

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

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:

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

etc.

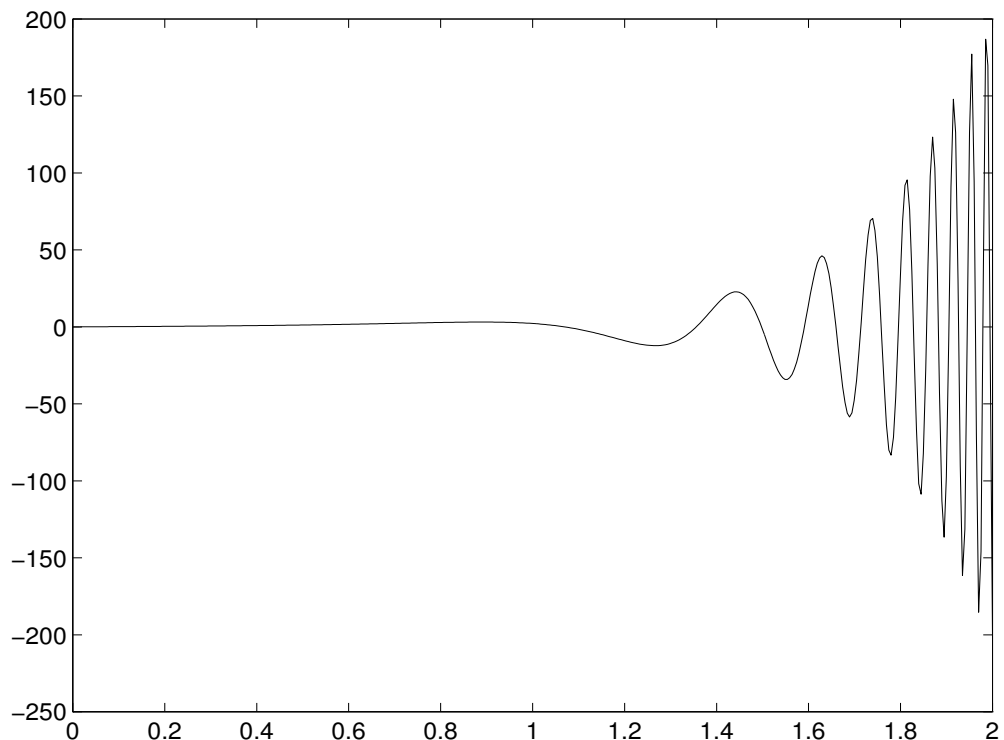
Most of time in Global Optimization goes to evaluating functions and their derivatives.

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(sqrt(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.
}
```

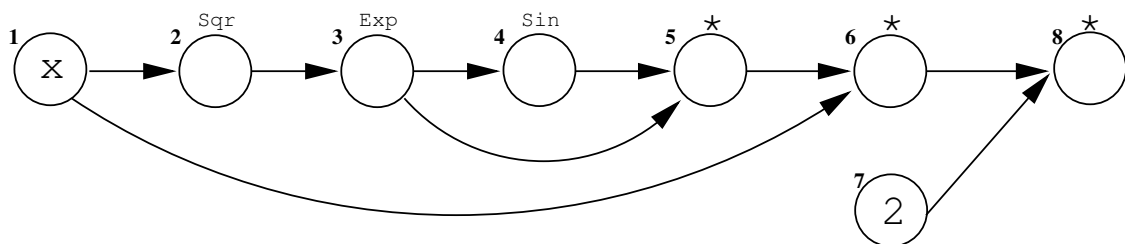
Quite efficient and only the function and not the derivatives has to be implemented!

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*:

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

PROBLEM FORMULATION

In general we have:

initialize the values:

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

compute:

for $i = m + 1$ to n ,

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

The “chain rule” for composite functions:

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

Or in matrix formulation:

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

where

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

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

OBTAINING THE MATRIX $D\tau$

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

From (1) we get: $I =$

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

From (2) we get: $I =$

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

THE TWO METHODS

Let a_i be the arity of f_i and define the map

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

so that

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

The **FORWARD** method is given by:

initialize the values:

$$\tau_i = x_i, \hat{\tau}_{ij} = \delta_{ij}, \text{ for } i, j = 1, \dots, m.$$

compute:

for $i = m + 1$ to n ,

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

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

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

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

Using the forward method:

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

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:

$$\tau_i = x_i, \text{ for } i = 1, \dots, m.$$

forward sweep (function evaluation):

for $i = m + 1$ to n ,

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

initialize the reverse sweep:

$$\hat{\tau}_{i,j} = \delta_{ij} \text{ for } i \in \mathcal{D}, j = 1, \dots, n.$$

reverse sweep (function differentiation):

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

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

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

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

Using the backward method:

$\tau_1 = x$		
$\tau_2 = \text{sqr}(\tau_1)$	$\hat{\tau}_{8,1} + = 2\tau_1 \cdot \hat{\tau}_{8,2}$	
$\tau_3 = \text{exp}(\tau_2)$	$\hat{\tau}_{8,2} + = \text{exp}(\tau_2) \cdot \hat{\tau}_{8,3}$	
$\tau_4 = \text{sin}(\tau_3)$	$\hat{\tau}_{8,3} + = \text{cos}(\tau_3) \cdot \hat{\tau}_{8,4}$	
$\tau_5 = \tau_3 \cdot \tau_4$	$\hat{\tau}_{8,4} + = \tau_3 \cdot \hat{\tau}_{8,5}$	
	$\hat{\tau}_{8,3} + = \tau_4 \cdot \hat{\tau}_{8,5}$	
$\tau_6 = \tau_1 \cdot \tau_5$	$\hat{\tau}_{8,5} + = \tau_1 \cdot \hat{\tau}_{8,6}$	
	$\hat{\tau}_{8,1} + = \tau_5 \cdot \hat{\tau}_{8,6}$	
$\tau_7 = 2$		
$\tau_8 = \tau_7 \cdot \tau_6$	$\hat{\tau}_{8,6} + = \tau_7 \cdot \hat{\tau}_{8,8}$	
	$\hat{\tau}_{8,7} + = \tau_6 \cdot \hat{\tau}_{8,8}$	
	$\hat{\tau}_{8,8} = 1$	

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.

$$\text{INTERVAL} \rightarrow \begin{cases} \text{FINTERVAL} \\ \text{BINTERVAL} \end{cases}$$

- The computational Graph (DAG) is “recorded” using the backward method.
- Higher order derivatives are found by overloading the computations in the package itself. eg.

$$\text{FINTERVAL} \rightarrow \begin{cases} \text{FFINTERVAL} \\ \text{BFINTERVAL} \end{cases}$$

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;

}
```