Error estimates of floating-point numbers and Jacobian matrix computation in Clad

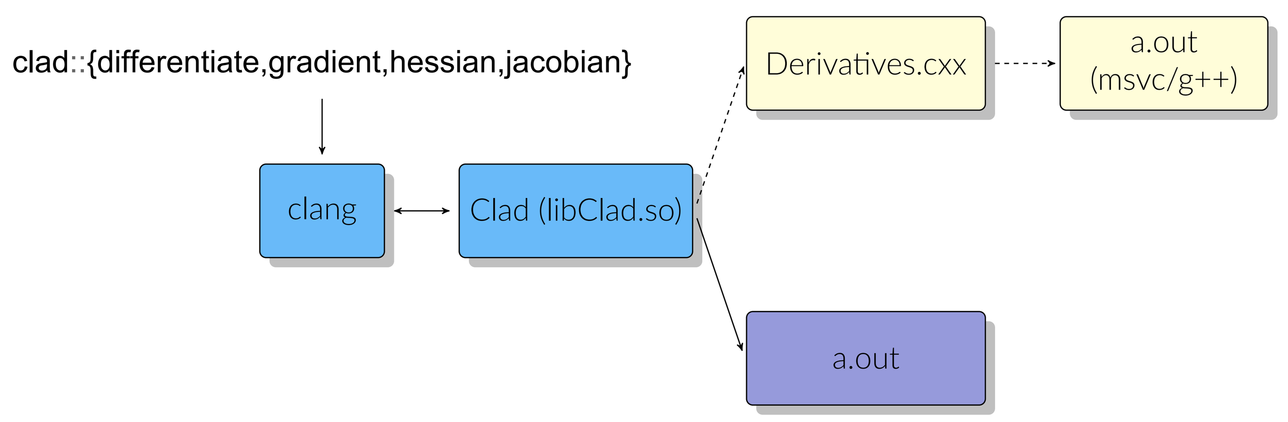
Vassil Vassilev [1] Alexander Penev [2] Roman Shakhov [3]

[1]Princeton University [2]University of Plovdiv ``Paisii Hilendarski'' [3]University of Voronezh

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 C++ compiler Clang, which allows Clad to be transparently 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 complicated 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.



Clad operates on the *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**::**gradient** for reverse mode, **clad**::**jacobian** 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 in several variables is (the transpose of) its gradient and, the gradient of a scalar-valued function of a single variable is its derivative. The Jacobian matrix can also be thought of as describing the

$$\mathbf{J} = \left[\frac{\partial \mathbf{f}}{\partial x_1} \cdots \frac{\partial \mathbf{f}}{\partial x_n}\right] = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}.$$

amount of "stretching", "rotating" or "transforming" that the function implies. For example, if $(x, y) = f(x, y)$ is used to smoothly transform an image, the Jacobian matrix $Jf(x, y)$, 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:

$$p(x) = \frac{1}{\pi}\frac{\frac{1}{2}\Gamma}{x^2 + (\frac{1}{2}\Gamma)^2} \quad (1) \qquad \frac{\partial p(x)}{\partial \Gamma} = -\frac{2}{\pi}\frac{\Gamma^2 - 4x^2}{(\Gamma^2 + 4x^2)^2} \quad (2)$$

The function is the PDF of *Breit-Wigner* distribution (Eq. 1), whose derivative with respect to Γ (Eq. 2) has critical points at $\Gamma = \pm 2x$. AD provides exact result while ND suffers from the loss of accuracy. The function can be implemented as in (Listing 1).

```
inline double breitwigner_pdf(double x, double gamma, double x0 = 0) {
  double gammahalf = gamma/2.0;
  return gammahalf/(M_PI * ((x-x0)*(x-x0) + gammahalf*gammahalf));
}
```

Listing 1: Example Breit-Wigner PDF implementation

When evaluating the derivative of **breitwigner_pdf** with respect to **gamma** at **x**=1, **gamma**=2, ND the yields a result close to 0 with an absolute error of 10^{-13}, even though the function is smooth and well-conditioned at this point. The approximation error becomes larger when the derivative is evaluated further from the critical point. In contrast, AD yields *exact* result of 0.

Jacobian Matrix. Results

The computational cost of evaluating the Jacobian matrix is also relatively high for numeric differentiation. It requires $2N^2$ function invocations for square Jacobian of N (**nd_jaco** in Listing 2). AD provides techniques to reduce the computational cost by reducing the function calls or computing the Jacobian in a single pass.

Figure 1 represents a performance benchmark of 4 different ways to obtain a Jacobian -- one numeric and three based on AD. The numeric computation of the partial derivatives uses the finite differences method and requires two evaluations per direction (other numerical approaches make more evaluations to improve numerical stability). The AD Jacobian can be computed using AD generated derivatives: in forward mode (with N^2 evaluations) and reverse mode (with N evaluations)

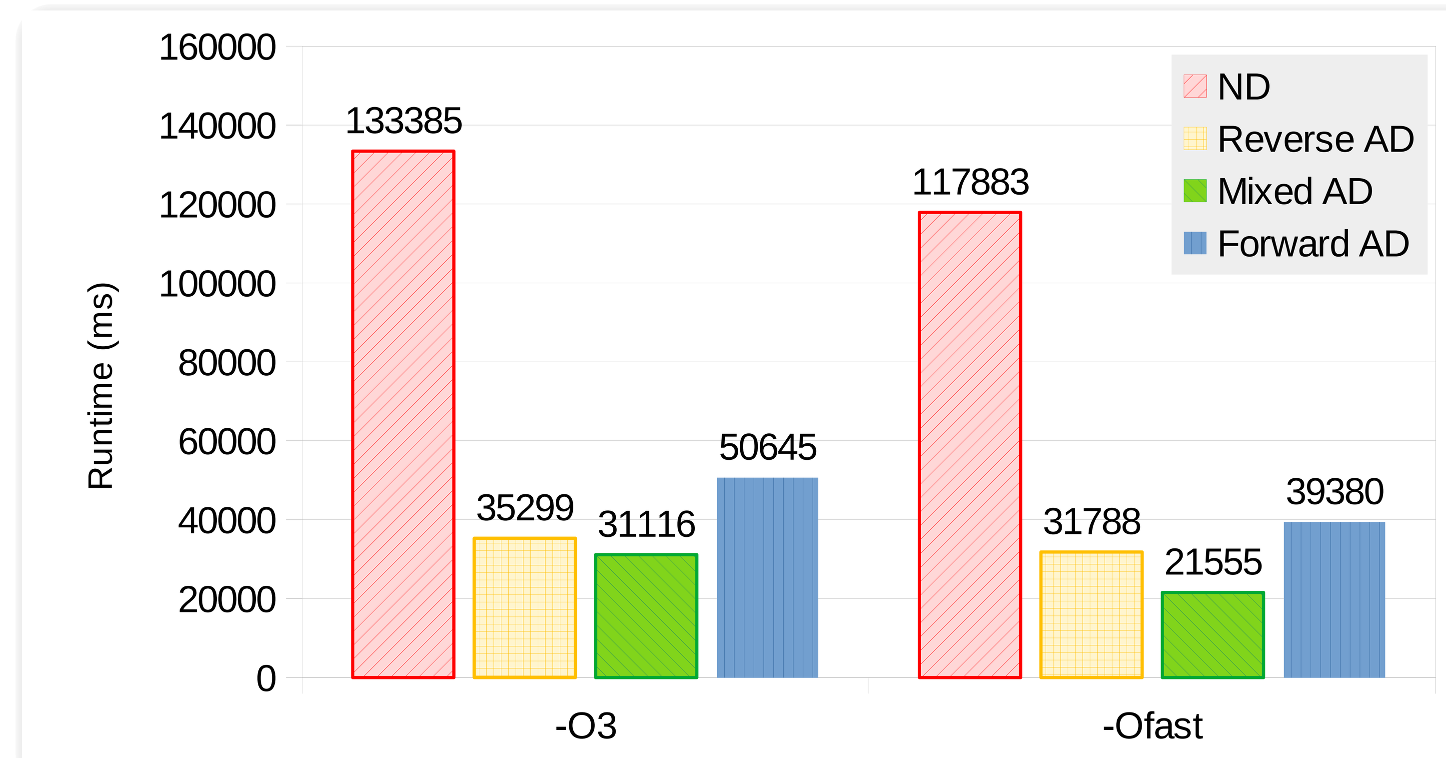


Figure 1: Performance comparison of Jacobian computed numerically and using Clad

and mixed mode (single evaluation). Listing 2 shows a part of the benchmark. The *lhs* shows the non-linear transformations **f1** and **f2** which participate in the computed Jacobian. The *rhs* shows the skeleton of the generated code by Clad.

```
#include <clad/Differentiator/Differentiator.h>
double f1(double x, double y) {
  return 2 * x * y;
}
double f2(double x, double y) {
  return x * x + y * y;
}
void fn(double x, double y, double res[2]) {
  res[0] = f1(x, y); res[1] = f2(x, y);
}

void nd_jaco(double x, double y,
             double j[4]) {
  const double E = 1e-8;
  j[0] += (f1(x + E, y) - f1(x - E, y)) / (2 * E);
  j[1] += (f1(x, y + E) - f1(x, y - E)) / (2 * E);
  j[2] += (f2(x + E, y) - f2(x - E, y)) / (2 * E);
  j[3] += (f2(x, y + E) - f2(x, y - E)) / (2 * E);
}

int main() {
  // double x=, y=
  auto clad_jaco = clad::jacobian(fn);
  double res[2]; double jaco[4];
  clad_jaco.execute(x, y, res, jaco);
  nd_jaco(x, y, jaco);
}
```

```
/*Generated Code. Simplified for expository reasons*/
double f1_darg0(double x, double y) { /* ... */ }
double f1_darg1(double x, double y) { /* ... */ }
double f2_darg0(double x, double y) { /* ... */ }
double f2_darg1(double x, double y) { /* ... */ }

void f1_grad(double x, double y, double _res[2]) { /* ... */ }
void f2_grad(double x, double y, double _res[2]) { /* ... */ }

void fn_jac(double x, double y, double res[2], double j[4]) {
  double _t0 = x; double _t1 = y;
  double _t2 = x; double _t3 = y;
  res[0] = f1(x, y); res[1] = f2(x, y);
    double _jac1[2] = {};
    f2_grad(_t2, _t3, _jac1);
    double _r2 = 1 * _jac1[0UL];
    j[2UL] += _r2;
    double _r3 = 1 * _jac1[1UL];
    j[3UL] += _r3;

    double _jac0[2] = {};
    f1_grad(_t0, _t1, _jac0);
    double _r0 = 1 * _jac0[0UL];
    j[0UL] += _r0;
    double _r1 = 1 * _jac0[1UL];
    j[1UL] += _r1;
}
```

Listing 2: Illustrative code examples of Jacobian computed using numerical and 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 put the derivative code close to its use and allows the optimizers to heavily optimize it. The expected algorithmic complexity of ND is $O(2{*}N^2)$, Forward AD -- $O(N^2)$, Reverse AD -- $O(N)$ and Mixed AD -- $O(1)$. This is confirmed by the performance results also in Figure 1.

Error Estimation

Estimating floating point computation errors is as important as the computation itself. Accurate error estimation requires processing the code, arithmetic operations, and assignments for each input variable and dependent intermediate ones. It is virtually impossible to make an accurate error estimation by hand. In cases where it is possible, it can make the code less readable and maintainable.

The AD technology decomposes the computation graph into atomic operations, which can then be used to follow differential calculus rules to produce a derivative. Adding a set of extra rules to estimate the floating point error for automatic differentiated functions is straightforward. Moreover, generalizing this towards automatically estimating errors of any computation can offer a way to reducing the used precision bits and a way forward towards implementing lossy compression.

Using Jacobians to estimate errors

Menon et al [1] describe a prominent error estimation model is using Taylor series estimation (Eq. 3).

$$y = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \cdots \approx f(a) + f'(a)(x-a) \qquad (3)$$

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

$$\Delta y = |f(a + \Delta x) - f(a)| \approx |f'(a) \cdot \Delta x| \quad (4) \qquad y = f(x_1, ..., x_n), \quad \Delta y \approx \sum_{i=1}^{n} \left|\frac{\partial f}{\partial x_i}(a_1, ..., a_n) \cdot \Delta x_i\right| \quad (5)$$

Finally, the error estimator of a vector valued function is the scalar product of the Jacobian and the vector of the errors of the input variables (6).

$$\Delta y \approx \left[\left|\frac{\partial f}{\partial x_1}\right| \left|\frac{\partial f}{\partial x_2}\right| \cdots \left|\frac{\partial f}{\partial x_n}\right|\right]^T \cdot \left[|\Delta x_1| \; |\Delta x_2| \; \cdots \; |\Delta x_n|\right] = J_f^T \cdot \left[|\Delta x_1| \; |\Delta x_2| \; \cdots \; |\Delta x_n|\right] \qquad (6)$$

Clad's design allows producing Jacobians of the inspected code without modifications by hand and makes it very suitable for implementing the model. Estimating the error of the described in [1] algorithm for approximating π is straightforward.

```
double fn(double Sn) {
  double e = Sn * Sn;
  double tmp = std::sqrt(4 - e);
  double Sn1 = std::sqrt(2 - tmp);
  return Sn1;
}
// Error at Sn=1.5: 6.889822e-09
```

```
double fn_stable(double Tn) {
  double e = Tn * Tn;
  double tmp = std::sqrt(4 + e);
  double Tn1 = 2 * Tn / (2 + tmp);
  return Tn1;
}
// Error at Tn=1.5: 3.555556e-09
```

Listing 3: Round-off error extimation of two algorithms approximating π using Jacobian

The mechanism for error estimation shows that one of the algorithms on the *lhs* 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*.

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 estimator.

References

[1] Harshitha Menon, Michael Lam, D Kuffour, Markus Schordan, S Llyod, Kathryn Mohror, and Jeff Hittinger. Adapt: Algorithmic differentiation for floating-point precision tuning. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2018.

[2] V Vassilev, M Vassilev, A Penev, L Moneta, and V Ilieva. Clad -- automatic differentiation using clang and llvm. In *Journal of Physics: Conference Series*, volume 608, page 012055. IOP Publishing, 2015.

[3] Vassil Vassilev. Clad -- automatic differentiation for C/C++. https://github.com/vgvassilev/clad/, 2014.