Shaun A. Forth\textsuperscript{1} \quad Naveen Kr. Sharma\textsuperscript{2}

\textsuperscript{1}\texttt{S.A.Forth@cranfield.ac.uk}
Applied Mathematics & Scientific Computation
Cranfield University, Shrivenham, UK

\textsuperscript{2}\text{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

1. Introduction - Automatic Differentiation
   - Forward Mode AD
2. Sparse Storage of Derivatives
3. Jacobian Compression
4. Extended Jacobian Approaches
5. MATLAB Implementation
6. Employing Pre-Eliminations
7. Source Transformation
8. Related Work and Conclusions
Griewank and Walther [GW08] state that

**Automatic Differentiation**

Algorithmic, or automatic, differentiation (AD) 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.
We consider the a function of the form:

$$y = f(x, a),$$

with

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

For which we need to calculate the Jacobian $J_f$,

$$J_f = \begin{bmatrix} \frac{\partial y_j, j=1,...,m}{\partial x_i, i=1,...,n} \end{bmatrix}. $$
Consider the example function [FTPR04] \( f : \mathbb{R}^3 \rightarrow \mathbb{R}^2 \).

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

```matlab
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);
end
```
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

```matlab
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);
end
```
Define a differentiation operator,

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

and differentiate the code list line-by-line,

**Forward Mode Differentiated Code**

```matlab
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));
    \vdots
    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
```
Computational Complexity of Forward Mode AD

\[
\frac{\text{cost}(J_f(\text{forward mode}))}{\text{cost}(f)} \leq 1 + 3n
\]

Where \( \text{cost} \) is the sum of the number of floating point and nonlinear operations.

- c.f. one-sided finite differencing,

\[
\frac{\text{cost}(J_f(\text{FD}))}{\text{cost}(f)} \approx 1 + n
\]

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

<table>
<thead>
<tr>
<th>Jac. Tech</th>
<th>problem size n</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>25</td>
</tr>
<tr>
<td>on</td>
<td>2.7</td>
</tr>
<tr>
<td>FiniteDiff</td>
<td>29.9</td>
</tr>
<tr>
<td>fmad</td>
<td>128.0</td>
</tr>
<tr>
<td>fmadsparse</td>
<td>136.5</td>
</tr>
</tbody>
</table>

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,

\[ f(x) = \sum_k g_k(x_j, j \in N_k), \]

with \( N_k \) a small subset of 1, 2, \ldots, \( n \).
- e.g., gradient for the Optimal Design of Composites problem [ACMX92].
<table>
<thead>
<tr>
<th>Grad. Tech</th>
<th>25</th>
<th>100</th>
<th>2500</th>
<th>10000</th>
<th>40000</th>
<th>90000</th>
</tr>
</thead>
<tbody>
<tr>
<td>hand-coded</td>
<td>1.9</td>
<td>1.9</td>
<td>2.0</td>
<td>1.7</td>
<td>1.7</td>
<td>2.0</td>
</tr>
<tr>
<td>FiniteDiff</td>
<td>27.4</td>
<td>101.9</td>
<td>2641.7</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>fmad</td>
<td>61.0</td>
<td>128.0</td>
<td>1906.4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>fmad(sparse)</td>
<td>64.0</td>
<td>54.5</td>
<td>111.7</td>
<td>x</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table:** ODC Gradient evaluation cpu time ratio cpu($\nabla f$)/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.
Jacobian is tridiagonal so 3 groups used.

<table>
<thead>
<tr>
<th>Jac. Tech</th>
<th>problem size $n$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>25</td>
</tr>
<tr>
<td>on</td>
<td>2.7</td>
</tr>
<tr>
<td>FiniteDiff</td>
<td>29.9</td>
</tr>
<tr>
<td>fmad</td>
<td>128.0</td>
</tr>
<tr>
<td>fmadsparse</td>
<td>136.5</td>
</tr>
<tr>
<td>fmadcmp</td>
<td>128.0</td>
</tr>
</tbody>
</table>

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,

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

(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.
Write, \( DY \) = 
\[
\begin{bmatrix}
0 & 0 & I_m
\end{bmatrix}
\begin{bmatrix}
DX \\
DV \\
DY
\end{bmatrix}
\]

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

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

with adjoints \( \bar{X} \), \( \bar{V} \) and \( \bar{Y} \) by back-substitution on the system
\[
\begin{bmatrix}
-I_n & B^T & R^T \\
0 & L^T - I_p & T^T \\
0 & 0 & -I_m
\end{bmatrix}
\begin{bmatrix}
\bar{X} \\
\bar{V} \\
\bar{Y}
\end{bmatrix}
= \begin{bmatrix}
0 \\
0 \\
-I_m
\end{bmatrix}.
\]
Computational Complexity of Reverse/Adjoint Mode AD

\[
\frac{\text{cost}(J_f(\text{reverse mode}))}{\text{cost}(f)} \leq 1 + 4m
\]

- Cheap gradient result - \( m = 1 \) gives,

  \[\text{cost}(J_f(\text{reverse mode})) \leq 5 \text{cost}(f)\].

- 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.
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,

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

the Schur complement of \( R \).

- **Forward substitution approach**

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

- **Back substitution approach**

  \( J = R + [T(I_p - L)^{-1}] B. \) (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.
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 initialise 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


<table>
<thead>
<tr>
<th>Jac. Tech</th>
<th>problem size $n$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>25</td>
</tr>
<tr>
<td>on</td>
<td>2.7</td>
</tr>
<tr>
<td>FiniteDiff</td>
<td>29.9</td>
</tr>
<tr>
<td>fmad</td>
<td>128.0</td>
</tr>
<tr>
<td>fmadsparse</td>
<td>136.5</td>
</tr>
<tr>
<td>fmadcmp</td>
<td>128.0</td>
</tr>
<tr>
<td>ExtJacMAD</td>
<td>150.8</td>
</tr>
</tbody>
</table>

**Table:** NLSF1 Jacobian cpu time ratio $\text{cpu}(Jf)/\text{cpu}(f)$. Multiple evaluations for 1 cpu s: timed out (-).
**Table:** ODC Gradient evaluation cpu time ratio $\frac{\text{cpu}(\nabla f)}{\text{cpu}(f)}$. 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:
  
  \[
  v(1) = x(1) \times x(2); \\
  v(2) = \log(v(1)); \\
  v(8) = b \times v(2); \\
  v(9) = v(8) + v(7);
  \]

  The corresponding rows of the extended Jacobian are:

<table>
<thead>
<tr>
<th></th>
<th>x(1)</th>
<th>x(2)</th>
<th>x(3)</th>
<th>v(1)</th>
<th>v(2)</th>
<th>⋯</th>
<th>v(7)</th>
<th>v(8)</th>
<th>v(9)</th>
</tr>
</thead>
</table>
  v(1) |        |       |      | −1   |      |   |      |      |      |
  v(2) | x(1)  |       |      |      |      | 1 |      | −1   |      |
  v(8) |       | x(2)  |      |      |      |   |      |      | b    |
  v(9) |       |       |      |      |      |   |      |      |      | 1    |

- There’s scope for some Gaussian eliminations here!
Employing Pre-Eliminations (ctd)

- $v(2)$ has a single predecessor

\[
\begin{array}{cccccccc}
  x(1) & x(2) & x(3) & v(1) & v(2) & \cdots & v(7) & v(8) & v(9) \\
  v(1) & x(2) & x(1) & -1 & -1 & & & & \\
  v(2) & & & \frac{1}{v(1)} & -1 & b & & & \\
  v(8) & & & & & & b & -1 & \\
  v(9) & & & & & & 1 & 1 & -1
\end{array}
\]
Employming Pre-Eliminations (ctd)

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

\[
\begin{array}{c|ccc|cccc}
\text{ } & x(1) & x(2) & x(3) & v(1) & v(2) & \cdots & v(7) & v(8) & v(9) \\
v(1) & x(2) & x(1) & & -1 & & & & & \\
v(2) & & & & & -1 & & & & \\
v(8) & & & & \frac{b}{v(1)} & & & -1 & & \\
v(9) & & & & & & 1 & 1 & -1 & \\
\end{array}
\]
Employing Pre-Eliminations (ctd)

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

\[
\begin{array}{cccc|cccc}
& x(1) & x(2) & x(3) & v(1) & v(2) & \cdots & v(7) & v(8) & v(9) \\
v(1) & x(2) & x(1) & & -1 & & & & & \\
v(2) & & & -1 & & & & & & \\
v(8) & & & & \frac{b}{v(1)} & & & -1 & & \\
v(9) & & & & & 1 & 1 & -1 & & \\
\end{array}
\]
- \( v(2) \) has a single predecessor - Gaussian eliminate uses of \( v(2) \).
- \( v(8) \) has a single predecessor - Gaussian eliminate uses of \( v(8) \).

\[
\begin{array}{ccc|ccccccc}
  & x(1) & x(2) & x(3) & v(1) & v(2) & \cdots & v(7) & v(8) & v(9) \\
\hline
v(1) & x(2) & x(1) & & -1 & & & & & \\
v(2) & & & & & -1 & & & & \\
v(8) & & & & \frac{b}{v(1)} & & -1 & & & \\
v(9) & & & & b & & 1 & -1 & & \\
\end{array}
\]
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.

\[
\begin{array}{c|ccc|cccc}
& x(1) & x(2) & x(3) & v(1) & v(2) & \cdots & v(7) & v(8) & v(9) \\
\hline
v(1) & x(2) & x(1) & & -1 & & & & & \\
v(9) & & & & b & 1 & & -1 & & \\
\end{array}
\]
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.

$$
\begin{array}{c|ccc|cccc}
& x(1) & x(2) & x(3) & v(1) & v(2) & \cdots & v(7) & v(8) & v(9) \\
\hline
v(1) & x(2) & x(1) & & -1 & & & & & \\
v(9) & & & & & & b & 1 & & -1 \\
\end{array}
$$
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.

\[
\begin{array}{ccc|cccc}
& x(1) & x(2) & x(3) & v(1) & v(2) & \cdots & v(7) & v(8) & v(9) \\
v(1) & x(2) & x(1) & -1 \\
v(9) & & b & 1 & -1 \\
\end{array}
\]
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**

```matlab
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.
## Table: NLSF1 Jacobian cpu time ratio \( \frac{\text{cpu}(Jf)}{\text{cpu}(f)} \).

Multiple evaluations for 1 cpu s: timed out (-).

<table>
<thead>
<tr>
<th>Jac. Tech</th>
<th>problem size ( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>25</td>
</tr>
<tr>
<td>on</td>
<td>2.7</td>
</tr>
<tr>
<td>FiniteDiff</td>
<td>29.9</td>
</tr>
<tr>
<td>fmad</td>
<td>128.0</td>
</tr>
<tr>
<td>fmadsparse</td>
<td>136.5</td>
</tr>
<tr>
<td>fmadcmp</td>
<td>128.0</td>
</tr>
<tr>
<td>ExtJacMAD</td>
<td>150.8</td>
</tr>
<tr>
<td>ExtJacMAD-H</td>
<td>105.2</td>
</tr>
<tr>
<td>Grad. Tech</td>
<td>25</td>
</tr>
<tr>
<td>---------------</td>
<td>-----</td>
</tr>
<tr>
<td>on</td>
<td>1.9</td>
</tr>
<tr>
<td>FiniteDiff</td>
<td>27.4</td>
</tr>
<tr>
<td>fmad</td>
<td>61.0</td>
</tr>
<tr>
<td>fmad sparse</td>
<td>64.0</td>
</tr>
<tr>
<td>ExtJacMAD</td>
<td>70.1</td>
</tr>
<tr>
<td>ExtJacMAD-H</td>
<td>45.0</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Problem</th>
<th>Hand-coded</th>
<th>CPU((J, \nabla f)/CPU(f))</th>
<th>(m, n)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>msad((\text{cmp}))</td>
<td>fmad((\text{cmp}))</td>
</tr>
<tr>
<td><code>nlsf1a(J)</code></td>
<td>4.4</td>
<td>6.9</td>
<td>22.5</td>
</tr>
<tr>
<td><code>brownf(\nabla)</code></td>
<td>4.6</td>
<td>9.3</td>
<td>13.7</td>
</tr>
<tr>
<td><code>browng(J)</code></td>
<td>5.2</td>
<td>4.2</td>
<td>8.4</td>
</tr>
<tr>
<td><code>tbroyf(\nabla)</code></td>
<td>3.8</td>
<td>8.8</td>
<td>15.9</td>
</tr>
<tr>
<td><code>tbroyg(J)</code></td>
<td>3.3</td>
<td>10.1</td>
<td>15.8</td>
</tr>
</tbody>
</table>

**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

The MINPACK-2 test problem collection.  

Christian H. Bischof, Alan Carle, Paul D. Hovland, Peyvand Khademi, and Andrew Mauer.
ADIFOR 2.0 user’s guide (revision D).  

Christian Bischof, Peyvand Khademi, Ali Bouaricha, and Alan Carle.
Efficient computation of gradients and Jacobians by dynamic exploitation of sparsity in automatic differentiation.  

Christian H. Bischof, Lucas Roh, and Andrew Mauer.
ADIC — An extensible automatic differentiation tool for ANSI-C.  

A. R. Curtis, M. J. D. Powell, and J. K. Reid.
On the estimation of sparse Jacobian matrices.  

Shaun A. Forth.
An efficient overloaded implementation of forward mode automatic differentiation in MATLAB.  

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.  
DOI: http://doi.acm.org/10.1145/1024074.1024076.
Andreas Griewank, David Juedes, and Jean Utke.
Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++.

Generating efficient derivative code with TAF: Adjoint and tangent linear Euler flow around an airfoil.

Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen.
What color is your Jacobian? Graph coloring for computing derivatives.

Andreas Griewank and Shawn Reese.
On the calculation of Jacobian matrices by the Markowitz rule.

Andreas Griewank and Andrea Walther.
*Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*.

INRIA Tropics Project.
TAPENADE 2.0.

Rahul Kharche and Shaun A. Forth.
Source transformation for MATLAB automatic differentiation.
DOI=http://dx.doi.org/10.1007/11758549_77.

Uwe Naumann.
Elimination techniques for cheap Jacobians.

Uwe Naumann.

John D. Pryce and John K. Reid.

John D. Pryce and Mohamed Tadjouddine.

Louis B. Rall.
Perspectives on automatic differentiation: Past, present, and future?