $\partial\hat{K}/\partial p$. Note that the L^2 norm of the solution at the final time is more sensitive to changes in κ near the inflow.

The optimality constrained derivative of K with respect to the diffusivity, subject to the constraint that the optimisation problem is solved, is shown on the right of figure 4.2. This is a value of the optimality constrained derivative $\mathrm{d}\tilde{K}/\mathrm{d}p$. Note that the L^2 norm of the inverted state at the final time is more sensitive to changes in κ near the outflow, in constrast to the forward model constrained derivative $\partial\hat{K}/\partial p$. Note also the significantly increased maximum sensitivity magnitude. As previously observed in [39], if one is solving inverse problems, accuracy requirements for model parameters may differ significantly from the corresponding accuracy requirements of the forward model.

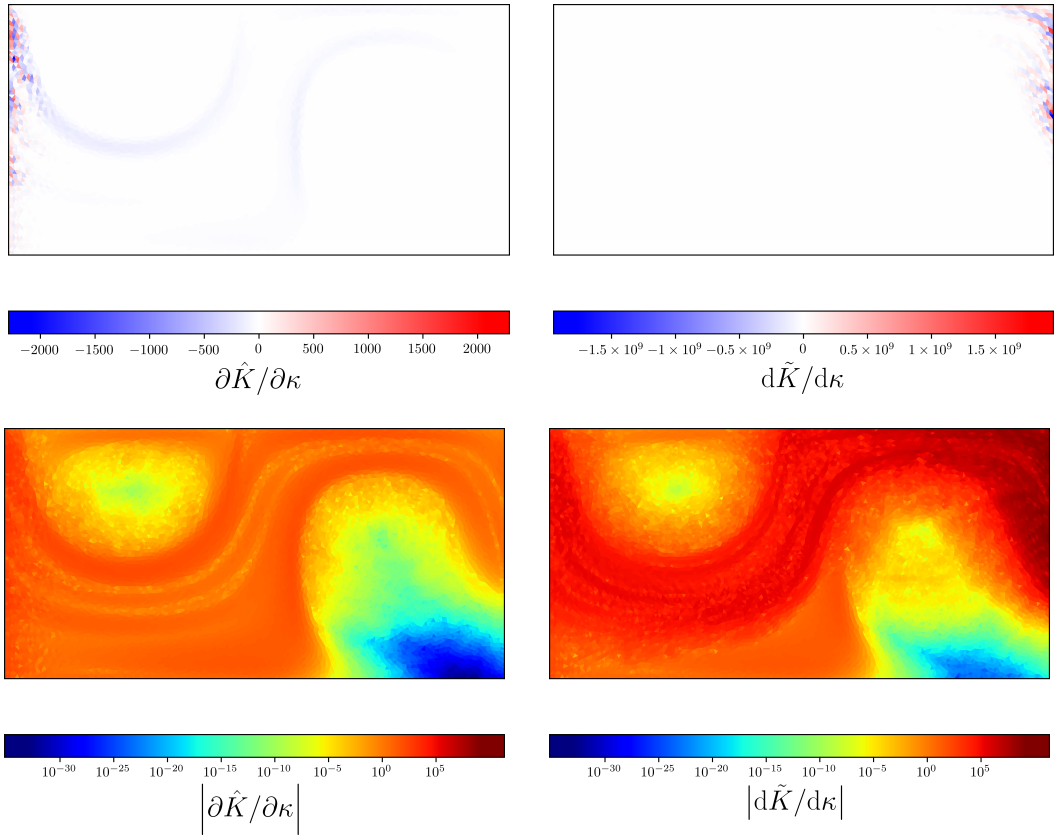

FIG. 4.2. *Upper left: Forward model constrained derivative $\partial\hat{K}/\partial p$ for the advection-diffusion model, with the inflow boundary condition defined via interpolation at the inflow boundary vertices of (4.11), with $\kappa = 10^{-3}$, and where p corresponds to the degrees of freedom associated with κ. Upper right: Optimality constrained derivative $\mathrm{d}\tilde{K}/\mathrm{d}p$, with $\kappa = 10^{-3}$. Lower left/right: The magnitude of the upper figures, with a logarithmic colour scale. Representer functions for the derivatives, defined as in Appendix B of [42], are shown.*

The derivative is verified by considering a Taylor remainder convergence test (see e.g. [16, 42]), measuring the two error norms

$$E_1 = \left|\tilde{K}\left(p+\varepsilon\zeta\right) - \tilde{K}\left(p\right)\right| = \mathcal{O}\left(\varepsilon\right), \tag{4.17a}$$

$$E_2 = \left|\tilde{K}\left(p+\varepsilon\zeta\right) - \tilde{K}\left(p\right) - \varepsilon\frac{\mathrm{d}\tilde{K}}{\mathrm{d}p}\delta p\right| = \mathcal{O}\left(\varepsilon^2\right). \tag{4.17b}$$

Note that each evaluation of $\tilde{K}$ requires the solution of a forward model constrained optimisation problem. The elements of ζ are set equal to pseudorandom values in $[-1, 1)$.[5] The resulting error magnitudes are shown in figure 4.3, demonstrating the second order convergence of the Taylor remainder E_2.

4.1.7. Performance. A performance test is conducted using a higher resolution mesh, generated with Gmsh 3.0.6 [17] with a requested mesh size of 1/120, leading to a mesh with 38371

[5] Pseudorandom values in such intervals are generated using scaling and translation of values generated using the `numpy.random.random` function.

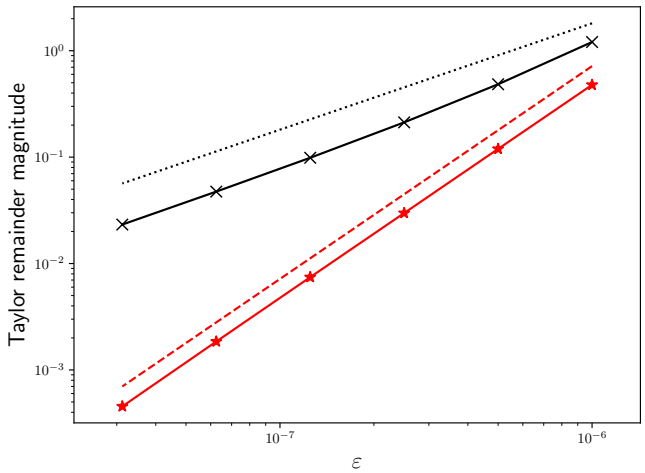

FIG. 4.3. *Taylor remainder convergence test for the optimality constrained derivative. Black crosses / solid line: First order remainder magnitude E_1. Red asterisks / solid line: Second order remainder magnitude E_2. Black dotted line: First order reference. Red dashed line: Second order reference.*

mesh vertices and 76020 triangle elements. The system is integrated for $N = 480$ timesteps of size $\Delta t = 1/480$ with $\kappa = 1/2400$. Tangent-linear calculations compute the derivative of J contracted against a direction ζ with elements taking pseudorandom values in $[-1, 1)$, where ζ corresponds to the degrees of freedom associated with the considered control parameter, and additionally include the solution of the forward equations. First order adjoint calculations compute the derivative of J with respect to the control, and additionally include the solution of the forward equations. Second order adjoint calculations compute the second derivative of J with respect to the control, contracted against the direction ζ, and additionally include the solution of forward, tangent-linear, and first order adjoint equations. Timings are recorded using the Python 3 `time.time` function, with the mean of three timings taken. Initialisation time, the time taken for an additional forward calculation to compute the reference state, and the compilation of low-level code, are excluded. Where storage is used data are fully stored in memory with no checkpointing or recalculation. The forward only calculations, even with annotation and storage disabled, still make use of the `tlm_adjoint` library, for example for the application of the optimisations described in section 3.4. The performance test is conducted on a machine with an Intel Core i5-6300U processor.

Performance results are given in table 4.1. The runtimes relative to the forward only calculation, with no annotation or storage, are considered. A basic estimate of the cost of the calculations, based on a basic estimate of the number of equations which must be solved, is a relative runtime of 2 for tangent-linear and first order adjoint calculations and 4 for a second order adjoint calculation. The calculations with T_D as the control parameter have a runtime which is comparable to these estimates – for example the second order adjoint calculation has a mean relative runtime of 3.948. The tangent-linear calculation with κ as the control parameter and with no annotation or storage, has a comparable efficiency to that with T_D as the control parameter, with a mean relative runtime of 2.196. However the first order adjoint with κ as the control parameter is significantly more expensive with a mean relative runtime of 5.703. Note that, in the tangent-linear calculations, both κ and the direction used to define derivatives are fixed and constant throughout the calculation. Hence all terms appearing in the forward and

Calculation	Annotation / storage enabled?	Control	Mean time (s)	Normalised time
Forward	No	–	4.894	1
Forward	Yes	–	5.337	1.091
Tangent-linear	No	T_D	9.670	1.976
Tangent-linear	Yes	T_D	10.444	2.134
1st order adjoint	Yes	T_D	9.168	1.873
2nd order adjoint	Yes	T_D	19.322	3.948
Tangent-linear	No	κ	10.747	2.196
Tangent-linear	Yes	κ	11.404	2.330
1st order adjoint	Yes	κ	27.908	5.703
2nd order adjoint	Yes	κ	59.009	12.058

TABLE 4.1

Performance results for the advection-diffusion model. The normalised time is the mean runtime, divided by the mean runtime of the forward only calculation.

tangent-linear model with κ as the control parameter (except for the evaluation of the functional) are amenable to a form of optimisation as described in section 3.4. By contrast, in a first order adjoint calculation with κ as the control parameter, terms appear in the adjoint calculation (specifically in the calculation of the forward model constrained derivative of the functional – equation (2.11)) which are not amenable to these optimisations, and are instead calculated using finite element assembly.

If right-hand-side assembly caching optimisations (described in section 3.4) are disabled then the forward only calculation, with annotation and storage disabled, has a mean relative runtime of 5.148. If right-hand-side assembly caching and left-hand-side Jacobian and linear solver caching are disabled then this increases to 39.672.

4.2. Hessian eigendecomposition. In many inverse problems the forward model does not fully constrain the data sought. The forward model constrained Hessian can be used to describe the degree to which different components of the unknown parameter space are constrained, and an eigendecomposition of the Hessian can be used to provide this information (e.g. [37, 36]). An example of such a parameter inversion is that of the subglacial environment of an ice sheet – an oft-solved inverse problem in glacial flow modelling, as the subglacial environment exerts a strong influence on the flow of an ice sheet yet is not easy to observe. In realistic settings the unknown parameter space can be very large – on the order of 10^6 for models of the entire Antarctic continent [36] – and therefore the calculations involved should scale efficiently.

4.2.1. Experiment and equations solved. The inverse experiment is based on that of [25], their section 5.3. Glacial ice evolves over slow time scales as a non-inertial, power-law viscous material. Approximations based on low-aspect ratio lead to the following equations for

ice horizontal velocity $(u, v)^T$ [44, 41],

$$\partial_x \left[(H+h)\nu \left(4\frac{\partial u}{\partial x} + 2\frac{\partial v}{\partial y} \right) \right] + \partial_y \left[(H+h)\nu \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right) \right] - \alpha u$$

$$= \rho g (H+h) \left(\frac{\partial h}{\partial x} + \tan\theta \right), \tag{4.18a}$$

$$\partial_x \left[(H+h)\nu \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) \right] + \partial_y \left[(H+h)\nu \left(2\frac{\partial u}{\partial x} + 4\frac{\partial v}{\partial y} \right) \right] - \alpha u$$

$$= \rho g (H+h) \frac{\partial h}{\partial y}, \tag{4.18b}$$

where

$$\nu(u, v) = \frac{B}{2} \left[\left(\frac{\partial u}{\partial x} \right)^2 + \left(\frac{\partial v}{\partial y} \right)^2 + \frac{1}{4} \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right)^2 + \epsilon \right]^{\frac{1-n}{2n}}. \tag{4.19}$$

Here H and θ are the reference thickness and surface elevation gradient of the ice sheet, and h is the change in height as the ice dynamically evolves. B is a viscosity parameter that in general depends on temperature, but is left constant as in [25]. The quantity $\epsilon = 10^{-12}$ m^2 a^{-2} added here is a small positive real constant which avoids potential singularities when the velocity is spatially uniform. The equations for ice velocity are coupled to a time evolution equation for the thickness h,

$$\frac{\partial h}{\partial t} + \nabla \cdot (\boldsymbol{u}(H+h)) = 0. \tag{4.20}$$

In Experiment 3 of [25] an ice sheet flows through an idealised, periodic domain, which is 40 km $\times$ 40 km. At the initial time, a "slippery spot" appears in the center of the domain, imposed by a spatially varying α,

$$\alpha(x, y) = \left[1000 - 750 e^{-(8r/L)^2} \right] \text{ Pa (m a}^{-1})^{-1}, \tag{4.21}$$

where r is the distance from the centre of the domain, and the surface of the ice sheet adjusts slowly over a decade. See [25], and Table 1 of [24], for physical parameters.

The problem is discretised in space using a continuous Galerkin finite element discretisation. The domain is partitioned into a "cross" mesh constructed using FEniCS, consisting of a 20×20 grid of square cells, each divided into 4 isosceles triangles by dividing each cell with corner-to-corner diagonals. h is discretised as a $P1$ function, i.e. it is linear within each triangle and globally continuous. u and v are discretised as $P2$ functions, i.e. are quadratic within each triangle and globally continuous. The parameter α is approximated using a $P1$ function, via interpolation at the mesh vertices of (4.21). All function spaces are doubly periodic. The equations are further discretised in time using third order Adams-Bashforth, started with a single second order Runge-Kutta step, followed by a single second order Adams-Bashforth step, taking 120 timesteps over the 10 a (year) integration.

A functional is defined

$$J = \sum_{n \in \{60,72,84,96,108,120\}} \left[\frac{1}{\sigma_u^2} (u_n - u_{\text{ref},n})^2 + \frac{1}{\sigma_u^2} (v_n - v_{\text{ref},n})^2 + \frac{1}{\sigma_h^2} (h_n - h_{\text{ref},n})^2 \right], \tag{4.22}$$

where (as in [25] Experiment 3) $\sigma_u = 1$ m a^{-1} and $\sigma_h = 0.02$ m. The values of $u_{\text{ref},n}$, $v_{\text{ref},n}$, and $h_{\text{ref},n}$ are obtained from a reference calculation. The model is initialised with $h = 0$, and for the

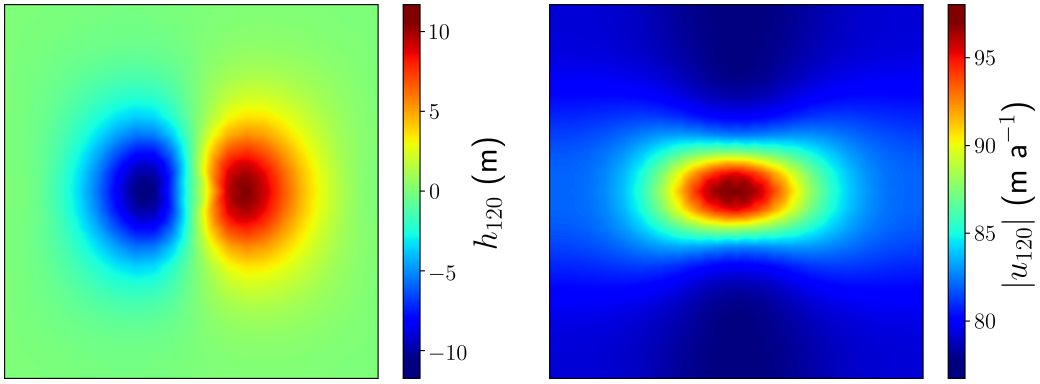

FIG. 4.4. *Forward solution for the glacial flow model at $t = 10$ a. Left: Elevation perturbation. Right: Velocity magnitude, evaluated at the mesh vertices.*

analysis to follow we consider the perfectly optimised state at which all $u_n = u_{\mathrm{ref},n}$, $v_n = v_{\mathrm{ref},n}$, and $h_n = h_{\mathrm{ref},n}$.

The model is implemented using FEniCS 2018.1.0. The non-linear velocity equation is solved using a two-stage fixed-point iteration. In the first stage an approximated form of Newton's method is applied, with an approximate Jacobian defined by not applying the chain rule to differentiate through the viscosity. This is iterated until a very weak tolerance criterion is satisfied, or until 100 iterations have been taken. In the second stage Newton's method is applied, starting from the initial guess of the first stage. Linear systems appearing in the solution of the non-linear velocity equation, and in the associated adjoint and tangent-linear equations, are solved using BoomerAMG [33] using HYPRE 2.14.0 [14] via PETSc 3.9.2 [5, 12, 4]. Other linear systems are solved using successive over relaxation preconditioned conjugate gradient using PETSc 3.9.2.

The results of the forward calculation are shown in figure 4.4. Surface speed has a very similar pattern to that of [25] (their Fig. 4(d)), but examination will show that the speed here is lower than that of [25] by $\sim$ 23-25 m a^{-1}. This is due to the fact those authors use a higher order approximation to the Stokes equations that includes the effects of vertical shearing [24], whereas (4.18) assumes depth-independent flow. The discrepancy can be accounted for by the absence of vertical shearing. If α were uniform, then the contribution of vertical shear to surface velocity in a doubly-periodic ice sheet with the same parameters as those of our model would be $\sim$23 m/a [10]. Since this effect is modified in the presence of horizontal deformation, this "offset" is not spatially uniform.

4.2.2. Eigendecomposition. A generalised eigendecomposition of the forward model constrained Hessian of J defined by (4.22) is considered, with the Hessian defined through differentiation with respect to α at the perfectly optimised state at which the forward solution is equal to the reference. The eigendecomposition defined by the generalised eigenvalue problem

$$\frac{\mathrm{d}}{\mathrm{d}m}\left(\frac{\mathrm{d}\hat{J}}{\mathrm{d}m}v_i\right) = \mu_i M v_i, \tag{4.23}$$

where m corresponds to the degrees of freedom associated with α. M is a symmetric positive definite matrix, here set equal to the mass matrix associated with the discrete function space for α. With this construction the eigenvectors may be defined so that they are orthonormal in the L^2 inner product; it also avoids potential issues with variable mesh resolution skewing the eigenvalue spectrum. The eigendecomposition is performed using the SLEPc 3.9.1 Krylov-Schur

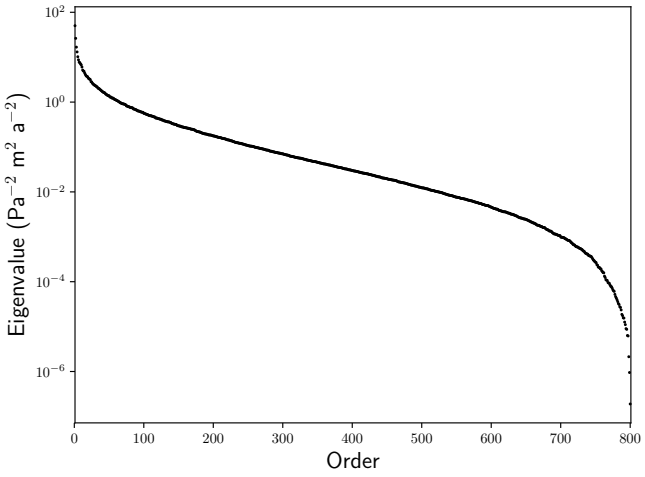

FIG. 4.5. *Eigenvalues of the forward model constrained Hessian, sorted into order from largest to smallest. The Hessian is defined via the forward model constrained second derivative of the functional* (4.22) *with respect to the basal sliding parameter α, at the reference state.*

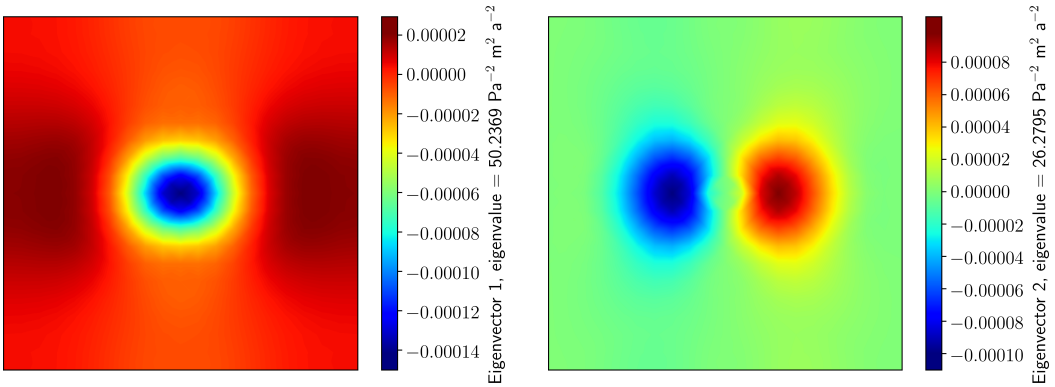

FIG. 4.6. *First two eigenvectors associated with the forward model constrained Hessian, where the Hessian is defined via the forward model constrained second derivative of the functional* (4.22) *with respect to the basal sliding parameter α, at the reference state. The eigenvectors are normalised so that they have an L^2 norm of* 1 *m.*

eigensolver using the slepc4py 3.9.0 Python interface [34, 12, 49]. Forward model constrained Hessian actions are evaluated using caching of the forward solution as described in section 4.1.5.

The resulting eigenvalues are shown in figure 4.5, and the leading two eigenvectors are shown in figure 4.6. In practice the fact that the eigenvalues decay so sharply can be taken advantage of to construct low-rank approximations to the Hessian of the functional which, in combination with *a priori* constraints on the inverted parameters, can be used to find approximate inverses of the Hessian [37, 36].

4.2.3. Performance. The performance of the calculation of derivative information associated with the functional (4.22), with a configuration as described above and with α as a control parameter, is considered. The tangent-linear perturbation direction is defined using a field with coefficients vector with elements taking pseudorandom values in $[-1, 1)$ Pa $(\mathrm{m\ a}^{-1})^{-1}$. Other

Calculation	Annotation / storage enabled?	Control	Mean time (s)	Normalised time
Forward	No	–	64.416	1
Forward	Yes	–	64.577	1.002
Tangent-linear	No	α	81.082	1.259
Tangent-linear	Yes	α	81.708	1.268
1st order adjoint	Yes	α	80.172	1.245
2nd order adjoint	Yes	α	117.153	1.819

TABLE 4.2

Performance results for the glacial flow model. The normalised time is the mean runtime, divided by the mean runtime of the forward only calculation.

relevant details of the performance test are as described in section 4.1.7. In these calculations the value for α is perturbed, with pseudorandom values in $[-10, 10)$ Pa $(\text{m a}^{-1})^{-1}$ added to the reference value in (4.21).

Performance results are given in table 4.2. The runtimes relative to the forward only calculation, with no annotation or storage, are considered. Note the particular efficiency of the tangent-linear and first order adjoint calculations. Even the second order adjoint calculation, including the solution of the forward and tangent-linear with annotation and storage enabled, and the solution of associated first order adjoint equations, has a relative runtime of 1.819. The efficiency here is due to the replacement of multiple linear solves in the solution of the non-linear forward problem for the velocity, with a single linear solve in associated tangent-linear and adjoint equations – an analogous performance benefit to that described in [16].

To test the use the binomial checkpointing strategy described in [29], a further test with a higher resolution "cross" mesh constructed using FEniCS from a 40×40 grid of square cells, and with 600 timesteps over the 10 a interval, is performed. The functional (4.22) is modified so that mismatch terms are added for timesteps 300, 360, 420, 480, 540, and 600. A maximum of 14 disk checkpoints are permitted, which is the smallest number of permitted checkpoints associated with a maximal rerun of 3 (i.e. so that no timestep is run more than 4 times in total). All checkpoints are stored using HDF5. Other details are as in the previous test. The forward calculation (with no annotation and storage) has a mean runtime of 865.298 s. The second order adjoint calculation has a mean runtime of 5268.791 s, which is 6.089 times the forward calculation runtime. Note that for this checkpointing configuration, including the initial forward calculation and all required forward rerunning, a total of 2264 forward timesteps are taken (see [29], equation (3)).

5. Limitations and future work.

5.1. Symbolic differentiation and scaling. The high level algorithmic differentiation considered in this article requires the ability to differentiate symbolic representations of expressions. It is known that differentiation at a symbolic level can be prone to poor scaling as the number of derivatives is increased [27, 30].

Excessive growth in the number of terms appearing in higher derivatives can be mitigated by breaking apart the forward problem into simpler constituent equations. Such a simplification forms a key ingredient in the use of algorithmic differentiation [30]. This may necessitate the conversion of a symbolic expression for a non-linear forward equation into a fixed-point problem, consisting of the successive solution of problems with a simpler form. `tlm_adjoint` includes an `Equation`, in the `FixedPointSolver` class, which can be used to define and solve fixed-point problems. The methodology of [8] (see also [26]) and its tangent-linear analogue [21]

are used to solve tangent-linear or adjoint equations to arbitrary order. This facilitates the manual construction of forward problems involving simpler symbolic expressions, although such constructions are not currently automated.

5.2. Re-use of lower order adjoint solutions. The calculation of a Kth order forward model constrained derivative, contracted against $(K-1)$ directions, involves the solution of adjoint equations of order up to and including order K. The solutions to lower order adjoint equations can be re-used for additional calculations. For example the calculation of a forward model constrained Hessian action involves the solution of both the first and second order adjoint equations, and the first of these (together with the forward solution) allows the forward model constrained derivative of the functional to be computed at a relatively low additional cost, alongside the calculation of a forward model constrained Hessian action. Moreover the first order adjoint solution is independent of the Hessian action direction, meaning that if multiple actions are desired, the first order adjoint equations need only be solved once [20]. At present such an optimised approach is not implemented in `tlm_adjoint`.

5.3. Limitations of the automated code generation system. Since `tlm_adjoint` integrates with and makes use of the FEniCS system, its applications may inherit limitations of the FEniCS system itself. For example it may be impossible or inefficient to implement a limiter scheme using the Unified Form Language. This is addressed to a degree by the simple "escape hatch" interface provided by `tlm_adjoint`, which enables the definition of custom equations and the specification of custom defined derivative information. In principle it may be possible to integrate `tlm_adjoint` with a source-to-source algorithmic differentiation tool to facilitate the definition of such custom equations – such an extension is left for future work.

The primary focus has been on the use of `tlm_adjoint` with the FEniCS system. However key functionality provided by `tlm_adjoint` is independent of the precise backend used. Basic tests using a Firedrake backend [47], and a NumPy backend [46], have already been performed using `tlm_adjoint`.

5.4. Dependency graph optimisations. `tlm_adjoint` performs some very limited optimisations based upon the dependency graph of tangent-linear of adjoint equations – for example avoiding solving adjoint equations whose solutions are known to be zero, as they have no direct or indirect dependency upon the functional. However more advanced optimisations are possible. For example adjoint equations whose solutions do not subsequently contribute to a forward model constrained derivative need not be solved. When applying checkpointing and rerunning, forward equations whose solutions are not (directly or indirectly) dependencies of the adjoint model also need not be solved. Such optimisations are not currently applied by `tlm_adjoint`.

6. Conclusions. This article has described the calculation of partial differential equation constrained derivative information through the automated derivation of tangent-linear equations, and through the automated derivation of associated adjoint information, to arbitrary order. This is achieved by extending the high level approach of [16] to include the automated derivation of tangent-linear equations, with these new tangent-linear equations treated on an equal footing to the original forward equations. This allows adjoint information associated with tangent-linear equations to be derived, using the same machinary as is used for the originating forward equations. Further, this allows a higher order tangent-linear equation to be derived from a lower order tangent-linear equation, and for adjoint information associated with the higher order tangent linear equations to be derived.

The approach is implemented in the `tlm_adjoint` library. The library integrates with the FEniCS automated code generation system [40, 1] for the calculation of higher order derivative information associated with finite element models. The library exposes simple escape hatches

to allow the definition of custom forward equations, and in particular to allow the definition of custom equations which cannot conveniently be represented symbolically. Binomial offline multi-stage checkpointing [29, 50] may be used in adjoint calculations. `tlm_adjoint` further provides the ability to solve appropriate tangent-linear and adjoint fixed-point problems, associated with a given forward fixed-point problem.

The principal focus of this article has been on the calculation of partial differential equation constrained Hessian information. In variational inverse problems this Hessian provides information on the conditioning of the inverse problem (e.g. [55]), can be used to compute derivatives of functionals constrained such that the inversion problem is solved (here termed an "optimality constrained derivative"), and can be used to compute error estimates for inversion products. The latter has potential applications in uncertaintly quantification for variational data assimilation.

Acknowledgements. `tlm_adjoint` is based upon the principles implemented in dolfin-adjoint and libadjoint. `tlm_adjoint` was developed out of a custom extension to dolfin-adjoint, which enabled the use of more general equations with dolfin-adjoint, and this is reflected in many of the design choices in the the library, particularly in the design of the `Equation` class. The earlier custom extension used code derived from dolfin-adjoint, including in the implementation of an earlier version of the `EquationSolver.adjoint_derivative_action` method. `tlm_adjoint` is available under a free and open source license at https://github.com/jrmaddison/tlm_adjoint. JRM acknowledges funding from the U.K. Natural Environment Research Council (NE/L005166/1) during which early development work leading to `tlm_adjoint` was conducted. JRM acknowledges funding from the U.K. Engineering and Physical Sciences Research Council (EP/R021600/1). DNG acknowledges funding from the U.K. Natural Environment Research Council (NE/M003590/1). BDG acknowledges funding from EPSRC (EP/L025159/1) during which work on the parametrization of PDE models was performed. We thank the three anonymous reviewers for their comments on the manuscript.

REFERENCES

[1] M. S. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells, *The FEniCS project version 1.5*, Archive of Numerical Software, 3 (2015), pp. 9–23.

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

[3] N. L. Baker and R. Daley, *Observation and background adjoint sensitivity in the adaptive observation-targeting problem*, Quarterly Journal of the Royal Meteorological Society, 126 (2000), pp. 1431–1454.

[4] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. Gropp, D. Karpeyev, D. Kaushik, M. Knepley, D. May, L. Curfman McInnes, R. Mills, T. Munson, K. Rupp, P. Sanan, B. Smith, S. Zampini, H. Zhang, and H. Zhang, *PETSc Users Manual*, Tech. Report ANL-95/11 Rev 3.9, Argonne National Laboratory, 2018.

[5] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, *Efficient management of parallelism in object-oriented numerical software libraries*, in Modern Software Tools for Scientific Computing, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhäuser, Boston, MA, 1997, pp. 163–202.

[6] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland, *ADIFOR – Generating derivative codes from Fortran programs*, Scientific Programming, 1 (1992), pp. 1–29.

[7] I. Charpentier and J. Utke, *Fast higher-order derivative tensors with Rapsodia*, Optimization Methods & Software, 24 (2009), pp. 1–14.

[8] B. Christianson, *Reverse accumulation and attractive fixed points*, Optimization Methods and Software, 3 (1994), pp. 311–326.

[9] A. Cioaca, A. Sandu, and E. de Sturler, *Efficient methods for computing observation impact in 4D-Var data assimilation*, Computational Geosciences, 17 (2013), pp. 975–990.

[10] K. Cuffey and W. S. B. Paterson, *The Physics of Glaciers*, Butterworth Heinemann, Oxford, 4th ed., 2010.

[11] D. N. Daescu, *On the sensitivity equations of four-dimensional variational (4D-Var) data assimilation*,

Monthly Weather Review, 136 (2008), pp. 3050–3065.
[12] L. D. Dalcin, R. R. Paz, P. A. Kler, and A. Cosimo, *Parallel distributed computing using Python*, Advances in Water Resources, 34 (2011), pp. 1124–1139.
[13] T. A. Davis, *Algorithm 832: UMFPACK V4.3 – an unsymmetric-pattern multifrontal method*, ACM Transactions on Mathematical Software, 30 (2004), pp. 196–199.
[14] R. D. Falgout and U. M. Yang, *hypre: A library of high performance preconditioners*, in Computational Science – ICCS 2002: International Conference Amsterdam, The Netherlands, April 2002 Proceedings, Part III, P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, and A. G. Hoekstra, eds., vol. 2331 of Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg, 2002, pp. 632–641.
[15] P. E. Farrell and S. W. Funke, *libadjoint manual*, Applied Modelling & Computation Group, Department of Earth Science and Engineering, Royal School of Mines, Imperial College London, London, SW7 2AZ, UK, version 1.6 ed., 2018.
[16] P. E. Farrell, D. A. Ham, S. W. Funke, and M. E. Rognes, *Automated derivation of the adjoint of high-level transient finite element programs*, SIAM Journal on Scientific Computing, 35 (2013), pp. C369–C393.
[17] C. Geuzaine and J.-F. Remacle, *Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities*, International Journal for Numerical Methods in Engineering, 79 (2009), pp. 1309–1331.
[18] R. Giering and T. Kaminski, *Recipes for adjoint code construction*, ACM Transactions on Mathematical Software, 24 (1998), pp. 437–474.
[19] R. Giering and T. Kaminski, *Applying TAF to generate efficient derivative code of Fortran 77-95 programs*, Proceedings in Applied Mathematics and Mechanics, 2 (2003), pp. 54–57.
[20] R. Giering, T. Kaminski, and T. Slawig, *Generating efficient derivative code with TAF: Adjoint and tangent linear Euler flow around an airfoil*, Future Generation Computer Systems, 21 (2005), pp. 1345–1355.
[21] J. G. Gilbert, *Automatic differentiation and iterative processes*, Optimization Methods and Software, 1 (1992), pp. 13–21.
[22] B. D. Goddard, A. Nold, and S. Kalliadasis, *Dynamical density functional theory with hydrodynamic interactions in confined geometries*, The Journal of Chemical Physics, 145 (2016), p. 214106.
[23] B. D. Goddard, A. Nold, N. Savva, P. Yatsyshin, and S. Kalliadasis, *Unification of dynamic density functional theory for colloidal fluids to include inertia and hydrodynamic interactions: derivation and numerical experiments*, Journal of Physics: Condensed Matter, 25 (2013), p. 035101.
[24] D. N. Goldberg, *A variationally derived, depth-integrated approximation to a higher-order glaciologial flow model*, Journal of Glaciology, 57 (2011), pp. 157–170.
[25] D. N. Goldberg and P. Heimbach, *Parameter and state estimation with a time-dependent adjoint marine ice sheet model*, The Crysosphere, 7 (2013), pp. 1659–1678.
[26] D. N. Goldberg, S. H. K. Narayanan, L. Hascoet, and J. Utke, *An optimized treatment for algorithmic differentiation of an important glaciological fixed-point problem*, Geoscientific Model Development, 9 (2016), pp. 1891–1904.
[27] A. Griewank, *On automatic differentiation*, in Mathematical Programming: Recent Developments and Applications, M. Iri and K. Tanabe, eds., Kluwer Academic Publishers, 1989, pp. 83–108.
[28] A. Griewank, *Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation*, Optimization Methods and Software, 1 (1992), pp. 35–54.
[29] A. Griewank and A. Walther, *Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation*, ACM Transactions on Mathematical Software, 26 (2000), pp. 19–45.
[30] A. Griewank and A. Walther, *Evaluating derivatives: Principles and techniques of algorithmic differentiation*, SIAM, second ed., 2008.
[31] M. D. Gunzburger, *Perspectives in flow control and optimization*, Advances in design and control, SIAM, 2003.
[32] L. Hascoet and V. Pascual, *The Tapenade automatic differentiation tool: Principles, model, and specification*, ACM Transactions on Mathematical Software, 39 (2013), pp. 20:1–20:43.
[33] V. E. Henson and U. M. Yang, *BoomerAMG: A parallel algebraic multigrid solver and preconditioner*, Applied Numerical Mathematics, 41 (2002), pp. 155–177.
[34] V. Hernandez, J. E. Roman, and V. Vidal, *SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems*, ACM Transactions on Mathematical Software, 31 (2005), pp. 351–362.
[35] V. Heuveline and A. Walther, *Online checkpointing for parallel adjoint computation in PDEs: Application to goal-oriented adaptivity and flow control*, in Euro-Par 2006 Parallel Processing: 12th International Euro-Par Conference Dresden, Germany, August/September 2006 Proceedings, W. E. Nagel, W. V. Walter, and W. Lehner, eds., vol. 4128 of Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg, 2006, pp. 689–699.
[36] T. Isaac, N. Petra, G. Stadler, and O. Ghattas, *Scalable and efficient algorithms for the propagation*

of uncertainty from data through inference to prediction for large-scale problems, with application to flow of the Antarctic ice sheet, Journal of Computational Physics, 296 (2015), pp. 348–368.

[37] A. G. Kalmikov and P. Heimbach, *A Hessian-based method for uncertainty quantification in global ocean state estimation*, SIAM Journal on Scientific Computing, 36 (2014), pp. S267–S295.

[38] N. Kukreja, J. Hückelheim, M. Lange, M. Louboutin, A. Walther, S. W. Funke, and G. Gorman, *High-level python abstractions for optimal checkpointing in inversion problems*, (2018). https://arxiv.org/abs/1802.02474.

[39] F.-X. Le Dimet, H.-E. Ngodock, and B. Luong, *Sensitivity analysis in variational data assimilation*, Journal of the Meteorological Society of Japan, 75 (1997), pp. 245–255.

[40] *Automated solution of differential equations by the finite element method: The FEniCS book*, vol. 84 of Lecture Notes in Computational Science and Engineering, Springer-Verlag Berlin Heidelberg, 2012.

[41] D. R. MacAyeal, *Large-scale ice flow over a viscous basal sediment: Theory and application to ice stream B, Antarctica*, Journal of Geophysical Research: Solid Earth, 94 (1989), pp. 4071–4087.

[42] J. R. Maddison and P. E. Farrell, *Rapid development and adjoining of transient finite element models*, Computer Methods in Applied Mechanics and Engineering, 276 (2014), pp. 95–121.

[43] S. K. Mitusch, *An algorithmic differentiation tool for FEniCS*, master's thesis, University of Oslo, 2018.

[44] L. W. Morland, *Unconfined ice-shelf flow*, in Dynamics of the West Antarctic Ice Sheet, C. J. V. der Veen and J. Oerlemans, eds., Reidel Publ Co, 1987, pp. 99–116.

[45] U. Naumann, *The art of differentiating computer programs: An introduction to algorithmic differentiation*, SIAM, 2012.

[46] T. E. Oliphant, *Python for scientific computing*, Computing in Science & Engineering, 9 (2007), pp. 10–20.

[47] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly, *Firedrake: Automating the finite element method by composing abstractions*, ACM Transactions on Mathematical Software, 43 (2016), pp. 24:1–24:27.

[48] J. M. Restrepo, G. K. Leaf, and A. Griewank, *Circumventing storage limitations in variational data assimilation studies*, SIAM Journal on Scientific Computing, 19 (1998), pp. 1586–1605.

[49] J. E. Roman, C. Campos, E. Romero, and A. Tomás, *SLEPc Users Manual*, Tech. Report DSIC-II/24/02, Departamento de Sistemas Informáticos y Computación, Universitat Politècnica de València, 2018.

[50] P. Stumm and A. Walther, *MultiStage approaches for optimal offline checkpointing*, SIAM Journal on Scientific Computing, 31 (2009), pp. 1946–1967.

[51] P. Stumm and A. Walther, *New algorithms for optimal online checkpointing*, SIAM Journal on Scientific Computing, 32 (2010), pp. 836–854.

[52] The HDF Group, *Heirarchical Data Format, version 5*, 1997-NNNN. http://www.hdfgroup.org/HDF5/.

[53] J. Utke, U. Naumann, M. Fagan, N. Tallent, M. Strout, P. Heimbach, C. Hill, and C. Wunsch, *OpenAD/F: A modular open-source tool for automatic differentiation of Fortran codes*, ACM Transactions on Mathematical Software, 34 (2008), pp. 18:1–18:36.

[54] Q. Wang, P. Moin, and G. Iaccarino, *Minimal repetition dynamic checkpointing algorithm for unsteady adjoint calculation*, SIAM Journal on Scientific Computing, 31 (2009), pp. 2549–2567.

[55] Z. Wang, I. M. Navon, F. X. Le Dimet, and X. Zou, *The second order adjoint analysis: Theory and applications*, Meteorology and Atmospheric Physics, 50 (1992), pp. 3–20.