AD for Scale

Jean Utke / Paul Hovland

- motivation
- basic ideas
- OpenAD/F information
- strategies and concerns
- outlook

The case for source transformation AD

- the major advantages of AD are ... no need to repeat again
- source transformation AD
  - complexity of the tools (vs. operator overloading) ☹
  - efficiency gains ☺
- efficiency gains from source transformation AD come from
  - activity analysis
  - optimizing combinatorial problems at compile time
  - for reverse mode: high-level structural allows explicit control flow reversal
- forward mode source transformation considerably less complicated than reverse mode source transformation

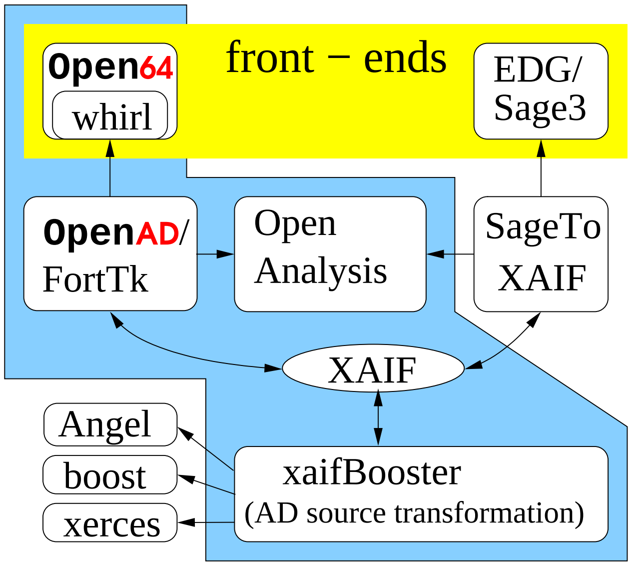
What is relevant for SCALE?

The model source code impacts AD capabilities

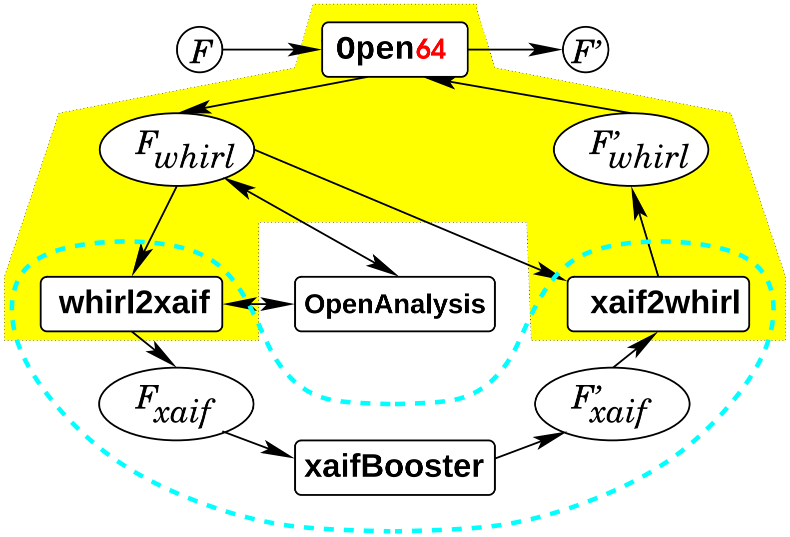
- Is activity analysis likely to help?
  e.g. want derivatives for subset of model data & routines
- if no and only forward mode ⇒ consider operator overloading facilitated by a global type change (btw - this already implies a bit of source transformation, see NEOS ☺)
- if yes:
  - activity analysis based on data flow analysis -
    supply the (entire) model source code (can have stubs)
  - split sources to filter out ancillary logic (monitoring, debugging, timing, I/O) to reduce conservative overestimate
  - semantically ambiguous data (union, equivalence)
    ⇒ overestimated active set
  - integrate the AD tool chain into the build process
- Will I need reverse mode, e.g. for gradients? If yes - avoid unstructured control flow and some data access patterns (e.g. linked lists), etc.

OpenAD General

- www.mcs.anl.gov/OpenAD
- forward and **reverse**
- currently first order
- source transformation
- large problems
- Fortran(90) side of a multi language design
- under development



Fortran pipeline:



OpenAD example

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
  y=sin(x*x)
end subroutine
```

result of pushing it through the pipeline →

```
SUBROUTINE head(X, Y)
use w2f__types
use active_module
IMPLICIT NONE
REAL(w2f__8) OpenAD_Symbol_0
...
REAL(w2f__8) OpenAD_Symbol_5
type(active) :: X
INTENT(IN)  X
type(active) :: Y
INTENT(OUT)  Y
OpenAD_Symbol_0 = (X%v*X%v)
Y%v = SIN(OpenAD_Symbol_0)
OpenAD_Symbol_2 = X%v
OpenAD_Symbol_3 = X%v
OpenAD_Symbol_1 = COS(OpenAD_Symbol_0)
OpenAD_Symbol_5 = ((OpenAD_Symbol_3 +
   OpenAD_Symbol_2) * OpenAD_Symbol_1)
CALL sax(OpenAD_Symbol_5,X,Y)
RETURN
END SUBROUTINE
```

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

on the website

www.mcs.anl.gov/openad

- more examples
- instructions to download & build
- source code documentation
- revision history
- bibliography
- wiki
- bug tracker

active type

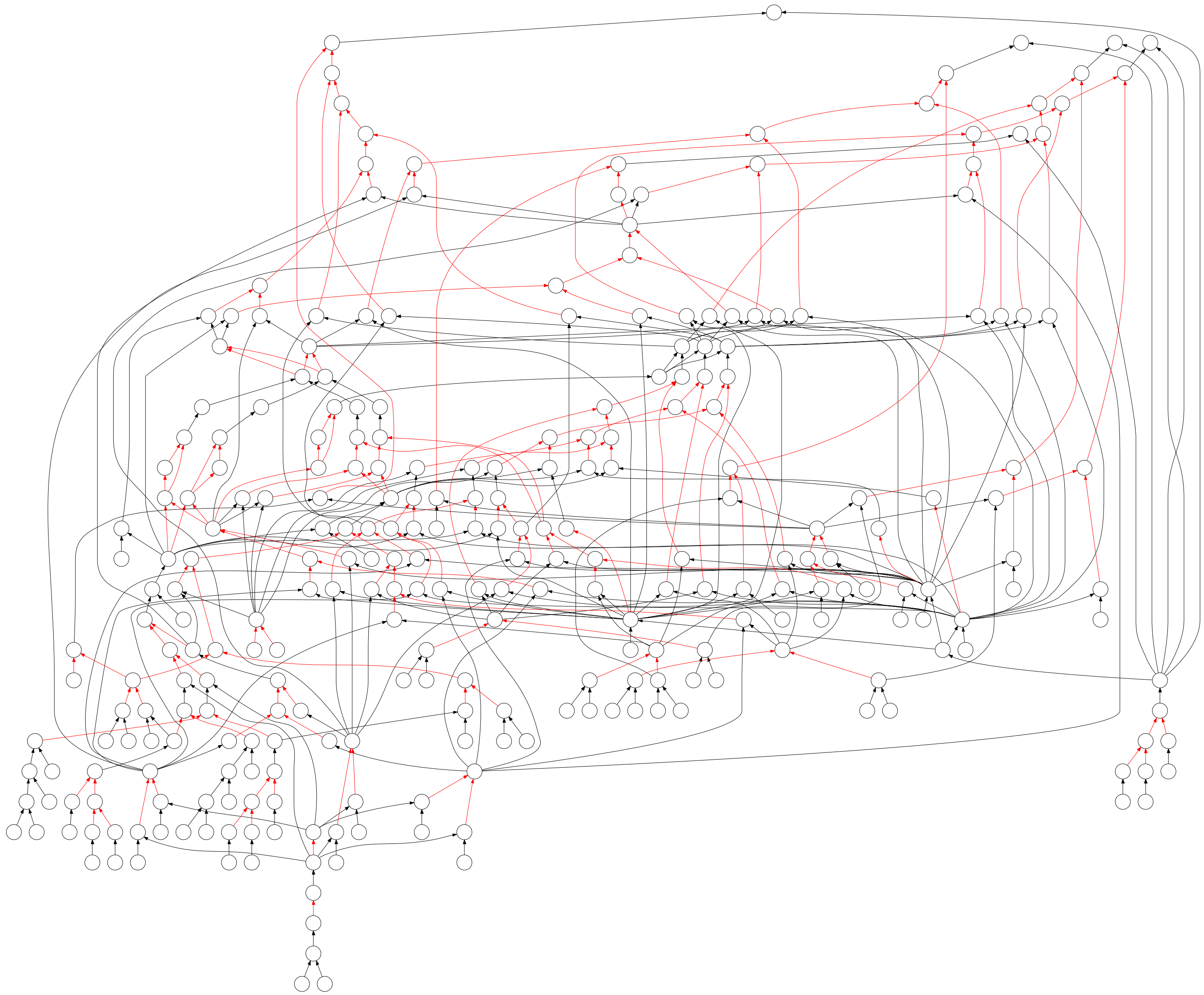
by address (active type):

- XAIF has discriminator flag (original vs. augmenting)
- XAIF does not need to know about user def'd types
- unparsing according to discriminator
- type in runtime library, not part of FE, except for member names
- readily supported "everywhere" except F77.
- impacts i/o and memory management! (netcdf and fotran i/o)

by name (shadowing variables):

- used by all F77 tools (no user def'd types)
- original data retains size, leaving memory allocation schemes and i/o formats undisturbed
- augmenting data can be allocated and managed independently from the original data
- in a language with user def'd types (requires XAIF to know user-def'd types):
  - All active variables have to be shadowed.
  - All subroutine signatures need to be expanded to contain the shadowing variables.
  - user defined types containing shadowed variables have to have shadow types (recursively) to maintain data separation.
  - Variables of shadowed types have to be shadowed.
  - Variables pointing to shadowed variables have to be shadowed (recursively) to properly replicate pointer arithmetic.

computational graphs in OpenAD



observations for CENTRM

- top level routine `CALC`
- identify independents (`xmd`) and dependents (`pxj`)
- filter out source files with code not called under `CALC`
  - excludes 58 of 148 files (+121 interface files)
  - e.g. the AD driver logic in the code calling CALC
- references files from `scaleLib`; mostly treated as black-box routines (except 10 files + 9 stubs)
- CALC allocates/deallocates dynamic memory (for reverse?)
- handling of `read_scratch()` and `write_scratch()` e.g. via wrappers
- processed files need to be ordered (currently fixed based on `make` output)

observations for PMC

- revealed an Open64 front-end bug, now fixed
- top level routine `process`
- independents `pnt_flx` initialized by `read_transfer_parameters()`
- transfer of derivatives from CENTRM in `flxrec.f90`
- dependents (`grp_xs_new` and `grp_xs_2`(?) see `xscal.f90`)
- filter out 9 of 30 source files with code not called under `process`
- include 9 file from `scalelib`
- processed files need to be ordered (currently fixed based on `make` output)

suggestions for source code

- make source separation easy (for the build process)
  - one method per file or file contents aligned with separation
  - extract setup (initialization, allocation, ...) and cleanup (deallocation, result output) logic from computation under CALC
  - factor out low level I/O
  - for modules - separate data (module variables) and interfaces from implementation (if impossible use stubs)
- avoid equivalence
- avoid gratuitous use of pointers
- avoid gratuitous local memory (de)allocation (e.g. in `pxarr` for `pei`).

Language Coverage

- array operations
- TRANS SUM DMIN AIMAG ALOG (now added)
- complex arithmetic & intrinsics in `bn, fabcz, qol, qratio, trisol`
- function to subroutine canonicalization
- except special functions with closed form partials (e.g. `ki3, e3`)
- question if `ki3` should be differentiated (doesn't appear to be covered by GRESS)
- question if the GRESS generated `e3g` is or should be called
- files reads with implied do loops, found in `epitoth`

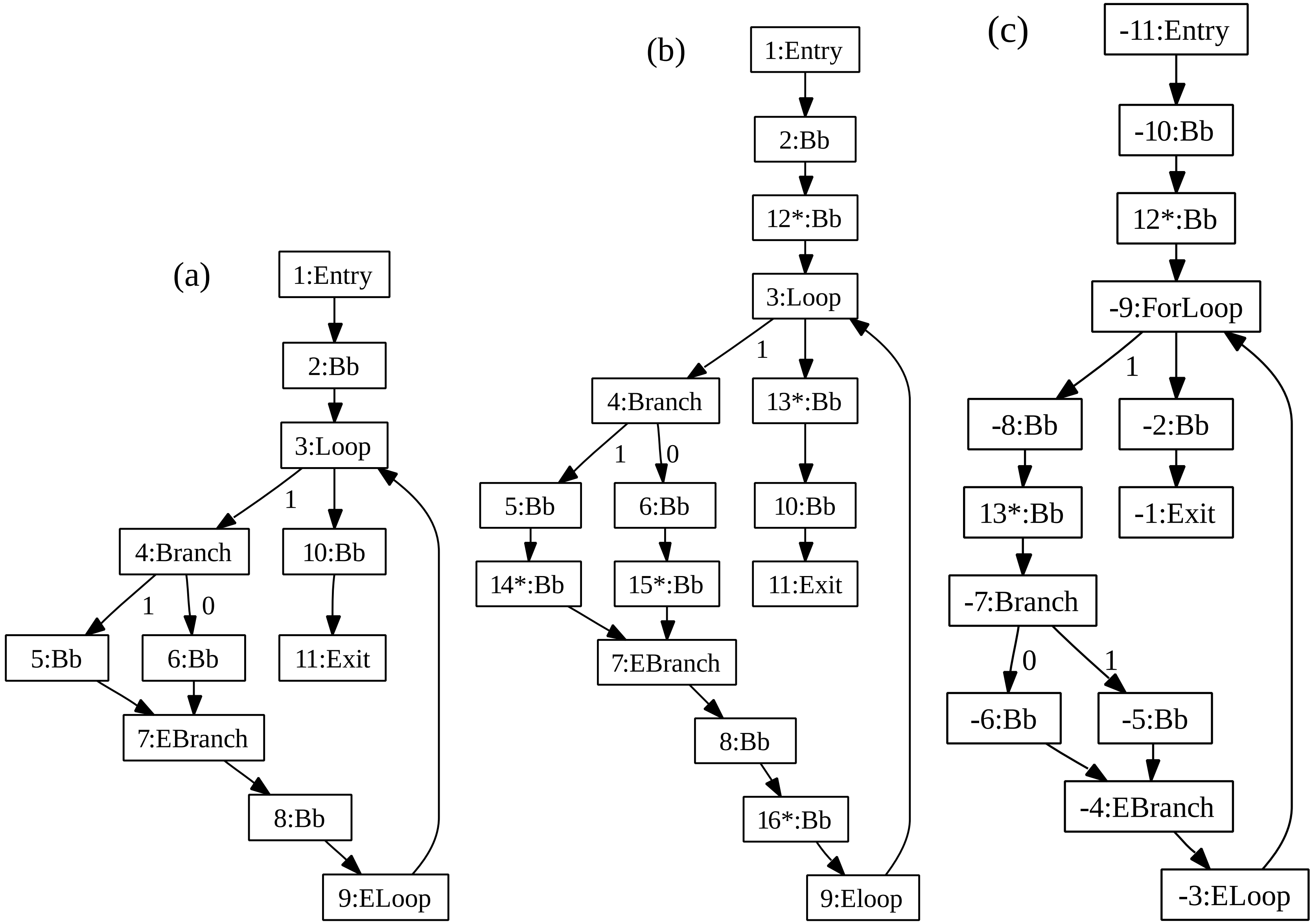
configurable sources and AD transformation

- often AD tool part of the build process
- ok for precompiled distribution
- not ok with configurable sources (e.g. preprocessor) because AD transformation is done per configuration
- front-end even performs constant folding for `PARAMETER` quantities

Further Information

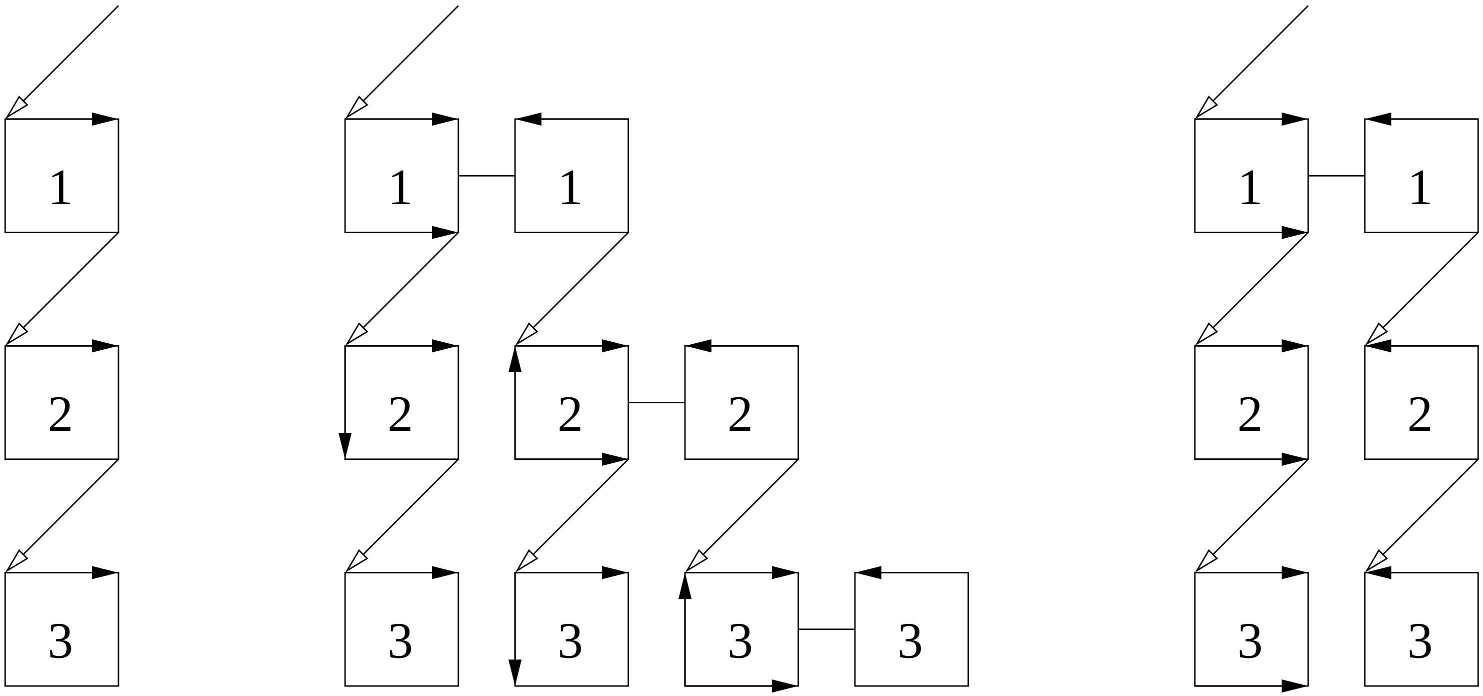
- A. Griewank, *Evaluating Derivatives*, SIAM, 2000.
- A. Griewank, *On Automatic Differentiation*; this and other technical reports available online at: `http://www.mcs.anl.gov/autodiff/tech_reports.html`
- AD in general: `http://www.autodiff.org/` ADIFOR: http://www.mcs.anl.gov/adifor/ ADIC: http://www.mcs.anl.gov/adic/ OpenAD: http://www.mcs.anl.gov/openad/ Other tools: http://www.autodiff.org/

control flow reversal



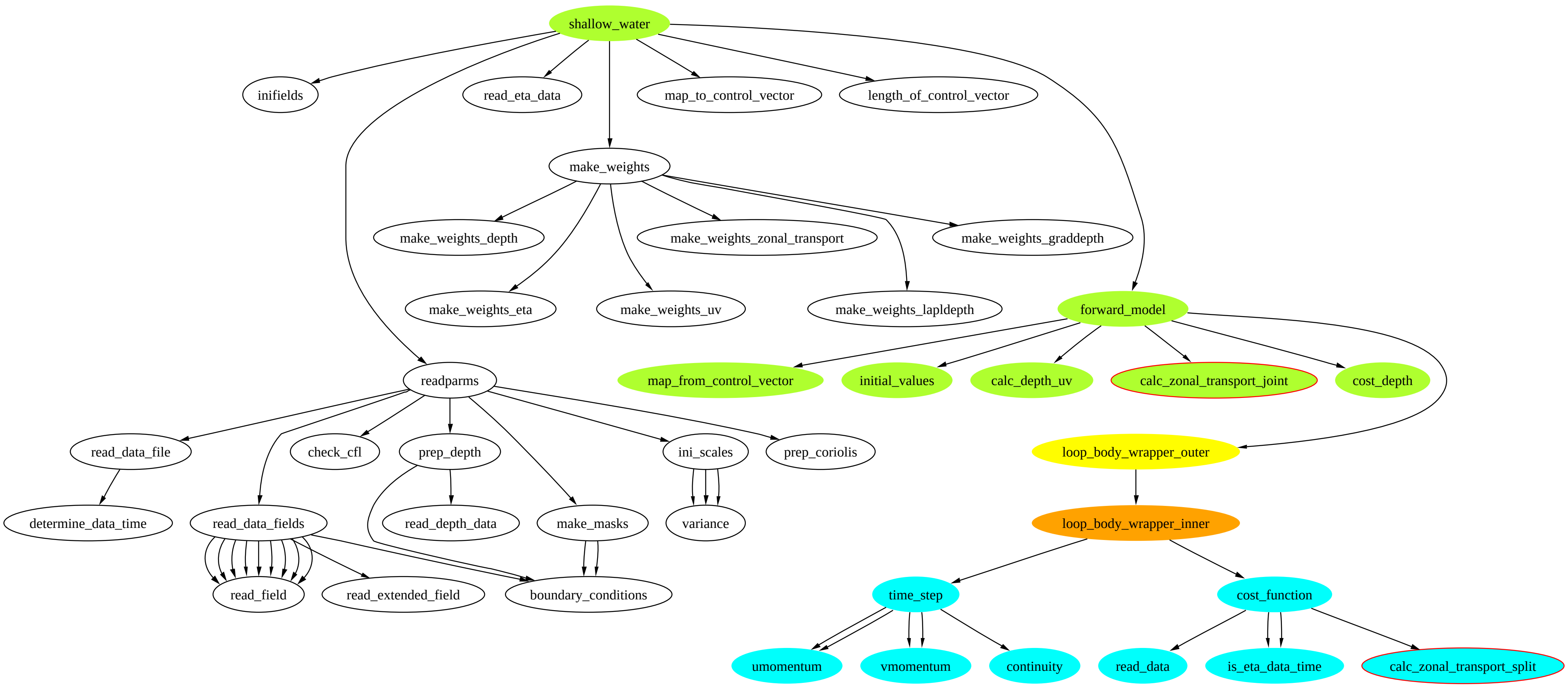
OpenAD reversal modes

subroutine level granularity



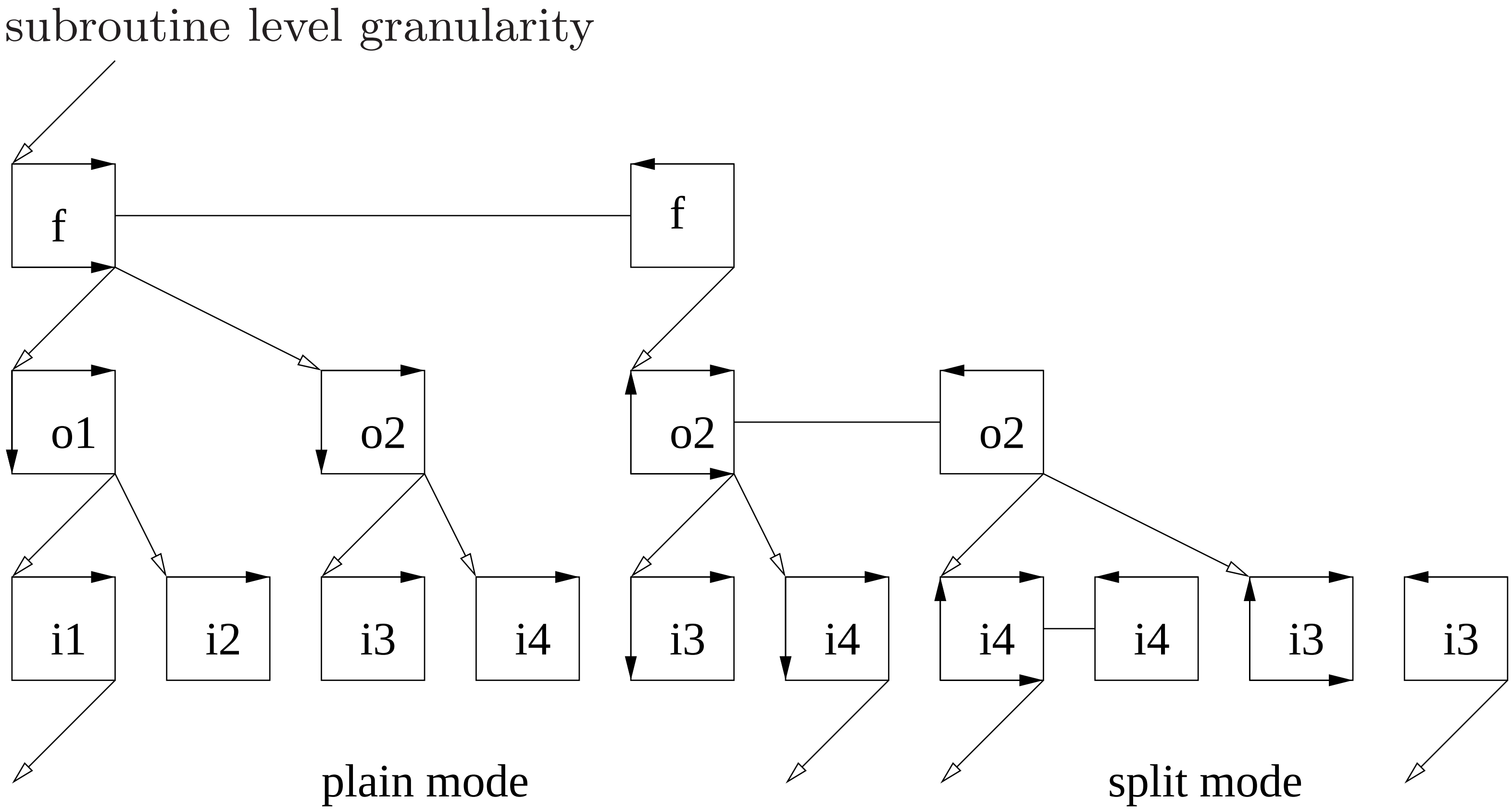
call tree joint mode call tree split mode call tree

ADified Shallow Water Call Graph



- `calc_zonal_transport` is split
- nested loop checkpointing in outer and inner loop body wrapper
- inner loop body in split mode

OpenAD reversal modes with checkpointing



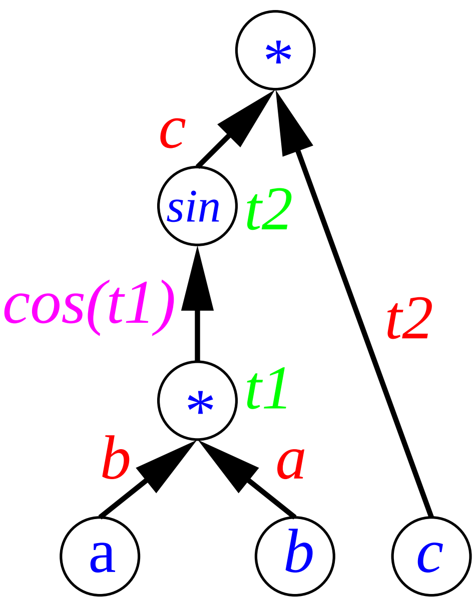
summary OpenAD features

- elimination techniques
  - vertex, edge, face
  - various heuristics
  - DAG per statement or basic block
- anonymous control flow graph reversal, “simple” loop designation
- flexibility & reversal schemes via templates/inlining
- constant folding
- OpenAnalysis integration

example - how do directional derivatives come about?

$f : y = sin(a * b) * c$

yields a graph representing the order of computation:



- intrinsics $\phi(\ldots, w, \ldots)$ have local partial derivatives $\frac{\partial \phi}{\partial w}$
- e.g. `sin(t1)` yields `cos(t1)`
- *code list* $\rightarrow$ intermediate values $t1$ and $t2$
- all others already stored in variables

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

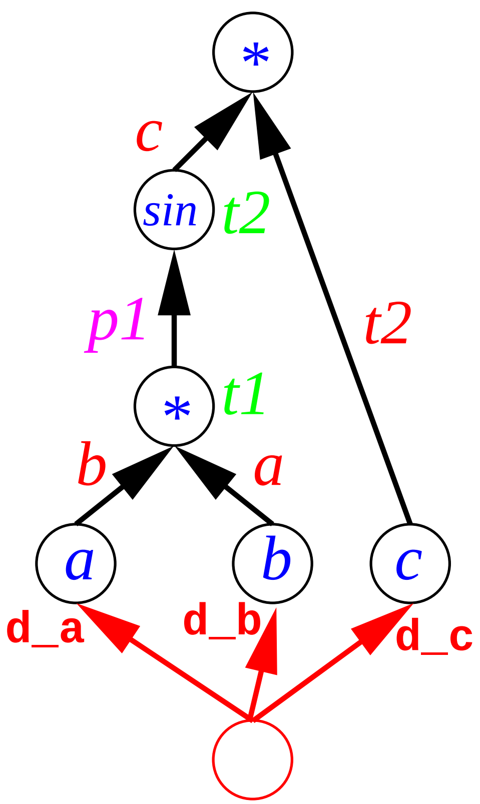
What can we do with this?

forward with directional derivatives

$f(g(x)) \Rightarrow \dot{f}(g(x))\dot{g}(x)\dot{x}$ multiplications along paths

Assume a point (a_0, b_0, c_0) and a direction $(\dot{a}, \dot{b}, \dot{c})$ =`(d_a,d_b,d_c)`

variable and directional derivatives associated in pairs `(v,d_v)`:



`d_a*b*p1*c+d_b*a*p1*c+d_c*t2`

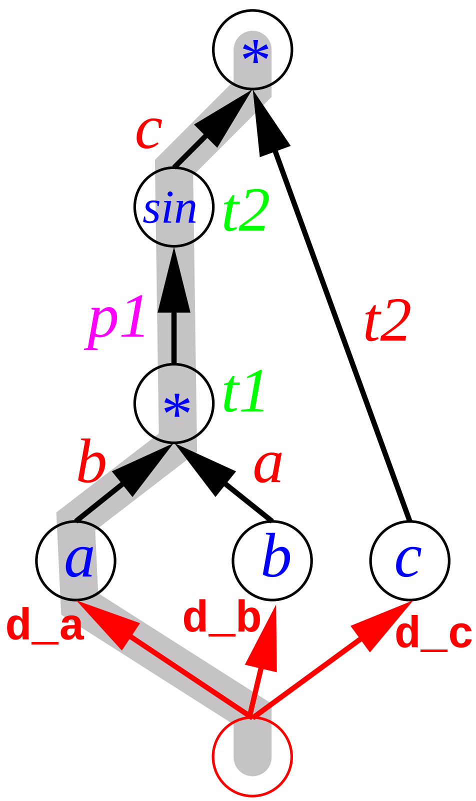
has common subexpressions

interleave computations of directional derivatives

```
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`?

forward with directional derivatives II



- if $(\dot{a},\dot{b},\dot{c})=(1,0,0)$ then d_y$=\frac{\partial f}{\partial a}(a_0,b_0,c_0)$

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

- 3 directions give $\nabla f(a_0,b_0,c_0)$ and d_y$=\nabla f^T(\dot{a},\dot{b},\dot{c})=\nabla f^T\dot{x}$
- floating point accuracy for derivative calculation !
- gradient calculation cost $\sim n$

Tangent-linear Models

The tangent-linear model of

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

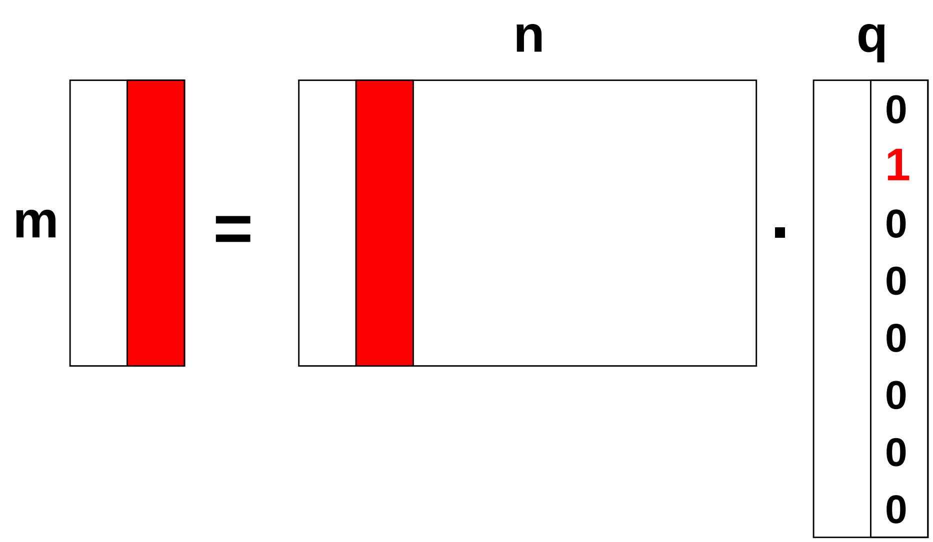
is

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

Jacobian matrix

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

column by column at $O(n)$.

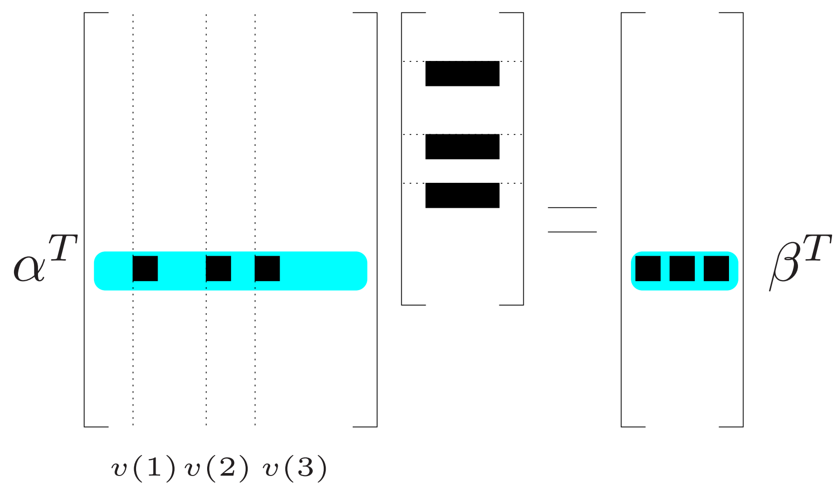


sparse Jacobians

many repeated Jacobian vector products → compress the Jacobian
$F' \cdot S = B \in \mathbb{R}^{m\times q}$ using a seed matrix $S \in \mathbb{R}^{n\times q}$
What are S and q?
Row i in F' has ρ_i nonzeros in columns $v(1), \ldots, v(\rho_i)$
$F'_i = (\alpha_1, \ldots, \alpha_{\rho_i}) = \alpha^T$ and the compressed row is $B_i = (\beta_1, \ldots, \beta_q) = \beta^T$ We choose S so we can solve:

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

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

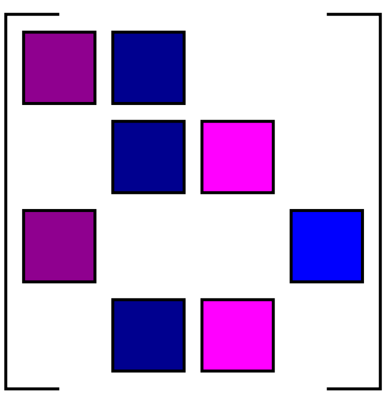


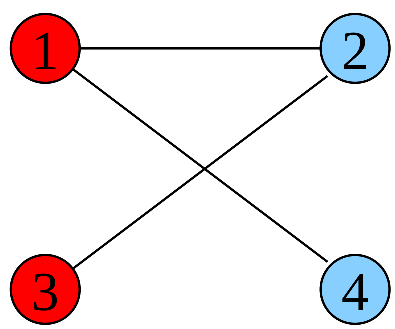
determining q, S (1)

direct:

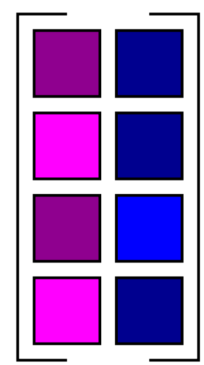
- Curtis/Powell/Reid: structurally orthogonal
- Coleman/Moré: column incidence graph coloring)

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.





$$S = \begin{bmatrix} \color{red}{1} & 0 \\ 0 & \color{blue}{1} \\ \color{red}{1} & 0 \\ 0 & \color{blue}{1} \end{bmatrix}$$



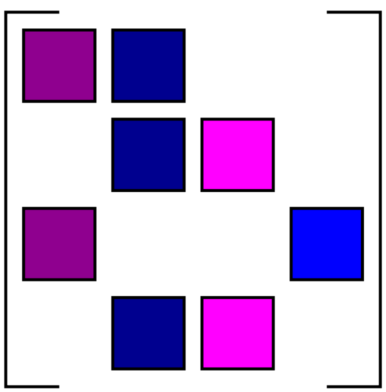
reconstruct F' by relocating nonzero elements (direct)

determining q, S (2)

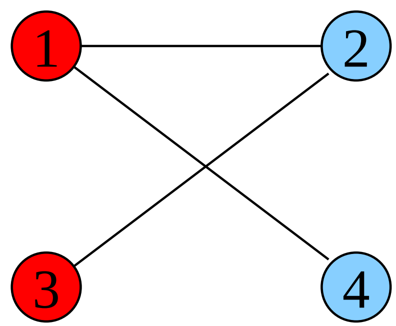
indirect:

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

same example



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



$$S = \begin{bmatrix} \lambda_1^0 & \lambda_1^1 \\ \lambda_2^0 & \lambda_2^1 \\ \lambda_1^0 & \lambda_1^1 \\ \lambda_2^0 & \lambda_2^1 \end{bmatrix}$$

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

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

tool support (1)

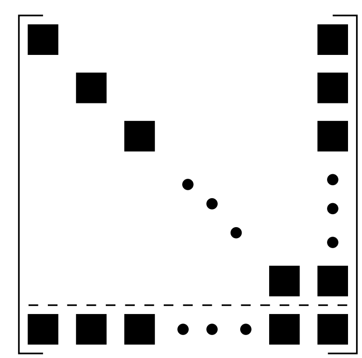
all tools: seeding & vector mode (forward)

Adifor:

- SparsLinC library
- pattern detection
- sparse forward propagation

Adol-C:

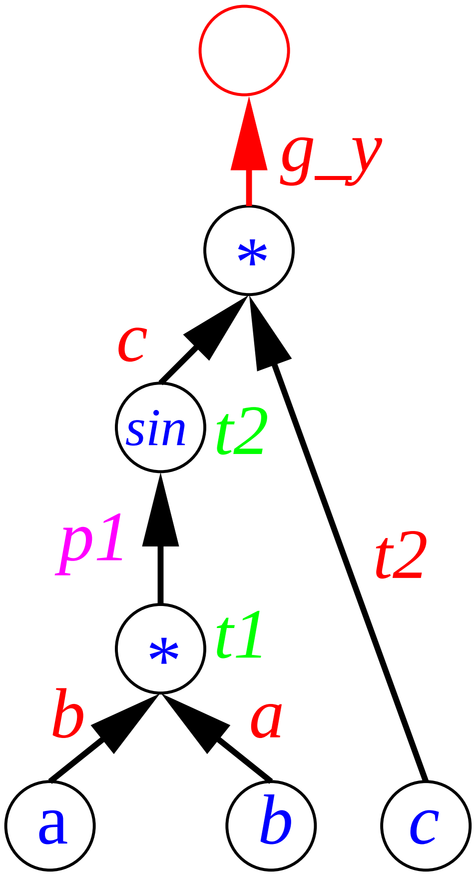
- pattern detection via bitmap propagation
- (dense) forward propagation



What about ....

reverse with adjoints

Assume variable and adjoints associated in pairs (`v,g_v`):



append computations of adjoints

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y  = t2*c
g_c  = g_y*t2
g_t2 = g_y*c
g_t1 = g_t2*p1
g_b  = g_t1*a
g_a  = g_t1*b
```

What is in (`g_a,g_b,g_c`)? If `g_y`=1, then $\nabla f(a_0, b_0, c_0)$

Adjoint Models

The adjoint model of

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

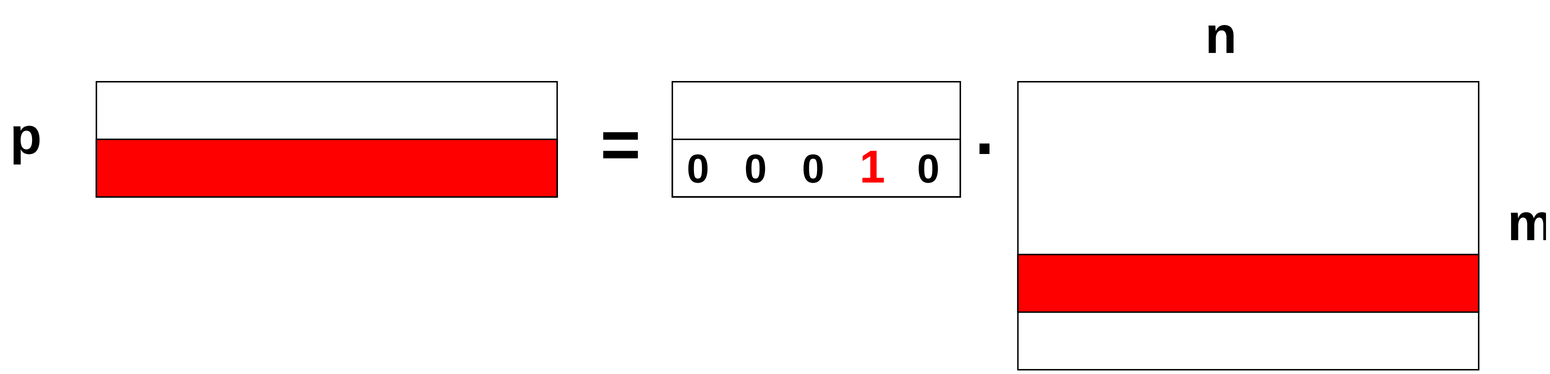
is

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

Jacobian matrix

$$F' = \left(\frac{\partial y_j}{\partial x_i}\right)_{i=1,\ldots,n}^{j=1,\ldots,m} = (F')^T \cdot I_m$$

row by row at $O(m)$ (cheap gradients ☺, tape intermediates / partials ☹)

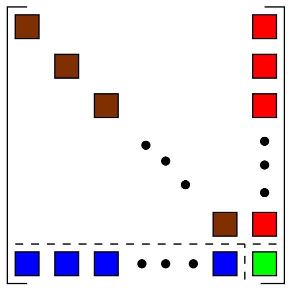


sparse Jacobians (2)

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 $p = 1$

$$F' \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ \vdots & \vdots \\ 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} \blacksquare & \blacksquare \\ \blacksquare & \blacksquare \\ \vdots & \vdots \\ \blacksquare & \blacksquare \\ \blacksquare & \sum\blacksquare \end{bmatrix} \quad \text{and} \quad F'^T \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \blacksquare \\ \blacksquare \\ \vdots \\ \blacksquare \\ \blacksquare \end{bmatrix}$$

tool support (2)

row compression / partitioning require reverse mode!
OpenAD/Tapenade/Adifor (v3.0):

- reverse mode

Adol-C:

- dependency propagation
- dynamic dependency kind estimation (none, linear, polynomial, rational, transcendental, non-smooth)

We care, e.g. because of partial separability!

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

$$f(\mathbf{x}) = \sum_i a_i f_i(\mathbf{x}_i) \quad \text{where} \quad \mathbf{x}_i \subseteq \mathbf{x} \quad \text{so that} \quad \nabla f_i \in \mathbb{R}^{n_i}, n_i << n$$

higher order

sparse tool support: (Adifor: hessian module) Adol-C:

- `hessian` driver: n Hessian-vector products (one reverse after one forward each)
- `hessian2` driver: Hessian-matrix product (one reverse after one vector forward)
- generally: univariate Taylor series up to an arbitrary degree ($\sim$ Rapsodia)

efficient Hessians subject of current research
higher order tensors:

- multivariate (direct ☺, coefficient management ☹) COSY INFINITY
- univariate (one coefficient per degree ☺, interpolation ☹) Adol-C/Rapsodia

COSY INFINITY: specialized, offers tight inclusion via remainder term intervals

non-smooth models

caused by:

- intrinsics (`max, ceil, sqrt,`...)
- branches `if (x<2.5) y=f1(x); else y=f2(x);`
- can cause seemingly erratic derivatives glossed over by FD
- approximate step lengths in linear model
- explicit g-stop facility using high order expansion

we assume fixed parameters!

- Adifor: catches all intrinsic problems via optional exception handling
- Adol-C: taping mechanism and intrinsic handling catches all non-smooth crossings; uses ±`INF` and `NaN`
- ATOMFT (g-stop), Tapenade (experimental estimator)

distinction

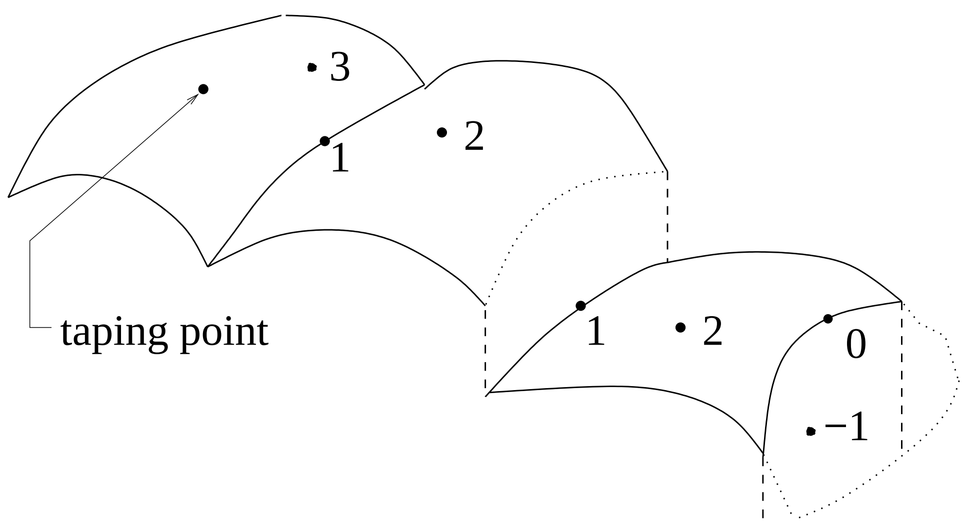
3 locally analytic

2 locally analytic but crossed a (potential) kink (`min,max,abs`) or discontinuity (`ceil`)

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

0 tie on arithmetic comparison (e.g. a branch condition) → potentially discontinuous

-1 arithmetic comparison yields a different value than before → sparsity structure may have changed



Adol-C - general

- `www.math.tu-dresden.de/~adol-c`
- operator overloading creates an execution trace (also called 'tape')
- execution trace is the function representation for all drivers

Speelpenning example $y = \prod_i x_i$

```
double *x = new double[n];
double t = 1;
double y;

for(i=0; i<n; i++) {
  x[i] = (i+1.0)/(2.0+i);
  t *= x[i];
}
y = t;
delete[] x;
```

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

simple overloaded operators for a*b

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;
}
```

in Fortran:

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

module Amult
  use ATypes
  interface operator(*)
    module procedure multArealAreal
    ...
  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 ⇒

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

Adol-C tape

- tape consists of records containing
  - op code
  - result location
  - argument location(s)
  - constant argument value
  - indicator for boolean value, integer results (branches, `max`, `ceil`, ...)
- forward and reverse interpret the tape
- look at `examples/speelpenning.cpp` using `gradient` and `hessian`
- look at the 8 page short reference for parameter values
- ! experimental tapeless forward

Adol-C tape size

- in `examples/additional_examples/speelpenning`
- observe tape and *value stack* sizes with $n = 10, 1000, 10000$
- estimating storage requirements using `tape_stats`
- look at execution times (100 computations for $n = 10000$)
- tape size $\sim$ execution time
- loop unrolling
- larger problems require *checkpointing*
- manual checkpointing, e.g. for time stepping scheme
- some improvements are under development

Adol-C sparsity

sparsity pattern detection

- *safe* and *tight* mode, think
  `P(max(a,b))=P(a)|P(b)` vs. `P(max(a,b))=P(a)` if `max(a,b)==a`
- propagation of `unsigned long`s
- `forward` or `reverse`
- convoluted example code in `examples/additional_examples/sparse`
- 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 pp. 31 and pp. 42

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. 25
- nonzero pattern:

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

 4 = transcend , 3 = rational , 2 = polynomial , 1 = linear , 0 = zero
 negative number k indicate that entries of all B_j with j < -k vanish
```

Adol-C non-smooth

```
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
- necessary safety measure for tape correctness

```
#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 directional derivatives & exceptions

tape at `1.0` and rerun at

- `0.5, xdot=1.0` → `ydot=3`
- `0.0, xdot=1.0` → `ydot=3`
- `0.0, xdot=-1.0` → `ydot=-2`
- `-0.5, xdot=1.0` → `ydot=2`

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

tape at `1.0` and rerun at

- `0.5, xdot=1.0` → `ydot=.707107`
- `0.0, xdot=1.0` → `ydot=INF`
- `0.0, xdot=-1.0` → `ydot=NaN`

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

Adol-C Miscellaneous

- various drivers
- tape dumping tool
- tapeless forward
- tape compression through (manual) loop identification
- non-persistent tape format