Introduction to Algorithmic Differentiation

J. Utke

Argonne National Laboratory
Mathematics and Computer Science Division

May/2013 at Ames Lab
outline

- motivation
- basic principles
- tools and methods
- considerations for the user
why algorithmic differentiation?

given: some numerical model $y = f(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m$
implemented as a (large / volatile) program

wanted: sensitivity analysis, optimization, parameter (state)
estimation, higher-order approximation...

1. don’t pretend we know nothing about the program
   (and take finite differences of an oracle)

2. get machine precision derivatives as $J \dot{x}$ or $\bar{y}^T J$ or ...
   (avoid approximation-versus-roundoff problem)

3. the reverse (aka adjoint) mode yields “cheap” gradients

4. if the program is large, so is the adjoint program, and
   so is the effort to do it manually ... easy to get wrong but hard to
debug

⇒ use tools to do it automatically!
why algorithmic differentiation?

given: some numerical model \( y = f(x): \mathbb{R}^n \mapsto \mathbb{R}^m \)
implemented as a (large / volatile) program

wanted: sensitivity analysis, optimization, parameter (state)
estimation, higher-order approximation...

1. don’t pretend we know nothing about the program
   (and take finite differences of an oracle)

2. get machine precision derivatives as \( J\dot{x} \) or \( \bar{y}^T J \) or ...
   (avoid approximation-versus-roundoff problem)

3. the reverse (aka adjoint) mode yields “cheap” gradients

4. if the program is large, so is the adjoint program, and
   so is the effort to do it manually ... easy to get wrong but hard to
   debug

⇒ use tools to do it automatically?
why algorithmic differentiation?

given: some numerical model \( y = f(x) : \mathbb{R}^n \mapsto \mathbb{R}^m \)
implemented as a (large / volatile) program

wanted: sensitivity analysis, optimization, parameter (state) estimation, higher-order approximation...

1. don’t pretend we know nothing about the program
   (and take finite differences of an oracle)

2. get machine precision derivatives as \( J \dot{x} \) or \( \bar{y}^T J \) or ...
   (avoid approximation-versus-roundoff problem)

3. the reverse (aka adjoint) mode yields “cheap” gradients

4. if the program is large, so is the adjoint program, and
   so is the effort to do it manually ...
   easy to get wrong but hard to debug

\[ \Rightarrow \] use tools to do it at least semi-automatically!
how does AD compute derivatives?

\[ f : y = \sin(a \ast b) \ast c : \mathbb{R}^3 \mapsto \mathbb{R} \]

yields a graph representing the order of computation:
how does AD compute derivatives?

\[ f : y = \sin(a \ast b) \ast c : \mathbb{R}^3 \mapsto \mathbb{R} \]
yields a graph representing the order of computation:

\[ t1 = a \ast b \]
\[ t2 = \sin(t1) \]
\[ y = t2 \ast c \]

\[ \diamond \text{ code list} \rightarrow \text{ intermediate values } t1 \text{ and } t2 \]
how does AD compute derivatives?

\[ f : y = \sin(a \times b) \times c : \mathbb{R}^3 \mapsto \mathbb{R} \]
yields a graph representing the order of computation:

- **code list** → intermediate values \( t_1 \) and \( t_2 \)
- each intrinsic \( v = \phi(w, u) \) has local partials \( \frac{\partial \phi}{\partial w} \), \( \frac{\partial \phi}{\partial u} \)
- e.g. \( \sin(t_1) \) yields \( p_1 = \cos(t_1) \)
- in our example all others are already stored in variables

\[
\begin{align*}
t_1 &= a \times b \\
p_1 &= \cos(t_1) \\
t_2 &= \sin(t_1) \\
y &= t_2 \times c
\end{align*}
\]
how does AD compute derivatives?

\[ f : y = \sin(a \ast b) \ast c : \mathbb{R}^3 \rightarrow \mathbb{R} \]
yields a graph representing the order of computation:

- code list → intermediate values \( t_1 \) and \( t_2 \)
- each intrinsic \( v = \phi(w, u) \) has local partials \( \frac{\partial \phi}{\partial w} \), \( \frac{\partial \phi}{\partial u} \)
- e.g. \( \sin(t_1) \) yields \( p_1 = \cos(t_1) \)
- in our example all others are already stored in variables

\[ \begin{align*}
  t_1 &= a \ast b \\
  p_1 &= \cos(t_1) \\
  t_2 &= \sin(t_1) \\
  y &= t_2 \ast c
\end{align*} \]

What do we do with this?
forward mode with directional derivatives

- **associate** each variable \( v \) with a derivative \( \dot{v} \)
- take a point \((a_0, b_0, c_0)\) and a direction \((\dot{a}, \dot{b}, \dot{c})\)
- for each \( v = \phi(w, u) \) propagate forward in order
  \[
  \dot{v} = \frac{\partial \phi}{\partial w} \dot{w} + \frac{\partial \phi}{\partial u} \dot{u}
  \]

- in practice: associate **by name** \([a, d_a]\) or **by address** \([a\%v, a\%d]\)
- interleave propagation computations

\[
\begin{align*}
  t1 &= a \times b \\
  p1 &= \cos(t1) \\
  t2 &= \sin(t1) \\
  y &= t2 \times c
\end{align*}
\]
forward mode with directional derivatives

- **associate** each variable $v$ with a derivative $\dot{v}$
- take a point $(a_0, b_0, c_0)$ and a direction $(\dot{a}, \dot{b}, \dot{c})$
- for each $v = \phi(w, u)$ propagate forward in order
  \[ \dot{v} = \frac{\partial \phi}{\partial w} \dot{w} + \frac{\partial \phi}{\partial u} \dot{u} \]

- in practice: associate by name $[a, d_a]$ or by address $[a\%v, a\%d]$
- interleave propagation computations

\[
\begin{align*}
t1 &= a*b \\
d_{\_t1} &= d_a*b + d_b*a \\
p1 &= \cos(t1) \\
t2 &= \sin(t1) \\
y &= t2*c
\end{align*}
\]
forward mode with directional derivatives

- **associate** each variable \( v \) with a derivative \( \dot{v} \)
- take a point \((a_0, b_0, c_0)\) and a direction \((\dot{a}, \dot{b}, \dot{c})\)
- for each \( v = \phi(w, u) \) propagate forward in order
  \[
  \dot{v} = \frac{\partial \phi}{\partial w} \dot{w} + \frac{\partial \phi}{\partial u} \dot{u}
  \]

- in practice: associate by name \([a, d_a]\) or by address \([a\%v, a\%d]\)
- interleave propagation computations

\[
\begin{align*}
  t1 &= a*b \\
  d_t1 &= d_a*b + d_b*a \\
  p1 &= \cos(t1) \\
  t2 &= \sin(t1) \\
  d_t2 &= d_t1*p1 \\
  y &= t2*c
\end{align*}
\]
forward mode with directional derivatives

- **associate** each variable $v$ with a derivative $\dot{v}$
- take a point $(a_0, b_0, c_0)$ and a direction $(\dot{a}, \dot{b}, \dot{c})$
- for each $v = \phi(w, u)$ propagate forward in order
  \[ \dot{v} = \frac{\partial \phi}{\partial w} \dot{w} + \frac{\partial \phi}{\partial u} \dot{u} \]

- in practice: associate by name $[a, d_a]$ or by address $[a\%v, a\%d]$
- interleave propagation computations
  \[
  t_1 = a \times b \\
  d\_t1 = d\_a*b + d\_b*a \\
  p1 = \cos(t1) \\
  t2 = \sin(t1) \\
  d\_t2 = d\_t1*p1 \\
  y = t2*c \\
  d\_y = d\_t2*c + d\_c*t2
  \]
forward mode with directional derivatives

- associate each variable $v$ with a derivative $\dot{v}$
- take a point $(a_0, b_0, c_0)$ and a direction $(\dot{a}, \dot{b}, \dot{c})$
- for each $v = \phi(w, u)$ propagate forward in order
  \[ \dot{v} = \frac{\partial \phi}{\partial w} \dot{w} + \frac{\partial \phi}{\partial u} \dot{u} \]

- in practice: associate by name $[a, d_a]$ or by address $[a%v, a%d]$
- interleave propagation computations
  \[
  t_1 = a \times b \\
  d_{t_1} = d_a \times b + d_b \times a \\
  p_1 = \cos(t_1) \\
  t_2 = \sin(t_1) \\
  d_{t_2} = d_{t_1} \times p_1 \\
  y = t_2 \times c \\
  d_y = d_{t_2} \times c + d_c \times t_2
  \]

What is in $d_y$?
\( \dot{y} \) contains a projection

\[ y = J \dot{x} \] computed at \( x_0 \)
d_y contains a projection

- $\dot{y} = J \dot{x}$ computed at $x_0$
- for example for $(\dot{a}, \dot{b}, \dot{c}) = (1, 0, 0)$
\( \dot{y} = J \dot{x} \) computed at \( x_0 \)

- for example for \((\dot{a}, \dot{b}, \dot{c}) = (1, 0, 0)\)

 y yields the first element of the gradient

all gradient elements cost \( O(n) \) function evaluations
applications

for instance

- ocean/atmosphere state estimation & uncertainty quantification, oil reservoir modeling
- computational chemical engineering
- CFD (airfoil shape optimization, suspended droplets e.g. by Dervieux, Forth, Gauger, Giles et al.)
- beam physics
- mechanical engineering (design optimization)

use

- gradients
- Jacobian projections
- Hessian projections
- higher order derivatives
  (full or partial tensors, univariate Taylor series)
applications
for instance

- ocean/atmosphere state estimation & uncertainty quantification, oil reservoir modeling
- computational chemical engineering
- CFD (airfoil shape optimization, suspended droplets e.g. by Dervieux, Forth, Gauger, Giles et al.)
- beam physics
- mechanical engineering (design optimization)

use

- gradients
- Jacobian projections
- Hessian projections
- higher order derivatives (full or partial tensors, univariate Taylor series)

How do we get the cheap gradients?
higher order AD (1)

propagation of (univariate) Taylor polynomials up to order \( o \) (in \( d \) directions) with coefficients \( a^{(i)}_j, j = 1 \ldots o, i = 1 \ldots d \) around a common point \( a_0 \equiv a^i_0 \) in the domain

\[
\phi(a_0 + h) = \phi(a_0) + \phi'(a_0) \cdot h + \frac{\phi''(a_0)}{2!} \cdot h^2 + \ldots + \frac{\phi^{(d)}(a_0)}{o!} \cdot h^o
\]
higher order AD (1)

- propagation of (univariate) Taylor polynomials up to order $o$ (in $d$ directions) with coefficients $a_{j}^{(i)}$, $j = 1 \ldots o$, $i = 1 \ldots d$ around a common point $a_0 \equiv a_0^i$ in the domain

$$
\phi(a_0 + h) = \phi(a_0) + \phi'(a_0) \cdot h + \frac{\phi''(a_0)}{2!} \cdot h^2 + \ldots + \frac{\phi^{(d)}(a_0)}{o!} \cdot h^o
$$

- i.e. again no numerical approximation using finite differences
higher order AD (1)

- propagation of (univariate) Taylor polynomials up to order $o$ (in $d$ directions) with coefficients $a_j^{(i)}$, $j = 1 \ldots o$, $i = 1 \ldots d$ around a common point $a_0 \equiv a_0^i$ in the domain

\[
\phi(a_0 + h) = \phi(a_0) + \phi'(a_0) \cdot h + \frac{\phi''(a_0)}{2!} \cdot h^2 + \ldots + \frac{\phi^{(d)}(a_0)}{o!} \cdot h^o
\]

- i.e. again no numerical approximation using finite differences

- for "general" functions $b = \phi(a)$ the computation of the $b_j^i$ can be costly (Faa di Bruno’s formula)
higher order AD (1)

- propagation of (univariate) Taylor polynomials up to order \( o \) (in \( d \) directions) with coefficients \( a_{j}^{(i)}, j = 1 \ldots o, i = 1 \ldots d \) around a common point \( a_0 \equiv a_0^i \) in the domain

\[
\phi(a_0 + h) = \phi(a_0) + \phi'(a_0) \cdot h + \frac{\phi''(a_0)}{2!} \cdot h^2 + \ldots + \frac{\phi^{(d)}(a_0)}{o!} \cdot h^o
\]

- i.e. again no numerical approximation using finite differences

- for “general” functions \( b = \phi(a) \) the computation of the \( b_j^i \) can be costly
  (Faa di Bruno’s formula)

- but the propagation is applied to the sequence of programming language intrinsics

- and all relevant non-linear univariate (Fortran/C++) intrinsics \( \phi \) can be seen as ODE solutions
higher order AD (2)

- using ODE approach permits (cheap) recurrence formulas for the coefficients, e.g. for $b = a^r$ we get

$$\tilde{b}_k = \frac{1}{a_o} \left( r \sum_{j=1}^{k} b_{k-j} \tilde{a}_j - \sum_{j=1}^{k-1} a_{k-j} \tilde{b}_j \right) \quad \text{with} \quad \tilde{c}_j = j c_j$$
higher order AD (2)

- using ODE approach permits (cheap) recurrence formulas for the coefficients, e.g. for $b = a^r$ we get

$$
\tilde{b}_k = \frac{1}{a_o} \left( r \sum_{j=1}^{k} b_{k-j} \tilde{a}_j - \sum_{j=1}^{k-1} a_{k-j} \tilde{b}_j \right) \quad \text{with } \tilde{c}_j = j c_j
$$

- sine and cosine are coupled

$$
s = \sin(u) : \tilde{s}_k = \sum_{j=1}^{k} \tilde{u}_j c_{k-j} \quad \text{and} \quad c = \cos(u) : \tilde{c}_k = \sum_{j=1}^{k} -\tilde{u}_j s_{k-j}
$$
higher order AD (2)

- using ODE approach permits (cheap) recurrence formulas for the coefficients, e.g. for \( b = a^r \) we get

\[
\tilde{b}_k = \frac{1}{a_o} \left( r \sum_{j=1}^{k} b_{k-j} \tilde{a}_j - \sum_{j=1}^{k-1} a_{k-j} \tilde{b}_j \right) \quad \text{with} \quad \tilde{c}_j = j c_j
\]

- sine and cosine are coupled

\[
s = \sin(u) : \tilde{s}_k = \sum_{j=1}^{k} \tilde{u}_j c_{k-j} \quad \text{and} \quad c = \cos(u) : \tilde{c}_k = \sum_{j=1}^{k} -\tilde{u}_j s_{k-j}
\]

- arithmetic operations are simple, e.g. for \( c = a \ast b \) we have the convolution

\[
c_k = \sum_{j=0}^{k} a_j \ast b_{k-j}
\]
higher order AD (2)

- using ODE approach permits (cheap) recurrence formulas for the coefficients, e.g. for $b = a^r$ we get:

$$\tilde{b}_k = \frac{1}{a_o} \left( r \sum_{j=1}^{k} b_{k-j} \tilde{a}_j - \sum_{j=1}^{k-1} a_{k-j} \tilde{b}_j \right) \quad \text{with } \tilde{c}_j = j c_j$$

- sine and cosine are coupled:

$$s = \sin(u) : \tilde{s}_k = \sum_{j=1}^{k} \tilde{u}_j c_{k-j} \quad \text{and} \quad c = \cos(u) : \tilde{c}_k = \sum_{j=1}^{k} -\tilde{u}_j s_{k-j}$$

- arithmetic operations are simple, e.g. for $c = a \ast b$ we have the convolution:

$$c_k = \sum_{j=0}^{k} a_j \ast b_{k-j}$$

- others see the AD book (Griewank, Walther SIAM 2008)
higher order AD (2)

- using ODE approach permits (cheap) recurrence formulas for the coefficients, e.g. for $b = a^r$ we get
  $$\tilde{b}_k = \frac{1}{a_o} \left( r \sum_{j=1}^{k} b_{k-j} \tilde{a}_j - \sum_{j=1}^{k-1} a_{k-j} \tilde{b}_j \right) \text{ with } \tilde{c}_j = j c_j$$

- sine and cosine are coupled
  $$s = \sin(u) : \tilde{s}_k = \sum_{j=1}^{k} \tilde{u}_j c_{k-j} \quad \text{and} \quad c = \cos(u) : \tilde{c}_k = -\sum_{j=1}^{k} \tilde{u}_j s_{k-j}$$

- arithmetic operations are simple, e.g. for $c = a \ast b$ we have the convolution
  $$c_k = \sum_{j=0}^{k} a_j \ast b_{k-j}$$

- others see the AD book (Griewank, Walther SIAM 2008)
- cost approx. $O(o^2)$ (arithmetic) operations
  (for first order underlying ODE up to one nonlinear univariate)
higher order AD (3)

- higher order AD preferably implemented via operator and intrinsic overloading (C++, Fortran)
higher order AD (3)

- higher order AD preferably implemented via operator and intrinsic overloading (C++, Fortran)
- want to avoid code explosion; have less emphasis on reverse mode
higher order AD (3)

- higher order AD preferably implemented via operator and intrinsic overloading (C++, Fortran)
- want to avoid code explosion; have less emphasis on reverse mode
- for example in Adol-C (Juedes, Griewank, U. in ACM TOMS 1996); library code (preprocessed & reformatted)

```c
Tres += pk-1; Targ1 += pk-1; Targ2 += pk-1;
for (l=p-1; l>=0; l--)
    for (i=k-1; i>=0; i--)
        {  
            *Tres = dp_T0[arg1]**Targ2-- + *Targ1--*dp_T0[arg2];
            Targ1OP = Targ1-i+1;
            Targ2OP = Targ2;
            for (j=0;j<i;j++)
                {  
                    *Tres += (*Targ1OP++) * (*Targ2OP--);
                }
            Tres--;
        }
    Tres--;
}
dp_T0[res] = dp_T0[arg1] * dp_T0[arg2];
```
higher order AD (3)

◊ higher order AD preferably implemented via operator and intrinsic overloading (C++, Fortran)

◊ want to avoid code explosion; have less emphasis on reverse mode

◊ for example in Adol-C (Juedes, Griewank, U. in ACM TOMS 1996); library code (preprocessed & reformatted)

```c
Tres += pk-1; Targ1 += pk-1; Targ2 += pk-1;
for (l=p-1; l>=0; l--)
  for (i=k-1; i>=0; i--)
  {
    *Tres = dp_T0[arg1]**Targ2--; + *Targ1--; dp_T0[arg2];
    Targ1OP = Targ1-i+1;
    Targ2OP = Targ2;
    for (j=0; j<i; j++)
    {
      *Tres += (*Targ1OP++) * (*Targ2OP--);
    }
    Tres--; 
  }
}

dp_T0[res] = dp_T0[arg1] * dp_T0[arg2];
```

◊ uses a work array and various pointers into it; the indices res, arg1, arg2 have been previously recorded; p = number of directions, k = derivative order makes compiler optimization difficult etc.; various AD tools
tools (i)

- special purpose tools: COSY, AD for R, Matlab
tools (i)

- special purpose tools: COSY, AD for R, Matlab
- general purpose tools: Adol-C, AD02, CppAD, ...
tools (i)

- special purpose tools: COSY, AD for R, Matlab
- general purpose tools: Adol-C, AD02, CppAD, ...
- ... with emphasis on performance - Rapsodia (Charpentier, U.; OMS 2009) - example of generated code

```plaintext
r.v = a.v * b.v;
r.d1_1 = a.v * b.d1_1 + a.d1_1 * b.v;
r.d1_2 = a.v * b.d1_2 + a.d1_1 * b.d1_1 + a.d1_2 * b.v;
r.d1_3 = a.v * b.d1_3 + a.d1_1 * b.d1_2 + a.d1_2 * b.d1_1 + a.d1_3 * b.v;
r.d2_1 = a.v * b.d2_1 + a.d2_1 * b.v;
r.d2_2 = a.v * b.d2_2 + a.d2_1 * b.d2_1 + a.d2_2 * b.v;
r.d2_3 = a.v * b.d2_3 + a.d2_1 * b.d2_2 + a.d2_2 * b.d2_1 + a.d2_3 * b.v;
```
tools (i)

- special purpose tools: COSY, AD for R, Matlab
- general purpose tools: Adol-C, AD02, CppAD, ...
- ... with emphasis on performance - Rapsodia (Charpentier, U.; OMS 2009) - example of generated code

\[
\begin{align*}
  r.v &= a.v \times b.v; \\
  r.d1_1 &= a.v \times b.d1_1 + a.d1_1 \times b.v; \\
  r.d1_2 &= a.v \times b.d1_2 + a.d1_1 \times b.d1_1 + a.d1_2 \times b.v; \\
  r.d1_3 &= a.v \times b.d1_3 + a.d1_1 \times b.d1_2 + a.d1_2 \times b.d1_1 + a.d1_3 \times b.v; \\
  r.d2_1 &= a.v \times b.d2_1 + a.d2_1 \times b.v; \\
  r.d2_2 &= a.v \times b.d2_2 + a.d2_1 \times b.d2_1 + a.d2_2 \times b.v; \\
  r.d2_3 &= a.v \times b.d2_3 + a.d2_1 \times b.d2_2 + a.d2_2 \times b.d2_1 + a.d2_3 \times b.v;
\end{align*}
\]

- C++ active types called: RAfloatS, RAfloatD
- in Fortran: RArealS, RArealD, RAcomplexS, RAcomplexD
- are flat data structures with fields \textit{v} and \textit{d1_1}...\textit{d2_3}
- code in Fortran: replace "." with "%"
- most differences are in the wrapping (also generated because of number the of interfaces, especially for Fortran)
Rapsodia Use Example

```cpp
#include <iostream>
#include <cmath>

int main(void){

    double x,y;
    // the point at which we execute
    x=0.3;

    // compute sine
    y=sin(x);
    // print it
    std::cout << "y=" << y << std::endl;

    return 0; }
```
Rapsodia Use Example

```c++
#include <iostream>
#include <cmath>

int main(void){

double x,y;
// the point at which we execute
x=0.3;

// compute sine
y=sin(x);
// print it
std::cout << "y=\"" << y << std::endl;

return 0; }
```

- figure out what to compute
- generate the library:
  `generate -d 2 -o 3 -c Rlib`
Rapsodia Use Example

```cpp
#include <iostream>
#include <cmath>
#include "RAinclude.ipp"

int main(void){

    RAfloatD x,y;
    // the point at which we execute
    x=0.3;

    // compute sine
    y=sin(x);
    // print it
    std::cout << "y=" << y.v << std::endl;

    return 0; }
```

- figure out what to compute
- generate the library:
  `generate -d 2 -o 3 -c Rlib`
- adjust the types/references
Rapsodia Use Example

```cpp
#include <iostream>
#include <cmath>
#include "RAinclude.ipp"
int main(void){
    int i,j;
    const int directions=2;
    const int order=3;
    RAfloatD x,y;
    // the point at which we execute
    x=0.3;
    // initialize the input coefficients
    // in the 2 directions
    for( i=0;i<directions;i++ ) {
        for( j=0;j<order; j++ ) {
            if (j==0) x.set(i+1,j+1,0.1*(i+1));
            else x.set(i+1,j+1,0.0);
        }
    }
    // compute sine
    y=sin(x);
    // print it
    std::cout << "y=" << y.v << std::endl;
    // get the output Taylor coefficients
    // for each of the 2 directions
    for( i=0;i<directions;i++ ) {
        for( j=0;j<order; j++ ) {
            std::cout<<"y["<<i+1<<","<<j+1<<"]="
                << y.get(i+1,j+1)
                << std::endl;
        }
    }
    return 0; }
```

- figure out what to compute
- generate the library:
  ```bash
generate -d 2 -o 3 -c Rlib
```
- adjust the types/references
- augment the “driver”
```
#include <iostream>
#include <cmath>
#include "RAinclude.ipp"
int main(void){
    int i,j;
    const int directions=2;
    const int order=3;
    RAfloatD x,y;
    // the point at which we execute
    x=0.3;
    // initialize the input coefficients
    // in the 2 directions
    for( i=0;i<directions;i++) {
        for( j=0;j<order; j++) {
            if (j==0) x.set(i+1,j+1,0.1*(i+1));
            else x.set(i+1,j+1,0.0);
        }
    }
    // compute sine
    y=sin(x);
    // print it
    std::cout << "y="<< y.v << std::endl;
    // get the output Taylor coefficients
    // for each of the 2 directions
    for( i=0;i<directions;i++) {
        for( j=0;j<order; j++) {
            std::cout<<"y["<<i+1<<","<<j+1<<"]=" << y.get(i+1,j+1) << std::endl;
        }
    }
    return 0; }
```
multivariate derivatives

have $n$ inputs, coefficient multi-indices track differentiation with respect to individual inputs; exploit symmetry

◊ direct w multi index management: COSY, AD02,..
multivariate derivatives

have \( n \) inputs, coefficient multi-indices track differentiation with respect to individual inputs; exploit symmetry

- direct w multi index management: COSY, AD02,..
- univariate + interpolation: Adol-C, Rapsodia
  (Griewank, U., Walther, Math. of Comp. 2000)
multivariate derivatives

have $n$ inputs, coefficient multi-indices track differentiation with respect to individual inputs; exploit symmetry

- direct with multi-index management: COSY, AD02, ..
- for all tensors up to order $o$ and $n$ inputs one needs $d \equiv \binom{n+o-1}{o}$ directions
multivariate derivatives

have $n$ inputs, coefficient multi-indices track differentiation with respect to individual inputs; exploit symmetry

- direct w multi index management: COSY, AD02,..
- for all tensors up to order $o$ and $n$ inputs one needs $d \equiv \binom{n+o-1}{o}$ directions
- the directions are the multi-indices $t \in \mathbb{N}_0^n$, where each $t_i, i = 1 \ldots n$ represents the derivative order with respect to input $x_i$
multivariate derivatives
have $n$ inputs, coefficient multi-indices track differentiation with respect to individual inputs; exploit symmetry

- direct w multi index management: COSY, AD02,..
- univariate + interpolation: Adol-C, Rapsodia
  (Griewank,U., Walther, Math. of Comp. 2000)
- for all tensors up to order $o$ and $n$ inputs one needs $d \equiv \binom{n+o-1}{o}$ directions
- the directions are the multi-indices $t \in \mathbb{N}_0^n$, where each $t_i, i = 1 \ldots n$ represents the derivative order with respect to input $x_i$
- exploits symmetry - e.g., the two Hessian elements $H_{12} = \frac{\partial^2}{\partial x_1 \partial x_2}$ and $H_{21} = \frac{\partial^2}{\partial x_2 \partial x_1}$ are both represented by $t = (1, 1)$. 

multivariate derivatives

have $n$ inputs, coefficient multi-indices track differentiation with respect to individual inputs; exploit symmetry

- direct w multi index management: COSY, AD02,..
- for all tensors up to order $o$ and $n$ inputs one needs $d \equiv \binom{n+o-1}{o}$ directions
- the directions are the multi-indices $t \in \mathbb{N}_0^n$, where each $t_i, i = 1 \ldots n$ represents the derivative order with respect to input $x_i$
- exploits symmetry - e.g., the two Hessian elements $H_{12} = \frac{\partial^2}{\partial x_1 \partial x_2}$ and $H_{21} = \frac{\partial^2}{\partial x_2 \partial x_1}$ are both represented by $t = (1, 1)$.
- interpolation coefficients are precomputed
multivariate derivatives

have $n$ inputs, coefficient multi-indices track differentiation with respect to individual inputs; exploit symmetry

- direct w multi index management: COSY, AD02,..
- for all tensors up to order $o$ and $n$ inputs one needs $d \equiv \binom{n+o-1}{o}$ directions
- the directions are the multi-indices $t \in \mathbb{N}_0^n$, where each $t_i, i = 1 \ldots n$ represents the derivative order with respect to input $x_i$
- exploits symmetry - e.g., the two Hessian elements $H_{12} = \frac{\partial^2}{\partial x_1 \partial x_2}$ and $H_{21} = \frac{\partial^2}{\partial x_2 \partial x_1}$ are both represented by $t = (1, 1)$.
- interpolation coefficients are precomputed
- practical advantage can be observed already for small $o > 3$
multivariate derivatives

have \( n \) inputs, coefficient multi-indices track differentiation with respect to individual inputs; exploit symmetry

- direct with multi index management: COSY, AD02, ...
- for all tensors up to order \( o \) and \( n \) inputs one needs \( d \equiv \binom{n+o-1}{o} \) directions
  - the directions are the multi-indices \( t \in \mathbb{N}_0^n \), where each \( t_i, i = 1 \ldots n \) represents the derivative order with respect to input \( x_i \)
  - exploits symmetry - e.g., the two Hessian elements \( H_{12} = \frac{\partial^2}{\partial x_1 \partial x_2} \) and \( H_{21} = \frac{\partial^2}{\partial x_2 \partial x_1} \) are both represented by \( t = (1, 1) \).
- interpolation coefficients are precomputed
- practical advantage can be observed already for small \( o > 3 \)
- interpolation error is typically negligible except in some cases; use modified schemes (Neidinger 2004 - )
Rapsodia vs AD02

run time for derivative tensors of an ocean acoustics model;
DISCLAIMER: big advantage mostly due to univariate propagation!

| o  | n  | AD02 g95 | ifort | NAG | -O3 | -O2 | -O2 | -O4 | Rapsodia g95 | ifort | NAG | -O3 | -O2 | -O2 | -O4 |
|----|----|---------|------|-----|-----|-----|-----|-----|-------|------|-----|-----|-----|-----|-----|-----|
| 2  | 5  | 0.599   | 0.460| 0.543| 0.658| 15  | 15  | 0.072| 0.106| 0.087| 0.086|
| 4  | 3  | 40.97   | 11.97| 13.67| 14.41| 15  | 15  | 0.161| 0.255| 0.181| 0.176|
| 6  | 3  | 185.4   | 58.88| 73.63| 71.21| 14  | 28  | 0.514| 0.794| 0.538| 0.515|
| 8  | 2  | 105.8   | 36.39| 45.41| 41.56| 9   | 9   | 0.250| 0.366| 0.262| 0.257|
| 8  | 3  | 651.1   | *    | 289.8| 285.2| 15  | 45  | 1.157| 1.762| 1.172| 1.101|
|13  | 3  | +       | *    | +    | +    | 10  | 105 | 5.677| 8.656| 5.673| 5.638|

◊ $o =$ derivative order, $n =$ number of inputs
◊ $+$ = we did not wait for completion; * = aborted because of lack of memory;
◊ to see the difference to loops we had to hand-write our own test lib
run time ratios of Rapsodia vs. hand written library with loops over PARAMETERized $o$ and $d^*$
Rapsodia vs Adol-C

- simple model of volcanic eruption
- small set of active variables
- for the test: repeated evaluations
- R1: Rapsodia
- R2: Rapsodia inlined
- A1: hov_forward
- A2: taping + hov_forward
- Note: no “inline” directive for Fortran, need to rely on interprocedural optimization
Parallelization

- outer loop over $d$ directions
- inner loop(s) over derivative order $o$
- identical amount of work in each direction
- all coefficients depend only on operation argument (result)
- no dependency between coefficients of different directions
- previously investigated with OpenMP by Bücker et al.
- only experimental prototypes (reuse?)
- have multicore hardware
- Can we parallelize:
  - within the library (w/o user code changes)?
  - models with side effects?

  to parallelize Rapsodia - limit the unrolling of the outer loop
limited unrolling

also aims at **constraining code bloat**, can help compiler optimization

Example: unrolled code for 4 directions:

\[
\begin{align*}
\text{r} \text{v} &= \text{a} \text{v} \times \text{b} \text{v} \\
\text{r} \text{d} \text{l} \text{1} &= \text{a} \text{v} \times \text{b} \text{d} \text{l} \text{1} + \text{a} \text{d} \text{l} \text{1} \times \text{b} \text{v} \\
\text{r} \text{d} \text{l} \text{2} &= \text{a} \text{v} \times \text{b} \text{d} \text{l} \text{2} + \text{a} \text{d} \text{l} \text{1} \times \text{b} \text{d} \text{l} \text{1} + \text{a} \text{d} \text{l} \text{2} \times \text{b} \text{v} \\
\text{r} \text{d} \text{l} \text{3} &= \text{a} \text{v} \times \text{b} \text{d} \text{l} \text{3} + \text{a} \text{d} \text{l} \text{1} \times \text{b} \text{d} \text{l} \text{2} + \text{a} \text{d} \text{1} \times \text{b} \text{d} \text{l} \text{1} + \text{a} \text{d} \text{1} \times \text{b} \text{d} \text{1} + \text{a} \text{d} \text{1} \times \text{b} \text{v} \\
\text{r} \text{d} \text{l} \text{4} &= \text{a} \text{v} \times \text{b} \text{d} \text{l} \text{4} + \text{a} \text{d} \text{l} \text{1} \times \text{b} \text{d} \text{l} \text{2} + \text{a} \text{d} \text{1} \times \text{b} \text{d} \text{l} \text{1} + \text{a} \text{d} \text{1} \times \text{b} \text{d} \text{1} + \text{a} \text{d} \text{1} \times \text{b} \text{v}
\end{align*}
\]

vs. partially unrolled for 4 directions using 2 slices; stay **flat within slice**

\[
\begin{align*}
\text{r} \text{v} &= \text{a} \text{v} \times \text{b} \text{v} \\
\text{d} \text{o} i &= 1, 2, 1 \\
\text{r} \text{s} \text{(i)} \text{d} \text{l} \text{1} &= \text{a} \text{v} \times \text{b} \text{s}(i) \text{d} \text{l} \text{1} + \text{a} \text{s}(i) \text{d} \text{l} \text{1} \times \text{b} \text{v} \\
\text{r} \text{s} \text{(i)} \text{d} \text{l} \text{2} &= \text{a} \text{v} \times \text{b} \text{s}(i) \text{d} \text{l} \text{2} + \text{a} \text{s}(i) \text{d} \text{l} \text{1} \times \text{b} \text{s}(i) \text{d} \text{l} \text{1} + \text{a} \text{s}(i) \text{d} \text{l} \text{2} \times \text{b} \text{v} \\
\text{r} \text{s} \text{(i)} \text{d} \text{l} \text{3} &= \text{a} \text{v} \times \text{b} \text{s}(i) \text{d} \text{l} \text{3} + \text{a} \text{s}(i) \text{d} \text{l} \text{1} \times \text{b} \text{s}(i) \text{d} \text{l} \text{2} + \text{a} \text{s}(i) \text{d} \text{l} \text{2} \times \text{b} \text{s}(i) \text{d} \text{l} \text{1} + \text{a} \text{s}(i) \text{d} \text{l} \text{3} \times \text{b} \text{v} \\
\text{r} \text{s} \text{(i)} \text{d} \text{l} \text{4} &= \text{a} \text{v} \times \text{b} \text{s}(i) \text{d} \text{l} \text{4} + \text{a} \text{s}(i) \text{d} \text{l} \text{1} \times \text{b} \text{s}(i) \text{d} \text{l} \text{2} + \text{a} \text{s}(i) \text{d} \text{l} \text{2} \times \text{b} \text{s}(i) \text{d} \text{l} \text{1} + \text{a} \text{s}(i) \text{d} \text{l} \text{3} \times \text{b} \text{v}
\end{align*}
\]
limited unrolling 2

- main problem: can only slice directions (not order),
- iteration complexity differs between ops.
- impact on register allocation differs between compilers/platforms
What is a good choice for the number of slices?
limited unrolling 4

contours of optimal slices for test case with

1. mostly non-linear
2. mix linear/non-linear
3. mostly linear operations
## Limited Unrolling 5

<table>
<thead>
<tr>
<th>( (o, d) )</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>11</td>
<td>2</td>
<td>13</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>2</td>
<td>13</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>11</td>
<td>4</td>
<td>13</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>9</td>
<td>2</td>
<td>11</td>
<td>6</td>
<td>13</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>9</td>
<td>2</td>
<td>11</td>
<td>2</td>
<td>13</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>2</td>
<td>13</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>11</td>
<td>2</td>
<td>13</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>9</td>
<td>5</td>
<td>11</td>
<td>2</td>
<td>13</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>2</td>
<td>11</td>
<td>4</td>
<td>13</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>14</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>3</td>
<td>10</td>
<td>11</td>
<td>2</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>11</td>
<td>2</td>
<td>13</td>
<td>7</td>
<td>15</td>
</tr>
</tbody>
</table>

![Diagram](image-url)
Asynchronous parallel loops

OpenMP direction loop parallelization is not efficient on operator level
so lets do something else (i.e. much less convenient than OpenMP)
Asynchronous parallel loops

OpenMP direction loop parallelization is not efficient on operator level
so lets do something else (i.e. much less convenient than OpenMP)
Asynchronous parallel loops

OpenMP direction loop parallelization is not efficient on operator level
so let's do something else (i.e. much less convenient than OpenMP)

use of open portable atomics lib for spinlocks is crucial
reverse mode with adjoints

- same association model
- take a point \((a_0, b_0, c_0)\), compute \(y\), pick a weight \(\overline{y}\)
- for each \(v = \phi(w, u)\) propagate backward

\[
\begin{align*}
\bar{w}^+ &= \frac{\partial \phi}{\partial w} \bar{v}; \\
\bar{u}^+ &= \frac{\partial \phi}{\partial u} \bar{v}; \\
\bar{v} &= 0
\end{align*}
\]

backward propagation code appended:

\[
\begin{align*}
t1 &= a*b \\
p1 &= \cos(t1) \\
t2 &= \sin(t1) \\
y &= t2*c
\end{align*}
\]
reverse mode with adjoints

- same association model
- take a point \((a_0, b_0, c_0)\), compute \(y\), pick a weight \(\bar{y}\)
- for each \(v = \phi(w, u)\) propagate backward
  \[\bar{w}^+ = \frac{\partial \phi}{\partial w} \bar{v}; \quad \bar{u}^+ = \frac{\partial \phi}{\partial u} \bar{v}; \quad \bar{v} = 0\]

backward propagation code appended:

\[
\begin{align*}
t1 &= a*b \\
p1 &= \cos(t1) \\
t2 &= \sin(t1) \\
y &= t2*c \\
d_c &= t2*d_y
\end{align*}
\]
reverse mode with adjoints

- same association model
- take a point \((a_0, b_0, c_0)\), compute \(y\), pick a weight \(\bar{y}\)
- for each \(v = \phi(w, u)\) propagate backward
\[
\bar{w}^+ = \frac{\partial \phi}{\partial w} \bar{v}; \quad \bar{u}^+ = \frac{\partial \phi}{\partial u} \bar{v}; \quad \bar{v} = 0
\]

backward propagation code appended:
\[
\begin{align*}
t1 &= a \times b \\
p1 &= \cos(t1) \\
t2 &= \sin(t1) \\
y &= t2 \times c \\
d_{c} &= t2 \times d_y \\
d_{t2} &= c \times d_y
\end{align*}
\]
reverse mode with adjoints

- same association model
- take a point \((a_0, b_0, c_0)\), compute \(y\), pick a weight \(\bar{y}\)
- for each \(v = \phi(w, u)\) propagate backward

\[
\bar{w}^+ = \frac{\partial \phi}{\partial w} \, \bar{v}; \quad \bar{u}^+ = \frac{\partial \phi}{\partial u} \, \bar{v}; \quad \bar{v} = 0
\]

backward propagation code appended:

\[
t_1 = a \times b \\
p_1 = \cos(t_1) \\
t_2 = \sin(t_1) \\
y = t_2 \times c \\
d_c = t_2 \times d_y \\
d_{t2} = c \times d_y \\
d_y = 0
\]
reverse mode with adjoints

- same association model
- take a point \((a_0, b_0, c_0)\), compute \(y\), pick a weight \(\bar{y}\)
- for each \(v = \phi(w, u)\) propagate backward

\[
\bar{w}^+ = \frac{\partial \phi}{\partial w} \bar{v} ; \quad \bar{u}^+ = \frac{\partial \phi}{\partial u} \bar{v} ; \quad \bar{v} = 0
\]

backward propagation code appended:

\[
\begin{align*}
t_1 &= a \cdot b \\
p_1 &= \cos(t_1) \\
t_2 &= \sin(t_1) \\
y &= t_2 \cdot c \\
d_{-c} &= t_2 \cdot d_y \\
d_{t_2} &= c \cdot d_y \\
y &= 0 \\
d_{t_1} &= p_1 \cdot d_{t_2}
\end{align*}
\]
reverse mode with adjoints

- same association model
- take a point \((a_0, b_0, c_0)\), compute \(y\), pick a weight \(\bar{y}\)
- for each \(v = \phi(w, u)\) propagate backward
  \[
  \bar{w} + = \frac{\partial \phi}{\partial w} \bar{v}; \quad \bar{u} + = \frac{\partial \phi}{\partial u} \bar{v}; \quad \bar{v} = 0
  \]

backward propagation code appended:
- \(t1 = a*b\)
- \(p1 = \cos(t1)\)
- \(t2 = \sin(t1)\)
- \(y = t2*c\)
- \(d_c = t2*d_y\)
- \(d_t2 = c*d_y\)
- \(d_y = 0\)
- \(d_t1 = p1*d_t2\)
- \(d_b = a*d_t1\)
reverse mode with adjoints

- same association model
- take a point \((a_0, b_0, c_0)\), compute \(y\), pick a weight \(\bar{y}\)
- for each \(v = \phi(w, u)\) propagate backward
  \[
  \bar{w}^+ = \frac{\partial \phi}{\partial w} \bar{v}; \quad \bar{u}^+ = \frac{\partial \phi}{\partial u} \bar{v}; \quad \bar{v} = 0
  \]

backwards propagation code appended:

```plaintext
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y = t2*c
d_c = t2*d_y
d_t2 = c*d_y
d_y = 0
d_t1 = p1*d_t2
d_b = a*d_t1
d_a = b*d_t1
```
reverse mode with adjoints

- same association model
- take a point \((a_0, b_0, c_0)\), compute \(y\), pick a weight \(\bar{y}\)
- for each \(v = \phi(w, u)\) propagate backward

\[
\bar{w}+ = \frac{\partial \phi}{\partial w} \bar{v}; \quad \bar{u}+ = \frac{\partial \phi}{\partial u} \bar{v}; \quad \bar{v} = 0
\]

backward propagation code appended:

\[
\begin{align*}
t_1 &= a \cdot b \\
p_1 &= \cos(t_1) \\
t_2 &= \sin(t_1) \\
y &= t_2 \cdot c \\
d_{c} &= t_2 \cdot d_y \\
d_{t2} &= c \cdot d_y \\
d_y &= 0 \\
d_{t1} &= p_1 \cdot d_{t2} \\
d_{b} &= a \cdot d_{t1} \\
d_{a} &= b \cdot d_{t1}
\end{align*}
\]

What is in \((d_a, d_b, d_c)\)?
$(d_a, d_b, d_c)$ contains a projection

$\bar{x} = \bar{y}^T J$ computed at $x_0$
\((d_a, d_b, d_c)\) contains a projection

- \(\bar{x} = \bar{y}^T J\) computed at \(x_0\)
- for example for \(\bar{y} = 1\) we have \([\bar{a}, \bar{b}, \bar{c}] = \nabla f\)

- all gradient elements cost \(O(1)\) function evaluations
\( (d_a, d_b, d_c) \) contains a projection

\[ \bar{x} = \bar{y}^T J \] computed at \( x_0 \)

\[ \text{for example for } \bar{y} = 1 \text{ we have } [\bar{a}, \bar{b}, \bar{c}] = \nabla f \]

\[ \text{all gradient elements cost } O(1) \text{ function evaluations} \]

\[ \text{but consider when } p1 \text{ is computed and when it is used} \]
\((d_a, d_b, d_c)\) contains a projection

- \(\bar{x} = \bar{y}^T J\) computed at \(x_0\)
- for example for \(\bar{y} = 1\) we have \([\bar{a}, \bar{b}, \bar{c}] = \nabla f\)

- all gradient elements cost \(O(1)\) function evaluations
- but consider when \(p1\) is computed and when it is used
- storage requirements grow with the length of the computation
- typically mitigated by recomputation from checkpoints
\((d_a, d_b, d_c)\) contains a projection

- \(\overline{x} = \overline{y}^T J\) computed at \(x_0\)
- for example for \(\overline{y} = 1\) we have \([\overline{a}, \overline{b}, \overline{c}] = \nabla f\)

- all gradient elements cost \(O(1)\) function evaluations
- but consider when \(p1\) is computed and when it is used
- **storage requirements** grow with the length of the computation
- typically mitigated by recomputation from checkpoints

Reverse mode with Adol-C.
ADOL-C

- [http://www.coin-or.org/projects/ADOL-C.xml](http://www.coin-or.org/projects/ADOL-C.xml)

- Operator overloading creates an execution trace (also called 'tape')

Speelpenning example $y = \prod_{i} x_i$ evaluated at $x_i = \frac{i+1}{i+2}$

```c
double *x = new double[n];
double t = 1;
double y;
for(i=0; i<n; i++) {
    x[i] = (i+1.0)/(i+2.0);
    t *= x[i]; }
y = t;
delete[] x;
```
ADOL-C

- http://www.coin-or.org/projects/ADOL-C.xml
- operator overloading creates an execution trace (also called 'tape')

Speelpenning example \( y = \prod_i x_i \) evaluated at \( x_i = \frac{i+1}{i+2} \)

```c
#include "adolc.h"
adouble *x = new adouble[n];
adouble t = 1;
double y;
trace_on(1);
for(i=0; i<n; i++) {
    x[i] <<= (i+1.0)/(i+2.0);
    t *= x[i]; }
t >>= y;
trace_off();
delete[] x;
```
ADOL-C

- http://www.coin-or.org/projects/ADOL-C.xml
- operator overloading creates an execution trace (also called 'tape')

Speelpenning example \( y = \prod_{i} x_i \) evaluated at \( x_i = \frac{i+1}{i+2} \)

```cpp
#include "adolc.h"
adouble *x = new adouble[n];
adouble t = 1;
double y;
trace_on(1);
for(i=0; i<n; i++) {
    x[i] <<= (i+1.0)/(i+2.0);
    t *= x[i]; }
t >>= y;
trace_off();
delete[] x;
```

use a driver:
```cpp
gradient(tag,
    n,
    x[n],
    g[n])
```
Sidebar: preaccumulation & propagation

- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians $J$
- long program with control flow $\Rightarrow$ sequence of graphs $\Rightarrow$ sequence of $J_i$

![Diagram of expression graph]

- $t_2$
- $\sin$
- $p_1$
- $\ast$
- $c$
- $a$
- $b$
sidebar: preaccumulation & propagation

- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians $J$
- long program with control flow $\Rightarrow$ sequence of graphs $\Rightarrow$ sequence of $J_i$
Sidebar: Preaccumulation & Propagation

- Build expression graphs (limited by aliasing, typically to a basic block)
- **Preaccumulate** them to local Jacobians $\mathbf{J}$
- Long program with control flow $\Rightarrow$ sequence of graphs $\Rightarrow$ sequence of $\mathbf{J}_i$

$t3 = c*p1$
sidebar: preaccumulation & propagation

- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians $J$
- long program with control flow $\Rightarrow$ sequence of graphs $\Rightarrow$ sequence of $J_i$

```
t3 = c*p1
```
preaccumulation & propagation

- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians $J$
- long program with control flow $\Rightarrow$ sequence of graphs $\Rightarrow$ sequence of $J_i$
  
  $$
  \begin{align*}
  t3 &= c*p1 \\
  t4 &= t3*a
  \end{align*}
  $$

```
\begin{itemize}
  \item build expression graphs (limited by aliasing, typically to a basic block)
  \item **preaccumulate** them to local Jacobians $J$
  \item long program with control flow $\Rightarrow$ sequence of graphs $\Rightarrow$ sequence of $J_i$
\end{itemize}

t3 = c*p1  

\begin{itemize}
  \item build expression graphs (limited by aliasing, typically to a basic block)
  \item **preaccumulate** them to local Jacobians $J$
  \item long program with control flow $\Rightarrow$ sequence of graphs $\Rightarrow$ sequence of $J_i$
\end{itemize}

t4 = t3*a
```
sidebar: preaccumulation & propagation

- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians $\mathbf{J}$
- long program with control flow $\Rightarrow$ sequence of graphs $\Rightarrow$ sequence of $J_i$

```plaintext
  t3 = c*p1
  t4 = t3*a
```
sidebar: preaccumulation & propagation

- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians $J$
- long program with control flow $\Rightarrow$ sequence of graphs $\Rightarrow$ sequence of $J_i$

\[
\begin{align*}
t3 &= c*p1 \\
t4 &= t3*a \\
t5 &= t3*b
\end{align*}
\]
sidebar: preaccumulation & propagation

- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians $\mathbf{J}$
- long program with control flow $\Rightarrow$ sequence of graphs $\Rightarrow$ sequence of $\mathbf{J}_i$

$$
t3 = c*p1 \\
t4 = t3*a \\
t5 = t3*b
$$

- $(t5, t4, t2)$ is the preaccumulated $\mathbf{J}_i$
- $\min_{ops}(\text{preacc.})$? a combinatorial problem $\Rightarrow$ compile time AD optimization!

- forward propagation of $\dot{x}$
  $$(\mathbf{J}_k \circ \ldots \circ (\mathbf{J}_1 \circ \dot{x}) \ldots)$$

- adjoint propagation of $\bar{y}$
  $$(\ldots (\bar{y}^T \circ \mathbf{J}_k) \circ \ldots \circ \mathbf{J}_1)$$
sidebar: toy example - source transformation reverse mode

code preparation

numerical “model” program:

```fortran
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
  !$openad INDEPENDENT(x)
  y=sin(x*x)
  !$openad DEPENDENT(y)
end subroutine
```

Intro to AD - Utke - May/2013
sidebar: toy example - source transformation reverse mode

code preparation ⇒ reverse mode OpenAD pipeline

numerical “model” program:

```fortran
subroutine head(x,y)
double precision,intent(in) :: x
double precision,intent(out) :: y
!$openad INDEPENDENT(x)
y = sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

preaccumulation & store $J_i$:

```fortran
... 
oadS_0 = (X%v*X%v)
y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_4 = (oadS_2 * oadS_1)
oadS_5 = (oadS_3 * oadS_1)
oadD(oadD_ptr) = oadS_4
oadD_ptr = oadD_ptr+1
oadD(oadD_ptr) = oadS_5
oadD_ptr = oadD_ptr+1
...
```
code preparation ⇒ reverse mode OpenAD pipeline

numerical “model” program:

```
subroutine head(x,y)
  double precision, intent(in) :: x
  double precision, intent(out) :: y
  !$openad INDEPENDENT(x)
  y = sin(x*x)
  !$openad DEPENDENT(y)
end subroutine
```

preaccumulation & store $J_{ij}$:

```
... 
oadS_0 = (X\%v*X\%v)
Y\%v = SIN(oadS_0)
oadS_2 = X\%v
oadS_3 = X\%v
oadS_1 = COS(oadS_0)
oadS_4 = (oadS_2 * oadS_1)
oadS_5 = (oadS_3 * oadS_1)
oadD(oadD_ptr) = oadS_4
oadD_ptr = oadD_ptr+1
oadD(oadD_ptr) = oadS_5
oadD_ptr = oadD_ptr+1
... 
```
code preparation ⇒ reverse mode OpenAD pipeline

numerical “model” program:

```fortran
subroutine head(x,y)
    double precision, intent(in) :: x
    double precision, intent(out) :: y
    !$openad INDEPENDENT(x)
    y = sin(x*x)
    !$openad DEPENDENT(y)
end subroutine
```

preaccumulation & store $J_i$:

```fortran
...  
oadS_0 = (x%v*x%v)  
Y%v = SIN(oadS_0)  
oadS_2 = X%v  
oadS_3 = X%v  
oadS_1 = COS(oadS_0)  
oadS_4 = (oadS_2 * oadS_1)  
oadS_5 = (oadS_3 * oadS_1)  
oadD(oadD_ptr) = oadS_4  
oadD_ptr = oadD_ptr+1  
oadD(oadD_ptr) = oadS_5  
oadD_ptr = oadD_ptr+1  
...  
```
sidebar: toy example - source transformation reverse mode

code preparation ⇒ reverse mode OpenAD pipeline

numerical “model” program:

```fortran
subroutine head(x,y)
  double precision, intent(in) :: x
  double precision, intent(out) :: y
  !$openad INDEPENDENT(x)
  y = sin(x*x)
  !$openad DEPENDENT(y)
end subroutine
```

preaccumulation & store $J_i$:

```fortran
oadS_0 = X%v*X%v
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = x%v
oadS_1 = COS(oadS_0)
oadS_4 = (oadS_2 * oadS_1)
oadS_5 = (oadS_3 * oadS_1)
oadD(oadD_ptr) = oadS_4
oadD_ptr = oadD_ptr+1
oadD(oadD_ptr) = oadS_5
oadD_ptr = oadD_ptr+1
...
```

retrieve stored $J_i$ & propagate:

```fortran
oadD_ptr = oadD_ptr-1
oadS_6 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_6
oadD_ptr = oadD_ptr-1
oadS_7 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_7
Y%d = 0.0d0
...
```
sidebar: toy example - source transformation reverse mode

code preparation $\Rightarrow$ reverse mode OpenAD pipeline

numerical "model" program:

```
subroutine head(x,y)
double precision,intent(in) :: x
double precision,intent(out) :: y
  !$openad INDEPENDENT(x)
  y=sin(x*x)
  !$openad DEPENDENT(y)
end subroutine
```

preaccumulation & store $J_i$:

```
... oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_4 = (oadS_2 * oadS_1)
oadS_5 = (oadS_3 * oadS_1)
oadD(oadD_ptr) = oadS_4
oadD_ptr = oadD_ptr+1
oadD(oadD_ptr) = oadS_5
oadD_ptr = oadD_ptr+1
...
```

retrieve stored $J_i$ & propagate:

```
... oadD_ptr = oadD_ptr-1
oadS_6 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_6
oadD_ptr = oadD_ptr-1
oadS_7 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_7
Y%d = 0.0d0
...
```
sidebar: toy example - source transformation reverse mode

code preparation ⇒ reverse mode OpenAD pipeline

numerical “model” program:

```fortran
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
  !$openad INDEPENDENT(x)
  y=sin(x*x)
  !$openad DEPENDENT(y)
end subroutine
```

preaccumulation & store $J_i$:

```
...  
oadS_0 = (X%v*X%v)  
Y%v = SIN(oadS_0)  
oadS_2 = X%v  
oadS_3 = X%v  
oadS_1 = COS(oadS_0)  
oadS_4 = (oadS_2 * oadS_1)  
oadS_5 = (oadS_3 * oadS_1)  
oadD(oadD_ptr) = oadS_4  
oadD_ptr = oadD_ptr+1  
oadD(oadD_ptr) = oadS_5  
oadD_ptr = oadD_ptr+1  
...  
```

retrieve stored $J_i$ & propagate:

```
...  
oadD_ptr = oadD_ptr-1  
oadS_6 = oadD(oadD_ptr)  
X%d = X%d+Y%d*oadS_6  
oadD_ptr = oadD_ptr-1  
oadS_7 = oadD(oadD_ptr)  
X%d = X%d+Y%d*oadS_7  
Y%d = 0.0d0  
...  ```
sidebar: toy example - source transformation reverse mode

code preparation ⇒ reverse mode OpenAD pipeline
⇒ adapt the driver routine

numerical “model” program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
  !$openad INDEPENDENT(x)
  y=sin(x*x)
  !$openad DEPENDENT(y)
end subroutine
```

driver modified for reverse mode:

```
program driver
  use OAD_active
  implicit none
  external head
  type(active):: x, y
  x%v=.5D0
  y%d=1.0
  our_rev_mode%tape=.TRUE.
  call head(x,y)
  print *, "F(1,1)=",x%d
end program driver
```

preaccumulation & store $J_i$:  

```
...  
oadS_0 = (X%v*X%v)
oadS_1 = COS(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_4 = (oadS_2 * oadS_1)
oadS_5 = (oadS_3 * oadS_1)
oadD(oadD_ptr) = oadS_4
oadD_ptr = oadD_ptr+1
oadD(oadD_ptr) = oadS_5
oadD_ptr = oadD_ptr+1
...  
```

retrieve stored $J_i$ & propagate:

```
...  
oadD_ptr = oadD_ptr-1
oadS_6 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_6
oadD_ptr = oadD_ptr-1
oadS_7 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_7
Y%d = 0.0d0
...  
```
code preparation ⇒ reverse mode OpenAD pipeline
⇒ adapt the driver routine

numerical “model” program:

```fortran
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
  !$openad INDEPENDENT(x)
  y=sin(x*x)
  !$openad DEPENDENT(y)
end subroutine
```

driver modified for reverse mode:

```fortran
program driver
  use OAD_active
  implicit none
  external head
  type(active):: x, y
  x%v=.5D0
  y%d=1.0
  our_rev_mode%tape=.TRUE.
  call head(x,y)
  print *, "F(1,1)=", x%d
end program driver
```

preaccumulation & store $J_i$:

```fortran
... oadS_0 = (X%v*X%v)
  Y%v = SIN(oadS_0)
  oadS_2 = X%v
  oadS_3 = X%v
  oadS_1 = COS(oadS_0)
  oadS_4 = (oadS_2 * oadS_1)
  oadS_5 = (oadS_3 * oadS_1)
  oadD(oadD_ptr) = oadS_4
  oadD_ptr = oadD_ptr+1
  oadD(oadD_ptr) = oadS_5
  oadD_ptr = oadD_ptr+1
...```

retrieve stored $J_i$ & propagate:

```fortran
... oadD_ptr = oadD_ptr-1
  oadS_6 = oadD(oadD_ptr)
  X%d = X%d+Y%d*oadS_6
  oadD_ptr = oadD_ptr-1
  oadS_7 = oadD(oadD_ptr)
  X%d = X%d+Y%d*oadS_7
  Y%d = 0.0d0
...```
forward vs. reverse

- simplest rule: given $y = f(x) : \mathbb{R}^n \mapsto \mathbb{R}^m$ use reverse if $n \gg m$ (gradient)
- what if $n \approx m$ and large
  - want only projections, e.g. $J \dot{x}$
  - sparsity (e.g. of the Jacobian)
  - partial separability (e.g. $f(x) = \sum(f_i(x_i)), x_i \in D_i \subseteq D \ni x$)
  - intermediate interfaces of different size
- the above may make forward mode feasible (projection $\bar{y}^T J$ requires reverse)
- higher order tensors (practically feasible for small $n$) $\rightarrow$ forward mode (reverse mode saves factor $n$ in effort only once)
- this determines overall propagation direction, not necessarily the local preaccumulation (combinatorial problem)
source transformation vs. operator overloading

- complicated implementation of tools
- especially for reverse mode
- full front end, back end, analysis
- efficiency gains from
  - compile time AD optimizations
  - activity analysis
  - explicit control flow reversal
- source transformation based type change & overloaded operators appropriate for higher-order derivatives.
- efficiency depends on analysis accuracy

simple tool implementation
- reverse mode: generate & reinterpret an execution trace → inefficient
- implemented as a library
- efficiency gains from:
  - runtime AD optimization
  - optimized library
  - inlining (for low order)
- manual type change
  - formatted I/O, allocation,…
  - matching signatures (Fortran)
  - easier with templates

higher-order derivatives ⇒ source transformation based type change + overloaded operators.
Reversal Schemes

- why it is needed
- major modes
- alternatives
recap: store intermediate values / partials
storage also needed for control flow trace and addresses...

original CFG $\Rightarrow$ record a path through the CFG $\Rightarrow$ adjoint CFG

often cheap with **structured control flow and simple address computations** (e.g. index from loop variables)

unstructured control flow and **pointers** are expensive
trace all at once = global split mode

subroutine A()
  call B();
  call D();
  call B();
end subroutine A

subroutine B()
  call C()
end subroutine B

subroutine C()
  call E()
end subroutine C

- \( S^n \) is the \( n \)-th invocation of subroutine \( S \)
- subroutine call
- run forward
- store checkpoint
- run forward and tape
- order of execution
- restore checkpoint
- run adjoint

\( \Box \) have memory limits - need to create tapes for short sections in reverse order

\( \Box \) subroutine is “natural” checkpoint granularity, different mode...
trace one SR at a time = global joint mode

taping-adjoint pairs
checkpoint-recompute pairs
the deeper the call stack - the more recomputations
(unimplemented solution - result checkpointing)
familiar tradeoff between storing and recomputation at a higher level but in theory can be all unified.
in practice - hybrid approaches...
use of checkpointing to mitigate storage requirements

11 iters.
use of checkpointing to mitigate storage requirements

11 iters., memory limited to one iter. of storing $J_i$

run forward, store the last step, and adjoin
use of checkpointing to mitigate storage requirements

- 11 iters., memory limited to one iter. of storing $J_i$ & 3 checkpoints
- run forward, store the last step, and adjoin
use of checkpointing to mitigate storage requirements

- 11 iters., memory limited to one iter. of storing $J_i$ & 3 checkpoints
- run forward, store the last step, and adjoin
- restore checkpoints and recompute
The use of checkpointing to mitigate storage requirements:

- 11 iterations, memory limited to one iteration of storing $J_i$ & 3 checkpoints.
- Run forward, store the last step, and adjoin.
- Restore checkpoints and recompute (2 levels in this example).
- Reuse checkpoint space as it becomes available for new checkpoints.
use of checkpointing to mitigate storage requirements

diamond 11 iters., memory limited to one iter. of storing $J_i$ & 3 checkpoints

diamond run forward, store the last step, and adjoin

diamond restore checkpoints and recompute (2 levels in this example)

diamond reuse checkpoint space as it becomes available for new checkpoints
use of checkpointing to mitigate storage requirements

- 11 iters., memory limited to one iter. of storing $J_i$ & 3 checkpoints
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints
use of checkpointing to mitigate storage requirements

11 iters., memory limited to one iter. of storing $J_i$ & 3 checkpoints
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints
use of checkpointing to mitigate storage requirements

ifers., memory limited to one iter. of storing $J_i$ & 3 checkpoints

run forward, store the last step, and adjoin

restore checkpoints and recompute (2 levels in this example)

reuse checkpoint space as it becomes available for new checkpoints
use of checkpointing to mitigate storage requirements

- 11 iters., memory limited to one iter. of storing $J_i$ & 3 checkpoints
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints
use of checkpointing to mitigate storage requirements

- 11 iters., memory limited to one iter. of storing $J_i$ & 3 checkpoints
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints
use of checkpointing to mitigate storage requirements

- 11 iters., memory limited to one iter. of storing $J_i$ & 3 checkpoints
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints
use of checkpointing to mitigate storage requirements

- 11 iters., memory limited to one iter. of storing $J_i$ & 3 checkpoints
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints
use of checkpointing to mitigate storage requirements

- 11 iters., memory limited to one iter. of storing $J_i$ & 3 checkpoints
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints
- optimal (binomial) scheme encoded in revolve; C++ and F9X implementation
MPI - parallelization

◊ simple MPI program needs 6 calls:

```c
mpi_init     // initialize the environment
mpi_comm_size // number of processes in the communicator
mpi_comm_rank // rank of this process in the communicator
mpi_send     // send (blocking)
mpi_recv     // receive (blocking)
mpi_finalize // cleanup
```

◊ example adjoining blocking communication between 2 processes and interpret as assignments

```
c = a;
b = d;
P1 recv(c)
SEND(d) recv(b)
SEND(a)
```

◊ use the communication graph as model
options for non-blocking reversal

- ensure correctness $\Rightarrow$ use nonblocking calls in the adjoint

- transformations are provably correct

- **convey context** $\Rightarrow$ enables a transformation recipe per call (extra parameters and/or split interfaces into variants)

- promises to not read or **write** the respective buffer
Collective communication

- Example: Reduction followed by broadcast
  \[ b_0 = \sum a_i \text{ followed by } b_i = b_0 \forall i \]

- Conceptually simple; reduce ⏫ bcast and bcast ⏫ reduce

- Adjoint: \( t_0 = \sum \bar{b}_i \text{ followed by } \bar{a}_i += t_0 \forall i \)

- Has single transformation points (connected by hyper communication edge)

- Efficiency for product reduction because of increment
  \[ \bar{a}_i += \frac{\partial b_0}{\partial a_i} t_0, \forall i \]
AD and Language Features: not-so-structured control flow

- think - goto, exceptions, early return, ....
AD and Language Features: not-so-structured control flow

- think - goto, exceptions, early return, ....
- structured control flow is characterizable by some control flow graph properties; permits structured reverse control flow!
AD and Language Features: not-so-structured control flow

- think - goto, exceptions, early return, ....
- structured control flow is characterizable by some control flow graph properties; permits **structured reverse control flow**!
- simple view: **use only loops and branches** and no other control flow constructs (some things are easily fixable though, e.g. turn exits into some error routine call, ...)

Intro to AD - Utke - May/2013
AD and Language Features: not-so-structured control flow

- Think - goto, exceptions, early return, ...
- Structured control flow is characterizable by some control flow graph properties; permits **structured reverse control flow**!
- Simple view: **use only loops and branches** and no other control flow constructs (some things are easily fixable though, e.g. turn exits into some error routine call, ...)
- Example: early return from within a loop (CFG left, adjoint CFG right)
AD and Language Features: not-so-structured control flow

- think - goto, exceptions, early return, ....
- structured control flow is characterizable by some control flow graph properties; permits **structured reverse control flow**!
- simple view: **use only loops and branches** and no other control flow constructs (some things are easily fixable though, e.g. turn exits into some error routine call, ...)
- example: early return from within a loop (CFG left, adjoint CFG right)

- OK without the red arrow
- some jumps are not permitted
- unstruct. control flow $\not\supset$ compiler opt.
- Fortran fallback: trace/replay enumerated basic blocks; for C++: hoist local variables inst.;
- exceptions: catch to undo try side effects
Checkpointing and non-contiguous data

checkpointing = saving program data (to disk)

- “contiguous” data: scalars, arrays (even with stride > 1), strings, structures,...
- “non-contiguous” data: linked lists, rings, structures with pointers,...
- checkpointing is very similar to “serialization”
- Problem: decide when to follow a pointer and save what we point to
  - unless we have extra info this is not decidable at source transformation time
- possible fallback: runtime bookkeeping of things that have been saved (is computationally expensive, cf. python copy.deepcopy or pickle)
Semantically Ambiguous Data

- e.g. `union` (or its Fortran counterpart `equivalence`)
  - data dependence analysis: dependencies propagate from one variable to all equivalenced variables
  - “activity” (i.e. the need to generate adjoint code for a variable) leaks to all equivalenced variables whether appropriate or not
  - certain technical problems with the use of an active type (as in OpenAD)

- work-arrays (multiple, 0 semantically different fields are put into a (large) work-array); access via index offsets
  - data dependence analysis: there is `array section analysis` but in practice it is often not good enough to reflect the implied semantics
  - the entire work-array may become active / checkpointed

- programming patterns where the analysis has no good way to track the data dependencies:
  - data transfer via files (don’t really want to assume all read data depends on all written data)
  - non-structured interfaces: exchanging data that is identified by a “key” but passed as `void*` or something equivalent.
Recomputation from Checkpoints and Program Resources

think of memory, file handles, sockets, MPI communicators,...

- problem when resource allocation and deallocation happen in different partitions (see hierarchical checkpointing scheme in the figure on the left)
- current AD checkpointing does not track resources
- dynamic memory is “easy” as long as nothing is deallocated before the adjoint sweep is complete.
object-oriented syntactic encapsulation

- syntactic encapsulation of data and methods
object-oriented syntactic encapsulation

- syntactic encapsulation of data and methods
- Fortran/C recipes recommend extraction of “numerical core”, filtering out init/cleanup/debug code.
object-oriented syntactic encapsulation

- syntactic encapsulation of data and methods
- Fortran/C recipes recommend extraction of “numerical core”, filtering out init/cleanup/debug code.
- extraction would require (atypical) encapsulation based on control flow
object-oriented syntactic encapsulation

- syntactic encapsulation of data and methods
- Fortran/C recipes recommend extraction of “numerical core”, filtering out init/cleanup/debug code.
- extraction would require (atypical) encapsulation based on control flow
- selective augmentation for derivatives vs. deeply structured data types and low level containers
object-oriented syntactic encapsulation

- syntactic encapsulation of data and methods
- Fortran/C recipes recommend extraction of “numerical core”, filtering out init/cleanup/debug code.
- extraction would require (atypical) encapsulation based on control flow
- selective augmentation for derivatives vs. deeply structured data types and low level containers
object-oriented syntactic encapsulation

- syntactic encapsulation of data and methods
- Fortran/C recipes recommend extraction of “numerical core”, filtering out init/cleanup/debug code.
- extraction would require (atypical) encapsulation based on control flow
- selective augmentation for derivatives vs. deeply structured data types and low level containers

collaboration with Laurent Hascoët (Tapenade) at INRIA Sophia-Antipolis
usage concerns (1)

- availability of AD tools (forward, reverse, efficiency implications)
usage concerns (1)

- availability of AD tools (forward, reverse, efficiency implications)
- restrict tool use to volatile parts?
  - access to the code for all components
  - consider manual adjoints for static parts
  - consider the math (solvers, iterative processes, sparsity, self adjointedness, convergence criteria ...); **avoid** differentiating some algorithm portions
usage concerns (1)

diamond availability of AD tools (forward, reverse, efficiency implications)

diamond restrict tool use to volatile parts?

- access to the code for all components
- consider manual adjoints for static parts
- consider the math (solvers, iterative processes, sparsity, self adjointedness, convergence criteria ...); avoid differentiating some algorithm portions

diamond effort for

- initial implementation
- validation
- efficiency (generally - what is good for the adjoint is good for the model)
- implement volatile parts with a domain-specific language (cf. ampl)?
- robustness
adjoint robustness and efficiency are impacted by

- capability for data flow and (structured) control flow reversal
- code analysis accuracy
usage concerns (2)

◊ adjoint robustness and efficiency are impacted by
  ♦ capability for data flow and (structured) control flow reversal
  ♦ code analysis accuracy
  ♦ use of certain programming language features
  ♦ use of certain inherently difficult to handle patterns
adjoint robustness and efficiency are impacted by

- capability for data flow and (structured) control flow reversal
- code analysis accuracy
- use of certain programming language features
- use of certain inherently difficult to handle patterns
- smoothness of the model, utility of the cost function
is the model smooth?

- $y = \text{abs}(x)$; gives a kink
is the model smooth?

- $y = \text{abs}(x)$; gives a kink
- $y = (x > 0) ? 3x : 2x + 2$; gives a discontinuity
is the model smooth?

- $y=\text{abs}(x)$; gives a kink
- $y=(x>0)?3*x:2*x+2$; gives a discontinuity
- $y=\text{floor}(x)$; same
is the model smooth?

- $y = \text{abs}(x)$; gives a kink
- $y = (x > 0) ? 3x : 2x + 2$; gives a discontinuity
- $y = \text{floor}(x)$; same
- $Y = \text{REAL}(Z)$; what about $\text{IMAG}(Z)$
is the model smooth?

- $y=\text{abs}(x)$; gives a kink
- $y=(x>0)?3*x:2*x+2$; gives a discontinuity
- $y=\text{floor}(x)$; same
- $Y=\text{REAL}(Z)$; what about $\text{IMAG}(Z)$
- if $(a == 1.0)$
  - $y = b$; 
  else if $(a == 0.0)$ then
  - $y = 0$; 
else
  - $y = a*b$;
is the model smooth?

- $y = \text{abs}(x)$; gives a kink
- $y = (x > 0) ? 3x : 2x + 2$; gives a discontinuity
- $y = \text{floor}(x)$; same
- $Y = \text{REAL}(Z)$; what about $\text{IMAG}(Z)$
- if (a == 1.0)
  - $y = b$
- else if (a == 0.0) then
  - $y = 0$
- else
  - $y = a \times b$

intended: $\dot{y} = a \dot{b} + b \dot{a}$
is the model smooth?

- $y = \text{abs}(x)$; gives a kink
- $y = (x > 0) ? 3x : 2x + 2$; gives a discontinuity
- $y = \text{floor}(x)$; same
- $Y = \text{REAL}(Z)$; what about $\text{IMAG}(Z)$
- if $(a == 1.0)$
  
  $y = b$

  else if $(a == 0.0)$ then
  
  $y = 0$

  else
  
  $y = a*b$

  intended: $\dot{y} = a \dot{b} + b \dot{a}$

- $y = \sqrt{a^4 + b^4}$;
is the model smooth?

- $y = \text{abs}(x)$; gives a kink
- $y = (x>0) ? 3x : 2x + 2$; gives a discontinuity
- $y = \text{floor}(x)$; same
- $Y = \text{REAL}(Z)$; what about $\text{IMAG}(Z)$
- if ($a == 1.0$
  
  $y = b$;
  
  else if ($a == 0.0$) then
  
  $y = 0$;
  
  else
  
  $y = a*b$;
  
  intended: $\frac{\dot{y}}{\dot{a}} = a\dot{b} + b\dot{a}$
- $y = \text{sqrt}(a**4 + b**4)$;

AD does not perform algebraic simplification,

i.e. for $a,b \to 0$ it does $(\frac{d\sqrt{t}}{dt}) \to +\infty$. 

Intro to AD - Utke - May/2013
is the model smooth?

- \( y = \text{abs}(x) \); gives a kink
- \( y = (x > 0) ? 3 * x : 2 * x + 2 \); gives a discontinuity
- \( y = \text{floor}(x) \); same
- \( Y = \text{REAL}(Z) \); what about \( \text{IMAG}(Z) \)
- if \( a == 1.0 \)
  - \( y = b \);
  - else if \( a == 0.0 \) then
  - \( y = 0 \);
  - else
  - \( y = a * b \);
  - intended: \( \dot{y} = a \dot{b} + b \dot{a} \)
- \( y = \sqrt{\text{a}^4 + \text{b}^4} \);

Algorithmic differentiation computes derivatives of programs(!)

Know your application e.g. fix point iteration, self adjoint, step size computation, convergence
nonsmooth models

observed:

- \( \text{INF, NaN, e.g. for } \sqrt{0 \pm 0}; \text{ smoother in } [0, \varepsilon] \)?
- oscillating derivatives (may be glossed over by FD) or derivatives growing out of bounds
nonsmooth models II

- blame AD tool - verification problem
nonsmooth models II

- blame AD tool - verification problem
  - forward vs reverse (dot product check)
nonsmooth models II

- blame AD tool - verification problem
  - forward vs reverse (dot product check)
  - compare to FD
nonsmooth models II

- blame AD tool - verification problem
  - forward vs reverse (dot product check)
  - compare to FD
  - compare to other AD tool
nonsmooth models II

- blame AD tool - verification problem
  - forward vs reverse (dot product check)
  - compare to FD
  - compare to other AD tool

- blame code, model’s built-in numerical approximations, external optimization scheme or inherent in the physics?
nonsmooth models II

- blame AD tool - verification problem
  - forward vs reverse (dot product check)
  - compare to FD
  - compare to other AD tool

- blame code, model’s built-in numerical approximations, external optimization scheme or inherent in the physics?

- higher order models in mech. engineering, beam physics, AtomFT explicit g-stop facility for ODEs, DAEs
nonsmooth models II

- blame AD tool - verification problem
  - forward vs reverse (dot product check)
  - compare to FD
  - compare to other AD tool
- blame code, model’s built-in numerical approximations, external optimization scheme or inherent in the physics?
- higher order models in mech. engineering, beam physics, AtomFT explicit g-stop facility for ODEs, DAEs
- what to do about first order
nonsmooth models II

◊ blame AD tool - verification problem
  ♦ forward vs reverse (dot product check)
  ♦ compare to FD
  ♦ compare to other AD tool

◊ blame code, model’s built-in numerical approximations, external optimization scheme or inherent in the physics?

◊ higher order models in mech. engineering, beam physics, AtomFT explicit g-stop facility for ODEs, DAEs

◊ what to do about first order
  ♦ Adifor: optionally catches intrinsic problems via exception handling
nonsmooth models II

- blame AD tool - verification problem
  - forward vs reverse (dot product check)
  - compare to FD
  - compare to other AD tool

- blame code, model’s built-in numerical approximations, external optimization scheme or inherent in the physics?

- higher order models in mech. engineering, beam physics, AtomFT explicit g-stop facility for ODEs, DAEs

- what to do about first order
  - Adifor: optionally catches intrinsic problems via exception handling
  - Adol-C: tape verification and intrinsic handling
nonsmooth models II

- blame AD tool - verification problem
  - forward vs reverse (dot product check)
  - compare to FD
  - compare to other AD tool

- blame code, model’s built-in numerical approximations, external optimization scheme or inherent in the physics?

- higher order models in mech. engineering, beam physics, AtomFT explicit g-stop facility for ODEs, DAEs

- what to do about first order
  - Adifor: optionally catches intrinsic problems via exception handling
  - Adol-C: tape verification and intrinsic handling
  - OpenAD (comparative tracing)
piecewise differentiable function: $|x^2 - \sin(|y|)|$
is (locally) Lipschitz continuous; almost everywhere differentiable
(aside from on the 6 critical paths)

- Gâteaux: if $\exists \nabla f(x, \hat{x}) = \lim_{\tau \to 0} \frac{f(x+\tau \hat{x})-f(x)}{\tau}$ for all directions $\hat{x}$
- Bouligand: Lipschitz continuous and Gâteaux
- Fréchet: $df(., \hat{x})$ continuous for every fixed $\hat{x}$ ... not generally
- in practice: often benign behavior, directional derivative exists and is an element of the generalized gradient.
case distinction
case distinction

3 locally analytic
case distinction

3 locally analytic

2 locally analytic but crossed a (potential) kink (min, max, abs, ...) or discontinuity (ceil, ...) [for source transformation: also different control flow]
case distinction

3 locally analytic

2 locally analytic but crossed a (potential) kink (min, max, abs, ...) or discontinuity (ceil, ...) [for source transformation: also different control flow]

1 we are exactly at a (potential) kink, discontinuity
case distinction

3 locally analytic

2 locally analytic but crossed a (potential) kink (\(\min, \max, \abs, \ldots\)) or discontinuity (\(\text{ceil}, \ldots\)) [for source transformation: also different control flow]

1 we are exactly at a (potential) kink, discontinuity

0 tie on arithmetic comparison (e.g. a branch condition) \(\rightarrow\) potentially discontinuous (can only be determined for some special cases)
case distinction

3 locally analytic

2 locally analytic but crossed a (potential) kink \((\min, \max, \text{abs}, \ldots)\)
or discontinuity \((\text{ceil}, \ldots)\) [for source transformation: also
different control flow]

1 we are exactly at a (potential) kink, discontinuity

0 tie on arithmetic comparison (e.g. a branch condition) \(\rightarrow\) potentially

discontinuous (can only be determined for some special cases)

\([-1\) (operator overloading specific) arithmetic comparison yields a
different value than before (tape invalid \(\rightarrow\) sparsity pattern may be
changed,\ldots) \)]
many repeated Jacobian vector products → compress the Jacobian
\( F' \cdot S = B \in \mathbb{R}^{m \times q} \) using a seed matrix \( S \in \mathbb{R}^{n \times q} \)

What are \( S \) and \( q \)?

Row \( i \) in \( F' \) has \( \rho_i \) nonzeros in columns \( v(1), \ldots, v(\rho_i) \)

\( F'_i = (\alpha_1, \ldots, \alpha_{\rho_i}) = \alpha^T \) and the compressed row is

\( B_i = (\beta_1, \ldots, \beta_q) = \beta^T \)

We choose \( S \) so we can solve:

\[ \hat{S}_i \alpha = \beta \]

with \( \hat{S}_i^T = (s_{v(1)}, \ldots, s_{v(\rho_i)}) \)
direct:
  - Curtis/Powell/Reid: structurally orthogonal
  - Coleman/Moré: column incidence graph coloring)

$q$ is the color number in column incidence graph, each column in $S$ represents a color with a 1 for each entry whose corresponding column in $F'$ is of that color.

\[
S = \begin{bmatrix}
1 & 0 \\
0 & 1 \\
1 & 0 \\
0 & 1
\end{bmatrix}
\]

reconstruct $F'$ by relocating nonzero elements (direct)
sparsity (3)

indirect:

◇ Newsam/Ramsdell: \( q = \max_i \{ \# \text{nonzeros} \} \leq \chi \)

◇ \( S \) is a (generalized) Vandermonde matrix

\[
\begin{bmatrix}
\lambda_j^{i-1}
\end{bmatrix}, \quad j = 1 \ldots q, \quad \lambda_i \neq \lambda_i'
\]

◇ How many different \( \lambda_i \)?

same example

\[
S = \begin{bmatrix}
\lambda_0^0 & \lambda_1^1 \\
\lambda_2^0 & \lambda_2^1 \\
\lambda_3^0 & \lambda_3^1 \\
\lambda_4^0 & \lambda_4^1
\end{bmatrix}
\]

all combinations of columns (\( = \) rows of \( S \)): \( (1, 2), (2, 3), (1, 4) \)

improved condition via generalization approaches

related notions: partial separability, contraction points, scarcity
numerical libraries/frameworks (1)

- interfaces implement *fixed* mathematical meaning
- may be a "black box" (different language, proprietary)
numerical libraries/frameworks (1)

- interfaces implement *fixed* mathematical meaning
- may be a “black box” (different language, proprietary)
- hopefully has derivatives easily implementable with the library calls, e.g. blas,
- linear solves $x = A^{-1}b$
  - one can show $\dot{x} = A^{-1}(\dot{b} - \dot{A}x)$
  - $\bar{b} = A^{-T}\bar{x}; \bar{A}^+ = -\bar{b}\bar{x}^T$
- often requires single call encapsulation
numerical libraries/frameworks (1)

- interfaces implement *fixed* mathematical meaning
- may be a “black box” (different language, proprietary)
- hopefully has derivatives easily implementable with the library calls, e.g. blas,
- linear solves $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$
  - one can show $\dot{\mathbf{x}} = \mathbf{A}^{-1}(\dot{\mathbf{b}} - \dot{\mathbf{A}}\mathbf{x})$
  - $\bar{\mathbf{b}} = \mathbf{A}^{-T}\bar{\mathbf{x}}$; $\bar{\mathbf{A}}^{+} = -\bar{\mathbf{b}}\bar{\mathbf{x}}^T$
- often requires single call encapsulation
- brute force differentiation as last resort
numerical libraries/frameworks (1)

- interfaces implement *fixed* mathematical meaning
- may be a “black box” (different language, proprietary)
- hopefully has derivatives easily implementable with the library calls, e.g. blas,
- linear solves $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$
  - one can show $\dot{\mathbf{x}} = \mathbf{A}^{-1}(\dot{\mathbf{b}} - \dot{\mathbf{A}}\mathbf{x})$
  - $\tilde{\mathbf{b}} = \mathbf{A} - T\tilde{\mathbf{x}}$; $\tilde{\mathbf{A}} + = -\tilde{\mathbf{b}}\mathbf{x}^T$
- often requires single call encapsulation
- brute force differentiation as last resort
- *always consider* augment convergence criterion for iterative numerical methods (chapter 15 in Griewank/Walther)
numerical libraries/frameworks (1)

- interfaces implement *fixed* mathematical meaning
- may be a “black box” (different language, proprietary)
- hopefully has derivatives easily implementable with the library calls, e.g. blas,
- linear solves $x = A^{-1}b$
  - one can show $\dot{x} = A^{-1}(\dot{b} - \dot{Ax})$
  - $\bar{b} = A - T \bar{x}$; $\bar{A}+ = -\bar{b}x^T$
- often requires single call encapsulation
- brute force differentiation as last resort
- *always consider* augment convergence criterion for iterative numerical methods (chapter 15 in Griewank/Walther)
- efficiency considerations, see “delayed piggyback” e.g. for iterations $x_{k+1} = f(x_k)$
numerical libraries/frameworks (2)

- no generic “differentiated” libraries (attempt for MPI)
numerical libraries/frameworks (2)

- no generic “differentiated” libraries (attempt for MPI)
- efficient implementation tied to AD tool implementation
numerical libraries/frameworks (2)

- no generic “differentiated” libraries (attempt for MPI)
- efficient implementation tied to AD tool implementation
- high level uses of differentiation also to be considered for frameworks (examples neos, trilinos, petsc)
numerical libraries/frameworks (2)

- no generic “differentiated” libraries (attempt for MPI)
- efficient implementation tied to AD tool implementation
- high level uses of differentiation also to be considered for frameworks (examples neos, trilinos, petsc)
- advanced topics: Taylor coefficient recursions, mathematical mappings split over multiple library calls (reverse mode)
Summary

- basics of AD are deceptively simple
Summary

- basics of AD are deceptively simple
- AD tools offer semi-automatic differentiation of algorithms
Summary

- basics of AD are deceptively simple
- AD tools offer semi-automatic differentiation of algorithms
- specialized tools for higher order
Summary

- basics of AD are deceptively simple
- AD tools offer semi-automatic differentiation of algorithms
- specialized tools for higher order
- details in the code have a large impact on AD adjoint efficiency
Summary

- basics of AD are deceptively simple
- AD tools offer semi-automatic differentiation of algorithms
- specialized tools for higher order
- details in the code have a large impact on AD adjoint efficiency
- problems with certain language features are also problems for compiler optimization
Summary

- basics of AD are deceptively simple
- AD tools offer semi-automatic differentiation of algorithms
- specialized tools for higher order
- details in the code have a large impact on AD adjoint efficiency
- problems with certain language features are also problems for compiler optimization
- computational efficiency is improved by exploiting higher level insights