Automatic Differentiation in ROOT

Vassil Vassilev (Princeton University)

Aleksandr Efremov (Princeton University)

Oksana Shadura (University of Nebraska Lincoln)

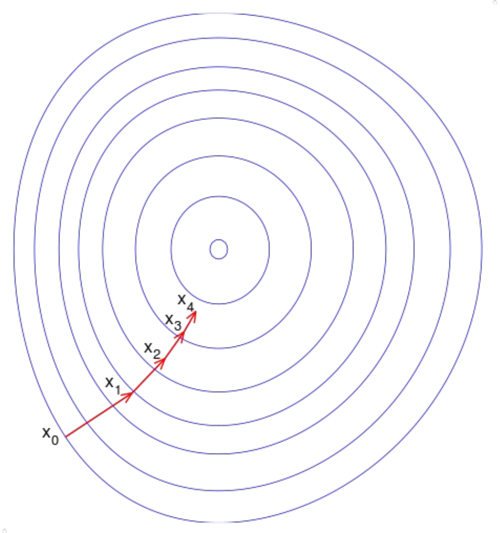
Gradient-based optimization

Gradient descent:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \alpha \nabla f(\mathbf{x}_i)$$

Applications:

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



[Wikipedia, Gradient descent]

What is automatic differentiation [1/2]

- Creates a function that computes the derivative(s) for you
- Alternative to numerical differentiation

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

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

Clad

```
double f_darg0(double x) {
  return 1*x + x*1;
}
```

What is 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

Clad and its goals

Clad enables **automatic differentiation (AD) for C++**. It is based on 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 an interpreter-generated routine)
- Provide an alternative way of gradient computations in ROOT's fitting algorithms

* https://github.com/vgvassilev/clad

What Clad does

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

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

Clad

```
double f_darg0(double x) {
  return 1 * x * x + x * 1 * x +
x * x * 1;
}
```

How Clad works

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

```
FunctionDecl f 'double (double)'
|-ParmVarDecl x 'double'
`-CompoundStmt
  `-ReturnStmt
    `-BinaryOperator 'double' '*'
      |-ImplicitCastExpr 'double' <LValueToRValue>
      | `-DeclRefExpr 'double' lvalue ParmVar 'x' 'double'
      `-ImplicitCastExpr 'double' <LValueToRValue>
        `-DeclRefExpr 'double' lvalue ParmVar 'x' 'double'
```

Clad

derivative AST

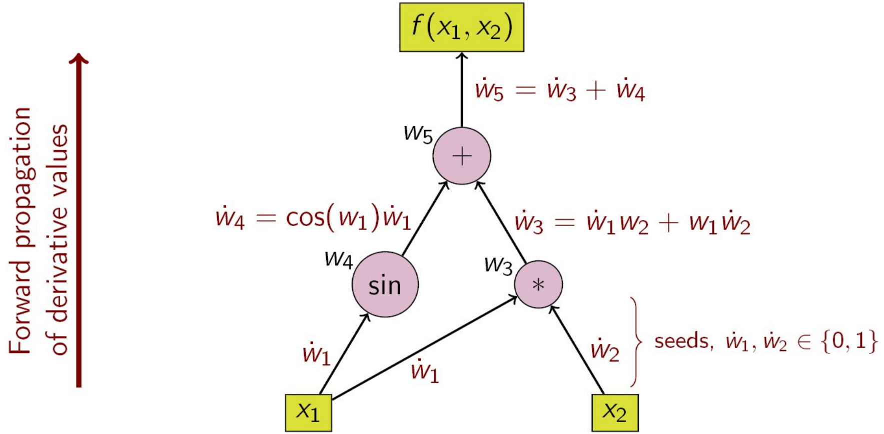
codegen

```
double f_darg0(double x) {
  return 1*x + x*1;
}
```

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

Forward mode `clad::differentiate`

- **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]

clad::differentiate

```
double f_cubed_add1(double a, double b) {
 return a * a * a + b * b * b;
}
```

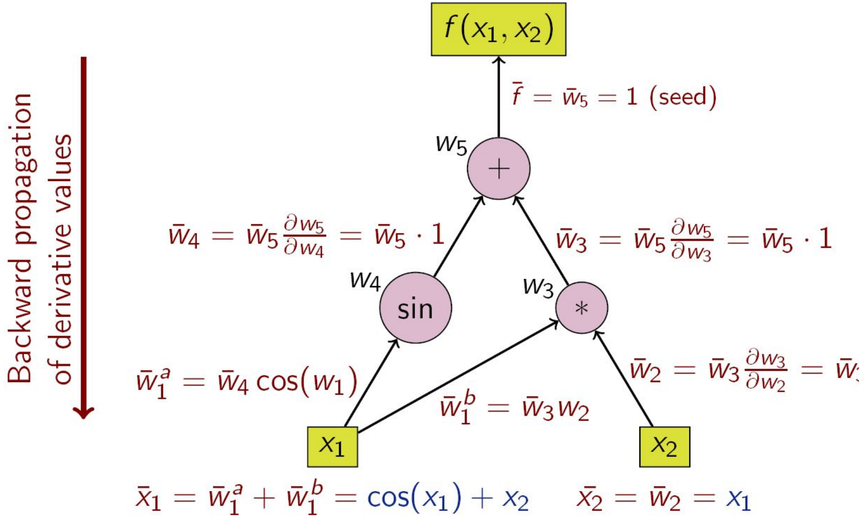
clad::differentiate

```
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

`clad::gradient`

- **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]

clad::gradient

```
double f_cubed_add1(double a,
double b) {
 return a * a * a + b * b * b;
}
```

clad::gradient

```
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, int)
- Built-in C input arrays
- Functions that have an arbitrary number of inputs
- Functions that return a single value
- Loops
- Conditionals

Benchmarks: in ROOT

```
TF1* form = new TF1("f1", "formula");
TFormula* f1 = form->GetFormula();
f1->GenerateGradientPar(); // clad
```

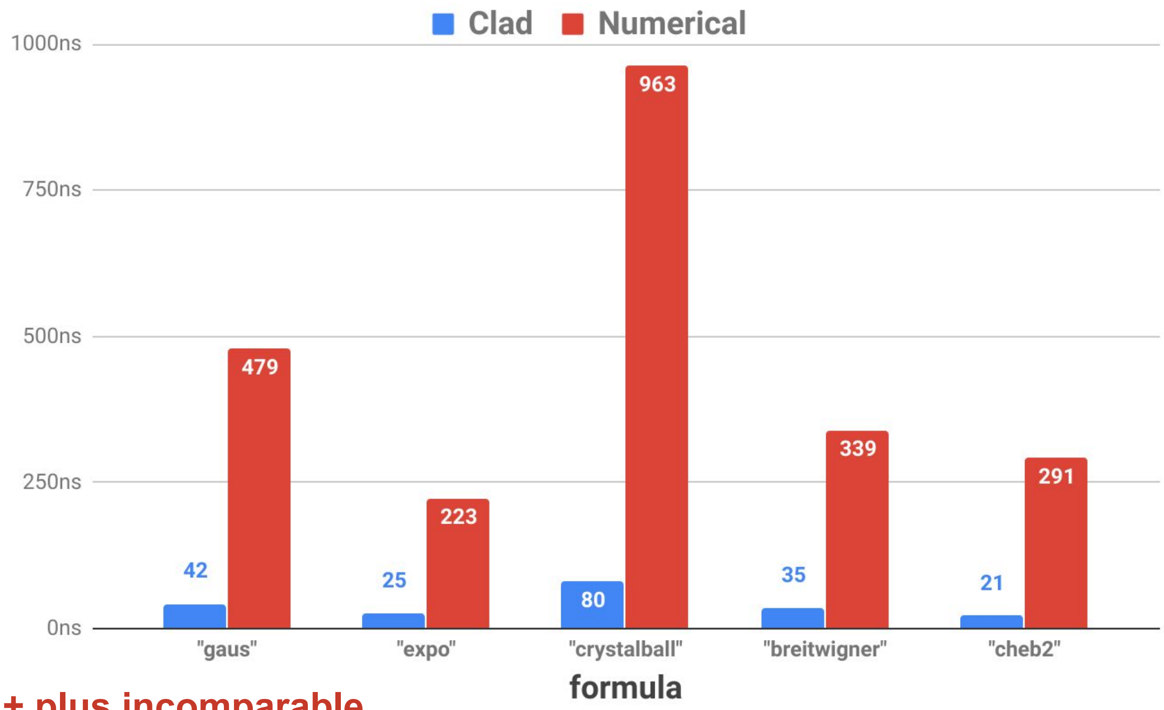
Clad:

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

Numerical:

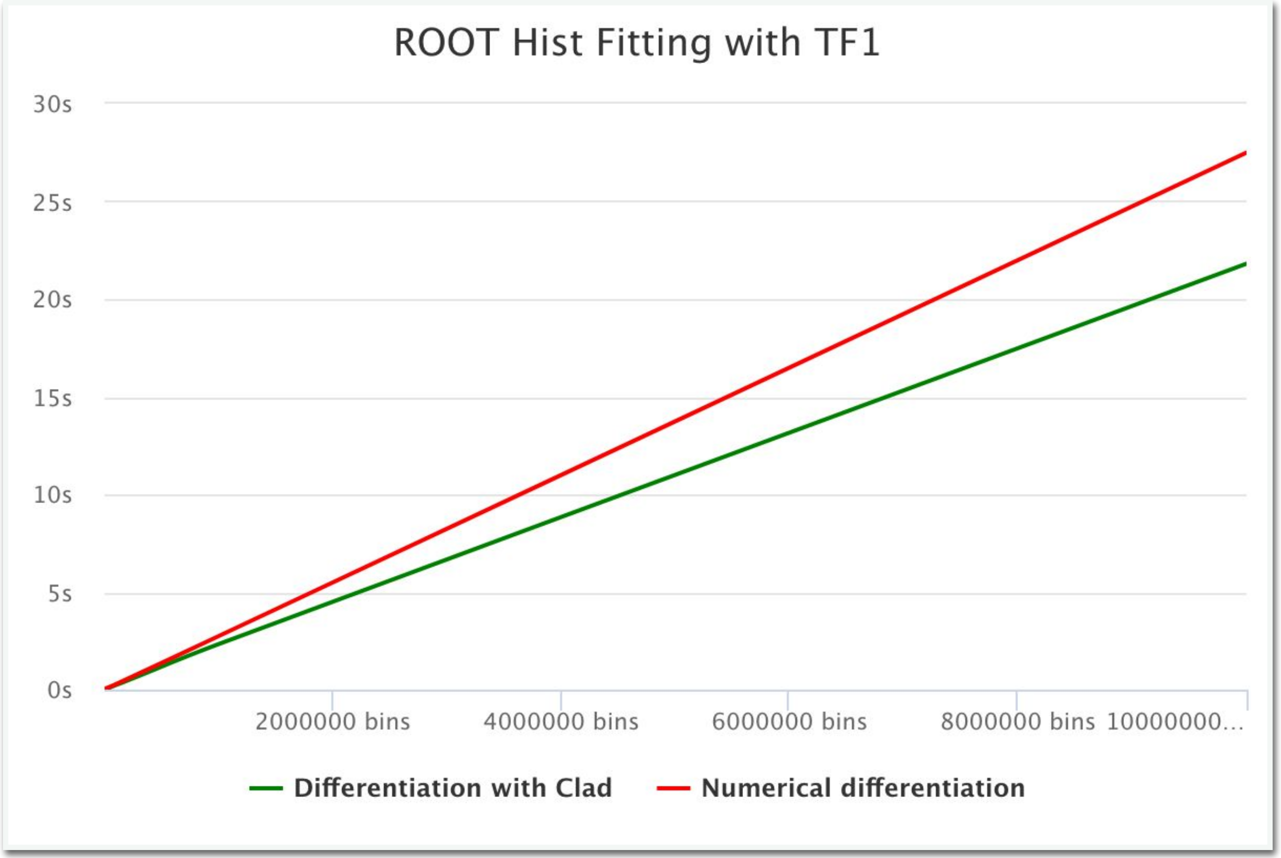
```
form->GradientPar(x, result);
```

- **gaus: Npar = 3**
- **expo: Npar = 2**
- **crystalball: Npar = 5**
- **breitwigner: Npar = 5**
- **cheb2: Npar = 4**



~10x faster + plus incomparable correctness!

ROOT Histogram Fitting



Benchmarks: C++, just clad, no ROOT

Tested function:

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

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

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);
    p[i] = pi;
  }
  return result;
}
```

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

Benchmarks: C++, just clad, no ROOT

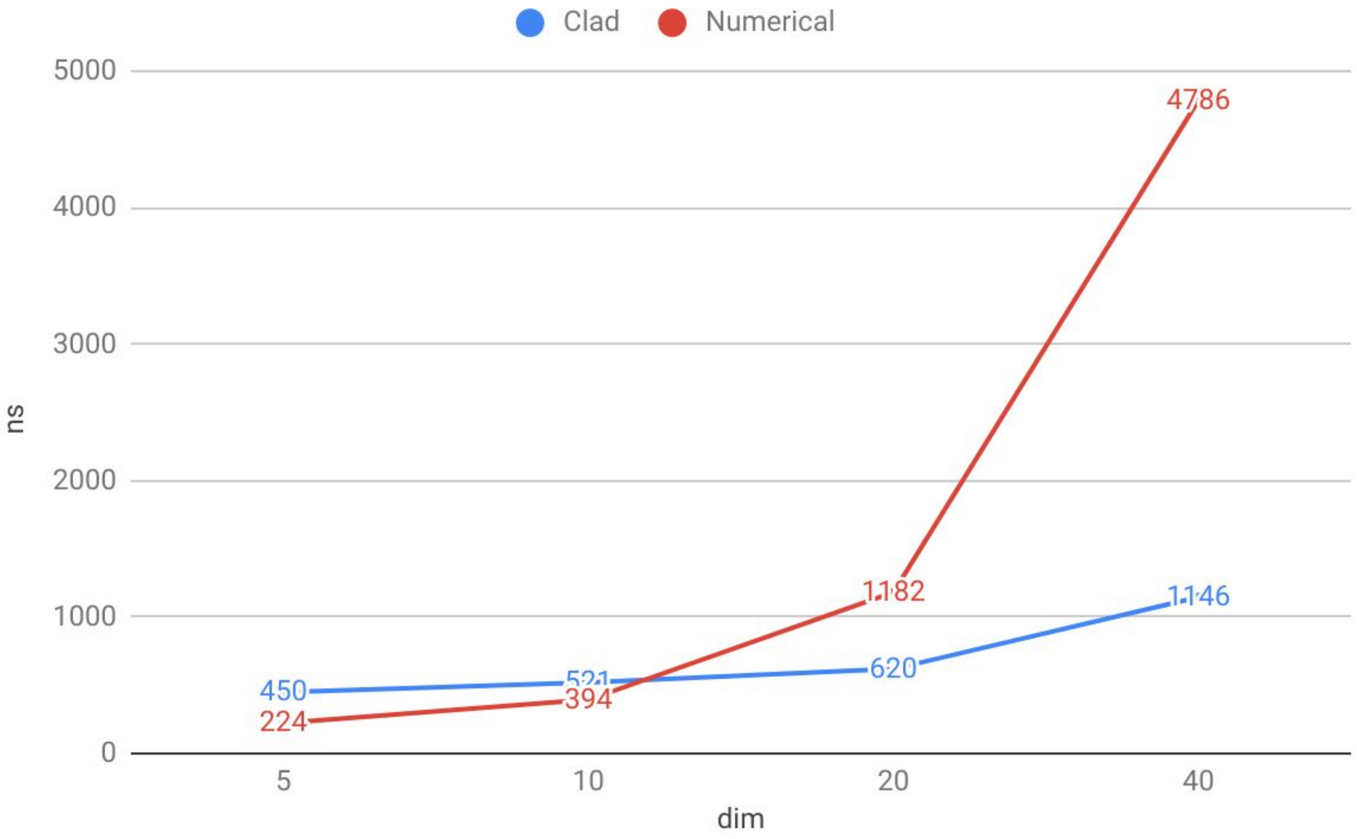
Original function:

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

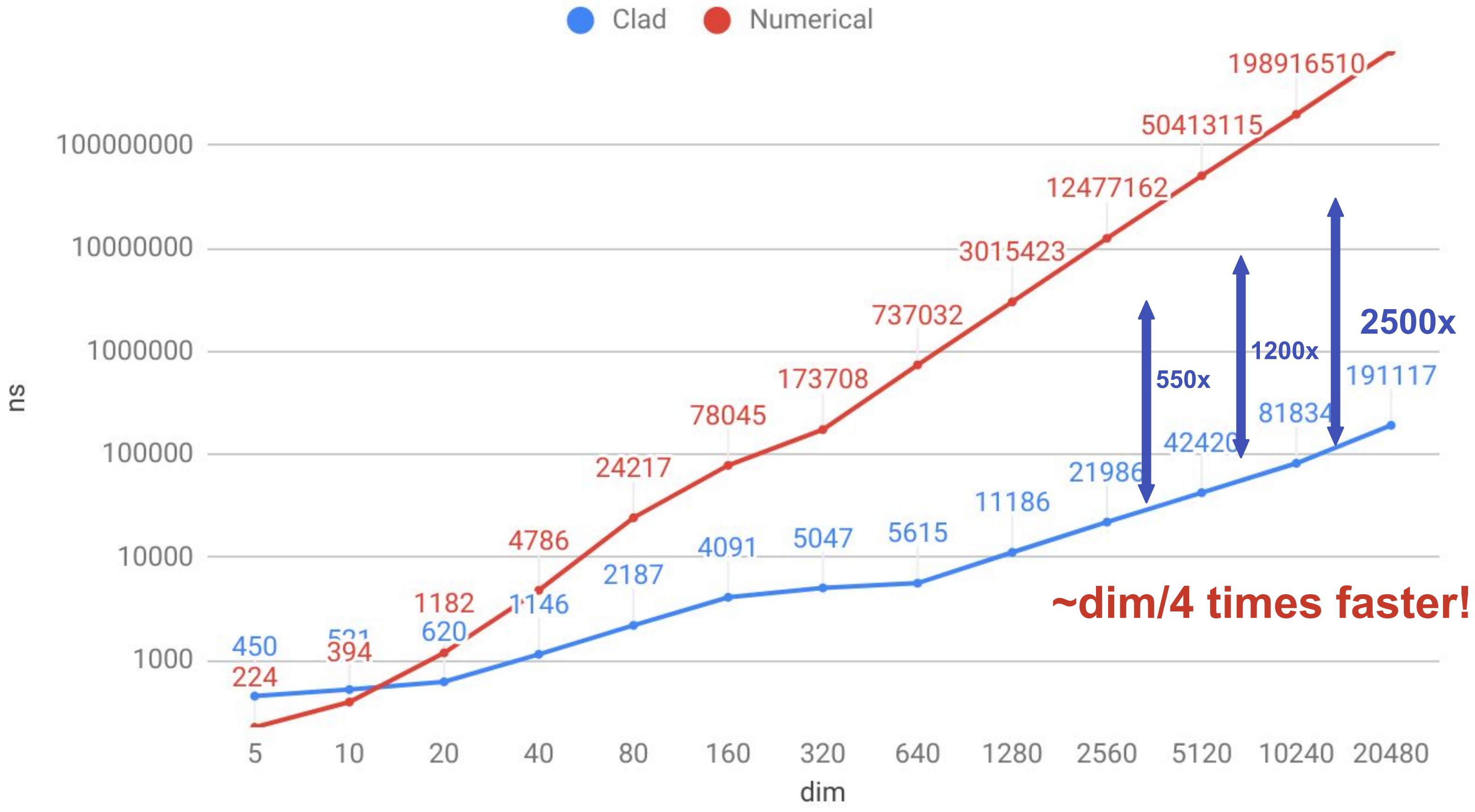
Clad's gradient:

```
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;
        _result[clad::pop(_t1)] += _r_d0;
        _d_r -= _r_d0;
    }
}
```

Benchmarks: C++, just clad, no ROOT



Benchmarks: C++, just clad, no ROOT



Benchmarks: C++, just clad, no ROOT

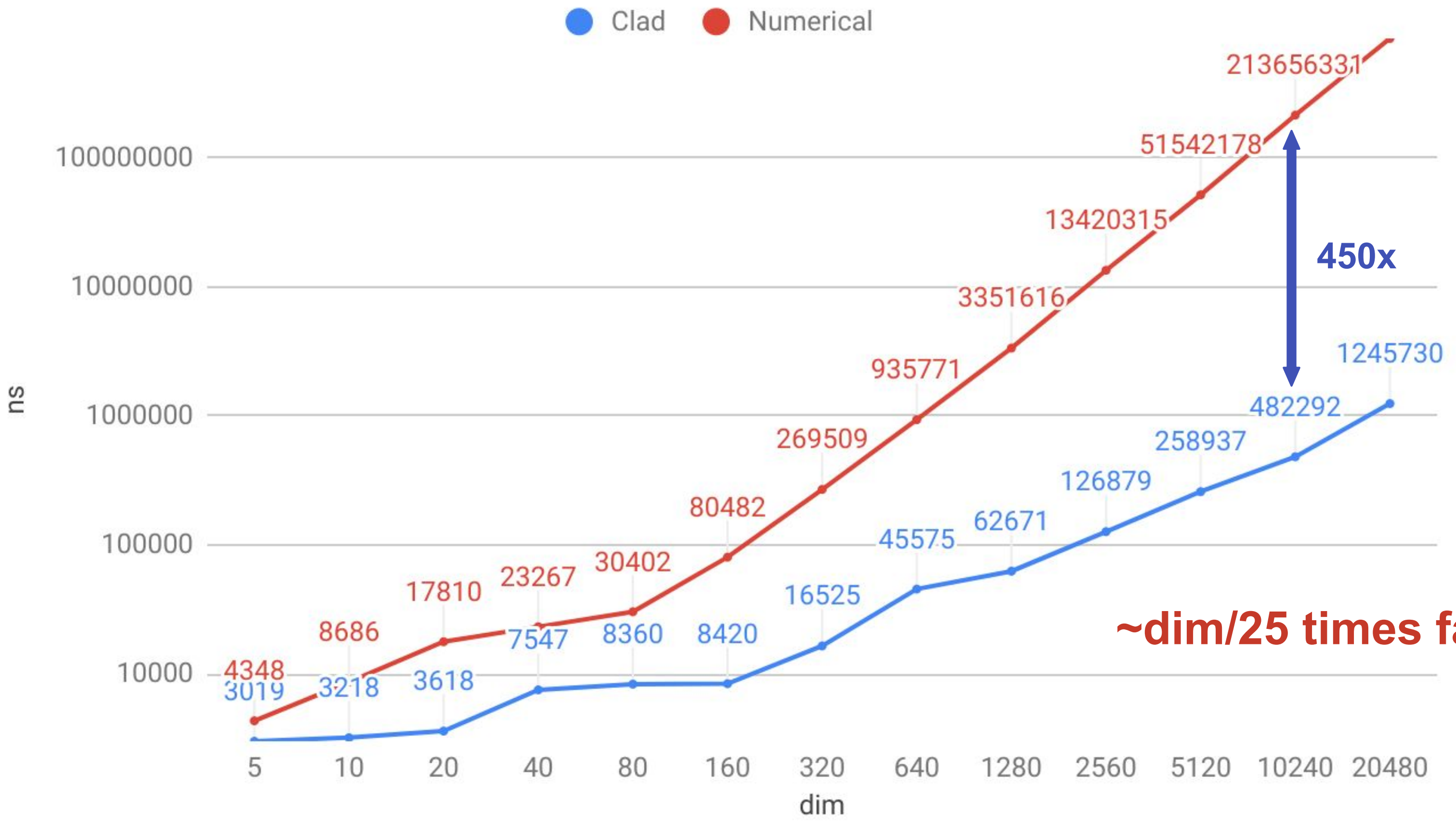
Original function:

```
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);
}
```

$$\frac{1}{\sqrt{(2\pi)^{dim}\sigma}} e^{-\frac{||x-p||_2^2}{2\sigma^2}}$$

**Artificial synthetic benchmark*
[e.g. one of the multivariate normal distribution applications is face detection]

Benchmarks: C++, just clad, no ROOT



~dim/25 times faster

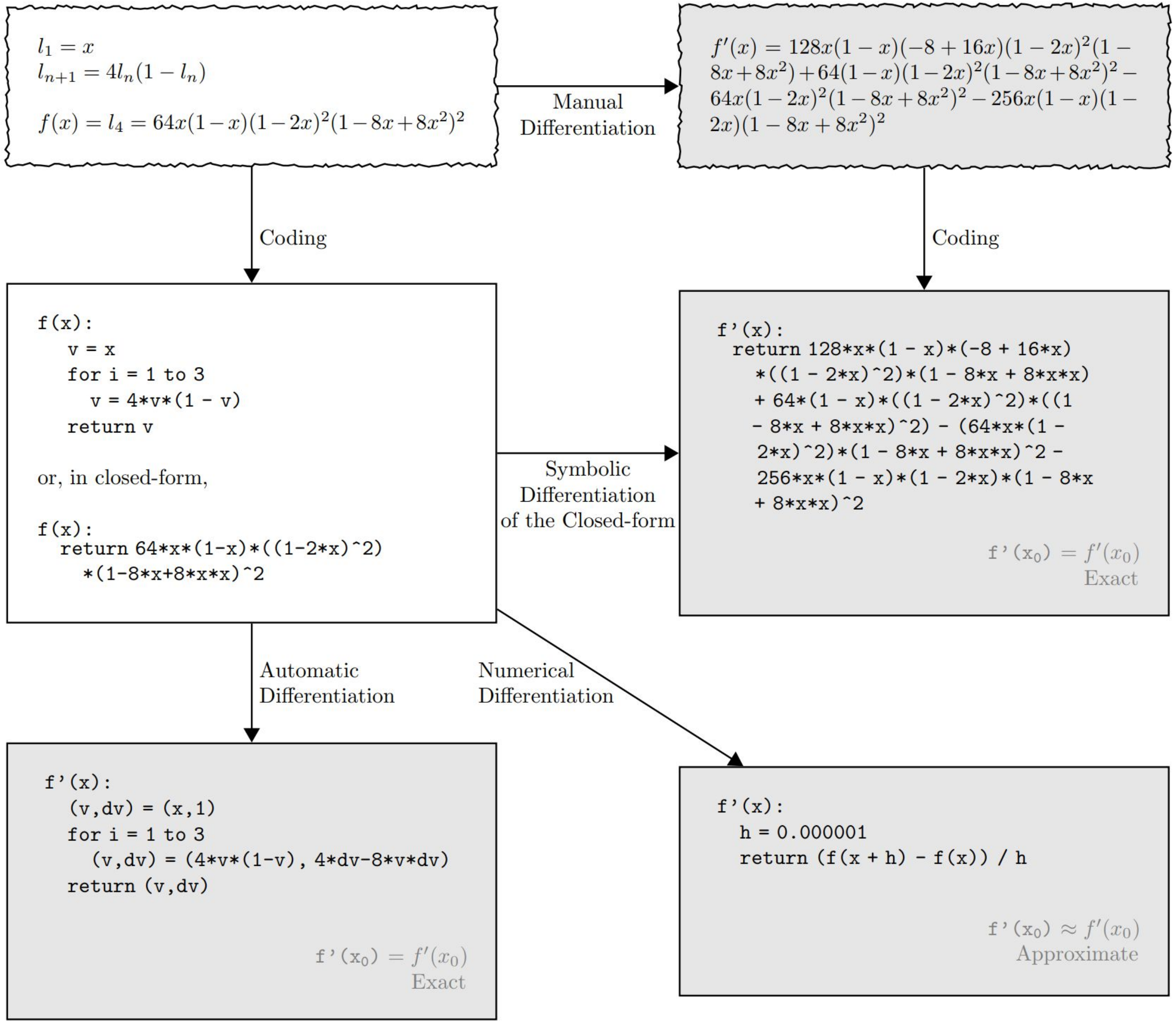
Future Work

- 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))
- Extend the usage of the TFormula differentiation backend
- Teach rootcling how to use clad and store the derivatives in the dictionaries

Thank you!

- Clad: **https://github.com/vgvassilev/clad**
- With any questions please contact:
  - Vassil Vassilev vgvassilev@cern.ch
  - Aleksandr Efremov efremovaleksandr@icloud.com
- More about automatic differentiation: **http://www.autodiff.org**

Backup



[Baydin et al., Automatic Differentiation in Machine Learning: a Survey, 2018]

Clad functionality comparison

Feature\Tool	Swift AD	Autograd (Python)	Tangent (Python)	ADOL-C	CppAD	Clad
Basic forward and reverse-mode AD	Supported	Supported	Supported	Supported	Supported	Supported
Loops, conditionals, recursion	Supported	Supported	Supported	Supported	Supported	Supported
Works with third-party libraries, e.g. TensorFlow	Supported (TensorFlow)	Supported (Numpy)	Supported (TensorFlow)	Not Supported	Not Supported	Not Supported
User-defined numeric types/custom type inputs	Supported	Not Supported	Not Supported	Not Supported	Not Supported	Not Supported
Works with vector functions / nested containers e.g. tuple(tuple)	Supported	Supported	Supported	Supported	Supported	Not Supported
Custom-defined derivatives/gradients	Supported	Supported	Supported	Not Supported	Not Supported	Supported
Can mix and match differential operators (e.g. mix/match forward and reverse modes)	Supported	Supported	Supported	Supported	Supported	Supported
Error diagnostics that pinpoint exact location of failure / has own debugging tool	Supported	Not Supported	Supported	Not Supported	Not Supported	Not Supported
Has a marker/attribute to tell tool max amount times to differentiate	Supported	Not Supported	Not Supported	Not Supported	Not Supported	Not Supported
#differential(f): Produces a function that takes the original arguments and returns the differential of f.	Supported	Not Supported	Not Supported	Supported	Supported	Not Supported
#pullback(f): Produces a function that takes the original arguments and returns the pullback of f.	Supported	Not Supported	Not Supported	Not Supported	Not Supported	Not Supported
CUDA/GPU Support/Multithreading	Not Supported	Not Supported	Not Supported	Partially Supported	Supported	Not Supported
Checkpointing	Not Supported	Supported	Not Supported	Supported	Supported	Not Supported
Traceless forward differentiation (Source code transformation forward mode)	Not Supported	Not Supported	Supported	Supported	Not Supported	Supported
Classes	Not Supported	Supported	Not Supported	Supported	Not Supported	Supported
Tensor evaluation	Not Supported	Not Supported	Not Supported	Supported	Not Supported	Not Supported
Jacobians	Supported	Supported	Supported	Supported	Supported	Not Supported
Hessians	Supported	Supported	Supported	Supported	Supported	Not Supported
Exploits sparsity of Jacobian/Hessian	Not Supported	Not Supported	Not Supported	Supported	Supported	Not Supported
Nested functions / lambdas	Not Supported	Supported	Not Supported	Not Supported	Not Supported	Not Supported
Nested calls to functions	Supported	Supported	Supported	Supported	Supported	Supported
Closures	Supported	Supported	Not Supported			
Complex Numbers	Not Supported	Supported	Supported	Supported	Supported	Not Supported
Static optimisation/dead code eliminiation	Not Supported	Not Supported	Supported	Not Supported	Supported	Not Supported
Templates				Supported	Supported	Supported
Works with atomic functions				Not Supported	Supported	Not Supported

Key: Supported | Not Supported | Partially Supported | Unfair Comparison

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

```
double c = a + b;    →    double d_c = d_a + d_b;

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

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

```
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 dr;
}
```

Automatic differentiation in Clad

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

$$f : \mathbb{R}^n \to \mathbb{R}$$

```
double f(double x0, double x1, ..., double xn);
```

- Will be extended soon with:
  - **vector** inputs

```
double f(vector<double> x);
```

```
double f(double* x);
```

- Can be extended with:
  - multiple outputs

$$f : \mathbb{R}^n \to \mathbb{R}^{m}$$

```
vector<double> f(vector<double> x);
```

Automatic differentiation in Clad

- For $f : \mathbb{R}^n \rightarrow \mathbb{R}$ can generate:
  - single derivative $\frac{\partial f}{\partial x_i}$ `clad::differentiate(f, i);`
  - gradient $\nabla f = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right)$ `clad::gradient(f);`
- Supports both ***forward*** and ***reverse*** mode AD:
  - `clad::differentiate` uses forward mode
  - `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:

- In **reverse** mode: variable mutation (reassignments), for loops
- 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 ~Npar?

From TF1::GradientPar():

...

```
// 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

$$\mathbf{H} = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1\,\partial x_n} \\ \dfrac{\partial^2 f}{\partial x_2\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_n\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

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

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1\,\partial x_n} \\ \frac{\partial^2 f}{\partial x_2\,\partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n\,\partial x_1} & \frac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

Work is done by GSOC student Jack Qui

Hessians - Demo

```
double f_cubed_add1(double a, double b) {
 return a * a * a + b * b * b;
}
```

```
auto func = clad::hessian(f_cubed_add_1);
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]);
}
```