Source Transformation for MATLAB Automatic Differentiation

Rahul V. Kharche and Shaun A. Forth

Cranfield University (Shrivenham Campus), Shrivenham, Swindon SN6 8LA, UK
{R.V.Kharche, S.A.Forth}@cranfield.ac.uk

Abstract. We present MSAD, a source transformation implementation of forward mode automatic differentiation for MATLAB. MSAD specialises and inlines operations from the `fmad` and `derivvec` classes of the MAD package. The operator overloading overheads inherent in MAD are eliminated while preserving the `derivvec` class's optimised derivative combination operations. Compared to MAD, results from several test cases demonstrate significant improvement in efficiency across all problem sizes.

1 Automatic Differentiation in MATLAB

MATLAB is popular for rapid prototyping and numerical computing owing to its high-level abstraction of matrices and its rich set of function and GUI libraries. MATLAB's interpreted nature and high-level language make programming intuitive and debugging easy. Optimised BLAS and LAPACK routines for internal matrix operations facilitate good performance. MATLAB may be extended by further general purpose and application specific *toolboxes* (e.g., for optimisation, partial differential equations, control, etc.). We believe the robustness and efficiency of many MATLAB toolboxes and user's applications would benefit from an effective automatic differentiation (AD) [1] package.

Coleman and Verma's ADMAT [2] was the first significant MATLAB AD tool and implemented forward and reverse mode differentiation, with support for Jacobian compression, via operator overloading. The later ADiMat tool [3] adopted a hybrid source transformation/operator overloading implementation of forward mode AD and out-performed ADMAT on several problems. Simultaneously the `fmad` class of MAD [4], an operator overloaded implementation of forward mode AD, was also shown to outperform ADMAT. MAD's efficiency is due to appropriate data-structures and use of high-level matrix operations within its `derivvec` class which holds and propagates derivatives. Use of MATLAB's `sparse` data-type to hold and propagate sparse derivatives enables runtime sparsity exploitation – greatly enhancing performance for problems where sparsity is unknown or difficult to exploit via compression techniques.

Because there is no compilation before execution of operator-overloaded MATLAB code, performance of overloaded implementations of AD suffer due to overheads from the interpreter and the *type check* and *dispatch* mechanism of overloading. Note that MATLAB's recent just-in-time (JIT) compiler is restricted to a subset of MATLAB's intrinsic classes and so is not applicable to the `derivvec` class. Moreover, overloaded operations typically involve substantial logic and branching dependent on the shape (scalar, vector, matrix, N-D array) or storage class (complex, sparse) used for derivatives. For example, consider the `times` operation of the `derivvec` class in Fig. 1. Here `.derivs` refers to the operand's derivative matrix, and `.shape` to the size of the operand. We see that the test on line 9 checks if operands have equal sizes and those of lines 15 and 17 test for scalar operands. Similarly, line 10 checks for `sparse` storage of derivatives. Such tests incur further run-time overheads. Other generic MATLAB overheads are described by [8] and their relevance to AD is discussed in [14].

```
function cdv = times(a,b)
if isa(b,'derivvec')
    cdv = b; mults = a;
else
    cdv = a; mults = b;
end
ssd = prod(cdv.shape); sm = size(mults); ssm = prod(sm);
mults = mults(:);
if ssd == ssm                                             % line 9
    if issparse(cvd.derivs)                               % line 10
        % sparse mode operations omitted for brevity
    else
        cdv.derivs = mults(:,ones(1,cdv.nderivs)).*cdv.derivs; % line 13
    end
elseif ssd == 1                                           % line 15
    cdv.derivs = mults*cdv.derivs; cdv.shape = sm;
elseif ssm == 1                                           % line 17
    cdv.derivs = mults.*cdv.derivs;
end
```

Fig. 1. `derivvec - times` operation from MAD

The MSAD (MATLAB Source transformation AD) tool aims to demonstrate the benefits obtained by combining source transformation with MAD's efficient data structures. An initial, hybrid source transformation/operator overloading approach, similar to that of ADiMat, showed significant speedup compared to MAD for smaller test cases but asymptotically reached the performance of MAD as the problem size increased [6]. Section 2 of this paper describes our improved source transformation approach, which now specialises and inlines all required derivative operations. The benefits of this approach are demonstrated by the test cases of Section 3. Conclusions are presented in Section 4.

2 Source Transformation Via Specialising and Inlining

MSAD uses ANTLR-based LL(k) scanner, parser and tree parsers [5] to analyse and source transform MATLAB programs for AD [6]. Program transformation is carried out via four phases - *scanning* and *parsing* for Abstract Syntax Tree (AST) and symbol table generation, *attribute synthesis* for activity analysis [7], size and class propagation, and finally derivative *code generation*. MSAD's parser recognises the complete MATLAB (Release 14) grammar, but differentiation of code involving branches, loops, structures, cells, nested functions and programs spanning multiple files is currently not implemented. Despite these restrictions, by replacing loops with array operations, many tests cases can be differentiated.

The attribute synthesis phase propagates flags that mark a variable's activity, class, storage type and derivative storage type. Input programs are prepared by using directives to indicate the active inputs and optionally sparse storage for their derivatives. Users may optionally supply size information of input variables. For example, the directives in Fig. 2 indicate to MSAD that the size parameters (`nx, ny`) and the vortex parameter (`vornum`) are scalars. The directives also label variable `x` as an active input and that its derivatives be stored as a sparse matrix. MSAD emulates MATLAB's sparse type propagation and size computation rules for each elementary operation of the source code to deduce the storage type and size of all variables. If a variable's size and storage type cannot be determined, MSAD marks these attributes as unknown. The sizes of scalar and array constants within a program are automatically propagated.

```
function fgrad = gdgl2(nx, ny, x, vornum)
    %! size(nx) = [1, 1], size(ny) = [1, 1], size(vornum) = [1, 1]
    %! active(x), sparseDer(x)
```

Fig. 2. User directives used with gradient function of MINPACK DGL2 problem

The derivative code is generated in a final pass during which the operations from MAD's `fmad` and `derivvec` classes are specialised and inlined. Specialisation uses a variable's size, class, storage class and activity information to resolve condition checks and simplify size computations in the `fmad` and `derivvec` class operations. For variables with unknown size and storage attributes, MSAD conservatively inlines operations involving size and storage checks.

We illustrate the process of specialisation and inlining by considering the FT-BROY function of Fig. 3 [11], specifically the subexpression `(3-2*x(n)).*x(n)` of line 8. Line 3 of the program implies `n` equals the length of the vector `x`. Although this length can be determined only at run-time, `n` can safely be deduced to be a scalar. This further implies `x(n)` is a scalar, as is `3-2*x(n)`. MSAD automatically carries out this size inference during the attribute synthesis phase. During specialisation, because the operands `x(n)` and `3-2*x(n)`, held in variables `tmp_5_` and `tmp_4_` in the generated code of Fig. 4, are inferred to be scalars, the condition on line 9 from the `derivvec-times` operation in Fig. 1 is satisfied. Assuming derivatives are stored in their full form, only lines 8 and 13 from Fig. 1 need to be inserted into the generated code as seen in lines 17 to 20 of Fig. 4. Comments in Fig. 4, and the later Fig. 5, were added by hand to indicate to the reader which line computes which expression or expression's derivatives; `D[a]` denotes the derivatives of variable `a`.

```
function f = ftbroy(x)
%! active(x)
n = length(x);                                                % line 3
p = 7/3; y = zeros(n,1);
i = 2:(n-1);
y(i) = abs((3-2*x(i)) .* x(i) - x(i-1) - x(i+1) + 1).^p;      % line 6
y(n) = abs((3-2*x(n)) .* x(n) - x(n-1) + 1).^p;               % line 7
y(1) = abs((3-2*x(1)) .* x(1) - x(2) + 1).^p;                 % line 8
j = 1:(n/2); z = zeros(length(j),1);
z(j) = abs(x(j) + x(j+n/2)).^p;
f = 1 + sum(y) + sum(z);
```

Fig. 3. FTBROY function

```
tmp_1_ = x(n);                                          %                x(n)
tmp_ind_ = reshape((1:numel(x)), size(x));
tmp_ind_ = tmp_ind_(n);
d_tmp_1_ = d_x(tmp_ind_(:),:);                          %             D[x(n)]
tmp_2_ = 2 .* tmp_1_;                                   %              2*x(n)
tmp_mults_ = 2;
d_tmp_2_ = tmp_mults_(:,ones(1,res_tmp1_)).*d_tmp_1_;   %           D[2*x(n)]
tmp_3_ = 3 - tmp_2_;                                    %            3-2*x(n)
d_tmp_3_ = -d_tmp_2_;                                   %         D[3-2*x(n)]
tmp_4_ = tmp_3_;                                        %          (3-2*x(n))
d_tmp_4_ = d_tmp_3_;                                    %       D[(3-2*x(n))]
tmp_5_ = x(n);                                          %                x(n)
tmp_ind_ = reshape((1:numel(x)), size(x));
tmp_ind_ = tmp_ind_(n);
d_tmp_5_ = d_x(tmp_ind_(:),:);                          %             D[x(n)]
tmp_6_ = tmp_4_ .* tmp_5_;                              %    (3-2*x(n)).*x(n)
tmp_mults_ = tmp_5_;                                    %             line 17
d_tmp_7_ = tmp_mults_(:,ones(1,res_tmp1_)).*d_tmp_4_;   % x(n).*D[(3-2*x(n))]
tmp_mults_ = tmp_4_;
d_tmp_8_ = tmp_mults_(:,ones(1,res_tmp1_)).*d_tmp_5_;   % (3-2*x(n)).*D[x(n)]
d_tmp_6_ = d_tmp_7_ + d_tmp_8_;                         % D[(3-2*x(n)).*x(n)]
```

Fig. 4. MSAD generated derivative code for the subexpression `(3-2*x(n)).*x(n)` of the TBROY function. (Comments added for clarity)

In the subexpression `(3-2*x(i)).*x(i)` on line 6 in Fig. 3, the size of `x(i)` cannot be determined since `i` is a vector dependent on the value of `n`. MSAD therefore conservatively inlines lines 7 to 19 of the `derivvec-times` operation. The first product of `D[3-2*x(i).*x(i)]`, analogous to lines 17 and 18 from Fig. 4, can be seen in Fig. 5.

```
d_tmp_4= d_tmp_3                                           %     D[(3-2*x(i))]
tmp_mults_ = tmp_5_(:);                                    %              x(i)
tmp_ssa_ = numel(tmp_mults_);                              %      length(x(i))
tmp_ssb_ = numel(tmp_4_);                                  % length((3-2*x(i)))
if tmp_ssa_ == tmp_ssb_                                    %       equal sizes
    d_tmp_7_ = tmp_mults_(:,ones(1,res_tmp1_)) .* d_tmp_4_;
elseif tmp_ssb_ == 1                                       %  (3-2*x(i)) scalar
    d_tmp_7_ = tmp_mults_ * d_tmp_4_;
elseif tmp_ssa_ == 1                                       %       x(i) scalar
    d_tmp_7_ = tmp_mults_ .* d_tmp_4_;
end
```

Fig. 5. Additional checks for vector `times` operation in `D[(3-2*x(i))].*x(i)`. (Comments added for clarity)

3 Test Results

MSAD computed derivatives were tested for correctness and performance on several optimisation, BVP and ODE problems [14]. A subset of those tests, all performed using MATLAB Release 14 on a Linux machine with a 2.8 GHz Pentium-4 processor and 512 MB of RAM, are presented here.

In Table 1 we compare use of MSAD and MAD's `fmad` class to compute derivatives by repeating the large-scale test cases from MATLAB's Optimisation Toolbox [11] performed in [4]. The test cases are: `nlsf1a`– sparse Jacobian from vector residual; `brownf`, `tbroyf` – gradient from objective function; `browng`, `tbroyg` – Hessian from hand-coded gradient. Both automatic differentiation tools may use Jacobian/Hessian compression (denoted `cmp`) [1, Chap. 7] or sparse storage (denoted `spr`) [1, Chap. 6] where appropriate. The only MSAD user directives required were those to specify the active input variables and use of sparse derivative storage. For comparison, we have included MATLAB's

Table 1. Ratio CPU($\nabla \boldsymbol{f} + \boldsymbol{f}$)/CPU($\boldsymbol{f}$) – Jacobian/gradient (including function) to function CPU time ratio for given techniques on MATLAB Optimisation Toolbox large-scale examples. (m, n) gives the number of dependents and independents, $\hat{n}$ the maximum number of non-zero entries in a row of the Jacobian and p the number of colours for compression

Problem	CPU($\nabla \boldsymbol{f} + \boldsymbol{f}$)/CPU($\boldsymbol{f}$) for						(m,n)	$\hat{n}$	p
	`Hand-coded`	`sfd-(nls)`	`msad (cmp)`	`fmad (cmp)`	`msad (spr)`	`fmad (spr)`			
`nlsf1a(Jac)`	4.4	38.3	6.9	22.5	19.4	35.1	(100,1000)	3	3
`brownf(grad)`	4.6	1064.9	–	–	9.3	13.7	(1,1000)	1000	–
`browng(Jac)`	5.2	9.5	4.2	8.4	15.3	19.6	(1000,1000)	3	3
`tbroyf(grad)`	3.8	810.7	–	–	8.8	15.9	(1,800)	800	–
`tbroyg(Jac)`	–	13.8	3.3	10.1	15.8	23.5	(800,800)	6	7

finite-difference (`sfd(nls)`) evaluation of the gradient/Jacobian/Hessian and, where available, hand-coding.

Clearly, MSAD yields significant savings compared to `fmad` in like-for-like computation of derivatives for these moderate sized problems ($n \approx 1000$). For compressed derivative computation we get savings of over 50% using `msad(cmp)` and for sparse storage gains of about 30%. Compressed AD (`msad(cmp)`, `fmad(cmp)`) out-performs compressed finite-differencing (`sfd(nls)`). For the gradient problems (`brownf`, `tbroyf`) sparse AD (`msad(spr)`, `fmad(spr)`) is several times faster than `sfd(nls)` because the functions `brownf` and `tbroy` are partially value separable [4] and the sparse derivative computation may utilise intermediate sparsity whereas finite-differencing cannot. For the `browng` problem `msad(cmp)` outperforms hand-coding due to the use of complicated expressions in the hand-coding.

Table 2 lists the total optimisation run-times with derivatives supplied using the methods of Table 1. Source transformed derivatives yield substantial savings in the total run-time compared to `fmad`'s overloading approach and run-times are comparable to those using hand-coded derivatives.

The 2-D Ginzburg-Landau unconstrained minimisation problem (GL2) [12, 13] uses an $n_x \times n_y$ mesh with 4 variables per mesh point yielding $n = 4n_x n_y$ independent variables. The objective function is again partially value separable and the gradient code is supplied. The sparse Hessian is computed as the Jacobian $\mathbf{J}\boldsymbol{g}$ of the gradient $\boldsymbol{g}$. Differentiated functions were generated using MSAD for full and sparse storage of derivatives; the user directives for sparse storage can be seen in Fig. 2. Table 3 gives the derivative computation ratio $\text{CPU}(\mathbf{J}\boldsymbol{g} + \boldsymbol{g})/\text{CPU}(\boldsymbol{g})$ for increasing problem size. Using compression, `msad(cmp)` is nearly 80% more efficient than `fmad(cmp)` for small n. With increasing problem size, the floating point operation cost of the derivative computation of either method increases relative to its overheads and the relative advantage of source transformation decreases. However, even for n as large as 65,536 `msad(cmp)` is nearly twice as fast as overloading. With sparse derivatives (`msad(spr)`, `fmad(spr)`) we see a similar trend but smaller relative improvement due to the common overhead of manipulating MATLAB's `sparse` data structures.

The total optimisation time using MATLAB's `fminunc` solver with the different Hessian calculation techniques of Table 3 is shown in Table 4. The decrease

Table 2. Averaged CPU time for optimisation of the large-scale examples from the MATLAB Optimisation Toolbox with derivatives supplied using given techniques

Problem	Optimisation CPU time (s) for					
	`Hand-coded`	`sfd-(nls)`	`msad (cmp)`	`fmad (cmp)`	`msad (spr)`	`fmad (spr)`
`nlsf1a`	0.16	0.36	0.17	0.31	0.20	0.35
`brownf`	0.56	–	–	–	0.7	1.25
`browng`	0.29	0.56	0.23	0.41	0.46	0.64
`tbroyf`	0.72	–	–	–	1.29	2.89
`tbroyg`	–	0.76	0.20	0.48	0.55	0.86

Table 3. Ratio CPU($\boldsymbol{J}\boldsymbol{g} + \boldsymbol{g}$)/CPU($\boldsymbol{g}$) – Hessian (including gradient) to gradient function CPU time ratio for the MINPACK 2-D Ginzburg-Landau problem using given techniques; p gives the number of colours for compression. For all problem sizes, the maximum number of non-zero entries in a row of the Jacobian is $\hat{n} = 14$.

	CPU($\boldsymbol{J}\boldsymbol{g} + \boldsymbol{g}$)/CPU($\boldsymbol{g}$) for problem size n					
Method	64	256	1024	4096	16384	65536
`msad(cmp)`	24.72	23.69	21.84	23.31	37.95	52.16
`fmad(cmp)`	115.32	105.89	89.57	72.82	72.11	90.47
`msad(spr)`	28.11	29.18	35.02	52.63	88.97	177.10
`fmad(spr)`	122.80	113.84	107.73	108.72	126.45	222.81
`#colours` p	20	23	25	24	25	25

in overall computation time obtained by using MSAD's more efficient derivative computation is seen – but this is not proportional to the decrease in derivative computation time. This is because for larger problem size the number of Newton iterations (which require a Hessian recalculation) stays fixed but the number of conjugate gradient iterations (which do not) increase [15].

Table 4. Optimisation CPU time for the MINPACK Ginzburg-Landau (GL2) problem using MATLAB's `fminunc` with derivatives supplied using given techniques

	Problem size n				
	64	256	1024	4096	16384
Method	CPU time (s) for optimisation				
`msad(cmp)`	0.74	0.59	1.34	6.95	29.62
`fmad(cmp)`	4.45	2.78	3.79	10.71	38.70
`msad(spr)`	1.23	1.14	2.87	13.05	61.93
`fmad(spr)`	4.79	3.32	5.41	17.05	73.83
`sfd`	1.41	1.29	3.60	19.91	216.30

4 Conclusion

The previous, hybrid source transformation/operator overloading implementation of MSAD [6] gave reasonable speedup over operator overloading for small problem sizes. This speedup diminished with increasing problem size. The improved implementation presented here inlines and, where possible specialises, the remaining overloaded function calls. This eliminates the type check and dispatch overhead of overloading, reduces logic and branching, and exposes a larger section of the augmented code to MATLAB's JIT acceleration. Section 3's test cases clearly demonstrate these benefits. Figure 4's code indicates the scope for further performance improvements by eliminating redundant temporaries and common subexpressions. Preliminary results obtained by implementing such improvements by hand on one test case produced a 42% speedup [14] and highlight the need for such compiler-like optimisations within a MATLAB AD-tool.

References

1. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, Penn. (2000)
2. Coleman, T.F., Verma, A.: ADMAT: An automatic differentiation toolbox for MATLAB. Technical report, Computer Science Department, Cornell University (1998)
3. Bischof, C.H., Bücker, H.M., Lang, B., Rasch, A., Vehreschild, A.: 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), Los Alamitos, CA, USA, IEEE Computer Society (2002) 65–72
4. Forth, S.A.: An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. Accepted ACM Trans. Math Softw. (2005)
5. Parr, T., Quong R.: ANTLR: A predicated LL(k) parser generator. Software, Practice and Experience, vol. 25, p. 789, July 1995
6. Kharche, R.V.: Source transformation for automatic differentiation in MATLAB. Master's thesis, Cranfield University (Shrivenham Campus), Engineering Systems Dept., Shrivenham, Swindon SN6 8LA, UK (2004)
7. Bischof, C.H., Carle A., Khademi P., Mauer A.: ADIFOR 2.0: Automatic Differentiation of Fortran 77 Programs. IEEE Computational Science & Engineering **3**(3) (1996) 18–32
8. Menon, V., Pingali, K.: A case for source-level transformations in MATLAB. In: PLAN '99: Proceedings of the 2nd conference on Domain-specific languages, New York, NY, USA, ACM Press (1999) 53–65
9. Rose, L.D., Padua, D.: Techniques for the translation of MATLAB programs into Fortran 90. ACM Trans. Program. Lang. Syst. **21**(2) (1999) 286–323
10. Elphick, D., Leuschel, M., Cox, S.: Partial evaluation of MATLAB. In: GPCE '03: Proceedings of the second international conference on Generative programming and component engineering, New York, NY, USA, Springer-Verlag New York, Inc. (2003) 344–363
11. The MathWorks Inc. 24 Prime Park Way, Natick, MA 01760-1500: MATLAB Optimization Toolbox - User's guide. (2005)
12. Averick, B.M., Moré, J.J.: User guide for the MINPACK-2 test problem collection. Technical Memorandum ANL/MCS-TM-157, Argonne National Laboratory, Argonne, Ill. (1991) Also issued as Preprint 91-101 of the Army High Performance Computing Research Center at the University of Minnesota.
13. Lenton, K.: An efficient, validated implementation of the MINPACK-2 test problem collection in MATLAB. Master's thesis, Cranfield University (Shrivenham Campus), Engineering Systems Dept., Shrivenham, Swindon SN6 8LA, UK (2005)
14. Kharche, R., Forth, S.: Source transformation for MATLAB automatic differentiation. Applied Mathematics & Operational Research Report AMOR 2005/1, Cranfield University (Shrivenham Campus), Engineering Systems Dept., Shrivenham, Swindon, SN6 8LA, UK (2005)
15. Bouaricha, A., Moré, J.J., Wu, Z.: Newton's method for large-scale optimization. Preprint MCS-P635-0197, Argonne National Laboratory, Argonne, Illinois (1997)