Math. Program., Ser. A (2016) 155:81–103
DOI 10.1007/s10107-014-0827-4

FULL LENGTH PAPER

Higher-order reverse automatic differentiation with emphasis on the third-order

R. M. Gower · A. L. Gower

Received: 22 June 2013 / Accepted: 10 October 2014 / Published online: 29 October 2014

Abstract It is commonly assumed that calculating third order information is too expensive for most applications. But we show that the directional derivative of the Hessian ($D^3 f(x) \cdot d$) can be calculated at a cost proportional to that of a state-of-the-art method for calculating the Hessian matrix. We do this by first presenting a simple procedure for designing high order reverse methods and applying it to deduce several methods including a reverse method that calculates $D^3 f(x) \cdot d$. We have implemented this method taking into account symmetry and sparsity, and successfully calculated this derivative for functions with a million variables. These results indicate that the use of third order information in a general nonlinear solver, such as Halley–Chebyshev methods, could be a practical alternative to Newton's method. Furthermore, high-order sensitivity information is used in methods for robust aerodynamic design. An efficient high-order differentiation tool could facilitate the use of similar methods in the design of other mechanical structures.

Keywords Automatic differentiation · High-order methods · Tensors vector products · Hessian matrix · Sensitivity analysis

Mathematics Subject Classification 15A69 (Tensor products) · 65D25 (Numerical differentiation) · 65F50 (Sparse matrices) · 49Q12 (Sensitivity analysis)

R. M. Gower (✉)
Maxwell Institute for Mathematical Sciences, School of Mathematics,
University of Edinburgh, Edinburgh, UK
e-mail: gowerrobert@gmail.com

A. L. Gower
School of Mathematics, Statistics and Applied Mathematics,
National University of Ireland Galway, Galway, Ireland

1 Introduction

Derivatives permeate mathematics and engineering right from the first steps of calculus, which together with the Taylor series expansion is a central tool in designing models and methods of modern mathematics. Despite this, successful methods for automatically calculating derivatives of n-dimensional functions is a relatively recent development. Perhaps most notably amongst recent methods is the advent of automatic differentiation (*AD*), which has the remarkable achievement of the "cheap gradient principle", wherein the cost of evaluating the gradient is proportional to that of the underlying function [1]. This AD success is not only limited to the gradient, there also exists a number of efficient AD algorithms for calculating Jacobian [2,3] and Hessian matrices [4,5], that can accommodate for large dimensional sparse instances. The same success has not been observed in calculating higher order derivatives.

The assumed cost in calculating high-order derivatives drives the design of methods, typically favouring the use of lower-order methods. In the optimization community it is generally assumed that calculating any third-order information is too costly, so the design of methods revolves around using first and second order information. We will show that third-order information can be used at a cost proportional to the cost of calculating the Hessian. This has an immediate application in third-order nonlinear optimization methods such as the Chebyshev–Halley Family [6] that require calculating the directional derivative of the Hessian matrix $D^3 f(x) \cdot d$, for a given $x, d \in \mathbb{R}^n$ and $f \in C^3(\mathbb{R}^n, \mathbb{R})$. Though much theory has been examined on the semi-local convergence of members of Halley–Chebyshev family [7–10], it is still unclear how it's domain of convergence compares to that of Newton's method. On one dimensional real and complex equations, where high-order derivatives become trivial, tests have pointed to a larger basin of convergence to roots when applying Halley–Chebyshev methods as compared to Newton's method [11,12]. While finding the solution to nonlinear systems in $\mathbb{R}^n$, there have been a number of successful, albeit limited, tests of the Halley–Chebyshev family, see [13–15] for tests of a modern implementation. These tests indicate that there exist unconstrained problems for which these third-order methods converge with a lower runtime, despite the fact that the entire third-order derivative tensor $D^3 f(x)$ is calculated at each iteration. We present a method that calculates the directional derivative $D^3 f(x) \cdot d$ using only matrix arithmetic, and without the need to form or store the entire third-order tensor. For problems of large dimension, this is a fundamental improvement over calculating the entire tensor. Furthermore, there lacks reports of a significant battery of tests to affirm any practical gain in using this third-order family over Newton's method. Efficient automatic third-order differentiation tools would greatly facilitate such tests.

Still within optimization, third-order derivatives are being used in robust aerodynamic design with promising results [16,17]. With the development of efficient automatic tools for calculating high-order derivatives, this success could be carried over to optimal design of other mechanical structures [18]. Third-order derivatives of financial options are also being considered to design optimal hedging portfolios [19].

The need for higher order differentiation also finds applications in calculating quadratures [20,21], bifurcations and periodic orbits [22]. In the fields of numerical integration and solution of PDE's, a lot of attention has been given to refining and adapting meshes to then use first and second-order approximations over these meshes. An alternative paradigm would be to use fixed coarse meshes and higher approximations. With the capacity to efficiently calculate high-order derivatives, this approach could become competitive and lift the fundamental deterrent in higher-order methods.

Current methods for calculating derivatives of order three or higher in the AD community typically propagate univariate Taylor series [23] or repeatedly apply the tangent and adjoint operations [24]. In these methods, each element of the desired derivative is calculated separately. If AD has taught us anything it is that we should not treat elements of derivatives separately, for their computation can be highly interlaced. The cheap gradient principle illustrates this well, for calculating the elements of the gradient separately yields a time complexity of n times that of simultaneously calculating all entries. This same principle should be carried over to higher order methods, that is, be wary of overlapping calculations in individual elements. Another alternative for calculating high order derivatives is the use of forward differentiation [25]. The drawback of forward propagation is that it calculates the derivatives of all intermediate functions, in relation to the independent variables, even when these do not contribute to the desired end result. For these reasons, we look at calculating high-order derivatives as a whole and focus on reverse AD methods.

An efficient alternative to AD is that the end users hand code their derivatives. Though with the advent of evermore complicated models, this task is becoming increasingly error prone, difficult to write efficient code, and boring. This approach also rules out methods that use high order derivatives, for no one can expect the end user to code the total and directional derivatives of high order tensors.

The article flows as follows, first we develop algorithms that calculate derivatives in a more general setting, wherein our function is described as a sequence of compositions of maps, Sect. 2. We then use Griewank and Walther's [1] state-transformations in Sect. 3, to translate a composition of maps into an AD setting and an efficient implementation. Numerical tests are presented in Sect. 4, followed by our conclusions in Sect. 5.

2 Derivatives of sequences of maps

In preparation for the AD setting, we first develop algorithms for calculating derivatives of functions that can be broken into a composition of operators

$$\mathrm{F}(x) = \Psi^{\ell} \circ \Psi^{\ell-1} \circ \cdots \circ \Psi^{1}(x). \tag{1}$$

for Ψ^i's of varying dimension: $\Psi^1(x) \in C^2(\mathbb{R}^n, \mathbb{R}^{m_1})$ and $\Psi^i(x) \in C^2(\mathbb{R}^{m_{i-1}}, \mathbb{R}^{m_i})$, each $m_i \in \mathbb{N}$ and for $i = 2, \ldots, \ell$, so that $\mathrm{F} : \mathbb{R}^n \to \mathbb{R}^{m_\ell}$. From this we define a functional $f(x) = y^T \mathrm{F}(x)$, where $y \in \mathbb{R}^{m_\ell}$, and develop methods for calculating the *gradient* $\nabla f(x) = y^T D\mathrm{F}(x)$, the *Hessian* $D^2 f(x) = y^T D^2\mathrm{F}(x)$ and the *tensor* $D^3 f(x) = y^T D^3\mathrm{F}(x)$.

For a given $d \in \mathbb{R}^n$, we also develop methods for the directional derivative $D\mathrm{F}(x)\cdot d$, $D^2\mathrm{F}(x) \cdot d$, the Hessian-vector product $D^2 f(x) \cdot d = y^T D^2\mathrm{F}(x) \cdot d$ and the tensor-vector product $D^3 f(x)\cdot d = y^T D^3\mathrm{F}(x)\cdot d$. Notation will be gradually introduced and clarified as is required, including the definition of the preceding directional derivatives.

2.1 First-order derivatives

Taking the derivative of F, Eq. (1), and recursively applying the chain rule, we get

$$y^T D\mathrm{F} = y^T D\Psi^{\ell} D\Psi^{\ell-1}\cdots D\Psi^1. \tag{2}$$

Note that $y^T D\mathrm{F}$ is the transpose of the gradient $\nabla(y^T\mathrm{F})$. For simplicity's sake, the argument of each function is omitted in (2), but it should be noted that $D\Psi^i$ is evaluated at the argument $(\Psi^{i-1}\circ\cdots\circ\Psi^1)(x)$, for each i from 1 to ℓ. In Algorithm 1, we record each of these arguments and then calculate the gradient of $y^T\mathrm{F}(x)$ with what's called a *reverse sweep*. Reverse, for it transverses the maps from the last Ψ^{ℓ} to the first Ψ^1, the opposite direction in which (1) is evaluated. The intermediate stages of the gradient calculation are accumulated in the vector $\overline{\mathrm{v}}$, its dimension changing from one iteration to the next. This will be a recurring fact in the matrices and vectors used to store the intermediate phases of the archetype algorithms presented in this article.

Algorithm 1: Archetype Reverse Gradient.

```
**initialization**: $\mathrm{v}^0 = x, \overline{\mathrm{v}} = y$
**for** $j = 1,\ldots,\ell-1$ **do**
    $\mathrm{v}^j \leftarrow \Psi^j\circ\mathrm{v}^{j-1}$
**end**
**for** $i = \ell,\ldots,1$ **do**
    $\overline{\mathrm{v}}^T \leftarrow \overline{\mathrm{v}}^T D\Psi^i\circ\mathrm{v}^{i-1}$
**end**
**Output**: $y^T D\mathrm{F}(x) = \overline{\mathrm{v}}^T$
```

For convenience, we define the operator D_i as the partial derivative in i-th argument. This way, for any function $g(z)$, the directional derivative of $g(z)$ in the direction d, becomes

$$Dg(z)\cdot d = D_i g(z) d_i, \tag{3}$$

where we have omitted the summation symbol for i, and instead, use Einstein notation where a repeated index implies summation over that index.

We use this notation throughout the article unless otherwise stated. Again using the chain rule and (1), we find

$$D\mathrm{F}\cdot d = D\Psi^{\ell} D\Psi^{\ell-1}\cdots D\Psi^1\cdot d,$$

where we have omitted the arguments. This can be efficiently calculated using a *forward sweep* of the computational graph shown in Algorithm 2. This algorithm is a direct application of the chain rule.

Algorithm 2: Archetype 1st Order Directional Derivative.

```
**initialization**: $\mathrm{v}^0 = x$, $\dot{\mathrm{v}}^0 = d$
**for** $i = 1, \ldots, \ell$ **do**
    $\mathrm{v}^i \leftarrow \Psi^i \circ \mathrm{v}^{i-1}$
    $\dot{\mathrm{v}}^i \leftarrow D\Psi^i(\mathrm{v}^{i-1})\dot{\mathrm{v}}^{i-1}$
**end**
**Output**: $D\mathrm{F}(x) \cdot d = \dot{\mathrm{v}}^\ell$
```

2.2 Second-order derivatives

Here we develop a reverse algorithm for calculating the Hessian $D^2(y^T\mathrm{F}(x))$. First we determine the Hessian for F as a composition of two maps, then we use induction to design a method for when F is a composition of ℓ maps.

For $\mathrm{F}(x) = \Psi^2 \circ \Psi^1(x)$ and $\ell = 2$, we find the Hessian by differentiating in the j-th and k-th coordinate,

$$D_{jk}(y_i F_i) = (y_i D_{rs}\Psi_i^2) D_j \Psi_r^1 D_k \Psi_s^1 + (y_i D_r \Psi_i^2) D_{jk} \Psi_r^1, \tag{4}$$

where the arguments have been omitted. So the (j, k) component of the Hessian $[D^2(y^T\mathrm{F})]_{jk} = D_{jk}(y^T\mathrm{F})$. The higher the order of the derivative, the more messy component notation becomes. A way around this issue is to use a tensor notation, where for every function $g(z)$ we define

$$D^2 g(z) \cdot (v, w) := D_{jk} g(z) v_j w_k,$$

and

$$(D^2 g(z) \cdot w) \cdot v := D^2 g(z) \cdot (v, w), \tag{5}$$

for any function $g(z)$ and compatible vectors v and w. In general,

$$[D^2 g(z) \cdot (\triangle, \square)]_{t_2 \cdots t_q s_2 \cdots s_p} := D_{t_1 s_1} g(z) \triangle_{t_1 t_2 \cdots t_q} \square_{s_1 s_2 \cdots s_p}, \tag{6}$$

and

$$(D^2 g(z) \cdot \square) \cdot \triangle := D^2 g(z) \cdot (\triangle, \square) \tag{7}$$

for any compatible $\triangle$ and $\square$. Using a matrix notation for a composition of maps can be aesthetically unpleasant. Using this tensor notation the Hessian of $y^T\mathrm{F}$, see Eq. (4), becomes

$$\boxed{y^T D^2\mathrm{F} = y^T D^2\Psi^2 \cdot (D\Psi^1, D\Psi^1) + y^T D\Psi^2 \cdot D^2\Psi^1}. \tag{8}$$

We recursively use the identity (8) to design Algorithm 3 that calculates the Hessian of a function $y^T\mathrm{F}(x)$ composed of ℓ maps, as defined in Eq. (1). Algorithm 3 corresponds to a very particular way of nesting the derivatives of the Ψ_i's maps, as detailed in [4]. Though the proof that Algorithm 3 produces the desired Hessian matrix is rather convoluted, thus following Algorithm 3 we present a far simpler proof.

Algorithm 3: Archetype Reverse Hessian.

```
initialization: $\mathrm{v}^0 = x, \bar{\mathrm{v}} = y, \mathrm{W} = 0 \in \mathbb{R}^{m_\ell \times m_\ell}$
for $j = 1, \ldots, \ell - 1$ do
    $\mathrm{v}^j \leftarrow \Psi^j \circ \mathrm{v}^{j-1}$
end
for $i = \ell, \ldots, 1$ do
    $\mathrm{W} \leftarrow \mathrm{W} \cdot (D\Psi^i(\mathrm{v}^{i-1}), D\Psi^i(\mathrm{v}^{i-1}))$
    $\mathrm{W} \leftarrow \mathrm{W} + \bar{\mathrm{v}}^T D^2\Psi^i(\mathrm{v}^{i-1})$
    $\bar{\mathrm{v}}^T \leftarrow \bar{\mathrm{v}}^T D\Psi^i(\mathrm{v}^{i-1})$
end
Output: $y^T D^2\mathrm{F} \leftarrow \mathrm{W},\ y^T D\mathrm{F} \leftarrow \bar{\mathrm{v}}^T$
```

Proof of Algorithm 3 We will use induction on the number of compositions ℓ. For $\ell = 1$ the output is $\mathrm{W} = y^T D^2\Psi^1(x)$. Now we assume that algorithm 3 is correct for functions composed of $\ell - 1$ maps, and use this assumption to show that for ℓ maps the output is $\mathrm{W} = y^T D^2\mathrm{F}(x)$. Let

$$y^T X = y^T \Psi^\ell \circ \cdots \circ \Psi^2,$$

so that $y^T\mathrm{F} = y^T X \circ \Psi^1$. Then at the end of the iteration $i = 2$, by the chain rule, $\bar{\mathrm{v}}^T = y^T DX(\mathrm{v}^1)$ and, by induction, $\mathrm{W} = y^T D^2X(\mathrm{v}^1)$. This way, at termination, or after the iteration $i = 1$, we get

$$\begin{aligned}\mathrm{W} &= y^T D^2X(\mathrm{v}^1) \cdot (D\Psi^1(x), D\Psi^1(x)) + y^T DX(\mathrm{v}^1) \cdot D^2\Psi^1(x) \\ &= y^T D^2(X \circ \Psi^1(x)) \qquad \big[\text{Eq. (8)}\big] \\ &= y^T D^2\mathrm{F}(x).\end{aligned}$$

□

2.3 Third-order methods

We define the directional derivative as

$$\lim_{t\to 0}\frac{d}{dt}D^2 g(z + td) = D_{jkm}g(z)d_m =: D^3g(z) \cdot d, \tag{9}$$

for any function g and compatible vector d. This tensor notation facilitates working with third-order derivatives as using matrix notation would lead to confusing equations and possibly hinder intuition. The notation conventions from before carries over naturally to third-order derivatives, with

$$(D^3g(z) \cdot (\triangle, \square, \diamondsuit))_{t_2\ldots t_q s_2\ldots s_p l_2\ldots l_r} := D_{t_1 s_1 l_1}g(z)\triangle_{t_1\ldots t_q}\square_{s_1\ldots s_p}\diamondsuit_{l_1\ldots l_r}, \tag{10}$$

and

$$D^3 g(z)\cdot(\triangle,\Box,\Diamond) = (D^3 g(z)\cdot\Diamond)\cdot(\triangle,\Box) = ((D^3 g(z)\cdot\Diamond)\cdot\Box)\cdot\triangle, \qquad (11)$$

for any compatible $\triangle$, $\Box$ and $\Diamond$. The Hessian of a composition of two maps $\mathrm{F} = \Psi^2\circ\Psi^1$ is given by Eq. (8), we can use the above to calculate the directional derivative of this Hessian,

$$\begin{aligned} y^T D^3\mathrm{F}\cdot d &= D\left(y^T D^2\Psi^2\cdot(D\Psi^1, D\Psi^1)\right)\cdot d + D\left((y^T D\Psi^2)\cdot D^2\Psi^1\right)\cdot d \\ &= (y^T D^3\Psi^2\cdot D\Psi^1\cdot d)\cdot(D\Psi^1, D\Psi^1) + (y^T D^2\Psi^2)\cdot(D\Psi^1, D^2\Psi^1\cdot d) \\ &\quad + (y^T D^2\Psi^2)\cdot(D^2\Psi^1\cdot d, D\Psi^1) + (y^T D\Psi^2)\cdot D^3\Psi^1\cdot d \\ &\quad + (y^T D^2\Psi^2\cdot D\Psi^1\cdot d)\cdot D^2\Psi^1, \end{aligned}$$

which after some rearrangement gives

$$\begin{aligned} y^T D^3\mathrm{F}\cdot d = y^T D^3\Psi^2\cdot(D\Psi^1, D\Psi^1, D\Psi^1\cdot d) + y^T D\Psi^2\cdot D^3\Psi^1\cdot d \\ + y^T D^2\Psi^2\cdot\Big((D\Psi^1, D^2\Psi^1\cdot d) + (D^2\Psi^1\cdot d, D\Psi^1) \\ + (D^2\Psi^1, D\Psi^1\cdot d)\Big). \end{aligned} \qquad (12)$$

As usual, we have omitted all arguments to the maps. The above applied recursively gives us the Reverse Hessian Directional Derivative Algorithm 4, or *RevHedir* for short. To prove that `RevHedir` produces the desired output, we use induction with $\mathrm{X}^m = y^T\Psi^\ell\circ\cdots\circ\Psi^m$ and work backwards from $m=\ell$ towards $m=1$ to calculate $y^T D^3\mathrm{F}(x)\cdot d$.

Algorithm 4: Archetype Reverse Hessian Directional Derivative (`RevHedir`)

```
initialization: $\mathrm{v}^0 = x, \dot{\mathrm{v}}^0 = d, \bar{\mathrm{v}} = y, \mathrm{W} = \mathrm{Td} = 0 \in \mathbb{R}^{m_\ell\times m_\ell}$
for $j = 1,\ldots,\ell-1$ do
    $\mathrm{v}^j \leftarrow \Psi^j\circ\mathrm{v}^{j-1}$
    $\dot{\mathrm{v}}^j \leftarrow D\Psi^j(\mathrm{v}^{j-1})\cdot\dot{\mathrm{v}}^{j-1}$
end
for $i = \ell,\ldots,1$ do
    $\mathrm{Td} \leftarrow \mathrm{Td}\cdot(D\Psi^i(\mathrm{v}^{i-1}), D\Psi^i(\mathrm{v}^{i-1}))$
    $\mathrm{Td} \leftarrow \mathrm{Td} + \mathrm{W}\cdot(D\Psi^i(\mathrm{v}^{i-1}), D^2\Psi^i(\mathrm{v}^{i-1})\cdot\dot{\mathrm{v}}^{i-1})$
    $\mathrm{Td} \leftarrow \mathrm{Td} + \mathrm{W}\cdot(D^2\Psi^i(\mathrm{v}^{i-1})\cdot\dot{\mathrm{v}}^{i-1}, D\Psi^i(\mathrm{v}^{i-1}))$
    $\mathrm{Td} \leftarrow \mathrm{Td} + \mathrm{W}\cdot(D^2\Psi^i(\mathrm{v}^{i-1}), D\Psi^i(\mathrm{v}^{i-1})\cdot\dot{\mathrm{v}}^{i-1})$;
    $\mathrm{Td} \leftarrow \mathrm{Td} + \bar{\mathrm{v}}^T D^3\Psi^i(\mathrm{v}^{i-1})\cdot\dot{\mathrm{v}}^{i-1}$
    $\mathrm{W} \leftarrow \mathrm{W}\cdot(D\Psi^i(\mathrm{v}^{i-1}), D\Psi^i(\mathrm{v}^{i-1})) + \bar{\mathrm{v}}^T D^2\Psi^i(\mathrm{v}^{i-1})$
    $\bar{\mathrm{v}}^T \leftarrow \bar{\mathrm{v}}^T D\Psi^i(\mathrm{v}^{i-1})$
end
Output: $y^T D^3\mathrm{F}(x)\cdot d \leftarrow \mathrm{Td},\ y^T D^2\mathrm{F} \leftarrow \mathrm{W},\ y^T D\mathrm{F} \leftarrow \bar{\mathrm{v}}^T$
```

Proof of Algorithm 4 Our induction hypothesis is that at the end of the $i=m$ iteration $\mathrm{Td} = y^T D^3\mathrm{X}^m(\mathrm{v}^{m-1})\cdot\dot{\mathrm{v}}^{m-1}$. After the first iteration $i=\ell$ of the second loop, paying attention to the initialization of the variables, we have that

$$\mathrm{Td} = \overline{\mathrm{v}}^T D^3\Psi^\ell(\mathrm{v}^{\ell-1}) \cdot \dot{\mathrm{v}}^{\ell-1} = y^T D^3\mathrm{X}^\ell(\mathrm{v}^{\ell-1}) \cdot \dot{\mathrm{v}}^{\ell-1}.$$

Now suppose the hypothesis is true for iterations up to $m+1$, so that at the beginning of the $i = m$ iteration $\mathrm{Td} = y^T D^3\mathrm{X}^{m+1}(\mathrm{v}^m) \cdot \dot{\mathrm{v}}^m$. To prove the hypothesis we need the following results: at the end of the $i = m$ iteration

$$\overline{\mathrm{v}}^T = y^T D\mathrm{X}^m(\mathrm{v}^{m-1}) \quad \text{and} \quad \mathrm{W} = y^T D^2\mathrm{X}^m(\mathrm{v}^{m-1}), \tag{13}$$

both are demonstrated in the proof of Algorithm 3. Now we are equipt to examine Td at the end of the $i = m$ iteration,

$$\begin{aligned}\mathrm{Td} \leftarrow &\mathrm{Td} \cdot (D\Psi^m(\mathrm{v}^{m-1}), D\Psi^m(\mathrm{v}^{m-1})) + \mathrm{W} \cdot (D\Psi^m(\mathrm{v}^{m-1}), D^2\Psi^m(\mathrm{v}^{m-1}) \cdot \dot{\mathrm{v}}^{m-1})\\ &+ \mathrm{W} \cdot (D^2\Psi^m(\mathrm{v}^{m-1}) \cdot \dot{\mathrm{v}}^{m-1}, D\Psi^m(\mathrm{v}^{m-1}))\\ &+ \mathrm{W} \cdot (D^2\Psi^m(\mathrm{v}^{m-1}), D\Psi^m(\mathrm{v}^{m-1}) \cdot \dot{\mathrm{v}}^{m-1}) + \overline{\mathrm{v}}^T D^3\Psi^m(\mathrm{v}^{m-1}) \cdot \dot{\mathrm{v}}^{m-1}.\end{aligned}$$

Now we use the induction hypothesis followed by property (11) to get

$$\begin{aligned}&\mathrm{Td} \cdot (D\Psi^m(\mathrm{v}^{m-1}), D\Psi^m(\mathrm{v}^{m-1}))\\ &\quad = y^T D^3\mathrm{X}^{m+1}(\mathrm{v}^m) \cdot \dot{\mathrm{v}}^m \cdot (D\Psi^m(\mathrm{v}^{m-1}), D\Psi^m(\mathrm{v}^{m-1}))\\ &\quad = y^T D^3\mathrm{X}^{m+1}(\mathrm{v}^m) \cdot \left(D\Psi^m(\mathrm{v}^{m-1}), D\Psi^m(\mathrm{v}^{m-1}), \dot{\mathrm{v}}^m\right),\end{aligned}$$

and $\dot{\mathrm{v}}^m = D\Psi^m(\mathrm{v}^{m-1}) \cdot \dot{\mathrm{v}}^{m-1}$. Then using Eq. (13) to substitute W and $\overline{\mathrm{v}}^T$ we arrive at

$$\begin{aligned}\mathrm{Td} = &\, y^T D^3\mathrm{X}^{m+1}(\mathrm{v}^m) \cdot \left(D\Psi^m(\mathrm{v}^{m-1}), D\Psi^m(\mathrm{v}^{m-1}), D\Psi^m(\mathrm{v}^{m-1}) \cdot \dot{\mathrm{v}}^{m-1}\right)\\ &+ y^T D^2\mathrm{X}^m(\mathrm{v}^{m-1}) \cdot (D\Psi^m(\mathrm{v}^{m-1}), D^2\Psi^m(\mathrm{v}^{m-1}) \cdot \dot{\mathrm{v}}^{m-1})\\ &+ y^T D^2\mathrm{X}^m(\mathrm{v}^{m-1}) \cdot (D^2\Psi^m(\mathrm{v}^{m-1}) \cdot \dot{\mathrm{v}}^{m-1}, D\Psi^m(\mathrm{v}^{m-1}))\\ &+ y^T D^2\mathrm{X}^m(\mathrm{v}^{m-1}) \cdot (D^2\Psi^m(\mathrm{v}^{m-1}), D\Psi^m(\mathrm{v}^{m-1}) \cdot \dot{\mathrm{v}}^{m-1})\\ &+ \left(y^T D\mathrm{X}^m(\mathrm{v}^{m-1})\right) D^3\Psi^m(\mathrm{v}^{m-1}) \cdot \dot{\mathrm{v}}^{m-1}\\ = &\, y^T D^3\mathrm{X}^m(\mathrm{v}^{m-1}) \cdot \dot{\mathrm{v}}^{m-1} \quad \text{[Using Eq. (12)]}.\end{aligned}$$

Finally, after iteration $i = 1$, we have

$$\mathrm{Td} = y^T D^3\mathrm{X}^1(x) \cdot \dot{\mathrm{v}}^0 = y^T D^3\mathrm{F}(x) \cdot d.$$

□

As is to be expected, in the computation of the tensor-vector product, only 2-dimensional tensor arithmetic, or matrix arithmetic, is used, and it is not necessary to form a 3-dimensional tensor. This is different from current hand-coded implementations of this tensor-vector product used in high-order methods [13–15]. In these articles, the entire third-order tensor $y^T D^3\mathrm{F}(x)$ is formed then contracted with a vector d.

If the entire $y^T D^3\mathrm{F}(x)$ tensor is required, then 3-dimensional arithmetic is unavoidable. A reverse method for calculating the entire third-order tensor $y^T D^3\mathrm{F}(x)$ is given in the final archetype algorithm. For this, we want an expression for the third-order derivative such that

$$\lim_{t\to 0} \frac{d}{dt} y^T D^2\mathrm{F}(x+td) = y^T D^3\mathrm{F}(x)\cdot d, \tag{14}$$

for any vector d. From Eq. (12) we see that d is contracted with the last coordinate in every term except one. To account for this term, we need a *switching tensor* S such that

$$y^T D^2\Psi^2 \cdot (D^2\Psi^1 \cdot d, D\Psi^1) = y^T D^2\Psi^2 \cdot (D^2\Psi^1, D\Psi^1)\cdot S \cdot d,$$

in other words we define S as

$$S\cdot(v,w,z) = (v,z,w) \quad \text{or} \quad S_{abcijk} v_i w_j z_k = v_a z_b w_c, \tag{15}$$

for any vectors v, w and z. This implies that S's components are $S_{abcijk} = \delta_{ai}\delta_{cj}\delta_{bk}$, where $\delta_{nm} = 1$ if $n = m$ and 0 otherwise. Then for $\mathrm{F} = \Psi^2 \circ \Psi^1$ we use Eq. (12) to reach

$$\begin{aligned} y^T D^3\mathrm{F}\cdot d = \Big(& y^T D^3\Psi^2 \cdot (D\Psi^1, D\Psi^1, D\Psi^1) + y^T D\Psi^2 \cdot D^3\Psi^1 \\ &+ y^T D^2\Psi^2 \cdot \Big((D\Psi^1, D^2\Psi^1) + (D^2\Psi^1, D\Psi^1)\cdot S + (D^2\Psi^1, D\Psi^1)\Big)\Big)\cdot d, \end{aligned} \tag{16}$$

as this is true for every d, we can remove d from both sides to arrive at $y^T D^3\mathrm{F}$. With this notation we have, as expected, $(y^T D^3\mathrm{F})_{ijk} = y^T D_{ijk}\mathrm{F}$. We can now use this result to build a recurrence for $D^3\mathrm{X}^m$, with $\mathrm{X}^m = y^T \Psi^\ell \circ \cdots \circ \Psi^m$, working from $m = \ell$ backwards towards $m = 1$ to calculate $y^T D^3\mathrm{F}(x)$, as is done in algorithm 5.

Algorithm 5: Archetype Reverse Third Order Derivative

```
**initialization**: $\mathrm{v}^0 = x, \bar{\mathrm{v}} = y, \mathrm{W} = 0 \in \mathbb{R}^{m_\ell \times m_\ell}, \mathrm{T} \in \mathbb{R}^{m_\ell \times m_\ell \times m_\ell}$
**for** $j = 1, \dots, \ell - 1$ **do**
    $\mathrm{v}^j \leftarrow \Psi^j \circ \mathrm{v}^{j-1}$
**end**
**for** $i = \ell, \dots, 1$ **do**
    $\mathrm{T} \leftarrow \mathrm{T}\cdot(D\Psi^i(\mathrm{v}^{i-1}), D\Psi^i(\mathrm{v}^{i-1}), D\Psi^i(\mathrm{v}^{i-1}))$
    $\mathrm{T} \leftarrow \mathrm{T} + \mathrm{W}\cdot\big((D\Psi^i(\mathrm{v}^{i-1}), D^2\Psi^i(\mathrm{v}^{i-1})) + (D^2\Psi^i(\mathrm{v}^{i-1}), D\Psi^i(\mathrm{v}^{i-1}))\big)$
    $\mathrm{T} \leftarrow \mathrm{T} + \mathrm{W}\cdot(D^2\Psi^i(\mathrm{v}^{i-1}), D\Psi^i(\mathrm{v}^{i-1}))\cdot S + \bar{\mathrm{v}}^T D^3\Psi^i(\mathrm{v}^{i-1})$
    $\mathrm{W} \leftarrow \mathrm{W}\cdot(D\Psi^i(\mathrm{v}^{i-1}), D\Psi^i(\mathrm{v}^{i-1})) + \bar{\mathrm{v}}^T D^2\Psi^i(\mathrm{v}^{i-1})$
    $\bar{\mathrm{v}}^T \leftarrow \bar{\mathrm{v}}^T D\Psi^i(\mathrm{v}^{i-1})$
**end**
**Output**: $y^T D^3\mathrm{F}(x) \leftarrow \mathrm{T},\ y^T D^2\mathrm{F} \leftarrow \mathrm{W},\ y^T D\mathrm{F} \leftarrow \bar{\mathrm{v}}^T$
```

Proof of Algorithm 5 the demonstration can be carried out in an analogous fashion to the proof of Algorithm 4.

The method presented for calculating second and third order derivatives can be extended to design algorithms of arbitrarily higher orders. To do so would require a closed expression for any order derivatives of a composition of two maps $(\Psi^2 \circ \Psi^1(x))$, found in [26], which is too long to reproduce here. Though we can see from this closed expression in [26] that the number of terms needed to be calculated grows combinatorially in the order of the derivative, thus posing a lasting computational challenge. □

3 Implementing through state transformations

When coding a function, the user would not write a composition of maps, such as shown in previous sections, see Eq. (1). Instead users implement functions in a number of different ways. AD packages standardize these hand written functions, through compiler tools and operator overloading, into an evaluation that fits the format of Algorithm 6. As an example, consider the function $f(x_1, x_2, x_3) = x_1 x_2 \sin(x_3)$, and its evaluation for a given (x_1, x_2, x_3) through the following list of commands

$$\begin{aligned} v_{-2} &= x_1 \\ v_{-1} &= x_2 \\ v_0 &= x_3 \\ v_1 &= v_{-2}v_{-1} \\ v_2 &= \sin(v_0) \\ v_3 &= v_2 v_1. \end{aligned}$$

By naming the functions $\phi_1(v_{-2}, v_{-1}) := v_{-2}v_{-1}$, $\phi_2(v_0) := \sin(v_0)$ and $\phi_3(v_2, v_1) := v_2 v_1$, this evaluation is the same as done by Algorithm 6.

In general, each ϕ_i is an *elemental function* such as addition, multiplication, $\sin(\cdot)$, $\exp(\cdot)$, etc, which together with their derivatives are already coded in the AD package. In order, the algorithm first copies the *independent* variables x_i into internal *intermediate* variables v_{i-n}, for $i = 1, \ldots, n$. Following convention, we use negative indexes for elements that relate to independent variables. For consistency, we will shift all indexes of vectors and matrices by $-n$ from here on, e.g., the components of $x \in \mathbb{R}^n$ are x_{i-n} for $i = 1 \ldots n$.

The next step in Algorithm 6 calculates the value v_1 that only depends on the intermediate variables v_{i-n}, for $i = 1, \ldots, n$. In turn, the value v_2 may now depend on v_{i-n}, for $i = 1, \ldots, n+1$, then v_3 may depend on v_{i-n}, for $i = 1, \ldots, n+2$ and so on for all ℓ intermediate variables. Each v_i is calculated using only one elemental function ϕ_i.

This procedure at each step establishes a dependency among the intermediate variables v_i for $i = 1, \ldots, \ell$. We say that j is a predecessor of i if v_j is needed to calculate v_i, that is, if v_j is an argument of ϕ_i. This way we let $P(i)$ be the set of predecessors of i, and $v_{P(i)}$ a vector of the predecessors, so that $\phi_i(v_{P(i)}) = v_i$. Note that $j \in P(i)$ implies that $j < i$. Analogously, we can define $S(i)$ as the set of successors of i.

Algorithm 6: Function evaluation

```
Input: $v_{i-n} = x_i$, for $i = 1, \dots n$
for $i = 1 \dots \ell$ do
    $v_i \leftarrow \phi_i(v_{P(i)})$
end
Output: $f(x) \leftarrow v_\ell$
```

We can bridge this algorithmic description of a function with that of compositions of maps (1) using Griewank and Walther's [1] state-transformations

$$\begin{aligned}\Phi^i : \mathbb{R}^{n+\ell} &\to \mathbb{R}^{n+\ell},\\ \mathrm{v} &\mapsto (v_{1-n}, \dots, v_{i-1}, \phi_i(v_{P(i)}), v_{i+1}, \dots, v_\ell)^T,\end{aligned} \tag{17}$$

for $i = 1, \dots \ell$, where v is a vector with components v_i. In components,

$$\Phi^i_r(\mathrm{v}) = v_r(1 - \delta_{ri}) + \delta_{ri}\phi_i(v_{P(i)}), \tag{18}$$

where here, and in the remainder of this article, we abandon Einstein's notation of repeated indexes, because having the limits of summation is important when implementing. With this, the value $f(x)$ given by Algorithm 6 can be written as

$$f(x) = e^T_{\ell+n}\Phi^\ell \circ \Phi^{\ell-1} \circ \dots \circ \Phi^1 \circ (P^T x), \tag{19}$$

where $e_{\ell+n}$ is the $(\ell + n)$th canonical vector and P is the immersion matrix $[I\ 0]$ with $I \in \mathbb{R}^{n\times n}$ and $0 \in \mathbb{R}^{n\times\ell}$. The Jacobian of the ith state transformation Φ^i, in coordinates, is simply

$$D_j\Phi^i_r(\mathrm{v}) = \delta_{rj}(1 - \delta_{ri}) + \delta_{ri}\frac{\partial\phi_i}{\partial v_j}(v_{P(i)}). \tag{20}$$

With the state-transforms and the structure of their derivatives, we look again at a few of the archetype algorithms in Sect. 2 and build a corresponding implementable version. Our final goal is to implement the `RevHedir` algorithm 4, for which we need the implementation of the reverse gradient and Hessian algorithms.

3.1 First-order derivatives

To design an algorithm to calculate the gradient of $f(x)$, given in Eq. (19), we turn to the Archetype Reverse Gradient Algorithm 1 and identify[1] the Φ^i's in place of the Ψ^i's. Using (20) we find that $\bar{\mathrm{v}}^T \leftarrow \bar{\mathrm{v}}^T D\Phi^i$ becomes

$$\bar{v}_j \leftarrow \bar{v}_j(1 - \delta_{ij}) + \bar{v}_i\frac{\partial\phi_i}{\partial v_j}(v_{P(i)}) \quad \forall j \in \{1 - n, \dots, \ell\} \tag{21}$$

[1] Specifically P^T would be Ψ^1 and Φ^i would be Ψ^{i+1}.

where $\bar{v}_i$ is the i-th component of $\bar{\mathrm{v}}$, also known as the i-th *adjoint* in the AD literature. Note that if $j \neq i$ in the above, then the above step will only alter $\bar{v}_j$ if $j \in P(i)$. Otherwise if $j = i$, then this update is equivalent to setting $\bar{v}_i = 0$. We can disregard this update, as $\bar{v}_i$ will not be used in subsequent iterations. This is because $i \notin P(m)$, for $m \leq i$. With these considerations, we arrive at the algorithm 7, the component-wise version of algorithm 1. Note how we have used the abbreviated operation $a+ = b$ to mean $a \leftarrow a + b$. Furthermore, the last step $\nabla f \leftarrow \bar{\mathrm{v}}^T P^T$ selects the adjoints corresponding to independent variables.

An abuse of notation that we will employ throughout, is that whenever we refer to $\bar{v}_i$ in the body of the text, we are referring to the value of $\bar{v}_i$ after iteration i of the Reverse Gradient algorithm has finished.

Algorithm 7: Reverse Gradient.

```
Input: $\bar{\mathrm{v}} = e_{n+\ell} \in \mathbb{R}^{\ell+n}$, $v_{i-n} = x_i$, for $i = 1, \ldots n$
for $i = 1 \ldots \ell$ do
    $v_i \leftarrow \phi_i(v_{P(i)})$
end
for $i = \ell, \ldots, 1$ do
    for $j \in P(i)$ do  $\bar{v}_j + = \bar{v}_i \partial\phi_i(v_{P(i)})/\partial v_j$
end
Output: $\nabla f \leftarrow \bar{\mathrm{v}}^T P^T = (\bar{v}_{1-n}, \ldots, \bar{v}_0)^T$
```

Similarly, by using (20) again, each iteration i of the Archetype 1st Order Directional Derivative Algorithm 2, can be reduced to a coordinate form

$$\dot{v}_r \leftarrow (1 - \delta_{ri})\dot{v}_r + \delta_{ri} \sum_{j \in P(i)} \dot{v}_j \frac{\partial \phi_i}{\partial v_j}(v_{P(i)}),$$

where $\dot{v}_j$ is the j-th component of $\dot{\mathrm{v}}$. If $r \neq i$ in the above, then $\dot{v}_r$ remains unchanged, while if $r = i$ then we have

$$\dot{v}_i \leftarrow \sum_{j \in P(i)} \dot{v}_j \frac{\partial \phi_i}{\partial v_j}(v_{P(i)}). \tag{22}$$

We implement this update by sweeping through the successors of each intermediate variable and incrementing a single term to the sum on the right-hand side of (22), see Algorithm 8. It is crucial to observe that the i-th component of $\dot{\mathrm{v}}$ will remain unaltered after the i-th iteration.

Again, when we refer to $\dot{v}_i$ in the body of the text from this point on, we are referring to the value of $\dot{v}_i$ after iteration i has finished in Algorithm 8.

Though we have included the explicit argument $v_{P(i)}$ of each ϕ_i function and each derivative of ϕ_i in this section, we now omit this argument from now on to avoid a cluttered notation.

Algorithm 8: 1st Order Directional Derivative.

```
initialization: $\dot{\mathrm{v}} = P^T d \in \mathbb{R}^{\ell+n}$, $v_{i-n} = x_i$, for $i = 1, \dots n$
for $j = 1, \dots, \ell$ do
    $v_j \leftarrow \phi_j(v_{P(j)})$
    for $i \in S(j)$ do $\dot{v}_i + = \dot{v}_j \partial\phi_i(v_{P(i)})/\partial v_j$
end
Output: $D\mathrm{F} \cdot d = [\dot{v}_{1-n}, \dots, \dot{v}_\ell]$
```

3.2 Second-order derivatives

Just by substituting Ψ^is for Φ^is in the Archetype Reverse Hessian, Algorithm 3, we can quickly reach a very efficient component-wise algorithm for calculating the Hessian of $f(x)$, given in Eq. (19). This component-wise algorithm is also known as edge_pushing, and has already been detailed in Gower and Mello [4]. Here we use a different notation which leads to a more concise presentation. Furthermore, the results below form part of the calculations needed for third order methods.

There are two steps of Algorithm 3 we must investigate, for we already know how to update $\overline{\mathrm{v}}$ from the above section. For these two steps, we need to substitute

$$D_{jk}\Phi^i_r(\mathrm{v}) = \frac{\partial^2 \Phi^i_r}{\partial v_j \partial v_k}(\mathrm{v}) = \delta_{ri}\frac{\partial^2 \phi_i}{\partial v_j \partial v_k}(v_{P(i)}), \tag{23}$$

and $D\Phi^i$, Eq. (20), in $\mathrm{W} \leftarrow \mathrm{W} \cdot (D\Phi^i, D\Phi^i) + \overline{\mathrm{v}}^T D^2\Phi^i$, resulting in

$$\begin{aligned} W_{jk} \leftarrow & \sum_{s,t=1-n}^{\ell} \frac{\partial \Phi^i_s}{\partial v_j} W_{st} \frac{\partial \Phi^i_t}{\partial v_k} + \sum_{s=1-n}^{\ell} \bar{v}_s \frac{\partial^2 \Phi^i_s}{\partial v_j \partial v_k} \\ =&(1-\delta_{ji})W_{jk}(1-\delta_{ki}) + \frac{\partial \phi_i}{\partial v_j} W_{ii} \frac{\partial \phi_i}{\partial v_k} \\ &+ \frac{\partial \phi_i}{\partial v_j} W_{ik}(1-\delta_{ki}) + (1-\delta_{ji})W_{ji}\frac{\partial \phi_i}{\partial v_k} \qquad (24) \\ &+ \bar{v}_i \frac{\partial^2 \phi_i}{\partial v_j \partial v_k}, \qquad (25) \end{aligned}$$

where W_{jk} is the jk component of W. Before translating these updates into an algorithm, we need a crucial result: at the beginning of iteration $i-1$, the element W_{jk} is zero if $j \geq i$ for all k. We show this by using induction on the iterations of Algorithm 3. Note that W is initially set to zero, so for the first iteration $i = \ell$ both (24) and (25) reduce to

$$W_{jk} \leftarrow \bar{v}_\ell \frac{\partial^2 \phi_\ell}{\partial v_j \partial v_k},$$

which is zero for $j = \ell$ because $\ell \notin P(\ell)$. Now we assume the induction hypothesis holds at the beginning of the iteration i, so that $W_{jk} = 0$ for $j \geq i+1$. So letting $j \geq i+1$ and executing the iteration i we get from the updates (24)and (25)

$$W_{jk} \leftarrow W_{jk} + W_{ji}\frac{\partial \phi_i}{\partial v_k},$$

because $j \notin P(i)$ so $\partial\phi_i/\partial v_j = 0$. Together with our hypothesis $W_{jk} = 0$ and $W_{ji} = 0$, we see that W_{jk} remains zero. While if $j = i$, then (24) and (25) sets $W_{jk} \leftarrow 0$ because $i \notin P(i)$. Hence at the beginning of iteration $i - 1$ we have that $W_{jk} = 0$ for $j \geq i$ and this completes the induction.

Furthermore, W is symmetric at the beginning of iteration i because it is initialized to $\mathrm{W} = 0$ and each iteration preserves symmetry. Consequentially W_{jk} is only updated when both $j, k \leq i$. We make use of this symmetry to avoid unnecessary calculations on symmetric counterparts.

Let $W_{\{kj\}} = W_{\{jk\}}$ denote both W_{jk} and W_{kj}. We rewrite update (24) and (25) considering only $j, k < i$, for W_{jk} only gets updated if $j, k \leq i$ and when $j = i$ or $k = i$ we know that $W_{jk} = 0$ at the end of iteration i. The first two terms of update (24) become,

$$W_{\{jk\}} += \frac{\partial \phi_i}{\partial v_j} W_{\{ii\}} \frac{\partial \phi_i}{\partial v_k}.$$

The next two terms can be written as

$$W_{\{jk\}} += \frac{\partial \phi_i}{\partial v_j} W_{\{ik\}} \quad \text{and} \quad W_{\{kj\}} += \frac{\partial \phi_i}{\partial v_k} W_{\{ij\}}. \tag{26}$$

Note that these two are the same with j changed for k. If $j \neq k$ we can replace both these operations with just

$$W_{\{jk\}} += \frac{\partial \phi_i}{\partial v_j} W_{\{ik\}}, \tag{27}$$

if we apply this update for every $j, k < i$ and consider that $W_{\{jk\}}$ and $W_{\{kj\}}$ represent the same number.

If $j = k$ these two operations become

$$W_{\{jj\}} += 2\frac{\partial \phi_i}{\partial v_j} W_{\{ik\}}.$$

The updates (24) and (25) have been implemented with these above considerations in the `Pushing` step in Algorithm 9. The names of the steps `Creating` and `Pushing` are elusive to a graph interpretation [4].

3.3 Third-order derivatives

The final algorithm that we translate to implementation is the Hessian directional derivative, the `RevHedir` Algorithm 4. This implementation has an immediate application in the Halley–Chebyshev class of third-order optimization methods, for at each step of these algorithms, such a directional derivative is required.

Algorithm 9: component-wise form of `edge_pushing`.

```
Input: Function evaluation 6, $x \in \mathbb{R}^n$.
Initialization: $\bar{v}_{1-n} = \cdots = \bar{v}_{\ell-1} = 0$, $\bar{v}_\ell = 1$, $W_{\{jk\}} = 0$ for $j, k \in \{1-n, \ldots, \ell\}$, $v_j = x_j$ for
$j \in \{1-n, \ldots, 0\}$, $v_1 = \cdots = v_\ell = 0$, ;
Calculate and store $v_i$, for $i \in \{1-n, \ldots, \ell\}$ using Algorithm 6;
for $i = \ell, \ldots, 1$ do
    Pushing;
    foreach $k \leq i$ such that $W_{\{ki\}} \neq 0$ do
        if $k < i$ then
            foreach $j \in P(i)$ do
                if $j = k$ then
                    $W_{\{jj\}} += 2 D_j\phi_i W_{\{ji\}}$
                else
                    $W_{\{jk\}} += D_j\phi_i W_{\{ki\}}$
                end
            end
        else $k = i$
            foreach unordered pair $\{j, p\} \subset P(i)$ do
                $W_{\{jp\}} += D_p\phi_i D_j\phi_i W_{\{ii\}}$
            end
        end
    end
    Creating;
    foreach unordered pair $\{j, p\} \subset P(i)$ do
        $W_{\{jp\}} += \bar{v}_i D_{pj}\phi_i$
    end
    Adjoint;
    foreach $j \in P(i)$ do
        $\bar{v}_j += \bar{v}_i D_j\phi_i$
    end
end
Output: $D^2 f = (W_{jk})_{1-n \leq j,k \leq 0}$
```

Identifying each Ψ^i with Φ^i, we address each of the five operations on the matrix Td in Algorithm 4 separately, pointing out how each one preserves the symmetry of Td and how to perform the component-wise calculations.

First, given that Td is symmetric, the *2D pushing* update

$$\mathrm{Td} \leftarrow \mathrm{Td} \cdot \left(D\Phi^i, D\Phi^i\right), \tag{28}$$

is exactly as detailed in (24) and the surrounding comments. While the update *3D creating*

$$\mathrm{Td} \leftarrow \mathrm{Td} + \bar{\mathrm{v}}^T D^3\Phi^i \cdot \dot{\mathrm{v}}^{i-1},$$

can be written in coordinate form as

$$\begin{aligned} Td_{jk} &\leftarrow Td_{jk} + \sum_{r,p=1-n}^{\ell} \bar{v}_r D_{jkp}\Phi^i_r \dot{v}^{i-1}_p \\ &= Td_{jk} + \sum_{p \in P(i)} \bar{v}_i \frac{\partial^3 \phi_i}{\partial v_j \partial v_k \partial v_p} \dot{v}_p, \end{aligned} \tag{29}$$

where $\dot{v}_p^{i-1}$ is the $p-$th component of $\dot{\mathrm{v}}^{i-1}$, $\dot{v}_p$ is the output of Algorithm 8 and Td_{jk} is the jk component of Td. Note that $\dot{v}_p^{i-1} = \dot{v}_p$ for $p \in P(i)$, because $p \leq i-1$, so on the iteration $i-1$ of Algorithm 8 the calculation of $\dot{v}_p$ will already have been finalized. Another trick we employ is that, since the above calculation is performed on iteration i, we know that $\bar{v}_i$ has already been calculated. These substitutions involving $\bar{v}_i$s and $\dot{v}_i$s will be carried out in the rest of the text with little or no comment. The update (29) also preserves the symmetry of Td.

To examine the update,

$$\mathrm{Td} \leftarrow \mathrm{Td} + \mathrm{W} \cdot \left(D\Phi^i, D^2\Phi^i \cdot \dot{\mathrm{v}}^{i-1} \right), \tag{30}$$

we use (20) and (23) to obtain the coordinate form

$$\begin{aligned} Td_{jk} &\leftarrow Td_{jk} + \sum_{r,s=1-n}^{\ell} W_{rs} \left(\delta_{rj}(1-\delta_{ri}) + \delta_{ri}\frac{\partial \phi_i}{\partial v_j} \right) \delta_{si} \frac{\partial^2 \phi_i}{\partial v_k \partial v_p} \dot{v}_p \\ &= Td_{jk} + W_{ji}(1-\delta_{ji}) \frac{\partial^2 \phi_i}{\partial v_k \partial v_p} \dot{v}_p + W_{ii} \frac{\partial \phi_i}{\partial v_j} \frac{\partial^2 \phi_i}{\partial v_k \partial v_p} \dot{v}_p. \end{aligned} \tag{31}$$

Upon inspection, the update

$$\mathrm{Td} \leftarrow \mathrm{Td} + \mathrm{W} \cdot \left(D^2\Phi^i \cdot \dot{\mathrm{v}}^{i-1}, D\Phi^i \right)$$

is the transpose of (31) due to the symmetry of W. So it can be written in coordinate form as

$$Td_{jk} \leftarrow Td_{jk} + W_{ik}(1-\delta_{ki}) \frac{\partial^2 \phi_i}{\partial v_j \partial v_p} \dot{v}_p + W_{ii} \frac{\partial \phi_i}{\partial v_k} \frac{\partial^2 \phi_i}{\partial v_j \partial v_p} \dot{v}_p. \tag{32}$$

Thus update (32) together with (31) gives a symmetry contribution to Td.

Last we translate

$$\mathrm{Td} \leftarrow \mathrm{Td} + \mathrm{W} \cdot \left(D^2\Phi^i, D\Phi^i \cdot \dot{\mathrm{v}}^{i-1} \right), \tag{33}$$

to its coordinate form

$$\begin{aligned} Td_{jk} &\leftarrow Td_{jk} + \sum_{r,s=1-n}^{\ell} W_{rs}\delta_{ri} D_{jk}\Phi_r^i \left(\delta_{sp}(1-\delta_{si}) + \delta_{si}\frac{\partial \phi_i}{\partial v_p} \right) \dot{v}_p \\ &= Td_{jk} + W_{ip} \frac{\partial^2 \phi_i}{\partial v_j \partial v_k}(1-\delta_{pi})\dot{v}_p + W_{ii} \frac{\partial^2 \phi_i}{\partial v_j \partial v_k} D_p \phi_i \dot{v}_p. \end{aligned} \tag{34}$$

No change is affected by interchanging the indices j and k on the right-hand side of (34), so once again Td remains symmetric. For convenience of computing, we group updates (31), (32) and (34) into a set of updates called `2D Connecting`. The name indicating that these updates "connect" objects that contain second order derivative information.

More than just symmetric, by closely inspecting these operations we see that the sparsity structure of Td is contained in the sparisty structure of W. This remains true even after execution, at which point Td $= D^3 f(x) \cdot d$ and W $= D^2 f(x)$ where, for each $j, k, p \in \{1-n, \dots, 0\}$, we have

$$D_{jk} f(x) = 0 \implies D_{jkp} f(x) d_p = 0.$$

This fact should be explored when implementing the method, in that, the data structure of Td should imitate that of W.

3.3.1 Implementing third-order directional derivative

The matrices Td and W are symmetric, and based on the assumption that they will be sparse, we will represent them using a symmetric sparse data structure. Thus we now identify each pair (W_{jk}, W_{kj}) and (Td_{jk}, Td_{kj}) with the element $W_{\{jk\}}$ and $Td_{\{jk\}}$, respectively. Much like in `edge_pushing`, Algorithm 9, we want an efficient implementation of the updates to $Td_{\{jk\}}$ that only takes the contributions from nonzero elements of $Td_{\{ik\}}$ and $W_{\{ik\}}$, and does not repeat unnecessary calculations.

We must take care when updating our symmetric representation of Td, both for the 2D pushing update (28) and for the redundant symmetric counterparts (31) and (32) which "double-up" on the diagonal, much like in the `Pushing` operations of Algorithm 9. Each operation (31), (32) and (34) depends on a diagonal element $W_{\{ii\}}$ and an off-diagonal element $W_{\{ik\}}$ of W, for $k \neq i$. Grouping together all terms that involve $W_{\{ii\}}$ we get the resulting update

$$Td_{\{jk\}} += W_{\{ii\}} \sum_{p \in P(i)} \dot{v}_p \left(\frac{\partial \phi_i}{\partial v_j} \frac{\partial^2 \phi_i}{\partial v_k \partial v_p} + \frac{\partial \phi_i}{\partial v_k} \frac{\partial^2 \phi_i}{\partial v_j \partial v_p} + \frac{\partial \phi_i}{\partial v_p} \frac{\partial^2 \phi_i}{\partial v_j \partial v_k} \right). \quad (35)$$

Similar to how the update (26) was split into two updates (27), here by appropriately renaming the indices in (31), (32) and (34), each nonzero off diagonal elements $W_{\{ik\}}$ results in the updates (36), (37) and (38), respectively.

$$Td_{jk} += \sum_{p \in P(i)} \dot{v}_p \frac{\partial^2 \phi_i}{\partial v_j \partial v_p} W_{ik}, \quad \forall j \in P(i) \quad (36)$$

$$Td_{kj} += \sum_{p \in P(i)} \dot{v}_p \frac{\partial^2 \phi_i}{\partial v_j \partial v_p} W_{ki}, \quad \forall j \in P(i) \quad (37)$$

$$Td_{jp} += \sum_{p \in P(i)} \dot{v}_k \frac{\partial^2 \phi_i}{\partial v_j \partial v_p} W_{ik}, \quad \forall j \in P(i) \quad (38)$$

Note that (36) and (37) are symmetric updates, and when $j = k$ these two operations "double-up" resulting in the update

$$Td_{jj} += 2 \sum_{p \in P(i)} \dot{v}_p \frac{\partial^2 \phi_i}{\partial v_j \partial v_p} W_{ij}.$$

Passing to our symmetric notation, both (37) and (36) can be accounted for by using (36) to update $Td_{\{jk\}}$ over every j and k, where $Td_{\{jk\}} = Td_{\{kj\}}$, and with an exception for this doubling effect in Algorithm 10. Finally we can eliminate redundant symmetric calculations performed in (38) by only performing this operation for each pair $\{j, p\}$. All these considerations relating to `2D connecting` have been factored into our implementation of the `RevHedir` Algorithm 10.

Performing `3D Creating` (29) using this symmetric representation is simply a matter of not repeating the obvious symmetric counterpart, but instead, performing these operations on $Td_{\{jk\}}$ once for each appropriate pair $\{j, k\}$, see `3D Creating` in to Algorithm 10.

Algorithm 10: component-wise form of `RevHedir`.

```
Input: Function evaluation 6, $x \in \mathbb{R}^n$.
Initialization: $\bar{v}_{1-n} = \cdots = \bar{v}_{\ell-1} = 0$, $\bar{v}_\ell = 1$, $W_{\{jk\}} = 0$ for $j, k \in \{1-n, \ldots, \ell\}$, $v_j = x_j$,
$Td_{\{jk\}} = 0$ for $j < k \in \{1-n, \ldots, \ell\}$
Calculate and store $\dot{v}_i$ and $v_i$ for $i \in \{1-n, \ldots, \ell\}$ using Algorithm 8
for $i = \ell, \ldots, 1$ do
    2D Pushing of $Td$, see Pushing in Algorithm 9
    2D Connecting
    foreach $p \in P(i)$, $\{j, k\} \subset P(i)$ do
        $Td_{\{jk\}} += W_{\{ii\}} \dot{v}_p \left( D_j\phi_i D_{kp}\phi_i + D_k\phi_i D_{jp}\phi_i + D_p\phi_i D_{jk}\phi_i \right)$
    end
    foreach $k < i$, $W_{\{ik\}} \neq 0$ do
        foreach $(j, p) \in P(i)^2$ do
            if $j = k$ then
                $Td_{\{kk\}} += 2 W_{\{ik\}} \dot{v}_p D_{jp}\phi_i$
            end
            if $j \neq k$ then
                $Td_{\{jk\}} += W_{\{ik\}} \dot{v}_p D_{jp}\phi_i$
            end
            if $j \geq p$ then
                $Td_{\{jp\}} += W_{\{ik\}} \dot{v}_k D_{jp}\phi_i$
            end
        end
    end
    3D Creating
    foreach $p \in P(i)$, $\{j, k\} \subset P(i)$ do
        $Td_{\{jk\}} += \bar{v}_i D_{jkp}\phi_i \dot{v}_p$
    end
    Pushing and creating applied to $W$, see Algorithm 9
    Adjoint Iteration applied to $\bar{v}$, see Algorithm 7
end
Output: $(D^3 f(x) \cdot d)_{jk} = Td_{\{jk\}}$, $D^2 f(x)_{jk} = W_{\{jk\}}$
 for each $j \leq k \in \{1-n, \ldots, 0\}$.
```

4 Numerical experiment

We have implemented the `RevHedir` Algorithm 10 as an additional driver of ADOL-C, a well established AD library coded in C and C++ [27]. We used version ADOL-C-2.4.0, the most recent available.[2] The tests were carried out on a personal laptop with 1.70 GHz dual core processors Intel Core i5-3317U, 4GB of RAM, with the Ubuntu 13.0 operating system.

For those interested in replicating our implementation, we used a sparse undirected weighted graph data structure to represent the matrices W and Td. The data structure is an array of weighted neighbourhood sets, one for each node, where each neighbourhood set is a dynamic array that resizes when needed. Each neighbourhood set is maintained in order and the method used to insert or increment the weight of an edge is built around a binary search.

We have hand-picked fourteen problems from the CUTE collection [28], augmlagn from [29], toiqmerg (Toint Quadratic Merging problem) and chainros_trigexp (Chained Rosenbrock function with Trigonometric and exponential constraints) from [30] for the experiments. We have also created a function

$$\text{heavy\_band}(x, band) = \sum_{i=1}^{n-band} \sin\left(\sum_{j=1}^{band} x_{i+j}\right).$$

For our experiments, we tested `heavy_band`$(x, 20)$. The problems were selected based on the sparsity pattern of $D^3 f(x) \cdot d$, dimension scalability and sparsity. Our goal was to cover a variety of patterns, to easily change the dimension of the function and work with sparse matrices.

In Table 1, the "Pattern" column indicates the type of sparsity pattern: bandwidth[3] of value x (B x), arrow, frame, number of diagonals (D x), or irregular pattern. The "nnz/n" column gives the number of nonzeros in $D^3 f(x) \cdot d$ over the dimension n, which serves as a measure of density. For each problem, we applied `RevHedir` and `edge_pushing` Algorithm 10 and 9 to the objective function $f : \mathbb{R}^n \to \mathbb{R}$, with $x_i = i$ and $d_i = 1$, for $i = 1, \ldots, n$, and give the runtime of each method for dimension $n = 10^6$ in Table 1. Note that all of these matrices are very sparse, partly due to the "thinning out" caused by the high order differentiation. This probably contributed to the relatively low runtime, for in these tests, the run-times have a 0.75 correlation with the density measure "nnz/n". This leads us to believe that the actual pattern is not a decisive factor in runtime.

We did not benchmark our results against an alternative algorithm for we could not find a known AD package that is capable of efficiently calculating such directional derivatives for such high dimensions. For small dimensions, we used the `tensor_eval` of ADOL-C to calculate the entire tensor using univariate forward Taylor series propagation [23]. Then we contract the resulting tensor with the vector

[2] As checked May 28th, 2013.

[3] The bandwidth of matrix $M = (m_{ij})$ is the maximum value of $2|i - j| + 1$ such that $m_{ij} \neq 0$.

Table 1 Description of problem set together with the execution time in seconds of `edge_push` and `RevHedir` for $n = 10^6$

Name	Pattern	nnz/n	`edge_pushing`	`RevHedir`
cosine	B 3	3.0000	2.89	5.25
bc4	B 3	3.0000	3.93	7.87
cragglevy	B 3	2.9981	5.41	10.6
chainwood	B 3	1.4999	4.04	7.22
morebv	B 3	3.0000	4.57	9.44
scon1dls	B 3	0.7002	3.99	8.12
bdexp	B 5	0.0004	2.21	3.86
pspdoc	B 5	4.9999	3.05	5.97
augmlagn	5 × 5 diagonal blocks	4.9998	4.15	9.28
brybnd	B 11	12.9996	14.19	38.79
chainros_trigexp	B 3 + D 6	4.4999	6.51	12.87
toiqmerg	B 7	6.9998	4.33	8.89
arwhead	arrow	3.0000	3.63	6.78
nondquar	arrow + B 3	4.9999	2.9	5.61
sinquad	frame + diagonal	4.9999	5.12	10.01
bdqrtic	arrow + B 7	8.9998	8.98	19.62
noncvxu2	irregular	6.9998	4.95	9.55
heavy_band	B 39	38.9995	20.74	61.27

d. This was useful to check that our implementation was correct,[4] though it would struggle with dimensions over $n = 100$, thus not an appropriate comparison.

Remarkably the time spent by `RevHedir` to calculate $D^3 f(x) \cdot d$ was on average 108% that of calculating $D^2 f(x)$ in the above tests. This means the user could gain third order information for approximately the same cost of calculating the Hessian. The code for these tests can be downloaded as part of a package called `HighOrderReverse` from the Edinburgh Research Group in Optimization website: http://www.maths.ed.ac.uk/ERGO/.

5 Conclusion

Our contribution boils down to a framework for designing high order reverse methods, and an efficient implementation of the directional derivative of the Hessian called `RevHedir`. The framework paves the way to obtaining a reverse method for all orders once and for all. Such an achievement could cause a paradigm shift in numerical method design, wherein, instead of increasing the number of steps or the mesh size, increasing the order of local approximations becomes conceivable. We have also shed

[4] On the function scon1dls, both methods generate different fill-ins that are five orders of magnitude smaller than the remaining entries.

light on existing AD methods, providing a concise proof of the `edge_pushing` [4] and the reverse gradient directional derivative [31] algorithms.

The novel algorithms 4 and 5 for calculating the third-order derivative and its contraction with a vector, respectively, fulfils what we set out to achieve: they accumulate the desired derivative “as a whole”, thus taking advantage of overlapping calculations amongst individual components. This is in contrast with what is currently being used, e.g., univariate Taylor expansions [23] and repeated tangent/adjoint operations [24]. These algorithms can also make use of the symmetry, as illustrated in our implementation of `RevHedir` Algorithm 10, wherein all operations are only carried out on a lower triangular matrix.

We implemented and tested the `RevHedir` with two noteworthy results. The first is its capacity to calculate sparse derivatives of functions with up to a million variables. The second is how the time spent by `RevHedir` to calculate the directional derivative $D^3 f(x) \cdot d$ was very similar to that spent by `edge_pushing` to calculate the Hessian. We believe this is true in general and plan on confirming this in future work through complexity analysis. Should this be confirmed, it would have an immediate consequence in the context of nonlinear optimization, in that the third-order Halley–Chebyshev methods could be used to solve large dimensional problems with an iteration cost proportional to that of Newton step. In more detail, at each step the Halley–Chebyshev methods require the Hessian matrix and its directional derivative. The descent direction is then calculated by solving the Newton system, and an additional system with the same sparsity pattern as the Newton system. If it is confirmed that solving these systems costs the same, in terms of complexity, then the cost of a Halley–Chebyshev iteration will be proportional to that of a Newton step. Though this comparison only holds if one uses these AD procedures to calculate the derivatives in both methods.

The CUTE functions used to test both `edge_pushing` and `RevHedir` are rather limited, and further tests on real-world problems should be carried out. Also, complexity bounds need to be developed for both algorithms.

A current limitation of reverse AD procedures, such as the ones we have presented, is their issue with memory usage. All floating point values of the intermediate variables must be recorded on a forward sweep and kept for use in the reverse sweep. This can be a very substantial amount of memory, and can be prohibitive for large-scale functions [32]. As an example, when we used dimensions of $n = 10^7$, most of our above test cases exhausted the available memory on the personal laptop used. A possible solution is to allow a trade off between run-time and memory usage by reversing only parts of the procedure at a time. This method is called checkpointing [32,33].

References

1. Griewank, A., Walther, A.: Evaluating derivatives, 2nd edn. Society for Industrial and Applied Mathematics (SIAM), Philadelphia (2008)
2. Gebremedhin, A.H., Manne, F., Pothen, A.: What color is your Jacobian? Graph coloring for computing derivatives. SIAM Rev. **47**(4), 629–705 (2005)
3. Griewank, A., Naumann, U.: Accumulating Jacobians as chained sparse matrix products. Math. Progr. **95**(3), 555–571 (2003)

4. Gower, R.M., Mello, M.P.: A new framework for the computation of Hessians. Optim. Methods Softw. **27**(2), 251–273 (2012)
5. Gebremedhin, A.H., Tarafdar, A., Pothen, A., Walther, A.: Efficient computation of sparse Hessians using coloring and automatic differentiation. INFORMS J. Comput. **21**(2), 209–223 (2009)
6. Gutiérrez, J.M., Hernández, M.A.: A family of Chebyshev–Halley type methods in Banach spaces. Bull. Aust. Math. Soc. **55**(01), 113–130 (1997)
7. Amat, S., Busquier, S.: Third-order iterative methods under Kantorovich conditions. J. Math. Anal. Appl. **336**(1), 243–261 (2007)
8. Wang, X., Kou, J.: Semilocal convergence and R-order for modified Chebyshev–Halley methods. J. Numer. Algorithms **64**(1),105–126 (2013)
9. Ezquerro, J.A., Hern, M.A.: New Kantorovich-type conditions for Halley's method. Appl. Numer. Anal. Comput. Math. **77**(1), 70–77 (2005)
10. Xu, X., Ling, Y.: Semilocal convergence for Halley's method under weak Lipschitz condition. Appl. Math. Comput. **215**(8), 3057–3067 (2009)
11. Susanto, H., Karjanto, N.: Newtons methods basins of attraction revisited. Appl. Math. Comput. **215**(3), 1084–1090 (2009)
12. Yau, L., Ben-Israel, A.: The Newton and Halley methods for complex roots. Am. Math. Mon. **105**(9), 806–818 (1998)
13. Gundersen, G., Steihaug, T.: Sparsity in higher order methods for unconstrained optimization. Optim. Methods Softw. **27**(2), 275–294 (2012)
14. Gundersen, G., Steihaug, T.: On diagonally structured problems in unconstrained optimization using an inexact super Halley method. J. Comput. Appl. Math. **236**(15), 3685–3695 (2012)
15. Gundersen, G., Steihaug, T.: On large-scale unconstrained optimization problems and higher order methods. Optim. Methods Softw. **25**(3), 337–358 (2010)
16. Papadimitriou, D.I., Giannakoglou, K.C.: Robust design in aerodynamics using third-order sensitivity analysis based on discrete adjoint. Application to quasi-1D flows. Int. J. Numer. Methods Fluids **69**, 691–709 (2012)
17. Papdimitriou, D.I., Giannakoglou, K.C.: Third-order sensitivity analysis for robust aerodynamic design using continuous adjoint. Int. J. Numer. Methods Fluids **71**, 652–670 (2013)
18. Ozaki, I., Kimura, F., Berz, M.: Higher-order sensitivity analysis of finite element method by automatic differentiation. Comput. Mech. **16**(4), 223–234 (1995)
19. Ederington, L.H., Guan, W.: Higher order greeks. J. Deriv. **14**(3), 7–34 (2007)
20. Kariwala, V.: Automatic differentiation-based quadrature method of moments for solving population balance equations. AIChE J. **58**(3), 842–854 (2012)
21. Corliss, G.F.F., Griewank, A., Henneberger, P.: High-order stiff ODE solvers via automatic differentiation and rational prediction. In: Vulkov, L., Waśniewski, J., Yalamov, P. (eds.) Lecture notes in computer science, pp. 114–125. Springer, Berlin, Heidelberg (1997)
22. Guckenheimer, J., Meloon, B.: Computing periodic orbits and their bifurcations with automatic differentiation. SIAM J. Sci. Comput. **22**(3), 951–985 (2000)
23. Griewank, A., Walther, A., Utke, J.: Evaluating higher derivative tensors by forward propagation of univariate Taylor series. Math. Comput. **69**(231), 1117–1130 (2000)
24. Naumann, U.: The art of differentiating computer programs: An introduction to algorithmic differentiation. Number 24 in Software, Environments, and Tools. SIAM, Philadelphia (2012)
25. Neidinger, R.D.: An efficient method for the numerical evaluation of partial derivatives of arbitrary order. ACM Trans. Math. Softw. **18**(2), 159–173 (1992)
26. Fraenkel, L.E.: Formulae for high derivatives of composite functions. Math. Proc. Camb. Philos. Soc. **83**(02), 159 (1978)
27. Griewank, A., Juedes, D., Utke, J.: ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. ACM Trans. Math. Softw. **22**(2), 131–167 (1996)
28. Bongartz, I., Conn, A.R., Gould, N., Toint, P.L.: CUTE: constrained and unconstrained testing environment. ACM Trans. Math. Softw. **21**(1), 123–160 (1995)
29. Hock, W., Schittkowski, K.: Test examples for nonlinear programming codes. J. Optim. Theory Appl. **30**(1), 127–129 (1980)
30. Luksan, L., Vlcek, J.: Test problems for unconstrained optimization. Technical Report 897, Academy of Sciences of the Czech Republic (2003)

31. Abate, J., Bischof, C., Roh, L., Carle, A.: Algorithms and design for a second-order automatic differentiation module. In: Proceedings of the 1997 International Symposium on Symbolic and Agebraic Computation (ACM), pp. 149–155, New York (1997)
32. Walther, A., Griewank, A.: Advantages of binomial checkpointing for memory-reduced adjoint calculations. In: Feistauer, M., Dolejší, V., Knobloch, P., Najzar, K. (eds.) Numerical Mathematics and Advanced Applications, pp. 834–843. Springer, Berlin, Heidelberg (2004). ISBN: 978-3-642-62288-5. doi:10.1007/978-3-642-18775-9_82
33. Sternberg, J., Griewank, A.: Reduction of storage requirement by checkpointing for time-dependent optimal control problems in ODEs. In: Norris, B., Bücker, M., Corliss, G., Hovland, P., Naumann, U. (eds.) Automatic Differentiation: Applications, Theory, and Implementations, 0, pp. 99–110. Springer, 1 edition (2006)