The Basic Idea

Vector func ++: $F : \mathbb{R}^n \to \mathbb{R}^m : x \mapsto y = F(x)$

⇓ r overloading (C++)

Internal representation of F ($\equiv$*tape*)

⇓ Interpretation ⇓

Forward mode

$$x(t) = \sum_{j=0}^{d} x_j t^j$$

$$\Downarrow$$

$$y(t) = \sum_{j=0}^{d} y_j t^j + O\left(t^{d+1}\right)$$

$\Longrightarrow$ **Directional derivatives**

Reverse mode

$$y_j = y_j(x_0, x_1, \ldots, x_j)$$

$$\Downarrow$$

$$\frac{\partial y_j}{\partial x_i} = \frac{\partial y_{j-i}}{\partial x_0} = A_{j-i}(x_0, x_1, \ldots, x_{j-i})$$

$\Longrightarrow$ **Gradients (adjoints)**

$$\begin{aligned}
y_0 &= F(x_0) \\
y_1 &= F'(x_0)\, x_1 \\
y_2 &= F'(x_0)\, x_2 + \frac{1}{2} F''(x_0)\, x_1 x_1 \\
y_3 &= F'(x_0)\, x_3 + F''(x_0)\, x_1 x_2 + \frac{1}{6} F'''(x_0)\, x_1 x_1 x_1 \\
&\ldots
\end{aligned}$$

$$\begin{aligned}
\frac{\partial y_0}{\partial x_0} = \frac{\partial y_1}{\partial x_1} = \frac{\partial y_2}{\partial x_2} = \frac{\partial y_3}{\partial x_3} &= A_0 = F'(x_0) \\
\frac{\partial y_1}{\partial x_0} = \frac{\partial y_2}{\partial x_1} = \frac{\partial y_3}{\partial x_2} &= A_1 = F''(x_0)\, x_1 \\
\frac{\partial y_2}{\partial x_0} = \frac{\partial y_3}{\partial x_1} &= A_2 = F''(x_0)\, x_2 + \frac{1}{2} F'''(x_0)\, x_1 x_1 \\
\frac{\partial y_3}{\partial x_0} &= A_3 = F''(x_0)\, x_3 + F'''(x_0)\, x_1 x_2 + \frac{1}{6} F^{(4)}(x_0)\, x_1 x_1 x_1 \\
&\ldots
\end{aligned}$$

Application

Operator overloading concept ⇒ Code modification

- Inclusion of appropriate ADOL-C headers
- Retyping of all involved variables to active data type `adouble`
- Marking active section to be "taped" (`trace_on`/`trace_off`)
- Specification of independent and dependent variables (`<<=`/`>>=`)
- Specification of differentiation task(s)
- Recompilation and Linking with ADOL-C library `libad.a`

Example:

```
#include <adolc/adolc.h>                   // inlusion of ADOL-C headers
...
adouble foo ( adouble x )                  // some activated function
{ adouble tmp;
  tmp = log(x);
  return 3.0*tmp*tmp + 2.0;
}
...
int main (int argc, char* argv[])          // main program or other procedure
{ ...
  double   x[2],  y;
  adouble ax[2], ay;                       // declaration of active variables
  x[0]=0.3; x[1]=2.3;
  trace_on(1);                             // starting active section
    ax[0]<<=x[0]; ax[1]<<=x[1];            // marking independent variables
    ay=ax[0]*sin(ax[1])+ foo(ax[1]);       // function evaluation
    ay>>=y;                                // marking dependend variables
  trace_off();                             // ending active section
  ...
  double g[2];
  gradient(1,2,x,g);                       // application of ADOL-C routine
  ...
  x[0]+=0.1; x[1]+=0.3;                    // application at different argument
  gradient(1,2,x,g);
  ...
}
```

Drivers for Optimization and Nonlinear Equations (C/C++)

$f : \mathbb{R}^n \to \mathbb{R}$

$F : \mathbb{R}^n \to \mathbb{R}^m$

function(tag,m,n,x[n],y[m])	$F(x_0)$
gradient(tag,n,x[n],g[n]) hessian(tag,n,x[n],H[n][n])	$\nabla f(x_0)$ $\nabla^2 f(x_0)$
jacobian(tag,m,n,x[n],J[m][n]) vec_jac(tag,m,n,repeat?,x[n],u[m],z[n]) jac_vec(tag,m,n,x[n],v[n],z[m])	$F'(x_0)$ $u^T F'(x_0)$ $F'(x_0)\,v$
hess_vec(tag,n,x[n],v[n],z[n]) lagra_hess_vec(tag,m,n,x[n],v[n],u[m],h[n])	$\nabla^2 f(x_0)\,v$ $u^T F''(x_0)\,v$
jac_solv(tag,n,x[n],b[n],sparse?,mode?)	$F'(x_0)\,w = b$

Example: Solution of $F(x) = 0$ by Newton's method

```
...
double x[n], r[n];
int i;
...
initialize(x);                    // setting up the initial x
...
function(ftag,n,n,x,r);           // compute residuum r
while (norm(r) > EPSILON)         // terminate if small residuum
{ jac_solv(ftag,n,x,r,0,2);       // compute r:=F'(x)^(-1)*r
  for (i=0; i<n; i++)             // update x
    x[i] -= r[i];
  function(ftag,n,n,x,r);         // compute residuum r
}
...
```

Lowest-level Differentiation Routines

: $\mathbb{R}^n \to \mathbb{R}^m$

Mode (C/C++)

```
zos_forward(tag,m,n,keep,x[n],y[m])
```


- zero-order scalar forward; computes $y = F(x)$
- $0 \leq$ `keep` ≤ 1; `keep` $= 1$ prepares for `fos_reverse` or `fov_reverse`


```
fos_forward(tag,m,n,keep,x0[n],x1[n],y0[m],y1[m])
```


- first-order scalar forward; computes $y_0 = F(x_0)$, $y_1 = F'(x_0)\, x_1$
- $0 \leq$ `keep` ≤ 2; `keep` $= \begin{cases} 1 & \text{prepares for } \texttt{fos\_reverse} \text{ or } \texttt{fov\_reverse} \\ 2 & \text{prepares for } \texttt{hos\_reverse} \text{ or } \texttt{hov\_reverse} \end{cases}$


```
fov_forward(tag,m,n,p,x[n],X[n][p],y[m],Y[m][p])
```

- first-order vector forward; computes $y = F(x)$, $Y = F'(x)\, X$

```
hos_forward(tag,m,n,d,keep,x[n],X[n][d],y[m],Y[m][d])
```

- higher-order scalar forward; computes $y_0 = F(x_0)$, $y_1 = F'(x_0)\, x_1$, $\ldots$, where $x = x_0$, $X = [x_1, x_2, \ldots, x_d]$ and $y = y_0$, $Y = [y_1, y_2, \ldots, y_d]$
- $0 \leq$ `keep` $\leq d + 1$; `keep` $\begin{cases} = 1 & \text{prepares for } \texttt{fos\_reverse} \text{ or } \texttt{fov\_reverse} \\ > 1 & \text{prepares for } \texttt{hos\_reverse} \text{ or } \texttt{hov\_reverse} \end{cases}$

```
hov_forward(tag,m,n,d,p,x[n],X[n][p][d],y[m],Y[m][p][d])
```

- higher-order vector forward; computes $y_0 = F(x_0)$, $Y_1 = F'(x_0)\, X_1$, $\ldots$, where $x = x_0$, $X = [X_1, X_2, \ldots, X_d]$ and $y = y_0$, $Y = [Y_1, Y_2, \ldots, Y_d]$

Reverse Mode (C/C++)

[illegible]e(tag,m,n,u[m],z[n])

- first-ord[illegible]putes $z^T = u^T F'(x)$
- after calling [illegible]s_forward, or hos_forward with keep $= 1$

fov_reverse(tag,m,n,q,U[q][m],Z[q][n])

- first-order vector reverse; computes $Z = UF'(x)$
- after calling zos_forward, fos_forward, or hos_forward with keep $= 1$

hos_reverse(tag,m,n,d,u[m],Z[n][d+1])

- higher-order scalar reverse; computes the adjoints $z_0^T = u^T F'(x_0) = u^T A_0$, $z_1^T = u^T F''(x_0)\, x_1 = u^T A_1$, ..., where $Z = [z_0, z_1, \ldots, z_d]$
- after calling fos_forward or hos_forward with keep $= d + 1 > 1$

hov_reverse(tag,m,n,d,q,U[q][m],Z[q][n][d+1],nz[q][n])

- higher-order vector reverse; computes the adjoints $Z_0 = UF'(x_0) = UA_0$, $Z_1 = UF''(x_0)\, x_1 = UA_1$, ..., where $Z = [Z_0, Z_1, \ldots, Z_d]$
- after calling fos_forward or hos_forward with keep $= d + 1 > 1$
- optional nonzero pattern nz ($\Rightarrow$ manual)

Example:

```
...
double x[n], y[m], **I, **J;
I=myallocI2(m);                  // allocation of identity matrix
J=myalloc2(m,n);                 // allocation of Jacobian matrix
...
initialize(x);                   // setting up the argument x
...
zos_forward(ftag,m,n,1,x,y);     // computing the Jacobian by
fos_reverse(ftag,m,n,m,I,J);     // reverse mode of AD
...
```

Low-level Differentiation Routines

Forward Mode (C++ interfaces)

`forward(tag,m,n,d,keep,X[n][d+1],Y[m][d+1])` `forward(tag,m=1,n,d,keep,X[n][d+1],Y[d+1])`	hos, fos, zos hos, fos, zos
`forward(tag,m,n,d=0,keep,x[n],y[m])` `forward(tag,m,n,keep,x[n],y[m])`	zos zos
`forward(tag,m,n,p,x[n],X[n][p],y[m],Y[m][p])`	fov
`forward(tag,m,n,d,p,x[n],X[n][p][d],` `y[m],Y[m][p][d])`	hov

Reverse Mode (C++ interfaces)

`reverse(tag,m,n,d,u[m],Z[n][d+1])` `forward(tag,m=1,n,d,u,Z[n][d+1])`	hos hos
`reverse(tag,m,n,d=0,u[m],z[n])` `reverse(tag,m=1,n,d=0,u,z[n])`	fos fos

`reverse(tag,m,n,d,q,U[q][m],Z[q][n][d+1],nz[q][n])`
`reverse(tag,m=1,n,d,q,U[q],Z[q][n][d+1],nz[q][n])`
`reverse(tag,m=1,n,d,Z[m][n][d+1],nz[m][n])` $(U = I_m)$
	hov
	hov
	hov

`reverse(tag,m,n,d=0,q,U[q][m],Z[q][n])`
`reverse(tag,m,n,q,U[q][m],Z[q][n]`
`reverse(tag,m=1,n,d=0,q,U[q],Z[q][n])`
	fov
	fov
	fov

Drivers for Ordinary Differential Equations (C/C++)

ODE: $x'(t) = F(x(t)), \qquad x(0) = x_0$

```
forodec(tag,n,tau,dold,d,X[n][d+1])
```


- recursive forward computation of $x_{d_{old}+1}, \ldots, x_d$ from $x_0, \ldots, x_{d_{old}}$ (by $x_{i+1} = \frac{1}{1+i} y_i$)
- application with $d_{old} = 0$ delivers truncated Taylor series $\sum_0^d x_j t^j$ at base point x_0


```
hov_reverse(tag,n,n,d-1,n,I[n][n],A[n][n][d],nz[n][n])
```


- reverse computation of $A_j = \frac{\partial y_j}{\partial x_0}$, $j = 0, \ldots, d$ after calling `forodec` with degree d
- optional nonzero pattern `nz` ($\Rightarrow$ manual)


```
accodec(n,tau,d-1,A[n][n][d],B[n][n][d],nz[n][n])
```


- accumulation of total derivatives $B_j = \frac{dx_j}{dx_0}$, $j = 0, \ldots, d$ from the partial derivatives $A_j = \frac{\partial y_j}{\partial x_0}$, $j = 0, \ldots, d$ after calling `hov_reverse`
- optional nonzero pattern `nz` ($\Rightarrow$ manual)

C++: Special C++ interfaces can be found in file `SRC/DRIVERS/odedrivers.h`!

Example:

```
...
double x[n], **I, **X, ***A, ***B;
I=myallocI2(n);                       // allocation of identity matrix
X=myalloc2(n,5);                      // allocation of matrix X
A=myalloc3(n,n,4); B=myalloc3(n,n,4); // allocation of tensors A and B
...
initialize(X);                        // setting up the argument x_0
...
forodec(ftag,n,1.0,0,4,X);            // compute x_1,...,x_4
hov_reverse(ftag,n,n,3,n,I,A,NULL);   // compute A_0,...,A_3
accodec(ftag,n,1.0,3,A,B,NULL);       // accumulate B_0,...,B_3
...
```

ADOL-C provides

- Low-level differentiation routines (`forward`/`reverse`)
- Easy-to-use drivers for
  - the solution of optimization problems and nonlinear equations
  - the integration of ordinary differential equations
  - the evaluation of higher derivative tensors (⇒ manual)
- Derivatives of implicit and inverse functions (⇒ manual)
- Forward and backward dependence analysis (⇒ manual)

Recent developments

- Efficient detection of Jacobian/Hessian sparsity structure
- Exploitation of Jacobian/Hessian sparsity by matrix compression
- Integration of checkpointing routines
- Exploitation of fixpoint iterations
- Differentiation of OpenMP parallel programs

Future developments

- Internal optimizations to reduce storage needed for reverse mode
- Recovery of structure for internal function representation
- Differentiation of MPI parallel programs

Contact/Resources

- E-mail: `adol-c@list.coin-or.org`
- WWW: `http://www.coin-or.org/projects/ADOL-C.xml`