Utilize second order derivatives from Clad in ROOT

**Student:** Baidyanath Kundu  
**Mentors:** Vassil Vassilev, Ioana Ifrim
Overview

What is ROOT and why are second order derivatives required in it?
- ROOT is a data processing framework created and used in CERN
- Physicists need to calculate the hessian matrix (second order derivative) for various calculations
- More importantly it is used in the **ROOT optimiser** (currently through Numerical Differentiation)

What is Clad and why should we use it to calculate derivatives?
- Clad is an Automatic Differentiation library built as a Clang Plugin
- Automatic Differentiation is **faster** and more **accurate** than Numerical Differentiation
- Thus using the hessian from Clad in the ROOT optimizer will improve its performance
Main Aim: Add hessian mode support to TFormula class in ROOT

- TFormula class acts as a bridge between compiled and interpreted code
- TFormula operates on arrays and generally speaking arrays are an important part of programming but Clad’s hessian mode did not have array support.
- Clad’s hessian mode depends on forward mode and reverse mode
- Reverse mode had support for arrays named “p” but forward mode didn’t have any support for arrays
- So the sub-objective was to add array differentiation support in forward, reverse and hessian mode
Add array diff support in forward mode

Basic idea:

Arrays are groups of single variables

BUT

Allocating more memory = slower calculation

Instead we enable or disable the derivative depending on the requested index
Add array diff support in forward mode

Challenge:

What to do when array subscript uses an expression to access the index?

double sum_arr_darg0_3(double *arr, unsigned int n) {
  unsigned int _d_n = 0;
  double d_sum = 0;
  double sum = 0;
  {
    unsigned int _d_i = 0;
    for (unsigned int i = 0; i < n; i++) {
      d_sum += (i == 3);
      sum += arr[i];
    }
  }
  return _d_sum;
}

double sum_arr_darg0_3(double *arr, unsigned int n) {
  double sum = 0;
  for (unsigned int i = 0; i < n; i++) {
    sum += arr[i];
  }
  return sum;
}

clad::differentiate(sum_arr, "arr[3]");

Solution:

Enable the diff by checking against the requested index
Add array differentiation in reverse mode

Basic idea:

Reverse mode already had support for arrays named "p". (For ROOT)

```
double TFormula_id(double *x, double* p) {
  return x[0] * p[0];
}
clad::gradient(TFormula_id, "p");
```

Extend it to add support for all arrays

```
double f(double* x, double* y) {
  return x[0] * y[0];
}
clad::gradient(f);
```

Mirror all input variables in the generated gradient to store the diff instead of having one variable(_result) contain all the derivatives

```
void f_grad(double *x, double* d_x, double *y, double *d_y)
```
Add array differentiation in reverse mode

**Challenge:**

Clad needs to know the signature of the generated gradient before it derives the function for the `execute` function to work. If only partial arguments are requested, Clad no longer has that information in the planned signature.

**Solution:**

Push all the derived variables to the end, create an overload with all input variables mirrored and use template meta-programming to fill the extra arguments with `nullptr` to emulate default argument like functionality.

```cpp
void f_grad(double *x, double *y, double *d_x, double *d_y) {...}
```
Add array differentiation in reverse mode

**Challenge:**

Clad needs to create a temporary variable when deriving `CallExprs` so it requires the size of the input array.

**Solution:**

```cpp
clad::array_ref
Takes in a pointer to an array and the size of the array and it is used by Clad to create temporaries.

```double g(double *y) {
    return y[0] + y[1];
}

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

clad::gradient(f);
```

```cpp
// Generated gradient signature:
// void f_grad(double *x, clad::array_ref<double> *_d_x)
auto f_gd = clad::gradient(f);

double dx[2];
clad::array_ref<double> dx_ref(dx, 2);
f_gd.execute(x, dx_ref);
```
Add array differentiation in hessian mode

Basic idea:

Call forward mode and then reverse mode on the function for each element in the array.

For this we need to know the size of the array. So hessian mode takes it using its `arg` parameter.

```c
double f(double *x) {
    return x[0] * x[1] * x[2] * x[3];
}
clad::hessian(f, "x[0:3]");
```

The size is specified using the minimum and maximum index that the array takes.
Add support for the new signature of `clad::gradient` and `clad::hessian` in ROOT

**Basic idea:**

ROOT had support for `clad::gradient` so the goal was to just add `clad::array_ref` support.

The plan was to use the ROOT interpreter to create the `clad::array_ref`.

Adding hessian mode support to ROOT was similar to adding gradient support.
Add support for the new signature of `clad::gradient` and `clad::hessian` in ROOT

**Challenge:**

Using the interpreter turned out to have a lot of overhead.

**Solution:**

A struct containing the data members of `clad::array_ref` was used. The trampoline function essentially `reinterpret_cast`s it to `clad::array_ref` and we save ourselves the overhead of including the `array_ref` header in the `TFormula` class definition:

```cpp
struct array_ref_interface {
  Double_t *arr;
  std::size_t size;
};
```
Demo

Hessian mode in ROOT

```cpp
root[0] TFormula f("f", "x*sin([0]) + y*cos([1])");
root[1] double x[] = {3, 4};
root[2] double p[] = {1.57079633, 0};
root[3] f.SetParameters(p);
root[4] TFormula::CladStorage hess(4);
root[5] f.HessianPar(x, hess);
root[6] printf("{%g, %g},\n{%g, %g}\n", hess[0], hess[1], hess[2], hess[3]);
{\ -3, 0},
{0, -4}
```
Acknowledgements

- Mentors: Vassil Vassilev, Ioana Ifrim
- Colleagues: Garima Singh, Parth Arora
- Special Thanks: Lénárd Szolnoki
Thankyou
You can access the full report of my GSoC work [here](#).
For a demo of array differentiation check out this [link](#).