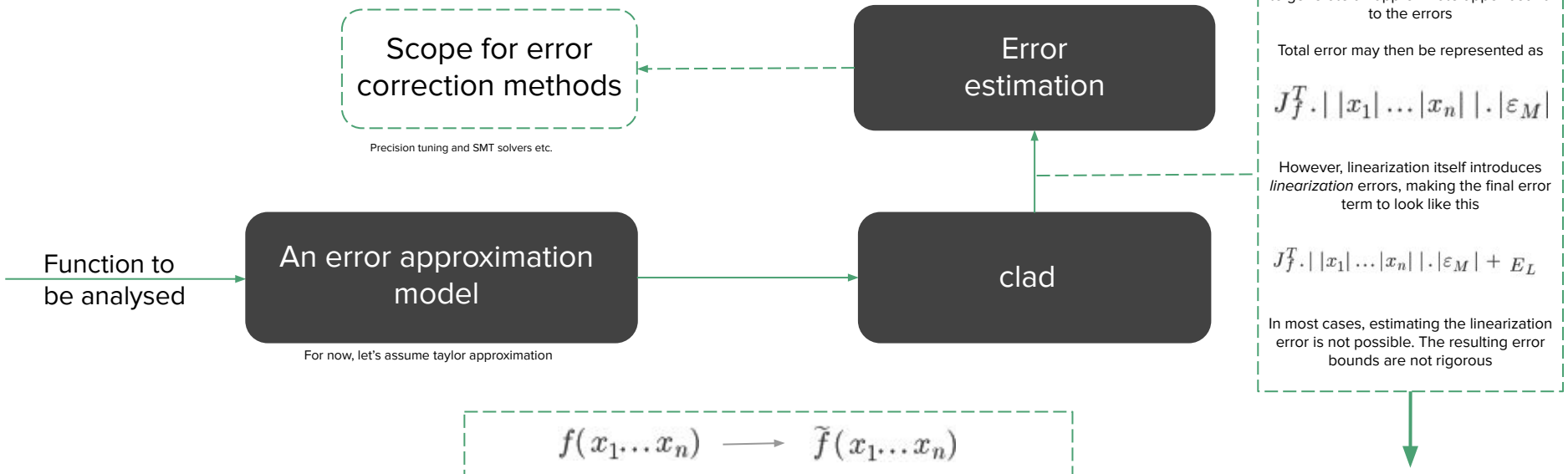
Project Roadmap

IRIS-HEP Fellowship • Floating point error estimation using Clad

Overview

A Simplified Overview



Reference: https://link.springer.com/referenceworkentry/10.1007%2F978-0-387-74759-0_26#Equ9_26

In a gist

Absolute error may be represented as,

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

Due to linearization, a more rigorous bound will be

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

This we know already as it is machine dependent

A bit hard to estimate

We can get this via the reverse mode in clad

Adjoint AD and Clad

The given computational graph represents the flow of adjoint AD method, which is implemented similarly in Clad.

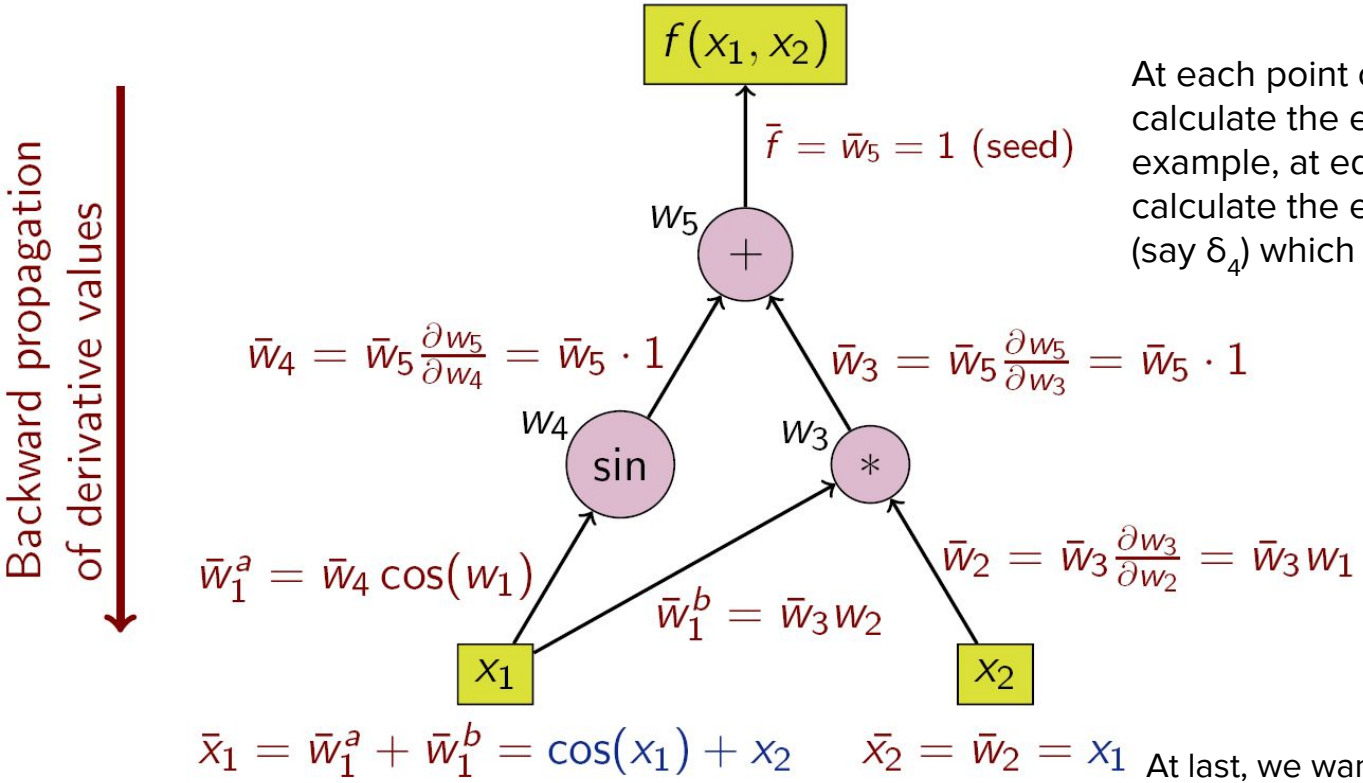


Image credit: https://en.wikipedia.org/wiki/Automatic_differentiation

At each point of derivative calculation, we can calculate the error of the subexpression. For example, at edge $W_4 \rightarrow W_5$, we want to calculate the error of the subexpression W_4 (say δ_4) which approximately equals $|W_4| \cdot |\varepsilon_M|$

At last, we want to realise the equations given in the slides before to get the final error term

Adjoint AD and Clad

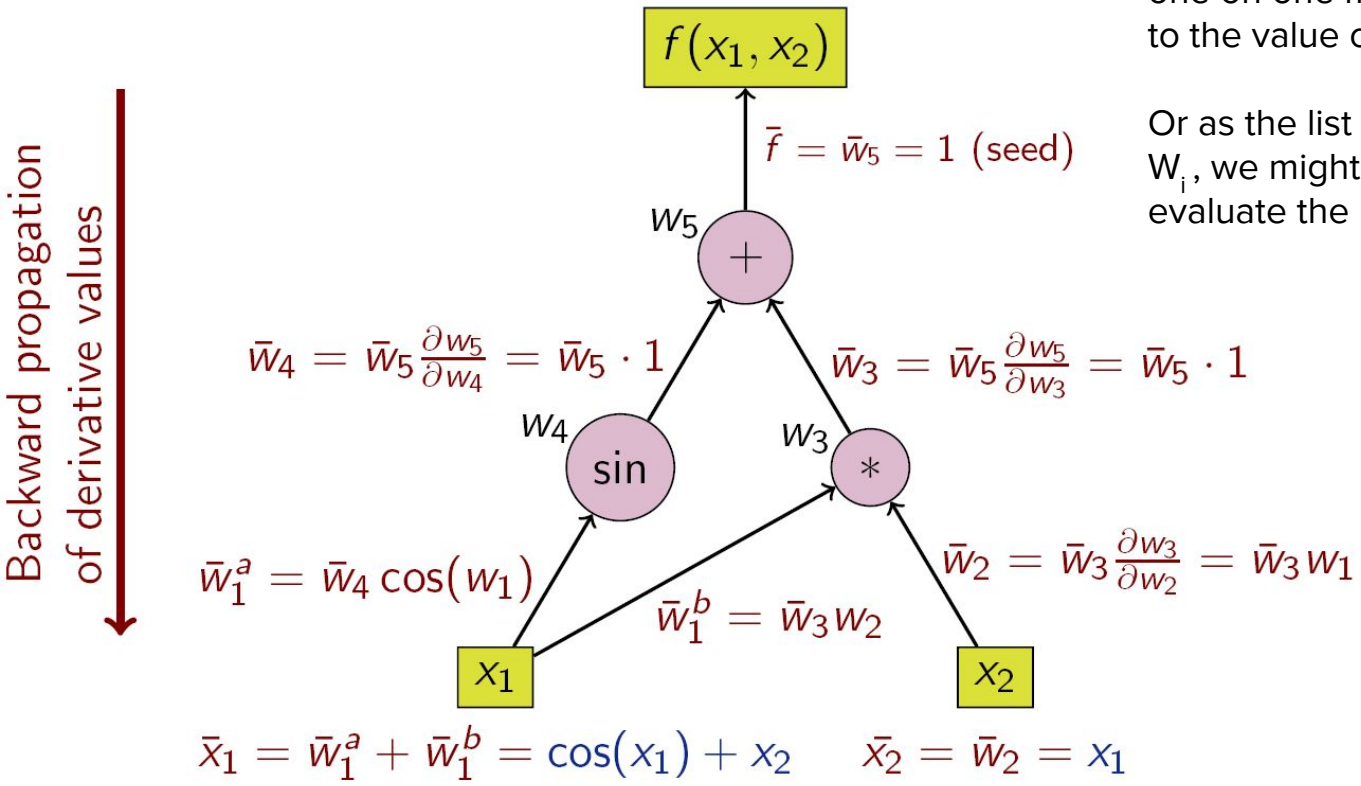


Image credit: https://en.wikipedia.org/wiki/Automatic_differentiation

A way of achieving this might be creating a one on one mapping of the AD Wengert's list to the value of the expression at runtime.

Or as the list already stores the expression at W_i, we might take a running sum and evaluate the list.

Towards better error computation

A very obvious flaw of the elementary approach might be the effect of linearization errors on the final error term.

This problem is somewhat mitigated in the methods of error estimation discussed hereafter.

Existing Software that use this method (in conjunction with others)

ADAPT: Algorithmic Differentiation for Floating-Point Precision Tuning

Towards automatic significance analysis for approximate computing

Interval estimates

Substitute each basic operation with an operation which takes in an interval $[x^l, x^u]$, and produces the narrowest possible interval that contains the result of the corresponding interval operation with the same arguments.

As an example,

```
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;
}
```

Interval estimates

```
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;
}
```

Here, `[RL, RU]` becomes the interval (say F) now contains both f and $\tilde{f}$

Computing A_F

In a similar manner, we can extend IA to calculate A_F as follows,

$$A_F \equiv | \bigoplus_{j=1}^{r} \overline{W}_j \otimes \overline{\Delta}_j |$$

Where,

$\bigoplus$ is summation in IA

$\otimes$ is multiplication in IA

$\overline{\Delta}_j = [\, -\delta_j, \delta_j \,], where\ \delta_j = max(\, |v_j| \,.\, \varepsilon_M, pmin, -nmax)$ where pmin and nmax are the minimum positive floating-point number and the maximum negative floating-point number.

Interval computability and hurdles

A major problem with IA based estimation methods is that the interval becomes larger as the computational steps grow, giving a large output interval and an overestimation of errors.

Difficulty in control flow and evaluation of relation statements also arise. For example, consider the condition $a < [x]$ where $[x]$ refers to the interval of x and $a \in [x]$, the result of the comparison becomes ambiguous. Same is the case with comparing two overlapping intervals

For a division operation involving $[x]$ where $0 \in [x]$, there are chances of a division by zero error in which case, the analysis may have to be terminated

To combat the last two issues, one may want to look into interval splitting as described here: https://www.math.kit.edu/ianm2/~kulisch/media/arjpkx.pdf

References: Vassiliadis, V., Riehme, J., Deussen, J., Parasyris, K., Antonopoulos, C. D., Bellas, N., … Naumann, U. (2016). *Towards automatic significance analysis for approximate computing; KUBOTA, Koichi & Iri, Masao. (1988). Fast Automatic Differentiation and Interval Estimates of rounding errors. Inf. Process. 14.*

Other interesting methods

SMT solvers

Continuing the spirit of IA arithmetic, another interesting topic might be reasoning about an algorithms' accuracy.

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. 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.

SMT theory of Floating Point Arithmetic (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. Some famous SMT solvers that support FPA are Z3 and MathSAT.

SMT Solvers and Integration

Integration of an SMT solver with Clad will roughly require the following steps:

- Code Annotation

In house code annotation will enable us to use existing SMT solvers. Generating code may also enable us to make use of libraries like symFPU on any other FPA supporting SMT solver directly as backend. These annotations maybe done by hand by the user, automatically or a mix of both.

- Setting preconditions and other dynamic parameters

Interfacing an SMT solver with the application we create will require setting of parameters that will define the behavior of the FPA verification. This is the fairly trivial part to implement.

Further reading

Computer-assisted verification of four interval arithmetic operators; Daisuke Ishii, Tomohito Yabu

- Verify IA libraries with code annotation + SMT backend
- Have a GUI and support manual setting of preconditions/postconditions and splitting verification conditions

Paganelli, G., & Ahrendt, W. (2013). Verifying (In-)Stability in Floating-Point Programs by Increasing Precision, Using SMT Solving.

- Checks if a calculation is stable by evaluating it at different precisions and checking if the resulting answers have a reasonable distance between them.
- Negates a weak precondition and solves it on two different precisions

W. Kahan, “How futile are mindless assessments of roundoff in floating-point computation?” 2006

- Just an interesting read!

Other interesting methods

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*

MCA Computation

To model errors on a Floating Point (FP) value *x* at virtual precision (random precision) *t*, Parker et al. 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)))
```

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

MCA Computation

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,

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

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

These Monte Carlo trials can be easily parallelized and hence make this method a feasible way to obtain error estimates for a given function. Clad can also use the existing MCA library backends with modification to aid in the calculations as is done by Verificarlo

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

Further Reading

Verificarlo | Verificarlo: checking floating point accuracy through Monte Carlo Arithmetic

- Existing tool based on MCA to analyse and debug FPE
- Talks about MCA and Discrete Stochastic Arithmetic
- Interesting case studies and comparisons with DSA

Stott Parker, D., Pierce, B., & Eggert, P. R. (2000). Monte Carlo arithmetic: how to gamble with floating point and win

- Interesting read on how MCA can help

Misc.

Goals

1. Cement my ideas and keep looking around for better ideas
2. As per the schedule, start working on the report on different error estimation methods
3. Familiarize myself with clad and all the clang functionalities it utilizes.