

# Recent Experiences with (semi-)automatic Differentiation

**Christian Bischof**

Institute for Scientific Computing  
and  
Center for Computing and Communications

University of Technology Aachen

[bischof@sc.rwth-aachen.de](mailto:bischof@sc.rwth-aachen.de)

SIAM Optimization Meeting, Toronto, May 2002



## Outline

---

- **Automatic Differentiation**
  - Theoretical Underpinnings
  - Available Tools
- **AD Applications**
  - FLUENT CFD Code
  - Neutron Scattering
- **Improving AD Tools**
  - Use of OpenMP
  - ADiMat Tool for Matlab
- **Concluding Remarks**

# The Need for Derivatives

- Many numerical methods, sensitivity analysis, design optimization, inverse problems, data assimilation need derivatives (Gradients, Jacobians, Hessians, explicit or as operators).

Local Model  
based on  
Taylor Series

$$f(x+h) = f(x) + \frac{df}{dx} * h + O(h^2)$$

- „f“ can be a formula, but in most cases it is a computer program written in some programming language, and sometimes of substantial size.

**The problem: Fast and accurate computation of derivatives of computer programs with little effort**

# Derivatives of Codes

**Oldfashioned Way:**  
Accuracy and Speed required human effort

	Work	Accuracy	Speed
Handcoding	☹☹	: ?	::
Divided Differences	☺	???	:
Symbolic-Assisted	☹	:	:

**Automatic Differentiation:**  
Fast & accurate derivatives, fast

Automatic Differentiation	☺☺	:	:
AD+Brains	☺	:	::

## AD and Optimization

---

- Many users are contemplating the transition from simulation to optimization-related approaches.
- If the optimization algorithm fails due to wobbly derivatives  
„The optimization algorithm isn't robust“.
- If the optimization approach takes too long because derivative evaluation is expensive  
„Optimization is still infeasible“
- **AD is no panacea, but it can help avoid all kinds of trouble for users of optimization algorithms.**

## AD Theory Review

---

### Messages:

- There are many ways to do AD
- „Optimal“ AD algorithms are a research issue

## Automatic Differentiation (AD)

Given a computer program, AD generates a new program, which applies the chain rule of differential calculus, e.g.,

$$\frac{df(w, v)}{dx} = \frac{\partial f}{\partial w} * \frac{dw}{dx} + \frac{\partial f}{\partial v} * \frac{dv}{dx}$$

to elementary operations (i.e.,  $\frac{\partial f}{\partial w}, \frac{\partial f}{\partial v}$  are known).

- AD does not generate truncation- or cancellation errors.
- AD generates a program for the computation of derivative values, not derivative formulae.

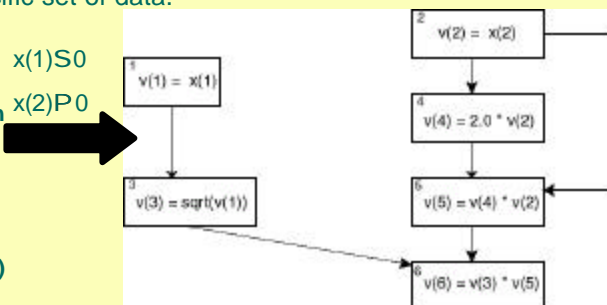
## Program ~ Computational Graph

- A computer program can be viewed as the “generator” of a “computational graph.”
- Each node in the graph corresponds to a value computed by the program executing with a specific set of data.

```

subroutine f(x,y)
real x(2), y(2)
do i = 1, 2
  if ((x(i) .ge. 0) then
    y(i) = sqrt(x(i))
  else
    y(i) = 2.0 * x(i)
  endif
enddo
y(2) = y(1) * y(2) * x(2)
return
end
  
```

**The code**

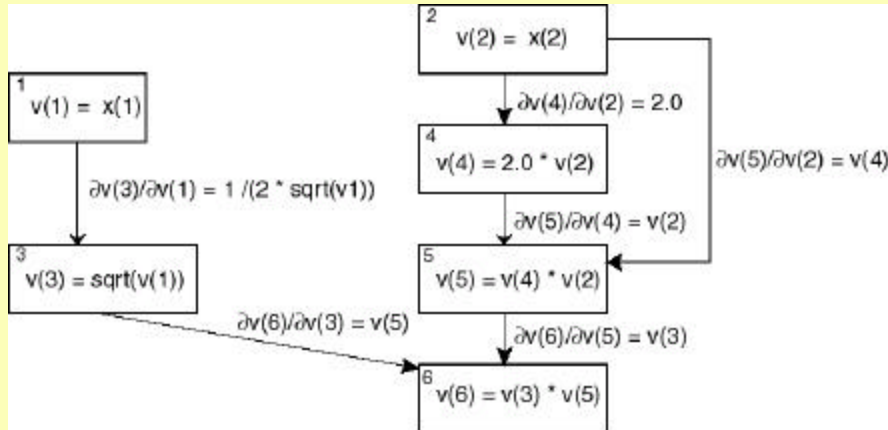


**The computational graph**

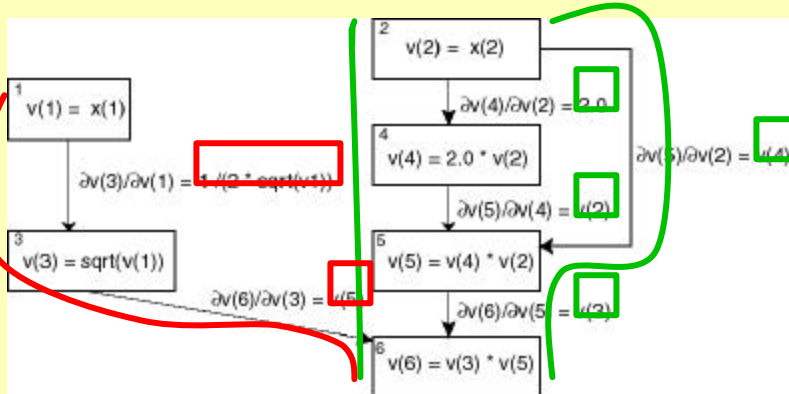
(v(1), v(2)) == inputs (x(1), x(2)), and (v(3), v(6)) == outputs (y(1), y(2)).

# Linearized Computational Graph

- The "linearized computational graph" extends the "computational graph" with partial derivatives for each arc (so-called elementary partial derivatives).



# Accumulating Elementary Partial



$$\frac{dy(2)}{dx(1)} = \frac{dv(6)}{dv(1)} = \frac{\partial v(6)}{\partial v(3)} * \frac{\partial v(3)}{\partial v(1)} \quad \text{---}$$

$$\frac{dy(2)}{dx(2)} = \frac{dv(6)}{dv(2)} = \frac{\partial v(6)}{\partial v(5)} * \frac{\partial v(5)}{\partial v(4)} * \frac{\partial v(4)}{\partial v(2)} + \frac{\partial v(6)}{\partial v(5)} * \frac{\partial v(5)}{\partial v(2)} \quad \text{---}$$



## AD ~ Path Accumulation

The derivative of  $v(n)$  with respect to  $v(m)$  is the sum over all paths of the product of the partials along a path from  $v(m)$  to  $v(n)$ .

The various AD techniques can be interpreted as exploiting different strategies for computing this sum over all paths (possible because of associativity of chain rule).

They may significantly differ in their computational requirements, but all deliver derivatives accurate to machine precision.

**There is no known algorithm for computing derivatives of a program/computational graph with minimal complexity.**



## The Forward Mode

Associate a “gradient”  $\tilde{N}$  with every program variable and apply derivative rules for elementary operations.

```
t:=1.0
do i=1 to n step 1
  if (t>0) then
    t := t*x(i);
  endif
endif
```

$\mathcal{P}$

```
t:=1.0;  $\tilde{N}t=0.0$ ;
do i=1 to n step 1
  if (t>0) then
     $\tilde{N}t:= x(i)* \tilde{N}t+t* \tilde{N}x(i)$ ;
    t := t*x(i);
  endif
endif
```

The computation of  $p$  (directional) derivatives requires gradients of length  $p$  many vector linear combinations, e.g.,  $\tilde{N}x(i) = (0,..,0,1,0,..,0)$  results in  $\tilde{N}t == dt/dx(1:n)$ .

## The Reverse Mode

Associate an “adjoint”  $a$  with every program variable and apply the following rules to the “inverted” program:

$s = f(v,w) \text{ } \textcircled{R} \text{ } a_v += ds/dv * a_s; a_w += ds/dw * a_s;$

```
tval(0):=1.0; tctr:=0;
do i=1 to n step 1
  if (tval(tctr)>0) then
    jump(i):='true'; tctr=tctr+1;
    tval(tctr):=tval(tctr-1)*x(i);
```

**1. Step:**  
Reversible program  
(Single Assignment Form)

**2. Step:**  
Adjoint computation

$a\_tval(tctr)=1.0,$   
all other  $a\_*=0.$   
Upon exit,  
 $a\_x(i)=dt/dx(i)$

```
do i=n downto 1 step -1
  if (jump(i) == 'true') then
     $\alpha\_tval(tctr-1)+=x(i) * \alpha\_tval(tctr);$ 
     $\alpha\_x(i)+=tval(tctr-1) * \alpha\_tval(tctr);$ 
     $\alpha\_tval(tctr)=0; tctr=tctr-1;$ 
```

## Remarks on AD (inputs $x$ , outputs $y$ )

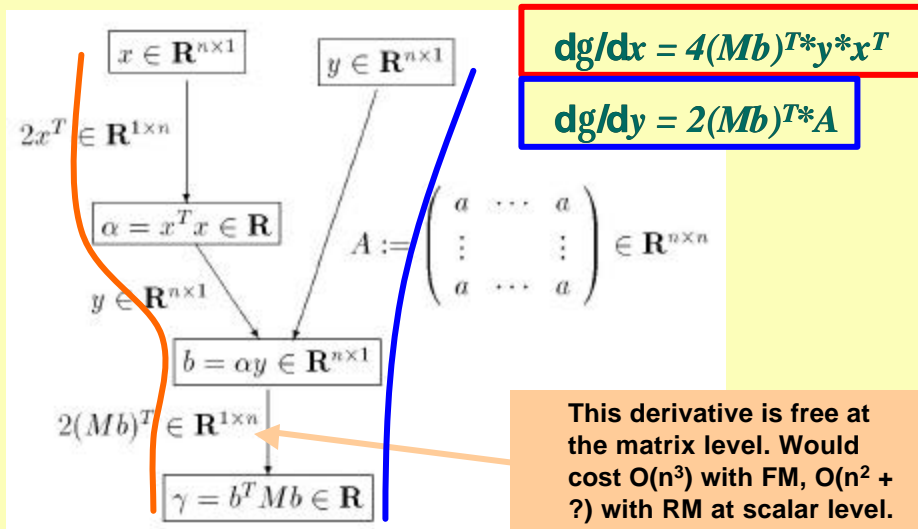
### FM = Forward Mode

- computes  $dy/dx * S$  ( $S$  is called “seed matrix”) by propagating sensitivities of intermediate values with respect to input values.
- For  $p$  input values of interest, runtime and memory scale at most with  $p$ . May be much less e.g. for sparse Jacobians.

### RM = Reverse Mode

- computes  $W^T * dy/dx$  by propagating sensitivities of output values with respect to intermediate values.
- For  $q$  output values of interest, RM runtime scales with  $q$ . Memory requirements are harder to predict, and greatly depend on structure of program. Great for computing „long gradients“.

## AD at a matrix level



## AD Tools

### Messages:

- AD tools are available for most programming languages
- Standard scenarios for computing derivatives require little user effort

## Some AD Tools

### Fortran 77, some Fortran 90:

- ADIFOR 2.0 (FM): C.B., Paul Hovland (ANL/MCS), Alan Carle & Mike Fagan (Rice U.)
- ADIFOR 3.0/ADJIFOR (FM/RM,  $d^2$  f): A. Carle & M. Fagan (Rice U.)
- AD01/AD02 (FM/RM,  $d^k$  f): John Reid (Rutherford Labs)
- TAPENADE (RM, d f): Laurent Hascoet (INRIA)
- TAMCTAF (RM, d f): Ralf Giering (Fastopt GmbH).

### ANSI-C/C++:

- ADIC (FM,  $d^2$  f): C.B., P. Hovland & Boyana Norris (ANL/MCS)
- ADOL-C (FM/RM,  $d^k$  f): Andreas Griewank & Co. (TU Dresden)

### Matlab:

- ADiMAT: C.B. & Andre Vehreschild (RWTH Aachen)
- ADMAT: Arun Verma (Cornell University)
- MAD: Shaun Forth & Robert Ketzscher (Cranfield University)

## The ADIFOR/ADIC Process

```

# subroutine to be differentiated
AD_TOP      = deba
# number of derivatives to be computed
AD_PMAX     = 5
# independent variables
AD_IVARS    = b,c,d,nu100,nu210
# dependent variables
AD_OVARS    = hmin,pmaxe,cfhp
# Name of file listing names of files
# containing deba and all subroutines called by it.
AD_PROG     = deba.cmp
    
```

## ADIFOR 2.0-generated Code

$w = z(1)*z(2)*z(3)*z(4)$  is transformed into

```
r1_v=z(1)*z(2)
r2_v=r1_v*z(3)   dw/dz(4)
r1_b=z(4)*z(3)
r2_b=z(4)*r1_v   dw/dz(3)
r3_b=r1_b*z(2)   dw/dz(1)
r4_b=r1_b*z(1)   dw/dz(2)
```

reverse (or  
adjoint) mode  
of AD

Forward Mode uses vector  
linear combination

$$| = \bullet * | + \dots \bullet * |$$

The SparsLinC library  
can transparently exploit  
sparsity.

```
do g_i = 1, g_p
  g_w(g_i) = r3_b * g_z(g_i,1)
             + r4_b * g_z(g_i,2)
             + r2_b * g_z(g_i,3)
             + r2_v * g_z(g_i,4)
enddo
```

$g_w(:)$ ,  $g_z(:,i)$  contain derivatives  
 $dw/dz(i)$  and  $dz(i)/dx$ .

forward mode of AD

```
w = r2_v * z(4) original value
```

Typical Code Expansion ~ Factor 3

## AD Implementation

- **Source Transformation (e.g. ADIFOR):**

- Requires significant infrastructure.
- Greater choice in applying chain rule and in code generation for specific architectures

- **Operator Overloading (e.g. ADOL-C):**

Define a new method for elementary operations in a new data type.

```
class Autoderiv{ ...
  Autoderivoperator*(const Autoderiv&, const Autoderiv&);
  { return Autoderiv(F(x)*F(y), F(y)*dF(x)+F(x)*dF(y)); }
  ... },
```

- Requires only redefinition of elementary operators.
- Difficult to go beyond one operation at a time in doing AD.
- Potential overhead due to compiler-generated temporaries,  
e.g.  $w = x*y*z \rightarrow tmp = x*y; w = tmp*z$ .

- **FLUENT CFD Code**

Message:

- AD technology scales to arbitrary large codes

- **AD in Neutron Scattering**

Messages:

- AD saves you trouble
- Speed improves significantly with insight

*(joint work w/ Martin Bückler, Bruno Lang, Arno Rasch, Emil Slusanschi)*

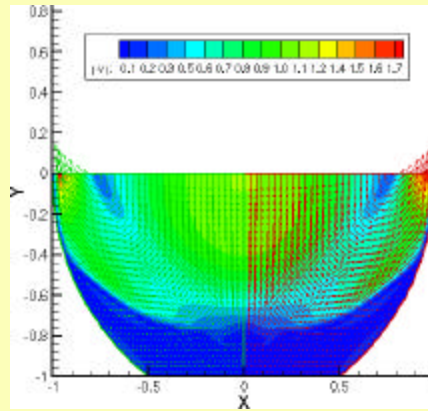
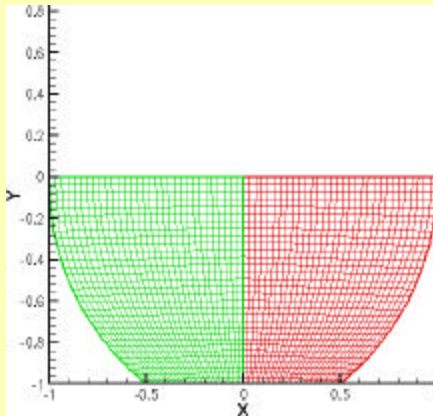
**Fluent V4.52 source code:**

- Fortran 77, partially F90 (dyn. memory management) and C (system functions)
- Approx. **1,600,000 lines of F77/F90 code** (670,000 non-comment), ~ 1,500 files and 2,400 subroutines.
- Approx. 16,000 lines C code.

**Fluent.AD required in SFB540 center investigating kinetic phenomena in multiphase reactive systems:**

- Wavy film running down vertical wall
- k- $\epsilon$  turbulence model (2 phase)
- Five Model parameters:  $c_{1e}$ ,  $c_{2e}$ ,  $c_m$ ,  $S_k$ ,  $S_e$
- Want derivatives of field values w.r.t model parameters.

## Fluent Sample Problem



Water and Air in a spinning Bowl

## Difficulties in ADIFOR processing

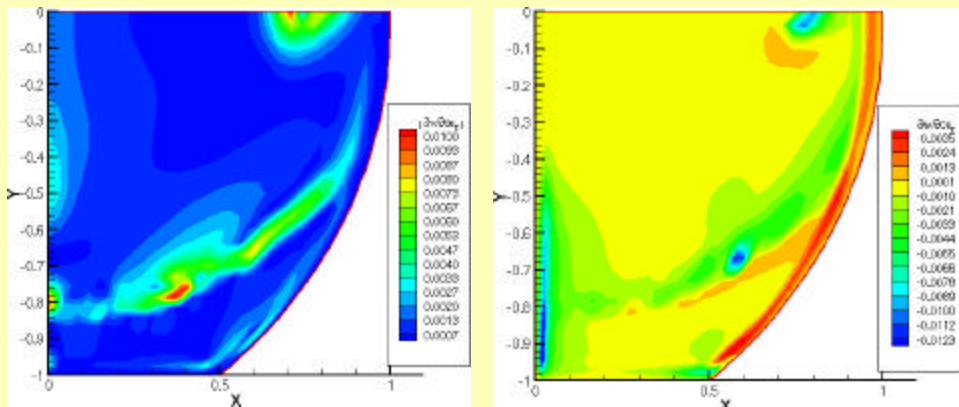
- Dubious programming techniques:
  - Type mismatches in actual & declared parameters
- Bugs:
  - inconsistent number of arguments in subroutine calls
- Not conforming to Fortran77 standard
  - while statement in one subroutine
- ADIFOR2.0 limitations:
  - I/O statements containing function invocations

**Please try to write clean code**  
**Any kind of code analysis will benefit!**

# FLUENT versus FLUENT.AD

	Fluent	Fluent.AD	Ratio AD/Original
Lines of Code (w/ comments)	1,592,188	1,620,430	1.02
Lines of Code (w/o comments)	673,774	<b>ADIFOR's interprocedural dependence analysis realizes that many files are not on the path from „independent“ to „dependent“ variables.</b>	
Number of Files	1474		
Number of subroutines and functions	2411	1249	0.52

# FLUENT.AD - Results



Derivative of velocity and swirl velocity w.r.t.  $c_{e2}$

## Resource Requirements

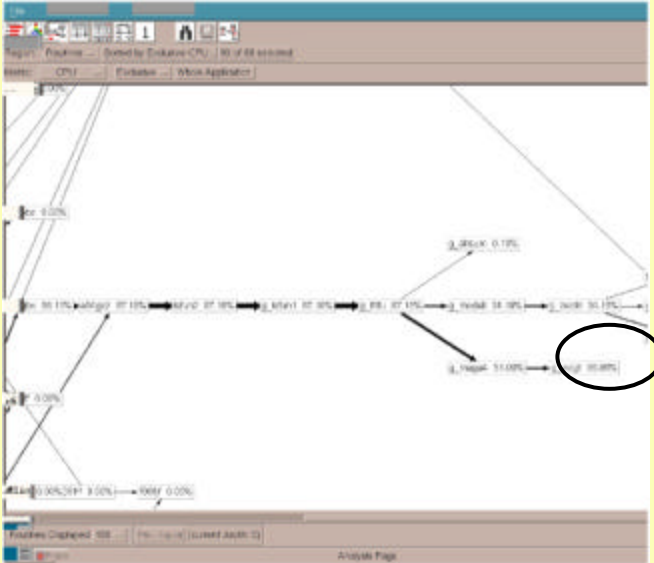
- **Memory Usage:**
  - 32 MB for original code
  - 168 MB for differentiated code
- **Runtime on Intel PIII (600 Mhz):**
  - Derivatives take 7,5 times longer than running of code alone. (-o2 optimization level on PGF, no experimentation with other options yet).

## AD in Neutron Scattering

*(joint work with Martin Bucker, Dieter an Mey)*

- **Nonlinear least-squares formulation to determine value of one (!) parameter from 12,237 data sets.**
- **Theoretical scattering function consists of about 3,000 lines of Fortran code.**
- **Originally used: NAG E04FDF, quasi-Newton method with divided-difference gradient:**  
 586 sec. on HP V-class, 13 evals of  $f(\cdot)$ ,  
**warning about dubious quality of result!**
- **AD enables use of modified Gauss-Newton Method NAG E04GEF:**  
 498 sec., 6 evals of  $(f, \tilde{N}f)$ , **no warnings.**
- **Eval. of  $f$  takes 38,7 sec.,  $(f, \tilde{N}f)$  takes 69,2 sec.**

## A Closer Look at the Call Graph



A „leaf“  
subroutine  
accounts for  
almost 40% of  
the runtime of  
the program!

## A Closer Look at g\_voigt(...)

subroutine voigt(u,v,x,y)

↓ Adifor

subroutine g\_voigt(n,u,g\_u,ldg\_u,v,x,y,g\_y,ldg\_y)

From parameter list:

- (x,y) input ( $\Leftrightarrow$  complex number). (u,v) output.
- v and x are passive (no gradients  $g_v, g_x$ )
- u and y are active (gradients  $g_u, g_y$ )

**ADIFOR's dependence analysis tells us that  
only  $du/dy$  is needed!**

## Original Function voigt(..)

Let  $i = \text{sqrt}(-1)$  and  $x, y, u, v$  real.  
Then for given  $z = x + i*y$

subroutine voigt(u,v,x,y)

computes  $w = u + i*v$  defined by

$$w(z) = e^{-z^2} \cdot \text{erfc}(-iz), \quad \text{Im}(z) > 0,$$

where

$$\text{erfc}(z) = 1 - \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt.$$

This was not apparent from the 53 lines of code, which was rather convoluted (8 goto's!). The author told us.

## „Handcoded“ Derivative

With some tedious, but elementary mathematics:

$$\nabla u = \left( 2(uy + vx) - \frac{2}{\sqrt{\pi}} \right) \nabla y.$$

Thus, derivatives can be computed by executing original function voigt(..) first, followed by

```

dtmp = 2.0d0*(u*y + v*x) - 1.12837916709551d0
do j = 1, n
  g_u(j) = dtmp*g_y(j)
enddo

```

## Impact of Strategic Math

- **Out of roughly 3000 lines of code, only one subroutine comprising 53 lines of code was treated „special“, ADIFOR automatically took care of the rest.**
- **Handcoded derivative for subroutine voigt() is 6.9 times faster than ADIFOR-generated one.**
- **Time to compute  $(f, \tilde{N}f)$  is reduced to 41.5 seconds from 69.2 seconds.**
- **Overall execution time is reduced to 333 seconds from 498 seconds for „vanilla AD“, almost half of „no AD“.**
- **Inspection and optimization is possible due to transparency of source transformation approach to AD.**

## Improving AD Tools

- **Use of OpenMP in AD-generated code**

**Messages:**

- **Derivatives will get „cheaper“ through transparent use of parallel hardware**

- **New AD Tool for Matlab**

**Message:**

- **AD tools are still improving**

# Parallelism via OpenMP

(joint work w/ Martin Bucker, Dieter an Mey)

- Fundamental operation in first-order AD is a vector linear combination, e.g.:  

$$w(:) = x*v1(:) + y*v2(:) + z*v3(:)$$
- OpenMP provides a set of directives allowing thread-based parallelization for shared-memory parallel machines (SMP):
  - Directives for defining „parallel regions“
  - Constructs for parallelizing loops
- Most users will soon have an SMP nearby or on their desk.

# Differentiated User Code

C+AD4 Start of executable statements.

```

c$omp parallel
  do i = 1, n - k
c$omp master
  X2_rtmp1 = 0.0d0
c$omp end master
  call g_accum_d_0(ad_p_, g_z(1,i))
c$omp master
  z(i) = dble(X2_rtmp1)
c$omp en
    subroutine g_accum_d_4(l, lhs, w_1, v_1, w_2, v_2)
      integer l
      double precision lhs(1), v_1(1), w_1, v_2(1), w_2
+g c$omp parallel do shared(l, lhs, w_1, v_1, w_2, v_2) private(i)
c$omp do
  do i = 1, l
    lhs(i) = w_1*v_1(i) + w_2*v_2(i)
  ...
c$omp en c$omp end do nowait
re      end
end
  
```

AD utility routine

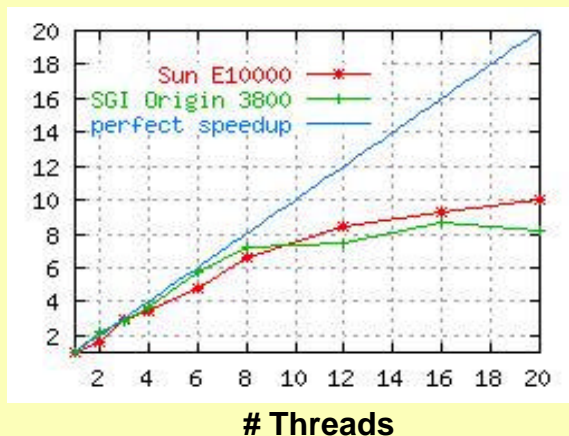
# Performance

- Derivatives for 3294 real variables from a biomagnetics problem
- SUN Enterprise 10000 (400 MHz MIPS UltraSparc II)
- SGI Origin 3800 (400 MHz MIPS R12000)

## Speedup

**This speedup is realized by the user without any work!**

Just one example of potential for optimization of AD-generated code.



# ADiMat: AD for MATLAB

*(joint work w/ Andre Vehreschild)*

- First tool to combine source transformation approach (to exploit context for computing derivatives) with operator overloading (to implement derivative objects), e.g. differentiating  $y=M*v$  with respect to  $M$  and  $v$  leads to:

$$g_y = g_M * v + M * g_v$$

- ADiMat analyzes MATLAB 6 code to determine variables that need derivatives. More difficult due to Matlab's incomplete type system:
  - Every object is an array (scalars are 1x1 arrays)
  - Sizes, shape, and type of objects are mostly only known at runtime.

# ADiMat by Example

```
function p = fit(x, d, m)
% FIT -- Given x and d, fit() returns p
% such that norm(V*p-d)=min,
% where V=[1, x, x.^2, ... x.^(m-1)].

dim_x = size(x, 1);
V = ones(dim_x, 1);
for count= 1: (m-1)
    V = [V, x.^ count];
end
P = V \ d;
```

Parsing

Activity analysis to determine variables needing derivatives

„Integer“-functions truncate dependencies

Canonicalization

Augmentation

Unparsing

# ADiMat Output

```
function [g_p, p] = g_fit(g_x, x, d, m)

dim_x= size(x, 1);

g_V= adgradobj(getNDD, [dim_x, 1], 'zeros');
V= ones(dim_x, 1);
for count= 1: (m- 1)
    t2= x.^ (count- 1);
    g_t0= count.* t2.* g_x;
    t0= t2.* x;
    g_V= [g_V, g_t0];
    v= [V, t0];
end

t1= v \ d;
g_p= v \ (-g_V* t1);
p= t1;
```

Allocate zero derivative object of appropriate size for V since g\_V appears first on right-hand side of an assignment

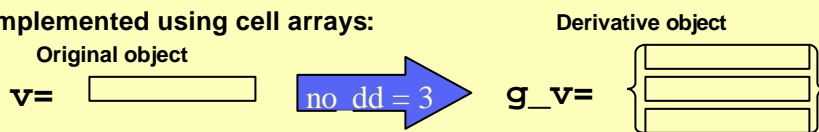
g\_t0 and g\_p are allocated automatically

Planned: Aggressive optimization of derivative operators

# ADiMat Derivative Class

- The ADiMAT derivative class can cope with dynamic types and shapes of objects and implements Matlab expressions generated to compute derivatives.
- Traditional operator overloading implementations (e.g. ADMAT) compute the original value and the derivative within the overloaded operator.

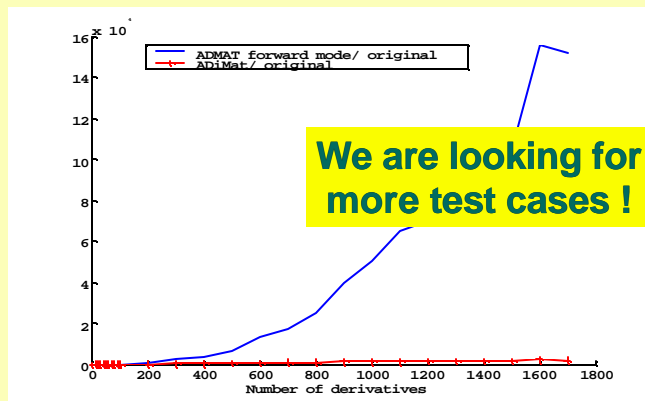
- Implemented using cell arrays:



- Small: ~660 lines of code (loc) <-> ADMAT: ~2220 loc  
(Counting the loc of the dense forward mode impl. of ADMIT/ADMAT)

# Performance

- ADiMat can significantly outperform pure operator overloading approaches to AD!



Measured on a PC Pentium IV, 1.7GHz, 1GB RAM using MATLAB 6 (Release 12).

## Other AD Projects at Aachen

- **SFB 401: Flow and Fluid-structure interaction at airplane wings.**
  - AD augmentation of TFS (The Flow Solver) of the Aerodynamic Institute of the RWTH
  - Assessment of hybrid algorithms for optimization (genetic algorithms & classical Newton approaches)
- **MNDO semi-empirical quantum-chemical code, Max-Planck Institute Mülheim.**
  - AD discovered a bug in hand-coded derivatives
  - Now tackling as yet undifferentiated sections.
- **“Deep Groundwater” structural identification project, Inst. Applied Geophysics, RWTH Aachen.**

## Concluding remarks

- AD tools have reached a good level of maturity.
- **There is no reason for not giving AD tools a try!**
- Higher-order derivatives are no problem and may be surprisingly cheap.
- Challenges for AD
  - Improving AD algorithms
  - Getting AD in the loop when codes are written, not afterwards
  - Getting AD into compilers

[www.sc.rwth-aachen.de](http://www.sc.rwth-aachen.de)

[www.autodiff.org](http://www.autodiff.org)

- A. Griewank, Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, SIAM, Philadelphia, 2000.
- G. Corliss, A. Griewank, C. Faure, L. Hascoet, and U. Naumann, editors, Automatic Differentiation: From Simulation to Optimization, Springer, 2001.
- M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, Computational Differentiation: Techniques, Applications, and Tools, SIAM, Philadelphia, 1996.
- A. Griewank and G. Corliss, editors, Automatic Differentiation of Algorithms, SIAM, Philadelphia, 1991.
- C. Bischof, A. Carle, P. Khademi, and A. Mauer, ADIFOR 2.0: Automatic Differentiation of Fortran 77 Programs, IEEE Computational Science & Engineering, 3(3):18-32, 1996.
- C. Bischof, L. Roh, and A. Mauer, ADIC - An Extensible Automatic Differentiation Tool for ANSIC, Software--Practice and Experience", 27(12):1427-1456, 1997.

- **Today at 10.30:**
  - MS24 - What is AD doing for Optimization (organized by J. Moré)**
    - An interactive optimization environment (A. Rasch, Aachen)
    - Large-scale optimization (P. Hovland, Argonne)
    - Faster Jacobians (U. Naumann, Argonne)
    - Constrained Opt. w/ adjoints (A. Griewank, Dresden)
- **Today at 2.45pm:**
  - CP22 – Automatic Differentiation (organized by M. Fagan)**
    - AD Tool for Fortran90 (M. Fagan, Rice)
    - Exploiting Sparsity (J. Riehme, Dresden)
    - Faster Jacobians (J. Pryce, Cranfield)
    - Multivariate Taylor Series (R. Neidinger, Davidson)
    - AD for Matlab (R. Ketzscher, Cranfield)
    - Sparse Jacobians (S. Hossain, Lethbridge)
- **Yesterday: AD Course (org. by A. Griewank, U. Naumann, A. Walther)**