Automatic Differentiation Tools for FreeFem++

Workshop FreeFem++

Sylvain Auliac (*s-auliac@netcourrier.com*) - LJLL

September 16, 2009

An Overview of FAD (Forward Automatic Differentiation)

- FAD theoretical principles
- Informatic point of view

FAD in FreeFem++

- FADed FreeFem++ in action
- Perspectives of development

FAD principle

What is Automatic Differentiation (AD)?

AD denominate a set of technics that allows to calculate automatically and “exactly” the derivatives of the outputs of a programm with respect to some of its inputs.

There are several kinds of AD, each of them shows up efficiency in a well defined field of aplication.

♡ FAD = Forward Automatic Differentiation

FAD principle

Let P be a programm and call :

- $f : (x_i)_{1 \leq i \leq n} \longmapsto (y_i)_{1 \leq i \leq m}$ the function implemented by P
- P' the differentiated version of P
- f' the function implemented by P'

One derivative forward mode

- $f' : (x_1, dx_1, x_2, dx_2, \ldots, x_n, dx_n) \longmapsto (y_1, dy_1, \ldots, y_m, dy_m)$
- $\forall j \in \{1, \ldots, m\}, dy_j = \sum_{i=1}^{n} \frac{\partial y_j}{\partial x_i} dx_i$
- with k fixed, if $\forall i, dx_i = \delta_{k,i}$ then $\forall j, dy_j = \frac{\partial y_j}{\partial x_k}$

FAD principle

Multi derivatives forward mode

Let p be an integer ($1 \leq p \leq n$).

- $f' : (x_1, \widetilde{\nabla x_1}, x_2, \widetilde{\nabla x_2}, \ldots, x_n, \widetilde{\nabla x_n}) \mapsto (y_1, \widetilde{\nabla y_1}, \ldots, y_m, \widetilde{\nabla y_m})$ where $\widetilde{\nabla x_k} \in \mathbb{R}^p$
- $\forall j \in \{1, \ldots, m\}, \widetilde{\nabla y_j} = \sum_{i=1}^{n} \frac{\partial y_j}{\partial x_i} \widetilde{\nabla x_i}$
- Special case when $p = n$ and $\forall i, \widetilde{\nabla x_i} = (\delta_{i,k})_{1 \leq k \leq n}$ then :

$$\forall j, \widetilde{\nabla y_j} = \nabla y_j = \begin{pmatrix} \frac{\partial y_j}{\partial x_1} \\ \frac{\partial y_j}{\partial x_2} \\ \vdots \\ \frac{\partial y_j}{\partial x_n} \end{pmatrix} \text{ such that } \left(\widetilde{\nabla y_j}\right)_{1 \leq j \leq m} = J_f \tag{1}$$

FAD and programming

How to modifie a programm to support FAD:

- Add and associate a new variable to each variable of the numeric type in the programm to store the derivative (p new variables are needed for each pre-existing variable for multi-derivative AD).

- write the update(s) for the associated derivative(s) just before each change of each variable induced by a calculus, using the following derivations formulas ($f, g : \mathbb{R}^n \longrightarrow \mathbb{R}$) :

$$\nabla(f \pm g) = \nabla f \pm \nabla g \; ; \; \nabla(f * g) = g \nabla f + f \nabla g \; ; \; \nabla(f/g) = \frac{g \nabla f - f \nabla g}{g^2}$$

$$\text{If } u : \mathbb{R} \to \mathbb{R}, v : \mathbb{R}^n \to \mathbb{R}, \nabla(u \circ v) = (u' \circ v) \nabla v$$

FAD and programming - exemple with 1 derivative

This is a C piece of code :

```
//u's computed earlier
double x=u-1./u;
double y=x+log(u);
double J=x+y;
```

and its AD associated one :

```
// so should be du
double dx=du + du/(u*u);
double dy=dx + du/u;
double dJ=dx+dy;
```

Whole new code :

```
dx=du + du/(u*u);
x=u-1./u;
dy=dx + du/u;
y=x+log(u);
dJ=dx+dy;
J=x+y;
```

Pros and cons of the forward mode

Advantages:

- Easy to handle.

Limitations

Pros and cons of the forward mode

Advantages:

- Easy to handle.
- Allows fast development in (low-level) pre-existing codes.

Limitations

Pros and cons of the forward mode

Advantages:

- Easy to handle.
- Allows fast development in (low-level) pre-existing codes.
- Modifications strongly reduced in programming languages with overloading features.

Limitations

Pros and cons of the forward mode

Advantages:

- Easy to handle.
- Allows fast development in (low-level) pre-existing codes.
- Modifications strongly reduced in programming languages with overloading features.

Limitations

- Decrinsing efficiency with relatively modest number of derivation parameters.

Pros and cons of the forward mode

Advantages:

- Easy to handle.
- Allows fast development in (low-level) pre-existing codes.
- Modifications strongly reduced in programming languages with overloading features.

Limitations

- Decrinsing efficiency with relatively modest number of derivation parameters.
- Dramatic augmentation of needed memory.

Optimal Control

Problem :

Let Ω be a domain of $\mathbb{R}^2$ partitioned in n subdomains Ω_i.

$$\min_{a \in \mathbb{R}^n} \int_\Omega |u_a - u_d|^2 \quad , -\Delta u_a = f_a \text{ and } u_a|_{\partial\Omega} = 0$$

$$f_a = \sum_{i=1}^{n} a_i I_{\Omega_i}$$

Let's call $J : \mathbb{R}^n \longrightarrow \mathbb{R}$ the functionnal $a \longmapsto \int_\Omega |u_a - u_d|^2$

Note that : $J(a) = \frac{1}{2} \int_\Omega |u_a|^2 - \int_\Omega u_a u_d + \frac{1}{2} \|u_d\|^2_{L^2(\Omega)}$

Optimal Control

Conjugued Gradient Algorithm:

Initialization:

- $\mathbb{R}^n \ni d_0 = \nabla J(a_0)$
- $\rho_0 = \frac{(\nabla J(a_0), d_0)}{\|u_{a_0}\|^2_{L^2(\Omega)}}$
- $a_1 = a_0 - \rho_0 d_0$

Iterations, if a_k is known :

- $d_k = \nabla J(a_k) + \frac{\|\nabla J(a_k)\|^2}{\|\nabla J(a_{k-1})\|^2} d_{k-1}$, we need u_{a_k} : $-\Delta u_{a_k} = f_{a_k}$
- $\rho_k = \frac{(\nabla J(a_k), d_k)}{\|u_{d_k}\|^2_{L^2(\Omega)}}$, we need u_{d_k} : $-\Delta u_{d_k} = f_{d_k}$
- $a_{k+1} = a_k - \rho_k d_k$

Optimal Control - FreeFem Script

FreeFem Script for n=5

```
real[int] A(5),D(5),DD(5);
real h = 1./N;
for(int i=1;i<N;i++) {
  A[i] = 0;                               //  initializations
  D[i] = 0; AAA[i] = 0;
  SetDiff(A[i],i);
}
A[0] = 1.;SetDiff(A[0],0);
                //  Definition of second member functions
func real R2(real xx,real yy)
  {return xx*xx + yy*yy;}
func real FA(real xx,real yy)
{
  int n = floor(R2(xx,yy)/h);
  n = n>4 ? 4 : n;                 //  can't use ''region''
  return A[n];             //  'cause of memmory leak :-(
}
func real FD ...                   //  the same with D array
```

```
func fA = FA(x,y);
func fD = FD(x,y);
func g = 0;          //  Dirichlet boundary condition

border C0(t=0,2*pi)
  {x=0.2*cos(t); y=0.2*sin(t);label=0;}
border C1(t=0,2*pi)
  {x=0.4*cos(t); y=0.4*sin(t);label=1;}
border C2(t=0,2*pi)
  {x=0.6*cos(t); y=0.6*sin(t);label=2;}
border C3(t=0,2*pi)
  {x=0.8*cos(t); y=0.8*sin(t);label=3;}
border C4(t=0,2*pi)
  {x=cos(t); y=sin(t);label=4;}
mesh Th =
  buildmesh(C0(10)+C1(20)+C2(30)+C3(40)+C4(50));
plot(Th,wait=1,ps="partitioned_disc.eps");

fespace Vh(Th,P1);
Vh ud = 1. - (x*x + y*y);     //  Exact solution for A=[4,4,4,4,4]
```

```
Vh uhA,vh,uhD,fff;

real Jm1 = 0;          //   to save J in CG iterations
real gradJ2 = 0, gradJ2m1 = 0;       //   to store and save square norm of grad(J)

ofstream file("poisson5/donnees.dat");


for(int iter=0;iter<60;iter++)
{
  solve Poisson(uhA,vh) =
      int2d(Th)(dx(uhA)*dx(vh) + dy(uhA)*dy(vh))
    - int2d(Th)(fA*vh)
    + on(4,uhA=g);
  real J0 = int2d(Th)(uhA*uhA);
  real J1 = int2d(Th)(uhA*ud);
  real J = 0.5*J0 - J1 + 0.5*int2d(Th)(ud*ud);
  gradJ2m1 = gradJ2;  //   Saving the square norm (to avoid recalculation)
```

```
gradJ2 = Grad2(J);
real rho;           //   Conjugued Gradient algorithm
if(iter==0){                         //   First CG iteration
  for(int i=0;i<N;i++) {DD[i] = J_i;}
  rho = Grad2(J);
  rho /= (J0>0 ? J0 : 1.);
  for(int i=0;i<N;i++) A[i] -= rho*DD[i];
  for(int i=0;i<N;i++){SetDiff(A[i],i);}}
else{                                //   real CG iterations
  for(int i=0;i<N;i++){ //   new descent direction
    D[i] = J_i + DD[i]*gradJ2/gradJ2m1;}
  solve PoissonD(uhD,vh) =
        int2d(Th)(dx(uhD)*dx(vh)+dy(uhD)*dy(vh))
      - int2d(Th)(fD*vh) + on(4,uhD=0);
  real J0D = int2d(Th)(uhD*uhD);
  for(int i=0;i<N;i++) {rho += (J_i * D[i]);}
  rho /= J0D;
  for(int i=0;i<N;i++){
    A[i] -= rho*D[i];
    SetDiff(A[i],i);}
}                                //   etc..  etc..
}
```

Finding Dirichlet to fit Neumann

The problem:

Let $\Omega \subset \mathbb{R}^2$ with $\partial\Omega = \Gamma_1 \cup \Gamma 2$, $f \in L^2(\Omega)$ and $g_n \in L^2(\Gamma_2)$
Consider the $J : L^2(\Gamma_2) \longrightarrow \mathbb{R}$ such that :

$$J(g) = \frac{1}{2}\int_{\Gamma_2} \left| \frac{\partial u_g}{\partial n} - g_n \right|^2 , \; u_g : \begin{cases} -\Delta u_g = f & \text{dans } \Omega \\ u_g = 0 & \text{sur } \Gamma_1 \\ u_g = g & \text{sur } \Gamma_2 \end{cases}$$

Resolution algorithm:

- Same algorithm as the preceding problem with a new functionnal.
- Control parameter living in an infinite dimensionnal space -¿ discretization needed.

Scripting details

Here Ω is a square with 21 segments on each of its side.

The control parameter:

With $P1$ finite elements, the differentiation variables are the values taken by g on each vertices of Γ_2.
For FreeFem :

```
real[int] A(20);   //   Initialize to zero and SetDiff...
func real g(real a,real b)
{
  int k = floor(a/h);          //   Uniform discretization
  if(b>0) return 0.;
  else
  {
    real Akp1 = k<20 ? A[k] : 0;
    real Ak = k>0 ? (k<21 ? A[k-1]:0) : 0;
    return((Akp1-Ak)*(a - k*h)/h + Ak);//   P1 by pieces
  }
}
```

The functionnal J:

Decomposition with bilinear and linear forms :

$$J(g) = \frac{1}{2}\int_{\Gamma_2}\left|\frac{\partial u_g}{\partial n}\right|^2 - \int_{\Gamma_2}\frac{\partial u_g}{\partial n} g_d + \frac{1}{2}\|g_d\|^2_{L^2(\Gamma_2)}$$

Freefem script:

```
real J0 = int1d(Th,1)(dy(uhg)*dy(uhg));
real J1 = int1d(Th,1)(dy(uhg)*gd);
real J2 = int1d(Th,1)(gd*gd);
J = 0.5*J0 - J1 + 0.5*J2;
```

Perspectives of development

Immediate works:

- Some bugs to fix...
- Possible use with BFGS.
- Setting the number of derivatives in the script.
- Optimisation and improvment of the AD-related syntax.

More difficult works:

- Merging AD version of FreeFem++ with the real one.
- Compatibility with complex numbers.
- Differentiation with respect to the geometry.
- "Mathematization" of the syntax.
- Adding new AD styles (inverse and andjoint models).