Automatic Differentiation Clad and Clang

In mathematics and computer algebra, automatic differentiation (AD) is a set of techniques to evaluate the derivative of a function specified by a computer program. That is, AD takes the source code of a function as input and produces the source code of the derived function. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.), elementary functions (exp, log, sin, cos, etc.), and control flow statements. By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, with an accurately limited by the working precision, and using at most a small constant factor more arithmetic operations than the original program.

Clad is a source transformation AD tool for C++ [2, 3]. It is based on LLVM compiler infrastructure and is implemented as a plugin for Clang, which allows Clad to be integrated into the compilation phase, and to utilize large parts of the compiler itself. Clad relies on Clang’s parsing and code generation functionalities and can differentiate complex C++ constructs in both forward and reverse mode. It is available as a standalone Clang plugin that, when attached to the compiler, produces derivatives in the compilation phase.

Jacobian Matrix. Results

The Jacobian matrix is a vector-valued function with several dependent variables generalizes the gradient of a scalar-valued function in several variables, which in turn generalizes the derivative of a scalar-valued function of a single variable. That is, the Jacobian of a scalar-valued function (the transpose of its gradient) and the gradient of a scalar-valued function is its derivative. The Jacobian matrix can also be thought of as describing the amount of ‘stretching’, ‘rotating’ or ‘transforming’ that the function implies. For example, if (x,y) → (f(x),g(x)) is used to smoothly transform an image, the Jacobian matrix Jx.y describes how the image in the neighborhood of (x,y) is transformed.

Clad operates on Clang AST (abstract syntax tree) by analyzing the original function and generating the AST of the derivative. Clad provides the API functions: clad:: differentiate for forward mode, clad::jacobian for reverse mode, and clad::gradient and clad::hessian for mixed mode [3].

The Jacobian matrix of a vector-valued function with several dependent variables generalizes the gradient of a scalar-valued function in several variables, which in turn generalizes the derivative of a scalar-valued function of a single variable. That is, the Jacobian of a scalar-valued function (the transpose of its gradient) and the gradient of a scalar-valued function is its derivative. The Jacobian matrix can also be thought of as describing the amount of ‘stretching’, ‘rotating’ or ‘transforming’ that the function implies. For example, if (x,y) → (f(x),g(x)) is used to smoothly transform an image, the Jacobian matrix Jx.y describes how the image in the neighborhood of (x,y) is transformed.

AD accuracy

Numerical differentiation (ND) may give imprecise results, while AD computes the derivatives accurately. We show an example of a function where this difference is apparent:

\[
J = \left[ \begin{array}{ccc} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \end{array} \right]
\]

The function is the PDF of Breit-Wigner distribution (Eq. 1), whose derivative with respect to \( \Gamma \) (Eq. 2) has critical points at \( \pm \Delta \). ND produces the exact result whereas SD fails from the loss of accuracy. The function can be implemented as in (Listing 1).

Listing 1: Example Breit-Wigner PDF implementation

```c
#include<clad/Differentiator/Differentiator.h>

double f1(double x, double y){
    return 2 * (x - 2) * y;
}

double jaco[3][3];

double f1(double x, double y){
    return 2 * (x - 2) * y;
}

int main(){
    double x[3][3];
    for(int i=0; i<3; i++)
        for(int j=0; j<3; j++)
            x[i][j] = 1.0;

    return 0;
}
```

Listing 2: Illustrative code examples of Clad computed numerically and using automatic differentiation.

The usual ND implementation is separated from the target functions which prevents excessive optimizations. ND shows very good results in O3 mode due to inlining which is generally not the case for production codes because often nd_jaco does not see the definitions of the target functions. Clad can fully integrate the derivative code to close its use and allows the optimizers to heavily optimize it. The expected algorithmic complexity of ND is \( O(N^2) \). Forward AD -- \( O(N^2) \). Reverse AD -- \( O(NN) \) and Mixed AD -- \( O(1) \). This is confirmed by the performance results also in Figure 1.

Conclusion

The AD systems provide powerful techniques to decompose the computation graphs. It provides opportunities evaluating derivatives faster. An AD tool implemented in the compiler, such as Clad, provides opportunities beyond computing derivatives. The implementation of Clad permits development of a generic error estimation framework which is not bound to a particular error approximation model. It should allow users to select their preferable estimation logic and should automatically generate functions augmented with code for the specified error estimation.

References


\[
y = f(a) + f'(a)(x-a) + \frac{f''(a)}{2} (x-a)^2 + \cdots + \frac{f^{(n)}(a)}{n!} (x-a)^n + \cdots
\]

The round-off error at point \( a \) is affected by the input error (\( \Delta a \)) and the first derivative (Eq. 4). Generalizing it for vector valued functions where the input variables and their intermediaries contribute to the final round-off error (Eq. 5).

\[
\Delta y = \sum_{i=1}^{N} \frac{\partial f}{\partial x_i} \Delta x_i
\]

Clad’s design allows producing Jacobians of the inspected code without modifications and makes it very suitable for implementing the model. Estimating the error of the algorithm for approximating \( a \) is straightforward.

\[
\text{Error} = |f(a + \Delta a) - f(a)|
\]

The mechanism for error estimation shows that one of the algorithms on the rhs shows numerical instability at point 1.5 and its round-off error is around 2 times greater than the one in it’s fixed counterpart on the rhs.

Error Estimation

Figure 1: Performance comparison of Jacobian computed numerically and using Clad

Figure 2: Error estimation of two algorithms approximating \( a \) using Jacobian

The mechanism for error estimation shows that one of the algorithms on the rhs shows numerical instability at point 1.5 and its round-off error is around 2 times greater than the one in it’s fixed counterpart on the rhs.