

Adjoint with automatic differentiation: parallel computation and programming language features

Jean Utke¹

¹University of Chicago and Argonne National Laboratory

June 2011



outline

assumption:

discrete adjoints \equiv discretize \rightarrow differentiate

aka automatic differentiation (AD) aka computational differentiation

- motivation
- AD basics, reversal schemes and checkpointing
- parallelism
- AD and programming language features

why automatic 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 automatic 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 automatic 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...

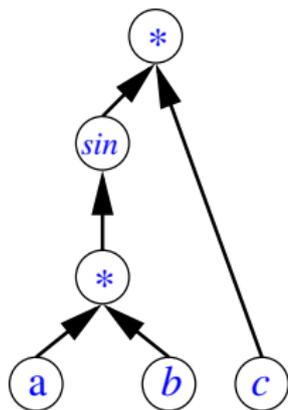
- 1 don't pretend we know nothing about the program
(and take finite differences of an oracle)
- 2 get machine precision derivatives as $\mathbf{J}\dot{\mathbf{x}}$ or $\bar{\mathbf{y}}^T \mathbf{J}$ or ...
(avoid approximation-versus-roundoff problem)
- 3 the reverse (aka adjoint) mode yields “cheap”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

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

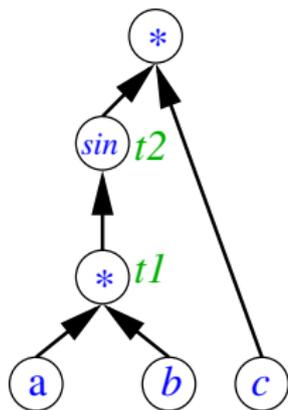
yields a graph representing the order of computation:



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 $t2$

$$t1 = a * b$$

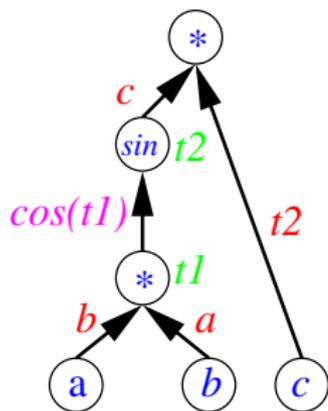
$$t2 = \sin(t1)$$

$$y = t2 * c$$

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 $\frac{\partial \phi}{\partial w}$, $\frac{\partial \phi}{\partial u}$
- e.g. $\sin(t1)$ yields $p1 = \cos(t1)$
- in our example all others are already stored in variables

$$t1 = a * b$$

$$p1 = \cos(t1)$$

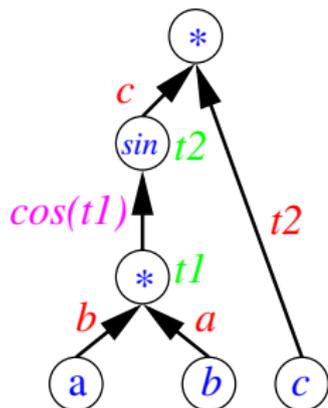
$$t2 = \sin(t1)$$

$$y = t2 * c$$

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

$$t1 = a * b$$

$$p1 = \cos(t1)$$

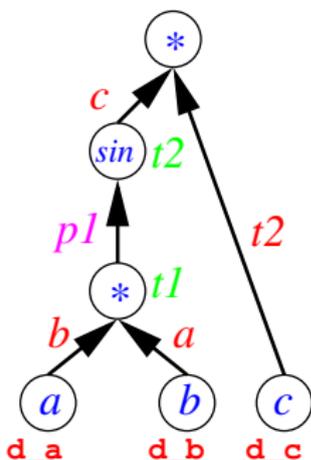
$$t2 = \sin(t1)$$

$$y = t2 * c$$

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

$t1 = a * b$

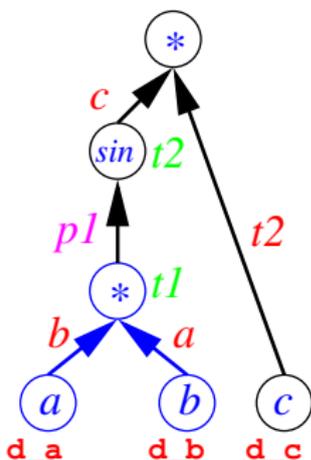
$p1 = \cos(t1)$

$t2 = \sin(t1)$

$y = t2 * c$

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

$$t1 = a * b$$

$$d_t1 = d_a * b + d_b * a$$

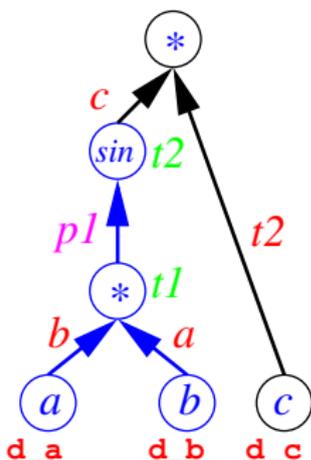
$$p1 = \cos(t1)$$

$$t2 = \sin(t1)$$

$$y = t2 * c$$

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

$$t1 = a * b$$

$$d_t1 = d_a * b + d_b * a$$

$$p1 = \cos(t1)$$

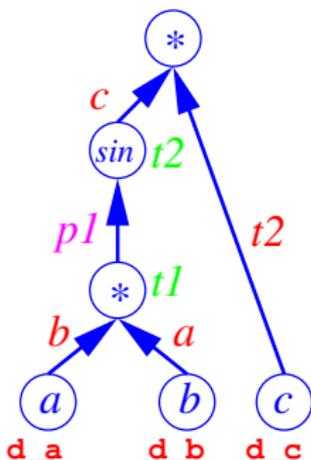
$$t2 = \sin(t1)$$

$$d_t2 = d_t1 * p1$$

$$y = t2 * c$$

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

$$t1 = a * b$$

$$d_t1 = d_a * b + d_b * a$$

$$p1 = \cos(t1)$$

$$t2 = \sin(t1)$$

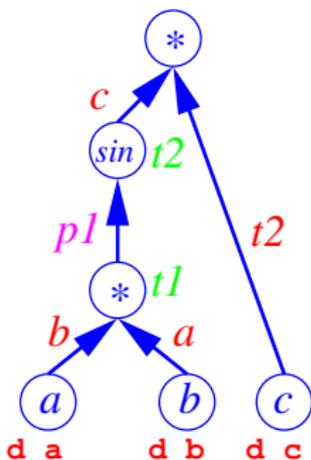
$$d_t2 = d_t1 * p1$$

$$y = t2 * c$$

$$d_y = d_t2 * c + d_c * t2$$

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 ?

\mathbf{d}_y contains a projection

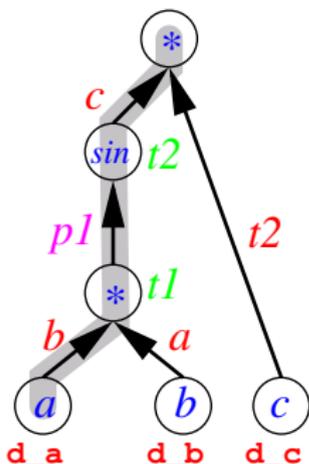
- $\dot{\mathbf{y}} = \mathbf{J}\dot{\mathbf{x}}$ computed at \mathbf{x}_0

\mathbf{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 (airfoil shape optimization, suspended droplets e.g. by Dervieux, Forth, Gauger, Giles et al.)
- beam physics
- mechanical engineering (design optimization)

use

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

applications

for instance

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

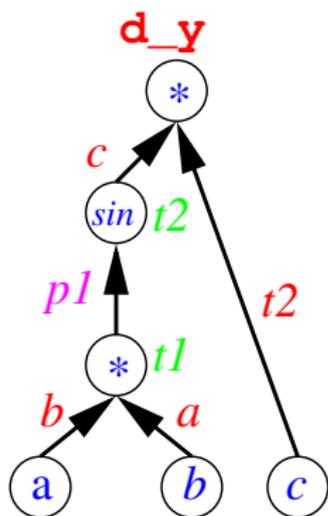
use

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

How do we get the cheap gradients?

reverse mode with adjoints

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

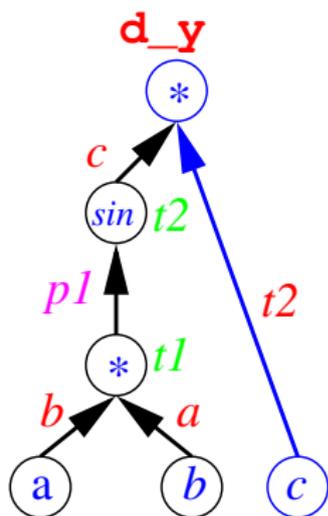


backward propagation code appended:

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

reverse mode with adjoints

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

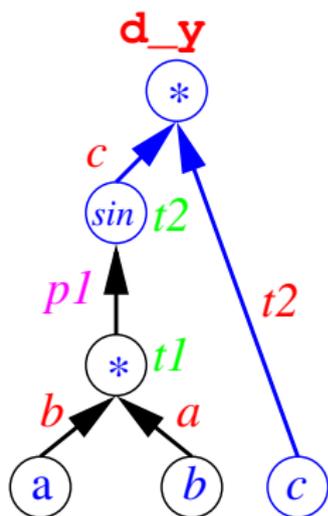


backward propagation code appended:

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

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

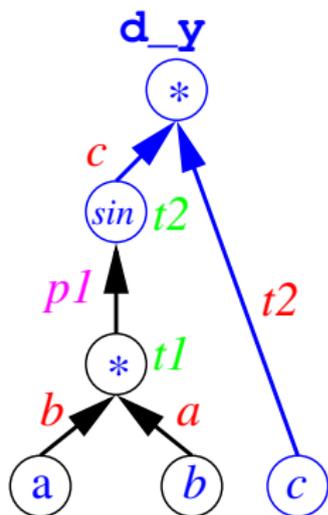


backward propagation code appended:

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

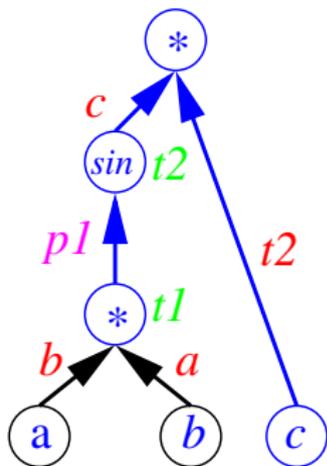


backward propagation code appended:

```
t1 = a*b  
p1 = cos(t1)  
t2 = sin(t1)  
y = t2*c  
d_c = t2*d_y  
d_t2 = c*d_y  
d_y = 0
```

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

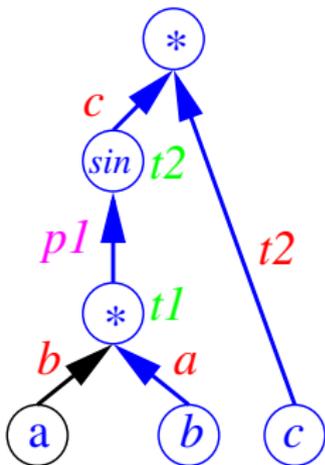


backward propagation code appended:

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

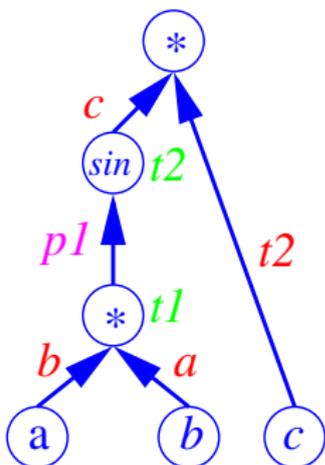


backward propagation code appended:

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

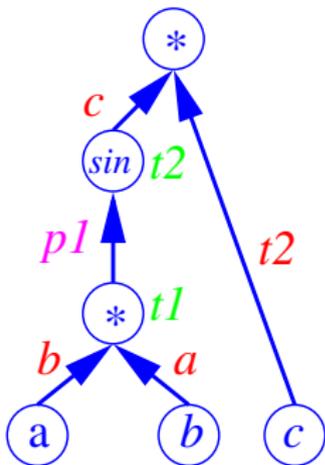


backward propagation code appended:

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



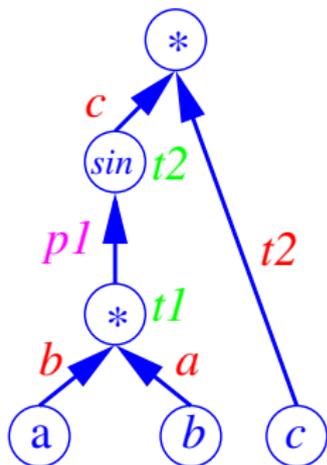
backward propagation code appended:

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

What is in (d_a, d_b, d_c) ?

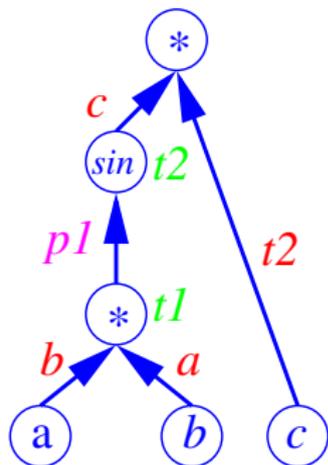
(d_a, d_b, d_c) contains a projection

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



(d_a, d_b, d_c) contains a projection

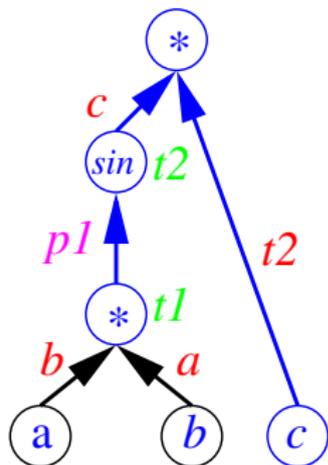
- $\bar{x} = \bar{y}^T J$ computed at x_0
- for example for $\bar{y} = 1$ we have $[\bar{a}, \bar{b}, \bar{c}] = \nabla f$



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

(d_a, d_b, d_c) contains a projection

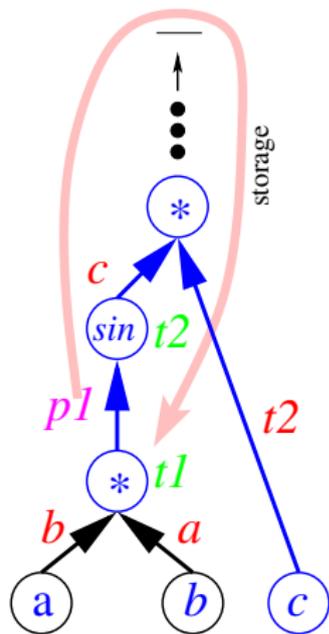
- $\bar{x} = \bar{y}^T J$ computed at x_0
- for example for $\bar{y} = 1$ we have $[\bar{a}, \bar{b}, \bar{c}] = \nabla f$



- 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

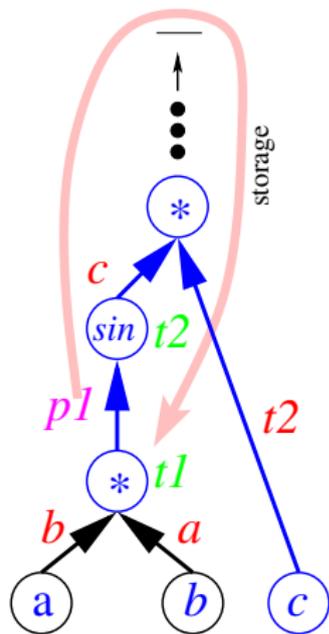
- $\bar{x} = \bar{y}^T J$ computed at x_0
- for example for $\bar{y} = 1$ we have $[\bar{a}, \bar{b}, \bar{c}] = \nabla f$



- 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
- 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
- typically mitigated by recomputation from checkpoints

Reverse mode with Adol-C.

ADOL-C

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

Spelling 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

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

Spelling 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

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

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

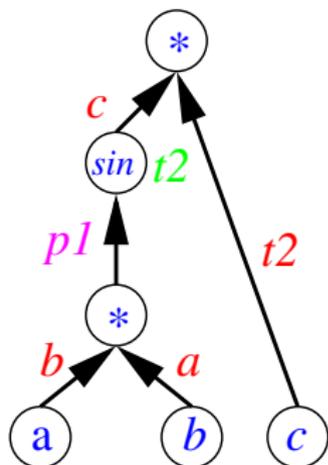
```
#include "adolc.h"
adouble *x = new
adouble[n];
adouble t = 1;
double y;
trace_on(1);
for(i=0; i<n; i++) {
    x[i]
<<= (i+1.0)/(i+2.0);
    t *= x[i]; }
t >>= y;
trace_off();
delete[] x;
```

use a driver :

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

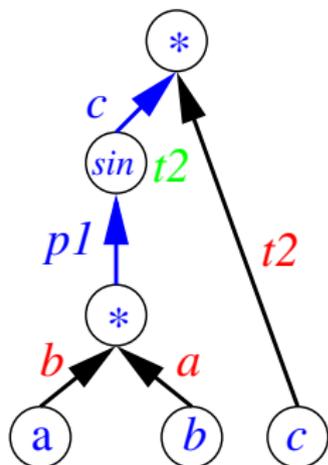
sidebar: preaccumulation & propagation

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



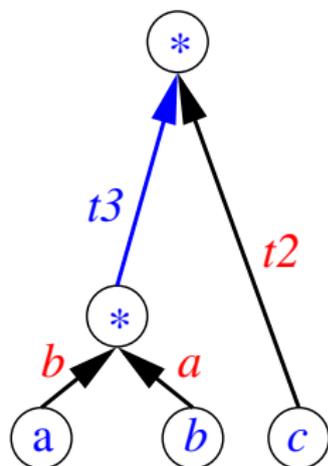
sidebar: preaccumulation & propagation

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



sidebar: preaccumulation & propagation

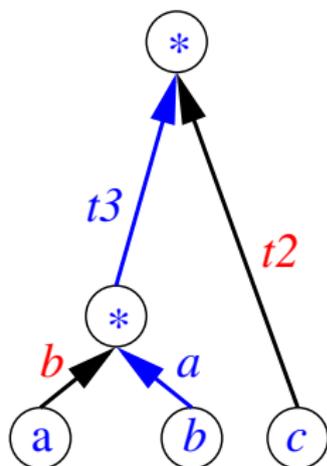
- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians \mathbf{J}
- long program with control flow \Rightarrow sequence of graphs \Rightarrow sequence of \mathbf{J}_i



$t3 = c * p1$

sidebar: preaccumulation & propagation

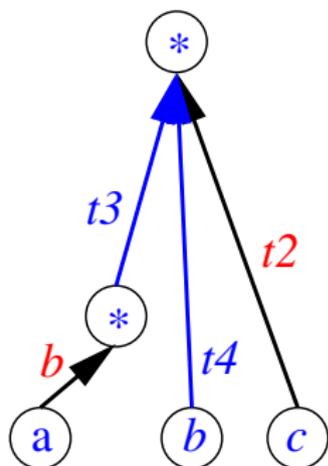
- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians \mathbf{J}
- long program with control flow \Rightarrow sequence of graphs \Rightarrow sequence of \mathbf{J}_i



$t3 = c * p1$

sidebar: preaccumulation & propagation

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

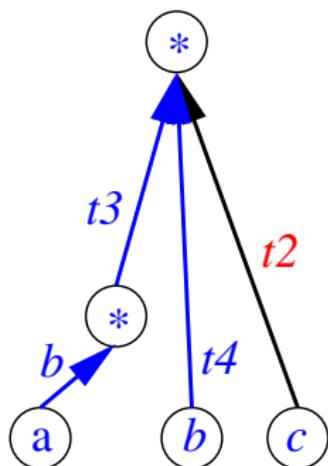


$$t3 = c * p1$$

$$t4 = t3 * a$$

sidebar: preaccumulation & propagation

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

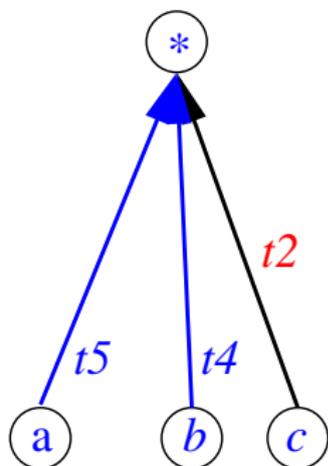


`t3 = c*p1`

`t4 = t3*a`

sidebar: preaccumulation & propagation

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



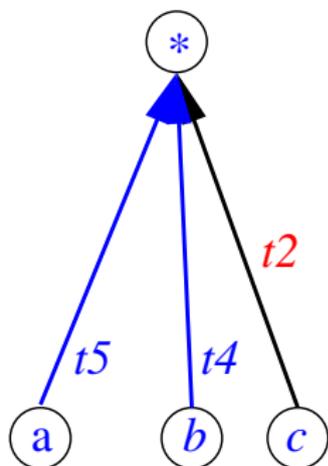
$$t3 = c * p1$$

$$t4 = t3 * a$$

$$t5 = t3 * b$$

sidebar: preaccumulation & propagation

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



$$t3 = c * p1$$

$$t4 = t3 * a$$

$$t5 = t3 * b$$

- $(t5, t4, t2)$ is the preaccumulated \mathbf{J}_i
- $\min_{ops}(\text{preaccumulation})$?
is a combinatorial problem
 \Rightarrow compile time AD optimization!
- forward propagation of $\dot{\mathbf{x}}$
 $(\mathbf{J}_k \circ \dots \circ (\mathbf{J}_1 \circ \dot{\mathbf{x}}) \dots)$
- adjoint propagation of $\bar{\mathbf{y}}$
 $(\dots (\bar{\mathbf{y}}^T \circ \mathbf{J}_k) \circ \dots \circ \mathbf{J}_1)$

sidebar: toy example - source transformation reverse mode

code preparation

numerical “model” program:

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

sidebar: toy example - source transformation reverse mode

code preparation \Rightarrow reverse mode OpenAD pipeline

numerical "model" program:

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

preaccumulation & store J_i :

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

sidebar: toy example - source transformation reverse mode

code preparation \Rightarrow reverse mode OpenAD pipeline

numerical “model” program:

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

preaccumulation & store J_i :

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

sidebar: toy example - source transformation reverse mode

code preparation \Rightarrow reverse mode OpenAD pipeline

numerical “model” program:

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

preaccumulation & store J_i :

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

sidebar: toy example - source transformation reverse mode

code preparation \Rightarrow reverse mode OpenAD pipeline

numerical “model” program:

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

preaccumulation & store J_i :

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

retrieve stored J_i & propagate:

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

sidebar: toy example - source transformation reverse mode

code preparation \Rightarrow reverse mode OpenAD pipeline

numerical “model” program:

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

preaccumulation & store J_i :

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

retrieve stored J_i & propagate:

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

sidebar: toy example - source transformation reverse mode

code preparation \Rightarrow reverse mode OpenAD pipeline

numerical “model” program:

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

preaccumulation & store J_i :

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

retrieve stored J_i & propagate:

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

sidebar: toy example - source transformation reverse mode

code preparation \Rightarrow reverse mode OpenAD pipeline

\Rightarrow adapt the driver routine

numerical “model” program:

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

driver modified for reverse mode:

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

preaccumulation & store J_i :

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

retrieve stored J_i & propagate:

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

sidebar: toy example - source transformation reverse mode

code preparation \Rightarrow reverse mode OpenAD pipeline

\Rightarrow adapt the driver routine

numerical “model” program:

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

driver **modified for reverse mode**:

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

preaccumulation & store J_i :

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

retrieve stored J_i & propagate:

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

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

source transformation vs. operator overloading

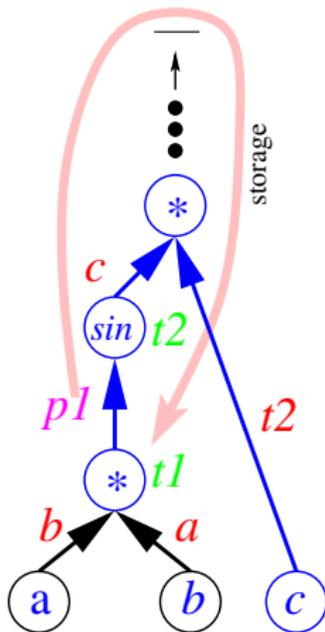
- complicated implementation of tools
 - especially for reverse mode
 - full front end, back end, analysis
 - efficiency gains from
 - **compile time AD optimizations**
 - activity analysis
 - explicit control flow reversal
 - source transformation based type change & overloaded operators appropriate for higher-order derivatives.
 - efficiency depends on analysis accuracy
- simple tool implementation
 - reverse mode: generate & reinterpret an execution trace → **inefficient**
 - implemented as a library
 - efficiency gains from:
 - runtime AD optimization
 - optimized library
 - inlining (for low order)
 - manual type change
 - ⚡ formatted I/O, allocation,...
 - matching signatures (Fortran)
 - easier with templates

higher-order derivatives \Rightarrow source transformation based type change
+ overloaded operators.

Reversal Schemes

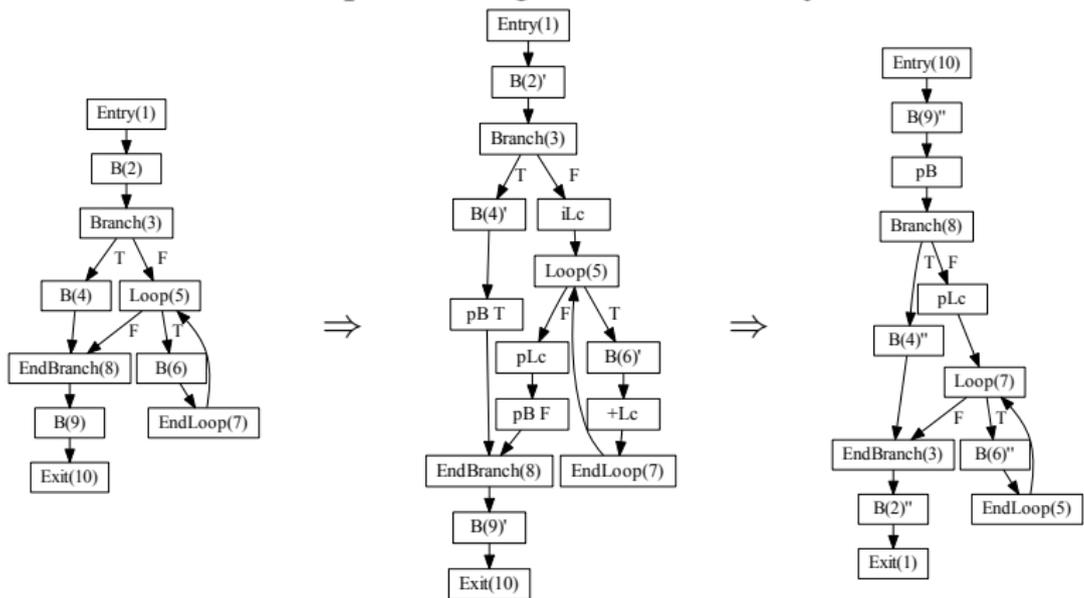
- why it is needed
- major modes
- alternatives

recap: store intermediate values / partials



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

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

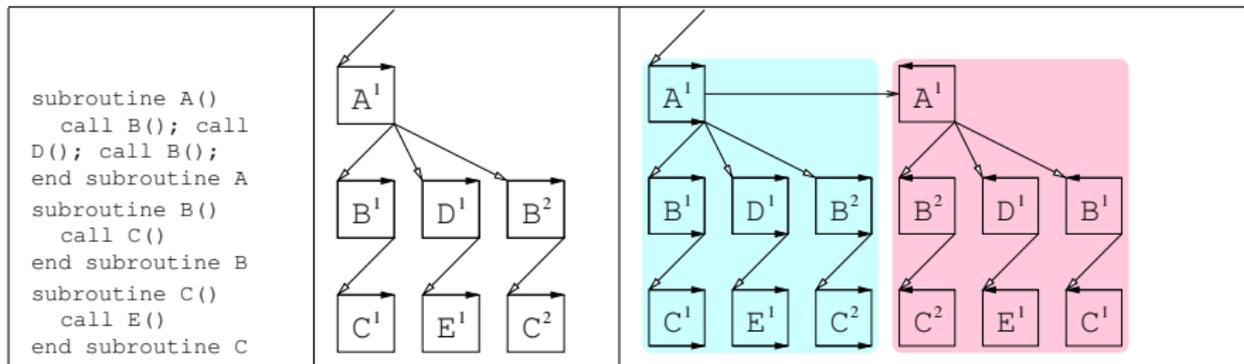


often cheap with **structured control flow** and **simple address computations** (e.g.

index from loop variables)

unstructured control flow and **pointers** are expensive

trace all at once = global *split* mode



S^n
n-th invocation of subroutine S



subroutine call



run forward



order of execution



store checkpoint



restore checkpoint



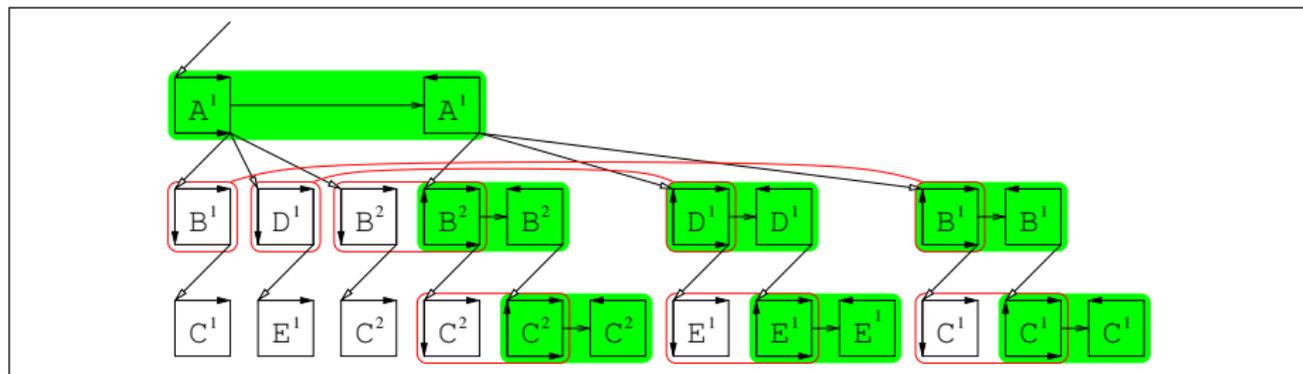
run forward and tape



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...

use of checkpointing to mitigate storage requirements



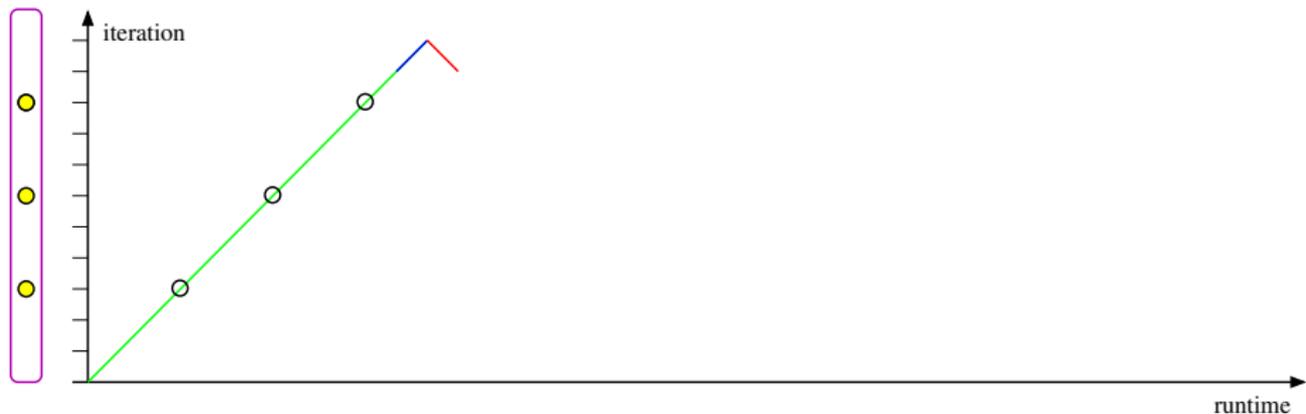
- 11 iters.

use of checkpointing to mitigate storage requirements



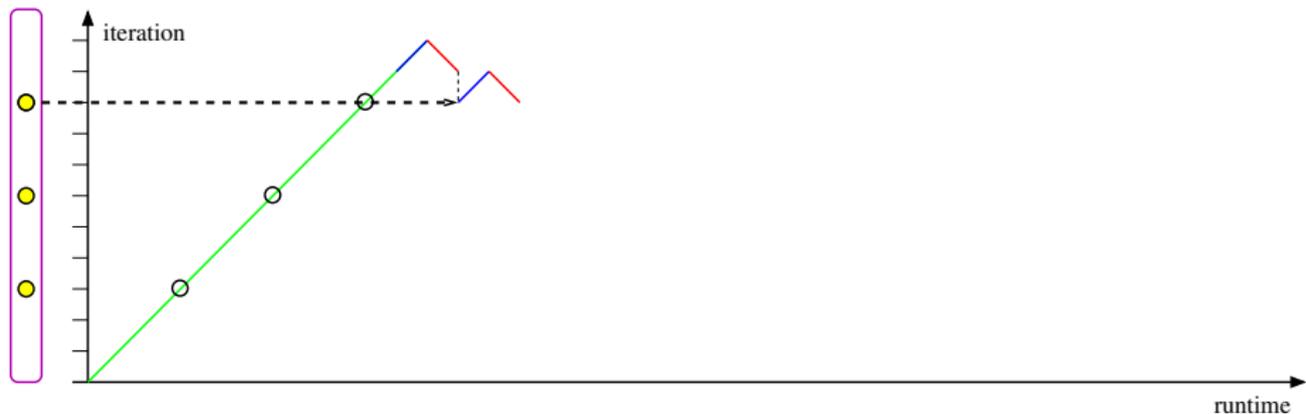
- 11 iters., memory limited to one iter. of storing J_i
- run forward, store the last step, and adjoint

use of checkpointing to mitigate storage requirements



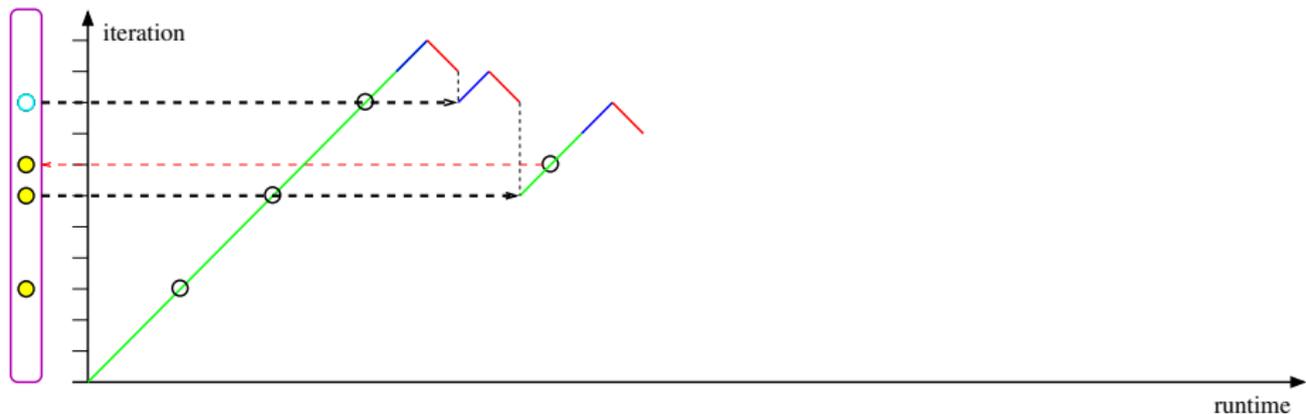
- 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- run forward, store the last step, and adjoint

use of checkpointing to mitigate storage requirements



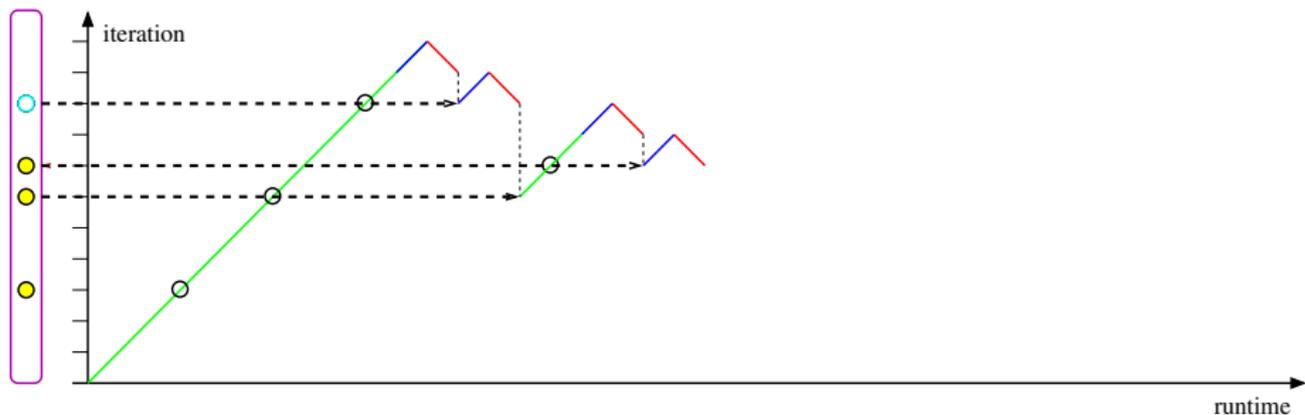
- 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- run forward, store the last step, and adjoint
- restore checkpoints and recompute

use of checkpointing to mitigate storage requirements



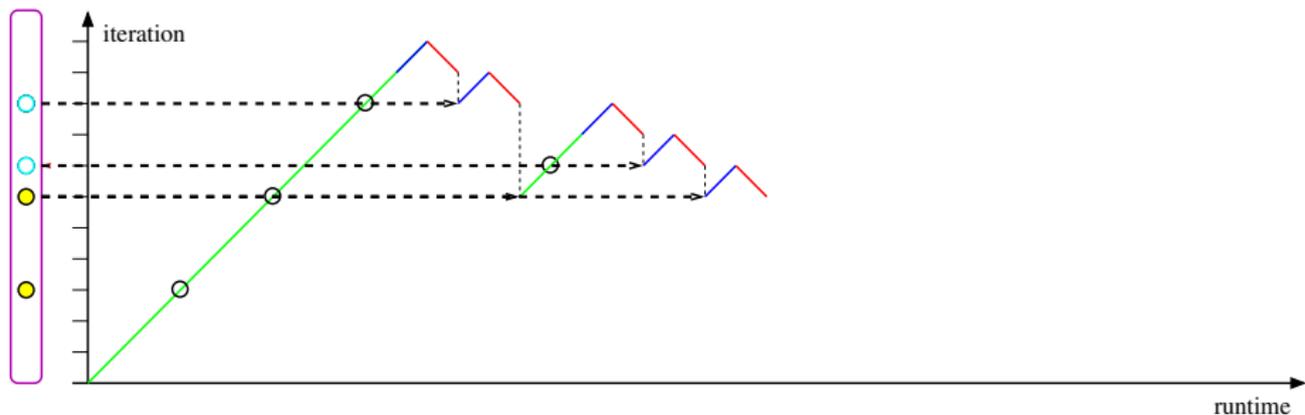
- 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- run forward, store the last step, and adjoint
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

use of checkpointing to mitigate storage requirements



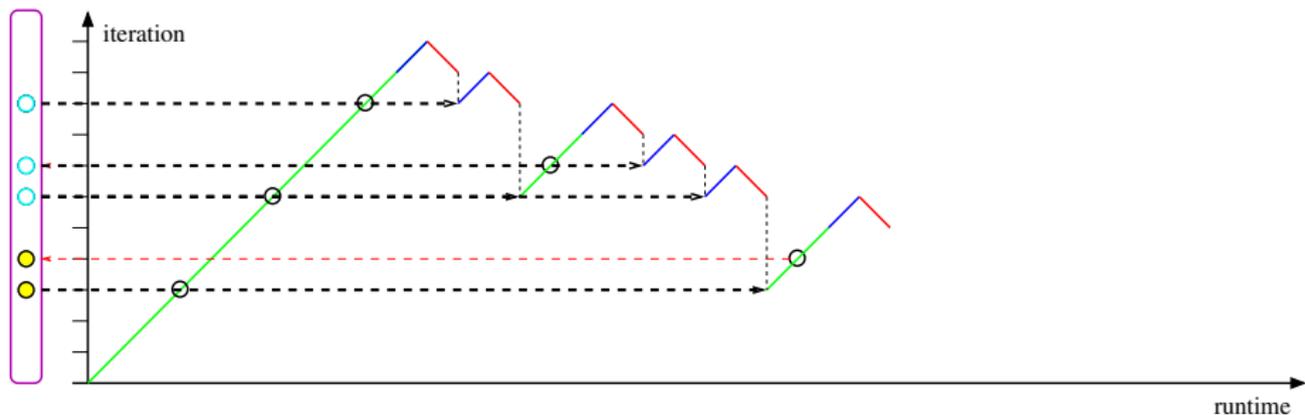
- 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- run forward, store the last step, and adjoint
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

use of checkpointing to mitigate storage requirements



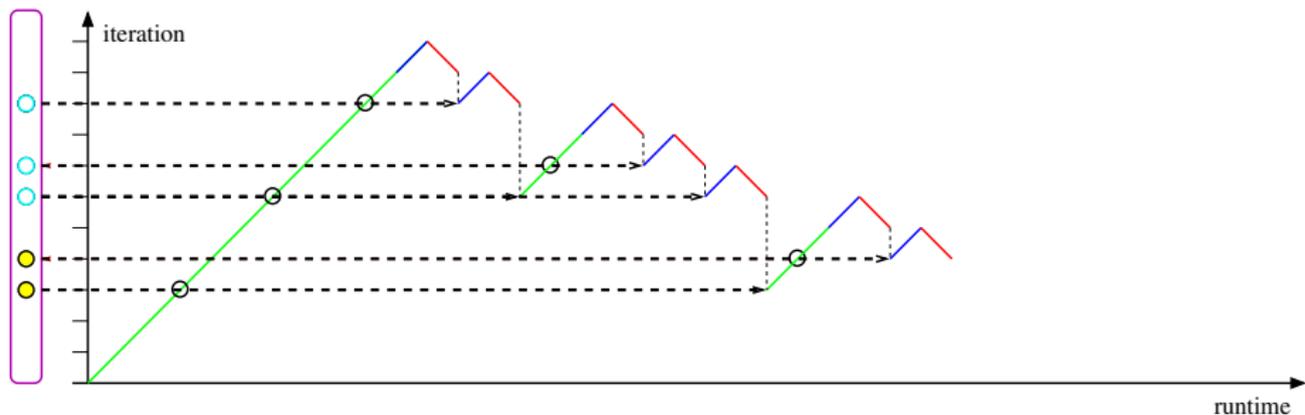
- 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- run forward, store the last step, and adjoint
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

use of checkpointing to mitigate storage requirements



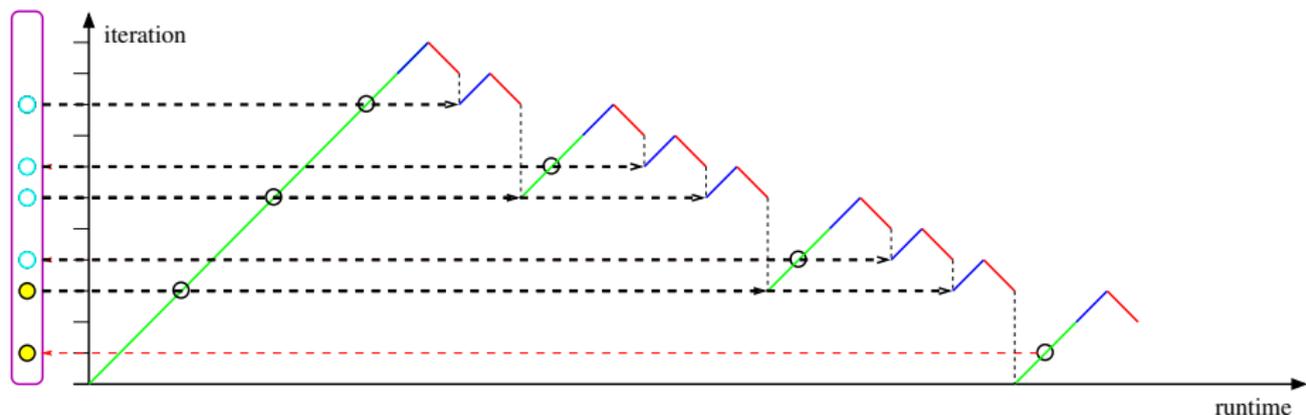
- 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- run forward, store the last step, and adjoint
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

use of checkpointing to mitigate storage requirements



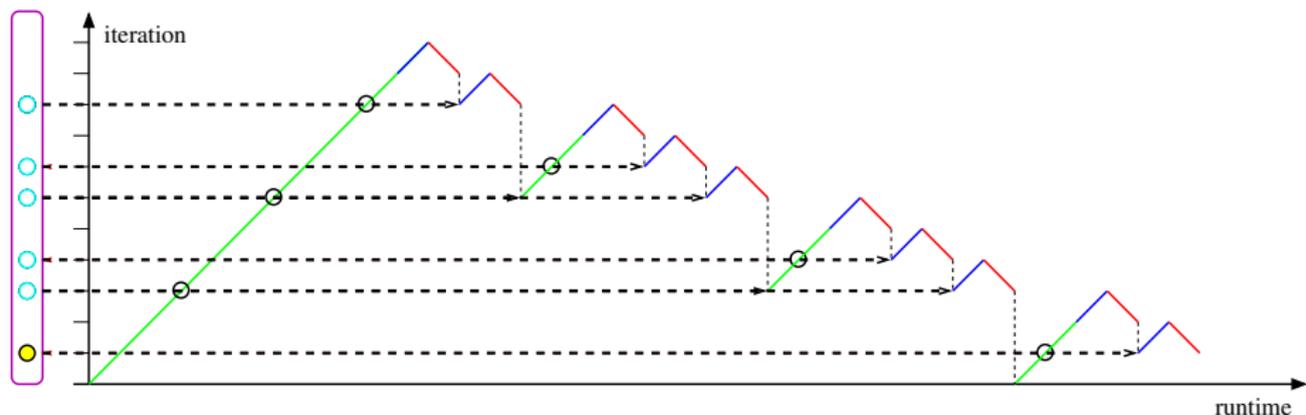
- 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- run forward, store the last step, and adjoint
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

use of checkpointing to mitigate storage requirements



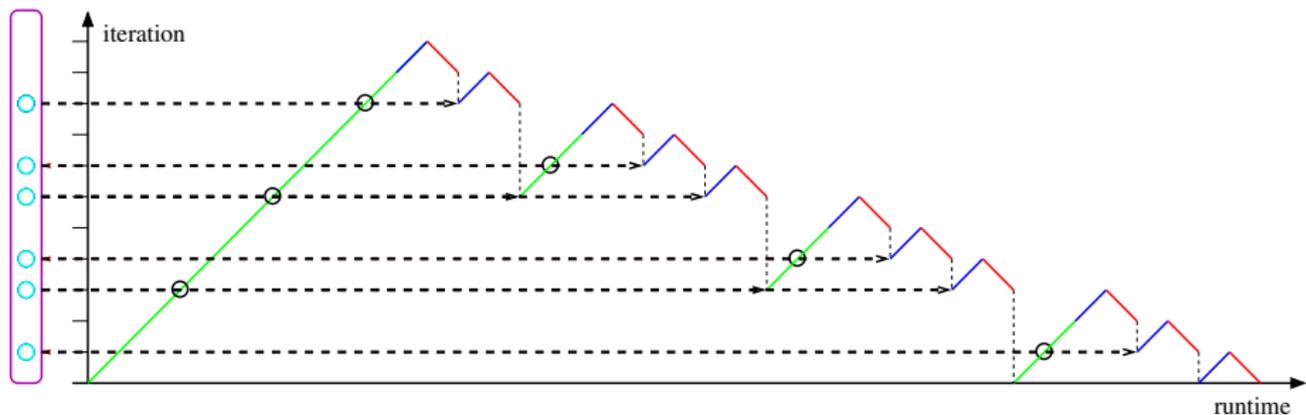
- 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- run forward, store the last step, and adjoint
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

use of checkpointing to mitigate storage requirements



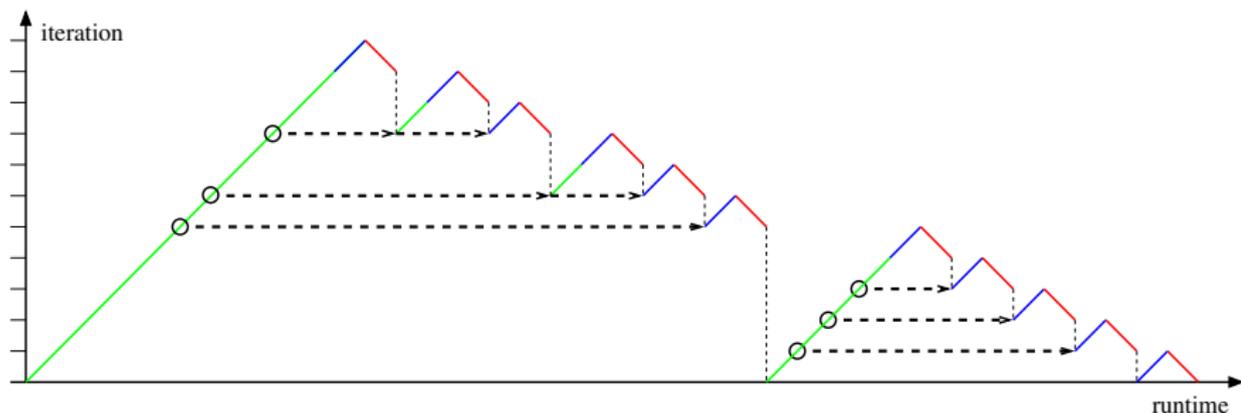
- 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- run forward, store the last step, and adjoint
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

use of checkpointing to mitigate storage requirements



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

use of checkpointing to mitigate storage requirements



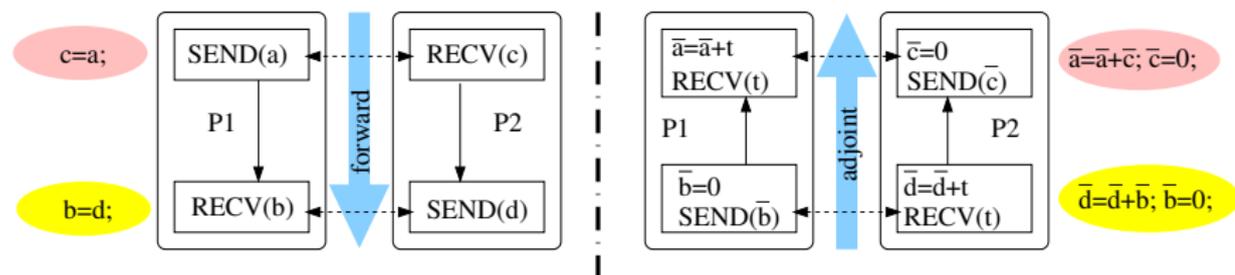
- 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- run forward, store the last step, and adjoint
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints
- optimal (binomial) scheme encoded in `revolve`;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

MPI - data transfer between processes

- simple MPI program needs 6 calls :

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

- example adjoining blocking communication between 2 processes:



- use the communication graph as model

MPI - halo exchange

- MPI usage in the MITgcm ocean model: exchange tile halos, reductions operations. synchronization...

MPI - halo exchange

- MPI usage in the MITgcm ocean model: exchange tile halos, reductions operations. synchronization...
- interfaces to various languages (here using Fortran), C implementation
- total 287 routines (MPI-2)
- covering: communication, setup, grouping of processes, I/O, status queries, topologies, debugging,...

MPI - halo exchange

- MPI usage in the MITgcm ocean model: exchange tile halos, reductions operations. synchronization...
- interfaces to various languages (here using Fortran), C implementation
- total 287 routines (MPI-2)
- covering: communication, setup, grouping of processes, I/O, status queries, topologies, debugging,...
- concentrate on portion “relevant” for AD (e.g. ignore one-sided comm. ?)
- need to enable **activity analysis**

MPI - halo exchange

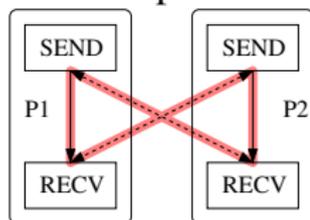
- MPI usage in the MITgcm ocean model: exchange tile halos, reductions operations. synchronization...
- interfaces to various languages (here using Fortran), C implementation
- total 287 routines (MPI-2)
- covering: communication, setup, grouping of processes, I/O, status queries, topologies, debugging,...
- concentrate on portion “relevant” for AD (e.g. ignore one-sided comm. ?)
- need to enable **activity analysis**
- consider the *communication modes*:
 - for send: `mpi_[i] [b|s|r] send`
 - for receive: `mpi_[i] recv`

MPI - halo exchange

- MPI usage in the MITgcm ocean model: exchange tile halos, reductions operations. synchronization...
- interfaces to various languages (here using Fortran), C implementation
- total 287 routines (MPI-2)
- covering: communication, setup, grouping of processes, I/O, status queries, topologies, debugging,...
- concentrate on portion “relevant” for AD (e.g. ignore one-sided comm. ?)
- need to enable **activity analysis**
- consider the *communication modes*:
 - for send: `mpi_[i] [b|s|r] send`
 - for receive: `mpi_[i] recv`
- ensure **correctness** and aim at improving **efficiency**
⇒ want the same for adjoints

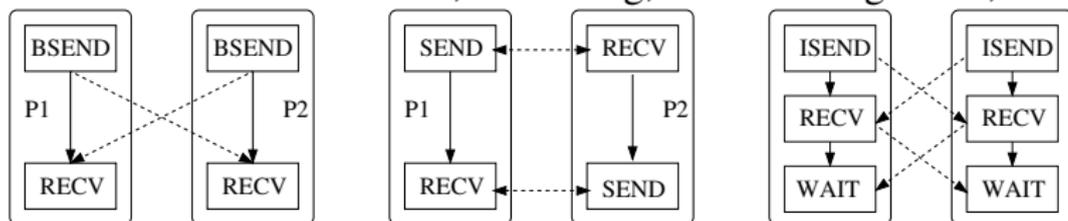
correctness as in ...

- correct parameters ? (data, endpoints)
- no deadlocks ? (look at communication graphs)
- for example: data exchange between P1 and P2



... has a cycle (involving comm.edges)

- break with buffered* sends, reordering, non-blocking sends, ...

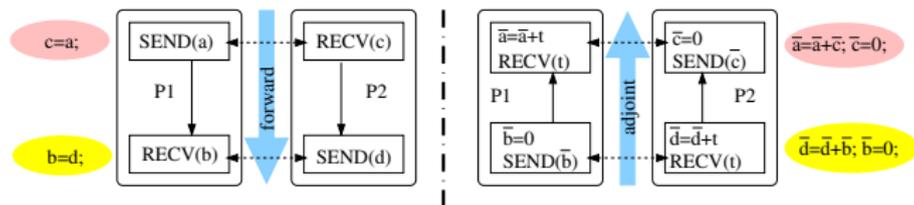


the last idiom is used in MITgcm

* resource starvation?

easy adjoints for blocking calls

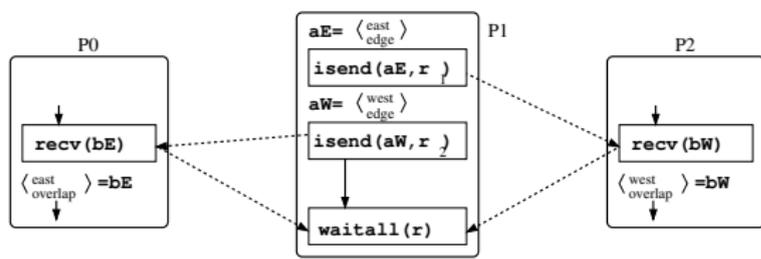
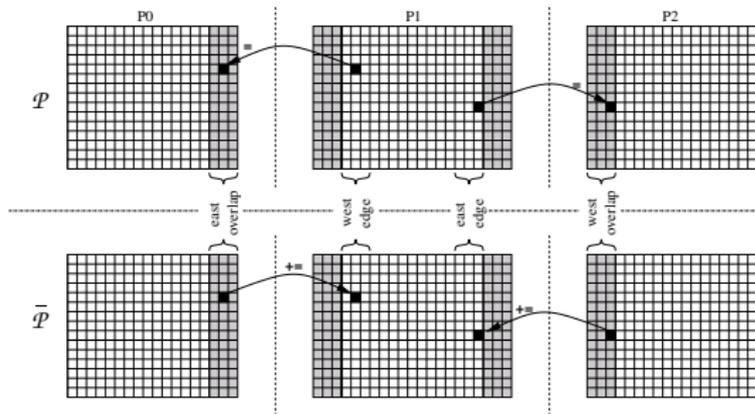
- easy adjoint transformation: $\text{send} \mapsto \text{recv}$ and $\text{recv} \mapsto \text{send}$



- hyp.: if the forward communication graph is acyclic, so is the adjoint; look at the communication graph with reversed edges
- for activity analysis: difficult to statically determine send/recv pairs; e.g. consider set of all possible dynamic comm. graphs
- with wildcards (but no threads): record actual sources/tags on receive and send with recorded tag to recorded source in the adjoint sweep
- hyp.: no forward deadlock \equiv no cycle in current dynamic comm. graph \Rightarrow no cycle in inverted dynamic comm. graph \equiv no adjoint deadlock

MITgcm uses manual adjoints for halo exchange

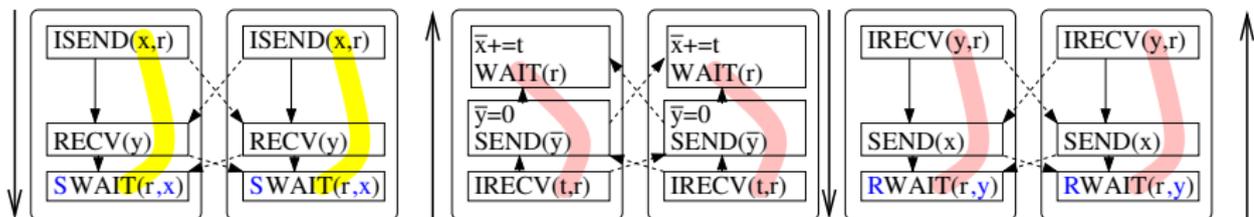
actually in E-W and N-S; need to consider corners...



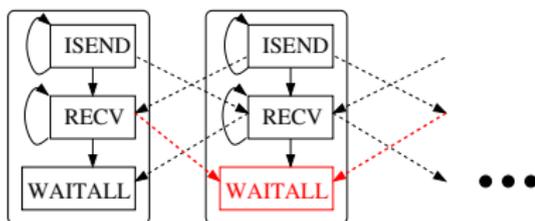
- every tile needs to talk to its neighbors
- prevent deadlocks by imposing ordering ? (requires high-level view and imposes order)
- prevent deadlocks by buffered communication? (can run out of bufferspace)
- use non-blocking calls, the idiom used here is
`ISEND* - RECV* - WAITALL`
- no “easy” transformation;
multiple in-edges at
`waitall`

no easy transformation because ...

- consider the communication graphs for simple nonblocking idioms
- need to retain correctness, i.e. use nonblocking calls in the adjoint

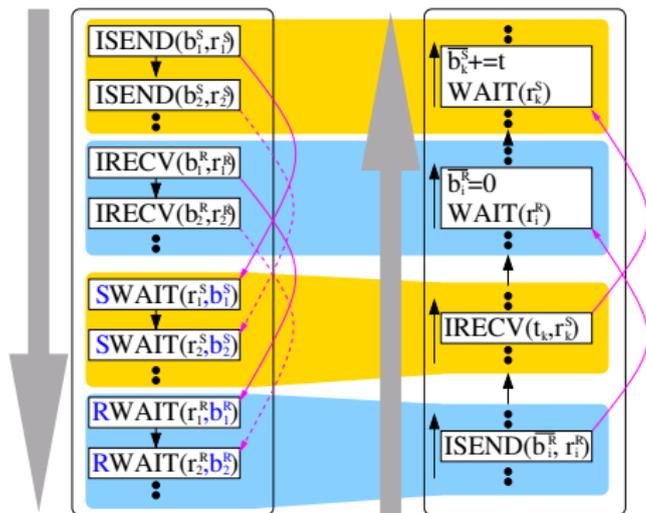


- the above transformations are provably correct
- **extensions** to convey context
⇒ enables a transformation recipe per call
- promises to not **read** or **write** the respective buffer



...solution A

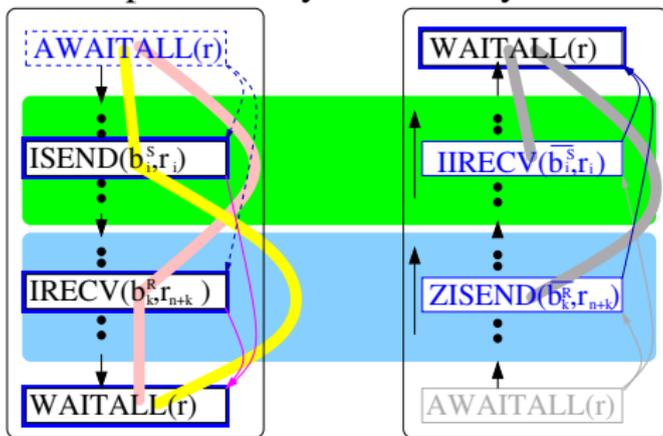
- require individual, **marked** WAITs (achieves 1 on 1 comm edges)



- extra arguments** permit simple transformation recipe with (otherwise) indistinguishable request and buffer arrays
- but individual WAITs impose artificial order \Rightarrow bad for performance

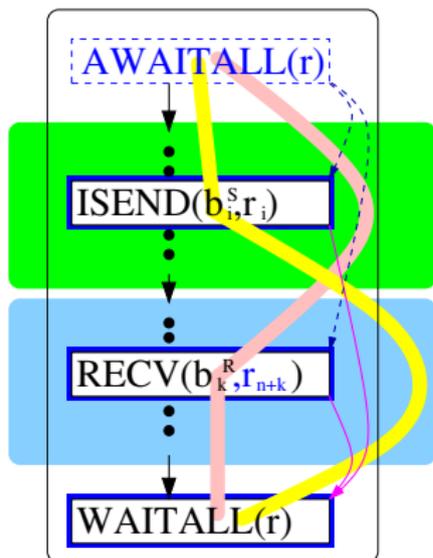
...solution B

- require a symmetric counterpart to the `waitall`; think “*anti_waitall*”
- retains more efficient pattern ☺
- extend promises symmetrically to `AWAITALL`



- wrapped MPI calls have logic to delay zeroing/increment buffers
- goal: “simple” adjoint interpretation for each call

in the OpenAD prototype



```
1  call ampi_awaitall(exchNReqsX(1,bi,bj), &  
2                      exchReqIdX(1,1,bi,bj), &  
3                      mpiStatus, mpiRC)
```

```
1  call ampi_isend(westSendBuf_RL(1,eBl,bi,bj),&  
2                  theSize, theType, theProc, theTag, &  
3                  MPI_COMM_MODEL,&  
4                  exchReqIdX(pReqI,1,bi,bj), &  
5                  exchNReqsX(1,bi,bj),&  
6                  mpiStatus , mpiRc)
```

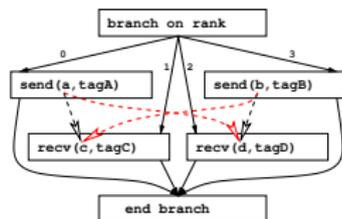
```
1  call ampi_wrecv(westRecvBuf_RL(1,eBl,bi,bj),&  
2                  theSize, theType, theProc, theTag,&  
3                  MPI_COMM_MODEL ,&  
4                  exchReqIdX(pReqI,1,bi,bj), &  
5                  exchNReqsX(1,bi,bj), &  
6                  mpiStatus, mpiRc)
```

```
1  call ampi_waitall(exchNReqsX(1,bi,bj),&  
2                    exchReqIdX(1,1,bi,bj), &  
3                    mpiStatus, mpiRC)
```

what about *<insert problem here>* ?

- solutions A & B disambiguate edges in dynamic comm. graph
- don't do much for static analysis:

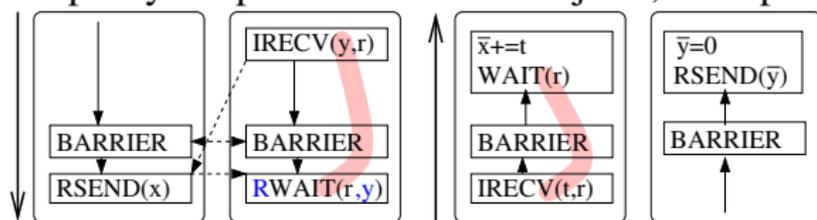
- e.g. modeled with MPI-enhanced CFG (Strout/Hovland/Kreaseck)
- useful for activity analysis



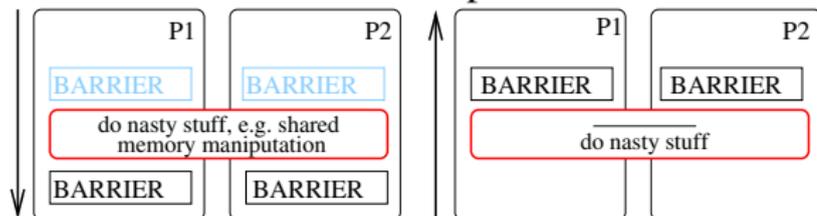
- would like to identify communication “channels” using
 - pragmas (can make up anything but no external support), or
 - aspects (existing systems w support, suggested by B. Gropp; no “Aspecttran”)
- group certain send - recv - wait / certain collective and barrier calls
 - would like runtime validity checks
 - reusable for analysis, debugging, adjoint generation
 - don't assume locality of channels in the source or single call location of collective ops

what about barrier ?

- simple synch point \Rightarrow same for adjoint, example:



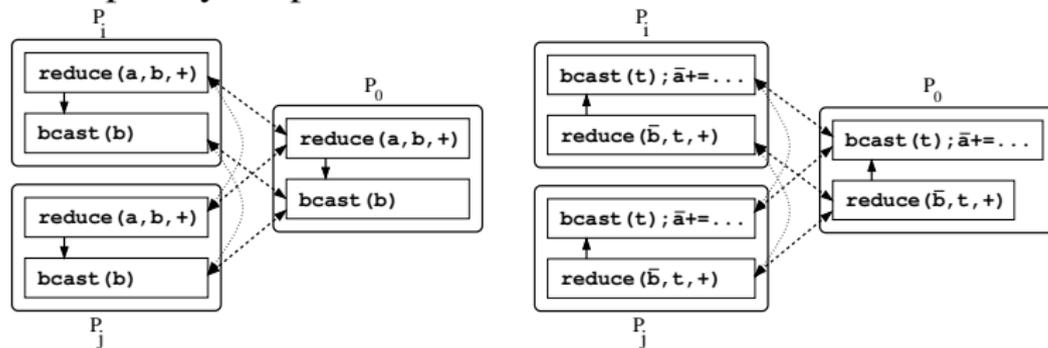
- retains pattern (rsend may improve performance by avoiding hand shake)
- barrier itself does not conceptualize a critical section



- can rationalize the need for barrier enclosed section for correctness of original program
- note - MPI's one-sided commun. has "fence" to demark section ☺

collective communication

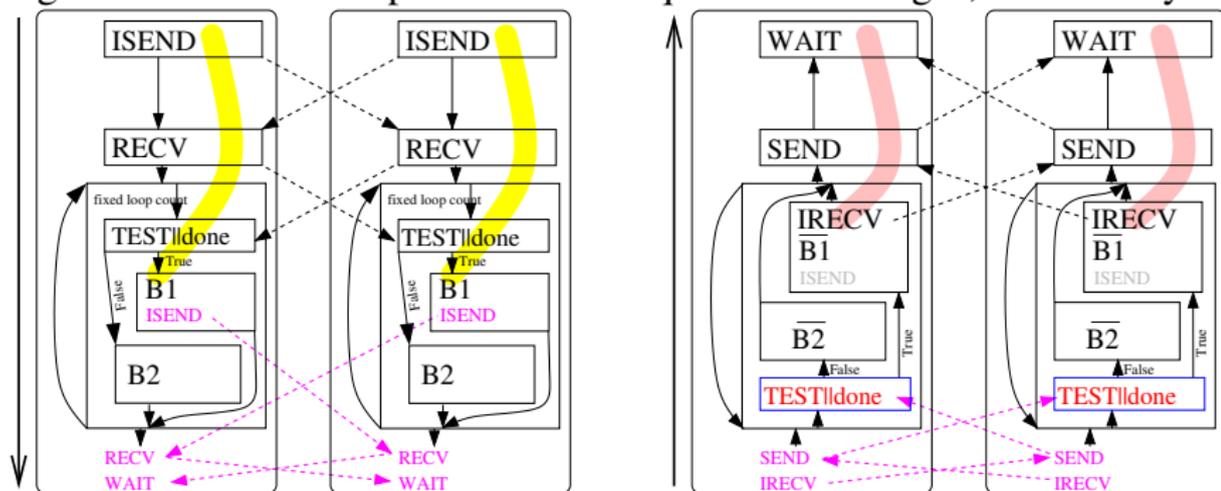
- example: reduction followed by broadcast
 $b_0 = \sum a_i$ followed by $b_i = b_0 \forall i$
- conceptually simple; reduce \mapsto bcast and bcast \mapsto reduce



- adjoint: $t_0 = \sum \bar{b}_i$ followed by $\bar{a}_i += t_0 \forall i$
- has single transformation points (connected by hyper communication edge)
- efficiency for product reduction because of increment $\bar{a}_i += \frac{\partial b_0}{\partial a_i} t_0, \forall i$

attempt to be “plausible”

- e.g. minimize order imposed on two sequential exchanges; B2 is costly



- can retain pattern only if TEST can be moved to subsequent exchange; i.e. TEST needs to be identified with both contexts
- safe (less efficient) fallback is WAIT
- how about `MPI_TEST{ALL|ANY|SOME}`?...

with openMP

work with Michael Förster, Uwe Naumann (Aachen)

with openMP

work with Michael Förster, Uwe Naumann (Aachen)

- has an established set of directives

with openMP

work with Michael Förster, Uwe Naumann (Aachen)

- has an established set of directives
- some known implications for adjoints

with openMP

work with Michael Förster, Uwe Naumann (Aachen)

- has an established set of directives
- some known implications for adjoints
- good entry point for improving data dependency analysis

with openMP

work with Michael Förster, Uwe Naumann (Aachen)

- has an established set of directives
- some known implications for adjoints
- good entry point for improving data dependency analysis

Which directives are important?

Sequential version of dot product

```
1  int main(int argc, char* argv[])
2  {
3      double sum;
4      double x[N], y[N];
5
6      init(x,y);
7      sum = 0.;
8      for(int i=0; i<N; i++)
9          sum = sum + x[i]*y[i];
10     cout << "sum_=" << sum << endl;
11     return 0;
12 }
```

OpenMP version of dot product

```
1  int main(int argc, char* argv[])
2  {
3    double sum;
4    double x[N], y[N];
5
6    init(x,y);
7    sum = 0.;
8    #pragma omp parallel for reduction(+:sum)
9    for(int i=0; i<N; i++)
10       sum = sum + x[i]*y[i];
11    cout << "sum_ = " << sum << endl;
12    return 0;
13 }
```

Supported subset of OpenMP

Subset of OpenMP that should be supported in a first step:

```
1 #pragma omp parallel for private(i, j) firstprivate(v)
   lastprivate(v)
2 for(i=0; i<n; i++) {
3   for(j=0; j<m; j++) { ... }
4 }
```

- All variables listed in clause `private` are defined as a local copy inside of the thread.
- `firstprivate` is as `private` but the variables are being initialized with the value of the global variable.
- `lastprivate` is as `private` but the thread that executes the last iteration copies the local value of `v` into the global variable `v`.

Parallelism in Derivative Code

Given a parallel loop matrix product in the original code:

```
1  #pragma omp parallel for private(j)
2  for(i=0; i<m; i++) {
3      y[i]=0;
4      for(j=0; j<n; j++)
5          y[i]+=A[i][j]*x[j];
6  }
```

Parallelism in Forward Mode

- Data flow stays the same.

```
1 #pragma omp parallel for private(j)
2 for(i=0;i<m;i++) {
3   t1_y[i]=0;
4   y[i]=0;
5   for(j=0;j<n;j++) {
6     t1_y[i]=t1_y[i]*1
7           +t1_A[i][j]*x[j]
8           +t1_x[j]*A[i][j];
9     y[i]+=A[i][j]*x[j];
10  }
11 }
```

Parallelism in Reverse Mode

Adjoint code would look something like:

```
1  #pragma omp parallel for private(j)
2  for(i=0; i<m; i++) {
3      y[i]=0;
4      for(j=0; j<n; j++)
5          y[i]+=A[i][j]*x[j];
6  }
7  // reverse run
8  #pragma omp parallel for private(j)
9  for(i=m-1; i>=0; i--) {
10     for(j=n-1; j>=0; j--) {
11         a1_A[i][j]+=a1_y[i]*x[j];
12         a1_x[j]+=a1_y[i]*A[i][j];
13     }
14     a1_y[i]=0;
15 }
```

Parallelism in Reverse Mode

Adjoint code would look something like:

```
1  #pragma omp parallel for private(j)
2  for(i=0; i<m; i++) {
3      y[i]=0;
4      for(j=0; j<n; j++)
5          y[i]+=A[i][j]*x[j];
6  }
7  // reverse run
8  #pragma omp parallel for private(j)
9  for(i=m-1; i>=0; i--) {
10     for(j=n-1; j>=0; j--) {
11         a1_A[i][j]+=a1_y[i]*x[j];
12         a1_x[j]+=a1_y[i]*A[i][j];
13     }
14     a1_y[i]=0;
15 }
```

Parallelism in Reverse Mode

- Need of attribute grammar that performs an analysis to get the set 'critical' variables (TBD).
- These 'critical' variables have to be handled:
 - Synchronization: Put assignments with critical variable on lhs in critical section.
 - Memory extension: Each thread gets its own private copy of this variable.
 - Only master writes critical variables (synchronization between master/slaves needed)

Under the Hood Handling of OpenMP

Compilers like Rose/gcc do outline the parallel region as indicated here as pseudocode:

```
1 #pragma omp parallel for private(v)
2 for(i=0;i<n;i++)
3   BB;
```

Outlining of the parallel regions:

```
1 void out_omp_1(...) {
2   double v;
3   for(i=my_low;i<my_upper;i++)
4     BB;
5 }
```

The original OpenMP loop is replaced with call to GNU OMP library:

```
GOMP_parallel_start(out_omp_1)
```

OpenMP Clause `private` in FM

- In AD forward mode we have to ensure that the derivative variable for `v` is a local variable as well.
- In case of ADIC there is no change needed since both variables are wrapped in a new data type `DERIV_TYPE`.

```
1 void out_omp_1_adic(...) {
2     DERIV_TYPE v;
3     for (i=my_low; i<my_upper; i++)
4         BB;
5 }
```

OpenMP Clause `firstprivate` in FM

- Only difference to the `private` case is the initialization of `v`.
- Struct assignments are supported in C if same type on both sides.

```
1 void out_omp_1(DERIV_TYPE* global_v) {
2     DERIV_TYPE v=*global_v;
3     for(i=my_low;i<my_upper;i++)
4         BB;
5 }
```

OpenMP Clause `lastprivate` in FM

- Only difference to the `private` case is that data is written to the global variable at the end of the last iteration.

```
1 void out_omp_1(DERIV_TYPE* global_v) {
2     DERIV_TYPE v;
3     for(i=my_low; i<my_upper; i++)
4         BB;
5     if(my_upper==n-1) *global_v=v;
6 }
```

OpenMP Clause `private` in RM

- Each iteration needs its own stack to store the values during forward sweep.

```
1 {
2   #pragma omp parallel for private(v)
3   for(i=0;i<n;i++) {
4     BB1;
5     push[i](v);v=x[i];
6     BB2;
7   }
8   adjoint_out_omp_1(...);
9 }
10 void adjoint_out_omp_1(...){
11   double v; double a1_v=0.;
12   for(i=my_low;i<my_upper;i++) {
13     adjoint(BB2);
14     v=pop[i](v); a1_x[i]+=a1_v; a1_v=0;
15     adjoint(BB1);
16   }
17 }
```

OpenMP Clause `firstprivate` in RM

```
1  {
2  #pragma omp parallel for firstprivate(v)
3  for(i=0;i<n;i++) {
4    // v=global_v;
5    BB1;
6    push[i](v);v=x[i];
7    BB2;
8  }
9  adjoint_out_omp_1(&v);
10 }
11 void adjoint_out_omp_1(double* a1_global_v){
12 double v; double a1_v=0.;
13 for(i=my_low;i<my_upper;i++) {
14   adjoint(BB2);
15   v=pop[i](v); a1_x[i]+=a1_v; a1_v=0;
16   adjoint(BB1);
17   critical{ *a1_global_v+=a1_v };
18 }
19 }
```

OpenMP Clause `lastprivate` in RM

```
1 {
2   #pragma omp parallel for lastprivate(v)
3   for(i=0;i<n;i++) {
4     BB1;
5     push[i](v);v=x[i];
6     BB2;
7     // if(i==n-1){ *global_v=v; }
8   }
9   adjoint_out_omp_1(...);
10 }
11 void adjoint_out_omp_1(double* a1_global_v){
12   double v; double a1_v=0.;
13   for(i=my_low;i<my_upper;i++) {
14     if(my_upper==n-1){ a1_v+=*a1_global_v; }
15     adjoint(BB2);
16     v=pop[i](v); a1_x[i]+=a1_v; a1_v=0;
17     adjoint(BB1);
18   }
19 }
```

AD and Language Features: not-so-structured control flow

- think - goto, exceptions, early return,

AD and Language Features: not-so-structured control flow

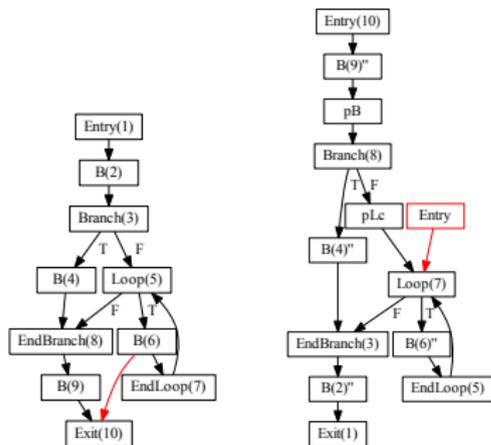
- think - goto, exceptions, early return,
- structured control flow is characterizable by some control flow graph properties; permits **structured reverse control flow!**

AD and Language Features: not-so-structured control flow

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

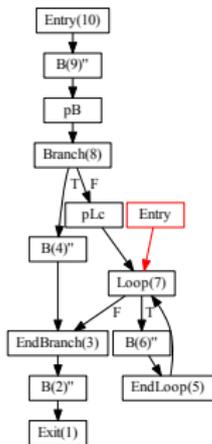
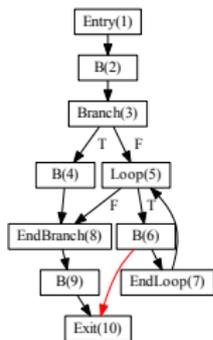
AD and Language Features: not-so-structured control flow

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



AD and Language Features: not-so-structured control flow

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

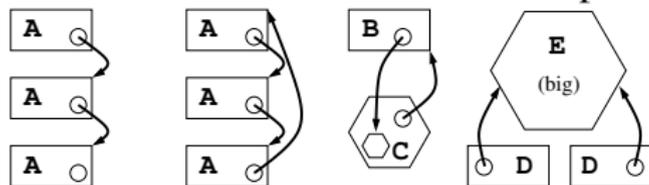


- all is fine without the **red arrow**
- by inspection: adjoint needs alternative entry (or `goto`); but difficult to automate in general
- need to trace more control flow path details
- unstructured control flow is bad for compiler optimization, already for the original model!
- Fortran fallback: trace/replay enumerated basic blocks; for C++: hoist local variables inst.;
- exceptions: `catch` clause needs to completely undo `try` effects

Checkpointing and non-contiguous data

checkpointing = saving program data (to disk)

- “contiguous” data: scalars, arrays (even with stride > 1), strings, structures,...
- “non-contiguous” data: linked lists, rings, structures with pointers,...
- checkpointing is very similar to “serialization”
- Problem: decide when to follow a pointer and save what we point to



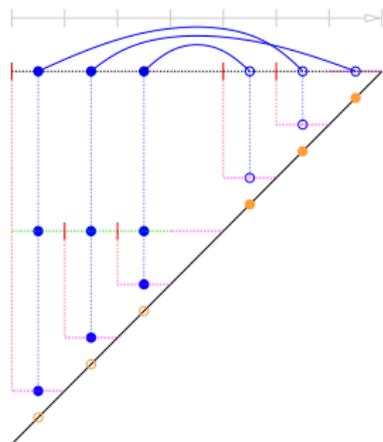
- unless we have extra info this is not decidable at source transformation time
- possible fallback: runtime bookkeeping of things that have been saved (is computationally expensive, cf. `python copy.deepcopy` or `pickle`)

Semantically Ambiguous Data

- e.g. union (or its Fortran counterpart equivalence)
 - data dependence analysis: dependencies propagate from one variable to **all** equivalenced variables
 - “activity” (i.e. the need to generate adjoint code for a variable) leaks to all equivalenced variables whether appropriate or not
 - certain technical problems with the use of an active type (as in OpenAD)
- work-arrays (multiple,0 semantically different fields are put into a (large) work-array); access via index offsets
 - data dependence analysis: there is *array section analysis* but in practice it is often not good enough to reflect the implied semantics
 - the entire work-array may become active / checkpointed
- programming patterns where the analysis has no good way to track the data dependencies:
 - data transfer via files (don't really want to assume all read data depends on all written data)
 - non-structured interfaces: exchanging data that is identified by a “key” but passed as `void*` or something equivalent.

Recomputation from Checkpoints and Program Resources

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



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

options to handle local deallocations

```
1  #include <iostream>
2  #include <cstdlib>
3
4  void foo(int *&p, int* t) {
5      // need adjoint of 'p' to point to (invisible) t1:
6      *t = *p * 2;
7      p = t; // pointer is overwritten
8  }
9
10 void bar(){
11     int *t1, *t2;
12     int *p;
13     t1=(int*)malloc(sizeof(int));
14     t2=(int*)malloc(sizeof(int));
15     *t1=1;
16     p=t1;
17     foo(p,t2);
18     std::cout << *p;
19     free(t1); free(t2);
20 }
21
22 int main(void) {
23     bar();
24     return 0;
25 }
```

- modify model to reuse/grow allocated memory (rather than repeatedly allocate/deallocate), e.g. turn t_1 t_2 into global vars,...
- potential solution for allocate/deallocate **within a checkpointing partition without pointers**: track allocated memory to turn deallocates (line 19) into allocates (of the appropriate size)
- potential (complicated) solution when pointers are involved: associate dynamic allocations in forward sweep to dynamic allocations in the adjoint sweep (adjoint needs to restore pointer overwritten on line 7, but stored pointer *value* references deallocated memory; need abstract association between forward allocate on line 13/14 and adjoint allocate corresponding to the deallocate on line 19)

object-oriented syntactic encapsulation

- syntactic encapsulation of data and methods

object-oriented syntactic encapsulation

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

object-oriented syntactic encapsulation

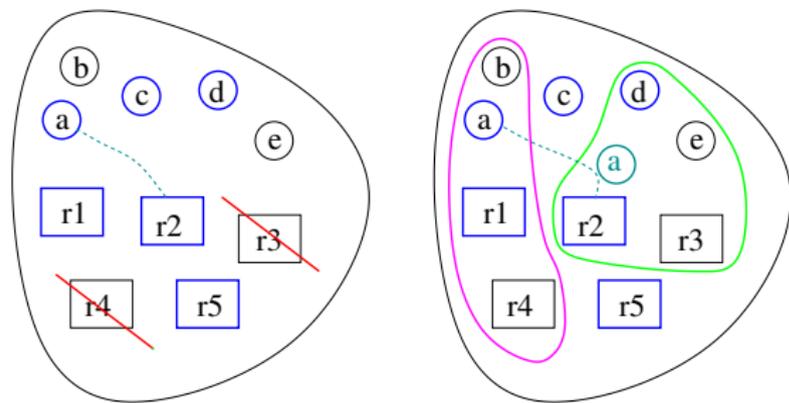
- syntactic encapsulation of data and methods
- Fortran/C recipes recommend extraction of “numerical core”, filtering out init/cleanup/debug code.
- extraction would require (atypical) encapsulation based on control flow

object-oriented syntactic encapsulation

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

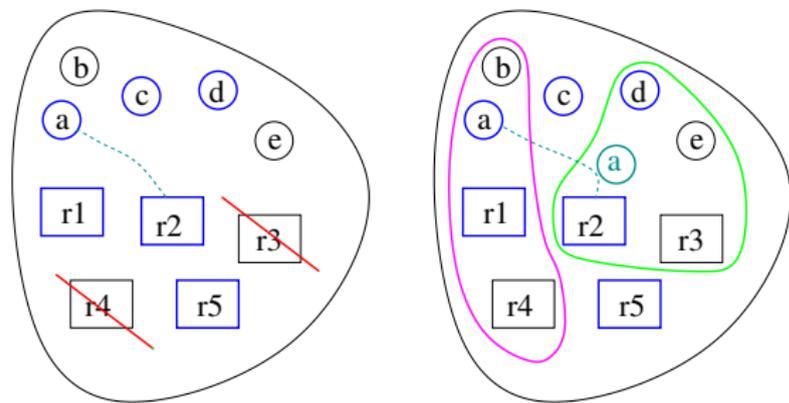
object-oriented syntactic encapsulation

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



object-oriented syntactic encapsulation

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



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

usage concerns

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

usage concerns

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

usage concerns

- availability of AD tools (forward, reverse, efficiency implications)
- restrict tool use to volatile parts?
 - access to the code for all components
 - consider manual adjoints for static parts
 - consider the math (solvers, iterative processes, sparsity, self adjointness, convergence criteria ...); **avoid** differentiating some algorithm portions
- effort for
 - initial implementation
 - **validation**
 - efficiency (generally - what is good for the adjoint is good for the model)
 - implement volatile parts with a domain-specific language (cf. ampl)?
 - **robustness**

usage concerns

- availability of AD tools (forward, reverse, efficiency implications)
- restrict tool use to volatile parts?
 - access to the code for all components
 - consider manual adjoints for static parts
 - consider the math (solvers, iterative processes, sparsity, self adjointness, convergence criteria ...); **avoid** differentiating some algorithm portions
- effort for
 - initial implementation
 - **validation**
 - efficiency (generally - what is good for the adjoint is good for the model)
 - implement volatile parts with a domain-specific language (cf. ampl)?
 - **robustness**
- adjoint robustness and efficiency are impacted by
 - capability for data flow and (structured) control flow reversal
 - code analysis accuracy

usage concerns

- availability of AD tools (forward, reverse, efficiency implications)
- restrict tool use to volatile parts?
 - access to the code for all components
 - consider manual adjoints for static parts
 - consider the math (solvers, iterative processes, sparsity, self adjointness, convergence criteria ...); **avoid** differentiating some algorithm portions
- effort for
 - initial implementation
 - **validation**
 - efficiency (generally - what is good for the adjoint is good for the model)
 - implement volatile parts with a domain-specific language (cf. ampl)?
 - **robustness**
- adjoint robustness and efficiency are impacted by
 - capability for data flow and (structured) control flow reversal
 - code analysis accuracy
 - use of certain programming language features
 - use of certain inherently difficult to handle patterns

usage concerns

- availability of AD tools (forward, reverse, efficiency implications)
- restrict tool use to volatile parts?
 - access to the code for all components
 - consider manual adjoints for static parts
 - consider the math (solvers, iterative processes, sparsity, self adjointness, convergence criteria ...); **avoid** differentiating some algorithm portions
- effort for
 - initial implementation
 - **validation**
 - efficiency (generally - what is good for the adjoint is good for the model)
 - implement volatile parts with a domain-specific language (cf. ampl)?
 - **robustness**
- adjoint robustness and efficiency are impacted by
 - capability for data flow and (structured) control flow reversal
 - code analysis accuracy
 - use of certain programming language features
 - use of certain inherently difficult to handle patterns
 - **smoothness of the model, utility of the cost function**

is the model smooth?

- $y = \text{abs}(x)$; gives a kink

is the model smooth?

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

is the model smooth?

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

is the model smooth?

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

is the model smooth?

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

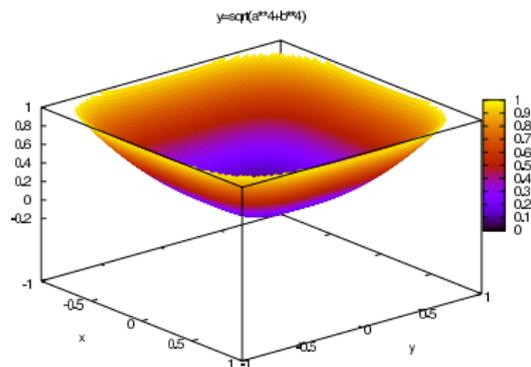
## is the model smooth?

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

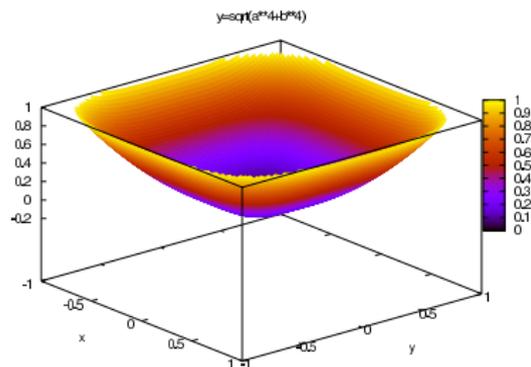
is the model smooth?

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



## is the model smooth?

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

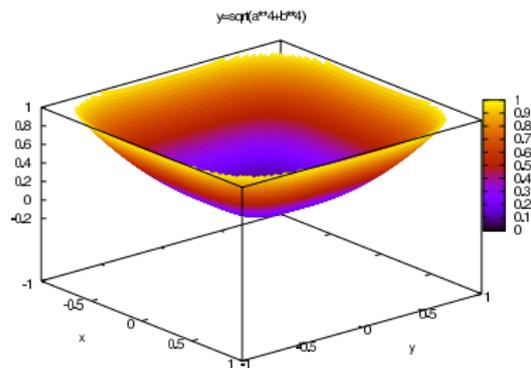


AD does not perform algebraic simplification,

i.e. for $a, b \rightarrow 0$ it does $\left(\frac{d\sqrt{t}}{dt}\right) \stackrel{t \rightarrow +0}{=} +\infty$.

is the model smooth?

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



**AD does not perform algebraic simplification,**

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

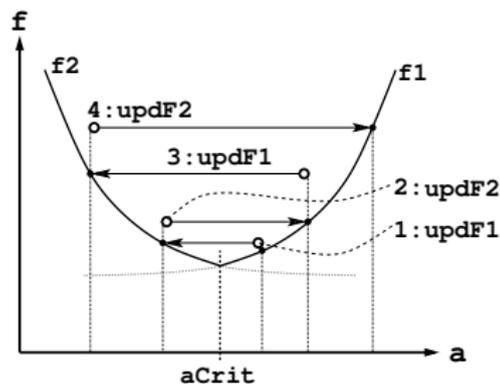
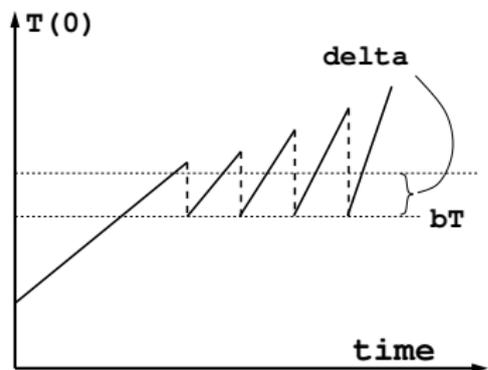
AD computes derivatives of programs(!)

know your application e.g. fix point iteration, self adjoint, step size computation, convergence criteria

# nonsmooth models

observed:

- INF, NaN
- oscillating derivatives (may be glossed over by FD) or derivatives growing out of bounds



## nonsmooth models II

- blame AD tool - verification problem

## nonsmooth models II

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

## nonsmooth models II

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

## nonsmooth models II

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

## nonsmooth models II

- blame AD tool - verification problem
  - forward vs reverse (dot product check)
  - compare to FD
  - compare to other AD tool
- blame code, model's built-in numerical approximations, external optimization scheme or inherent in the physics?

## nonsmooth models II

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

## nonsmooth models II

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

## nonsmooth models II

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

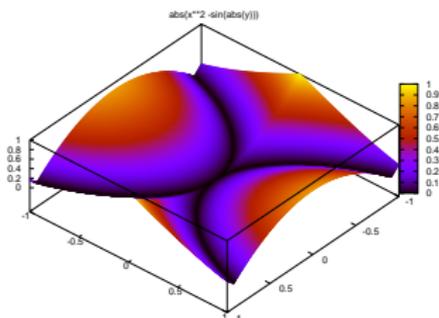
## nonsmooth models II

- blame AD tool - verification problem
  - forward vs reverse (dot product check)
  - compare to FD
  - compare to other AD tool
- blame code, model's built-in numerical approximations, external optimization scheme or inherent in the physics?
- higher order models in mech. engineering, beam physics, AtomFT  
explicit g-stop facility for ODEs, DAEs
- what to do about first order
  - Adifor: optionally catches intrinsic problems via exception handling
  - Adol-C: tape verification and intrinsic handling

## nonsmooth models II

- blame AD tool - verification problem
  - forward vs reverse (dot product check)
  - compare to FD
  - compare to other AD tool
- blame code, model's built-in numerical approximations, external optimization scheme or inherent in the physics?
- higher order models in mech. engineering, beam physics, AtomFT  
explicit g-stop facility for ODEs, DAEs
- what to do about first order
  - Adifor: optionally catches intrinsic problems via exception handling
  - Adol-C: tape verification and intrinsic handling
  - OpenAD (comparative tracing)

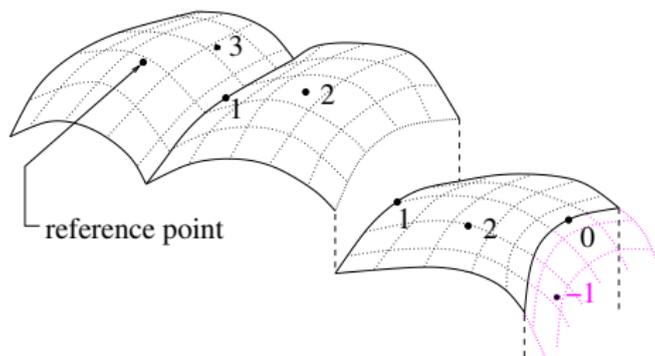
# differentiability



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

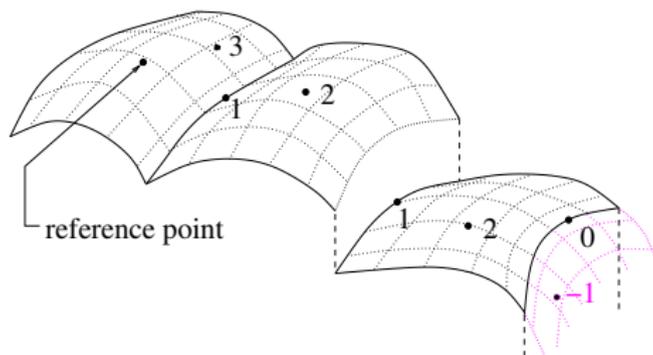
- Gâteaux: if  $\exists \quad df(x, \dot{x}) = \lim_{\tau \rightarrow 0} \frac{f(x + \tau \dot{x}) - f(x)}{\tau}$  for all directions  $\dot{x}$
- Bouligand: Lipschitz continuous and Gâteaux
- Fréchet:  $df(., \dot{x})$  continuous for every fixed  $\dot{x}$  ... not generally
- in practice: often benign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

# case distinction



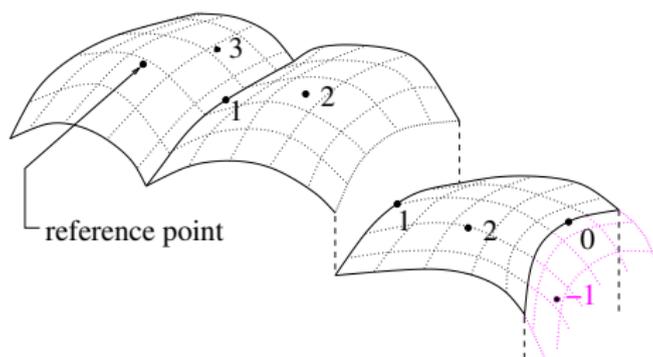
# case distinction

## 3 locally analytic



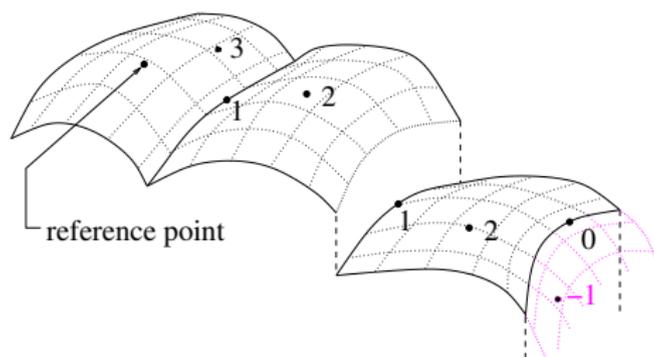
## case distinction

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



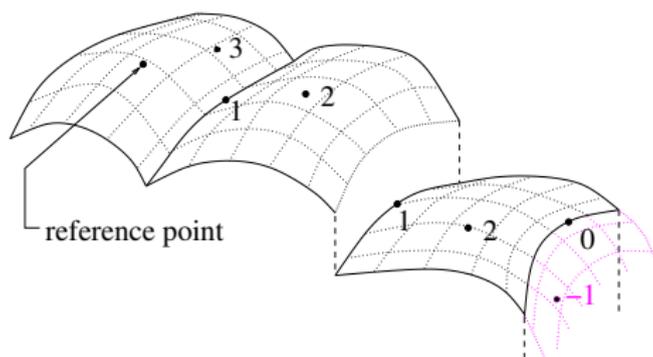
## case distinction

- 3 locally analytic
- 2 locally analytic but crossed a (potential) kink ( $\min, \max, \text{abs}, \dots$ ) or discontinuity ( $\text{ceil}, \dots$ ) [ for source transformation: also different control flow ]
- 1 we are exactly at a (potential) kink, discontinuity



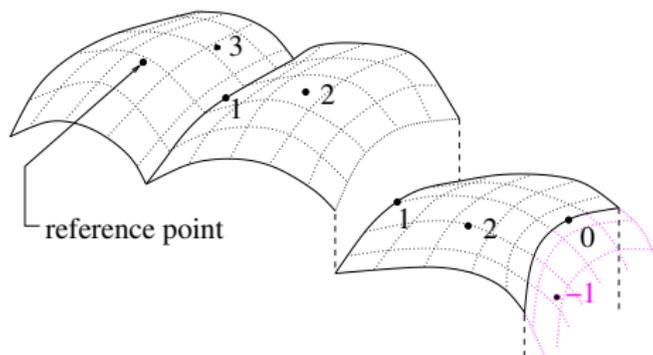
## case distinction

- 3 locally analytic
- 2 locally analytic but crossed a (potential) kink ( $\min, \max, \text{abs}, \dots$ ) or discontinuity ( $\text{ceil}, \dots$ ) [ 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 only be determined for some special cases)



## case distinction

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



# Summary

- basics of AD are deceptively simple

# Summary

- basics of AD are deceptively simple
- AD can handle common parallel idioms (as a proof of concept)

# Summary

- basics of AD are deceptively simple
- AD can handle common parallel idioms (as a proof of concept)
- there are inherent problems with certain language features

# Summary

- basics of AD are deceptively simple
- AD can handle common parallel idioms (as a proof of concept)
- there are inherent problems with certain language features
- avoiding some of them helps AD and is good for compiler optimization

# Summary

- basics of AD are deceptively simple
- AD can handle common parallel idioms (as a proof of concept)
- there are inherent problems with certain language features
- avoiding some of them helps AD and is good for compiler optimization
- details in the code have a large impact on AD adjoint efficiency

# Summary

- basics of AD are deceptively simple
- AD can handle common parallel idioms (as a proof of concept)
- there are inherent problems with certain language features
- avoiding some of them helps AD and is good for compiler optimization
- details in the code have a large impact on AD adjoint efficiency
- need new recipes/approaches for AD at a higher level

# Summary

- basics of AD are deceptively simple
- AD can handle common parallel idioms (as a proof of concept)
- there are inherent problems with certain language features
- avoiding some of them helps AD and is good for compiler optimization
- details in the code have a large impact on AD adjoint efficiency
- need new recipes/approaches for AD at a higher level

need to distill some new recipes from manual adjoints...

# Summary

- basics of AD are deceptively simple
- AD can handle common parallel idioms (as a proof of concept)
- there are inherent problems with certain language features
- avoiding some of them helps AD and is good for compiler optimization
- details in the code have a large impact on AD adjoint efficiency
- need new recipes/approaches for AD at a higher level

need to distill some new recipes from manual adjoints...

... for AD 2012: 6th International Conference on AD

July 23-27 2012, Fort Collins, CO: [www.autodiff.org/ad12](http://www.autodiff.org/ad12)