Automatic Differentiation in C++

Vassil Vassilev (Princeton U) <v.g.vassilev@gmail.com>; Marco Foco (NVIDIA) <mfoco@nvidia.com>
Function Derivatives

Many nonlinear optimization techniques exploit gradient and curvature information about the target and constraint functions being calculated.

Derivatives also play a key role in sensitivity analysis (model validation), inverse problems (data assimilation) and simulation (design parameter choice).

The derivative of a function of a single variable at a chosen input value, when it exists, is the slope of the tangent line to the graph of the function at that point.

[Wikipedia, The tangent line at \((x, f(x))\)]
Gradient-based optimization

Gradient descent:

$$x_{i+1} = x_i - \alpha \nabla f(x_i)$$

Applications:

- Function minimization
- Backpropagation for machine learning
- Fitting models to data

[Wikipedia, Gradient descent]
Backpropagation in ML

\[ y = \phi \left( \sum_i w_i x_i \right) \]

\[ E = \frac{1}{2} (t - y)^2 \]

\[ \Delta w_i = \eta \frac{\partial E}{\partial w_i} \]

\( x_i, w_i, \phi, \eta \) are inputs, input weights, activation function and learning rate of the neuron.

The error propagates back, through updates of the subtracted gradient ratio from the weights.

Training pattern is fed, forward generating corresponding output.

Error at output, the error between observed and desired state. Computed from the output \( y \) and seen desired output \( t \).
Computing Derivatives

Numerical Differentiation
- Precision loss, Rounding error problems
- Higher order and partial derivatives problems

Symbolic Differentiation
- Slow, Requires conversion of the program to a single expression, Requires closed-form expressions limiting algorithmic control flow and expressivity
- Higher order and partial derivatives problems

Algorithmic Differentiation
- Fixes all of the issues above at the cost of introducing extra software dependencies

\[ f'(x) \approx \frac{f(x+h) - f(x)}{h} \]
Algorithmic/Automatic differentiation [1/2]

- Creates a function that computes the derivative(s) for you by program by replacing the domain of the variables to incorporate derivative values and redefining the semantics of the operators to propagate derivatives per the chain rule of differential calculus.
- Alternative to numerical differentiation

\[ f'(x) \approx \frac{f(x+h) - f(x)}{h} \]

```c
double f(double x) {
    return x * x;
}
```

```c
double f_darg0(double x) {
    return 1*x + x*1;
}
```
Algorithmic/Automatic differentiation [2/2]

- **Benefits:** without additional precision loss
- **Benefits:** not limited to closed-form expressions
- **Benefits:** can take derivatives of algorithms (conditionals, loops, recursion)
  - Without inefficiently long expressions
- Implementations based on operator overloading/source transformation
Dual Numbers

- Forward Automatic Differentiation can be expressed in terms of Dual Numbers:
  - $a, b \in \mathbb{R}$, $\varepsilon \notin \mathbb{R}$, $\varepsilon^2 = 0 \Rightarrow a + \varepsilon b$ is a dual number

- Properties ($a, b, c, d \in \mathbb{R}$):

<table>
<thead>
<tr>
<th>Interactions with $\mathbb{R}$</th>
<th>Interactions between dual numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(a+\varepsilon b) + c = (a+c) + \varepsilon b$</td>
<td>$(a+\varepsilon b) + (c+\varepsilon d) = (a+c) + \varepsilon (b+d)$</td>
</tr>
<tr>
<td>$(a+\varepsilon b) \cdot c = (ac) + \varepsilon (bc)$</td>
<td>$(a+\varepsilon b) \cdot (c+\varepsilon d) = (ac) + \varepsilon (ad+bc) + \varepsilon^2 bd$</td>
</tr>
<tr>
<td>$(a+\varepsilon b)^2 = a^2 + \varepsilon (ab + ba) + \varepsilon^2 b^2 = a + 2\varepsilon (ab)$</td>
<td>$(a+\varepsilon b)^n = a^n + \varepsilon (a\ldots ab + aba\ldots a + a\ldots ab) + \ldots = a^n + \varepsilon (na^{n-1}b)$</td>
</tr>
</tbody>
</table>
## AD with Dual Numbers 1/2

<table>
<thead>
<tr>
<th>f(x)</th>
<th>f(z) = f(x+\epsilon)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>x</td>
<td>x + \epsilon</td>
</tr>
<tr>
<td>x+a</td>
<td>(x + a) + \epsilon</td>
</tr>
<tr>
<td>ax</td>
<td>ax + \epsilon a</td>
</tr>
<tr>
<td>x^n</td>
<td>x^n + \epsilon (nx^{n-1})</td>
</tr>
</tbody>
</table>
AD with Dual Numbers 2/2

And finally...

\[ f(z) = \sum_{k=0}^{n} a_k z^k = a_0 + \sum_{k=1}^{n} a_k z^k \]

\[ = a_0 + \sum_{k=1}^{n} a_k (x + \varepsilon)^k \]

\[ = \left( a_0 + \sum_{k=1}^{n} a_k x^k \right) + \varepsilon \sum_{k=1}^{n} a_k k z^{k-1} \]

\[ = \sum_{k=0}^{n} a_k z^k + \varepsilon \sum_{k=0}^{n} a_{k+1} (k + 1) z^k \]

\[ = f(x) + \varepsilon f'(x) \]
AD with Dual Numbers - Code

```cpp
auto f = [] (auto x, auto y) { return x*x+y*x+y*y; };

auto dfdx = [&f](double x, double y) {
    return f(dual{x, 1.0}, y).eps();
};

auto dfdy = [&f](double x, double y) {
    return f(x, dual{y, 1.0}).eps();
};
```
Clad: Clang C/C++ AD plugin

Clad* enables automatic differentiation (AD) for C/C++. It is based on the LLVM compiler infrastructure and is a plugin for Clang compiler.

- Improve numerical stability and correctness
- Replace iterative algorithms computing gradients with a single function call (of a compiler-generated routine)
- Provide an alternative way of gradient computations

* https://github.com/vgvassilev/clad
Clad: Source Transformation

- Clad performs **automatic differentiation** on C++ functions
- For a C++ function, creates another C++ function that computes its derivative(s)

```cpp
double f(double x) {
    return x * x * x;
}
```

```cpp
double f_darg0(double x) {
    return 1 * x * x + x * 1 * x + x * x * 1;
}
```
Clad: Implementation

double f(double x) {
    return x * x;
}

- Clad is a Clang compiler plugin
- Performs C++ source code transformation
- Operates on Clang AST (Clang Abstract Syntax Tree)
- AST transformation with `clang::StmtVisitor`
AD Transformation. Chain Rule

\[ y = f(g(h(x))) = f(g(h(w_0))) = f(g(w_1)) = f(w_2) = w_3 \]

\[ w_0 = x \]
\[ w_1 = h(w_0) \]
\[ w_2 = g(w_1) \]
\[ w_3 = f(w_2) = y \]

\[ \frac{dy}{dx} = \frac{dy}{dw_2} \frac{dw_2}{dw_1} \frac{dw_1}{dx} \]

The chain rule for differential calculus gives us nice visitation properties.
Forward mode

Consider: \( f(x_1, x_2) = \sin(x_1) + x_1 x_2 \)

```cpp
f(x1, x2) {
    x1 = x1
    x2 = x2
    a = x1 * x2
    b = \sin(x1)
    return a + b;
}

f_dx1(x1, x2) {
    dx1 = 1
    dx2 = 0
    da = dx1*x2 + x1*dx2
    db = \cos(x1) * dx1
    return da + db
}

f_dx2(x1, x2) {
    dx1 = 0
    dx2 = 1
    da = dx1*x2 + x1*dx2
    db = \cos(x1) * dx1
    return da + db
}
```
Forward mode

- Forward mode AD algorithm computes derivatives w.r.t. any (single) variable

\[ f(x_1, x_2) = \sin(x_1) + x_1 x_2 \]

[Wikipedia, Automatic differentiation]
double f_cubed_add1(double a, double b) {
    return a * a * a + b * b * b;
}

double f_cubed_add1_darg0(double a, double b) {
    double _d_a = 1;
    double _d_b = 0;
    double _t0 = a * a;
    double _t1 = b * b;
    return (_d_a * a + a * _d_a) * a + _t0 * _d_a + (_d_b * b + b * _d_b) * b + _t1 * _d_b;
}
Reverse mode

Consider: \( f(x_1, x_2) = \sin(x_1) + x_1 x_2 \)

```cpp
f(x1, x2) {
    x1 = x1
    x2 = x2
    a = x1 * x2
    b = \sin(x1)
    return a + b;
}
```

```cpp
f_dx1(x1, x2) {
    gz = 1
    gb = gz
    ga = gz
    gx2 = x1 * ga
    gx1 = x2 * ga + \cos(x1) * gb
}
```
Reverse mode AD computes gradients (w.r.t to all inputs at once)

\[ f(x_1, x_2) = \sin(x_1) + x_1 x_2 \]

[Wikipedia, Automatic differentiation]
double f_cubed_add1(double a, double b) {
    return a * a * a + b * b * b;
}

void f_cubed_add1_grad (double a, double b, double * _result) {
    double _t0;
    double _t1;
    double _t2;
    double _t3;
    double _t4;
    double _t5;
    double _t6;
    double _t7;
    _t2 = a;
    _t1 = a;
    _t3 = _t2 * _t1;
    _t0 = a;
    _t6 = b;
    _t5 = b;
    _t7 = _t6 * _t5;
    _t4 = b;
    double f_cubed_add1_return = _t3 * _t0 + _t7 * _t4;
    goto _label0;
_label0:
{
    double _r0 = 1 * _t0;
    double _r1 = _r0 * _t1;
    _result[0UL] += _r1;
    double _r2 = _t2 * _r0;
    _result[0UL] += _r2;
    double _r3 = _t3 * 1;
    _result[0UL] += _r3;
    double _r4 = 1 * _t4;
    double _r5 = _r4 * _t5;
    _result[1UL] += _r5;
    double _r6 = _t6 * _r4;
    _result[1UL] += _r6;
    double _r7 = _t7 * 1;
    _result[1UL] += _r7;
}
}
What can be differentiated

- Built-in C/C++ scalar types (e.g. double, float)
- Built-in C input arrays
- Functions that have an arbitrary number of inputs
- Functions that return a single value
- Loops
- Conditionals
Benchmarks: in High-Energy Physics Uses

Clad:

```cpp
f1->GradientPar(x, result);
```

Numerical:

```cpp
h1->GradientPar(x, result);
```

- **gaus**: $N_{\text{par}} = 3$
- **expo**: $N_{\text{par}} = 2$
- **crystalball**: $N_{\text{par}} = 5$
- **breitwigner**: $N_{\text{par}} = 5$
- **cheb2**: $N_{\text{par}} = 4$

~10x faster!
Benchmarks

Tested function:

double sum(double* p, int dim) {
    double r = 0.0;
    for (int i = 0; i < dim; i++)
        r += p[i];
    return r;
}

Numerical:

double* Numerical(double* p, int dim, double eps = 1e-8) {
    double result = new double[dim]{}
    for (int i = 0; i < dim; i++) {
        double pi = p[i];
        p[i] = pi + eps;
        double v1 = sum(p, dim);
        p[i] = pi - eps;
        double v2 = sum(p, dim);
        result[i] = (v1 - v2)/(2 * eps);
    }
    return result;
}

Clad:

double* Clad(double* p, int dim) {
    auto result = new double[dim]{};
    auto sum_grad = clad::gradient(sum, "p");
    sum_grad.execute(p, dim, result);
    return result;
}

Example how to use clad

\[
\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x+h_i)-f(x-h_i)}{2h}
\]
Benchmarks

Original function:

```cpp
double sum(double* p, int dim) {
    double r = 0.0;
    for (int i = 0; i < dim; i++)
        r += p[i];
    return r;
}
```

Clad’s gradient:

```cpp
void sum_grad_0(double *p, int dim, double *result) {
    double _d_r = 0;
    unsigned long _t0;
    int _d_i = 0;
    clad::tape<int> _t1 = {};
    double r = 0.;
    _t0 = 0;
    for (int i = 0; i < dim; i++) {
        _t0++;
        r += p[clad::push(_t1, i)];
    }
    double sum_return = r;
    _d_r += 1;
    for (; _t0; _t0--) {
        double _r_d0 = _d_r;
        _d_r += _r_d0;
        _d_r -= _r_d0;
        result[clad::pop(_t1)] += _r_d0;
    }
}
```
Benchmarks

![Graph showing benchmarks for Clad and Numerical with dim on the x-axis and ns on the y-axis.]

- Clad
- Numerical

- 5000
- 4000
- 3000
- 2000
- 1000
- 0

- 4786
- 1182
- 620
- 394
- 224
- 450

- 5
- 10
- 20
- 40
Benchmarks

~dim/4 times faster!
Benchmarks

Original function:

```cpp
double gaus(double* x, double* p /*means*/, double sigma, int dim) {
    double t = 0;
    for (int i = 0; i < dim; i++)
        t += (x[i] - p[i])*(x[i] - p[i]);
    t = -t / (2*sigma*sigma);
    return std::pow(2*M_PI, -n/2.0) * std::pow(sigma, -0.5) * std::exp(t);
}
```

\[
\frac{1}{\sqrt{(2\pi)^{\text{dim}} \sigma}} e^{-\frac{|x-p|^2}{2\sigma^2}}
\]
Benchmarks

Clad  Numerical

~dim/25 times faster

450x
Future Work

- Hessians
  - Finding a way to calculate the determinant
  - Resolving the 1-dimension array issue to allow for 2d array input and output
  - Benchmarking row-by-row approach
- Jacobians
  - Finding a way to compose forward and reverse mode together, i.e. 
    `clad::differentiate(clad::gradient(f))`
- Support OpenCL and CUDA
Thank you!

- Clad: [https://github.com/vgvassilev/clad](https://github.com/vgvassilev/clad)
- Special thanks to: A. Efremov, A. Penev, M. Vasilev, O. Shadura, V. Ilieva, J. Qui
- More about automatic differentiation: [http://www.autodiff.org](http://www.autodiff.org)
Backup
Automatic differentiation

\[ l_1 = x \]
\[ l_{n+1} = 4l_n(1 - l_n) \]
\[ f(x) = l_4 = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2 \]

Coding

```
f(x):
    v = x
    for i = 1 to 3
        v = 4*v*(1-v)
    return v
```

or, in closed-form,

```
f(x):
    return 64*x*(1-x)*((1-2*x)^2)*(1-8*x+8*x*x)*2
    *(1-8*x+8*x*x)^2
```

Symbolic differentiation of the closed-form

```
f'(x):
    return 128*x*(1-x)*(-8+16*x)*((1-2*x)^2)*(1-8*x+8*x*x) + 64*(1-x)*((1-2*x)^2)*((1-8*x+8*x*x)^2) - (64*x*(1-2*x)*2)*(1-8*x+8*x*x)^2 - 256*x*(1-x)*(1-2*x)*(1-8*x+8*x*x)^2
```

```
f'(x) = f'(x_0)
```

Exact

Automatic differentiation

```
f'(x):
    (v,dv) = (x,1)
    for i = 1 to 3
        (v,dv) = (4*v*(1-v), 4*dv-8*v*dv)
    return (v,dv)
```

\[ f'(x_0) = f'(x_0) \]

Exact

Numerical differentiation

```
f'(x):
    h = 0.000001
    return (f(x+h) - f(x)) / h
```

\[ f'(x_0) \approx f'(x_0) \]

Approximate

[Baydin et al., Automatic Differentiation in Machine Learning: a Survey, 2018]
Work is done by GSOC student Jack Qui
What automatic differentiation is

- Technique for evaluating the derivatives of mathematical functions
- Applies differentiation rules to each arithmetical operation in the code

```c
double c = a + b;

double d_c = d_a + d_b;
```

```c
double c = a * b;

double d_c = a * d_b + d_a * b;
```

...
What automatic differentiation is

- Not limited to closed-form expressions
- Can take **derivatives of algorithms** (conditionals, loops, recursion)

**Example: loops**

```c
double pow(double x, int n) {
    double r = 1;
    for (int i = 0; i < n; i++)
        r = r * x;
    return r;
}
```

```c
double pow_darg0(double x, int n) {
    double d_r = 0;
    double r = 1;
    for (int i = 0; i < n; i++) {
        d_r = d_r*x + r*1;
        r = r*x;
    }
    return d_r;
}
```
Automatic differentiation in Clad

- At the moment supports functions with:
  - multiple (*scalar*) inputs
  - single scalar output value

\[ f : \mathbb{R}^n \to \mathbb{R} \]

- Will be extended soon with:
  - vector inputs

\[
\begin{align*}
\text{double } f(\text{double } x_0, \text{double } x_1, \ldots, \text{double } x_n); \\
\text{double } f(\text{vector<double> } x); \\
\text{double } f(\text{double* } x);
\end{align*}
\]

- Can be extended with:
  - multiple outputs

\[ f : \mathbb{R}^n \to \mathbb{R}^m \]

\[
\text{vector<double> } f(\text{vector<double> } x);
\]
Automatic differentiation in Clad

- For \( f : \mathbb{R}^n \rightarrow \mathbb{R} \) can generate:
  - single derivative \( \frac{\partial f}{\partial x_i} \)
    - \texttt{clad::differentiate}(f, i);
  - gradient \( \nabla f = \left( \frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right) \)
    - \texttt{clad::gradient}(f);

- Supports both \textit{forward} and \textit{reverse} mode AD:
  - \texttt{clad::differentiate} uses forward mode
  - \texttt{clad::gradient} uses reverse mode
Current state

Support of C++ constructs:

- Tested with built-in floating point types: float, double
- In principle, should work with user-defined scalar types, needs testing
- Arithmetic operators, function calls, variable declarations, if statements …
- In forward mode:
  - variable mutation (reassignments), for loops

TODO:

- Arrays/vectors, struct/class methods, custom data structures…
- Occasional missing C++ constructs
- Rigorous documentation, error/warning handling
Why is the speedup factor higher than theoretical limit of \( \sim N_{\text{par}} \)?

From TF1::GradientPar():

```cpp
// save original parameters
Double_t par0 = parameters[ipar];

parameters[ipar] = par0 + h;
f1 = func->EvalPar(x, parameters);
parameters[ipar] = par0 - h;
f2 = func->EvalPar(x, parameters);
parameters[ipar] = par0 + h / 2;
g1 = func->EvalPar(x, parameters);
parameters[ipar] = par0 - h / 2;
g2 = func->EvalPar(x, parameters);

// compute the central differences
h2 = 1 / (2. * h);
d0 = f1 - f2;
d2 = 2 * (g1 - g2);
T grad = h2 * (4 * d2 - d0) / 3.;

// restore original value
parameters[ipar] = par0;
return grad;
```

- some initial bookkeeping
- 4 calls to \( f \)
- additional ops to improve accuracy
Hessians - How it is implemented

- Generated through using forward mode AD, then reverse mode AD
- Iteratively calculates each column of the Hessian at a time, which is encapsulated within a second-order partial derivative function
- Combines all of these helper functions that correspond to columns of a Hessian into a single Hessian function
- Encapsulated in Clad API through clad::hessian

\[
H = \begin{bmatrix}
\frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\
\frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2}
\end{bmatrix}
\]

Work is done by GSOC student Jack Qui
Hessians

- Square n x n matrix containing all second order partial derivatives w.r.t to all inputs
- Useful for optimisation problems and as a second derivative test

\[ H = \begin{bmatrix}
\frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\
\frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2}
\end{bmatrix} \]

Work is done by GSOC student Jack Qui
double f_cubed_add1(double a, double b) {
    return a * a * a + b * b * b;
}

auto func = clad::hessian(f_cubed_add1);
func.dump();

void f_cubed_add1_hessian(double a, double b, double *hessianMatrix) {
    f_cubed_add1_darg0_grad(a, b, &hessianMatrix[0UL]);
    f_cubed_add1_darg1_grad(a, b, &hessianMatrix[2UL]);
}