Automatic Differentiation of Computer Programs

University of Paris VI, Laboratoire J.-L. Lions

Olivier Pironneau[1]

LJLL-University of Paris VI

March 20, 2012

Outline

1
- Finite Differences
- Automatic Differentiation

In Search of Derivatives

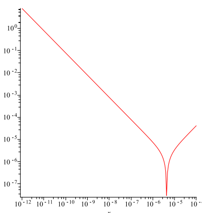
Suppose a computer program implements in C

$$x = f(a), \quad \texttt{double f(double a)}$$

Find x'_a? The solution is given by $x'_a = f'(a)$. How good is

```
double a=1., da=1e-4, dxda = (f(a+da)-f(a))/da
```

;



Example with $f(a) = sin(a), a = 1$. log-log plot of $|dxda - cos(1.)|$ (computed with Maple-14)

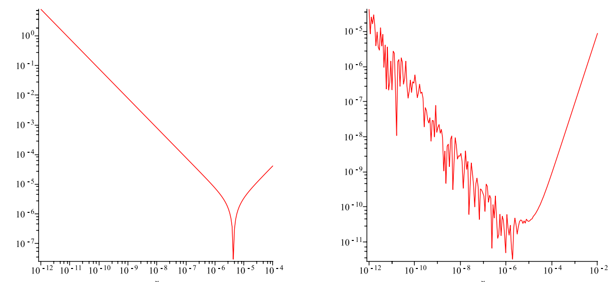
In Search of Derivatives (II)

Centered finite differences (handy in Java where operator overloading is lacking):

$$\frac{1}{2\delta a}(f(a+\delta a) - f(a-\delta a)) = f'(a) + f^{(3)}\frac{\delta a^2}{6} + o(\delta a^3)$$

```
double a=1., da=1e-4, dxda = (f(a+da)-f(a-da))/da/2
```

;



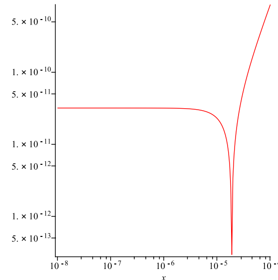
In Search of Derivatives (III)

Complex finite differences

$$Re[\frac{1}{\mathbf{i}\delta a}(f(a+\mathbf{i}\delta a)-f(a))]=f'(a)-f^{(3)}\frac{\delta a^2}{6}+o(\delta a^3)$$

```
double a=1., da=1e-4, dxda = ((f(a+I*da)-f(a))/da/I).Re()
```

;



Example with $f(a)=sin(a), a=1$. (computed with Maple-14).
It requires only one evaluation $\neq a$ and it is precise for all δa!

Outline

1
- Finite Differences
- Automatic Differentiation

Principle of Automatic Differentiation

Fundamental Remark Relation between derivatives and differentials
Let $J(u,x,a),\ u,x,a\in\mathbb{R}$ its differential is

$$dJ=J'_u(u,x,a)du+J'_x(u,x,a)dx+J'_a(u,x,a)da$$

where J'_u, J'_x, J'_a are the partial derivatives of J defined by, for example

$$J'_u(u,x,a)=\lim_{\delta u\to 0}\frac{J(u+\delta u,x,a)-J(u,x,a)}{\delta u}$$

By taking $dx=da=0$ and $du=1$, we have $J'_x(u,x,a)=dJ$.

A simple example. Let $J(u)=|u-u_d|^2$, then its differential is

$$dJ=2(u-u_d)(du-du_d),\qquad J'_u=2(u-u_d)(1.0-0.0)$$

Obviously J'_u is obtained by putting $du=1,\ \ du_d=0$.

A simple example

Let $J(u)=|u-u_d|^2$, then its differential is $dJ=2(u-u_d)(du-du_d)$.
Now suppose that J is programmed in C/C++ by

```
double J(double u, double u_d){
    double z = u-u_d;
    double J = z*(u-u_d);
    return J;
 }
int main(){  double u=2,u_d = 0.1;
    cout << J(u,u_d) << endl;
 }
```

A program which computes J and its differential can be obtained by writing above each differentiable line its differentiated form:

A simple example (cont)

```
 struct {double val[2];} ddouble;

ddouble J(double u, double u_d, double du, double du_d)
{                double dz = du - du_d;
    double  z = u-u_d;
                 double dJ = dz*(u-u_d) + z*(du - du_d);
    double J= z*(u-u_d);
    ddouble ddz;
    ddz.val[0]=J;  ddz.val[1] = dJ;
    return ddz;
}
int main()
     {      double u=2,u_d = 0.1;
            cout << J(u,u_d,1,0).val[1] << endl;
     }
```

A simple example (II)

```
ddouble JandDJ(ddouble u, ddouble u_d) {
    ddouble z,J;
    z.val[1] = u.val[1]-u_d.val[1];
    z.val[0] = u.val[0]-u_d.val[0];
    J.val[1] = z.val[1] * ( uval[0] - u_d.val[0] )
             + z.val[0] * ( uval[1] - u_d.val[1] );
    J.val[0] = z.val[0] * ( u.val[0] - u_d.val[0] );
    return J;
}
int main() {
    ddouble u, u_d;
    u.val[0]=2; u_d.val[0]=0.1;     u.val[1]=1;  u_d.val[1]=0;
    cout <<J(u,u_d).val[1]<<endl;
}
```

In C++ the program can be simplified further by redefining the operators =, - and *. Then a class has to be used instead of a struct!

The class ddouble

```
class ddouble{ public: double val[2];
       ddouble(double a, double b=0){ val[0] = a; val[1]=b;}
ddouble operator=(const ddouble& a)
  { val[1] = a.val[1]; val[0]=a.val[0];
    return *this;  }
friend ddouble operator - (const ddouble& a, const ddouble& b)
       {    ddouble c;
            c.val[1] = a.val[1] - b.val[1];   // (a-b)'=a'-b'
            c.val[0] = a.val[0] - b.val[0];
            return c;      }
friend ddouble operator * (const ddouble& a, const ddouble& b)
       {    ddouble c;
            c.val[1] = a.val[1]*b.val[0] + a.val[0]* b.val[1];
            c.val[0] = a.val[0] * b.val[0];
            return c;}
};
```

Limitations

```
program newtontest
x=0.0;
al=0.5                subroutine newton(x,n,al)
call newton(x,10,al)  do i=1,n
write(*,*) x             f = x-alpha*cos(x)
end                      fp= 1+alpha*sin(x)
                         x=x-f/fp
                       enddo
                       return
                       end
```

2n adjoint variables are needed! while the theory is

$$f(x,\alpha)=0 \quad\Rightarrow\quad x'f'_x+f'_\alpha=0 \quad\Rightarrow\quad x'=-\frac{f'_\alpha}{f'_x}$$

So it is better to understand the output of AD-reverse and clean it.

Historical Remarks

- The creator of Fortran in the sixties were aware of the capacity of compilers to differentiate a program
- The creator of modern AD is Andreas Griewank (Now in Germany)
- Adol-C is based on operator overloading but the authors didn't want to make the program public
- Meteo France and Dassault Aviation were strongly interested to support `odysse` (but never paid)
- Now INRIA supports `tapenade`

Uwe Naumann: *The art of differentiating computer programs*. SIAM series. Pittsburg (2012).
Andreas Griewank and Alan Walther Naumann: *Evaluating Derivatives: Principle and Techniques of Algorithmic Differentiation*. SIAM series. Pittsburg (2008).

Reverse Mode

Consider $J(u,x)$ where $A(u)x = f(u)$. It models a typical computer program where x are the intermediary variables. In computer programs A may be nonlinear but it is lower triangular. Then

$$\mathrm{d}J = J'_u \mathrm{d}u + J'_x \mathrm{d}x : A\mathrm{d}x = (f'_u - A'_u x)\mathrm{d}u$$

Introducing p leads to

$$A^T p = J'_x \quad \Rightarrow J'_x \mathrm{d}x = \mathrm{d}x A^T p = (A\mathrm{d}x)p = p(f'_u - A'_u x)\mathrm{d}u$$

Therefore

$$\mathrm{d}J = (J'_u + p(f'_u - A'_u x))\mathrm{d}u$$

Consider $u \in \mathbb{R}^m$, $x \in \mathbb{R}^n$, A is $n \times n$ and $p \in \mathbb{R}^n$. p is the same whatever $\mathrm{d}u$ so it is advantageous all partial derivatives of J are the goal.

Reverse Mode (II)

Builds the Lagrangian by associating to each program line a dual variable p: **each code line is seen as an equality constraint**.

```
float E(u){float x1,x2;x1=(1+u)*log(u);x2=x1+cos(u);E=x1*x2;}
```

$L = p_1[x_1 - (1+u)\log(u)] + p_2[x_2 - x_1 - \cos(u)] + E - x_1 x_2.$

Stationarity of L with respect to all variables

- $\partial_{p_i} L = 0$ gives back the program
- $\partial_{x_i} L = 0$ gives the adjoint state
- $\partial_u L = \mathrm{d}E/\mathrm{d}u$ gives the derivative
- $\frac{\partial L}{\partial x_i} = 0$ must be written in reverse order (x_2, x_1)

$$\frac{\partial L}{\partial x_2} = 0 = p_2 - x_1 \qquad \frac{\partial L}{\partial x_1} = 0 = p_1 - p_2 - x_2.$$

This gives p_2 first and then p_1. Now a theorem says that $E'_u = L'_u$ when $\partial_{x_i} L = \partial_{p_i} L = 0$, so

$$\frac{dE}{du} = \frac{\partial L}{\partial u} = p_2 x_1 \sin(u) - p_1(\log(u) + \frac{1+u}{u})$$