Floating point error estimation using clad

Garima Singh • IRIS-HEP Fellow, Winter 2020 Cohort

Project Mentors

Vassil Vassilev
Princeton University

David Lange
Princeton University

About the project

Develop a floating point error estimation framework using clad, a source transformation AD tool for C++ implemented as a clang plugin. The framework aims to provide the users with the choice of multiple error estimation models and even allow them to add custom estimation models as they wish.

Motivation

```
double c = -5e13;

for(unsigned int i = 0; i < 100000000; i++){

      if(i%2) c = c - 1e-6;

      else c = c + 1e6;

}
```

Exact solution,

$c = -5 \times 10^{13} + ½ * 10^{8} * 10^{6} - ½ * 10^{8} * 10^{-6}$
$= -50$

*Rounding Mode	c
rounded to the nearest	−0.02460...0
rounded towards −∞	−2073773.08... 0
rounded towards +∞	−0.008202... 0
rounded towards 0	−0.008202... 0

Some real life implications:

In January of 1982 the Vancouver Stock Exchange started a stock index accumulating total stock value for all 1,400 stocks listed on the exchange. but truncating (rounding down) that sum up to 3000 times per day resulting in a loss of index value of about $25 per month for about 23 months indicating an index value of $524.811 when the actual value was $1098.892.

*IEEE-754 double precision, Reference :https://www.truenorthfloatingpoint.com/problem

About clad

What is clad?

Clad is a source transformation automatic differentiation tool, implemented as a plugin to the clang compiler.

What is Automatic Differentiation (AD) ?

Simply speaking, it a set of techniques to evaluate the derivative of a function specified by a computer program.

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

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

About FP Error Estimation

A formula for most cases:

$$A_f \equiv \sum_1^n \frac{\partial f}{\partial x_i} . |x_i| . |\varepsilon_M| + E_L$$

This we know already as it is machine dependent

A bit hard to estimate

We can get this via the reverse mode in clad

A_f	The absolute error in a function f.
x_i	All input and intermediate variables.
ε_M	The maximum representational error in a floating point number. Machine dependent.
$\frac{\partial f}{\partial x_i}$	The derivative of f with respect to x_i.
E_L	The error due to linearization the Taylor series expansion.

About FP Error Estimation

What's reverse mode?

$$y = f(g(h(x))) = f(g(h(w_0))) = f(g(w_1)) = f(w_2) = w_3$$

$$\begin{aligned} w_0 &= x \\ w_1 &= h(w_0) \\ w_2 &= g(w_1) \\ w_3 &= f(w_2) = y \end{aligned}$$

forward-mode computes the recursive relation : $\frac{dw_i}{dx} = \frac{dw_i}{dw_{i-1}} \frac{dw_{i-1}}{dx}$

reverse mode computes the recursive relation: $\frac{dy}{dw_i} = \frac{dy}{dw_{i+1}} \frac{dw_{i+1}}{dw_i}$

What does fp error estimation with clad look like?

```
double ApproximatePi(double Tn) {
double e = Tn * Tn;
double tmp = std::sqrt(4 + e);
double Tn1 = 2 * Tn / (2 + tmp);
return Tn1;
}
```

```
void ApproximatePi_grad(double Tn,
double* _result, double& _final_error) {
// ...
{ ...
_delta_Tn1 += _d_Tn1 * _EERepl_Tn1 * Em;
... }
{ ...
_delta_tmp += _d_tmp * _EERepl_tmp * Em;
... }
{ ...
_delta_e += _d_e * _EERepl_e * Em;
... }
_delta_Tn += _result[0UL] * Tn * Em;

_final_error += _delta_e + _delta_tmp +
_delta_Tn1 + _delta_Tn;
}
```

Here, all '_delta' prefixed variables denote the error in the variable they are prefixed to.

At the end, we just add all the errors to get the final absolute error in the function!

How is clad’s fp error estimation different than others?

- Since clad generates error estimation code, means that we can generate error estimates on a variety of input without having to re-evaluate the gradient each time.

- Clad’s error estimation comes off-the-shelf with it, not requiring you to set up complex pipelines to use it.

- Clad’s fp error estimation does not bind the user to a single estimation model. Users can build their own models and use those to calculate estimates with. For example, consider the following user defined custom model…

$$A_f = \sum_{1}^{n} \frac{\partial f}{\partial x_i} \cdot \mid x_i - (float)x_i \mid$$

All the user has to do is turn it into a `clang::Expr`.

Custom models in action

- Once the conversion is complete, the augmented code will look as follows:

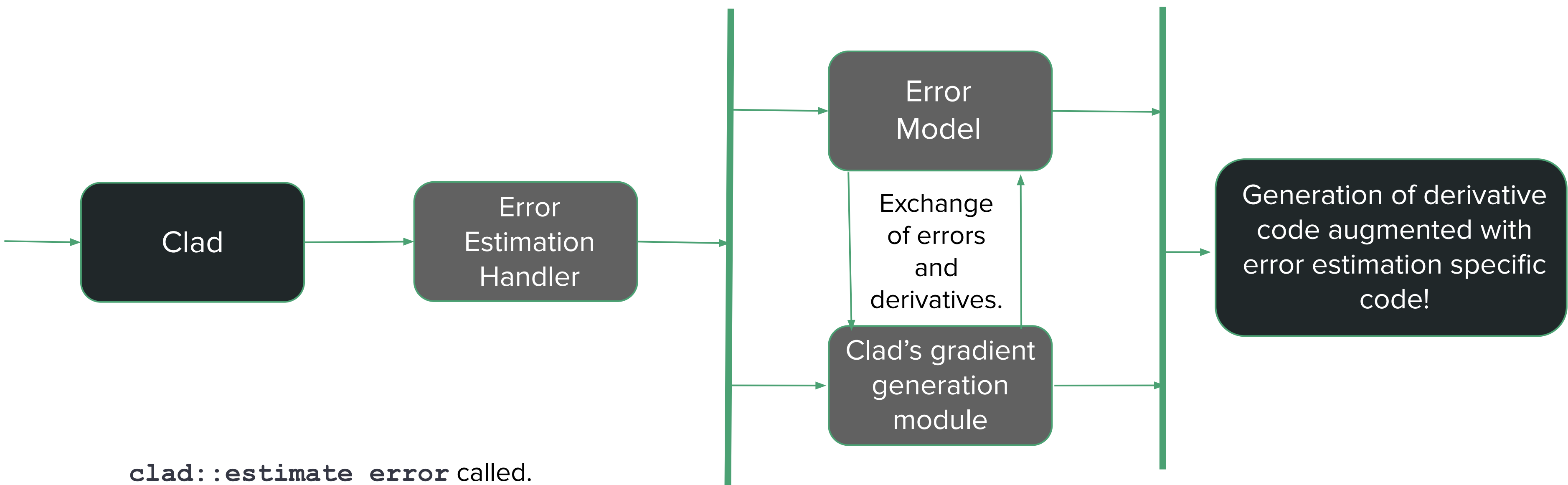
```
double ApproximatePi(double Tn) {
double e = Tn * Tn;
double tmp = std::sqrt(4 + e);
double Tn1 = 2 * Tn / (2 + tmp);
return Tn1;
}
```

clad::estimate_error

+ custom error model

```
void ApproximatePi_grad(double Tn,
double* _result, double& _final_error) {
// ...
_delta_Tn1 += _d_Tn1 * (_EERepl_Tn1 -
(float)_EERepl_Tn1);
// ...
_final_error += _delta_e + _delta_tmp +
_delta_Tn1 + _delta_Tn;
}
```

How it all comes together...



`clad::estimate_error` called.

If a custom model is provided by the user, clad registers it with its `ErrorEstimationModelRegistry`. Otherwise, the in-built default model is used.

Clad transfers all `estimate_error` calls to the handler.

While the derivatives are being generated, clad dispatches calls to the model's function to fetch the "**formula**" for calculating the error estimates. Clad provides the model function with the necessary derivatives and values.

These tasks are interleaved and happen simultaneously (not multithreaded as of yet).

Finally, we get the gradient along with fp error estimation specific code!

An interesting use-case: numerical stability of algorithms

Recall the **`ApproximatePi`** function we used before. Let's assume another function which does the same job -- approximates pi. However it looks a bit different...

```
double ApproximatePi(double Sn){
 double e = Sn * Sn;
 double tmp = std::sqrt(4 - e);
 double Sn1 = std::sqrt(2 - tmp);
 return Sn1;
}
```

Reference: [ADAPT: algorithmic differentiation applied to floating-point precision tuning]

The background setup

We use the **`ApproximatePi`** function iteratively in code as follows…

```
int count = 30;
double Sn= sqrt(2);
double pi;
double n = 4;

for (int i = 1; i < count; i++) {
   Sn = ApproximatePi(Sn);
   n = 2*n;
   pi = n * Sn / 2;
}
```

```
double ApproximatePi(double Sn){
 double e = Sn * Sn;
 double tmp = std::sqrt(4 - e);
 double Sn1 = std::sqrt(2 - tmp);
 return Sn1;
}
```

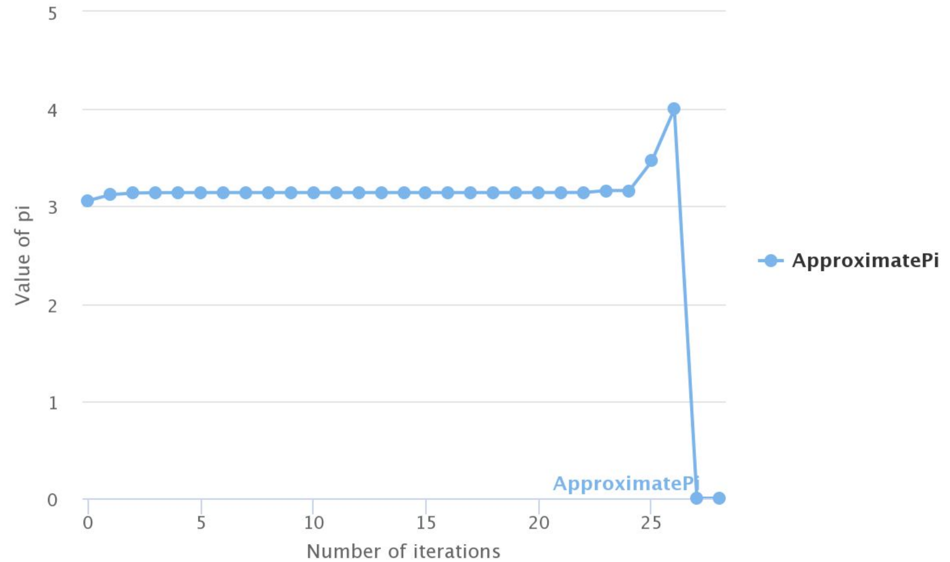
Reference: [ADAPT: algorithmic differentiation applied to floating-point precision tuning]()

Execution results

```
int count = 30;
double Sn= sqrt(2);
double pi;
double n = 4;

for (int i = 1; i < count; i++) {
   Sn = ApproximatePi(Sn);
   n = 2*n;
   pi = n * Sn / 2;
}
```

```
double ApproximatePi(double Sn){
 double e = Sn * Sn;
 double tmp = std::sqrt(4 - e);
 double Sn1 = std::sqrt(2 - tmp);
 return Sn1;
}
```

After executing the code for ~30 iterations, we get the following results:



What is happening at the ~25th iteration?

Reference: ADAPT: algorithmic differentiation applied to floating-point precision tuning

Sensitivity of intermediate variables

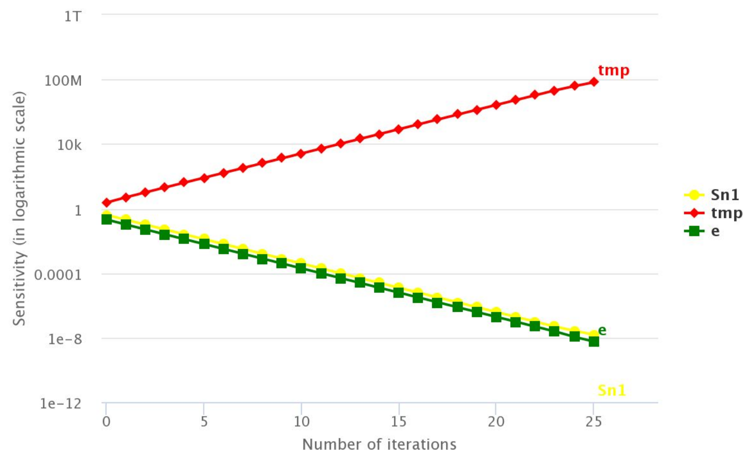
```
int count = 30;
double Sn= sqrt(2);
double pi;
double n = 4;

for (int i = 1; i < count; i++) {
   Sn = ApproximatePi(Sn);
   n = 2*n;
   pi = n * Sn / 2;
}
```

```
double ApproximatePi(double Sn){
 double e = Sn * Sn;
 double tmp = std::sqrt(4 - e);
 double Sn1 = std::sqrt(2 - tmp);
 return Sn1;
}
```

To figure out what is wrong, we can user clad's error estimation to analyse the sensitivity of each intermediate variable. Which (in our case) is given as follows:

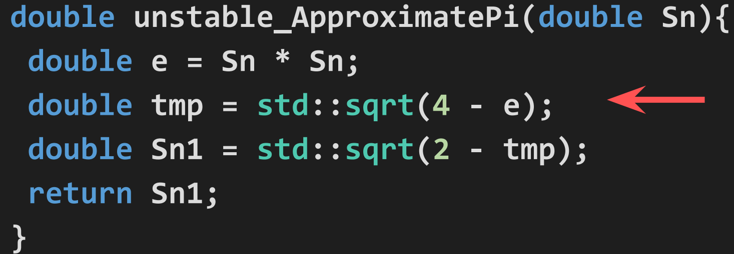
$$S_{x_i} = |\frac{\partial f}{\partial x_i} * x_i|$$

Now, let us look at the results of our sensitivity analysis...



Reference: [ADAPT: algorithmic differentiation applied to floating-point precision tuning]()

Analysing why ‘tmp’ causes instability and proposing a fix



```
double unstable_ApproximatePi(double Sn){
 double e = Sn * Sn;
 double tmp = std::sqrt(4 - e);
 double Sn1 = std::sqrt(2 - tmp);
 return Sn1;
}
```

For values of e closer to 4, the tmp terms results in catastrophic cancellation, causing the future iterations of the algorithm to be unstable.

We can fix this by rationalizing the term containing tmp.

```
double stable_ApproximatePi(double Tn) {
double e = Tn * Tn;
double tmp = std::sqrt(4 + e);
double Tn1 = 2 * Tn / (2 + tmp);
return Tn1;
}
```

Let’s analyse some results to ascertain if the instability is fixed.

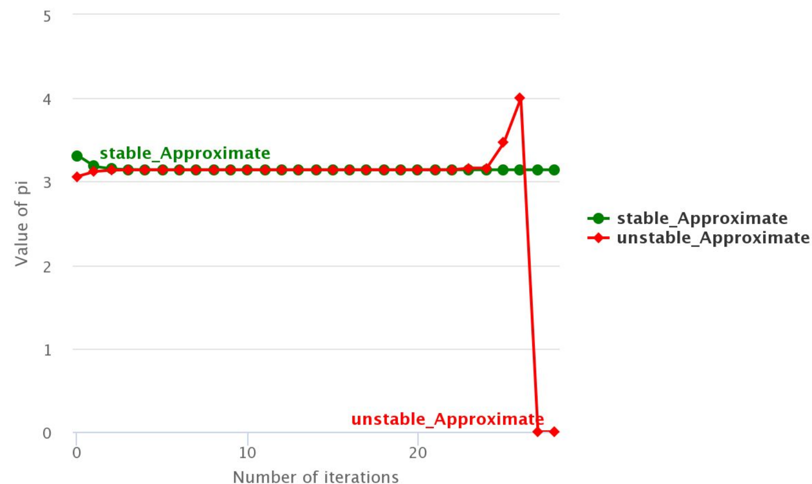
Reference: [ADAPT: algorithmic differentiation applied to floating-point precision tuning]

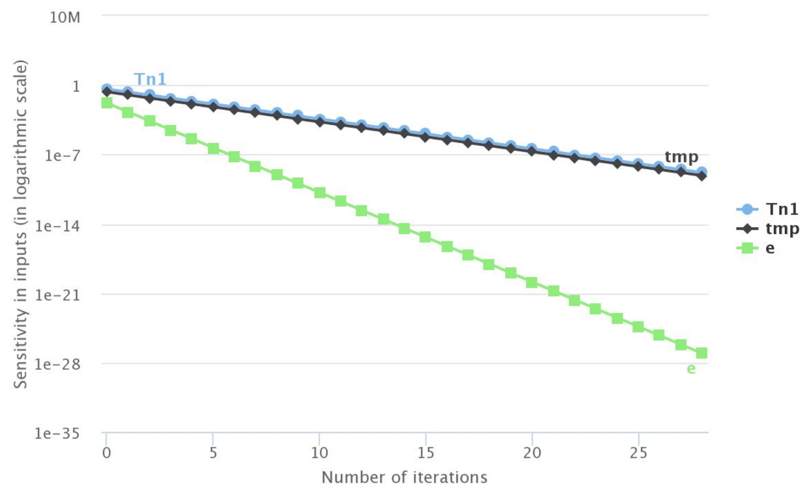
Analysing results of the fix

```
double stable_ApproximatePi(double Tn) {
double e = Tn * Tn;
double tmp = std::sqrt(4 + e);
double Tn1 = 2 * Tn / (2 + tmp);
return Tn1;
}
```

We will analyse 2 things:

- The comparison of results for 30 iterations
- Sensitivity in all intermediate variables





Reference: ADAPT: algorithmic differentiation applied to floating-point precision tuning

Current implementation

- Currently, one can estimate errors over different functions, including ones with complex sub-structures such as nested loops, multiple returns etc. We also partially support error estimation of arrays/pointers (supporting input arrays is in progress). To calculate fp error estimates of a function, one has to simply do the following

```
auto df = clad::estimate_error(myfunction);
df.execute(args...,grad, fp_error);
```

- We also support all types of custom models. The users have to create a shared object and pass it to clad, thereon, clad will register it with the FPErrorEstimationRegistry so that it can be used to generate code later.

- We also support a built-in TaylorApproximation model that users may use straight out of the box. This is the current default and will be used if no custom model is provided.

- We are also working on adding the ability to write the error data for select variables in the target function to a specified output stream. This may help for future analysis of the errors.

Future work

- More generalized error estimations. See how errors propagate through a function. Propagation of uncertainty using automatic differentiation.

- Lossy Compression and Mixed Precision Tuning. The task of reducing the precision of values that have an error lower than a maximum threshold specified.

- Utilize error estimates via clad in ROOT math libraries.

Find out more about clad here.

Thank you!

Github: Grimmmyshini

Email: Garimasingh0028@gmail.com