



# Flexible AD using templates and operator overloading in C++

---

August 2003 Ole Stauning  
([Ole.St@uning.dk](mailto:Ole.St@uning.dk))



# Agenda

---

1. Background
2. Introduction to FADBAD++
3. Advanced uses of FADBAD++
4. Conclusion & Discussion



# BACKGROUND

---



# Importance of derivatives

---

- Derivatives can be used to obtain narrow interval enclosures compared to simple interval evaluations. E.g. by using the (extended) mean value form.
- Used in interval Newton or Krawczyk to prove existence to non-linear systems of equations.
- Taylor expansion is used to solve ODE's.
- Divided differences does not enclose the true derivatives.



# Our design goals

---

We had the following goals when developing FADBAD++:

- Should be straightforward and easy to use.
- Should be flexible.
- Should blend into the C++ syntax as well as possible.

Performance and support for large-scale calculations is less important.

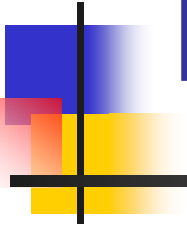


# Current status

---

- Implemented with operator overloading and templates in C++.
- No library files, only header files containing templates.
- Works with GCC 3.2, VC++ 6.0, .NET and other mature compilers.
- Implements Forward-, Backward-, Taylor methods and combinations.
- Differentiates functions based on doubles, intervals or other arithmetic types.
- Existing C++ code can easily be "differentiated" with search/replace.
- No code-changes needed for C++ code which is AD-prepared.

# INTRODUCTION TO FADBAD++





# Using the forward method

---

We start with the function we want to differentiate:

```
double func(const double& x, const double& y)
{
    double z=sqrt(x);
    return y*z+sin(z);
}
```

Replace occurrences of "double" with "F<double>":



# Using the forward method

---

Function evaluated with `F<double>` instead of `double`:

```
F<double> func(const F<double>& x, const F<double>& y)
{
    F<double> z=sqrt(x);
    return y*z+sin(z);
}
```

Calling `func` will now compute derivatives as well as the function value itself.



# Using the forward method

---

```
F<double> x,y,f;    // Declare variables x,y,f
x=1;               // Initialize variable x
x.diff(0,2);       // Differentiate with respect to x (index 0 of 2)
y=2;               // Initialize variable y
y.diff(1,2);       // Differentiate with respect to y (index 1 of 2)
f=func(x,y);       // Evaluate function and derivatives
double fval=f.x(); // Value of function
double dfdx=f.d(0); // Value of df/dx (index 0 of 2)
double dfdy=f.d(1); // Value of df/dy (index 1 of 2)
```



# Using the backward method

---

```
B<double> x,y,f;           // Declare variables x,y,f
x=1;                       // Initialize variable x
y=2;                       // Initialize variable y
f=func(x,y);              // Evaluate function and record DAG
f.diff(0,1);              // Differentiate function (index 0 of 1)
double fval=f.x();        // Value of function
double dfdx=x.d(0);      // Value of df/dx (index 0 of 1)
double dfdy=y.d(0);      // Value of df/dy (index 0 of 1)
```



# The backward- and forward methods

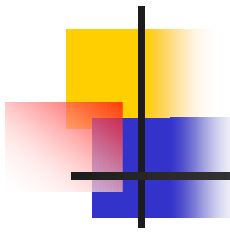
---

The forward method:

- Call `.diff` on the *independent* variables, *before* the function evaluation.
- Function values and derivatives are obtained by calling `.x` and `.d` on the *dependent* variables *after* the function evaluation.
- Function values and derivatives are calculated in one pass.

The backward method:

- Function values are obtained *after* the function evaluation by calling `.x` on the dependent variables.
- Call `.diff` on the *dependent* variables *after* the function evaluation.
- The values of the derivatives are obtained by calling the d-method on the independent variables.
- Function values are calculated in a forward- and derivatives in a backward pass.
- Memory consuming; records DAG.

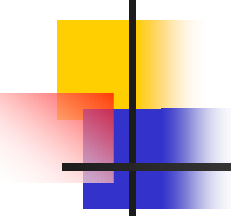


# Taylor expansion in FADBAD++

---

Expanding a function given explicit:

```
T<double> x,y,f;           // Declare variables x,y,f  
x=1;                       // Initialize variable x  
y=2;                       // Initialize variable y  
x[1]=1;                    // Taylor-expand wrt. x (dx/dx=1)  
f=func(x,y);              // Evaluate function and record DAG  
double fval=f[0];         // Value of function  
f.eval(10);               // Taylor-expand f to degree 10  
// f[0]...f[10] now contains the Taylor-coefficients.
```

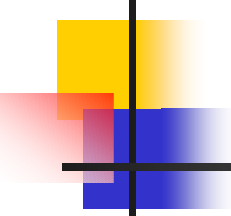


# Taylor expansion in FADBAD++

---

Expanding a solution to an ODE;  $x=f'(x)$

```
class TODE
{
    public:
    T<double> x;           // Independent variables
    T<double> xp;         // Dependent variables
    TODE() { xp=cos(x); } // record DAG at construction
};
```



# Taylor expansion in FADBAD++

---

Expanding a solution to an ODE;  $x=f'(x)$

```
TODE ode; // Construct ODE:  
ode.x[0]=1; // Set point of expansion:  
for(int i=0;i<10;i++)  
{  
  ode.xp.eval(i); // Evaluate i'th Taylor coefficient  
  ode.x[i+1]=ode.xp[i]/double(i+1); // Use dx/dt=ode(x).  
}  
// ode.x[0]...ode.x[10] now contains the Taylor-coefficients  
// to the solution of the ODE.
```



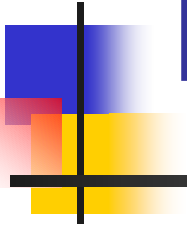
# The Taylor-Forward-Interval type

---

The AD-type `T< F< INTERVAL >>` is used in ADIODES++ to Taylor expand the solution of an ODE and compute derivatives of the Taylor coefficients with respect to the point of expansion.

The derivatives are the Taylor coefficients of the solution to the variational equation.

# ADVANCED USE OF FADBAD++





# Obtaining higher order derivatives

---

We start with the function we want to differentiate:

```
double func(const double& x, const double& y)
{
    double z=sqrt(x);
    return y*z+sin(z);
}
```

Replace occurrences of "double" with "B<double>":



# Obtaining higher order derivatives

---

Function evaluated with B<double> instead of double:

```
B<double> func(const B<double>& x, const B<double>& y)
{
    B<double> z=sqrt(x);
    return y*z+sin(z);
}
```

Now we can write a function dfunc for obtaining first order derivatives.

# Obtaining higher order derivatives

```
double dfunc( double& o_dfdx, double& o_dfdy,
              const double& i_x, const double& i_y )
{
    B<double> x(i_x),y(i_y);           // Initialize arguments
    B<double> f(func(x,y));           // Evaluate function and record DAG
    f.diff(0,1);                       // Differentiate
    o_dfdx=x.d(0);                     // Value of df/dx
    o_dfdy=y.d(0);                     // Value of df/dy
    return f.x();                       // Return function value
}
```

To obtain the second order derivatives we replace all occurrences of "double" with "B<double>" in func and dfunc.

# Obtaining higher order derivatives

```
B< F<double> > func(
    const B< F<double> >& x, const B< F<double> >& y )
{
    B< F<double> > z=sqrt(x);
    return y*z+sin(z);
}
F<double> dfunc( F<double>& o_dfdx, F<double>& o_dfdy,
    const F<double>& i_x, const F<double>& i_y )
{
    B< F<double> > x(i_x),y(i_y); // Initialize arguments
    B< F<double> > f(func(x,y)); // Evaluate function and record DAG
    f.diff(0,1); // Differentiate
    o_dfdx=x.d(0); // Value of df/dx
    o_dfdy=y.d(0); // Value of df/dy
    return f.x(); // Return function value
}
```





# Obtaining higher order derivatives

---

We can differentiate code by search-and-replace the underlying type and this way produce our own AD-type.

But do we really need to search-and-replace?

The answer is NO!

We can parametrize our functions instead of using a fixed type. The compilers will then instantiate the right type.



# What is a Functor?

---

A Functor is an object that can be evaluated as a function.

```
class Func
{
    public:
    template <class T>
    T operator()(const T& x, const T& y)
    {
        T z=sqrt(x);
        return y*z+sin(z);
    }
};
```

The compiler will instantiate **operator()** for each occurrence of the type T.



# Forward AD Functor

---

```
template <class C> class Fdiff
{
public:
  template <class T>
  T operator()( T& o_dfdx, T& o_dfdy, const T& i_x, const T& i_y )
  {
    F<T> x(i_x),y(i_y);           // Initialize arguments
    x.diff(0,2);                  // Differentiate wrt. X
    y.diff(1,2);                  // Differentiate wrt. Y
    C func;                       // Instantiate functor
    F<T> f(func(x,y));           // Evaluate function and derivatives
    o_dfdx=f.d(0);               // Value of df/dx
    o_dfdy=f.d(1);               // Value of df/dy
    return f.x();                // Return function value
  }
};
```



# Backward AD Functor

---

```
template <class C> class Bdiff
{
public:
  template <class T>
  T operator()( T& o_dfdx, T& o_dfdy, const T& i_x, const T& i_y )
  {
    B<T> x(i_x),y(i_y);           // Initialize arguments
    C func;                       // Instantiate functor
    B<T> f(func(x,y));           // Evaluate function and record DAG
    f.diff(0,1);                 // Differentiate
    o_dfdx=x.d(0);              // Value of df/dx
    o_dfdy=y.d(0);              // Value of df/dy
    return f.x();                // Return function value
  }
};
```



# Differentiating a Functor

---

The classes FDiff for the forward- and BDiff for the backward method are functors themselves and they share a common interface:

Using the Forward method:

```
FDiff<Func> FFunc;    // Functor for function and derivatives  
f=FFunc(dfdx,dfdy,x,y); // Evaluate function and derivatives
```

Using the Backward method:

```
BDiff<Func> BFunc;    // Functor for function and derivatives  
f=BFunc(dfdx,dfdy,x,y); // Evaluate function and derivatives
```



# AD-Functors summary

---

- We do not need to know the AD-type in advance.
- Easy to change AD-type at a later stage.
- Other people can differentiate your code without changing it.
- Easier to read instead of explicitly writing the AD-type.
- Their interfaces are independent of the AD method used.
- AD-functors can be combined. E.g. `FDiff< BDiff< Func > >` to compute higher order derivatives.



# CONCLUSION & DISCUSSION

---



# Conclusion

---

Automatic differentiation can be implemented in C++ in a transparent and generic way by using operator overloading and templates.

FADBAD++ provides three templates  $F\langle \rangle$ ,  $B\langle \rangle$ ,  $T\langle \rangle$  for forward-, backward- and Taylor methods.

- Existing code can be differentiated by search/replace. No code-changes are necessary for code that have been prepared for AD, e.g. by using Functors.
- Flexible since the underlying arithmetic type can be determined at compile-time and the underlying type can be another AD-type.
- Easy to use since technicalities is hidden from the user.
- General purpose, since three basic ways of differentiating and combinations can be used.



# Questions?

---

Homepage: <http://www.imm.dtu.dk/fadbad>

Email: Ole.St@uning.dk