Second Order Derivatives with ADTAGEO

Algorithmic Differentiation Through Automatic Graph Elimination Ordering

Andreas Griewank   Jan Riehme

Institute for Applied Mathematics
Humboldt Universität zu Berlin
{griewank,riehme}@math.hu-berlin.de

15th April 2005
Automatic Differentiation Workshop
Nice, France

Outline

Second Order Derivatives with ADTAGEO

ADjoints and TAngents by Graph Elimination Ordering

Andreas Griewank   Jan Riehme

Institute for Applied Mathematics
Humboldt Universität zu Berlin
{griewank,riehme}@math.hu-berlin.de

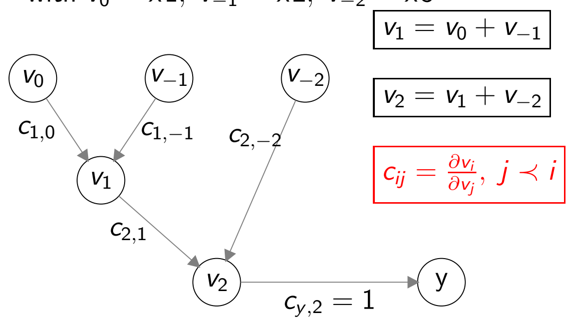
15th April 2005
Automatic Differentiation Workshop
Nice, France

ADTAGEO Gradient-Mode – Example

Computational graph of statement:

`y = x1 + x2 + x3;`

with $v_0 = x1,\ v_{-1} = x2,\ v_{-2} = x3$

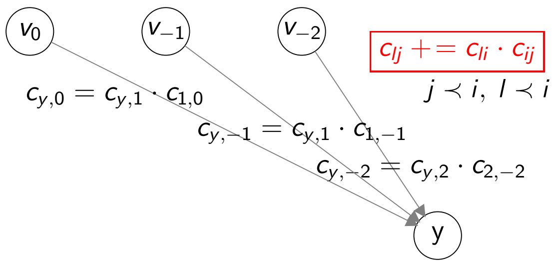


ADTAGEO Gradient-Mode – Elemination

After execution of the assignment:

Elimination of Intermediates:

```
y = x1 + x2 + x3;
```



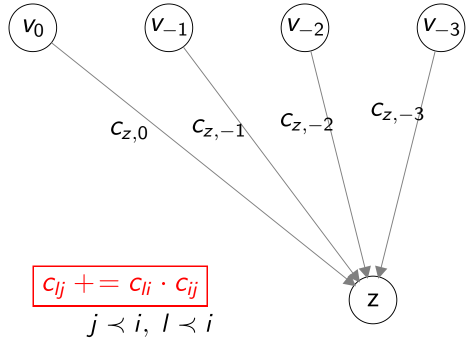
ADIFOR: Statement Level Reverse

AD-enabled NAGWare Fortran 95 compiler

ADTAGEO Gradient-Mode – Elemination

Program: y . . . local variable, leaving scope of y

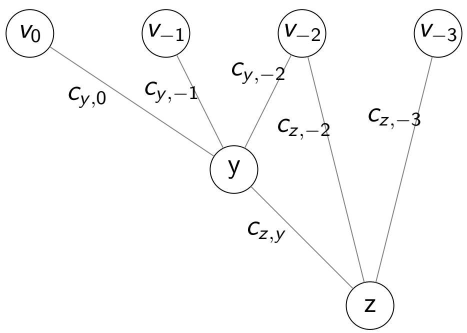
```
{ double y = x1 + x2 + x3;   z = x3 + x4 + y; }
```



ADTAGEO Gradient-Mode – Elemination

Program: y . . . local variable, inside scope of y

```
{ double y = x1 + x2 + x3;   z = x3 + x4 + y; }
```



ADTAGEO at a glance – The idea behind

- More talking about an **IDEA** than a another AD-*TOOL*
- A **new way** of doing Algorithmic Differentiation
- Do not build the computational graph of complete (sub)programs

  Instead:

Maintain a Life -DAG

- Eliminate as soon as possible as many vertexes as possible.
- Eliminate on the fly, Online elimination.
- DAG represents the active variables alive at any one time.

  → Small graph – Huge memory savings (gradients: factor 100)

ADTAGEO at a glance – Requirements

ADTAGEO performs vertex elimination whenever

(i) An active variable is deallocated/destroyed
(ii) An active variable is overwritten

Perfect fitting into OOP scenario

(i) is covered by Destructor (assuming it exists in language)
(ii) is covered by assignment operator

Implementation

- Proof of concept
  - optimized for understanding
  - not optimized for speed
- Implemented in C++
- Heavy use of class `map` from the *Standard Template Library* to store partials **locally** at every node (edges in graph)
- Rapid prototyping (First Order):
  - 140 lines of code for +-*/ and sin, cos, exp
  - One week (with basic testing)
- Any new operator / intrinsic requires 4 lines
  (2 lines for open and closing curly braces)
- Rapid prototyping – **Hessian**:
  - 100 additional lines of code for Hessian elimination
  - One additional day (plus two nights)

ADTAGEO – And Sourcetransformation

- Requirements of ADTAGEO??
  (i) Recognise leaving of the scope of variables (deallocation)
  (ii) Recognise assignments (overwrites)
- Produce source code for graph manipulations
- therefore: one have access to the storage associated with pointers at runtime
  - no pointer aliasing problem
  - DEALLOCATE becomes your best friend: Eliminate all array elements at once opens possibility to optimise the elimination order
- Elements of arrays are handled as single entities
  - partial overwrites are no topic

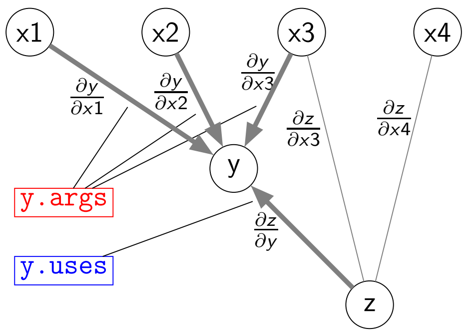
Implementation – DAGLAD

```
class daglad{
private:
 double val;                                            //function value
 map<daglad*, double> args;              //arguments = incoming edges
 map<daglad*, double> uses;                 //used by = outgoing edges
public:
 daglad() { ...};                                          //constructor
 void eliminate() {...};                        //eliminate current vertex
 ~daglad() { eliminate(); ...};                             //destructor
 void operator = (...)  { eliminate(); ...};     // asgnm.
 friend dagdoub operator + (...);       // arithmetic operators
 friend double operator % (...); ...               // retrieval op
}; /* class daglad */
```

Implementation – DAGLAD

Program:

```
y = x1 + x2 + x3;   z = x3 + x4 + y;
```



Implementation – Example

```
#include "daglad.hpp"
main(){
  daglad x1(0.5), x2(1.3), y;
  double xx1, xx2, yy, dy, dyy;

  y = exp(x1)*sin(x1+x2);                          // compute f(x)
  dyy = y%x1;                          // first element of gradient

  xx1 = x1.val(); xx2 = x2.val();                  //shortcuts
  dy = exp(xx1)*(sin(xx1+xx2)+cos(xx1+xx2));
  cout << " dF1 = " << dyy << " diff " << (dyy-dy) << endl;
  dyy = y%x2;                         // second element of gradient
  dy = exp(xx1)*cos(xx1+xx2);
  cout << " dF1 = " << dyy << " diff " << (dyy-dy) << endl;
  cout << " x1 = " << x1 << endl << " x2 = " << x2 << endl;
  cout << " y = " << y << endl;
}
```


Implementation – Usage (prototype)

- Easy mode:
  - Redeclare (required) variables to be of type `daglad`
  - Retrieve first order derivatives somewhere in the code using the % operator

$$\mathtt{y[j]\%x[i]} \equiv \frac{\partial y_j}{\partial x_i}$$

- Advanced mode:
  - Check/prepare/write code for better performance
    Right mixture of forward and reverse mode [see below]

Implementation – Example Output (reformatted)

```
dF1 = 1.23101 diff 2.22045e-16
dF2 = -0.374593 diff 0
x1 = |1,1:0,0.5,3, args={} , uses={[3,4,0,1.23101]}|
x2 = |2,1:0,1.3,2, args={} , uses={[3,4,0,-0.374593]}|
y = |3,1:4,1.6056,0,
    args={[2,0,2,-0.374593][1,0,3,1.23101]}
    uses={} |
```

Implementation – Highlights

- No specification of independents/dependents
- No call of forward / reverse sweeps
  mode is defined by variable allocation
- No tape, No top level routine
- Access to derivatives everywhere (Correctness of derivatives has to be ensured)
- Graph represents the sparsity structure
  - BUT: ADTAGEO is not only sparsity propagation
  - ADTAGEO computes derivatives in sparse mode, therefore no structural zeros are computed
  - Avoid propagation of a seed matrix / directions / ...
  - Avoid Jacobian compression

Implementation – Storing edges locally

Benefits of storing the edges locally

```
for (int i = 0; i < N; i++ )
    y = y*x1*x2*sin(x1)*x1+x2*sin(x1)*x2+x2;
```

N	100.000			250.000		
	CPU	SYS	ELP	CPU	SYS	ELP
map	7.19	0.63	7.85	19.22	2.40	72.00
hash-map	5.53	0.60	6.17	12.87	1.40	14.50
local	2.30	0.00	2.35	5.77	0.00	5.89

Implementation – Memory consumption

```
y[1] = 0;
for( i = 0; i < 100000; i++ ) {
     y[0] = y[1] + x[0] + x[1];
     y[1] = y[0] + x[0] + x[1];
     y[0] = x[0] + x[1];
}
```

complete DAG 82 Megabyte
ADTAGEO 880 Kilobyte

It is a tiny, but perfect example for ADTAGEO

- It is in fact a small **gather-scatter-loop** !!
- Eliminate instead of storing or recompute!

Implementation – Cache behavior (n=250.000)

```
for (int i = 0; i < N; i++ )
    y = y*x1*x2*sin(x1)*x1+x2*sin(x1)*x2+x2;
```

	major	minor
	page faults	
map	6.817	188.676
hash-map	–	≈ 70.000
local	–	≈ 300

Implementation – Mixing Forward and Reverse

Talking about the loop in Speelpennings example

```
void speelforw( int dim, daglad* x, daglad& y ) {
   y = 1;                                        // initialise
   for ( int i = 0; i < dim; i++ )         // loop over elements
      y = y * x[i];                          // compute product
}                                              // end of speelforw
```

Hybrid mode

- Split loop into chunks of C elements
  $\Longrightarrow$ spent small amount of additional memory (compared with forward)
- Loop over chunks
- Deallocate / Eliminate inside of loop over chunks

Runtime Forward / Reverse / R-Split / F-Split

Size of chunks: C = 100

N	1.000	2.500	5.000	10.000	25.000	50.000	100.000
Forward	1.9	14.8	62.5	–	–	–	–
Reverse	0.0	0.0	0.1	0.2	0.5	0.9	1.9
R-Split	0.0	0.1	0.7	2.8	17.3	70.0	280.1
F-Split	0.0	0.3	1.1	3.6	19.3	73.9	286.9

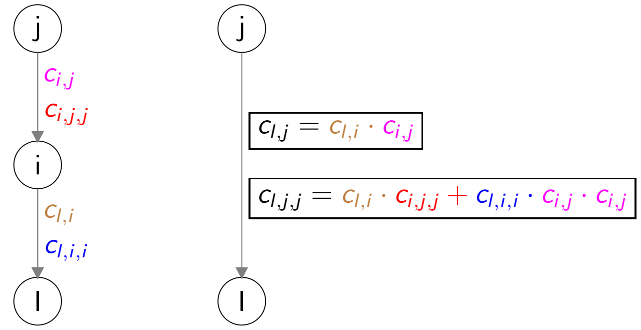
Notes:

- Surprising runtime behavior of Forward Split mode
- Memory used: Reverse 32MB R-Split 11MB F-Split 19MB

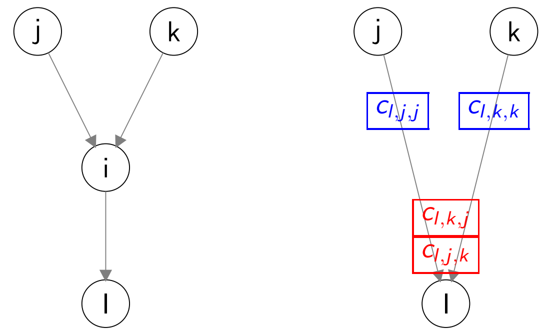
Hessian Elimination – Simplest Case

Looking at a graph snippet, only dealing with

$$c_{i,j} = \frac{\partial v_i}{\partial v_j} \qquad c_{i,j,k} = \frac{\partial^2 v_i}{\partial v_j \partial v_k}$$



Hessian Elimination – Becoming more general



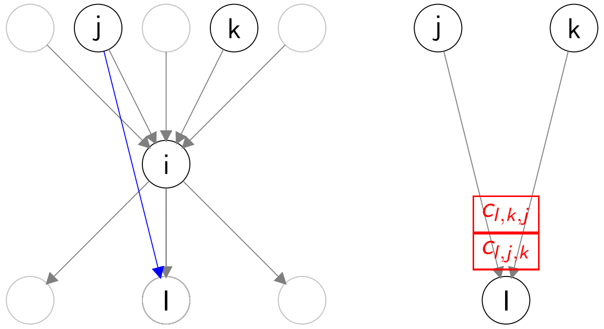
$$c_{l,j,j} = c_{l,i} \cdot c_{i,j,j} + c_{l,i,i} \cdot c_{i,j} \cdot c_{i,j}$$

$$c_{l,j,k} = c_{l,i} \cdot c_{i,j,k} + c_{l,i,i} \cdot c_{i,j} \cdot c_{i,k}$$

Hessian Elimination – Even more general

$$\boxed{c_{l,j,k} \mathrel{+}= c_{l,i} \cdot c_{i,j,k} + c_{l,i,i} \cdot c_{i,j} \cdot c_{i,k}}$$

$j \prec i,\ k \prec i,\ i \prec l$



Hessian Elimination – Summary

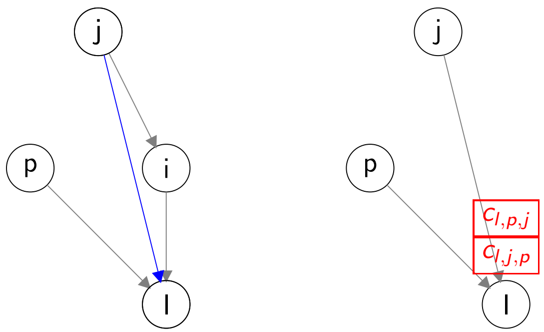
$j \prec i,\ k \prec i,\ i \prec l$:

$$\boxed{c_{l,j,k} \mathrel{+}= c_{l,i} \cdot c_{i,j,k} + c_{l,i,i} \cdot c_{i,j} \cdot c_{i,k}}$$

$j \prec i,\ i \prec l,\ p \prec l,\ p \neq i$:

$$\boxed{\begin{array}{l} c_{l,p,j} \mathrel{+}= c_{l,p,i} \cdot c_{i,j} \\ c_{l,j,p} \mathrel{+}= c_{l,i,p} \cdot c_{i,j} \end{array}}$$

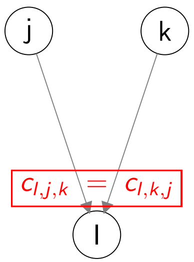
Hessian Elimination – Even more general



$$\boxed{c_{l,p,j} \mathrel{+}= c_{l,p,i} \cdot c_{i,j}}$$

$j \prec i,\ i \prec l,\ p \prec l,\ p \neq i$

Hessian Elimination – What's about Hessian Symmetry?



Can be exploited with canonicalised keys:

$$\boxed{(j,k) \equiv c_{l,j,k} \quad \textit{always fulfills} \quad j \geq k}$$

Hessian Elimination – Symmetric Elimination

$j \prec i,\ k \prec i,\ \boxed{j \geq k},\ i \prec l$:

$$\boxed{c_{l,j,k} \mathrel{+}= c_{l,i} \cdot c_{i,j,k} + c_{l,i,i} \cdot c_{i,j} \cdot c_{i,k}}$$

$j \prec i,\ i \prec l,\ p \prec l,\ p \neq i$:

if($\boxed{p \neq j}$)

$$\boxed{c_{l,p,j} \mathrel{+}= c_{l,p,i} \cdot c_{i,j}}$$

else

$$\boxed{c_{l,p,p} \mathrel{+}= 2 \cdot c_{l,p,i} \cdot c_{i,p}}$$

Hessian Elimination – Hessian Example Output (reformatted)

```
dF1 = 1.23101 difference 2.22045e-16
dF1 = -0.374593 difference 0
x1 = |1,1:0,0.5,3, args={} , uses={[3,4,0,1.23101]} |
x2 = |2,1:0,1.3,2, args={} , uses={[3,4,0,-0.374593]} |
y = |3,1:4,1.6056,0,
    args={[2,0,2,-0.374593][1,0,3,1.23101]} ,
    uses={} ,
    hessian={[(5,6),1],           // BUG has to be removed
            [(2,2),-1.6056],
            [(1,4),-0.374593],    // BUG has to be removed
            [(1,5),1.64872],      // BUG has to be removed
            [(1,2),-1.9802],
            [(1,1),-0.749186], }|
```


Hessian Elimination – Hessian Example

```
#include "daglad.hpp"
main(){
  daglad x1(0.5), x2(1.3), y;
  double xx1, xx2, yy, dy, dyy;

  y = exp(x1)*sin(x1+x2);                    // compute f(x)
  dyy = y%x1;                       // first element of gradient

  xx1 = x1.val(); xx2 = x2.val();              //shortcuts
  dy = exp(xx1)*(sin(xx1+xx2)+cos(xx1+xx2));
  cout << " dF1 = " << dyy << " diff " << (dyy-dy) << endl;
  dyy = y%x2;                      // second element of gradient
  dy = exp(xx1)*cos(xx1+xx2);
  cout << " dF1 = " << dyy << " diff " << (dyy-dy) << endl;
  cout << " x1 = " << x1 << endl << " x2 = " << x2 << endl;
  cout << " y = " << y << endl;
}
```


Hessian implementation – Easy part

- `map<pair<daglad*,daglad*>,double> hessian;`
  to store existing Hessian elements at node / active variable
- add additional parameters for Hessian elements to constructors (2 places)
- extend operators and intrinsics

```
daglad sin (const daglad &a)
{                              // has hessian: -sin(a) = -t
  double t = sin(a.val);
  return daglad( t, a, cos(a.val),
    true, -t
  );
};
```

Hessian implementation – Easy part

- extend operators and intrinsics (cont'd)

```
// daglad * daglad
daglad operator * (const daglad &a,
                   const daglad &b)
{                                //has hessian: [ 0 1; 1 0]
  return daglad( a.val * b.val, a, b.val, b, a.val,
                 true, 0, 1, 0
               );
};
```


Hessian implementation – Not so easy part

- extend `eliminate()` to deal with hessians based on the elimination rules seen

Overall changes on prototype to got Hessians

- roughly 100 lines of code added
- 80% in `eliminate()`

Outlook – Todo

Hessian retrieval – User interface

- Complete Hessians
- Hessian - Vector - Products

Bugfix

- Delete all Hessian elements storing derivatives with respect to eliminated nodes
  - Problems arises from the += if the corresponding variable is overwritten

Outlook – Future research

- Detect and exploit partial separability
- Propagate residuals

$$R \to 0 \quad \Longleftrightarrow \quad (A * R)' = \underbrace{A'R}_{\to 0} + AR' = AR'$$

- Performance Analysis

Outlook – ADTAGEO $\longrightarrow$ ALLEGRO

Making prototype faster:

- Instant elimination: reduce number of vertexes
  - Easy for unary operators
  - Open question: How to avoid copy/delete in DAG?
- Replace `maps` by `hashmap`, attemp to avoid use of `STL`
- Elimination of LHS intermediates in assignments already
  - Never more than 2 edges for intermediate vertexes
    $\longrightarrow$ Specialised class for intermediate vertexes
  - Statement Level Reverse Mode ala `ADIFOR`

Conclusions

We have seen (Pros)

- A new view to AD, strongly based **Life-DAG**
- Easy to implement
- Convenient to use (at least C++ implementation)
- Throws away/changes/mix up some of the good old AD-terms:
  - Independent / Dependent
  - Forward and Reverse mode
  - Seeding, Compression of Jacobians
- Elimination rules for Hessians keeping symmetry

We have also seen (Cons)

- Dynamic sparsity handling (Overhead)
- STL `map`: Handling dynamic data structures all the time (Overhead)

We have not seen (so far)

- Performance tests/Comparisons

Outlook – ADTAGEO $\longrightarrow$ ALLEGRO

- Classes for vectors of `daglad`'s
  - Destructor: access to a whole bunch of vertexes
  - Optimize elimination sequence: heuristics, `ANGEL`
  - Test: Speelpenning, randomised element ordering

N	FM	elim	RM	elim	OM	elim
500000	30s	92%	25s	92%	12s	75%

- Extend user interface
  - Develop Hessian retrieval machanism
  - Return compressed rows / columns of Jacobian / Hessian too
  - Sparse Jacobian/Hessian-Vector products
  - Enforce accumulation / elimination
  - Self verifying mode: Derivatives completely accumulated ?

Thank you!

Additionally:

Many thanks to Till Tantau, author of **BEAMER** and **PGF** (Portable Graphics Format, used to draw the graphs):

`http://sourceforge.net/projects/latex-beamer/`