

Introduction to Algorithmic Differentiation

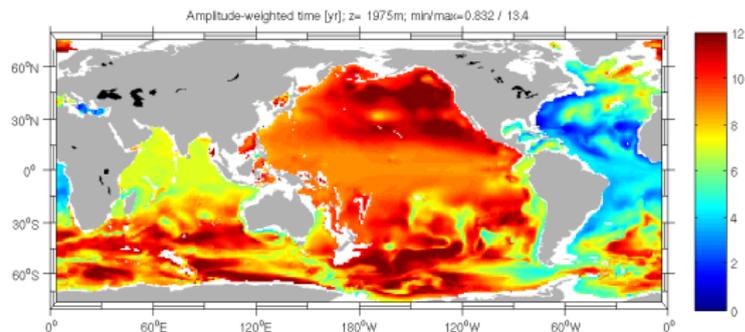
Narayanan/Utke

Argonne National Laboratory
Mathematics and Computer Science Division

12th USNCCM - July 2013 Raleigh NC

outline

- ◇ motivation / basics
- ◇ simple examples
- ◇ tool / algorithmic choices
- ◇ sparsity / partial separability
- ◇ differentiability / nonsmoothness
- ◇ checkpointing / reversal schemes
- ◇ practical approaches for big applications
- ◇ revolve*
- ◇ library interfaces*
- ◇ fast higher order derivatives*
- ◇ Q&A



why algorithmic differentiation?

given: some numerical model $\mathbf{y} = \mathbf{f}(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$
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 $\mathbf{J}\dot{\mathbf{x}}$ or $\bar{\mathbf{y}}^T \mathbf{J}$ 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 $\mathbf{y} = \mathbf{f}(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$
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 $\mathbf{J}\dot{\mathbf{x}}$ or $\bar{\mathbf{y}}^T \mathbf{J}$ 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 $\mathbf{y} = \mathbf{f}(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$
implemented as a (large / volatile) program

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

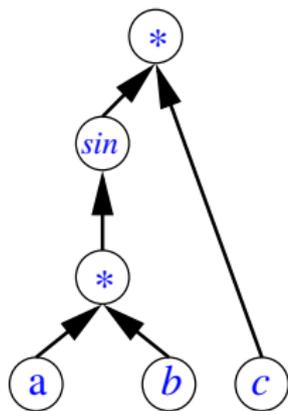
1. don't pretend we know nothing about the program
(and take finite differences of an oracle)
2. get machine precision derivatives as $\mathbf{J}\dot{\mathbf{x}}$ or $\bar{\mathbf{y}}^T \mathbf{J}$ or ...
(avoid approximation-versus-roundoff problem)
3. the reverse (aka adjoint) mode yields "cheap" gradients
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 at least **semi-automatically!**

how does AD compute derivatives?

$$f : y = \sin(a * b) * c : \mathbb{R}^3 \mapsto \mathbb{R}$$

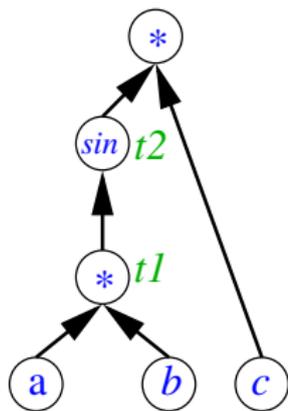
yields a graph representing the order of computation:



how does AD compute derivatives?

$$f : y = \sin(a * b) * c : \mathbb{R}^3 \mapsto \mathbb{R}$$

yields a graph representing the order of computation:



◇ *code list* → intermediate values $t1$ and $t2$

$$t1 = a * b$$

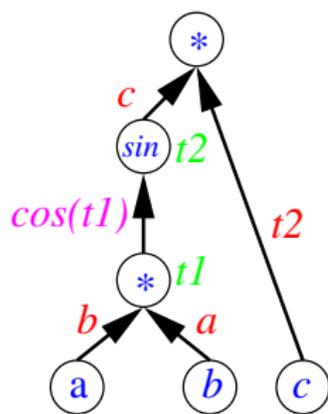
$$t2 = \sin(t1)$$

$$y = t2 * c$$

how does AD compute derivatives?

$$f : y = \sin(a * b) * c : \mathbb{R}^3 \mapsto \mathbb{R}$$

yields a graph representing the order of computation:



- ◇ *code list* \rightarrow intermediate values $t1$ and $t2$
- ◇ each intrinsic $v = \phi(w, u)$ has local partials $\frac{\partial \phi}{\partial w}$,
- ◇ e.g. $\sin(t1)$ yields $p1 = \cos(t1)$
- ◇ in our example all others are already stored in variables

$$t1 = a * b$$

$$p1 = \cos(t1)$$

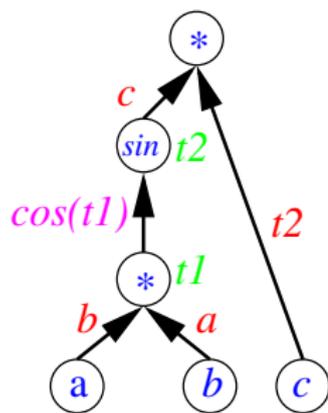
$$t2 = \sin(t1)$$

$$y = t2 * c$$

how does AD compute derivatives?

$$f : y = \sin(a * b) * c : \mathbb{R}^3 \mapsto \mathbb{R}$$

yields a graph representing the order of computation:



- ◇ *code list* → intermediate values $t1$ and $t2$
- ◇ each intrinsic $v = \phi(w, u)$ has local partials $\frac{\partial \phi}{\partial w}$,
- ◇ e.g. $\sin(t1)$ yields $p1 = \cos(t1)$
- ◇ in our example all others are already stored in variables

$$t1 = a * b$$

$$p1 = \cos(t1)$$

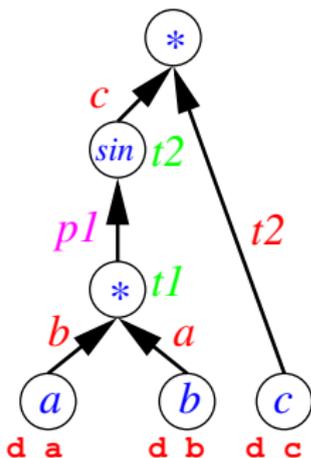
$$t2 = \sin(t1)$$

$$y = t2 * c$$

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

t1 = a*b

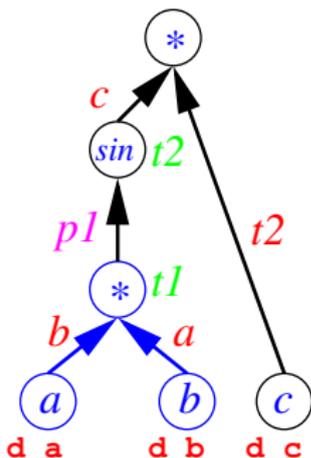
p1 = cos(t1)

t2 = sin(t1)

y = t2*c

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

$t1 = a * b$

$d_t1 = d_a * b + d_b * a$

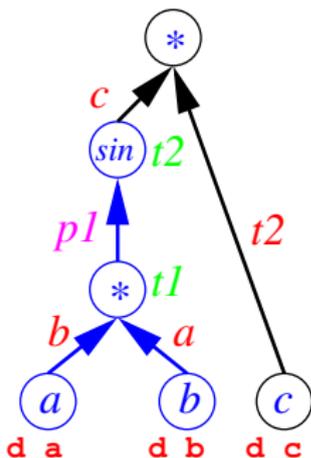
$p1 = \cos(t1)$

$t2 = \sin(t1)$

$y = t2 * c$

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

$t1 = a*b$

$d_t1 = d_a*b + d_b*a$

$p1 = \cos(t1)$

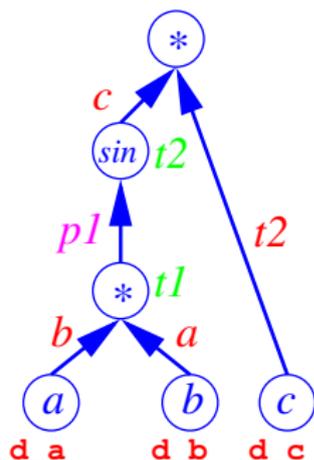
$t2 = \sin(t1)$

$d_t2 = d_t1*p1$

$y = t2*c$

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

$t1 = a * b$

$d_t1 = d_a * b + d_b * a$

$p1 = \cos(t1)$

$t2 = \sin(t1)$

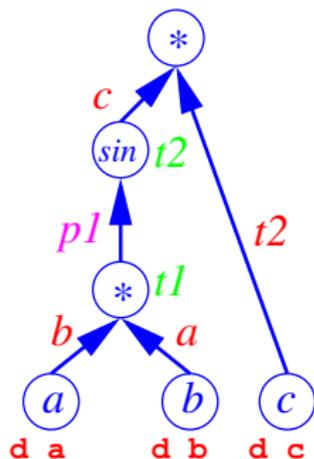
$d_t2 = d_t1 * p1$

$y = t2 * c$

$d_y = d_t2 * c + d_c * t2$

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

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

d_y = d_t2*c + d_c*t2

What is in d_y ?

d_y contains a projection

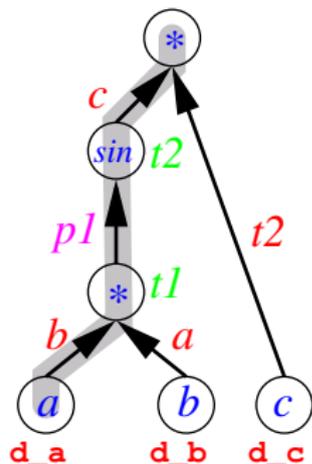
◇ $\dot{y} = J\dot{x}$ computed at x_0

d_y contains a projection

- ◇ $\dot{\mathbf{y}} = \mathbf{J}\dot{\mathbf{x}}$ computed at \mathbf{x}_0
- ◇ for example for $(\dot{a}, \dot{b}, \dot{c}) = (1, 0, 0)$

d_y contains a projection

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



- ◇ yields the first element of the gradient
- ◇ all gradient elements cost $\mathcal{O}(n)$ function evaluations

applications

for instance

- ◇ ocean/atmosphere state estimation & uncertainty quantification, oil reservoir modeling
- ◇ computational chemical engineering
- ◇ CFD
- ◇ beam physics
- ◇ mechanical engineering

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
- ◇ beam physics
- ◇ mechanical engineering

use

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

How do we get the cheap gradients? ... later

sidebar: simple overloaded operators for a*b

in Fortran:

in C++:

```
struct Afloat{float v; float d;};

Afloat operator *(Afloat a, Afloat b) {
  Afloat r; int i;
  r.v=a.v*b.v; // value
  r.d=a.d*b.v+a.v*b.d; // derivative
  return r;
};

// other argument combinations
```

```
module ATypes
public :: Areal
type Areal
sequence
real :: v,d
end type
end module ATypes

module Amult
use ATypes
interface operator(*)
module procedure multArealAreal
! other argument combinations
end interface
contains
function multArealAreal(a,b) result(r)
type(Areal),intent(in)::a,b
type(Areal)::r
r%v=a%v*b%v ! value
r%d=a%d*b%v+a%v*b%v ! derivative
end function multArealAreal
end module Amult
```

sidebar: simple overloaded operators for a*b

in Fortran:

in C++:

```
struct Afloat{float v; float d;};

Afloat operator *(Afloat a, Afloat b) {
  Afloat r; int i;
  r.v=a.v*b.v; // value
  r.d=a.d*b.v+a.v*b.d; // derivative
  return r;
};

// other argument combinations
```

```
module ATypes
public :: Areal
type Areal
sequence
real :: v,d
end type
end module ATypes

module Amult
use ATypes
interface operator(*)
module procedure multArealAreal
! other argument combinations
end interface
contains
function multArealAreal(a,b) result(r)
type(Areal),intent(in)::a,b
type(Areal)::r
r%v=a%v*b%v ! value
r%d=a%d*b%v+a%v*b%v ! derivative
end function multArealAreal
end module Amult
```

Operator Overloading \Rightarrow

A simple, relatively unintrusive way to augment semantics via a type change!

Rapsodia - overview

- ◇ similar code in the overloaded operators - use a code generator - Rapsodia
- ◇ main motivation is higher-order (later)

Rapsodia - overview

- ◇ similar code in the overloaded operators - use a code generator - Rapsodia
- ◇ main motivation is higher-order (later)
- ◇ generates C++/Fortran overloading libraries
- ◇ some support code
- ◇ open source see www.mcs.anl.gov/Rapsodia/

Rapsodia - overview

- ◇ similar code in the overloaded operators - use a code generator - Rapsodia
- ◇ main motivation is higher-order (later)
- ◇ generates C++/Fortran overloading libraries
- ◇ some support code
- ◇ open source see www.mcs.anl.gov/Rapsodia/
- ◇ work flow:
 - ◆ generate library

Rapsodia - overview

- ◇ similar code in the overloaded operators - use a code generator - Rapsodia
- ◇ main motivation is higher-order (later)
- ◇ generates C++/Fortran overloading libraries
- ◇ some support code
- ◇ open source see www.mcs.anl.gov/Rapsodia/
- ◇ work flow:
 - ◆ generate library
 - ◆ type change the original source code to an active type

Rapsodia - overview

- ◇ similar code in the overloaded operators - use a code generator - Rapsodia
- ◇ main motivation is higher-order (later)
- ◇ generates C++/Fortran overloading libraries
- ◇ some support code
- ◇ open source see www.mcs.anl.gov/Rapsodia/
- ◇ work flow:
 - ◆ generate library
 - ◆ type change the original source code to an active type
 - ◆ write driver logic to initialize/retrieve derivatives

Rapsodia - overview

- ◇ similar code in the overloaded operators - use a code generator - Rapsodia
- ◇ main motivation is higher-order (later)
- ◇ generates C++/Fortran overloading libraries
- ◇ some support code
- ◇ open source see www.mcs.anl.gov/Rapsodia/
- ◇ work flow:
 - ◆ generate library
 - ◆ type change the original source code to an active type
 - ◆ write driver logic to initialize/retrieve derivatives
 - ◆ compile/link

Rapsodia - overview

- ◇ similar code in the overloaded operators - use a code generator - Rapsodia
- ◇ main motivation is higher-order (later)
- ◇ generates C++/Fortran overloading libraries
- ◇ some support code
- ◇ open source see www.mcs.anl.gov/Rapsodia/
- ◇ work flow:
 - ◆ generate library
 - ◆ type change the original source code to an active type
 - ◆ write driver logic to initialize/retrieve derivatives
 - ◆ compile/link
- ◇ look at an example...

Rapsodia - simple example

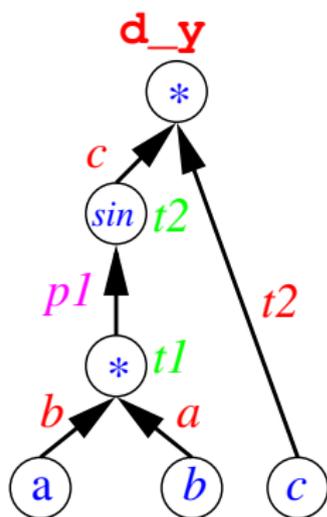
- ◇ get into the VM
- ◇ `cd ~/Rapsodia`
- ◇ `export RAPSODIAROOT=$PWD`
- ◇ `cd ../RapsodiaExamples/CppOneMinute/`
- ◇ look at
 - ◆ original driver (`driver0.cpp`)
 - ◆ augmented driver (`driver.cpp`)
 - ◆ `Makefile`
- ◇ `make clean`
- ◇ `make`

ADIC - overview and simple example

pass to Krishna ...

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
 $\bar{w} += \frac{\partial \phi}{\partial w} \bar{v}$; $\bar{u} += \frac{\partial \phi}{\partial u} \bar{v}$; $\bar{v} = 0$



backward propagation code appended:

```
t1 = a*b
```

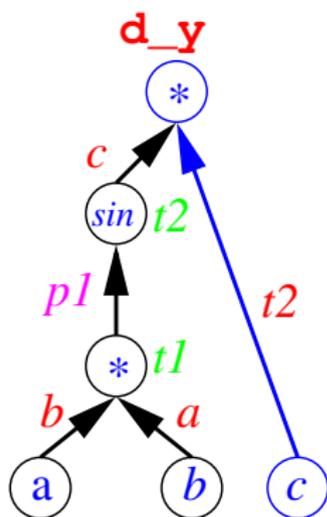
```
p1 = cos(t1)
```

```
t2 = sin(t1)
```

```
y = t2*c
```

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
 $\bar{w} += \frac{\partial \phi}{\partial w} \bar{v}$; $\bar{u} += \frac{\partial \phi}{\partial u} \bar{v}$; $\bar{v} = 0$



backward propagation code appended:

```
t1 = a*b
```

```
p1 = cos(t1)
```

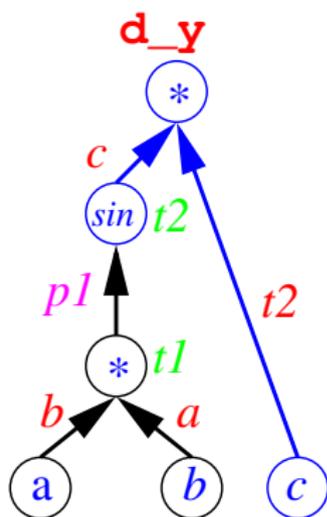
```
t2 = sin(t1)
```

```
y = t2*c
```

```
d_c = t2*d_y
```

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
 $\bar{w} += \frac{\partial \phi}{\partial w} \bar{v}$; $\bar{u} += \frac{\partial \phi}{\partial u} \bar{v}$; $\bar{v} = 0$

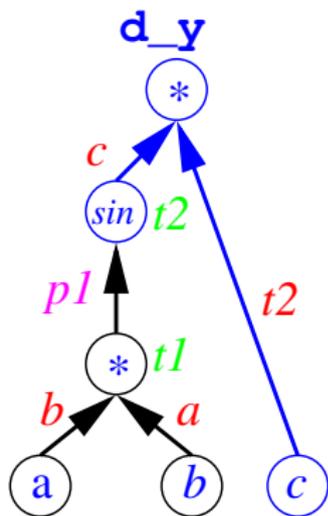


backward propagation code appended:

```
t1 = a*b  
p1 = cos(t1)  
t2 = sin(t1)  
y = t2*c  
d_c = t2*d_y  
d_t2 = c*d_y
```

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
 $\bar{w} += \frac{\partial \phi}{\partial w} \bar{v}$; $\bar{u} += \frac{\partial \phi}{\partial u} \bar{v}$; $\bar{v} = 0$

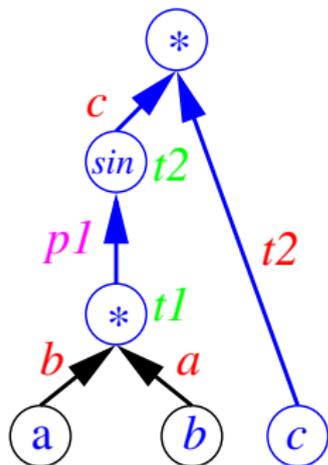


backward propagation code appended:

```
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
```

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
 $\bar{w} += \frac{\partial \phi}{\partial w} \bar{v}$; $\bar{u} += \frac{\partial \phi}{\partial u} \bar{v}$; $\bar{v} = 0$



backward propagation code appended:

```
t1 = a*b
```

```
p1 = cos(t1)
```

```
t2 = sin(t1)
```

```
y = t2*c
```

```
d_c = t2*d_y
```

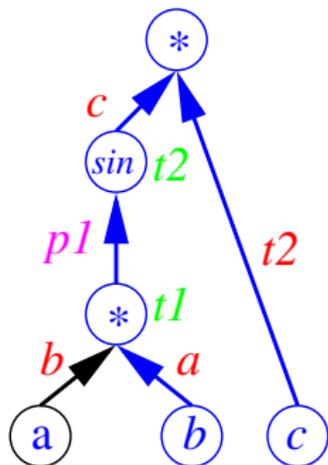
```
d_t2 = c*d_y
```

```
d_y = 0
```

```
d_t1 = p1*d_t2
```

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
 $\bar{w} += \frac{\partial \phi}{\partial w} \bar{v}$; $\bar{u} += \frac{\partial \phi}{\partial u} \bar{v}$; $\bar{v} = 0$

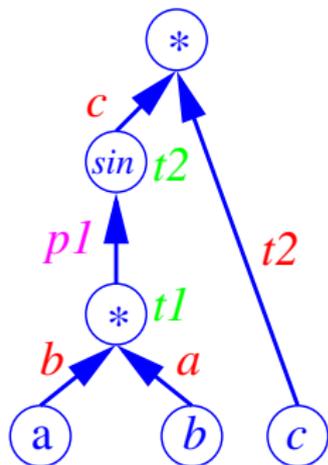


backward propagation code appended:

```
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
 $\bar{w} += \frac{\partial \phi}{\partial w} \bar{v}$; $\bar{u} += \frac{\partial \phi}{\partial u} \bar{v}$; $\bar{v} = 0$

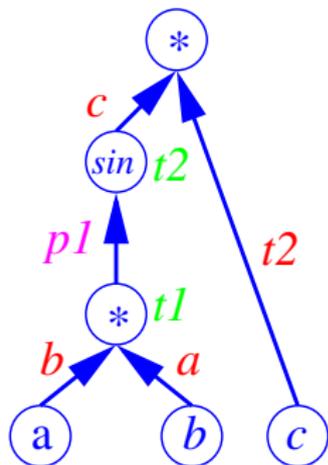


backward propagation code appended:

```
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
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
 $\bar{w} += \frac{\partial \phi}{\partial w} \bar{v}$; $\bar{u} += \frac{\partial \phi}{\partial u} \bar{v}$; $\bar{v} = 0$



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
```

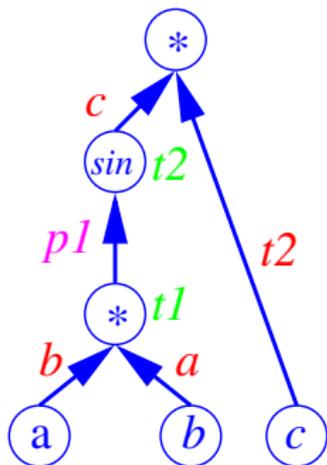
```
d_b = a*d_t1
```

```
d_a = b*d_t1
```

What is in (d_a, d_b, d_c) ?

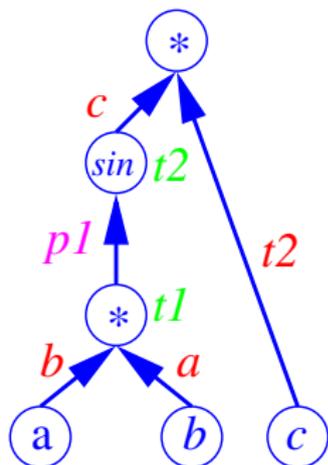
(d_a, d_b, d_c) contains a projection

◇ $\bar{x} = \bar{y}^T J$ computed at x_0



(d_a, d_b, d_c) contains a projection

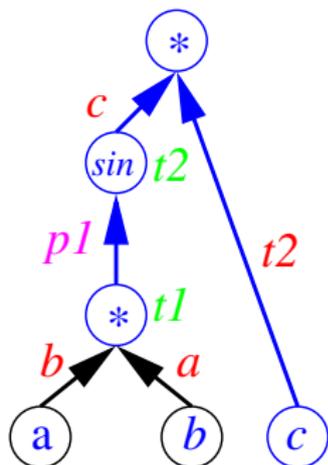
- ◇ $\bar{x} = \bar{y}^T \mathbf{J}$ computed at x_0
- ◇ for example for $\bar{y} = 1$ we have $[\bar{a}, \bar{b}, \bar{c}] = \nabla f$



- ◇ all gradient elements cost $\mathcal{O}(1)$ function evaluations

(d_a, d_b, d_c) contains a projection

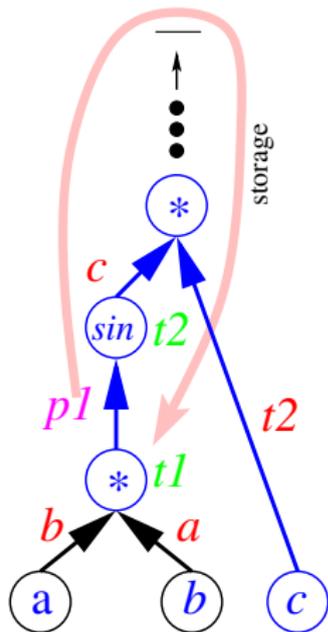
- ◇ $\bar{x} = \bar{y}^T \mathbf{J}$ computed at x_0
- ◇ for example for $\bar{y} = 1$ we have $[\bar{a}, \bar{b}, \bar{c}] = \nabla f$



- ◇ all gradient elements cost $\mathcal{O}(1)$ function evaluations
- ◇ but consider when $p1$ is computed and when it is used

(d_a, d_b, d_c) contains a projection

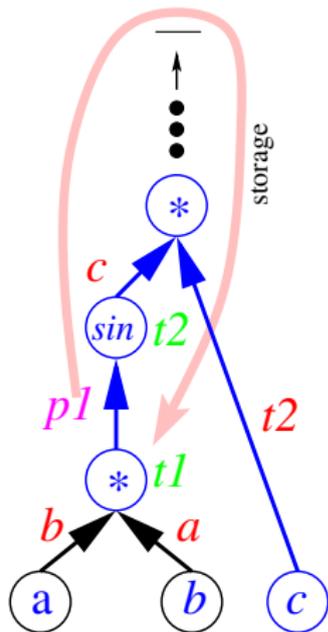
- ◇ $\bar{x} = \bar{y}^T J$ computed at x_0
- ◇ for example for $\bar{y} = 1$ we have $[\bar{a}, \bar{b}, \bar{c}] = \nabla f$



- ◇ all gradient elements cost $\mathcal{O}(1)$ function evaluations
- ◇ but consider when $p1$ is computed and when it is used
- ◇ **storage requirements** grow with the length of the computation [▶ jump to CP](#)
- ◇ typically mitigated by recomputation from checkpoints

(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 $\mathcal{O}(1)$ function evaluations
- ◇ but consider when $p1$ is computed and when it is used
- ◇ **storage requirements** grow with the length of the computation [▶ jump to CP](#)
- ◇ typically mitigated by recomputation from checkpoints

Reverse mode with Adol-C.

Adol-C - overview

- ◇ operator overloading library for C++
- ◇ open source
- ◇ see www.coin-or.org/projects/ADOL-C.xml

Adol-C - overview

- ◇ operator overloading library for C++
- ◇ open source
- ◇ see www.coin-or.org/projects/ADOL-C.xml
- ◇ overloaded operators create an execution trace, called the tape
- ◇ tape interpreters run forward/reverse on the tape

Adol-C - overview

- ◇ operator overloading library for C++
- ◇ open source
- ◇ see www.coin-or.org/projects/ADOL-C.xml
- ◇ overloaded operators create an execution trace, called the tape
- ◇ tape interpreters run forward/reverse on the tape
- ◇ work flow:
 - ◆ configure/compile/install Adol-C headers/library

Adol-C - overview

- ◇ operator overloading library for C++
- ◇ open source
- ◇ see www.coin-or.org/projects/ADOL-C.xml
- ◇ overloaded operators create an execution trace, called the tape
- ◇ tape interpreters run forward/reverse on the tape
- ◇ work flow:
 - ◆ configure/compile/install Adol-C headers/library
 - ◆ type change the original source code to an active type

Adol-C - overview

- ◇ operator overloading library for C++
- ◇ open source
- ◇ see www.coin-or.org/projects/ADOL-C.xml
- ◇ overloaded operators create an execution trace, called the tape
- ◇ tape interpreters run forward/reverse on the tape
- ◇ work flow:
 - ◆ configure/compile/install Adol-C headers/library
 - ◆ type change the original source code to an active type
 - ◆ write driver logic to initialize/retrieve derivatives

Adol-C - overview

- ◇ operator overloading library for C++
- ◇ open source
- ◇ see www.coin-or.org/projects/ADOL-C.xml
- ◇ overloaded operators create an execution trace, called the tape
- ◇ tape interpreters run forward/reverse on the tape
- ◇ work flow:
 - ◆ configure/compile/install Adol-C headers/library
 - ◆ type change the original source code to an active type
 - ◆ write driver logic to initialize/retrieve derivatives
 - ◆ compile/link

Adol-C - overview

- ◇ operator overloading library for C++
- ◇ open source
- ◇ see www.coin-or.org/projects/ADOL-C.xml
- ◇ overloaded operators create an execution trace, called the tape
- ◇ tape interpreters run forward/reverse on the tape
- ◇ work flow:
 - ◆ configure/compile/install Adol-C headers/library
 - ◆ type change the original source code to an active type
 - ◆ write driver logic to initialize/retrieve derivatives
 - ◆ compile/link
- ◇ look at an example ...

Adol-C - example I

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

```
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 - example I

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

```
#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 - example I

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

```
#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 :

```
gradient(tag,
         n,
         x[n],
         g[n])
```

Adol-C - example II

- ◇ get into the VM
- ◇ `cd ~/adol-c/ADOL-C/examples/`
- ◇ look at `speelpenning.cpp`
- ◇ run it by invoking
`./speelpenning`
- ◇ look at the implementation of gradient in
`~/adol-c/ADOL-C/src/drivers/drivers.c`

sidebar: Adol-C drivers

running the example produces a tape;
driver logic interprets the tape;
xp can be some point in \mathbb{R}^n ;

```
double* g = new double[n];

gradient(1,n,xp,g); // gradient

double** H = (double**)malloc(n*
    sizeof(double*));
for(i=0; i<n; i++)
    H[i] = (double*)malloc((i+1)*
        sizeof(double));

hessian(1,n,xp,H); // Hessian
```

drivers use tag as tape identifier;
gradient(tag,n,xp,g)

and similar for:

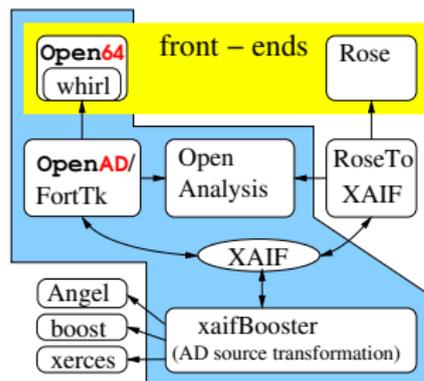
hessian(tag,n,xp,H)

need only H's lower triangle

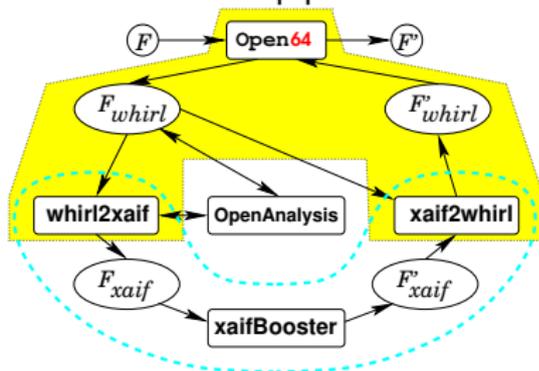
- ◇ various drivers use combinations of forward and reverse sweeps
- ◇ “tapeless” forward with slightly different usage patterns

OpenAD - overview

- ◇ www.mcs.anl.gov/OpenAD
- ◇ forward and **reverse**
- ◇ source transformation
- ◇ modular design
- ◇ large problems
- ◇ language independent transformation
- ◇ researching combinatorial problems
- ◇ current Fortran front-end Open64 (Open64/SL branch from Rice U)
- ◇ migration to Rose (already used for C/C++ with EDG)
- ◇ uses association by address as opposed to association by name



Fortran pipeline:



OpenAD - example II: simple scripted pipeline

- ◇ look at `Makefile`
- ◇ `openad` is a Python pipeline wrapper for `simple(!)` settings

OpenAD - example II: simple scripted pipeline

- ◇ look at `Makefile`
- ◇ `openad` is a Python pipeline wrapper for `simple(!)` settings
- ◇ invoke `openad -h` to see a usage message

OpenAD - example II: simple scripted pipeline

- ◇ look at `Makefile`
- ◇ `openad` is a Python pipeline wrapper for `simple(!)` settings
- ◇ invoke `openad -h` to see a usage message
- ◇ invoke `make`

OpenAD - example II: simple scripted pipeline

- ◇ look at Makefile
- ◇ openad is a Python pipeline wrapper for simple(!) settings
- ◇ invoke openad -h to see a usage message
- ◇ invoke make
- ◇ run ./driver

OpenAD - example II: simple scripted pipeline

- ◇ look at Makefile
- ◇ openad is a Python pipeline wrapper for simple(!) settings
- ◇ invoke openad -h to see a usage message
- ◇ invoke make
- ◇ run ./driver
- ◇ look at head.prepped.pre.xb.x2w.w2f.post.f90 ...

OpenAD - example II: simple scripted pipeline

- ◇ look at Makefile
- ◇ openad is a Python pipeline wrapper for simple(!) settings
- ◇ invoke openad -h to see a usage message
- ◇ invoke make
- ◇ run ./driver
- ◇ look at head.prepped.pre.xb.x2w.w2f.post.f90 ...
or rather not!

OpenAD - example II: simple scripted pipeline

- ◇ look at Makefile
- ◇ openad is a Python pipeline wrapper for simple(!) settings
- ◇ invoke openad -h to see a usage message
- ◇ invoke make
- ◇ run ./driver
- ◇ look at head.prepped.pre.xb.x2w.w2f.post.f90 ...
or rather not!
- ◇ individual make steps - invoke make driverE
- ◇ see the rules in MakeExplRules.inc

Source Transformation vs. Operator Overloading

- ◇ complicated implementation of tools
- ◇ especially for reverse mode
- ◇ full front end, back end, analysis
- ◇ efficiency gains from
 - ◆ compile time optimizations
 - ◆ activity analysis
 - ◆ explicit control flow reversal for reverse mode
- ◇ source transformation based type change & overloaded operators appropriate for higher-order derivatives.
- ◇ benefits from external information
- ◇ efficiency depends on analysis accuracy
- ◇ simple tool implementation
- ◇ reverse mode (generating and reinterpreting an execution trace → inefficient)
- ◇ implemented as some library
- ◇ impact on efficiency:
 - ◆ library implementation (narrow scope)
 - ◆ compiler inlining capabilities (for low order)
 - ◆ use external information (sparsity etc.)
 - ◆ can do only runtime optimizations
- ◇ manual type change for operator overloading
 - ◆ complicated for formatted I/O, allocation
 - ◆ need matching signatures in Fortran
 - ◆ helped by use of templates

Source Transformation vs. Operator Overloading

- ◇ complicated implementation of tools
- ◇ especially for reverse mode
- ◇ full front end, back end, analysis
- ◇ efficiency gains from
 - ◆ compile time optimizations
 - ◆ activity analysis
 - ◆ explicit control flow reversal for reverse mode
- ◇ source transformation based type change & overloaded operators appropriate for higher-order derivatives.
- ◇ benefits from external information
- ◇ efficiency depends on analysis accuracy
- ◇ simple tool implementation
- ◇ reverse mode (generating and reinterpreting an execution trace → inefficient)
- ◇ implemented as some library
- ◇ impact on efficiency:
 - ◆ library implementation (narrow scope)
 - ◆ compiler inlining capabilities (for low order)
 - ◆ use external information (sparsity etc.)
 - ◆ can do only runtime optimizations
- ◇ manual type change for operator overloading
 - ◆ complicated for formatted I/O, allocation
 - ◆ need matching signatures in Fortran
 - ◆ helped by use of templates

For higher-order derivatives combining source transformation based type change with overloaded operators is appropriate.

what to pick...

i.e. matching application requirements with AD tools and techniques

the major advantages of AD are ... no need to repeat again

- ◇ knowing AD tool “internal” algorithms is of interest to the user
(compare to compiler vector optimization)
- ◇ except for simple models and low computational complexity
→ can get away with “something”
- ◇ complicated models → worry about tool applicability
- ◇ high computational complexity → worry about efficiency of derivative computations
- ◇ tool availability (e.g. source transformation for C++ ?)

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. $f(x) = \sum(f_i(x_i)), x_i \in \mathcal{D}_i \subseteq \mathcal{D} \ni x$)
 - ◆ 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)

sparsity, partial separability,...

pass to Krishna ...

is the model f smooth?

examples:

- ◇ $y = \text{abs}(x)$; gives a kink

is the model f smooth?

examples:

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

is the model f smooth?

examples:

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

is the model f smooth?

examples:

- ◇ $y = \text{abs}(x)$; 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 $\text{IMAG}(Z)$

is the model f smooth?

examples:

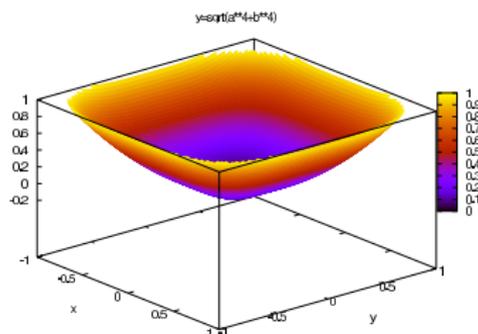
- ◇ $y = \text{abs}(x)$; gives a kink
- ◇ $y = (x > 0) ? 3 * x : 2 * x + 2$; gives a discontinuity
- ◇ $y = \text{floor}(x)$; same
- ◇ $Y = \text{REAL}(Z)$; what about $\text{IMAG}(Z)$
- ◇ **if** ($a == 1.0$)
 $y = b$;
else if ($a == 0.0$) then
 $y = 0$;
else
 $y = a * b$;

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

is the model f smooth?

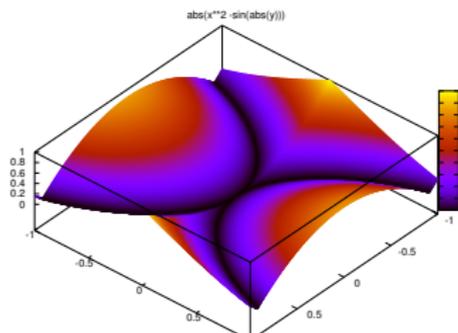
examples:

- ◇ $y = \text{abs}(x)$; 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: $\dot{y} = a * \dot{b} + b * \dot{a}$



$y = \sqrt{a^4 + b^4}$;
AD does not perform algebraic simplification,
i.e. for $a, b \rightarrow 0$ it does
 $\left(\frac{d\sqrt{t}}{dt}\right) \stackrel{t \rightarrow +0}{=} +\infty$.

sidebar: differentiability

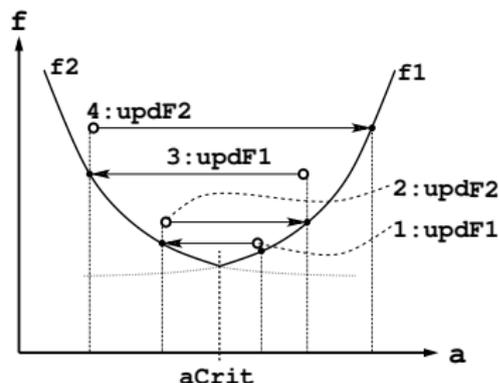
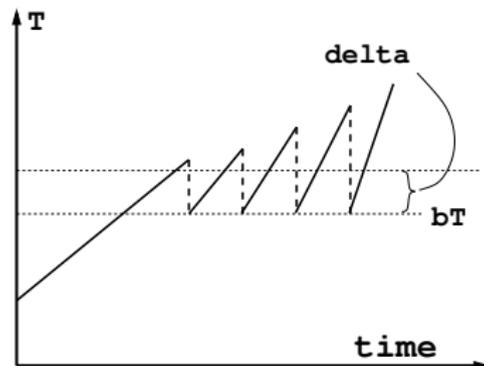


piecewise differentiable function:
 $|x^2 - \sin(|y|)|$
is (locally) Lipschitz continuous;
almost everywhere differentiable
(except on the 6 critical paths)

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

non-smooth models

- ◇ typically caused by:
 - ◆ the examples mentioned before
 - ◆ intrinsics: `max`, `ceil`, `sqrt`, `tan`,... (domain boundaries!)
 - ◆ branches if `(x < 2.5) y = f1(x); else y = f2(x);`
 - ◆ approximation methods (e.g. partially converged solves)
- ◇ may be observed as: oscillating derivatives (may be glossed over by FD); derivatives growing out of bounds; INF/NaN proliferation



non-smooth models AD vs FD

have: complicated AD source transformation

want: consistency check with FD

get: no match

non-smooth models AD vs FD

have: complicated AD source transformation

want: consistency check with FD

get: no match

- ◇ blame the AD tool - or -

non-smooth models AD vs FD

have: complicated AD source transformation

want: consistency check with FD

get: no match

- ◇ blame the AD tool - or -
 - ◆ compare forward to reverse
 - ◆ compare to other AD tool

non-smooth models AD vs FD

have: complicated AD source transformation

want: consistency check with FD

get: no match

- ◇ blame the AD tool - or -
 - ◆ compare forward to reverse
 - ◆ compare to other AD tool
- ◇ blame code: model's built-in numerical approximations, external optimization scheme, inherent in the physics, inconsistent initial state, initial state at domain boundary?

non-smooth models AD vs FD

have: complicated AD source transformation

want: consistency check with FD

get: no match

- ◇ blame the AD tool - or -
 - ◆ compare forward to reverse
 - ◆ compare to other AD tool
- ◇ blame code: model's built-in numerical approximations, external optimization scheme, inherent in the physics, inconsistent initial state, initial state at domain boundary?
→ fixes needed by model/application expert

non-smooth models AD vs FD

have: complicated AD source transformation

want: consistency check with FD

get: no match

- ◇ blame the AD tool - or -
 - ◆ compare forward to reverse
 - ◆ compare to other AD tool
- ◇ blame code: model's built-in numerical approximations, external optimization scheme, inherent in the physics, inconsistent initial state, initial state at domain boundary?
→ fixes needed by model/application expert

higher-order handling of nonsmoothness: interval inclusions - beam physics(COSY), distance approximation - explicit g-stop facility for ODEs, DAEs (ATOM-FT)

non-smooth models AD vs FD

have: complicated AD source transformation

want: consistency check with FD

get: no match

- ◇ blame the AD tool - or -
 - ◆ compare forward to reverse
 - ◆ compare to other AD tool
- ◇ blame code: model's built-in numerical approximations, external optimization scheme, inherent in the physics, inconsistent initial state, initial state at domain boundary?
→ fixes needed by model/application expert

higher-order handling of nonsmoothness: interval inclusions - beam physics(COSY), distance approximation - explicit g-stop facility for ODEs, DAEs (ATOM-FT)

what to do about first order:

non-smooth models AD vs FD

have: complicated AD source transformation

want: consistency check with FD

get: no match

- ◇ blame the AD tool - or -
 - ◆ compare forward to reverse
 - ◆ compare to other AD tool
- ◇ blame code: model's built-in numerical approximations, external optimization scheme, inherent in the physics, inconsistent initial state, initial state at domain boundary?
→ fixes needed by model/application expert

higher-order handling of nonsmoothness: interval inclusions - beam physics(COSY), distance approximation - explicit g-stop facility for ODEs, DAEs (ATOM-FT)

what to do about first order:

- ◇ Adol-C: tape verification and intrinsic handling
- ◇ OpenAD: (comparative tracing)

Adol-C directional derivatives & exceptions

tape at 1.0 and rerun at

- ◇ 0.5, $x_{\text{dot}}=1.0 \rightarrow y_{\text{dot}}=3$
- ◇ 0.0, $x_{\text{dot}}=1.0 \rightarrow y_{\text{dot}}=3$
- ◇ 0.0, $x_{\text{dot}}=-1.0 \rightarrow y_{\text{dot}}=-2$
- ◇ -0.5, $x_{\text{dot}}=1.0 \rightarrow y_{\text{dot}}=2$

```
adouble foo(adouble x) {  
    adouble y;  
    y=fmax(2*x,3*x);  
    return y;  
}
```

tape at 1.0 and rerun at

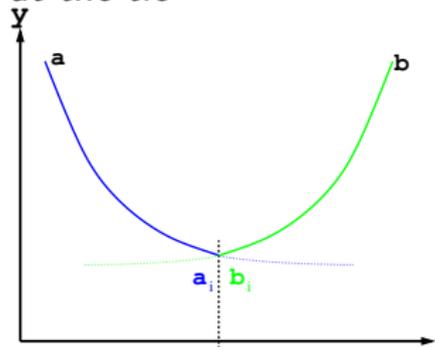
- ◇ 0.5, $x_{\text{dot}}=1.0 \rightarrow y_{\text{dot}}=.707107$
- ◇ 0.0, $x_{\text{dot}}=1.0 \rightarrow y_{\text{dot}}=INF$
- ◇ 0.0, $x_{\text{dot}}=-1.0 \rightarrow y_{\text{dot}}=NaN$

```
adouble foo(adouble x) {  
    adouble y;  
    y=sqrt(x);  
    return y;  
}
```

and on a higher level...

Should AD make educated guesses?

consider $y = \max(a(x), b(x))$
at the tie



pick direction from Taylor coefficients of first non-tied $\max(a_i, b_i)$?

consistency for unresolved ties:

take \dot{a} or \dot{b}

and compare that to an adjoint split:

$$\bar{a}_+ = \frac{\bar{y}}{2} \text{ and } \bar{b}_+ = \frac{\bar{y}}{2}$$

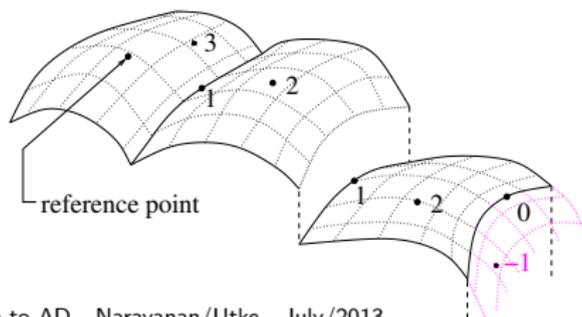
$$\text{consider } y = \sqrt{x} \text{ and } \dot{y}|_{x=+0} = \begin{cases} 0 & \text{if } \dot{x} = 0 \text{ ???} \\ +\text{INF} & \text{if } \dot{x} > 0 \\ \text{NaN} & \text{if } \dot{x} < 0 \end{cases}$$

option: (manually) inject a C^1 regularization $r(t)$ for $t \in [0, \epsilon]$
such that $\dot{r}(0) = 0$ and $\dot{r}(\epsilon) = \frac{1}{2\sqrt{\epsilon}}$

consider `maxloc`: tie-breaking argument `maxval` may differ from argument identified by `maxloc`

case distinction

- 3 locally analytic
- 2 locally analytic but crossed a (potential) kink ($\min, \max, \text{abs}, \dots$) or discontinuity (ceil, \dots) [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 be determined only for some special cases)
- [-1 (operator overloading specific) arithmetic comparison yields a different value than before (tape invalid \rightarrow sparsity pattern may be changed,...)]



Adol-c: classifying non-smooth events

```
adouble foo(adouble x) {  
    adouble y;  
    if (x<=2.5)  
        y=2*fmax(x,2.0);  
    else  
        y=3*floor(x);  
    return y;  
}
```

◇ tape at 2.2 and rerun at

- ◆ 2.3 → 3
- ◆ 2.0 → 1
- ◆ 2.5 → 0
- ◆ 2.6 → -1

```
#include "adolc.h"  
adouble foo(adouble x);  
  
int main() {  
    adouble x,y;  
    double xp,yp;  
    std::cout << "_tape_at:_" ;  
    std::cin >> xp;  
    trace_on(1);  
    x <<= xp;  
    y=foo(x);  
    y >>= yp;  
    trace_off();  
    while (true) {  
        std::cout << "rerun_at:_" ;  
        std::cin >> xp;  
        int rc=function(1,1,1,&xp,&yp);  
        std::cout<<"return_code:_" <<rc<<std::endl;  
    }  
}
```


Adol-c: classifying non-smooth events

```
adouble foo(adouble x) {
  adouble y;
  if (x<=2.5)
    y=2*fmax(x,2.0);
  else
    y=3*floor(x);
  return y;
}
```

◇ tape at 2.2 and rerun at

- ◆ 2.3 → 3
- ◆ 2.0 → 1
- ◆ 2.5 → 0
- ◆ 2.6 → -1

◇ tape at 3.5 and rerun at

- ◆ 3.6 → 3
- ◆ 4.5 → 2
- ◆ 2.5 → -1

```
#include "adolc.h"
adouble foo(adouble x);

int main() {
  adouble x,y;
  double xp,yp;
  std::cout << "_tape_at:_" ;
  std::cin >> xp;
  trace_on(1);
  x <<= xp;
  y=foo(x);
  y >>= yp;
  trace_off();
  while (true) {
    std::cout << "rerun_at:_" ;
    std::cin >> xp;
    int rc=function(1,1,1,&xp,&yp);
    std::cout<<"return_code:_" <<rc<<std::endl;
  }
}
```

validates tape but is **unspecific** ☹

more specific: OpenAD tracing (setup)

```
1  subroutine foo(t)
2    real :: t
3    call bar(t)
4  end subroutine
5  subroutine bar(t)
6    real :: t
7    t=tan(t)
8  end subroutine
9  subroutine head(x,y)
10   real :: x
11   real :: y
12   !$openad INDEPENDENT(x)
13   call foo(x)
14   call bar(x)
15   y=x
16   !$openad DEPENDENT(y)
17  end subroutine
```

```
1  program driver
2    use OAD_active
3    use OAD_rev
4    use OAD_trace
5    implicit none
6    external head
7    type(active) :: x, y
8
9    x%v=.5D0
10   ! first trace
11   call oad_trace_init()
12   call oad_trace_open()
13   call head(x,y)
14   call oad_trace_close()
```

OpenAD tracing (output I)

(on the preprocessed source)

```
3  subroutine foo(t)
4    use OAD_intrinsics
5    real :: t
6    call bar(t)
7  end subroutine
8  subroutine bar(t)
9    use OAD_intrinsics
10   real :: t
11   t=tan(t)
12 end subroutine
13 subroutine head(x,y)
14   use OAD_intrinsics
15   real :: x
16   real :: y
17   !$openad INDEPENDENT(x)
18   call foo(x)
19   call bar(x)
20   y=x
21   !$openad DEPENDENT(y)
22 end subroutine
```

```
<Trace number="1" >
  <Call name="foo" line="18" >
    <Call name="bar" line="6" >
      <Call name="tan_scal" line="11" ></Call>
      <Tan sd="0" />
    </Call>
  </Call>
  <Call name="bar" line="19" >
    <Call name="tan_scal" line="11" ></Call>
    <Tan sd="0" />
  </Call>
</Trace>
```

OpenAD tracing (output II)

```
3  subroutine head(x1,x2,y)
4    use OAD_intrinsics
5    real,intent(in) :: x1,x2
6    real,intent(out) :: y
7    integer i
8    !$openad INDEPENDENT(x1)
9    !$openad INDEPENDENT(x2)
10   y=x1
11   do i=int(x1),int(x2)+2
12     y = y*x2
13     if (y>1.0) then
14       y = y*2.0
15     end if
16   end do
17   !$openad DEPENDENT(y)
18 end subroutine head
```

```
<Trace number="1" >
  <Loop line="11" >
    <Branch line="13" >
      <Cfval val="0" />
    </Branch>
    <Branch line="13" >
      <Cfval val="0" />
    </Branch>
    <Branch line="13" >
      <Cfval val="0" />
    </Branch>
    <Cfval val="3" />
  </Loop>
</Trace>
```

note context is the condition (rather than the comparison operator or int)

OpenAD tracing (output III)

```
3  subroutine head(x,y)
4    use OAD_intrinsics
5    real :: x(2),y
6    !$openad INDEPENDENT(x)
7    y=0.0
8    do i=1,2
9      y = y+sin(x(i))+tan(x(i))
10   end do
11   !$openad DEPENDENT(y)
12  end subroutine
```

```
<Trace number="1" >
  <Call name="tan_scal" line="9" >
    <Arg name="X" >
      <Index val="1" />
    </Arg>
  </Call>
  <Tan sd="0" />
  <Call name="tan_scal" line="9" >
    <Arg name="X" >
      <Index val="2" />
    </Arg>
  </Call>
  <Tan sd="0" />
</Trace>
```

note - sine doesn't show up

OpenAD tracing (filtering)

basic filtering - static

- ◇ by file/line number

trace allows encoding an enumeration of nonsmooth events

OpenAD tracing (filtering)

basic filtering - static

- ◇ by file/line number
- ◇ by call stack context

trace allows encoding an enumeration of nonsmooth events

OpenAD tracing (filtering)

basic filtering - static

- ◇ by file/line number
- ◇ by call stack context
- ◇ by argument name (iffy)

trace allows encoding an enumeration of nonsmooth events

OpenAD tracing (filtering)

basic filtering - static

- ◇ by file/line number
- ◇ by call stack context
- ◇ by argument name (iffy)
- ◇ by intrinsic/condition type

trace allows encoding an enumeration of nonsmooth events

OpenAD tracing (filtering)

basic filtering - static

- ◇ by file/line number
- ◇ by call stack context
- ◇ by argument name (iffy)
- ◇ by intrinsic/condition type

dynamic - by comparing traces

trace allows encoding an enumeration of nonsmooth events

OpenAD tracing (filtering)

basic filtering - static

- ◇ by file/line number
- ◇ by call stack context
- ◇ by argument name (iffy)
- ◇ by intrinsic/condition type

dynamic - by comparing traces

- ◇ against a reference
- ◇ subsequent
- ◇ e.g. for time stepping schemes

trace allows encoding an enumeration of nonsmooth events

OpenAD tracing (comparing)

```
<Trace number="1" >  
  <Call name="tan_scal" line="9" >  
    <Arg name="X" >  
      <Index val="1" />  
    </Arg>  
  </Call>  
  <Tan sd="0" />  
  <Call name="tan_scal" line="9" >  
    <Arg name="X" >  
      <Index val="2" />  
    </Arg>  
  </Call>  
  <Tan sd="0" />  
</Trace>
```

```
<Trace number="2" >  
  <Call name="tan_scal" line="9" >  
    <Arg name="X" >  
      <Index val="1" />  
    </Arg>  
  </Call>  
  <Tan sd="0" />  
  <Call name="tan_scal" line="9" >  
    <Arg name="X" >  
      <Index val="2" />  
    </Arg>  
  </Call>  
  <Tan sd="1" />  
</Trace>
```

OpenAD tracing (comparing)

```
<Trace number="1">  
  <Call name="tan_scal" line="9">  
    <Arg name="X">  
      <Index val="1" />  
    </Arg>  
  </Call>  
  <Tan sd="0" />  
  <Call name="tan_scal" line="9">  
    <Arg name="X">  
      <Index val="2" />  
    </Arg>  
  </Call>  
  <Tan sd="0" />  
</Trace>
```

```
<Trace number="2">  
  <Call name="tan_scal" line="9">  
    <Arg name="X">  
      <Index val="1" />  
    </Arg>  
  </Call>  
  <Tan sd="0" />  
  <Call name="tan_scal" line="9">  
    <Arg name="X">  
      <Index val="2" />  
    </Arg>  
  </Call>  
  <Tan sd="1" />  
</Trace>
```

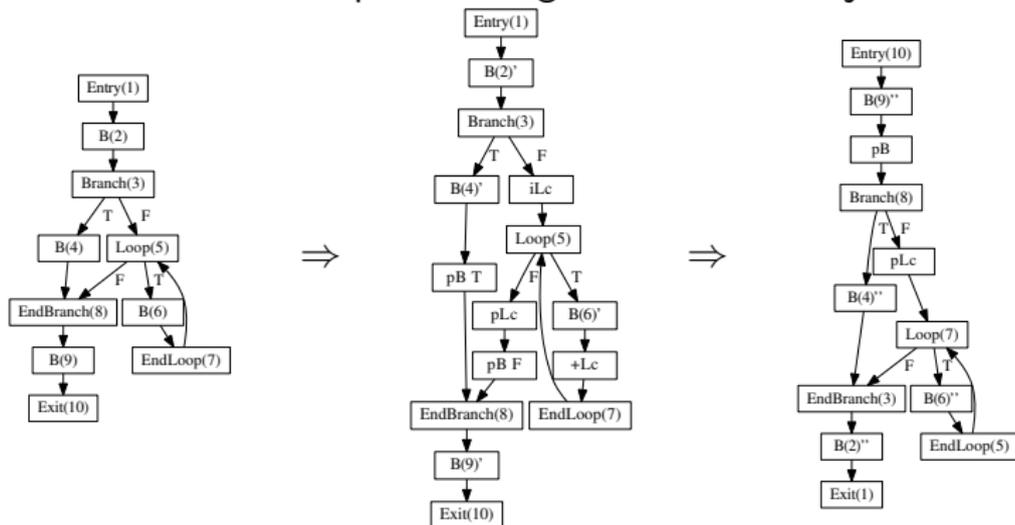
note - tangent subdomain $\left[\frac{x + \frac{\pi}{2}}{\pi} \right]$ changed

Checkpointing

- ◇ have model with high computational complexity and need adjoints
- ◇ spatial requirements (NP complete DAG/call tree reversal)
 - ▶ back to adjoint
- ◇ in theory: no distinction between checkpoints and trace
- ◇ limited automatic support
- ◇ in practice: well defined location for argument checkpoints
 - ◆ fix checkpoint location and spacing (trace fits into memory)
 - ◆ tool determines checkpoint elements
 - ◆ use hierarchical checkpointing (to limit number of checkpoints)
- ◇ optimize scheme e.g. with revolve (uniform steps)

storage also needed for control flow trace and addresses...

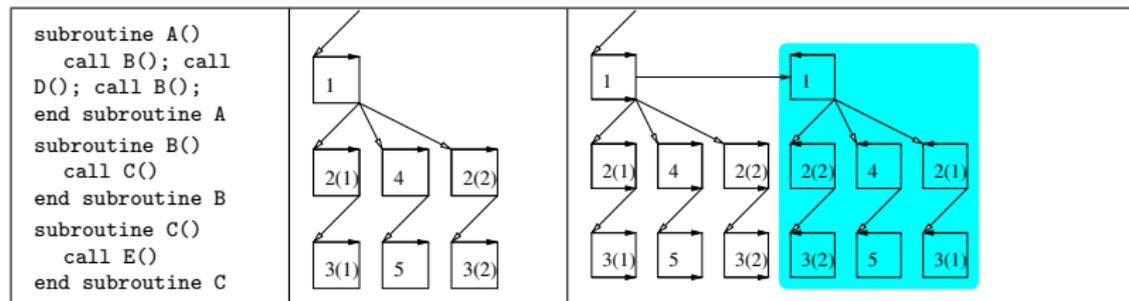
original CFG \Rightarrow record a path through the CFG \Rightarrow adjoint CFG



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



S^n n -th invocation of subroutine S



subroutine call



run forward



order of execution



store checkpoint



restore checkpoint



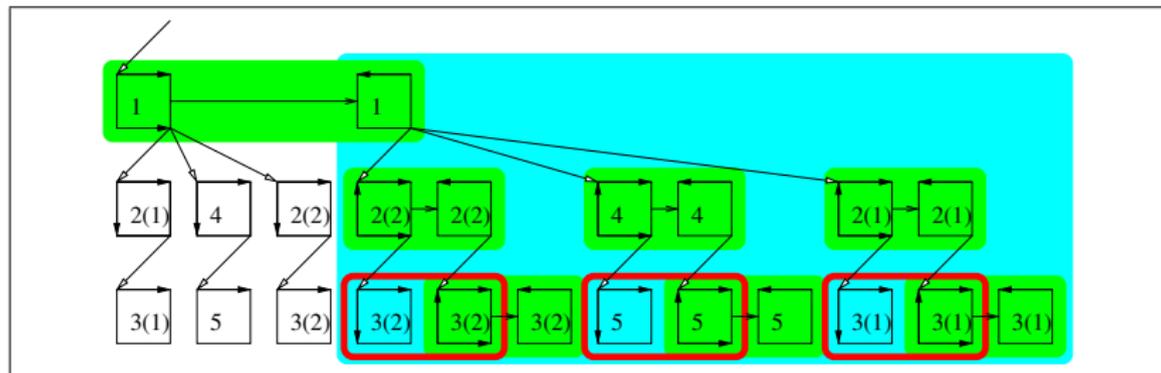
run forward and tape



run adjoint

- ◇ have memory limits - need to create tapes for **short** sections in reverse order
- ◇ subroutine is “natural” checkpoint granularity, different mode...

trace one SR at a time = global *joint* mode



taping-adjoint pairs

checkpoint-recompute pairs

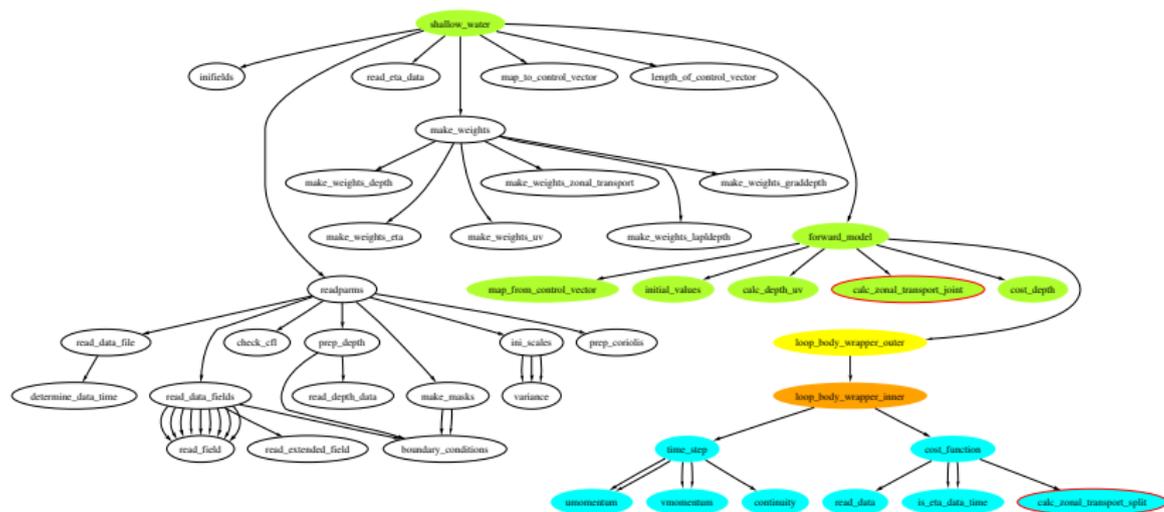
the deeper the call stack - the more recomputations

(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...

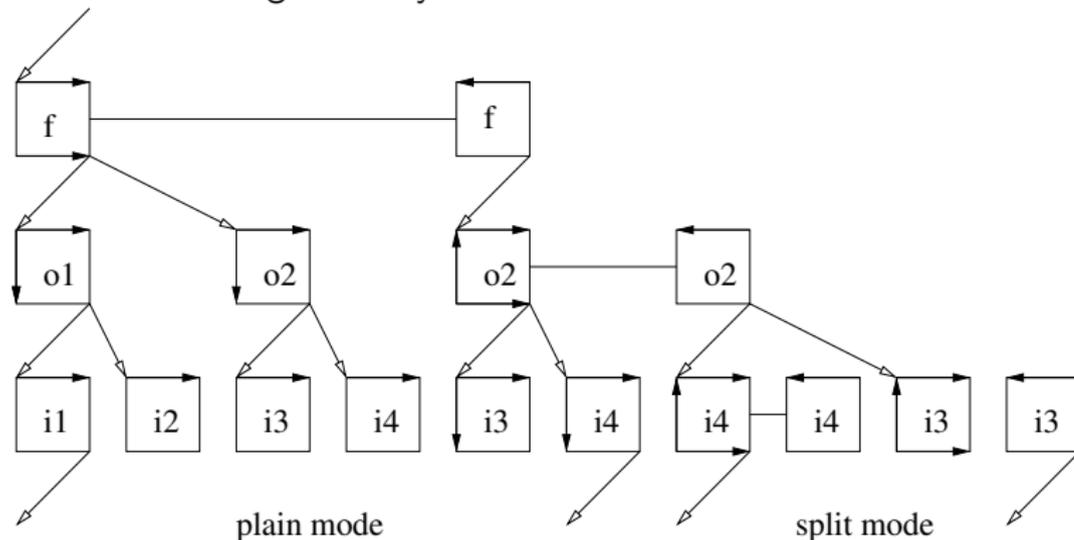
ADified Shallow Water Call Graph



- ◇ mix joint and split mode
- ◇ nested loop checkpointing in **outer** and **inner** loop body wrapper
- ◇ inner loop body in split mode
- ◇ **calc_zonal_transport** is used in both contexts

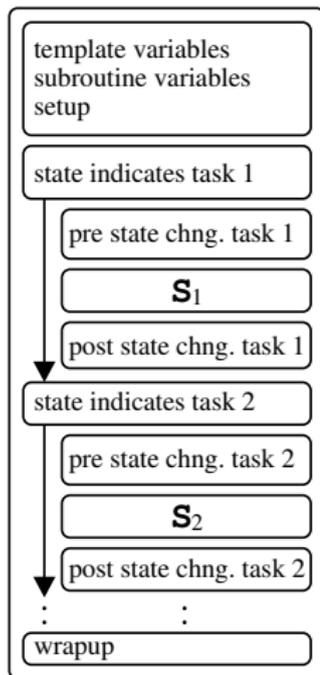
OpenAD reversal modes with checkpointing

subroutine level granularity



in OpenAD orchestrated with templates

- ◇ OpenAnalysis provides side-effect analysis
- ◇ provides checkpoint sets as references to (local/global) variables
- ◇ we ask for four sets: $\text{ModLocal} \subseteq \text{Mod}$, $\text{ReadLocal} \subseteq \text{Read}$



```
subroutine template()  
  use OAD_tape ! tape storage  
  use OAD_rev ! state structure  
  !$TEMPLATE_PRAGMA_DECLARATIONS  
  if (rev_modetape) then  
    ! the state component  
    ! 'taping' is true  
    !$PLACEHOLDER_PRAGMA$ id=2  
    end if  
  
    if (rev_modeadjoin) then  
      ! the state component  
      ! 'adjoint' run is true  
      !$PLACEHOLDER_PRAGMA$ id=3  
      end if  
  
end subroutine template
```

look again at the shallow water example

OpenAD - example shallow water mmodel

- ◇ `cd ~/OpenAD`
- ◇ `./setenv.sh`
- ◇ `cd Examples/ShallowWater/; make clean`
- ◇ `make`
- ◇ look at files under `OADrts`
 - ◆ `wad_template.joint.f`
 - ◆ `ad_template.joint_split_iif.f`
 - ◆ `ad_template.joint_split_oif.f`
 - ◆ `ad_template.split.f`
 - ◆ `ad_template_timing.joint.f`
- ◇ referenced by directives such as
`c$openad XXX Template OADrts/ad_template.split.f`

AD tools applied to practical applications

- ◇ goal: maintain single code base

AD tools applied to practical applications

- ◇ goal: maintain single code base
- ◇ operator overloading
 - ◆ transparent type change
 - ◆ transparent I/O adjustments

AD tools applied to practical applications

- ◇ goal: maintain single code base
- ◇ operator overloading
 - ◆ transparent type change
 - ◆ transparent I/O adjustments
 - ◆ driver logic

AD tools applied to practical applications

- ◇ goal: maintain single code base
- ◇ operator overloading
 - ◆ transparent type change
 - ◆ transparent I/O adjustments
 - ◆ driver logic
 - ◆ preprocessing and templates (C++)

AD tools applied to practical applications

- ◇ goal: maintain single code base
- ◇ operator overloading
 - ◆ transparent type change
 - ◆ transparent I/O adjustments
 - ◆ driver logic
 - ◆ preprocessing and templates (C++)
- ◇ source transformation
 - ◆ "whole source" transformation - regardless of code modularization

AD tools applied to practical applications

- ◇ goal: maintain single code base
- ◇ operator overloading
 - ◆ transparent type change
 - ◆ transparent I/O adjustments
 - ◆ driver logic
 - ◆ preprocessing and templates (C++)
- ◇ source transformation
 - ◆ "whole source" transformation - regardless of code modularization
 - ◆ single "file" transformation step many-to-many make rule problem

AD tools applied to practical applications

- ◇ goal: maintain single code base
- ◇ operator overloading
 - ◆ transparent type change
 - ◆ transparent I/O adjustments
 - ◆ driver logic
 - ◆ preprocessing and templates (C++)
- ◇ source transformation
 - ◆ "whole source" transformation - regardless of code modularization
 - ◆ single "file" transformation step many-to-many make rule problem
 - ◆ no easy separation of numerical core because of syntactic envelopes

AD tools applied to practical applications

- ◇ goal: maintain single code base
- ◇ operator overloading
 - ◆ transparent type change
 - ◆ transparent I/O adjustments
 - ◆ driver logic
 - ◆ preprocessing and templates (C++)
- ◇ source transformation
 - ◆ "whole source" transformation - regardless of code modularization
 - ◆ single "file" transformation step many-to-many make rule problem
 - ◆ no easy separation of numerical core because of syntactic envelopes
- ◇ common issues:
 - ◆ (external) library calls
 - ◆ numerical approximations
 - ◆ coding issues (things to be avoided)

AD tools applied to practical applications

- ◇ goal: maintain single code base
- ◇ operator overloading
 - ◆ transparent type change
 - ◆ transparent I/O adjustments
 - ◆ driver logic
 - ◆ preprocessing and templates (C++)
- ◇ source transformation
 - ◆ "whole source" transformation - regardless of code modularization
 - ◆ single "file" transformation step many-to-many make rule problem
 - ◆ no easy separation of numerical core because of syntactic envelopes
- ◇ common issues:
 - ◆ (external) library calls
 - ◆ numerical approximations
 - ◆ coding issues (things to be avoided)
- ◇ no "standardized" solutions - but have examples for good practice

ADIC: larger code example

pass to Krishna ...

OpenAD: MITgcm example I

- ◇ code design with AD & performance in mind

OpenAD: MITgcm example I

- ◇ code design with AD & performance in mind
- ◇ modularity by packaging selected at configuration time
- ◇ link all selected source files into a single build directory

OpenAD: MITgcm example I

- ◇ code design with AD & performance in mind
- ◇ modularity by packaging selected at configuration time
- ◇ link all selected source files into a single build directory
- ◇ F77 → no compile dependencies
- ◇ numerical core (i.e. the code to AD transformed) identified as a subset

OpenAD: MITgcm example I

- ◇ code design with AD & performance in mind
- ◇ modularity by packaging selected at configuration time
- ◇ link all selected source files into a single build directory
- ◇ F77 → no compile dependencies
- ◇ numerical core (i.e. the code to AD transformed) identified as a subset
- ◇ discretization fixed at configure time → fixed-size arrays and loop bounds

OpenAD: MITgcm example I

- ◇ code design with AD & performance in mind
- ◇ modularity by packaging selected at configuration time
- ◇ link all selected source files into a single build directory
- ◇ F77 → no compile dependencies
- ◇ numerical core (i.e. the code to AD transformed) identified as a subset
- ◇ discretization fixed at configure time → fixed-size arrays and loop bounds
- ◇ → better compiler optimization & less data to trace for control flow reversal

OpenAD: MITgcm example I

- ◇ code design with AD & performance in mind
- ◇ modularity by packaging selected at configuration time
- ◇ link all selected source files into a single build directory
- ◇ F77 → no compile dependencies
- ◇ numerical core (i.e. the code to AD transformed) identified as a subset
- ◇ discretization fixed at configure time → fixed-size arrays and loop bounds
- ◇ → better compiler optimization & less data to trace for control flow reversal
- ◇ know about and exploit self-adjoint operators

OpenAD: MITgcm example I

- ◇ code design with AD & performance in mind
- ◇ modularity by packaging selected at configuration time
- ◇ link all selected source files into a single build directory
- ◇ F77 → no compile dependencies
- ◇ numerical core (i.e. the code to AD transformed) identified as a subset
- ◇ discretization fixed at configure time → fixed-size arrays and loop bounds
- ◇ → better compiler optimization & less data to trace for control flow reversal
- ◇ know about and exploit self-adjoint operators
- ◇ AD specific constructs enabled by preprocessing
- ◇ extensive regression testing, including the adjoint

OpenAD: MITgcm example I

- ◇ code design with AD & performance in mind
- ◇ modularity by packaging selected at configuration time
- ◇ link all selected source files into a single build directory
- ◇ F77 → no compile dependencies
- ◇ numerical core (i.e. the code to AD transformed) identified as a subset
- ◇ discretization fixed at configure time → fixed-size arrays and loop bounds
- ◇ → better compiler optimization & less data to trace for control flow reversal
- ◇ know about and exploit self-adjoint operators
- ◇ AD specific constructs enabled by preprocessing
- ◇ extensive regression testing, including the adjoint

let's have a look... `cd ~/MITgcm/verification/OpenAD/build`

OpenAD: MITgcm example II

- ◇ generated `Makefile` is model-configuration / machine / compiler specific
- ◇ placeholder target `postProcess.tag` for the many-to-one-to-many dependency

OpenAD: MITgcm example II

- ◇ generated `Makefile` is model-configuration / machine / compiler specific
- ◇ placeholder target `postProcess.tag` for the many-to-one-to-many dependency
- ◇ top level routine exposed to OpenAD is in `the_main_loop.F`
- ◇ look for `DEPENDENT/INDEPENDENT` pragmas

OpenAD: MITgcm example II

- ◇ generated `Makefile` is model-configuration / machine / compiler specific
- ◇ placeholder target `postProcess.tag` for the many-to-one-to-many dependency
- ◇ top level routine exposed to OpenAD is in `the_main_loop.F`
- ◇ look for `DEPENDENT/INDEPENDENT` pragmas
- ◇ OpenAD coexists with TAF/TAMC

OpenAD: MITgcm example II

- ◇ generated `Makefile` is model-configuration / machine / compiler specific
- ◇ placeholder target `postProcess.tag` for the many-to-one-to-many dependency
- ◇ top level routine exposed to OpenAD is in `the_main_loop.F`
- ◇ look for `DEPENDENT/INDEPENDENT` pragmas
- ◇ OpenAD coexists with TAF/TAMC
- ◇ time stepping in `main_do_loop.F`

OpenAD: MITgcm example II

- ◇ generated Makefile is model-configuration / machine / compiler specific
- ◇ placeholder target `postProcess.tag` for the many-to-one-to-many dependency
- ◇ top level routine exposed to OpenAD is in `the_main_loop.F`
- ◇ look for `DEPENDENT/INDEPENDENT` pragmas
- ◇ OpenAD coexists with TAF/TAMC
- ◇ time stepping in `main_do_loop.F`
- ◇ template pragmas inserted by script during make

OpenAD: MITgcm example II

- ◇ generated Makefile is model-configuration / machine / compiler specific
- ◇ placeholder target `postProcess.tag` for the many-to-one-to-many dependency
- ◇ top level routine exposed to OpenAD is in `the_main_loop.F`
- ◇ look for `DEPENDENT/INDEPENDENT` pragmas
- ◇ OpenAD coexists with TAF/TAMC
- ◇ time stepping in `main_do_loop.F`
- ◇ template pragmas inserted by script during make
- ◇ notable in

```
ad_input_code_sf.pre.s2p.xb.x2w.w2f.td.ff90:255796
```

```
C$openad XXX Template ../../tools/OAD_support/ad.template.revolve.f
```

Adol-C: ISSM

- ◇ C++ model
cd ~/issm
- ◇ global type change double to IssmDouble and IssmPDouble
look at src/c/shared/Numerics/types.h

Adol-C: ISSM

- ◇ C++ model
cd ~/issm
- ◇ global type change double to IssmDouble and IssmPDouble
look at src/c/shared/Numerics/types.h
- ◇ templated allocation / deallocation
look at ./src/c/shared/MemOps/MemOps.h

Adol-C: ISSM

- ◇ C++ model
cd ~/issm
- ◇ global type change double to IssmDouble and IssmPDouble
look at src/c/shared/Numerics/types.h
- ◇ templated allocation / deallocation
look at ./src/c/shared/MemOps/MemOps.h
- ◇ replace all new/delete and malloc/free

Adol-C: ISSM

- ◇ C++ model
cd ~/issm
- ◇ global type change double to IssmDouble and IssmPDouble
look at src/c/shared/Numerics/types.h
- ◇ templated allocation / deallocation
look at ./src/c/shared/MemOps/MemOps.h
- ◇ replace all new/delete and malloc/free
- ◇ templatize containers
look at ./src/c/toolkits/issm/IssmMat.h

Adol-C: ISSM

- ◇ C++ model
cd ~/issm
- ◇ global type change double to IssmDouble and IssmPDouble
look at src/c/shared/Numerics/types.h
- ◇ templated allocation / deallocation
look at ./src/c/shared/MemOps/MemOps.h
- ◇ replace all new/delete and malloc/free
- ◇ templatize containers
look at ./src/c/toolkits/issm/IssmMat.h
- ◇ treatment of solvers as external functions
look at ./src/c/toolkits/gsl/DenseGslSolve.cpp
- ◇ no brute-force differentiation through the solver code

Adol-C: ISSM

- ◇ C++ model
cd ~/issm
- ◇ global type change double to IssmDouble and IssmPDouble
look at src/c/shared/Numerics/types.h
- ◇ templated allocation / deallocation
look at ./src/c/shared/MemOps/MemOps.h
- ◇ replace all new/delete and malloc/free
- ◇ templatize containers
look at ./src/c/toolkits/issm/IssmMat.h
- ◇ treatment of solvers as external functions
look at ./src/c/toolkits/gsl/DenseGslSolve.cpp
- ◇ no brute-force differentiation through the solver code
- ◇ MPI/PETsc treatment is underway (not in the public SVN repo)
- ◇ passing data to passive code with reCast
look at ./src/c/shared/Numerics/recast.h

Adol-C: ISSM

- ◇ C++ model
cd ~/issm
- ◇ global type change double to IssmDouble and IssmPDouble
look at src/c/shared/Numerics/types.h
- ◇ templated allocation / deallocation
look at ./src/c/shared/MemOps/MemOps.h
- ◇ replace all new/delete and malloc/free
- ◇ templatize containers
look at ./src/c/toolkits/issm/IssmMat.h
- ◇ treatment of solvers as external functions
look at ./src/c/toolkits/gsl/DenseGslSolve.cpp
- ◇ no brute-force differentiation through the solver code
- ◇ MPI/PETsc treatment is underway (not in the public SVN repo)
- ◇ passing data to passive code with reCast
look at ./src/c/shared/Numerics/recast.h
- ◇ reCast injections represent majority of the manual adaptation work

model coding standard & AD tool capabilities I

obvious (by now) recommendations regarding smoothness:

- ◇ avoid introducing numerical special cases
- ◇ pathological cases at domain boundaries, initial conditions
- ◇ filter out computations outside of the actual domain (e.g. $\sqrt{0}$)
- ◇ consider explicit logic to smooth (e.g. C^1 ?) kinks and discontinuities

alternative (to be implemented on demand) approaches:

- ◇ slopes (interval based)
- ◇ Laurent series (w different rules regarding $\pm\text{INF}$ and NaN)

model coding standard and AD tool capabilities II

want: precise compile-time data flow analysis (activity, side effect, etc...)

have: conservative overestimate of aliasing, MOD sets, ...

how to reduce the overestimate:

- ◇ extract the numerical core (if possible)
 - ◆ encapsulate ancillary logic (monitoring, debugging, timing, I/O,...)
 - ◆ small classes, routines, source files (good coding practice anyway)
 - ◆ extraction via source file selection
 - ◆ filtered-out routines (“black box”) - with optimistic(!) assumptions
 - ◆ provide stubs when optimistic assumptions are inappropriate
 - ◆ transformation shielded from dealing with non-numeric language features
 - ◆ note: the top level model driver needs to be manually adjusted
- ◇ avoid semantic ambiguities (void*, union, equivalence)
- ◇ avoid unstructured control flow (analysis, control flow reversal)
- ◇ beware of non-contiguous data, e.g. linked lists (checkpointing, reverse access)
- ◇ beware of indirection, e.g. a[h[i]] vs. a[i] (data dependence)
- ◇ avoid implicit F77 style reshaping (overwrite detection)

model coding standard & AD tool capabilities III

want: to use nice feature \mathcal{N}

have: a tool that has no clue how to deal with \mathcal{N}

- ◇ dynamic resource handling in reverse mode, some examples:
 - ◆ dynamic memory (when locally released)
 - ◆ file handles (same)
 - ◆ MPI communicators (same)
 - ◆ garbage collectors ...

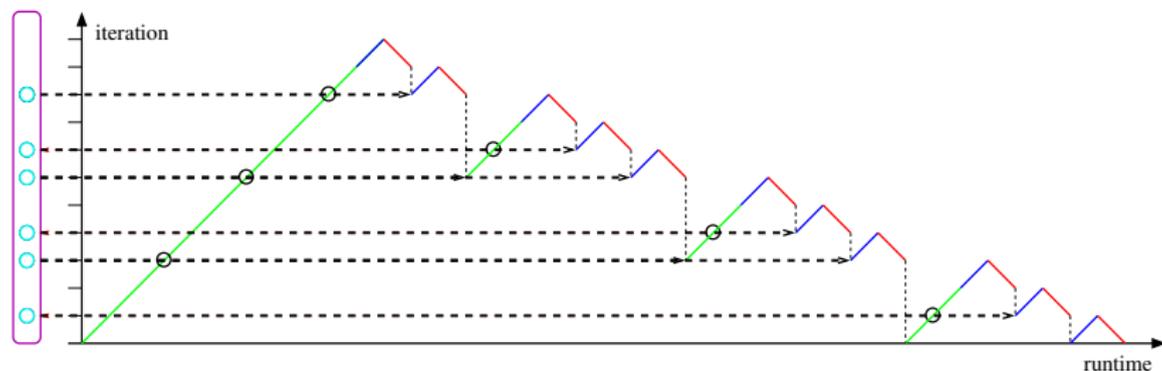
no generic tool support (yet), requires extensive bookkeeping

- ◇ concerns when dealing with third party libraries
 - ◆ availability of the source code
 - ◆ numerical core extraction
 - ◆ smoothness
 - ◆ analysis overhead (e.g. MPI ?)

research underway for blas, lapack, MPI, openMP

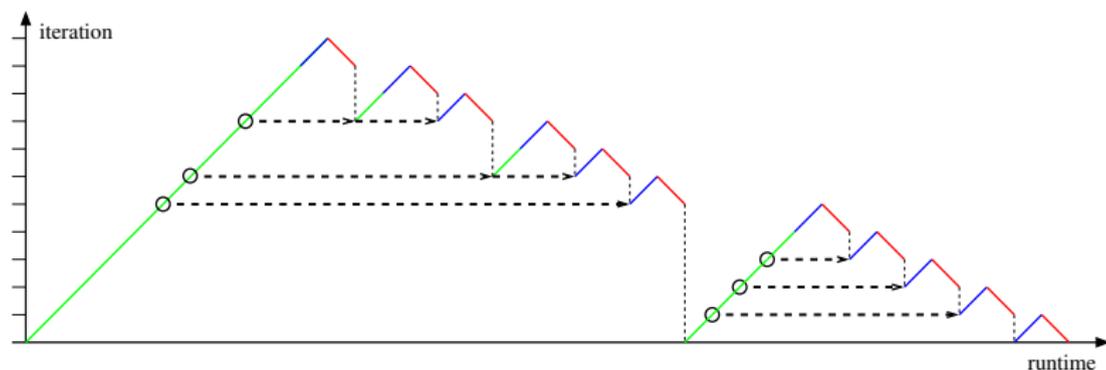
- ◇ beware of out-of-core data dependencies (data transfer via files)

use of checkpointing to mitigate storage requirements



- ◇ 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints

use of checkpointing to mitigate storage requirements



- ◇ 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- ◇ optimal (binomial) scheme encoded in `revolve`; F9X implementation available at <http://mercurial.mcs.anl.gov/ad/RevolveF9X>

external libraries/frameworks (1)

- ◇ interfaces implement *fixed* mathematical meaning
- ◇ may be a “black box” (different language, proprietary)

external 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 $\dot{\mathbf{x}} = \mathbf{A}^{-1}(\dot{\mathbf{b}} - \dot{\mathbf{A}}\mathbf{x})$
 - ◆ $\bar{\mathbf{b}} = \mathbf{A}^{-T}\bar{\mathbf{x}}$; $\bar{\mathbf{A}}_+ = -\bar{\mathbf{b}}\mathbf{x}^T$
- ◇ often requires single call encapsulation

external 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 $\dot{\mathbf{x}} = \mathbf{A}^{-1}(\dot{\mathbf{b}} - \dot{\mathbf{A}}\mathbf{x})$
 - ◆ $\bar{\mathbf{b}} = \mathbf{A}^{-T}\bar{\mathbf{x}}$; $\bar{\mathbf{A}}_+ = -\bar{\mathbf{b}}\mathbf{x}^T$
- ◇ often requires single call encapsulation
- ◇ brute force differentiation as last resort

external 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 $\dot{\mathbf{x}} = \mathbf{A}^{-1}(\dot{\mathbf{b}} - \dot{\mathbf{A}}\mathbf{x})$
 - ◆ $\bar{\mathbf{b}} = \mathbf{A}^{-T}\bar{\mathbf{x}}$; $\bar{\mathbf{A}}_+ = -\bar{\mathbf{b}}\mathbf{x}^T$
- ◇ 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)

external 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 $\dot{\mathbf{x}} = \mathbf{A}^{-1}(\dot{\mathbf{b}} - \dot{\mathbf{A}}\mathbf{x})$
 - ◆ $\bar{\mathbf{b}} = \mathbf{A}^{-T}\bar{\mathbf{x}}$; $\bar{\mathbf{A}}_+ = -\bar{\mathbf{b}}\mathbf{x}^T$
- ◇ 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)
- ◇ efficiency considerations, see “delayed piggyback” e.g. for iterations $\mathbf{x}_{k+1} = \mathbf{f}(\mathbf{x}_k)$

external libraries/frameworks (2)

- ◇ no generic “differentiated” libraries (attempt for MPI)

external libraries/frameworks (2)

- ◇ no generic “differentiated” libraries (attempt for MPI)
- ◇ efficient implementation tied to AD tool implementation

external 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, trinos, petsc)

external 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, trinos, petsc)
- ◇ advanced topics: Taylor coefficient recursions, mathematical mappings split over multiple library calls (reverse mode)

external 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, trinos, petsc)
- ◇ advanced topics: Taylor coefficient recursions, mathematical mappings split over multiple library calls (reverse mode)
- ◇ examples:
 - ◆ UMFPACK:
`cd ~/OpenAD/Examples/LibWrappers/UmfPack_2.2_active`
 - ◆ self-adjoint:
`vi ~/MITgcm/tools/OAD_support/ad_template.sa_cg2d.F`
 - ◆ GSL:
`vi ~/issm/src/c/toolkits/gsl/DenseGslSolve.cpp`

higher order AD (1)

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

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

higher order AD (1)

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

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

- ◇ 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 $a_j^{(i)}, j = 1 \dots o, i = 1 \dots d$ around a common point $a_0 \equiv a_0^i$ in the domain

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

- ◇ 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 $a_j^{(i)}, j = 1 \dots o, i = 1 \dots d$ around a common point $a_0 \equiv a_0^i$ in the domain

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

- ◇ 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 ϕ 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

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

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

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

- ◇ 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 $c = a * b$ we have the convolution

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

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 $c = a * b$ we have the convolution

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

- ◇ 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

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

- ◇ 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 $c = a * b$ we have the convolution

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

- ◇ 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 conveniently implemented via operator and intrinsic overloading (C++, Fortran)

higher order AD (3)

- ◇ higher order AD conveniently 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 conveniently 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)

```
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];
```

higher order AD (3)

- ◇ higher order AD conveniently 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)

```
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

multivariate derivatives - interpolation approach

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

- ◇ direct w multi index management: COSY, AD02,...

multivariate derivatives - interpolation approach

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 - interpolation approach

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

multivariate derivatives - interpolation approach

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 $\mathbf{t} \in \mathbb{N}_0^n$, where each $t_i, i = 1 \dots n$ represents the derivative order with respect to input x_i

multivariate derivatives - interpolation approach

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 $\mathbf{t} \in \mathbb{N}_0^n$, where each $t_i, i = 1 \dots n$ 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 $\mathbf{t} = (1, 1)$.

multivariate derivatives - interpolation approach

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 $\mathbf{t} \in \mathbb{N}_0^n$, where each $t_i, i = 1 \dots n$ 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 $\mathbf{t} = (1, 1)$.
- ◇ interpolation coefficients are precomputed

multivariate derivatives - interpolation approach

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 $\mathbf{t} \in \mathbb{N}_0^n$, where each $t_i, i = 1 \dots n$ 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 $\mathbf{t} = (1, 1)$.
- ◇ interpolation coefficients are precomputed
- ◇ practical advantage can be observed already for small $o > 3$

multivariate derivatives - interpolation approach

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 $\mathbf{t} \in \mathbb{N}_0^n$, where each $t_i, i = 1 \dots n$ 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 $\mathbf{t} = (1, 1)$.
- ◇ 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 -)

multivariate derivatives - tools

- ◇ special purpose tools: COSY, AD for R, Matlab

multivariate derivatives - tools

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

multivariate derivatives - tools

- ◇ 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

```
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;
```

multivariate derivatives - tools

- ◇ 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

```
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;
```

- ◇ look again at Rapsodia
cd ~/RapsodiaExamples/CppStepByStep

Q&A

thanks!

Tangent-linear Models

The tangent-linear model of

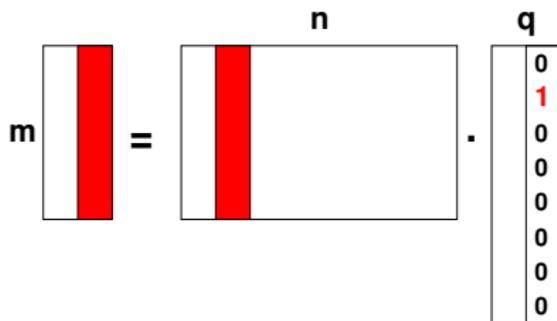
$$f : \mathbb{R}^n \rightarrow \mathbb{R}^m, \quad y = f(x)$$

is

$$\dot{f} : \mathbb{R}^{n+n} \rightarrow \mathbb{R}^m, \quad \dot{y} = \dot{F}(x, \dot{x}) \equiv F'(x) \cdot \dot{x}.$$

Jacobian matrix

$$F' = \left(\frac{\partial y_j}{\partial x_i} \right)_{\substack{j=1, \dots, m \\ i=1, \dots, n}} = F' \cdot I_n$$



column by column at $\mathcal{O}(n)$.

sparse Jacobians

many repeated Jacobian vector products \rightarrow 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 ?

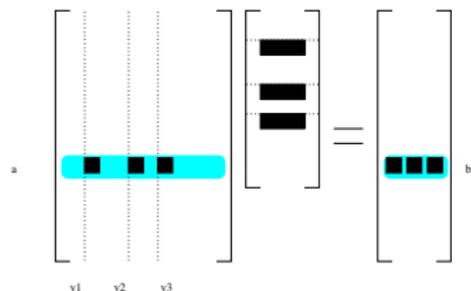
Row i in F' has ρ_i nonzeros in columns $v(1), \dots, v(\rho_i)$

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

$B_i = (\beta_1, \dots, \beta_q) = \beta^T$ We choose S so we can solve:

$$\hat{S}_i \alpha = \beta$$

with $\hat{S}_i^T = (s_{v(1)}, \dots, s_{v(\rho_i)})$

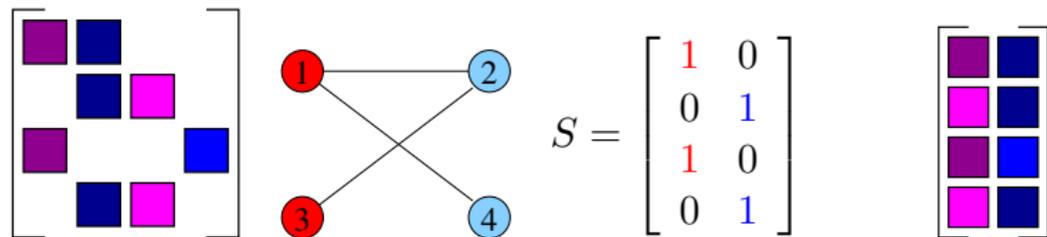


determining q, S (1)

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.



reconstruct F' by relocating nonzero elements (direct)

determining q, S (2)

indirect:

- ◇ Newsam/Ramsdell: $q = \max_i \{\#nonzeros\} \leq \chi$
- ◇ S is a (generalized) Vandermonde matrix
$$\left[\lambda_i^{j-1} \right], \quad j = 1 \dots q, \quad \lambda_i \neq \lambda_{i'}$$
- ◇ How many different λ_i ?

same example

The diagram illustrates the relationship between a sparse matrix structure, its Vandermonde representation S , a graph, and another Vandermonde matrix S .

1. Sparse matrix structure: A 4x4 matrix with colored blocks. Row 1: purple, dark blue. Row 2: dark blue, pink. Row 3: purple, dark blue, blue. Row 4: dark blue, pink.

2. Vandermonde matrix S :
$$S = \begin{bmatrix} \lambda_1^0 & \lambda_1^1 \\ \lambda_2^0 & \lambda_2^1 \\ \lambda_3^0 & \lambda_3^1 \\ \lambda_4^0 & \lambda_4^1 \end{bmatrix}$$

3. Graph: A graph with 4 nodes (1, 2, 3, 4). Nodes 1 and 3 are red, nodes 2 and 4 are blue. Edges connect (1,2), (1,4), (2,3), and (3,4).

4. Vandermonde matrix S :
$$S = \begin{bmatrix} \lambda_1^0 & \lambda_1^1 \\ \lambda_2^0 & \lambda_2^1 \\ \lambda_1^0 & \lambda_1^1 \\ \lambda_2^0 & \lambda_2^1 \end{bmatrix}$$

all combinations of columns (= rows of S): (1, 2), (2, 3), (1, 4)
improved condition via generalization approaches

example with a difference

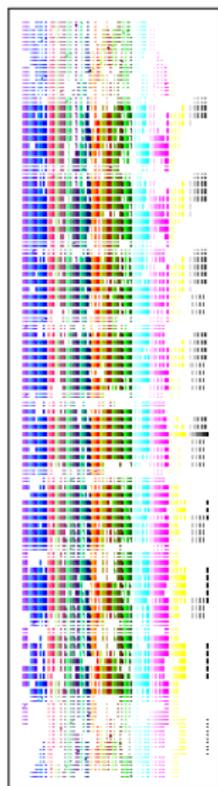
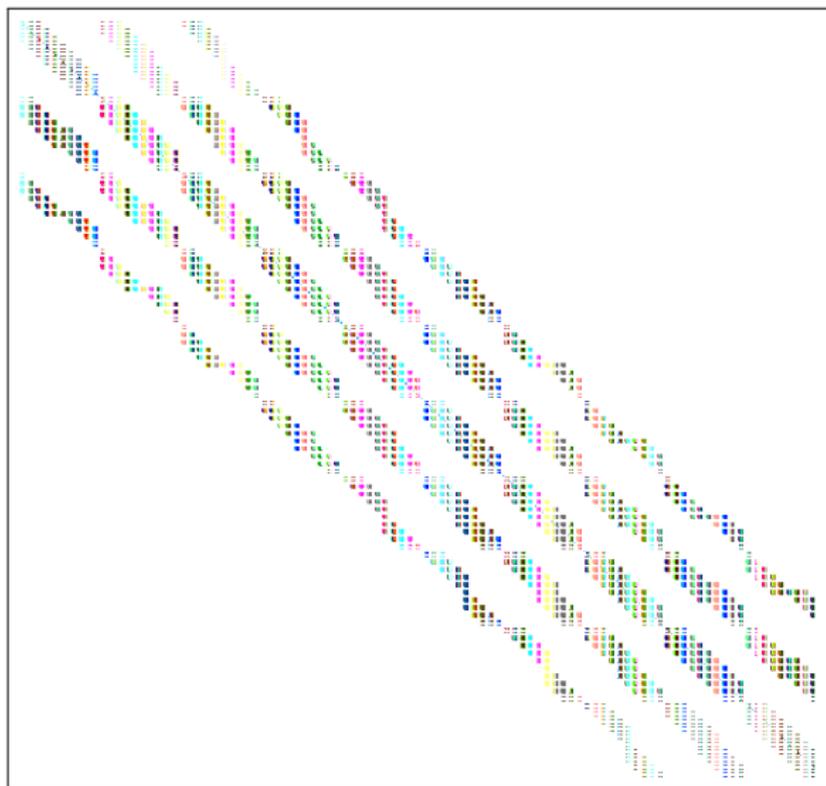
3 colors

$$\begin{bmatrix} a & b & 0 & 0 \\ c & 0 & d & 0 \\ e & 0 & 0 & f \\ 0 & 0 & g & h \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & 0 & f \\ 0 & g & h \end{bmatrix}$$

but with $\lambda \in -1, 0, 1$

$$\begin{bmatrix} a & b & 0 & 0 \\ c & 0 & d & 0 \\ e & 0 & 0 & f \\ 0 & 0 & g & h \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 1 & 0 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} a+b & -a \\ c+d & -c \\ e+f & f-e \\ g+h & h \end{bmatrix}$$

example forward compression



©Hovland

Adjoint Models

The adjoint model of

$$F : \mathbb{R}^n \rightarrow \mathbb{R}^m, \quad y = F(x)$$

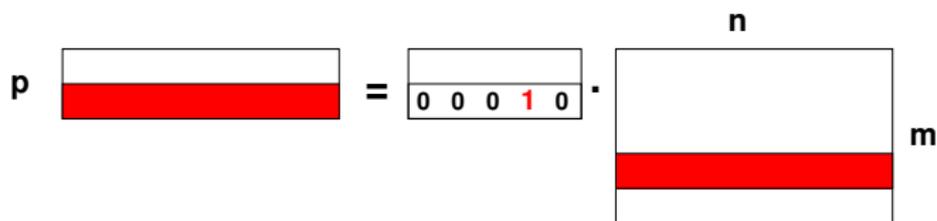
is

$$\bar{F} : \mathbb{R}^{n+m} \rightarrow \mathbb{R}^n, \quad \bar{x} = \bar{F}(x, \bar{y}) \equiv F'(x)^T \cdot \bar{y}.$$

Jacobian matrix

$$F' = \left(\frac{\partial y_j}{\partial x_i} \right)_{\substack{j=1, \dots, m \\ i=1, \dots, n}} = (F')^T \cdot I_m \text{ row by row at } \mathcal{O}(m) \text{ (cheap}$$

gradients ☺, tape intermediates / partials ☹)

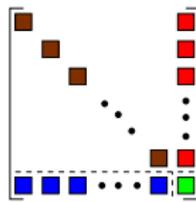


sparse Jacobians (2)

compress the Jacobian:

$F'^T \cdot \bar{S} = B \in \mathbb{R}^{n \times p}$, with a seed matrix $\bar{S} \in \mathbb{R}^{m \times p}$:

Here q as maximal number of nonzeros in columns, or color number in row incidence graph.



Combination through partitioning (Coleman/Verma):

◇ forward sweep
with $q = 2$

◇ reverse sweep
with $p = 1$

$$F' \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ \vdots & \vdots \\ 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} \color{red}\blacksquare & \color{brown}\blacksquare \\ \color{red}\blacksquare & \color{brown}\blacksquare \\ \vdots & \vdots \\ \color{red}\blacksquare & \color{brown}\blacksquare \\ \color{green}\blacksquare & \Sigma \color{blue}\blacksquare \end{bmatrix} \quad \text{and} \quad F'^T \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \color{blue}\blacksquare \\ \color{blue}\blacksquare \\ \vdots \\ \color{blue}\blacksquare \\ \color{green}\blacksquare \end{bmatrix}$$

Adol-C sparsity

sparsity pattern detection (needs ColPack & config flag, ... suggested for homework)

- ◇ safe (conservatively correct) and tight mode, think
 $P(\max(a, b)) = P(a) \mid P(b)$ vs. $P(\max(a, b)) = P(a)$
if $\max(a, b) == a$
- ◇ propagation of unsigned longs
- ◇ forward or reverse
- ◇ convoluted example code in
`examples/additional_examples/sparse/jacpatexam.cpp`
- ◇ e.g. choice -4 with an arrow-like structure (non-negative numbers indicate the use of a test tape)
- ◇ possibility of collecting entries into blocks of rows and columns for (cheaper) block wise propagation using `jac_pat`
 - ◆ -1: contiguous blocks
 - ◆ -2: non-contiguous blocks
 - ◆ -3: one block per variable (as in -4)
- ◇ see also User Guide

Adol-C dependencies

- ◇ example code in `examples/odexam.cpp`

- ◇ rhs $\mathbb{R}^3 \mapsto \mathbb{R}^3$

```
yprime[0] = -sin(y[2]) + 1.0e8*y[2]*(1.0-1.0/y[0]);  
yprime[1] = -10.0*y[0] + 3.0e7*y[2]*(1-y[1]);  
yprime[2] = -yprime[0] - yprime[1];
```

- ◇ uses active vector class `adoublev` (there is also an active matrix class `adboublem` and `along` for active subscripting, see `examples/gaussexam.cpp`)

- ◇ `forode/accode`: generate Taylor coefficients and Jacobians for $x'(t) = F(x(t))$, see User Guide pp. 66

- ◇ nonzero pattern:

```
3 -1 4  
1 2 2  
3 2 4
```

4 = transcend , 3 = rational , 2 = polynomial , 1 = linear , 0 = zero

negative k indicate that entries of all dx_j/dx_0 with $j < -k$ vanish

partial separability

- ◇ reverse mode yields cheap gradient ... at a considerable cost.
- ◇ forward takes $\mathcal{O}(n)$ but sparse Hessian indicates

$$f(\mathbf{x}) = \sum_i a_i f_i(\mathbf{x}_i) + b_i$$

where

$$\mathbf{x}_i \in \mathcal{D}_i \subseteq \mathcal{D} \ni \mathbf{x} \quad \text{so that} \quad \nabla f_i \in \mathbf{R}^{n_i}, n_i \ll n$$

- ◇ use compressed forward propagation
- ◇ research: identify linear sections

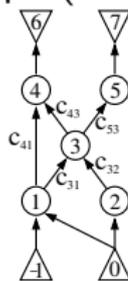
... more general question - how to preserve structure

sidebar: preaccumulation & propagation I

- ◇ propagation = overall mode forward or reverse
- ◇ preaccumulation = local application of chain rule (view as graph operation)
- ◇ example: source code \Rightarrow ssa form \Rightarrow computational graph (DAG)

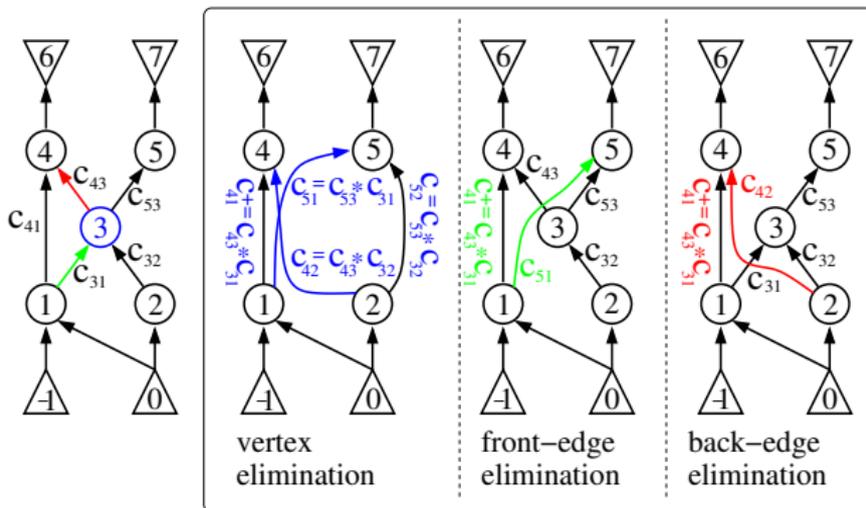
```
t1 = x(1) + x(2)
t2 = t1 + sin(x(2))
y(1) = cos(t1 * t2)
y(2) = -sqrt(t2)
```

$$\Rightarrow \begin{aligned} v_1 &= v_{-1} + v_0; & v_2 &= \sin(v_0); \\ v_3 &= v_1 + v_2; & v_4 &= v_1 * v_3; \\ v_5 &= \sqrt{v_3}; & v_6 &= \cos(v_4); \\ v_7 &= -v_5 \end{aligned} \Rightarrow$$



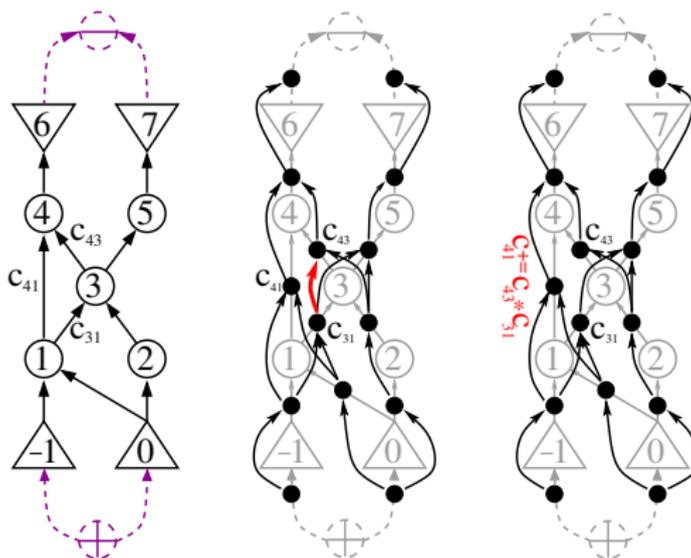
- ◇ chain rule application: multiplication of edge labels along paths & absorption of parallel edges by addition
- ◇ in the graph: elimination of (intermediate) vertices, edges, faces

sidebar: preaccumulation & propagation II



- ◇ efficiency measure is operations count (at runtime)
- ◇ combinatorial problem (heuristics for optimization)
- ◇ problem: granularity \Rightarrow face elimination

sidebar: preaccumulation & propagation III



- ◇ granularity is single fused multiply add
- ◇ also requires heuristics
- ◇ elimination sequence terminates with tripartite dual graph, i.e. Jacobian

sidebar: preaccumulation & propagation IV

have preaccumulated local Jacobians;

given the $\mathbf{J}_i, i = 1, \dots, k$ we want to do:

- ◇ forward: $(\mathbf{J}_k \circ \dots \circ (\mathbf{J}_1 \circ \dot{x}) \dots)$, or
- ◇ reverse: $(\dots (\bar{y}^T \circ \mathbf{J}_k) \circ \dots \circ \mathbf{J}_1)$

the total cost:

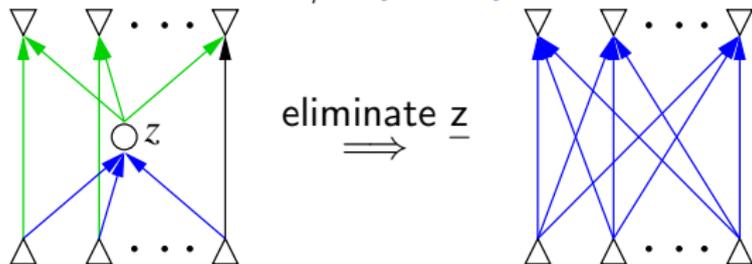
- ◇ function evaluation + local partials (fixed)
- ◇ preaccumulation (NP-hard, varying with heuristic)
- ◇ propagation (fixed for a given preaccumulation)
 - ◆ for simplicity: one saxpy per non-unit \mathbf{J}_i element
 - ◆ potential for n-ary saxpys (generated)

What – other than the preaccumulation heuristic - can vary?

scarcity

observation: Jacobian accumulation can obscure sparse / low rank dependencies

example: consider $f(x) = (D + ax^T)x$ with an intermediate variable $z = x^T x$ that has $\partial z / \partial x_i = 2x_i$

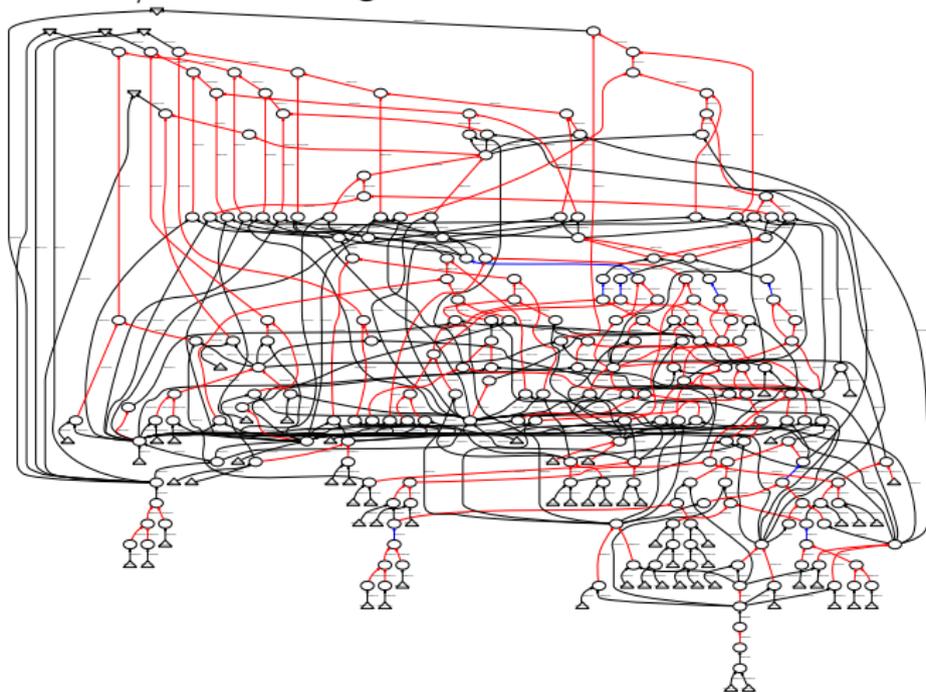


now we have n^2 variable edge labels vs. n variable and $2n$ constant ones

- ◇ want: “minimal” representation
- ◇ scarcity: discrepancy of nm vs dimension of the manifold of all $J(x), x \in \mathcal{D}$
- ◇ required ops: edge eliminations, reroutings, normalization
- ◇ avoid refill, backtrack, randomized heuristics, propagate through remainder graph

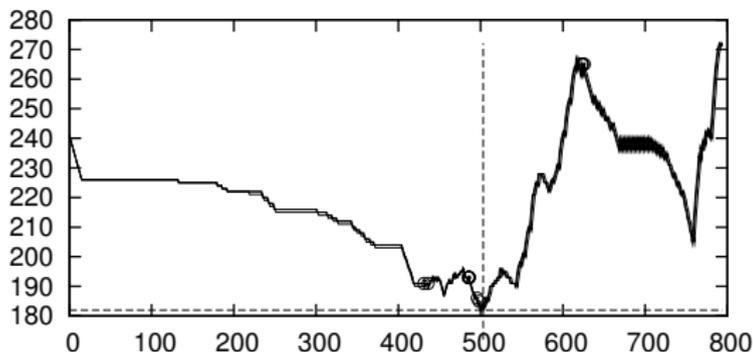
example

DAG with **unit**/**constant** edges



scarcity heuristics - example behavior

non-unit edge count over edge elimination step; variation via avoiding refill:



at minimum 26 reroutings performed; further post-elimination reduction via 8 normalizations

Note: relies heavily on precise data dependency analysis \Leftarrow coding style (!)

similar concerns as with sparsity: (local) automatic improvement observed up to factor 2 but application-level exploitation is desired.