Jacobians by Vertex Elimination and Inner Products

ADFest, Univ Hertfordshire, Nov 2004
John Pryce, RMCS, Cranfield University
j.d.pryce@cranfield.ac.uk
with the help of Shaun Forth & Mohamed Tadjouddine

This work is about using AD to generate Jacobian code, to compute Jacobian matrix $J = J(\mathbf{x}) = \mathbf{f}'(\mathbf{x})$ of a vector function $\mathbf{y} = \mathbf{f}(\mathbf{x})$.

- Using source text translation approach.
- Applies to code with no loops or branches. May sound trivial but
  - There are many 'flux functions' in CFD that are like this and are invoked once per mesh cell each time global flux-function is required — millions of times per run.
  - For other computations, think of basic blocks in the code.

This is an alternative way to implement the Vertex Elimination method (VE) of Griewank and Reese (1991). Standard implementations produce code with many, short, statements. This way produces fewer, longer statements of inner-product form

$$c_{ij} := c_{ij} + \sum_{k \in K} c_{ik} c_{kj}.$$

Produces generally faster code than produced by other VE approaches, and more human-readable.

We view the problem in terms of sparse linear algebra. The method is based on the well-known equivalence of Gaussian Elimination with Crout-Doolittle "compact" LU factorization.

Simple example

Illustrate by a simple function $\mathbf{y} = \mathbf{f}(\mathbf{x})$ with three inputs and two outputs. Left column: code list for $\mathbf{f}$, in MATLAB-like notation. Right column: the basic linear relations of AD, obtained by differentiating the code line-by-line:

`function [y1,y2] = f(x1,x2,x3)`				
`v1 = x1*x2`	$\mathrm{d}v_1$	$=$	$x_2\,\mathrm{d}x_1 + x_1\,\mathrm{d}x_2$	
`v2 = sin(v1)`	$\mathrm{d}v_2$	$=$	$\cos(v_1)\,\mathrm{d}v_1$	
`v3 = 2*v2`	$\mathrm{d}v_3$	$=$	$2\,\mathrm{d}v_2$	(1)
`v4 = v3-x1`	$\mathrm{d}v_4$	$=$	$\mathrm{d}v_3 - \mathrm{d}x_1$	
`y1 = x3*v4`	$\mathrm{d}y_1$	$=$	$x_3\,\mathrm{d}v_4 + v_4\,\mathrm{d}x_3$	
`y2 = 3*v4`	$\mathrm{d}y_2$	$=$	$3\,\mathrm{d}v_4$	

d's mean "derivatives wrt. whatever independent variables we are interested in".

Eliminate intermediate $\mathrm{d}v_k$ to get the $\mathrm{d}y_i$ as linear combinations of the $\mathrm{d}x_j$:

$$\mathrm{d}y_i = \sum_j J_{ij}\,\mathrm{d}x_j$$

This gives $J = [J_{ij}]$, the desired Jacobian matrix.

In *classical Forward AD*, d means "gradient wrt. the independent variables". Here,

$$\mathrm{d} = \left(\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \frac{\partial}{\partial x_3}\right)$$

Eliminate the $\mathrm{d}v_k$ by *forward substitution*, taking no account of sparsity in the row vectors on the right.

For the example:

Initialize with

$$
\begin{array}{lllll}
\mathrm{d}x_1 & = & & (1 & 0 \quad 0) \\
\mathrm{d}x_2 & = & & (0 & 1 \quad 0) \\
\mathrm{d}x_3 & = & & (0 & 0 \quad 1)
\end{array}
$$

and continue

$$
\begin{array}{lllll}
\mathrm{d}v_1 & = & x_2\,\mathrm{d}x_1 + x_1\,\mathrm{d}x_2 & = & (x_2 \quad x_1 \quad 0) \\
\mathrm{d}v_2 & = & \cos(v_1)\,\mathrm{d}v_1 & = & (\cos(v_1)\,x_2 \quad \cos(v_1)\,x_1 \quad 0) \\
\mathrm{d}v_3 & = & 2\,\mathrm{d}v_2 & = & \ldots \\
\mathrm{d}v_4 & = & \mathrm{d}v_3 - \mathrm{d}x_1 & = & \ldots
\end{array}
\qquad (2)
$$

and output

$$
\begin{array}{lllll}
\mathrm{d}y_1 & = & x_3\,\mathrm{d}v_4 + v_4\,\mathrm{d}x_3 & = & \ldots \\
\mathrm{d}y_2 & = & 3\,\mathrm{d}v_4 & = & \ldots
\end{array}
$$

(Of course do it numerically, not symbolically.)

If straightforwardly done, costs 12 adds and 24 mults.

Matrix view.

The relations in (1) constitute a sparse linear system

$$\mathbf{0} = \left[\begin{array}{ccc|c|cccc|cc} x_2 & x_1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \cos(v_1) & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ \hline 0 & 0 & v_4 & 0 & 0 & 0 & x_3 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & -1 \end{array}\right] \left[\begin{array}{c} \mathrm{d}x_1 \\ \mathrm{d}x_2 \\ \mathrm{d}x_3 \\ \hline \mathrm{d}v_1 \\ \mathrm{d}v_2 \\ \mathrm{d}v_3 \\ \mathrm{d}v_4 \\ \hline \mathrm{d}y_1 \\ \mathrm{d}y_2 \end{array}\right] \tag{3}$$

$$= \begin{array}{c} \\ p \\ m \end{array} \begin{array}{c} \begin{array}{ccc} n & p & m \end{array} \\ \left[\begin{array}{c|c|c} B & L-I & 0 \\ R & T & -I \end{array}\right] \end{array} \left[\begin{array}{c} \mathrm{d}\mathbf{x} \\ \mathrm{d}\mathbf{v} \\ \mathrm{d}\mathbf{y} \end{array}\right], \text{ in Griewank's notation} \tag{4}$$

That is

$$B\,\mathrm{d}\mathbf{x} + (L - I)\,\mathrm{d}\mathbf{v} = \mathbf{0}$$
$$R\,\mathrm{d}\mathbf{x} + T\,\mathrm{d}\mathbf{v} - \mathrm{d}\mathbf{y} = \mathbf{0}$$

Solving,

$$\mathrm{d}\mathbf{y} = \left(R + T(I - L)^{-1}B\right)\mathrm{d}\mathbf{x},$$

hence Jacobian is given by

$$J = R + T(I - L)^{-1}B \tag{5}$$

Vertex Elimination (VE)

Griewank–Reese–Naumann define VE in terms of the Computational Graph, but it is equivalent to the following process on the matrix

$$C = \begin{array}{c} \\ p \\ m \end{array} \begin{array}{c} \begin{array}{cc} p & n \end{array} \\ \begin{bmatrix} L - I & B \\ T & R \end{bmatrix} \end{array} \tag{6}$$

Algorithm 0.1 (Vertex Elimination)
for each intermediate-variable row $i = 1, \ldots, p$, in some order,
the pivot order

- Remove multiples of row i from later rows, to zero all entries in column i
- Delete row and column i

This is just Gaussian Elimination (GE) on the matrix after permuting rows and columns into pivot order. GE is known to be equivalent to Crout–Doolittle compact LU factorization. Hence, VE can be implemented by LU factorizing, "to p stages only",

$$C^* = \begin{array}{c} \\ p \\ m \end{array} \begin{array}{c} \begin{array}{cc} p & n \end{array} \\ \begin{bmatrix} P(L-I)P^T & PB \\ TP^T & R \end{bmatrix} \end{array} \text{ producing } \begin{array}{c} \\ p \\ m \end{array} \begin{array}{c} \begin{array}{cc} p & n \end{array} \\ \begin{bmatrix} * & * \\ * & \boxed{\begin{array}{l} J \text{ ends} \\ \text{up here} \end{array}} \end{bmatrix} \end{array} \quad (7)$$

where P is a permutation matrix describing the pivot order

In AD applications, many original entries are constants, e.g. the local derivatives for $x \pm y$ are $1, \pm 1$. In addition to using LU-style, the right hand column shows the shorter code that can be produced using this fact.

Code generation for simple example

Entries of matrix are represented by simple variables, e.g. cy1v4 is entry in y1 row, v4 column. Pivot order is 3, 2, 1, 4.

TRADITIONAL VE STYLE

```
!Input: elementary derivatives
!cv1x1, cv1x2, cv2v1, cv3v2, cv4x1,
!cv4v3, cy1x3, cy1v4, cy2v4
!eliminate v3
cv4v2 = cv4v3 * cv3v2
!eliminate v2
cv4v1 = cv4v2 * cv2v1
!eliminate v1
cv4x1 = cv4x1 + cv4v1 * cv1x1
cv4x2 = cv4v1 * cv1x2
!eliminate v4
cy1x1 = cy1v4 * cv4x1
cy2x1 = cy2v4 * cv4x1
cy1x2 = cy1v4 * cv4x2
cy2x2 = cy2v4 * cv4x2
```

1 add, 8 mults

LU STYLE+EXPLOITING CONSTS

```
!Input: non-constant elem derivs
! cv1x1, cv1x2, cv2v1, cy1x3, cy1v4
!Use cv3v2=2, cv4v3=1, cy2v4=3
cv4v1 = 2 * cv2v1
cv4x1 = –1 + cv4v1 * cv1x1
cv4x2 = cv4v1 * cv1x2
cy1x1 = cy1v4 * cv4x1
cy2x1 = 3 * cv4x1
cy1x2 = cy1v4 * cv4x2
cy2x2 = 3 * cv4x2
```

1 add, 7 mults — cf. costs on Slide 3

At end J is formed from $J_{ij} = \text{cy}i\text{x}j$. (Note cy1x3 is input, and cy2x3 is 0).

Jacobians by Inner Products (JIP)

JIP is a prototype written in MATLAB. Input is (1) a representation of the Computational Graph G, produced e.g. by ELIAD tool's Fortran parser; (2) a pivot order π (you choose one of several heuristics to compute π). Output is code in the above "LU style".

Tests on real code showed many (even over 50%) statements generated by this process were simple copies $c_{ij} = c_{rs}$ or $c_{ij} = -c_{rs}$. JIP exploits this to shorten code further — "alias" feature.

Examples

Roe flux function from CFD. 10 inputs, 5 outputs, 62 intermediates. Best heuristic was "pre-eliminated VLR", giving code with 240 lines, 445 add/subtracts, 662 multiplications. (ELIAD: 1222 to 3175 lines depending on pivot order. Slightly unfair as includes 278 lines elementary derivative and function code.)

(Derived from) **Flow in Channel** function from MINPACK test set. 128 inputs, 34 outputs, 582 intermediates. J is sparse with 342 nonzeros out of possible 4352. We get code with one line per nonzero, 352 add/subtracts, 494 multiplications. Without the "alias" feature the code would have been 518 lines longer. (ELIAD: 4420 to 9982 lines.)

technique	$W(\nabla\mathbf{F})/W(\mathbf{F})$	Platform		
		Ultra10	Alpha	AMD
ADIFOR	15.95	17.39	9.74	9.10
FD	12.14	13.32	12.47	9.08
TAMC-ftl	21.18	18.00	10.02	12.34
TAMC-ad	12.69	23.87	8.03	8.97
VE-SLP-R	6.78	9.70	4.24	4.00
VE-SLP-R-DFT	6.78	9.25	4.05	5.69
VE-SLP-Mark	7.35	11.92	4.62	4.17
VE-SLP-VLR	6.60	10.43	4.00	3.90
VE-SLP-VLR-DFT	6.60	8.32	4.36	5.51
LU-SLF	?	9.84	4.92	3.98
LU-SLR	?	8.01	4.62	3.47
LU-SL-Mark	?	9.10	4.44	3.72
LU-SL-VLR	?	8.52	4.06	3.41
LU-SLPF	?	10.52	4.47	3.89
LU-SLPR	?	7.63	3.93	3.38
LU-SLP-Mark	?	8.96	4.40	3.72
LU-SLP-VLR	?	8.80	4.04	3.41

Table 1: Tests on Roe flux

technique	$W(\nabla \mathbf{F})/W(\mathbf{F})$	Platform		
		Ultra10	Alpha	AMD
MINPACK	1.91	13.53	12.55	6.52
HAND-CODED	1.91	9.25	7.17	2.22
HAND-CODED-(UNROLLED)	?	4.75	3.63	1.26
ADIFOR(UNROLLED)	34.42	22.50	42.02	14.59
TAMC-F(UNROLLED)	35.99	21.23	44.38	13.97
TAMC-R	44.64	73.89	73.09	25.87
FD(UNROLLED)	33.73	116.79	172.38	38.49
VE-SL-F	3.49	3.14	3.38	4.92
VE-SL-R	2.25	3.12	3.32	1.82
VE-SLP-F	2.25	3.16	3.33	1.84
VE-SLP-R	2.25	3.14	3.33	1.82
LU-SLF	?	2.98	3.36	1.38
LU-SLR	?	3.09	3.42	1.35
LU-SLMARK	?	3.07	3.37	1.35
LU-SLVLR	?	2.99	3.35	1.37
LU-SLPF	?	3.12	3.34	1.37
LU-SLPR	?	3.10	3.34	1.37

Table 2: Tests on FIC problem

Q & A

What's this technique good for?

It applies to the same kind of code as ELIAD tool handles — say up to 1000 lines of Fortran. You must unroll any loops. Branches could, in principle, be handled as ELIAD presently does.

The Cranfield AD group intend to include this code generation method in ELIAD in due course.

What are the algorithms?

Standard graph manipulation — which MATLAB is good at because of its sparse matrix features. Combination of symbolic/numerical linear algebra — done by overloading the Matrix-Multiply operator on a new data type using MATLAB's OO features.

Can't compiler optimization "exploit constants and aliases" just as well?

Our optimizations are fairly specific to AD code. Speed tests on several workstations showed J-code without "constant and alias" optimization generally runs a lot slower than J-code with it, even at highest level of compiler optimization.