Automatic Differentiation and Sparse Matrices

Shaun A. Forth[1] Naveen Kr. Sharma[2]

[1]S.A.Forth@cranfield.ac.uk
Applied Mathematics & Scientific Computation
Cranfield University, Shrivenham, UK

[2]Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur

Sparse Matrices for Scientific Computation:
In Honour of John Reid's 70th Birthday.
15 - 16 July 2009, Cosener's House Abingdon, Oxfordshire

Outline

Introduction - Automatic Differentiation

Griewank and Walther [GW08] state that

Automatic Differentiation

Algorithmic, or automatic, differentiation (AD) is is a growing area of theoretical research and software development concerned with the accurate and efficient evaluation of derivatives for function evaluations given as computer programs.

- AD is not
  1. the finite, or divided difference approximation,
  2. symbolic differentiation (e.g., Maple, Mathematica).
- In contrast, for AD all derivatives:
  1. are calculated without truncation error and frequently with greater efficiency,
  2. stored as floating point numbers and arbitrarily complicated computer programs may be differentiated.

Cranfield UNIVERSITY Shrivenham

Notation and Terminology

We consider the a function of the form:

$$\mathbf{y} = f(\mathbf{x}, \mathbf{a}),$$

with

- independent variables $\mathbf{x} \in \mathbb{R}^n$
- dependent variables $\mathbf{y} \in \mathbb{R}^m$
- function parameters $\mathbf{a}$ (may be from $\mathbb{R}$ or $\mathbb{I}$)

For which we need to calculate the Jacobian Jf,

$$Jf = \left[\frac{\partial y_{j, j=1,\ldots,m}}{\partial x_{i, i=1,\ldots,n}} \right].$$

Forward Mode AD

Consider the example function [FTPR04] $f : \mathbb{R}^3 \rightarrow \mathbb{R}^2$.

Example Function y=f(x,a,b)

```
function y = f(x,a,b)
   w(1) = log(x(1) * x(2));
   w(2) = x(2) * x(3)^2-a;
   w(3) = b * w(1) + x(2)/x(3);
   y(1) = w(1)^2 + w(2) - x(2);
   y(2) = sqrt(w(3)) - w(2);
```

Evaluation Trace (or Code List)

We automatically rewrite f as a sequence of unary or binary operations known as the Evaluation Trace [GW08] or Code List [Ral05].

Example Evaluation Trace

```
function y = f_eval_trace(x,a,b)
  v(1)  = x(1) * x(2);
  v(2)  = log(v(1));
  v(3)  = x(3)^2;
  v(4)  = v(3) * x(2);
  v(5)  = v(4) - a;
  v(6)  = 1 / x(3);
  v(7)  = x(2) * v(6);
        :
```

```
        :
v(8)  = b * v(2);
v(9)  = v(8) + v(7);
v(10) = v(5) - x(2);
v(11) = v(2)^2;
v(12) = sqrt(v(9));
y(1)  = v(11) + v(10);
y(2)  = v(12) - v(5);
```

Differentiated Code

Define a differentiation operator,

$$\mathtt{D} = \left[\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \frac{\partial}{\partial x_3}\right]$$

and differentiate the code list line-by-line,

Forward Mode Differentiated Code

```
function [y,Dy] = Df_eval_trace(x,Dx,a,b)
  Dv(1,:)  = x(1) * Dx(2,:) + Dx(1,:) * x(2);
  v(1)  = x(1) * x(2);
  Dv(2,:)  = (1/v(1)) * Dv(1,:);
  v(2)  = log(v(1));
        :
  Dy(1,:) = Dv(11,:) + Dv(10,:);
  y(1)  = v(11) + v(10);
  Dy(2,:) = Dv(12,:) - Dv(5,:);
  y(2)  = v(12) - v(5);
```

Using the Differentiated Code

```
>> x=[1,2,3]
x =
     1     2     3
>> a=0.5;b=2;
>> Dx=eye(length(x))
Dx =
     1     0     0
     0     1     0
     0     0     1
>> [y,Dy] = Df_eval_trace(x,Dx,a,b)
y =
   15.9805  -16.0672
Dy =
    1.3863    8.6931   12.0000
    0.6979   -8.5347  -12.0775
```

Cranfield UNIVERSITY Shrivenham

Computational Complexity of Forward Mode AD

$$\frac{\text{cost}(Jf\,(\text{forward mode}))}{\text{cost}(f)} \leq 1 + 3n$$

- Where *cost* is the sum of the number of floating point and nonlinear operations.
- c.f. one-sided finite differencing,

$$\frac{\text{cost}(Jf\,(\text{FD}))}{\text{cost}(f)} \approx 1 + n$$

- Griewank and Walther have a more involved result which includes memory operations [GW08].
- Upper bound attained for a function consisting entirely of multiplications.

Implementation

AD algorithms are implemented using either:

- Source Transformation - compiler-like tools read in a users code and produce differentiated code with derivative statements included e.g., ADIFOR [BCH+98], ADIC [BRM97], Tapenade [INR05], TAF [GKS05].
- Operator Overloading - modern programming languages allow a programmer to define their own class/type for which arithmetic operations may be defined so as to propagate derivative information e.g., AD01/AD02 [PR98], ADOL-C [GJU96].

Sparse Storage of Derivatives

- If Jf is sparse (also see later) we might sparse storage for derivative vectors e.g. value-index pairs.
- Used in John Reid's ADO1[PR98] and ADO2 (HSL library `http://www.cse.scitech.ac.uk/nag/hsl/`). [1]
- ADIFOR's SparseLinC library [BKBC96] may be used in Fortran (and C?).
- The MAD package [For06] uses MATLAB's sparse matrices to store derivatives for forward mode AD in MATLAB.

[1] Aside - If you Google John Reid AD01, hit 2 is Victoria Beckham's New Armani Underwear Ad 01.

```
F(1) = (3-2*x(1)).*x(1)-2*x(2) + 1;
i = 2:(n-1);
F(i) = (3-2*x(i)).*x(i)-x(i-1)-2*x(i+1) + 1;
F(n) = (3-2*x(n)).*x(n)-x(n-1) + 1;
```

Jacobian is tridiagonal.

	problem size n				
Jac. Tech	25	100	2500	10000	40000
on	2.7	2.6	2.5	2.6	2.2
FiniteDiff	29.9	107.4	2594.1	-	-
fmad	128.0	121.1	2252.8	-	-
fmadsparse	136.5	109.7	54.4	140.0	711.1

Table: NLSF1 Jacobian cpu time ratio cpu(Jf)/cpu(f). Multiple evaluations for 1 cpu s: timed out (-).

The Jacobian Needn't be Sparse

- If sufficient intermediate variables have sparse derivatives then sparse storage may be advantageous.
- e.g., partially separable cases,

$$f(\mathbf{x}) = \sum_k g_k(x_j, j \in N_k),$$

  with N_k a small subset of $1, 2, \ldots, n$.
- e.g., gradient for the Optimal Design of Composites problem [ACMX92].

Optimal Design of Composites Gradient - Matlab 2008b

	problem size n					
Grad. Tech	25	100	2500	10000	40000	90000
hand-coded	1.9	1.9	2.0	1.7	1.7	2.0
FiniteDiff	27.4	101.9	2641.7	-	-	-
fmad	61.0	128.0	1906.4	-	-	-
fmad(sparse)	64.0	54.5	111.7	x	-	-

Table: ODC Gradient evalution cpu time ratio cpu(∇f)/cpu(f). Multiple evaluations for 1 cpu s: timed out (-), out of memory(x).

Jacobian Compression

- Overhead of manipulating sparse data structure is limiting.
- For Jacobians with known sparsity pattern we may use compression.

Seminal paper by Curtis, Powell and Reid [CPR74]

We divide the columns of J into groups. To form the first group we inspect the columns in turn and include each that has no unknowns in common with those columns already included. The other groups are formed successively by applying the same procedure to those columns not already included in a group.

- Now a research area in itself [GW08, Chap. 8], [GMP05].
- Substantial mathematical framework and numerous other techniques - CPR Compression still very widely used.

NLSF1 from Optimization Toolbox - Matlab 2008b

Jacobian is tridiagonal so 3 groups used.

	problem size n				
Jac. Tech	25	100	2500	10000	40000
on	2.7	2.6	2.5	2.6	2.2
FiniteDiff	29.9	107.4	2594.1	-	-
fmad	128.0	121.1	2252.8	-	-
fmadsparse	136.5	109.7	54.4	140.0	711.1
fmadcmp	128.0	102.9	18.1	10.0	8.2

Table: NLSF1 Jacobian cpu time ratio cpu(Jf)/cpu(f). Multiple evaluations for 1 cpu s: timed out (-).

Extended Jacobian Approaches

We write the linear system for a forward mode AD differentiation as,

$$\begin{bmatrix} -I_n & 0 & 0 \\ B & L - I_p & 0 \\ R & T & -I_m \end{bmatrix} \begin{bmatrix} DX \\ DV \\ DY \end{bmatrix} = \begin{bmatrix} -I_n \\ 0 \\ 0 \end{bmatrix} \tag{1}$$

where the coefficient matrix is the extended Jacobian,

- p is the number of intermediate variables v_i in the evaluation trace.
- The $p \times p$ matrix L is strictly lower triangular.
- And,

$$DX = \begin{bmatrix} Dx_1 \\ \vdots \\ Dx_n \end{bmatrix}, DV = \begin{bmatrix} Dv_1 \\ \vdots \\ Dv_p \end{bmatrix}, DY = \begin{bmatrix} Dy_1 \\ \vdots \\ Dy_m \end{bmatrix}.$$

- Forward mode AD is seen as solving (1) via forward substitution

Cranfield UNIVERSITY

$$\text{Write, } DY = [0\ 0\ I_m] \begin{bmatrix} DX \\ DV \\ DY \end{bmatrix}$$

$$= [0\ 0\ I_m] \begin{bmatrix} -I_n & 0 & 0 \\ B & L - I_p & 0 \\ R & T & -I_m \end{bmatrix}^{-1} \begin{bmatrix} -I_n \\ 0 \\ 0 \end{bmatrix}$$

$$= -[\bar{X}^T\ \bar{V}^T\ \bar{Y}^T] \begin{bmatrix} -I_n \\ 0 \\ 0 \end{bmatrix} = \bar{X}^T,$$

with adjoints $\bar{X}$, $\bar{V}$ and $\bar{Y}$ by back-substitution on the system

$$\begin{bmatrix} -I_n & B^T & R^T \\ 0 & L^T - I_p & T^T \\ 0 & 0 & -I_m \end{bmatrix} \begin{bmatrix} \bar{X} \\ \bar{V} \\ \bar{Y} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -I_m \end{bmatrix}. \tag{2}$$

Computational Complexity of Reverse/Adjoint Mode AD

Computational Complexity of Reverse/Adjoint Mode AD

$$\frac{\text{cost}(Jf(\text{reverse mode}))}{\text{cost}(f)} \leq 1 + 4m$$

- Cheap gradient result - $m = 1$ gives,

$$\text{cost}(Jf(\text{reverse mode})) \leq 5\,\text{cost}(f).$$

- An AD Tool facilitates reverse mode/back-substitution by either:
  - Recomputing intermediates v_i and extended Jacobian entries as required.
  - Storing required values in a forward pass through the code and retrieving them as needed.
  - Hybrid of above.
- Significantly more complex than forward mode tools.

Cranfield UNIVERSITY Shrivenham

Schur Complements

From (1),

$$\begin{bmatrix} -I_n & 0 & 0 \\ B & L - I_p & 0 \\ R & T & -I_m \end{bmatrix} \begin{bmatrix} DX \\ DV \\ DY \end{bmatrix} = \begin{bmatrix} -I_n \\ 0 \\ 0 \end{bmatrix}$$

we see that,

$$J = DY = R + T(I_p - L)^{-1}B, \tag{3}$$

the Schur complement of R.

- Forward substitution approach

$$J = R + T\left[(I_p - L)^{-1}B\right]. \tag{4}$$

- Back substitution approach

$$J = R + \left[T(I_p - L)^{-1}\right]B. \tag{5}$$

MATLAB Implementation

- MATLAB class `ExtJacMAD` with components
  - `value` - stores object's value.
  - `index` - stores row index of object in the extended Jacobian.
  - `jacobian` - handle (MATLAB pointer) to storage for extended Jacobian.
- As user's function is evaluated the extended Jacobian is automatically built up for `y = f(x)`.
- Final call to `getJacobian(y)`:
  - Forms the extended Jacobian as a sparse matrix.
  - Extracts blocks B, L, R and T.
  - Calculates Jacobian via (4) or (5) depending on whether $n \leq m$ or not.

Cranfield UNIVERSITY Shrivenham

ExtJacMAD plus Operation

```
function z = plus(x,y)
% PLUS  Implement obj1 + obj2 for ExtJacMAD
isx = isa(x,'ExtJacMAD');
isy = isa(y,'ExtJacMAD');
if isx && isy
    % Both x and y are of class ExtJacMAD
    z = x; % deep copy to initisalise z
    z.value = x.value + y.value;
    z.index = z.jacobian.n_entry + (1:numel(z.value));
    z.jacobian.n_entry = z.jacobian.n_entry + numel(z.value);
    z.index = reshape(z.index, size(z.value))
    array1 = ones(1,numel(z.index));
    z.jacobian.add_entry(z.index, x.index, array1);
    z.jacobian.add_entry(z.index, y.index, array1);
elseif isx
:
```

NLSF1 from Optimization Toolbox - Matlab 2008b

Jac. Tech	problem size n 25	100	2500	10000	40000
on	2.7	2.6	2.5	2.6	2.2
FiniteDiff	29.9	107.4	2594.1	-	-
fmad	128.0	121.1	2252.8	-	-
fmadsparse	136.5	109.7	54.4	140.0	711.1
fmadcmp	128.0	102.9	18.1	10.0	8.2
ExtJacMAD	150.8	134.9	83.2	86.0	192.0

Table: NLSF1 Jacobian cpu time ratio cpu(Jf)/cpu(f). Multiple evaluations for 1 cpu s: timed out (-).

Optimal Design of Composites Gradient - Matlab 2008b

Grad. Tech	problem size n 25	100	2500	10000	40000	90000
on	1.9	1.9	2.0	1.7	1.7	2.0
FiniteDiff	27.4	101.9	2641.7	-	-	-
fmad	61.0	128.0	1906.4	-	-	-
fmadsparse	64.0	54.5	111.7	x	-	-
ExtJacMAD	70.1	62.8	65.4	81.9	157.2	-

Table: ODC Gradient evalution cpu time ratio cpu(∇f)/cpu(f). Multiple evaluations for 1 cpu s: timed out (-), out of memory(x).

Employing Pre-Eliminations

- The number of entries in the extended Jacobian may become large - can we easily reduce it?
- Consider selected assignments of example problem's evaluation trace:

```
v(1)  = x(1) * x(2);
v(2)  = log(v(1));
v(8)  = b * v(2);
v(9)  = v(8) + v(7);
```

  The corresponding rows of the extended Jacobian are:

	$x(1)$	$x(2)$	$x(3)$	$v(1)$	$v(2)$	$\cdots$	$v(7)$	$v(8)$	$v(9)$
v(1)	$x(2)$	$x(1)$		-1					
v(2)				$\frac{1}{v(1)}$	-1				
v(8)					b			-1	
v(9)							1	1	-1

- There's scope for some Gaussian eliminations here!

Employing Pre-Eliminations (ctd)

- $v(2)$ has a single predecessor

	$x(1)$	$x(2)$	$x(3)$	$v(1)$	$v(2)$	$\cdots$	$v(7)$	$v(8)$	$v(9)$
v(1)	$x(2)$	$x(1)$		-1					
v(2)				$\frac{1}{v(1)}$	-1				
v(8)					b			-1	
v(9)							1	1	-1

Employing Pre-Eliminations (ctd)

- $v(2)$ has a single predecessor - Gaussian eliminate uses of $v(2)$.

	$x(1)$	$x(2)$	$x(3)$	$v(1)$	$v(2)$	$\cdots$	$v(7)$	$v(8)$	$v(9)$
v(1)	$x(2)$	$x(1)$		-1					
v(2)					-1				
v(8)				$\frac{b}{v(1)}$				-1	
v(9)							1	1	-1

- $v(2)$ has a single predecessor - Gaussian eliminate uses of $v(2)$.
- $v(8)$ has a single predecessor

	$x(1)$	$x(2)$	$x(3)$	$v(1)$	$v(2)$	$\cdots$	$v(7)$	$v(8)$	$v(9)$
v(1)	$x(2)$	$x(1)$		-1					
v(2)					-1				
v(8)				$\frac{b}{v(1)}$				-1	
v(9)							1	1	-1

Cranfield UNIVERSITY Shrivenham

Employing Pre-Eliminations (ctd)

- $v(2)$ has a single predecessor - Gaussian eliminate uses of $v(2)$.
- $v(8)$ has a single predecessor - Gaussian eliminate uses of $v(8)$.

	$x(1)$	$x(2)$	$x(3)$	$v(1)$	$v(2)$	$\cdots$	$v(7)$	$v(8)$	$v(9)$
v(1)	$x(2)$	$x(1)$		-1					
v(2)					-1				
v(8)				$\frac{b}{v(1)}$				-1	
v(9)				b			1		-1

- $v(2)$ has a single predecessor - Gaussian eliminate uses of $v(2)$.
- $v(8)$ has a single predecessor - Gaussian eliminate uses of $v(8)$.
- Remove $v(2)$ and $v(8)$ from the system.

	$x(1)$	$x(2)$	$x(3)$	$v(1)$	$v(2)$	$\cdots$	$v(7)$	$v(8)$	$v(9)$
v(1)	$x(2)$	$x(1)$		-1					
v(9)				b			1		-1

Employing Pre-Eliminations (ctd)

- $v(2)$ has a single predecessor - Gaussian eliminate uses of $v(2)$.
- $v(8)$ has a single predecessor - Gaussian eliminate uses of $v(8)$.
- Remove $v(2)$ and $v(8)$ from the system.
- Computational cost is one flop for each eliminated entry - gives a net saving in flops and extended Jacobian storage.

	$x(1)$	$x(2)$	$x(3)$	$v(1)$	$v(2)$	$\cdots$	$v(7)$	$v(8)$	$v(9)$
v(1)	$x(2)$	$x(1)$		-1					
v(9)				b			1		-1

Cranfield UNIVERSITY Shrivenham

Employing Pre-Eliminations (ctd)

- $v(2)$ has a single predecessor - Gaussian eliminate uses of $v(2)$.
- $v(8)$ has a single predecessor - Gaussian eliminate uses of $v(8)$.
- Remove $v(2)$ and $v(8)$ from the system.
- Computational cost is one flop for each eliminated entry - gives a net saving in flops and extended Jacobian storage.
- Key to efficiency - perform eliminations on the fly before assembly.

	$x(1)$	$x(2)$	$x(3)$	$v(1)$	$v(2)$	$\cdots$	$v(7)$	$v(8)$	$v(9)$
v(1)	$x(2)$	$x(1)$		-1					
v(9)				b			1		-1

MATLAB Implementation

- MATLAB class `ExtJacMAD_H` with additional component
  - `entry` - stores entry/coefficient in the extended Jacobian.
- e.g. `index`=3, `entry`=2.3 : object has entry 2.3 in column 3 of the Extended Jacobian.
- Element-wise functions and binary functions with just one active argument do not create entries in extended Jacobian.

sin function

```
function z = sin(x)
z = x;
z.value = sin(x.value);
z.entry = x.entry.*cos(x.value);
```

- Only binary or matrix operations create extended Jacobian entries.

NLSF1 from Optimization Toolbox - Matlab 2008b

Jac. Tech	problem size n 25	100	2500	10000	40000
on	2.7	2.6	2.5	2.6	2.2
FiniteDiff	29.9	107.4	2594.1	-	-
fmad	128.0	121.1	2252.8	-	-
fmadsparse	136.5	109.7	54.4	140.0	711.1
fmadcmp	128.0	102.9	18.1	10.0	8.2
ExtJacMAD	150.8	134.9	83.2	86.0	192.0
ExtJacMAD-H	105.2	96.0	57.6	54.0	55.1

Table: NLSF1 Jacobian cpu time ratio cpu(Jf)/cpu(f). Multiple evaluations for 1 cpu s: timed out (-).

Cranfield UNIVERSITY Shrivenham

Optimal Design of Composites Gradient - Matlab 2008b

Grad. Tech	problem size n 25	100	2500	10000	40000	90000
on	1.9	1.9	2.0	1.7	1.7	2.0
FiniteDiff	27.4	101.9	2641.7	-	-	-
fmad	61.0	128.0	1906.4	-	-	-
fmadsparse	64.0	54.5	111.7	x	-	-
ExtJacMAD	70.1	62.8	65.4	81.9	157.2	-
ExtJacMAD-H	45.0	40.3	33.4	32.0	39.1	35.0

Table: ODC Gradient evalution cpu time ratio cpu(∇f)/cpu(f). Multiple evaluations for 1 cpu s: timed out (-), out of memory(x).

Cranfield UNIVERSITY Shrivenham

- Efficiency of overloaded AD ultimately limited by function call overheads.
- MSAD source transformation tool [KF06] inlines and specialises `fmad` class operations.

Problem	CPU($J, \nabla f$)/CPU(f)					(m, n)
	Hand-coded	msad (cmp)	fmad (cmp)	msad (spr)	fmad (spr)	
nlsf1a(J)	4.4	6.9	22.5	19.4	35.1	(1000,1000)
brownf(∇)	4.6			9.3	13.7	(1,1000)
browng(J)	5.2	4.2	8.4	15.3	19.6	(1000,1000)
tbroyf(∇)	3.8			8.8	15.9	(1,800)
tbroyg(J)		3.3	10.1	15.8	23.5	(800,800)

Table: Jacobian/gradient to function CPU time ratio for MATLAB Optimisation Toolbox largescale examples.

Related Work

- Gaussian eliminate all off-diagonal entries involving intermediates yields the Jacobian with different elimination orderings giving different flop counts [GR91] - vertex elimination [GW08, Chap. 9.3].
- Source transformation makes such techniques efficient [FTPR04].
- John Reid has shown that a poor choice of elimination ordering may result in instability [GW08, Chap. p203].
- Theoretical possibility of fewer flops from eliminating one entry at a time - edge elimination [Nau01].
- So-called face elimination may be more efficient yet [Nau04].
- Griewank has investigated structure-preserving transformations of the Extended Jacobian [GW08, Chap 10.3].
- Implementations based on pivoted LU-factorisation may give better efficiency [PT08]

Conclusions

- Sparse matrix techniques underpin the mathematics behind modern algorithms for automatic differentiation.
- Efficient sparse matrix libraries may even be used to implement automatic differentiation algorithms.
- Much theory is now presented via graphs - Naumann's face elimination only has a graph based interpretation.
- Interplay between the sparse matrix and automatic differentiation communities remains fruitful (Reid, Pothen, . . .)

References

ACMX92: Brett M. Averick, Richard G. Carter, Jorge J. Moré, and Guo-Liang Xue.
The MINPACK-2 test problem collection.
Preprint MCS–P153–0692, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1992.

BCH+98: Christian H. Bischof, Alan Carle, Paul D. Hovland, Peyvand Khademi, and Andrew Mauer.
ADIFOR 2.0 user's guide (revision D).
Technical Memorandum ANL/MCS–TM–192, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1998.

BKBC96: Christian Bischof, Peyvand Khademi, Ali Bouaricha, and Alan Carle.
Efficient computation of gradients and Jacobians by dynamic exploitation of sparsity in automatic differentiation.
Optim. Methods Software, 7:1–39, 1996.

BRM97: Christian H. Bischof, Lucas Roh, and Andrew Mauer.
ADIC — An extensible automatic differentiation tool for ANSI-C.
Software–Practice and Experience, 27(12):1427–1456, 1997.

CPR74: A. R. Curtis, M. J. D. Powell, and J. K. Reid.
On the estimation of sparse Jacobian matrices.
Journal of the Institute of Mathematics and Applications, 13:117–119, 1974.

For06: Shaun A. Forth.
An efficient overloaded implementation of forward mode automatic differentiation in MATLAB.
ACM Trans. Math. Softw., 32(2):195–222, June 2006.
DOI: `http://doi.acm.org/10.1145/1141885.1141888`.

FTPR04: Shaun A. Forth, Mohamed Tadjouddine, John D. Pryce, and John K. Reid.
Jacobian code generated by source transformation and vertex elimination can be as efficient as hand-coding.
ACM Transactions on Mathematical Software, 30(3):266–299, 2004.
DOI: http://doi.acm.org/10.1145/1024074.1024076.

GJU96: Andreas Griewank, David Juedes, and Jean Utke.
Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++.
ACM Transactions on Mathematical Software, 22(2):131–167, 1996.

GKS05: Ralf Giering, Thomas Kaminski, and T. Slawig.
Generating efficient derivative code with TAF: Adjoint and tangent linear Euler flow around an airfoil.
Future Generation Computer Systems, 21(8):1345–1355, 2005.

GMP05: Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen.
What color is your Jacobian? Graph coloring for computing derivatives.
SIAM Review, 47(4):629–705, 2005.

GR91: Andreas Griewank and Shawn Reese.
On the calculation of Jacobian matrices by the Markowitz rule.
In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, chapter 13, pages 126–135. SIAM, Philadelphia, PA, 1991.

GW08: Andreas Griewank and Andrea Walther.
Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.
Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2008.

INR05: INRIA Tropics Project.
TAPENADE 2.0.
`http://www-sop.inria.fr/tropics`, 2005.

KF06: Rahul Kharche and Shaun A. Forth.
Source transformation for MATLAB automatic differentiation.
In Vassil Alexandrov, Dick van Albada, Peter Sloot, and Jack Dongarra, editors, *Computational Science - ICCS 2006 6th International Conference, Reading, UK, May 28-31, 2006, Proceedings, Part IV*, volume 3994 of *Lecture Notes in Computer Science*, pages 558–565. Springer-Verlag, 2006.
DOI=`http://dx.doi.org/10.1007/11758549_77`.

Cranfield UNIVERSITY

Nau01: Uwe Naumann.
Elimination techniques for cheap Jacobians.
In George Corliss, Christèle Faure, Andreas Griewank, Laurent Hascoët, and Uwe Naumann, editors, *Automatic Differentiation: From Simulation to Optimization*, Computer and Information Science, chapter 29, pages 241–246. Springer, New York, 2001.

Nau04: Uwe Naumann.
Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph.
Mathematical Programming, Ser. A, 99(3):399–421, 2004.

PR98: John D. Pryce and John K. Reid.
ADO1, a Fortran 90 code for automatic differentiation.
Technical Report RAL-TR-1998-057, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, OX11 OQX, England, 1998.
See `ftp://ftp.numerical.rl.ac.uk/pub/reports/prRAL98057.ps.gz`.

PT08: John D. Pryce and Mohamed Tadjouddine.
Fast automatic differentiation Jacobians by compact LU factorization.
SIAM Journal on Scientific Computing, 30(4):1659–1677, 2008.

Ral05: Louis B. Rall.
Perspectives on automatic differentiation: Past, present, and future?
In H. Martin Bücker, George F. Corliss, Paul D. Hovland, Uwe Naumann, and Boyana Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, volume 50 of *Lecture Notes in Computational Science and Engineering*, pages 1–14. Springer, New York, NY, 2005.