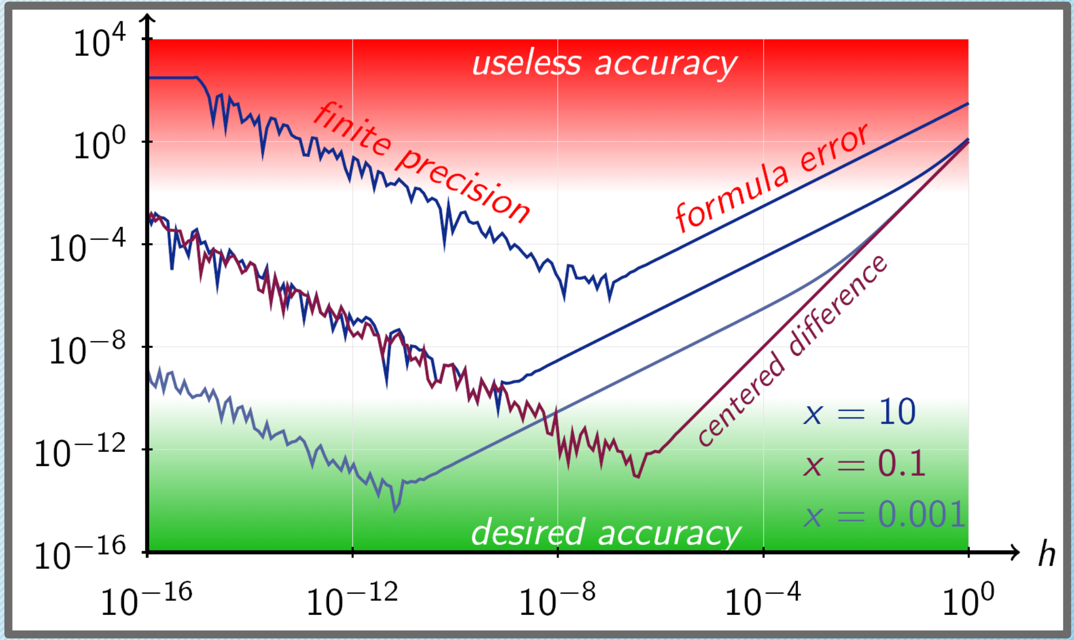
Automatic Differentiation with Clang

Vassil Vassilev
PH-SFT, CERN

Numerical Differentiation

Flavors of the finite differences method: $\frac{f(x+h)-f(x)}{h}$

- Precision losses due to the floating point arithmetic
  - round-offs
  - getting worse for higher order derivatives
  - slow gradient calculation



*Images from wikipedia

Avoiding Numerical Differentiation

$$f(x_1, x_2, \ldots, x_n) = \ldots$$

Symbolic/Mental Differentiation →

$$\frac{\partial f(x_1, x_2, \ldots, x_n)}{\partial x_1} = \ldots$$

↓

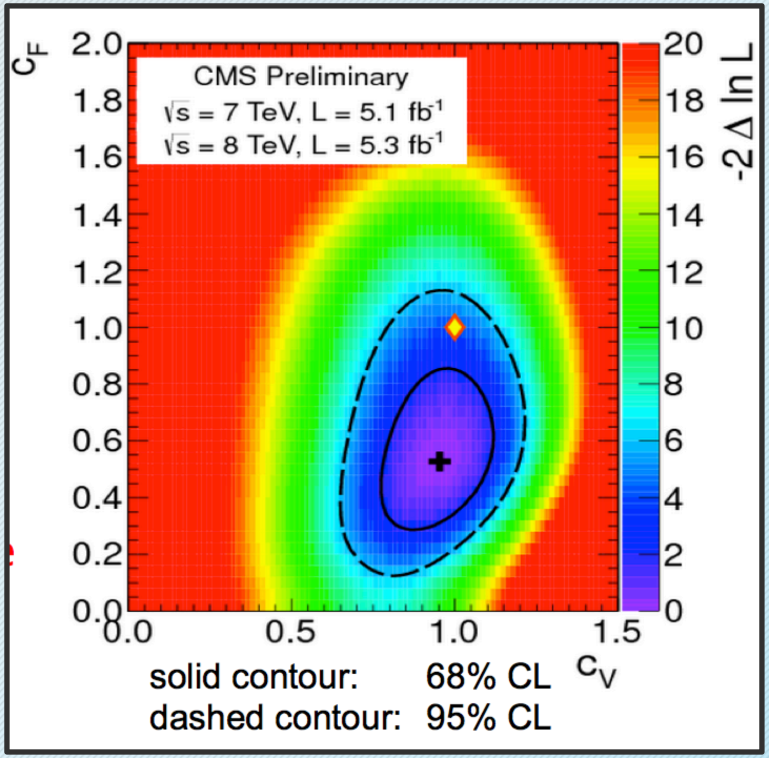
```
float f(x1,x2,..., xN) {
...
}
```

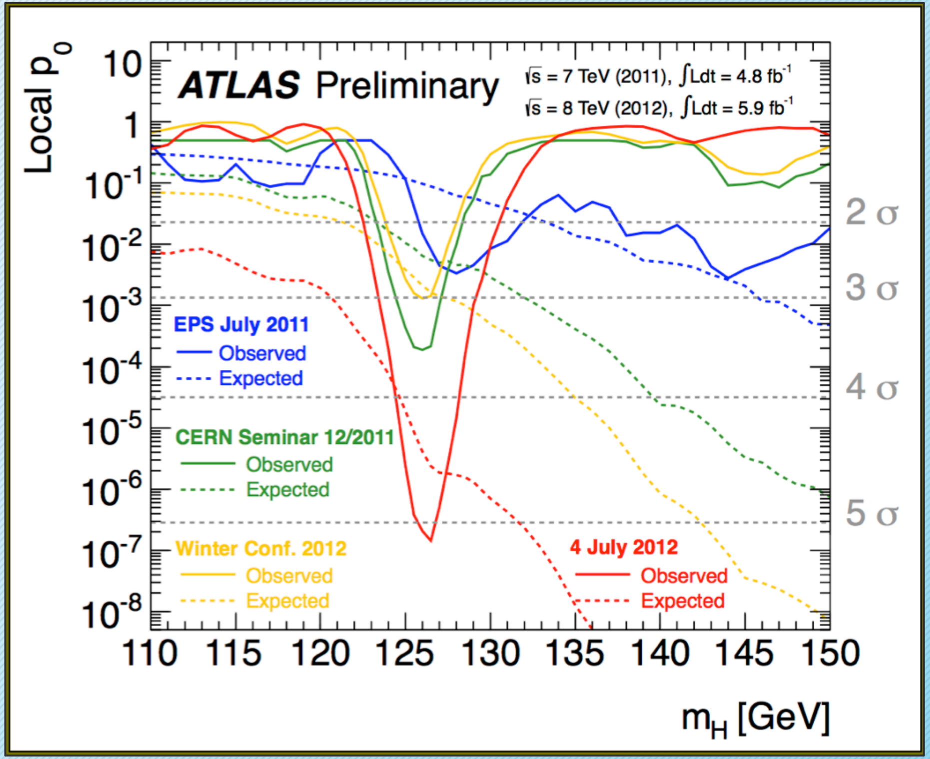
Automatic Differentiation →

```
float f_dx(x1,x2,..., xN) {
...
}
```

Derivatives in C++ in HEP

- Relevant for building gradients used in fitting and minimization.
- Minimization of likelihood function with ~1000 parameters





Automatic/Algorithmic Differentiation

- Employs the chained rule in differential calculus

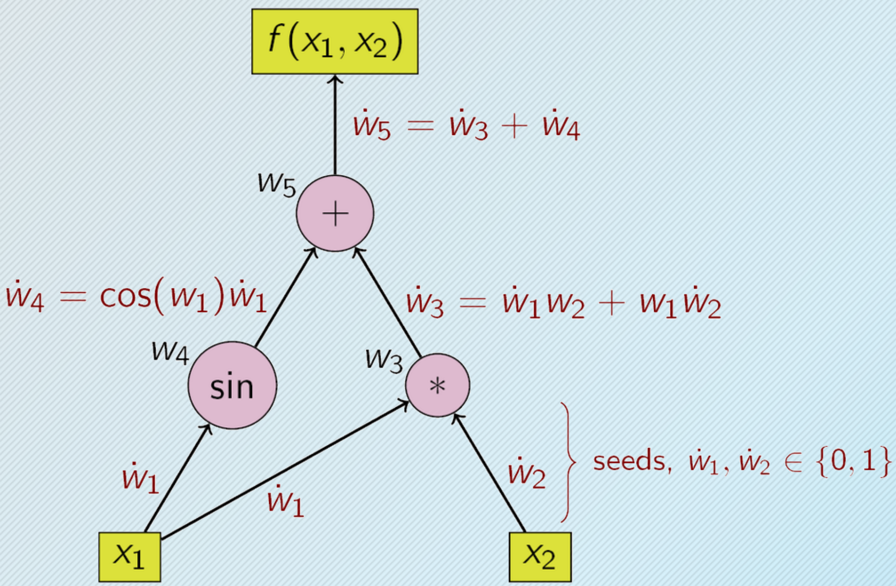
$$z = f(x,y), x = g(t), y = h(t)$$

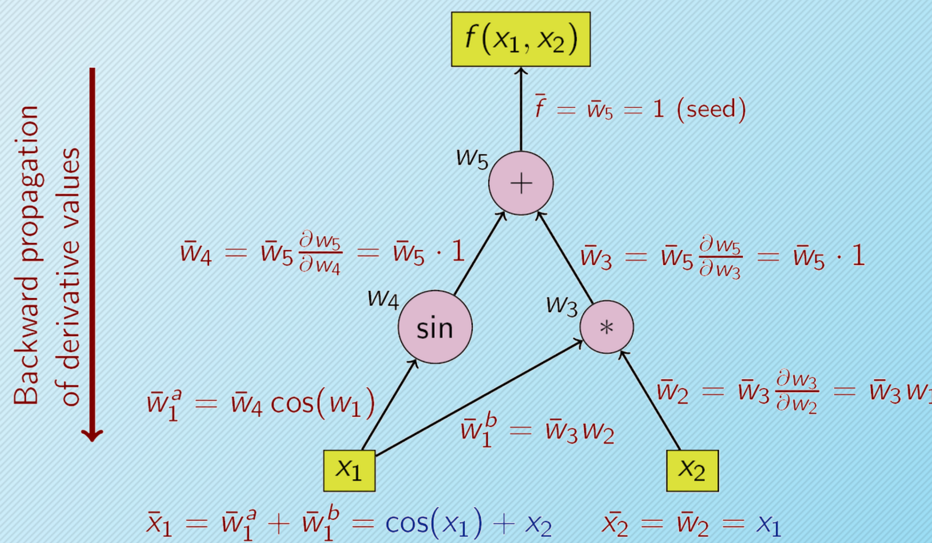
$$\frac{\partial z}{\partial t} = \frac{\partial z}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial z}{\partial y}\frac{\partial y}{\partial t}$$

- Solves all issues with numerical differentiation
- Assumptions: continuity…

General implementation modes

$$f(x_1, x_2) = \sin(x_1) + x_1 x_2$$





*Images from wikipedia

Reverse mode better for gradient calculation of > 30 seeds

Possible AD implementation

- Source-to-source transform
- Operator overloading
- A compiler extension?
  - Benefits : the only one who knows all about the source is the compiler
  - Drawbacks : difficult to port.
  - NAG has AD compiler extension for FORTRAN
  - NAG has AD based on C++ operator overloading

clad – Clang/Cling Automatic Differentiator

- Compiler module, very similar to the template instantiator by idea and design.
- Generates f’ of any given f.

Usage

All clad needs:
Register the plugin
#include

*Find more at: https://github.com/vgvassilev/clad/blob/master/demos/

```
// clang -Xclang -add-plugin -Xclang clad -Xclang -load –Xclang libclad.so T.cpp
#include "clad/Differentiator/Differentiator.h”

double pow2(double x) { return x * x; }
// The body will be generated by clad:
double pow2_dx(double);
int main() {
  // Differentiate pow2. Clad will define a function named
  // pow2_dx(double) with the derivative, ready to be called.
  clad::differentiate(pow2, 0);
  printf("Result is %d\n", pow2_dx(4.2));
  return 0;
}
```

Upon use clad would differentiate pow2

The derivative can be used as a ‘normal’ function

N-th order derivatives


```
// clang -Xclang -add-plugin -Xclang clad -Xclang -load –Xclang libclad.so T.cpp
#include "clad/Differentiator/Differentiator.h"

float func(float x) { return 3.14 * x * x; }
// The body will be generated by clad:
float func_d2x(float x);
int main() {
  // Differentiate func. Clad will define a function named
  // func_d2x(double) with the derivative, ready to be called.
  clad::differentiate<2>(func, 0);
  printf("Result is %d\n", func_d2x(1.1));
  return 0;
}
```

The order of the derivative. It will produce func_dx and func_d2x

Mixed derivatives


```
// clang -Xclang -add-plugin -Xclang clad -Xclang -load –Xclang libclad.so T.cpp
#include "clad/Differentiator/Differentiator.h”

float f1(float x, float y) { return x * x + y * y; }
// The body will be generated by clad:
float f1_dx(float x, float y);
float f1_dx_dy(float x, float y);
int main() {
  // Differentiate f1 with respect to the first argument.
  clad::differentiate(f1, 0);
 // Differentiate f1 with respect to the second argument.
  clad::differentiate(f1_dx, 1);
  printf("Result is %d\n", func_dx_dy(1.1, 2.1));
  return 0;
}
```

Debugging clad


```
// clang -Xclang -add-plugin -Xclang clad -Xclang -load –Xclang libclad.so T.cpp
#include "clad/Differentiator/Differentiator.h”

float f1(float x) { return x * x; }
int main() {
  // Differentiate f1. Clad will define a function named
  // f1_dx(double) with the derivative, ready to be called.

  // f1_dx_obj is of type CladFunction, which is a tiny wrapper over the
  // derived function pointer.
  auto f1_dx_obj = clad::differentiate(f1, 0);
  if (f1_dx_obj.execute(1.) != 2)
    f1_dx_obj.dump(); // unexpected result, broken derivative!?
  printf("Result is %d\n", f1_dx_obj.execute(100.));
  return 0;
}
```

Builtin Derivatives

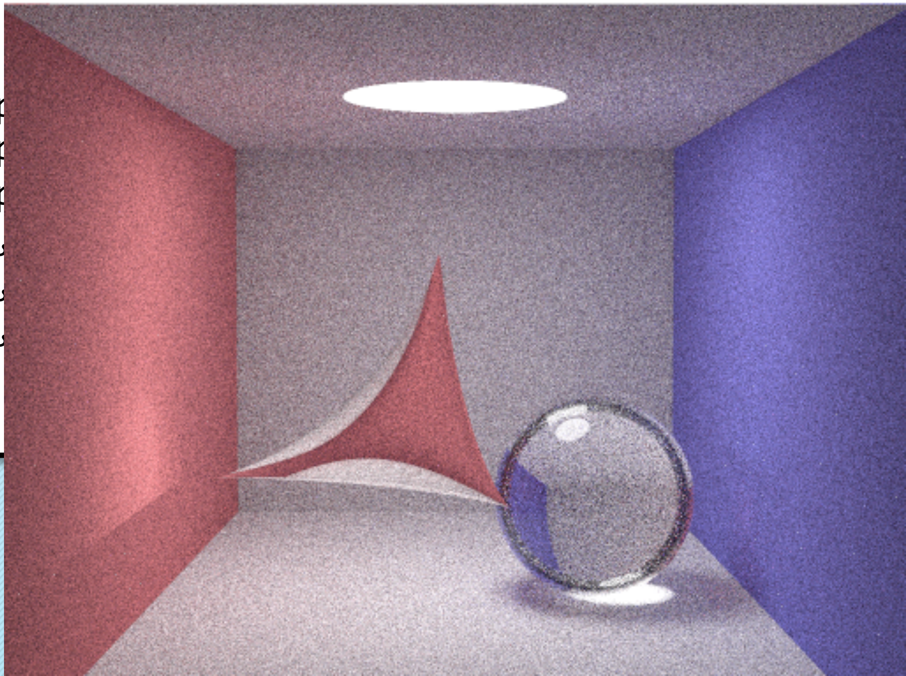

```
// BuiltinDerivatives.h
namespace custom_derivatives {
  template <typename T> T sin_dx(T x) { return cos(x); }
  template <typename T> T cos_dx(T x) { return (-1) * sin(x); }
  template <typename T> T sqrt_dx(T x) {
    return ((T)1)/(((T)2) * sqrt(x));
  }
  ...
}
```

User-defined substitutions

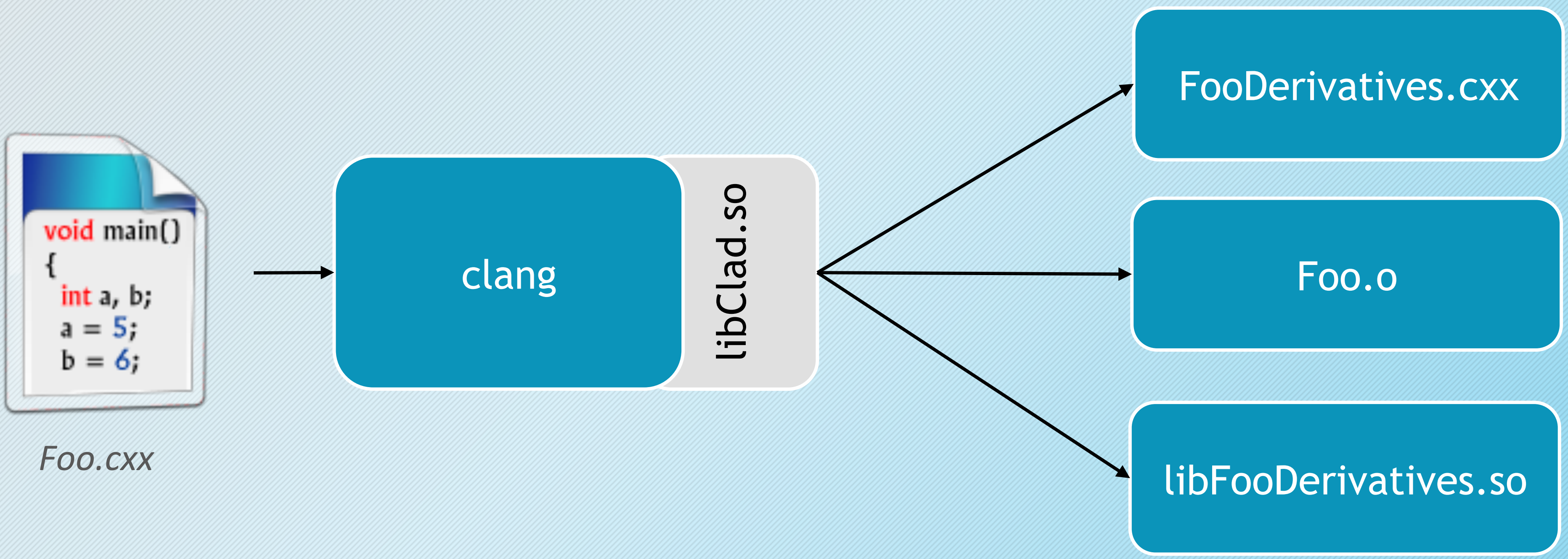

```
// MyCode.h
float custom_fn(float x) {
  return x;
}
namespace custom_derivatives {
  float custom_fn_dx(float x) {
    return x * x;
  }
}

float do_smth(float x) {
  return x * x + custom_fn(x);
}

int main() {
  clad::differentiate(do_smth, 0).execute(2);// will return 6
  return 0;
}
```

clad in Computer Graphics


```
// SmallPT.cpp
float hyperbolic_octahedron_func(float x, float y, float z, const Vec &p, float
r) {
#define sin_a 0.965925826289068
#define cos_a 0.258819045102521
  return pow((x-p.x)*cos_a+(z-p.z)*sin_a, 2./3.) + pow(y-p.y, 2./3.) + pow((x-
p.x)*-sin_a+(z-p.z)*cos_a, 2./3.) - pow(r, 2./3.);
}

Vec normal(const Vec &pt) const override {
  auto ho_func_dx = clad::differentiate(hyp
  auto ho_func_dy = clad::differentiate(hyp
  auto ho_func_dz = clad::differentiate(hyp
  float Nx = ho_func_dx.execute(pt.x, pt.y,
  float Ny = ho_func_dy.execute(pt.x, pt.y,
  float Nz = ho_func_dz.execute(pt.x, pt.y,
  return Vec(Nx, Ny, Nz).norm();
}
```



Portability



Performance

- Clad performance as a part of the compilation process (debug build of clang and clad)
  - Diff a function with ~ 1000 lines with simple expressions

user	system	process	all
0.1115 (100.0%)	0.0007 (100.0%)	0.1122 (100.0%)	0.1122 (100.0%)

  - Diff a function with a complex expression ~ 1000 operators

user	system	process	all
2.7270 (100.0%)	0.0235 (100.0%)	2.7505 (100.0%)	2.7505 (100.0%)

- Clad memory footprint
  - A well educated guess would be not more than an average template instantiation from STL/Boost

Future

- Gradient and Jacobian calculations
- Mix forward and reverse mode
- Retargeting to OpenCL/CUDA
- Integrate clad in ROOT6 though its C++ interpreter cling

Thank you!

Acknowledgement: All the extra manpower for the project is thanks to Google Summer of Code program, done by Violeta Ilieva and Martin Vasilev

Further reading:

- https://github.com/vgvassilev/clad
- http://www.autodiff.org