A sparse matrix approach to reverse mode automatic differentiation in Matlab

Shaun A. Forth*and Naveen Kr. Sharma†‡

International Conference on Computational Science, ICCS 2010,
University of Amsterdam, The Netherlands
May 31 - June 2, 2010

Abstract

We review the extended Jacobian approach to automatic differentiation of a user-supplied function and highlight the Schur complement form's forward and reverse variants. We detail a Matlab operator overloaded approach to construct the extended Jacobian that enables the function Jacobian to be computed using Matlab's sparse matrix operations. Memory and runtime costs are reduced using a variant of the hoisting technique of Bischof (*Issues in Parallel Automatic Differentiation*, 1991). On five of the six mesh-based gradient test problems from *The MINPACK-2 Test Problem Collection* (Averick et al, 1992) the reverse variant of our extended Jacobian technique with hoisting outperforms the sparse storage forward mode of the MAD package (Forth, ACM T. Math. Software. **32**, 2006). For increasing problems size the ratio of gradient to function cpu time is seen to be bounded, if not decreasing, in line with Griewank and Walther's (*Evaluating Derivatives*, SIAM, 2008) cheap gradient principle.

1 Introduction

Automated program generation techniques have been used to augment numerical simulation programs in order to compute derivatives (or sensitivities) of desired simulation outputs with respect to nominated inputs since the 1960's. Such techniques go by the collective name *Automatic*, or *Algorithmic, Differentiation* (AD) [1]. Advances in AD theory, techniques, tools and wide-ranging applications may be found in the many references available on the international website `www.autodiff.org`.

Historically, development has focussed on AD tools for programs written in Fortran and C/C++ as these languages have dominated large simulations for technical computing. AD of Matlab was pioneered by Verma [2] who used the, then new, object-oriented features of Matlab in the overloaded AD package ADMAT 1.0 facilitating calculation of first and second derivatives and determination of Jacobian and Hessian sparsity patterns.

Our experience is that ADMAT's derivative computation run times are significantly higher than expected from AD complexity analysis leading us to develop the MAD package [3] (the recent ADMAT 2.0 may have addressed these issues [4]). Presently MAD facilitates first derivative computations via its `fmad` class, an operator-overloaded implementation of forward mode AD. The `fmad` class's efficiency may be attributed to its use of the optimised `derivvec` class for storing and combining directional derivatives. The `derivvec` class permits multiple directional derivatives to be stored as full or sparse matrices: sparse storage gave good performance on a range of problems.

*Applied Mathematics and Scientific Computing, Department of Engineering Systems and Management, Cranfield University, Shrivenham, Swindon, SN6 8LA, UK

†Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur 721302, West Bengal, India

‡Naveen Kr. Sharma gratefully acknowledges the support of a Summer Internship from the Department of Engineering Systems and Management, Cranfield University.

Bischof et al [5] investigated forward mode Matlab AD by source transformation combined with storing and combining direction derivatives via an overloaded library. This hybrid approach gave a significant performance improvement over ADMAT. Source transformation permits the use of *compile-time* performance optimisations [6, 7]: forward substitution was found particularly effective. Kharche and Forth [8] investigated specialising their source transformation inlining of functions of MAD's `fmad` and `derivvec` classes in the cases of scalar and inactive variables. This was particularly beneficial for small problem sizes for which overloading's run time requirements are dominated by the large relative cost of function call overheads and branching (required for code relevant to scalar and inactive variables) compared to arithmetic operations.

Our thesis is that automated program generation techniques, and specifically AD, should take advantage of the most efficient features of a target language. For example, MAD's `derivvec` class's exploitation [3] of the optimised sparse matrix features of Matlab [9]. As we recap in Sec. 2, it is well known that forward and reverse mode AD may be interpreted in terms of forward and back substitution on the sparse, so-called, extended Jacobian system [1, Chap. 9]. In this article we investigate whether we might use Matlab's sparse matrix features to effect reverse mode AD without recourse to the usual tape-based mechanisms [1, Chap 6.1]. Our implementation, including optimised variants, is described in Sec. 3 and the performance testing of Sec. 4 demonstrates our approach's potential benefits. Conclusions and further work are presented in Sec. 5.

2 Matrix Interpretation of automatic differentiation

Following Griewank and Walther [1], we consider a function $F : R^n \mapsto R^m$ of the form,

$$y = F(x), \tag{1}$$

in which $x \in R^n$ and $y \in R^m$ are the vectors of *independent*, and *dependent*, *variables* respectively. Our aim is to calculate the function Jacobian $F'(x) \in R^m \times R^n$ for any given x from the source code of a computer program that calculates $F(\mathbf{x})$: this is AD's primary task. We consider the evaluation of $F(x)$ as a three-part evaluation procedure of the form,

$$\begin{aligned} v_i &= x_i, & i &= 1, \ldots, n, & (2)\\ v_i &= \varphi_i(v_j)_{j \prec i}, & i &= n+1, \ldots, n+p, & (3)\\ v_i &= v_{j \prec i}, & i &= n+p+1, \ldots, n+p+m. & (4) \end{aligned}$$

In (2) we copy the independent variables x to *internal variables* $v_1, \ldots, v_n$. In (3) each $v_i, i = n+1, \ldots, n+p$ is obtained as a result of p successive *elementary operations* or *elementary functions* (e.g., additions, multiplications, square roots, cosines, etc.), acting on a small number of already calculated v_j. Finally in (4) the appropriate internal variables v_j are copied to the dependent variables in such a way that $y_i = v_{n+p+i}, i = 1, \ldots, m$, to complete the function evaluation.

We define the gradient operator $\nabla = \left(\frac{\partial}{\partial x_1}, \ldots, \frac{\partial}{\partial x_n}\right)$, differentiate (2)-(4) and arrange the resulting equations for the ∇v_i as a linear system,

$$\begin{bmatrix} -I_n & 0 & 0 \\ B & L - I_p & 0 \\ R & T & -I_m \end{bmatrix} \begin{bmatrix} \nabla v_{1,\ldots,n} \\ \nabla v_{n+1,\ldots,n+p} \\ \nabla v_{n+p+1,\ldots,n+p+m} \end{bmatrix} = \begin{bmatrix} -I_n \\ 0 \\ 0 \end{bmatrix}. \tag{5}$$

In (5): I_n is the $n \times n$ identity matrix (I_p and I_m are defined similarly); $\nabla v_{1,\ldots,n}$ is the matrix,

$$\nabla v_{1,\ldots,n} = \begin{bmatrix} \nabla v_1 \\ \vdots \\ \nabla v_n \end{bmatrix},$$

with $\nabla v_{n+1,\ldots,n+p}$ and $\nabla v_{n+p+1,\ldots,n+p+m}$ defined similarly; from (3) the $p \times n$ B and $p \times p$ L are both sparse and contain partial derivatives of elementary operations and assignments; from

(4) the $m \times n$ R and $m \times p$ T are such that $[R\ T]$ contains exactly one unit entry per row. The $(n+p+m) \times (n+p+m)$ coefficient matrix in (5) is known as the *Extended Jacobian* denoted $C - I_{n+p+m}$ and has sub-diagonal entries c_{ij}.

By forward substitution on (5) we see that the system Jacobian $J = \nabla y_{1,\ldots,m} = \nabla v_{n+p+1,\ldots,n+p+m}$ is given by,

$$J = R + T(L - I_p)^{-1}B, \tag{6}$$

the *Schur complement* of $L - I_p$. Using (6), J may be evaluated in two ways [1, p.188]:

1. *Forward variant*: derivatives $\nabla v_{n+1,\ldots,n+p}$ and Jacobian J are determined by,

$$\begin{aligned} (L - I_p)\nabla v_{n+1,\ldots,n+p} &= B, & (7)\\ J &= R + T\nabla v_{n+1,\ldots,n+p}. & (8) \end{aligned}$$

   i.e., forward substitution on the lower triangular system (7) followed by a matrix multiplication and addition (8).

2. *Reverse variant*: the $p \times m$ adjoint matrix $\bar{Z}$ and Jacobian J are determined by,

$$\begin{aligned} (I_p - L^T)\bar{Z} &= T^T, & (9)\\ J &= R + \bar{Z}^T B. & (10) \end{aligned}$$

   i.e., back-substitution of the upper triangular system (9) followed by matrix multiplication and addition (10).

The arithmetic cost of both variants is dominated by the solution of the linear systems (7) and (9) with common (though transposed in (9)) sparse coefficient matrix. These systems have n and m right-hand sides respectively giving the *rule of thumb* that the forward variant is *likely to be preferred* for $n < m$ and reverse for $m > n$ (see [1, p.189] for a counter example). Since matrices B, R and T in (7) to (10) are also sparse, further reductions in arithmetic cost might be obtained by storing and manipulating the v or $\bar{Z}$ as sparse matrices [1, Chap. 7].

Other approaches to reducing arithmetic cost are based on performing row operations to reduce the number of entries in (5) or, indeed, to entirely eliminate some rows [1, Chaps. 9-10]. Such approaches have been generalised by Naumann [10] and may be very efficient if performed at *compilation time* by a source transformation AD tool [11]. In Sec. 3.3 we will adapt Bischof's *hoisting* technique [12] to reduced the size and number of entries of the extended Jacobian: reducing memory and runtime costs. Unlike Bischof, ours is a runtime approach more akin to Christianson et al's [13] *dirty vectors*.

3 Three Implementations

In Secs. 3.2-3.4 we describe our three closely related overloaded classes designed to generate the extended Jacobian's entries as the user's function is executed. First, however, we introduce the `MADExtJacStore` class, objects of which are used by all three overloaded classes to store the extended Jacobian entries.

3.1 Extended Jacobian Storage

A `MADExtJacStore` object has components to store: the number of independent variables, the number of rows of the Extended Jacobian for which entries have been determined, the number of entries determined, and a three column matrix to store the i, j and coefficient c_{ij} for each entry. The number of rows of the matrix component is doubled whenever its current size is exhausted. The class inherits the Matlab *handle class* attribute rendering all assignments of such objects to be pointer assignments allowing us to have a single object of this class accessible to multiple objects of our overloaded classes.

3.2 A First Implementation - `ExtJacMAD`

Figure 1: Example code.

```
function y = F(x)
  y = 2 .* x(2:3) .* x(1);
```

We describe our classes of this and the next three sections with reference to a vector of independent inputs $x = [0.5\ 1\ -2.5]^T$ and the example code of Figure 1 which corresponds to the function $(y_1, y_2) = (2x_2x_1, 2x_3x_1)$.

`ExtJacMAD` class objects have three properties. The first is the `value` property which stores the numeric value of the object and, since Matlab is an array-based language, may be an arbitrary size array. The second is the `row_index` property which is an array of the same size as, and holds the extended Jacobian row index i for each corresponding element of, the object's `value` property. Finally, the `JacStore` component is a *handle* to a `MADExtJacStore` object used to store the extended Jacobian entries.

The statement

```
x = ExtJacMAD([0.5; 1; -2.5])
```

creates an object of `ExtJacMAD` class using the `ExtJacMAD` constructor function to set the three properties of `x`. Firstly, `x.value` is assigned the column vector `[0.5; 1; -2.5]`. Secondly, `x.row_index` is set to the column vector `[1; 2; 3]` indicating that the elements of the `value` component array correspond to the first three rows of the extended Jacobian. Thirdly, `x.JacStore` is set to point to a new object of `MADExtJacStore` class with its array of entries zeroed and both the number of independent variables and the numbers of rows set to three.

Applying the function of Fig. 1, `y = F(x)`, initiates four overloaded operations:

1. Our class's overloaded subscript reference function `subsref` forms a temporary object, call it `tmp1`, with `tmp1 = x(2:3)`. This function simply applies subscripting to the `value` and `row_index` properties of `x` such that `tmp1.value = x.value(2:3)` and `tmp1.row_index = x.row_index(2:3)`. `JacStore` is pointed to the same `MADExtJacStore` object as for `x`. This approach, involving just two subscripting operations and a pointer copy, is intrinsically more efficient than that for MAD's forward mode [3] which involves further overloading on `derivvec` objects and multiple operations required to subscript on their derivative storage arrays.

2. Now `2 .* x(2:3)` is effected as `2 .* tmp1` with the result stored in temporary object `tmp2`. The associated overloaded `times` operation performs `tmp2.value = 2 .* tmp1.value = [2 -5]` and copies the `JacStore` component of `tmp1`. However, the derivatives of `tmp2.value`'s elements are twice those of `tmp1.value` so two new extended Jacobian entries must be made for new rows `i = [4 5]`, columns `j = [2 3]` (i.e., the row indices of `tmp1`), and local derivatives `c = 2`. If `j` or `c` is scalar our software automatically and efficiently replicates them by one-to-many assignments to be the same size as i: thus `c = [2 2]`. The new row indices are assigned `tmp2.row_index = [4 5]`.

3. As in step 1, `tmp3 = x(1)` effects `tmp3.value = x.value(1) = 0.5`, `tmp3.row_index = x.row_index(1) = 1` and `tmp3.JacStore = x.JacStore`.

4. Finally, `y` is formed by the overloaded `times` operation so that: `y.value = tmp2.value .* tmp3.value = [1 -2.5]`; `y.JacStore=tmp2.JacStore`; and another four entries in the extended Jacobian are created. Entries for the `tmp2` argument with `i = [6 7]`, `j = tmp2.row_index = [4 5]`, `c = tmp3.value = 0.5` (expanded to `[0.5 0.5]`) are first stored. These are followed by those for `tmp3` with `i = [6 7]`, `j = tmp2.row_index = [4 5]`, and `c = tmp2.value = [2 -5]`.

As a result of these steps the indices `i = [4 5 6 7 6 7]`, `j = [2 3 4 5 1 1]` and coefficients `c = [2 2 0.5 0.5 2 -5]` of the extended Jacobian's off-diagonal entries have been determined.

We may now obtain the Jacobian of `y` with respect to `x`,

```
J = getJacobian(y)
```

returning,

```
J =
     2     1     0
    -5     0     1
```

as expected. The `ExtJacMAD` class's `getJacobian` function first adds m rows to the extended Jacobian corresponding to a copy of the elements of `y` to ensure the extended Jacobian is of the form (5) with the last m rows linearly independent. It then forms the sparse sub-matrices B, L, R and T before evaluating the Jacobian using one of four subtly different Matlab statements,

```
J  =  R - T *( (L - eye(p)) \ full(B))        (11)
J  =  R - T * ( (L - speye(p)) \ B)           (12)
J  =  R - (full(T) / (L - eye(p))) * B        (13)
J  =  R - (T / (L - speye(p))) * B            (14)
```

Equation (11) uses full storage for the intermediate derivatives $\nabla v_{n+1,\ldots,n+p}$ of the forward substitution (7) and Jacobian J of (8). Approach (12) performs the same operations (7)-(8) using sparse storage. Equations (13) and (14) correspond to full and sparse storage variants respectively of the back-substitution approach of (9)-(10).

3.3 Hoisting - `ExtJacMAD_H`

As an alternative to the computational graph approach used by Bischof et al [12], hoisting can be interpreted as Gaussian elimination using pivot rows with a single off-diagonal entry in the extended Jacobian. Consider the extended Jacobian arising from the example code of Fig. 1 using the approach of Sec. 3.2. As shown in (15), the two sub-diagonal entries of the L block arising from the use of the two elements of the vector variable `tmp1` may be eliminated by row operations for the cost of two multiplications while generating two entries by *fill-in* (though in general fill-in need not occur).

$$C\!-\!I = \left(\begin{array}{ccc|cc|c} -1 & & & & & \\ & -1 & & & & \\ & & -1 & & & \\ \hline & 2 & & -1 & & \\ & & 2 & & -1 & \\ 2 & & & .5 & & -1 \\ -5 & & & & .5 & -1 \\ \hline & & & & 1 & \\ & & & & & 1 \end{array}\right) \begin{array}{c} \text{row 6}\to\text{row 6}+0.5\times\text{row 4} \\ \text{row 7}\to\text{row 7}+0.5\times\text{row 5} \\ \longrightarrow \end{array} \left(\begin{array}{ccc|cc|c} -1 & & & & & \\ & -1 & & & & \\ & & -1 & & & \\ \hline & 2 & & -1 & & \\ & & 2 & & -1 & \\ 2 & 1 & & & & -1 \\ -5 & & 1 & & & \\ \hline & & & & 1 & \\ & & & & & 1 \end{array}\right) \tag{15}$$

In the extended Jacobian, rows 4 and 5 may now be removed from blocks L and B, and columns 4 and 5 may be removed from blocks L and T. This also eliminates two rows from the intermediate matrices $\nabla v_{n+1,\ldots,n+p}$ or $\bar{Z}$ leaving the extended Jacobian as,

$$C - I = \left(\begin{array}{ccc|cc|c} -1 & & & & & \\ & -1 & & & & \\ & & -1 & & & \\ \hline 2 & 1 & & -1 & & \\ -5 & & 1 & & -1 & \\ \hline & & & 1 & & -1 \\ & & & & 1 & \end{array}\right) \tag{16}$$

We may now extract B, L, R and T from (16) and calculate the Jacobian J. Hoisting is an example of a *safe pre-elimination* which never increases the number of arithmetic operations [1, p. 212] but can drastically reduce both these and memory costs. In Matlab hoisting may be applied to element-wise operations or functions with a single array argument (e.g., `-x`, `sin(x)`, `sqrt(x)`) and element-wise binary operations or functions with one inactive argument (e.g., `2 + x`, `A .* x` with `A` inactive). Hoisting is not applicable to matrix operations or functions (e.g, linear solve `X \ Y` or determinant `det(X)`).

We effect hoisting by a run-time mechanism distinguishing our work from Bischof et al [12]. We use a similar technique to that of Christianson et al [13] who used it to reduce forward or back substitution costs on the full extended Jacobian: we reduce the size of the extended Jacobian.

Our hoisted class `ExtJacMAD_H` has an additional property to that of class `ExtJacMAD`, an accumulated extended Jacobian entry array `Cij`. When we initialise our `ExtJacMAD_H` object,

```
x = ExtJacMAD_H([0.5; 1; -2.5])
```

we assign `x.Cij = 1` indicating that the derivatives of the elements of `x` are a multiple of one times those associated with `x.row_index`, i.e., rows 1 to 3. Step 1 of the overloaded operations of Sec. 3.2 is as before but with the additional copy `tmp1.Cij = x.Cij`. Step 2 differs more substantially with no additions to the extended Jacobian, we merely copy `tmp2.row_index = tmp1.row_index` and set `tmp2.Cij = 2*tmp1.Cij`. Step 3 is similar to the revised step 1. Step 4 is modified to account for the objects' accumulated extended Jacobian entry, so when dealing with the entries associated with `tmp2` we have `c = temp2.Cij .* tmp3.value = 1` (expanded to `[1 1]`), and when dealing with `tmp3` we have `c = tmp3.Cij .* tmp2.value = 1 .* [2 -5] = [2 -5]`. The assembled extended Jacobian then directly takes the form (16) and we see that the effects of hoisting have been mimicked at runtime. Note that the `Cij` component is maintained as a scalar whenever possible. Array values of `Cij` would be created if our example function's coding were `y = sin(x(2:3)) .* x(1)` as then `tmp2.Cij = cos(tmp1) = [0.5403 -0.8011]` though this would not prevent hoisting.

3.4 Using Matlab's New Objected Oriented Programming Style

Matlab release R2008a introduced substantially new object oriented programming styles and capabilities compared to those used by both our implementations of Secs. 3.2 and 3.3 and previous Matlab AD packages [2, 3]. Instead of the *old style*'s definition of all an object's methods within separate source files in a common folder, in the *new style* all properties and methods are defined in a single source file.

4 Performance Testing

All tests involved calculating the gradient of a function from the MINPACK-2 Test Problem collection [14] with the original Fortran functions recoded into Matlab by replacing loops with array operations and array functions. We performed all tests for a range of problem sizes n: for each n five different sets of independent variables x were used. For each derivative technique and each problem size the set of five derivative calculations were repeated sufficiently often that the cumulative cpu time exceeded 5 seconds. If a single set of five calculations exceeded 5 seconds cpu time then that technique was not used for larger n. All tests were performed using Matlab R2009b on a Windows XP SP3 PC with 2GB RAM. We first consider detailed results from one problem.

4.1 MINPACK-2 Optimal Design of Composites (ODC) test problem

Table 1 presents the run time ratio of gradient cpu time to function cpu time for the Optimal Design of Composites (ODC) problem for varying number of independent variables n and using the extended Jacobian techniques of Sec. 3. We also give run time ratios for a hand-coded adjoint,

one-sided finite differences (FD) and sparse forward mode AD using MAD's `fmad` class. For these, and all other results presented, AD generated derivatives agreed with the hand-coded technique to within round-off: errors for FD were in line with the expected truncation error. Within our tables a dash indicates that memory or cpu time limits were exceeded.

Table 1: Gradient evaluation cpu time ratio $\text{cpu}(\nabla f)/\text{cpu}(f)$ for the MINPACK-2 Optimal Design of Composites (ODC) test problem.

	$\text{cpu}(\nabla f)/\text{cpu}(f)$ for problem size n				
Grad. Tech	25	100	2500	10000	40000
hand-coded	1.8	1.9	2.0	2.0	1.7
FD	26.2	102.8	2684.0	11147.2	-
sparse forward AD	66.0	55.8	134.7	-	-
Extended Jacobian: Sec. 3.2					
forward full	58.6	79.9	-	-	-
forward sparse	61.3	56.9	62.0	64.7	53.7
reverse full	57.9	51.4	42.1	42.0	35.4
reverse sparse	57.2	57.4	51.2	55.1	48.0
Extended Jacobian + Hoisting: Sec. 3.3					
forward full	46.3	51.4	-	-	-
forward sparse	48.3	42.7	34.0	33.9	28.6
reverse full	47.0	41.0	24.3	23.1	19.9
reverse sparse	44.9	39.9	26.3	26.4	23.3
Extended Jacobian + Hoisting + New Object Orientation: Sec. 3.4					
forward full	99.4	95.2	-	-	-
forward sparse	99.4	83.1	38.4	35.3	28.6
reverse full	100.8	88.6	31.0	26.4	21.0
reverse sparse	98.0	82.1	33.7	29.0	23.8

The hand-coded results show what might be achievable for a source transformation AD tool in Matlab. The FD cpu time ratio is in line with theory ($\approx n+1$) but FD exceeded our maximum permitted run time for large n. Sparse forward mode AD outperformed FD with increasing n but exceeded our PC's 2 GB RAM for larger n.

The extended Jacobian approaches of Sec. 3.2, particularly the reverse variant with full storage, are seen to be competitive with, or outperform, sparse forward AD with the exception of the forward variant with full storage. Since $m \ll n$ we expect the reverse variants to outperform forward and as $m = 1$ there is no point employing sparse storage. Employing the hoisting technique of Sec. 3.3 was always beneficial and for larger problem sizes halved the run time. This is because hoisting reduces the number of entries in the Extended Jacobian by approximately 55% for all problem sizes tested.

Employing Matlab's new object oriented features, described in Sec. 3.4, had a strongly detrimental effect doubling required cpu times compared to using the old features for small to moderate n. The two sets of run times converge for large n because the underlying Matlab built-in functions and operations are identical for both. Matlab's run time overheads must be significantly higher for the new approach and so dominate for small n.

4.2 MINPACK-2 mesh-based minimisation test problem

Table 2 presents selected results for the remaining mesh-based MINPACK-2 minimisation problems. We only present the most efficient reverse full variants of the Extended Jacobian approach. Performance of the FD, hand-coded and new Object Oriented Extended Jacobian approaches was in line with that for the ODC problem of Sec. 4.1.

From Table 2, only for the GL1 problem did sparse forward mode AD outperform the Extended Jacobian approach. For all other problems the Extended Jacobian approach with hoisting gave

Table 2: Gradient evaluation cpu time ratio $\text{cpu}(\nabla f)/\text{cpu}(f)$ for the MINPACK-2 mesh-based minimisation test problems. Ext. Jac. indicates use of the reverse full variant of the Extended Jacobian approach.

		$\text{cpu}(\nabla f)/\text{cpu}(f)$ for problem size n					
Problem	Grad. Tech	25	100	2500	10000	40000	160000
EPT	sparse fwd. AD	102.1	99.6	341.3	-	-	-
	Ext. Jac.	106.4	108.6	120.2	120.8	111.8	60.6
	Ext. Jac. + Hoisting	93.5	98.3	91.2	91.9	86.0	46.1
GL1	sparse fwd. AD	138.0	93.1	13.4	7.5	6.6	7.3
	Ext. Jac.	160.6	104.7	34.3	26.9	27.5	25.6
	Ext. Jac. + Hoisting	115.5	85.6	23.2	18.0	17.9	16.6
MSA	sparse fwd. AD	85.1	69.2	158.7	-	-	-
	Ext. Jac.	82.4	71.3	54.0	60.3	66.0	-
	Ext. Jac. + Hoisting	68.8	57.7	31.5	33.9	39.6	27.9
PJB	sparse fwd. AD	74.7	79.4	-	-	-	-
	Ext. Jac.	61.5	69.7	131.8	71.6	68.5	-
	Ext. Jac. + Hoisting	57.4	60.8	99.8	51.8	48.3	-
		$\text{cpu}(\nabla f)/\text{cpu}(f)$ for problem size n					
Problem	Grad. Tech	100	400	10000	40000	160000	640000
GL2	sparse fwd. AD	86.9	86.7	142.8	292.8	-	-
	Ext. Jac.	76.8	78.6	88.1	66.2	82.8	-
	Ext. Jac. + Hoisting	58.5	59.1	53.3	39.4	50.1	-

equivalent, or substantially faster, performance and used less memory allowing larger problem sizes to be addressed. Hoisting reduced the number of extended Jacobian entries by between 34% (EPT problem) to 49% (MSA problem) leading to significant performance benefits. In all cases, except perhaps the GL2 problem, the Extended Jacobian with Hoisting approach gave run time ratios decreasing with n for large n in line with the *cheap gradient principle* [1, p. 88].

5 Conclusions and further work

Our extended Jacobian approach of Sec. 3 allowed us to use operator overloading to build a function's extended Jacobian before employing Matlab's sparse matrix operations to calculate the Jacobian itself. Bischof's hoisting technique [12] was adapted for run time use to reduce the size of the extended Jacobian. The performance testing of Sec. 4 shows that the reverse variant of our approach with full storage of adjoints and hoisting was substantially more efficient and able to cope with larger problem sizes than MAD's forward mode [3] for five of six gradient test problems from the MINPACK-2 collection [14]. Performance is an order of magnitude worse than for a hand-coded adjoint due to the additional costs of overloaded function calls and of branching between multiple control flow paths in the overloaded functions. Additionally, the tailoring of the hand-coded adjoint's back-substitution to the sparsity of a particular source function's extended Jacobian will likely outperform our use of a general sparse solve.

Future work will address Jacobian problems. Equations (11)-(14) may be employed directly for this together with *compression* [1, Chap. 8]. Compression requires Jacobian-matrix, $JS = RS + T(L - I_p)^{-1}(BS)$, or matrix-Jacobian, $WJ = WR + (WT)(L - I_p)^{-1}B$, products where S and W are so-called *seed matrices*. Hessians can be computed using the `fmad` class to differentiate the extended Jacobian computations: a *forward-over-reverse* strategy [1, Chap. 5].

The extended Jacobian approach is not suited to all problems. For example, the product of two $N \times N$ matrices would create some $2N^3$ extended Jacobian entries. Verma [2] noted that by working at the matrix, and not the element level, just the $2N^2$ elements of the two matrices are needed to enable reverse mode. An under-development tape-based AD implementation will

shortly be compared with this article's extended Jacobian approach.

References

[1] A. Griewank, A. Walther, Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, 2nd Edition, SIAM, Philadelphia, PA, 2008.

[2] A. Verma, ADMAT: Automatic differentiation in MATLAB using object oriented methods, in: M. E. Henderson, C. R. Anderson, S. L. Lyons (Eds.), Object Oriented Methods for Interoperable Scientific and Engineering Computing: Proceedings of the 1998 SIAM Workshop, SIAM, Philadelphia, 1999, pp. 174–183.

[3] S. A. Forth, An efficient overloaded implementation of forward mode automatic differentiation in MATLAB, ACM T. Math. Software. 32 (2) (2006) 195–222. doi:10.1145/1141885.1141888.

[4] Cayuga Research Associates, LLC, ADMAT: Automatic Differentiation Toolbox for use with MATLAB. Version 2.0. (2008).
URL `http://www.math.uwaterloo.ca/CandO_Dept/securedDownloadsWhitelist/Manual.pdf`

[5] C. H. Bischof, H. M. Bücker, B. Lang, A. Rasch, A. Vehreschild, Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs, in: Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002), IEEE Computer Society, Los Alamitos, CA, USA, 2002, pp. 65–72. doi:10.1109/SCAM.2002.1134106.

[6] C. H. Bischof, H. M. Bücker, A. Vehreschild, A macro language for derivative definition in ADiMat, in: H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, B. Norris (Eds.), Automatic Differentiation: Applications, Theory, and Implementations, Lecture Notes in Computational Science and Engineering, Springer, 2005, pp. 181–188. doi:10.1007/3-540-28438-9_16.

[7] H. M. Bücker, M. Petera, A. Vehreschild, Code optimization techniques in source transformations for interpreted languages, in: C. H. Bischof, H. M. Bücker, P. D. Hovland, U. Naumann, J. Utke (Eds.), Advances in Automatic Differentiation, Springer, 2008, pp. 223–233. doi:10.1007/978-3-540-68942-3_20.

[8] R. V. Kharche, S. A. Forth, Source transformation for MATLAB automatic differentiation, in: V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, J. Dongarra (Eds.), Computational Science – ICCS 2006, Vol. 3994 of Lect. Notes Comput. Sc., Springer, Heidelberg, 2006, pp. 558–565. doi:10.1007/11758549_77.

[9] J. Gilbert, C. Moler, R. Schreiber, Sparse matrices in Matlab - design and implementation, SIAM J. Matrix Anal. Appl. 13 (1) (1992) 333–356.

[10] U. Naumann, Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph, Math. Program., Ser. A 99 (3) (2004) 399–421. doi:10.1007/s10107-003-0456-9.

[11] S. A. Forth, M. Tadjouddine, J. D. Pryce, J. K. Reid, Jacobian code generated by source transformation and vertex elimination can be as efficient as hand-coding, ACM T. Math. Software. 30 (3) (2004) 266–299. doi:10.1145/1024074.1024076.

[12] C. H. Bischof, Issues in parallel automatic differentiation, in: A. Griewank, G. F. Corliss (Eds.), Automatic Differentiation of Algorithms: Theory, Implementation, and Application, SIAM, Philadelphia, PA, 1991, pp. 100–113.

[13] B. Christianson, L. C. W. Dixon, S. Brown, Sharing storage using dirty vectors, in: M. Berz, C. Bischof, G. Corliss, A. Griewank (Eds.), Computational Differentiation: Techniques, Applications, and Tools, SIAM, Philadelphia, PA, 1996, pp. 107–115.

[14] B. M. Averick, R. G. Carter, J. J. Moré, G.-L. Xue, The MINPACK-2 test problem collection, Preprint MCS–P153–0692, MCS ANL, Argonne, IL (1992).