# Floating-Point Error Estimation Using Automatic Differentiation

SIAM Uncertainty Quantification Mini-Symposium 2022



*Vassil Vassilev,*
*Research Software Engineer,*
*Princeton University*



*Garima Singh,*
*Undergrad Student,*
*Manipal Institute of Technology*
*Princeton University*

# Motivation: Floating-Point Errors

- Floating-point (FP) errors can have severe implications for programs with high-precision calculations or simple, but repetitive computations.

```
double c = -5e13;
for (unsigned int i = 0; i < 1e8; i++){
    if (i % 2) c = c - 1e-6;
    else c = c + 1e6;
}
```

Exact solution,
$c = -5 \times 10^{13} + \frac{1}{2} * 10^8 * 10^6 - \frac{1}{2} * 10^8 * 10^{-6}$
$= -50$

| *Rounding Mode | c |
|---|---|
| Rounded to the nearest | −0.02460 |
| Rounded towards −∞ | -207373.07020 |
| Rounded towards +∞ | −0.00820 |
| Rounded towards 0 | −0.00820 |

*IEEE-754 double precision. Example from: Problem — True North Floating Point

- Using higher precision is one solution, but this leads to larger program sizes. Other approaches require reimplemenation to prevent error accumulation (such as Kahan summation).

# Floating Point Errors in Data Intensive Science

- Floating point stability is important for data intensive sciences work with increasing data volume and often in heterogeneous computing environments. For example, float vs. double can make a large difference in performance as many GPUs either do not provide double-precision float, do not follow the IEEE  compliance or have fairly slow access.

- High Energy Physics is an example field seeing a large influx of data. Huge data rates (100s PB/yr this decade) require selectively saving data and to optimally save the data to optimize as much space as possible. Robust and automatized detection of floating-point errors can help reducing the data re-processing costs and foster development of important new lossy compression algorithms.
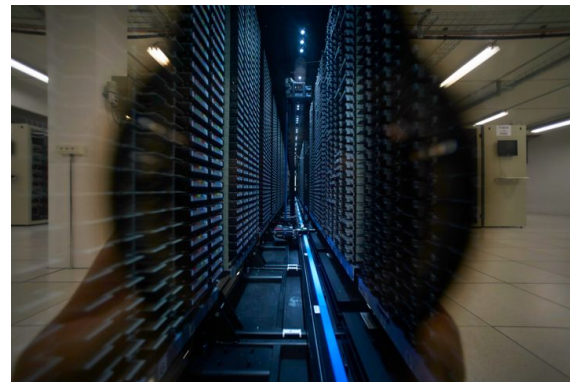


In October 2017 the CERN data centre broke its own record for data storage when it collected 12.3 petabytes of data over a single month. Breaking data records bit-by-bit

# Estimating Floating Point Errors

- The absolute error due to floating point limitations in a function can be expressed using a Taylor series. In one dimension:

$$f(x+h) = f(x) + \frac{h}{1!}f'(x) + \frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + \ldots$$

where "$h$" represents the upper bound on the floating point error of $x$.
The maximum error induced on function $f$ is then

$$A_f = |f(x+h) - f(x)|$$

To first order in $h$, this becomes

$$A_f \equiv |hf'(x)|$$

- The machine epsilon gives the maximum relative representation error in floating point variables due to rounding: (machine dependent while following IEEE standard)

$$h = |x|\varepsilon_M$$

- Then the absolute error is simply:

$$A_f \equiv |f'(x_i)| \cdot |x_i| \cdot \varepsilon_M$$

# Our Approach to Estimating FP Errors

- A more general representation of floating point errors in arbitrary dimensions is then:

*We can get this from automatic differentiation.*

*We know this - machine dependent.*

$$A_f \equiv \sum_{i=0}^{n} |\frac{\partial f}{\partial x_i}| \cdot |x_i| \cdot \varepsilon_M + E_L$$

*Approximation error - hard to estimate.*

$A_f$   The absolute error in a function f.

$x_i$   All input and intermediate variables.

$E_L$   The error due to linearization of the Taylor series expansion.

$\varepsilon_M$   The upper bound on the relative approximation error due to rounding. Machine dependent.

$\frac{\partial f}{\partial x_i}$   The derivative of f with respect to $x_i$.

- This formula gives a good upper bound estimate to $A_f$, and serves our general purpose use case. Automatic differentiation (AD) is an efficient means to compute the needed derivatives

# AD Evaluates the Exact Derivative of a Function

- AD applies the chain rule of differential calculus throughout the semantics of the original program.
- For a complex nested function, two recursive relationships calculate the derivative.

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

*Forward-mode AD:* $\dfrac{dw_i}{dx} = \dfrac{dw_i}{dw_{i-1}} \dfrac{dw_{i-1}}{dx}$

*Reverse-mode AD:* $\dfrac{dy}{dw_i} = \dfrac{dy}{dw_{i+1}} \dfrac{dw_{i+1}}{dw_i}$

- In the context of FP error estimation, we use reverse-mode AD as it provides the derivative of the function with respect to all intermediate and input variables.
- There are multiple approaches to implementing AD on programs, including Operator Overloading and Source Transformation.

# Clad: Source-Transformation AD Tool

- [Clad](#) is implemented as a plugin to the clang compiler. It inspects the internal compiler representation of the target function to generates its derivative.

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

`clad::differentiate(sqr, "x")`

```
double sqr_darg0(double x){
    double _d_x = 1;
    return _d_x * x + x * _d_x;
}
```

Advantages of Clad's approach

- Requires little or no code modification for computing derivatives of existing codebase.
- Supports a growing subset of C++ constructs, statements and data-types as well as differentiating CUDA-based programs.
- Enables efficient gradient computation (independent time complexity from inputs) in its reverse accumulation mode enabling scalable FP error-estimation.
- Well integrated into the compiler, allowing automatic generation of error estimation code.

# AD-Based FP Error Estimation Framework
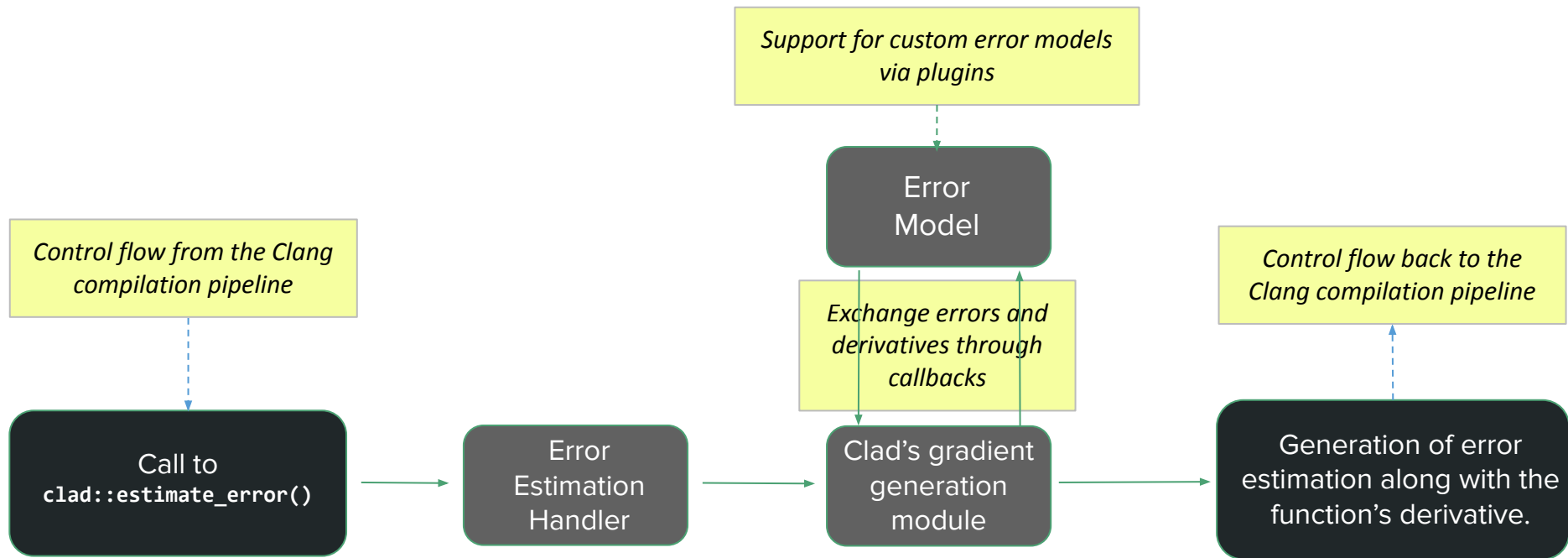
# AD-Based FP Error Estimation Framework

```
float func(float x, float y) {
 float z;
 z = x + y;
 return z;
}
```

*A function we want to estimate the error of.*

```
auto df = clad::estimate_error(func);
```

*In the main function, we call a function
that tells clad to estimate the error in
'func'.*

```
float x = 1.95e-5, y = 1.37e-7;
float dx = 0, dy = 0;
double fp_error = 0;
df.execute(x, y, &dx, &dy, fp_error);
std::cout << "FP error in func: " << fp_error;
```

*Finally, call the generated function through the 'execute'
interface of clad objects. After the execution, the last parameter
will store the accumulated FP error in the function.*

# Sensitivity Analysis

- Our uncertainty framework enables a sensitivity analysis of all input and intermediate variables to floating point errors and reason about the numerical stability of algorithms.

$$S_{x_i} \equiv |\frac{\partial f}{\partial x_i} \cdot x_i|$$

- If the sensitivity of any variable is high, then the function is more prone to floating-point errors in computations involving that variable.
- This information can be useful when developing programs. The appropriate precision can be used throughout the program given a requirement on floating-point accuracy of the result.

One example use case is **mixed precision tuning**: "demoting" certain types to lower precision while still maintaining the desired accuracy can be beneficial for optimizing speed, size and precision.

# Case Study: Simpson's Rule

The program on the right side is used to to evaluate the integral of a function using Simpson's rule for numerical integration.

```cpp
// defines f(x) = pi * sin(x * pi)
// integral = 2 over [0, 1]
long double f(long double x) {
 long double pi = M_PI;
 long double tmp = x * pi;
 long double tmp2 = sin(tmp) * pi;
 return tmp2;
}
```

The function is implemented with all variables in extended precision. Is that necessary?

```cpp
long double simpsons(long double a, long double b) {
int n = 1000000;
// calculates the integral of a function f
// over the interval [a, b] for n iterations.
long double h = (b - a) / (2.0 * n);
long double x = a;
long double tmp;
long double fa = f(a), fb = f(b);
long double s1 = fa;

for(int l = 0; l < n; l++) {
  x = x + h;
  s1 = s1 + 4.0 * f(x);
  x = x + h;
  s1 = s1 + 2.0 * f(x);
}

s1 = s1 - fb;
tmp = h / 3.0;
s1 = s1 * tmp;
return s1;
}
```

Example adapted from ADAPT: Algorithmic Differentiation Applied to Floating-Point Precision Tuning.

# Clad's Results

| Precision configurations<br>`simspons(0, 1)` | Absolute Error | Clad's Estimated Upperbound |
|---|---|---|
| 10-byte extended precision (*long double*) | 4.1e-14 | 3.1e-12 |
| IEEE double-precision (*double*) | 6.8e-11 | 6.2e-9 |
| IEEE single-precision (*float*) | 0.038 | 3.31 |

- Clad provides a fair estimate for the asymptotic error bound on the FP errors.
- More accurate estimates can be derived using custom error models with the FP error estimation framework.

```
long double simpsons(long double a, long double b) {
int n = 1000000;
// calculates the integral of a function f
// over the interval [a, b] for n iterations.
long double h = (b - a) / (2.0 * n);
long double x = a;
long double tmp;
long double fa = f(a), fb = f(b);
long double s1 = fa;

for(int l = 0; l < n; l++) {
  x = x + h;
  s1 = s1 + 4.0 * f(x);
  x = x + h;
  s1 = s1 + 2.0 * f(x);
}

s1 = s1 - fb;
tmp = h / 3.0;
s1 = s1 * tmp;
return s1;
}
```

# A Peek Into the Generated Code

*The **_final_error** variable gives us the fp error accumulated throughout the function.*

```
long double f(double x) {
 long double pi = M_PI;
 long double tmp = x * pi;
 long double tmp2 = sin(tmp) * pi;
 return tmp2;
}
```

*A "**_delta_*"** variable is created for each LHS operand of all assignment operations in the target function to assess the contribution of each variable to the total FP error*

```
void f_grad(long double x, clad::array_ref<long double> _d_x,
            long double &_final_error) {
  // . . .
  _d_tmp2 += 1;
  {
      // . . .
      _delta_tmp2 += std::abs(_d_tmp2 * _EERepl_tmp20 * 1.0842021724855e-19);
      // . . .
  }
  {
      // . . .
      _delta_tmp += std::abs(_d_tmp * _EERepl_tmp0 * 1.0842021724855e-19);
      // . . .
  }
  _delta_pi += std::abs(_d_pi * _EERepl_pi0 * 1.0842021724855e-19);
  long double _delta_x = 0;
  _delta_x += std::abs(* _d_x * x * 1.0842021724855e-19);
  _final_error += _delta_x + _delta_tmp2 + _delta_tmp + _delta_pi;
}
```

# Choosing Variables for Precision Tuning

Clad calculates the total FP error contribution of every variable, we can then ask clad to print this information to some file (let's assume the file is called 'errors'). Clad will print the error for each variable as follows:

*variable-name*: *error-value*

Note: It is also possible to configure what clad prints i.e. the value of 'error-value' can be customized to print derivatives or even sensitivities of variables.

With this information, it is trivial to filter out the variables whose error violates a certain boundary or threshold.

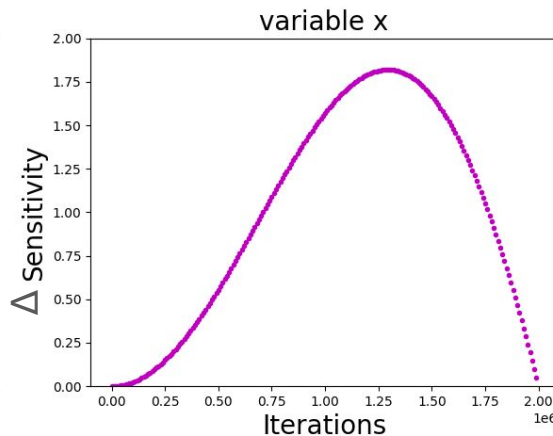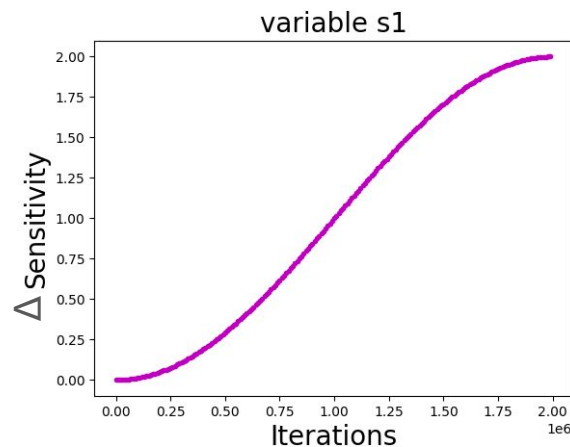These are the variables we want to keep in higher precision.

```cpp
long double simpsons(long double a, long double b) {
int n = 1000000;
// calculates the integral of a function f
// over the interval [a, b] for n iterations.
long double h = (b - a) / (2.0 * n);
long double x = a;
long double tmp;
long double fa = f(a), fb = f(b);
long double s1 = fa;

for(int l = 0; l < n; l++) {
  x = x + h;
  s1 = s1 + 4.0 * f(x);
  x = x + h;
  s1 = s1 + 2.0 * f(x);
}

s1 = s1 - fb;
tmp = h / 3.0;
s1 = s1 * tmp;
return s1;
}
```

# Choosing Variables for Precision Tuning

- Our analysis identified that variable **s1** and **x** (in green) have the highest error value when compared with a threshold of 1e-12.
- Plotting the changes in sensitivity of **s1** and **x** across iterations (in purple) gives hints about how to apply further optimizations.

```cpp
long double simpsons(long double a, long double b) {
int n = 1000000;
// calculates the integral of a function f
// over the interval [a, b] for n iterations.
long double h = (b - a) / (2.0 * n);
long double x = a;
long double tmp;
long double fa = f(a), fb = f(b);
long double s1 = fa;

for(int l = 0; l < n; l++) {
  x = x + h;
  s1 = s1 + 4.0 * f(x);
  x = x + h;
  s1 = s1 + 2.0 * f(x);
}

s1 = s1 - fb;
tmp = h / 3.0;
s1 = s1 * tmp;
return s1;
}
```



variable s1



variable x

# Case Study: Simpson's Rule
## Results

| Precision configurations | Absolute Error | Clad's Estimated Upperbound | Variables in lower precision (out of 11) |
|---|---|---|---|
| 10-byte extended precision (*long double*) | 4.07e-14 | 3.1e-12 | 0 |
| Clad's mixed precision | 4.08e-14 | 3.0e-12 | 6 |
| IEEE double-precision (*double*) | 6.8e-11 | 6.2e-9 | - |
| IEEE single-precision (*float*) | 0.038 | 3.31 | - |

"Demoting" low-sensitivity variables to lower precision improves performance by ~10% in this example.

Clad's estimate also agrees that there is no significant change in the final error. This can be useful in the cases where an accurate ground-truth comparison is not available.

```cpp
long double f(long double x) {
 long double pi = M_PI;
 long double tmp = x * pi;
 long double tmp2 = sin(tmp) * pi;
 return tmp2;
}
```

```cpp
long double simpsons(double a, double b) {
int n = 1000000;

double h = (b - a) / (2.0 * n);
long double x = a;
double tmp;
double fa = f(a), fb = f(b);
long double s1 = fa;

for(int l = 0; l < n; l++) {
  x = x + h;
  s1 = s1 + 4.0 * f(x);
  x = x + h;
  s1 = s1 + 2.0 * f(x);
}

s1 = s1 - fb;
tmp = h / 3.0;
s1 = s1 * tmp;
return s1;
}
```

# Case Study: Simpson's Rule
## Further Possible Optimizations

- Another interesting point to note here is how the changes in sensitivity of variables vary with iterations. This exposes another point for optimizing the mixed precision configuration.
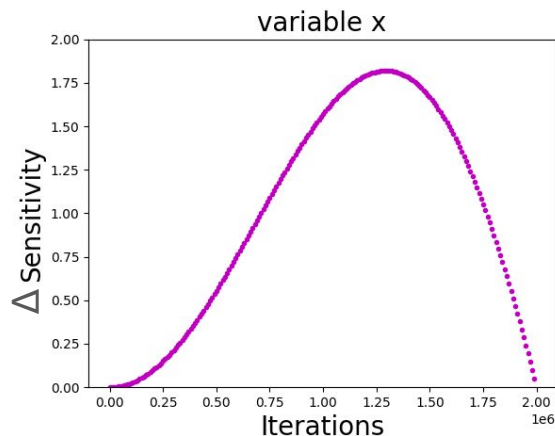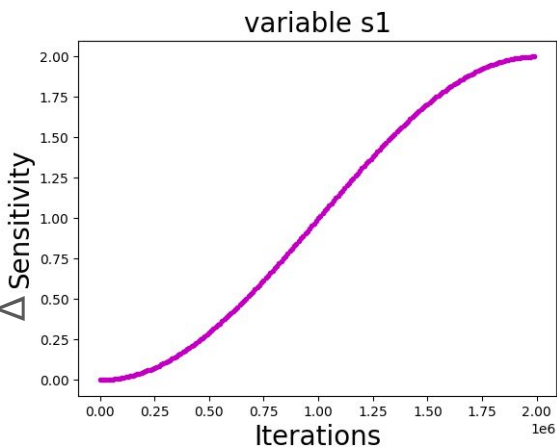


For example, variable x can be in higher precision towards the beginning of the iterations and then switch to lower precision midway.

However, this is only recommended if the variables with high sensitivity are not strongly related as in this case.

Here, splitting the loop, gives you about the same speedup and similar accuracy as the double precision implementation.

# Using Custom Models

Clad supports the usage of custom defined models:

1. Implement the **clad::FPErrorEstimationModel** class, a generic interface that provides the error expressions for clad to generate.
2. Override the **AssignError()** function. This function is called for all LHS of every assignment expression in the target function.

The function **AssignError()** essentially represents the mathematical formula of an error model in a form that clang can understand and convert to code. It provides users with the reference to the variable of interest and its derivative. The user in turn has to return an expression which will be used to accumulate the error.

As of now, filling in these functions requires knowledge of the clang APIs. We plan to provide higher level APIs to make it simpler to use, but meanwhile we are happy to provide assistance in writing custom models if necessary!

For more information on how to build a custom model from scratch, check here! And for more information the FP error estimation framework, check here!

# Summary

- AD based FP error analysis enables understanding the largest contributions to FP errors and enables mixed-precision program optimization.
- Clad has a novel, customizable, AD-based error estimation framework that automates FP error analysis, available with `conda install clad`.
- We demonstrated a case study for sensitivity analysis of a Simpson's rule program
- We illustrated how to create a custom FP error analysis model capable of incorporating domain-specific knowledge

We plan to add functionality to use the mixed precision recommendations to automatically generate optimized code which will allow further research in the area of lossy compression.

https://github.com/vgvassilev/clad | Binder - Jupyter Notebook

Thank you!