Garima Singh
Manipal Institute of Technology
garimasingh0028@gmail.com

Project Mentors
Vassil Vassilev
vvasilev@cern.ch
David Lange
David.Lange@cern.ch

Floating Point Error Evaluation With Clad

INTRODUCTION

Floating-point estimation errors have been a testament to the finite nature of computing. Moreover, the predominance of Floating-point numbers in real-valued computation does not help that fact. Float computations are highly dependent on precision, and in most cases, very high precision calculation is not only not possible but very inefficient. Here, one has no choice but to stick to lower precision computing, which in turn is quite prone to errors. These seemingly small individual errors amalgamate into high inaccuracies, which might become difficult to overlook. Due to their effects on computation, it becomes imperative to estimate these errors and prevent unwanted results accurately.

OVERVIEW

Accurate error estimation requires code processing for each dependent and independent variable, making these tasks infeasible to do by hand. One might spin up another algorithm to precisely measure the errors for every floating-point computation; however, this algorithm itself requires arbitrary precision and may become lengthy and cumbersome to do. In this scenario, one might want to *estimate* errors rather than precisely determine them. One approach to achieve this is Automatic Differentiation.

Automatic Differentiation (AD) decomposes a computational graph into atomic operations and then applies differential calculus rules to each operation to obtain the final derivative. AD can be easily extended to estimate floating-point errors via approximating the target function by an estimation model such as Taylor Series, Laurent series, etcetera. There are multiple tools to perform AD, one of which is Clad.

Clad is a source transformation AD tool for C++. It is based on the LLVM compiler infrastructure and is implemented as a plugin for the C++ compiler Clang [1]. Clad borrows Clang’s code generation and parsing functionality to integrate its derivative computation at compile-time seamlessly.

This project aims to develop a generic error estimation framework using Clad that allows users to choose the error estimation mode of choice and generate the augmented functions corresponding to the specified error estimation model. An example of how this can be achieved is illustrated as follows.

Let **y** be a function which can be represented using a linear approximation of the Taylor series expansion,

$$y = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \ldots \approx f(a) + f'(a)(x-a) \tag{1}$$

Then a small change $\triangle y$ can be represented as

$$\begin{aligned} \triangle y &= |f(a+\triangle x) - f(a)| \\ &= |f'(a)\triangle x| \end{aligned} \tag{2}$$

Where $\triangle x$ represents the error in the input variable, for floating-point numbers, this error does not exceed a pre-defined value known as the *machine epsilon (ε)*. The machine epsilon is a small machine-dependent floating-point number that gives the upper bound to rounding off errors. Then for some variable x, the maximum possible absolute error due to rounding off can be represented as follows

$$\varepsilon_x = |x|\varepsilon$$

From [2], we can generalize this solution for vector-valued functions as the scalar product of the Jacobian and the respective errors of each input

$$\triangle y = J_f^T \,.\, ||\triangle x_1|\ |\triangle x_2|\ |\triangle x_3| \cdots|$$

Clad provides the internal functionality of computing Jacobians via `clad::Jacobian.` The task of calculating errors becomes trivial, given an approximation model. The class `clad::JacobianModeVisitor` can then be extended to form a wrapper for the framework's base class. The user may also provide the error estimation model of their choice via an interface that makes estimation model management simple and easy.

Future work here can include a system that identifies code segments that are the most prone to error, providing insightful code statistics. As Menon et al. [2] mentioned, mixed-precision tuning may also be integrated to further improve the code's performance. Work towards lossy compression may also be pursued.

BACKGROUND

Multiple methods exist to estimate floating-point errors, such as precision tuning, variable length arithmetic, interval method, numerical analysis etcetera. Perhaps out of all of those methods, the interval method and numerical analysis are most popular. Below I have taken a code snippet to illustrate how we might be able to generate relevant code for some error estimation methods.

Here is a simple code snippet to calculate the value of `Sin(x)` using Taylor expansion up to `n` terms

```
float sine(float x, int n){

     float denom, sinx, x1 = x;
     for(int i = 1; i<=n; i++){

         denom = 4 * i * i + 2 * i;
         x1 = -x1 * n * n / denom;
         sinx = sinx + x1;
      }

      return sinx;
}
```

Variable Length Arithmetic (VLA)

Variable length arithmetic represents numbers as a string of digits of variable length limited only by the memory available. A good example of VLA is the `BigInteger` class in Java. This seems to be a very straightforward solution to the problem; however, it is significantly slower than all other mentioned methods. It may come in handy if the speed of the program is compromisable.

Interval Arithmetic (IA)

Interval arithmetic is an algorithm for bounding the error of each operation between two limits as opposed to storing just one estimated value. IA determines the lower and upper error bounds of expressions via computing their interval enclosures with careful control of rounding models.

For the above code, let's assume `RL` and `RU` are the lower and upper limits of error in the function respectively. And let there be a class `inv_float,` which essentially acts as a wrapper for float types. The class contains

1. The variable value (`val`)
2. The error range (`range = [rl, ru]; where rl ≤ val ≤ ru`)
3. Overloaded elementary operations (`+, -, /, etc`) to update the error range per operation

Note: I have generalized the class for brevity and have not covered any operator overloads in detail. All intermediate variables are considered separately, and all operator overloads follow the general IA rules and rounding model control.

The intermediate variables are denoted by a leading `_ix.` The generated code then would look something like this,

```
template<float RL, float RU>
float sine_w_err(inv_float x, int n){

     inv_float denom, sinx, x1(x);
     for(int i = 1; i<=n; i++){

         int _i0 = 2 * i + 4 * i * i;
         // implicit cast to denom, default error value assigned to
         // denom.rl, denom.ru
         denom = _i0;

         int _i1 = n * n;
         // following the IA operation rule,
         // S = {x1.rl * _i1.r1, x1.r1 * _i1.ru, x1.ru * _i1.r1, x1.ru * _i1.rl}
         //                  _i2.range := {min(S), max(S)}
         inv_float _i2 = -x1 * _i1

         // Similar to above, except here this operation is seen as
         //                  _i2.range * 1/denom.range
         // And implicitly,
         //                  denom.range := {1/denom.ru, 1/denom.rl}
         // then calculation similar to the above are performed
         inv_float _i3 = _i2 / denom;
         // x1.range := _i2.range on reassignment
         x1 = _i2;

         // For addition,
         //         _i4.range := {sinx.ru + x1.ru, sinx.rl + x1.rl}
         inv_float _i4 = sinx + x1

         sinx = _i4;
      }
      // Assign return variable's error range
      RU = sinx.ru;
      RL = sinx.rl;
```

```
    // Return scalar value
    return sinx.val;
}
```

A con of IA based estimation is that it may seriously overestimate error boundaries and may fail to provide a tight error bound. This limitation may be overcome by coupling IA with AD or other estimation methods.

A Step Towards Automated Testing and Reasoning

Satisfiability Modulo Theories (SMT) solvers essentially solve a set of constraints using symbolic execution. This constraint solving makes SMT solvers a great tool to judge programs' accuracy and robustness and essentially prove that they *work*. And naturally, considering all of the discussion regarding the inaccuracy of floats representing real numbers, a lot of work has been done to model Floating-Point Arithmetic (FPA) as a set of decision procedures that can then be verified by SMT solvers.

FPA is vastly different from real arithmetic because it does not follow many of the latter's rules. Because of this, testing programs with FPA becomes increasingly difficult. Here, the SMT theory of FPA comes in great help to emulate floating-point numbers as real numbers with *infinite* precision and generate code to reason these float numbers. Most SMT solvers use a bit-blasting technique wherein floating-point theory is reduced to bit-vectors, and the corresponding operations are modeled as circuits [4]. Some famous SMT solvers that support FPA are Z3[5] and MathSAT[6].

Coupling an SMT solver with the above-generated code via IA (or any other estimation method) one may be able to identify the points of inaccuracy of the code, which may help in improving code correctness. One such approach is implemented by Daisuke et al. [7] where they add additional lemmas and automated code annotation to different SMT backends for verification of float IA operations.

Monte Carlo Arithmetic (MCA)

MCA's error analysis is accomplished by repeatedly injecting small errors into an algorithm's data values and determining the relative effect on the results. These small errors are usually injected by random rounding and unrounding (random extension of precision) and serve as a purely unbiased method of fabricating random data.

A quick gist of how MCA works is as follows;

Float computation ➡ *injection of randomness* ➡ *Monte Carlo computation*

Round-off analysis ➡ *Statistical analysis*

This statistical analysis provides a good measure of a particular real valued function's sensitivity with respect to small errors in its floating-point inputs. To model errors on a FP value *x* at virtual precision (random precision) *t*, Parker et al. [8] propose the following function:

$$inexact(x) = x + 2^{e_x}\xi$$

Where e_x is the exponent of x and ξ refers to a uniformly distributed random variable in the range `[-0.5, 0.5]`. Each floating point operation `x*y` can then be transformed into an MCA FP operation by employing one of the following models.

- Random Rounding: introduces errors in the result of the operation.

  `x*y → round(inexact(x*y))`

- Precision Bounding: introduces errors in the respective inputs of the operation.

  `x*y → round(inexact(x)*inexact(y))`

- Full MCA: introduces errors in the result and inputs of the operation.

  `x*y → round(inexact(inexact(x)*inexact(y)))`

Parker et al. also show that the significant digits of a MCA result at virtual precision *t* is given by the magnitude of the relative standard deviation of a distribution which can be estimated by a large number of Monte Carlo trials,

$$s = -log_{\beta}\frac{\sigma}{\mu}$$

Where σ represents the standard deviation, μ represents the means and β represents the base of the numbers.

GOALS

1. Develop the generic error estimation framework over Clad
2. Implement a rich and scalable interface for the estimation framework
3. Improve the current Clad AD implementations and add missing features to provide accurate estimation results

PERSONAL INFORMATION

I am an Information Technology Undergraduate currently in my 3rd year of study. I have prior experience working with C++, Java, and Python. I am a Physics and Robotics enthusiast and have worked on a multitude of things from self-driving cars to developing point matching algorithms from the ground up.

I recall being stuck on a problem which involved floating-point computations. It was indeed a nightmare figuring out how to get across the floating-point errors without compromising the algorithm's simplicity and clarity. Having experienced the importance of error estimation first hand and how much of an influence it has over the computation primarily serves as a motivation for me to be working on this project.

Availability: ~35 hours/week, more during college holidays.

Timezone: Indian Standard Time (GMT +5:30)

I have also opened two pull requests in clad, here and here.

TIMELINE

Weeks 1	• Build familiarization with the codebase, solve some issues to get a better insight into the code • Test out a few error estimation methods using clad independently to get a better understanding of the best way to pursue development • Prepare a report on different error estimation techniques
Weeks 2-4	• Design and discussion of the framework architecture, with a major focus on the modularity and object-oriented nature of code • Select a good estimation model (for now, Taylor Series) as a starting point to approximate the target function and perform calculations as mentioned in the previous sections • The starting would be to extend the `Derive` method in Reverse AD and the `JacobianModeVisitor` to calculate the partial derivatives of the function with respect to each function argument and intermediate variable • Present the possible design choices and decide on a design
Weeks 5-6	• Now the focus shifts on accumulation of errors per function argument/intermediate variable so that they can be used to derive the final output error as illustrated in the overview section. • Write intermediate tests to verify the proper working of the development till now
Weeks 7-8	• Utilize results from previous weeks, wrap up the development of estimation for the Taylor model, write a few tests and debug if necessary • Find several real world error estimation examples for verification of the implementation
Weeks 9-10	• Think of how to incorporate support for user-defined datatypes like classes, structs etcetera and possibly incorporate it into Clad as well as the estimation framework • Add more estimation models and test them out as necessary • Investigate the implementability of a SMT-based error estimation in clad

Week 11	● Move the focus to writing tests and solving issues in the code. Also, start writing the usage and contribution guides. ● Benchmark against popular floating point error estimation tools and methods. FPBench[3] can be used for this purpose.
Week 12	● If time permits, implement a method to overcome *cancellation errors* by approximating target functions using Taylor series expansion. This could either be an extended functionality in the error estimation framework or could be demoed separately. The primary motivation to do this is to encompass both kinds of floating-point errors - rounding and cancellation. ● Buffer for debugging and solving issues that may arise during the development phase. ● Build a MCA analysis demo

REFERENCES

[1] Vassilev, Vassil, Penev, Alexander, & Shakhov, Roman. (2020). Error estimates of floating-point numbers and Jacobian matrix computation in Clad. Zenodo. http://doi.org/10.5281/zenodo.4134097

[2] Menon, H, Lam, M, Kuffour, D, Schordan, M, Llyod, S, Mohror, K, and Hittinger, J. ADAPT: Algorithmic Differentiation for Floating-Point Precision Tuning. United States: N. p., 2018. Web.

[3] Toward a Standard Benchmark Format and Suite for Floating-Point Analysis, NSV'16: N. Damouche, M. Martel, P. Panchekha, C. Qiu, A. Sanchez-Stern, and Z. Tatlock

[4] Brain M., Schanda F., Sun Y. (2019) Building Better Bit-Blasting for Floating-Point Problems. In: Vojnar T., Zhang L. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2019. Lecture Notes in Computer Science, vol 11427. Springer, Cham. https://doi.org/10.1007/978-3-030-17462-0_5

[5] de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24

[6] Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_7

[7] Ishii, Daisuke and Tomohito Yabu. "Computer-Assisted Verification of Four Interval Arithmetic Operators." ArXiv abs/2003.10623 (2020)

[8] Scott Parker. Monte carlo arithmetic: exploiting randomness in floating-point arithmetic. Technical Report CSD-970002, UCLA Computer Science Dept., 1997.