Introduction to FADBAD, a C++ program package for automatic differentiation

C0202 – Topics in Numerical Analysis

Ole Stauning <os@imm.dtu.dk>
Department of Mathematical Modelling
Technical University of Denmark

11 March 1997

IMM

Motivation

How do we obtain derivatives?

1) **Symbolic differentiation:** On expressions. By hand or by a computer algebra system (Maple, Axiom or Mathematica).

   PROBLEMS: Some "simple" expressions will lead to complicated derivatives which are expensive to evaluate

   Example:

$$
\begin{aligned}
g(x) &= 2xe^{x^2}\sin(e^{x^2}) \\
g'(x) &= 2\,e^{x^2}\sin(e^{x^2}) + 4\,x^2e^{x^2}\sin(e^{x^2}) + \\
&\quad 4\,x^2\left(e^{x^2}\right)^2\cos(e^{x^2}) \\
g''(x) &= 12\,xe^{x^2}\sin(e^{x^2}) + 12\,e^{2\,x^2}\cos(e^{x^2})x + \\
&\quad 8\,x^3e^{x^2}\sin(e^{x^2}) + 24\,x^3e^{2\,x^2}\cos(e^{x^2}) - \\
&\quad 8\,x^3e^{3\,x^2}\sin(e^{x^2})
\end{aligned}
$$

   etc.

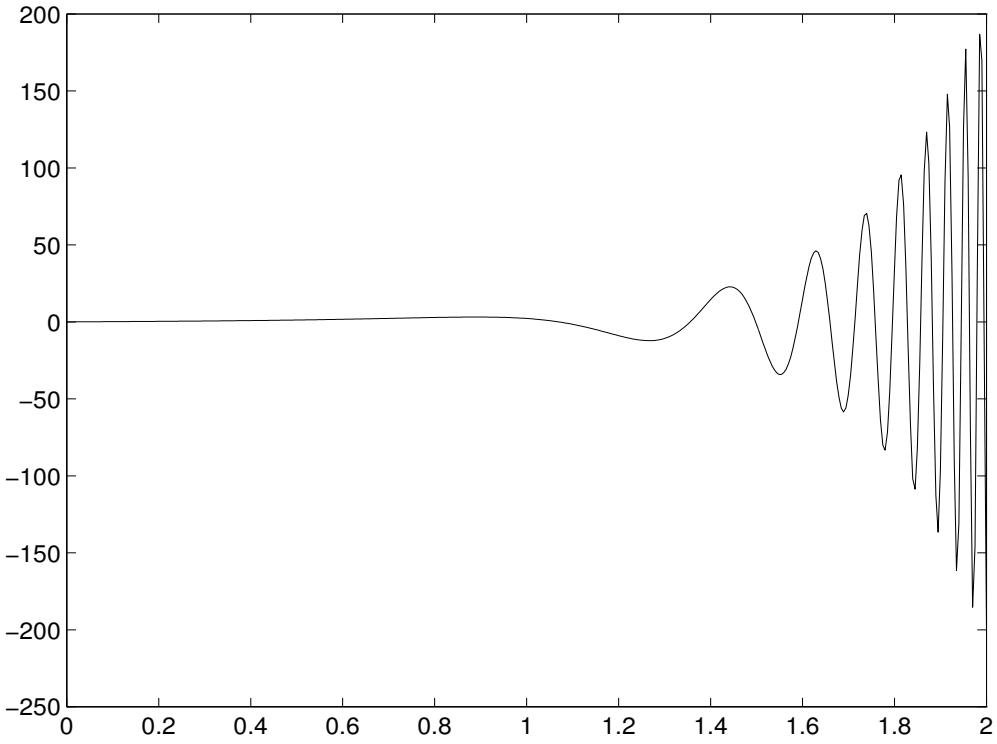
   Most of time in Global Optimization goes to evaluating functions and their derivatives.

Motivation

How do we obtain derivatives?

2) **Numerical differentiation (divided differences):** On a "black box" which evaluates the function.

$$g'(x) \approx \frac{g(a) - g(b)}{a - b}, \quad a < x < b.$$



The function $g(x) = 2xe^{x^2}\sin(e^{x^2})$.

PROBLEMS: Truncation errors, inaccurate and no interval enclosures.

Motivation

How do we obtain derivatives?

3) **Automatic Differentiation** on a program which computes function values.

$g(x), g'(x), g''(x)$ can be computed by:

```
void main(){
FFINTERVAL x,gx,tmp;

  x=1.5;        // The point of evaluation.

  x.diff(0,1);      // Get first order derivatives wrt. x.
  x.x().diff(0,1); // Get second order derivatives wrt. x.

  tmp=exp(sqr(x));        // This is the function
  gx=2*x*tmp*sin(tmp);    // implementation.

  cout << gx.x().x() << endl   // Print function value,
       << gx.d(0).x() << endl  // first order derivative,
       << gx.d(0).d(0) << endl;// second order derivative.
}
```

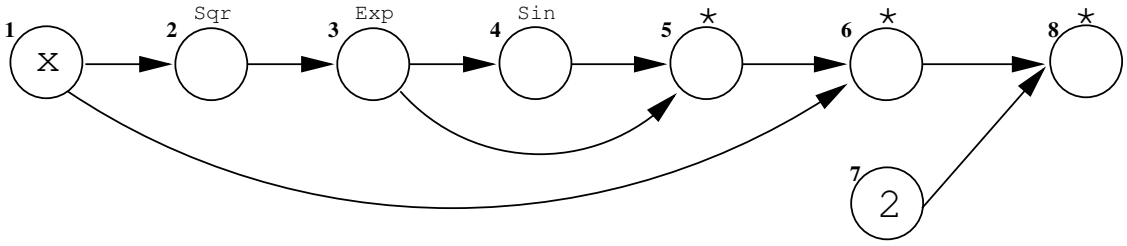
Quite efficient and only the function and not the derivatives has to be implemeted!

PROBLEM FORMULATION

Consider the interval function:

```
INTERVAL g(INTERVAL x){
  INTERVAL tmp(Exp(Sqr(x)));
  return 2*x*tmp*Sin(tmp);
}
```

The equivalent *Directed Acyclic Graph (DAG)*:



And the equivalent *Codelist*:

$$
\begin{aligned}
\tau_1 &= x \\
\tau_2 &= \mathrm{sqr}(\tau_1) \\
\tau_3 &= \exp(\tau_2) \\
\tau_4 &= \sin(\tau_3) \\
\tau_5 &= \tau_3 \cdot \tau_4 \\
\tau_6 &= \tau_1 \cdot \tau_5 \\
\tau_7 &= 2 \\
\tau_8 &= \tau_7 \cdot \tau_6
\end{aligned}
$$

PROBLEM FORMULATION

In general we have:

initialize the values:

$\tau_i = f_i = x_i, \quad \text{for } i = 1, \ldots m.$

compute:

for $i = m + 1$ to n,

$\tau_i = f_i(\tau_1, \ldots, \tau_{i-1}).$

The "chain rule" for composite functions:

$$\frac{\partial \tau_i}{\partial \tau_j} = \delta_{ij} + \sum_{k=j}^{i-1} \frac{\partial f_i}{\partial \tau_k} \frac{\partial \tau_k}{\partial \tau_j}, \quad \text{where} \quad \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{otherwise} \end{cases}$$

Or in matrix formulation:

$$\mathbf{D}\tau = \mathbf{I} + \mathbf{D}\mathbf{f}\mathbf{D}\tau,$$

where

$$\mathbf{Df} = \left\{ \frac{\partial f_i}{\partial \tau_j} \right\}_{i,j=1,\ldots,n} = \begin{pmatrix} 0 & \ldots & & \\ \frac{\partial f_2}{\partial \tau_1} & 0 & \ldots & \\ \frac{\partial f_3}{\partial \tau_1} & \frac{\partial f_3}{\partial \tau_2} & 0 & \ldots \\ \ldots & \ddots & \ddots & \ddots \end{pmatrix},$$

$$\mathbf{D}\tau = \left\{ \frac{\partial \tau_i}{\partial \tau_j} \right\}_{i,j=1,\ldots,n} = \begin{pmatrix} 1 & 0 & \ldots & \\ \frac{\partial \tau_2}{\partial \tau_1} & 1 & \ddots & \\ \frac{\partial \tau_3}{\partial \tau_1} & \frac{\partial \tau_3}{\partial \tau_2} & 1 & \ddots \\ \ldots & \ddots & \ddots & \ddots \end{pmatrix}.$$

OBTAINING THE MATRIX $\mathbf{D}\tau$

$$
\begin{aligned}
\mathbf{D}\tau = \mathbf{I} + \mathbf{D}\mathbf{f}\mathbf{D}\tau \quad &\Leftrightarrow \\
(\mathbf{I} - \mathbf{D}\mathbf{f})\mathbf{D}\tau = \mathbf{I} \quad &\Leftrightarrow \qquad (1) \\
\mathbf{D}\tau = (\mathbf{I} - \mathbf{D}\mathbf{f})^{-1} \quad &\Leftrightarrow \\
\mathbf{D}\tau(\mathbf{I} - \mathbf{D}\mathbf{f}) = \mathbf{I} \quad &\Leftrightarrow \\
(\mathbf{I} - \mathbf{D}\mathbf{f})^T \mathbf{D}\tau^T = \mathbf{I} \quad &\qquad (2)
\end{aligned}
$$

From (1) we get: $\mathbf{I} =$

$$
\begin{pmatrix}
1 & 0 & \dots & & 0 \\
-\frac{\partial f_2}{\partial \tau_1} & 1 & \ddots & & 0 \\
-\frac{\partial f_3}{\partial \tau_1} & -\frac{\partial f_3}{\partial \tau_2} & 1 & \ddots & 0 \\
\dots & \ddots & \ddots & \ddots & \vdots \\
-\frac{\partial f_n}{\partial \tau_1} & -\frac{\partial f_n}{\partial \tau_2} & \dots & -\frac{\partial f_n}{\partial \tau_{n-1}} & 1
\end{pmatrix}
\cdot
\begin{pmatrix}
1 & 0 & \dots & & 0 \\
\frac{\partial \tau_2}{\partial \tau_1} & 1 & \ddots & & 0 \\
\frac{\partial \tau_3}{\partial \tau_1} & \frac{\partial \tau_3}{\partial \tau_2} & 1 & \ddots & 0 \\
\vdots & \ddots & \ddots & \ddots & \vdots \\
\frac{\partial \tau_n}{\partial \tau_1} & \frac{\partial \tau_n}{\partial \tau_2} & \dots & \frac{\partial \tau_n}{\partial \tau_{n-1}} & 1
\end{pmatrix}
$$

From (2) we get: $\mathbf{I} =$

$$
\begin{pmatrix}
1 & -\frac{\partial f_2}{\partial \tau_1} & \dots & -\frac{\partial f_{n-1}}{\partial \tau_1} & -\frac{\partial f_n}{\partial \tau_1} \\
\vdots & \ddots & \ddots & \ddots & \dots \\
0 & \ddots & 1 & -\frac{\partial f_{n-1}}{\partial \tau_{n-2}} & -\frac{\partial f_n}{\partial \tau_{n-2}} \\
0 & & \ddots & 1 & -\frac{\partial f_n}{\partial \tau_{n-1}} \\
0 & & \dots & 0 & 1
\end{pmatrix}
\cdot
\begin{pmatrix}
1 & \frac{\partial \tau_2}{\partial \tau_1} & \dots & \frac{\partial \tau_{n-1}}{\partial \tau_1} & \frac{\partial \tau_n}{\partial \tau_1} \\
\vdots & \ddots & \ddots & \ddots & \dots \\
0 & \ddots & 1 & \frac{\partial \tau_{n-1}}{\partial \tau_{n-2}} & \frac{\partial \tau_n}{\partial \tau_{n-2}} \\
0 & & \ddots & 1 & \frac{\partial \tau_n}{\partial \tau_{n-1}} \\
0 & & \dots & 0 & 1
\end{pmatrix}
$$

THE TWO METHODS

Let a_i be the arity of f_i and define the map

$$\kappa_i : \{1, \ldots, a_i\} \mapsto \mathcal{I}_i \subset \{1, \ldots, i-1\},$$

so that

$$\tau_i = f_i(\tau_{\kappa_i 1}, \ldots, \tau_{\kappa_i a_i}).$$

The **FORWARD** method is given by:

initialize the values:

$\tau_i = x_i$, $\widehat{\tau}_{ij} = \delta_{ij}$, for $i, j = 1, \ldots m$.

compute:

for $i = m + 1$ to n,

$$\tau_i = f_i(\tau_{\kappa_i 1}, \ldots, \tau_{\kappa_i a_i}),$$

$$\widehat{\tau}_{i,j} = \sum_{k=1}^{a_i} \frac{\partial f_i}{\partial \tau_{\kappa_i k}} \widehat{\tau}_{\kappa_i k, j} \text{ for } j = 1, \ldots m.$$

$\widehat{\tau}_{i,j} = \frac{\partial \tau_i}{\partial \tau_j}$ for $i = 1, \ldots, n$, $j = 1, \ldots, m$ after the algorithm has stopped.

Differentiating $g(x) = 2xe^{x^2}\sin(e^{x^2})$

Using the forward method:

$$
\begin{array}{lcl@{\qquad}lcl}
\tau_1 &=& x & \widehat{\tau}_{1,1} &=& 1 \\
\tau_2 &=& \mathrm{sqr}(\tau_1) & \widehat{\tau}_{2,1} &=& 2\tau_1 \cdot \widehat{\tau}_{1,1} \\
\tau_3 &=& \exp(\tau_2) & \widehat{\tau}_{3,1} &=& \exp(\tau_2) \cdot \widehat{\tau}_{2,1} \\
\tau_4 &=& \sin(\tau_3) & \widehat{\tau}_{4,1} &=& \cos(\tau_3) \cdot \widehat{\tau}_{3,1} \\
\tau_5 &=& \tau_3 \cdot \tau_4 & \widehat{\tau}_{5,1} &=& \tau_4 \cdot \widehat{\tau}_{3,1} + \tau_3 \cdot \widehat{\tau}_{4,1} \\
\tau_6 &=& \tau_1 \cdot \tau_5 & \widehat{\tau}_{6,1} &=& \tau_5 \cdot \widehat{\tau}_{1,1} + \tau_1 \cdot \widehat{\tau}_{5,1} \\
\tau_7 &=& 2 & \widehat{\tau}_{7,1} &=& 0 \\
\tau_8 &=& \tau_7 \cdot \tau_6 & \widehat{\tau}_{8,1} &=& \tau_7 \cdot \widehat{\tau}_{6,1} + \tau_6 \cdot \widehat{\tau}_{7,1}
\end{array}
$$

Total 20 flops.

THE TWO METHODS

Let $\mathcal{D}$ be the set of indices of the dependent variables which is to be differentiated.

The **BACKWARD** method is given by:

initialize the forward sweep:

$\tau_i = x_i$, for $i = 1, \ldots m$.

forward sweep (function evaluation):

for $i = m + 1$ to n,

$\tau_i = f_i(\tau_{\kappa_i 1}, \ldots, \tau_{\kappa_i a_i})$,

initialize the reverse sweep:

$\widehat{\tau}_{i,j} = \delta_{ij}$ for $i \in \mathcal{D}$, $j = 1, \ldots, n$.

reverse sweep (function differentiation):

for $j = n$ downto $m + 1$,

$$\widehat{\tau}_{i,\kappa_j k} = \widehat{\tau}_{i,\kappa_j k} + \frac{\partial f_j}{\partial \tau_{\kappa_j k}} \widehat{\tau}_{i,j} \text{ for } i \in \mathcal{D},\ k = 1, \ldots, a_j.$$

$\widehat{\tau}_{i,j} = \frac{\partial \tau_i}{\partial \tau_j}$ for $i \in \mathcal{D}$, $j = 1, \ldots, n$ after the algorithm has stopped.

Differentiating $g(x) = 2xe^{x^2}\sin(e^{x^2})$

Using the backward method:

$$
\begin{array}{lcl}
\tau_1 & = & x \\
\tau_2 & = & \mathrm{sqr}(\tau_1) \\
\tau_3 & = & \exp(\tau_2) \\
\tau_4 & = & \sin(\tau_3) \\
\tau_5 & = & \tau_3 \cdot \tau_4 \\
 & & \\
\tau_6 & = & \tau_1 \cdot \tau_5 \\
 & & \\
\tau_7 & = & 2 \\
\tau_8 & = & \tau_7 \cdot \tau_6
\end{array}
\qquad
\begin{array}{lcl}
\widehat{\tau}_{8,1} & += & 2\tau_1 \cdot \widehat{\tau}_{8,2} \\
\widehat{\tau}_{8,2} & += & \exp(\tau_2) \cdot \widehat{\tau}_{8,3} \\
\widehat{\tau}_{8,3} & += & \cos(\tau_3) \cdot \widehat{\tau}_{8,4} \\
\widehat{\tau}_{8,4} & += & \tau_3 \cdot \widehat{\tau}_{8,5} \\
\widehat{\tau}_{8,3} & += & \tau_4 \cdot \widehat{\tau}_{8,5} \\
\widehat{\tau}_{8,5} & += & \tau_1 \cdot \widehat{\tau}_{8,6} \\
\widehat{\tau}_{8,1} & += & \tau_5 \cdot \widehat{\tau}_{8,6} \\
 & & \\
\widehat{\tau}_{8,6} & += & \tau_7 \cdot \widehat{\tau}_{8,8} \\
\widehat{\tau}_{8,7} & += & \tau_6 \cdot \widehat{\tau}_{8,8} \\
\widehat{\tau}_{8,8} & = & 1
\end{array}
$$

Total 26 flops.

The C++ package "FADBAD"

FADBAD is available from the WWW-page:

<http://www.imm.dtu.dk/~os/fadbad.html>

How it works:

- Overloading every elementary operation by changing the arithmetic type. eg.
  `INTERVAL` $\rightarrow \begin{cases} \texttt{FINTERVAL} \\ \texttt{BINTERVAL} \end{cases}$

- The computational Graph (DAG) is "recorded" using the backward method.

- Higher order derivatives are found by overloading the computations in the package ifself. eg.
  `FINTERVAL` $\rightarrow \begin{cases} \texttt{FFINTERVAL} \\ \texttt{BFINTERVAL} \end{cases}$

Main strategy:

- **Transparency:** Programs does not change functionality when differentiating them.

- **Flexibility:** Can be used with `INTERVAL`, `double`, etc.

Using the C++ package "FADBAD"

Example:

```
void main(){
int i,j;
FINTERVAL x[2],r;

 x[0]=3;x[1]=-5; // Initialize independent variables.

 x[0].diff(0,2); // Specify the variables to differentiate
 x[1].diff(1,2); // the computations with respect to.

 r=0;
 for(i=0;i<2;i++)
   for(j=1;j<=5;j++)          // Do some
     r+=j*sin((j+1)*x[i]+j); // computations

 cout << r.x() << endl   // Print the function value
      << r.d(0) << endl  // And the partial derivatives
      << r.d(1) << endl; //  of r wrt. x[0] and x[1].
}
```

Members of the forward type:

.x() Returns the value of the variable.

.diff(i,j) differentiate wrt. this variable which is the $(i+1)$th out of j. This operation should be performed on the *independent* variables *before* running the code which is to be differentiated.

.d(i) Returns derivative number i in the overloaded type.

Using the C++ package "FADBAD"

Example:

```
void main(){
int i,j;
BINTERVAL x[2],r;

 x[0]=3;x[1]=-5;      // Initialize independent variables.

 r=0;
 for(i=0;i<2;i++)
   for(j=1;j<=5;j++)          // Do some
     r+=j*sin((j+1)*x[i]+j); // computations

 r.diff(0,1); // Specify the variables to differentiate

cout << r.x() << endl       // Print the function value
     << x[0].d(0) << endl   // And the partial derivatives
     << x[1].d(0) << endl;  //  of r wrt. x[0] and x[1].
}
```

Members of the backward type:

.x() Returns the value of the variable.

.diff(i,j) differentiate this variable which is the $(i+1)$th out of j. This operation should be performed on *all* of the *dependent* variables *after* running the code which is to be differentiated.

.d(i) Returns derivative number i in the overloaded type.

Using the C++ package "FADBAD"

Using backward-forward differentiation:

```
void main(){
int i,j;
BFINTERVAL x[2],r;

 x[0]=3;x[1]=-5;     // Initialize independent variables

 x[0].x().diff(0,2); // compute second order derivatives
 x[1].x().diff(1,2); // with respect to x[0] and x[1].

 r=0;
 for(i=0;i<2;i++)
   for(j=1;j<=5;j++)         // Do some
     r+=j*sin((j+1)*x[i]+j);// computations

 r.diff(0,1);              // Specify the variables to
                           // differentiate.

 cout << r.x().x() << endl  // Print the function value.

  << x[0].d(0).x() << endl  // And the partial derivatives
  << x[1].d(0).x() << endl  //  of r wrt. x[0] and x[1].

  << x[0].d(0).d(0) << endl // And the second order partial
  << x[0].d(0).d(1) << endl // derivatives of r wrt.
  << x[1].d(0).d(0) << endl // x[0] and x[1].
  << x[1].d(0).d(1) << endl;

}
```