Compiler-Assisted Relative Error Analysis for Floating-Point: Tools and Reproducibility Challenges

SIAM Conference on Parallel Processing for Scientific Computing (PP26)

March 5, 2026

Ignacio Laguna

Center for Applied Scientific Computing
LLNL

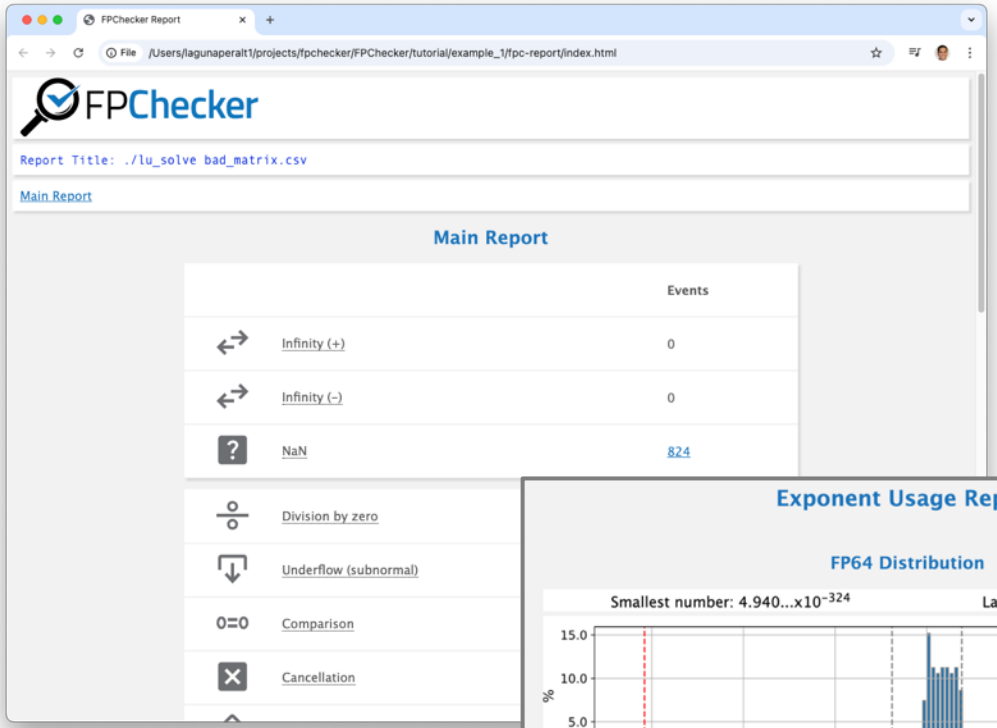
Prepared by LLNL under Contract DE-AC52-07NA27344.

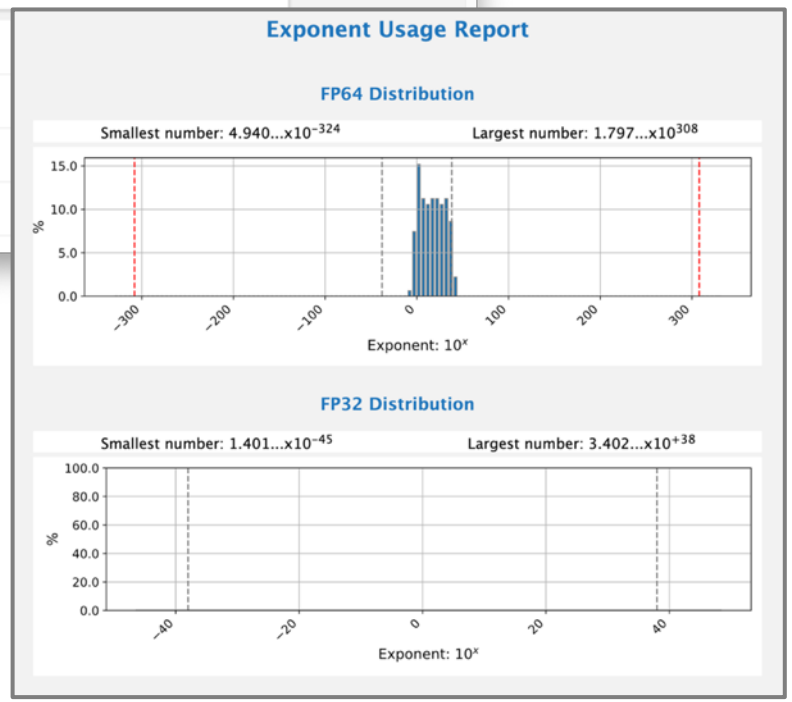
Need for Tools to Analyze Low-Precision Arithmetic

- Architectures promote lower precision formats:
  - FP16, FP8
  - Bfloat16
  - TensorFlow-32
- We need tools to understand mixed-precision codes
- Important behavior to understand:
  - ✓ Dynamic range of lower precision: FP32, FP16
  - ✓ Rounding error accumulation
  - ✓Matrix multiplication error and propagation

FPChecker

- Detects floating-point exceptions
  - `NaN, Infinity`
- Shows impacted *lines of code*
- Shows other "code smells"
  - `Cancellations, underflows`
- Analyzes dynamic range
  - Is FP32 or FP64 enough?
- Works with compiler instrumentation
  - Uses Clang/LLVM
- Documentation: https://fpchecker.org/

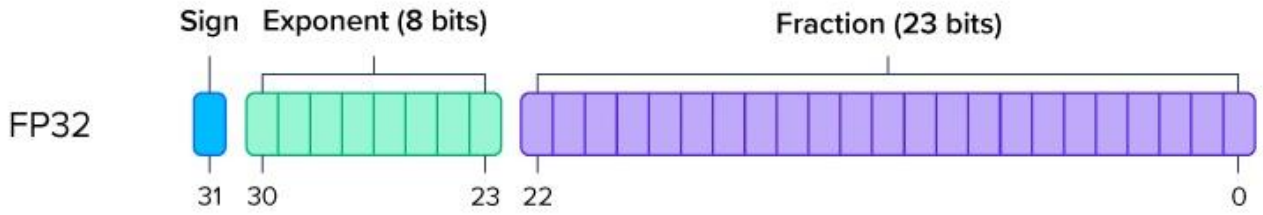




Laguna, Ignacio, et al. "FPChecker: Floating-point exception detection tool and benchmark for parallel and distributed HPC." *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2022.

Dynamic range analysis
from FP64 → FP32 → FP16 …

- Identifying **underflow** and **overflow** risks:
  - See if values fit within the format
  - If values ranges between 10^{-20} and 10^{20}, safely use FP32
  - FP32 covers roughly: 1.18×10^{-38} and 1.18×10^{38}



- **Targeted** precision reduction
  - Not every part of your code needs the same level of precision

Instruction-level LLVM Instrumentation to Extract Numerical Magnitudes

Program

```
void function(double *x, double *y, ..)
{
  for (...) {
    x[i] = y[i] + ...
  }
    double a = b * c;
    ...
}
```

LLVM Instructions

```
%x = fadd float %y, %var
%a = fmul float %b, %c
```

Runtime Calls

```
----  extract_exponent(%x)
----  extract_exponent(%a)
```

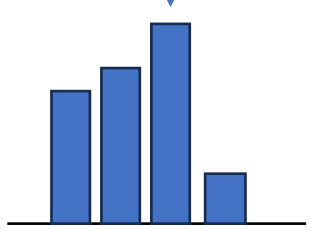
Fast method

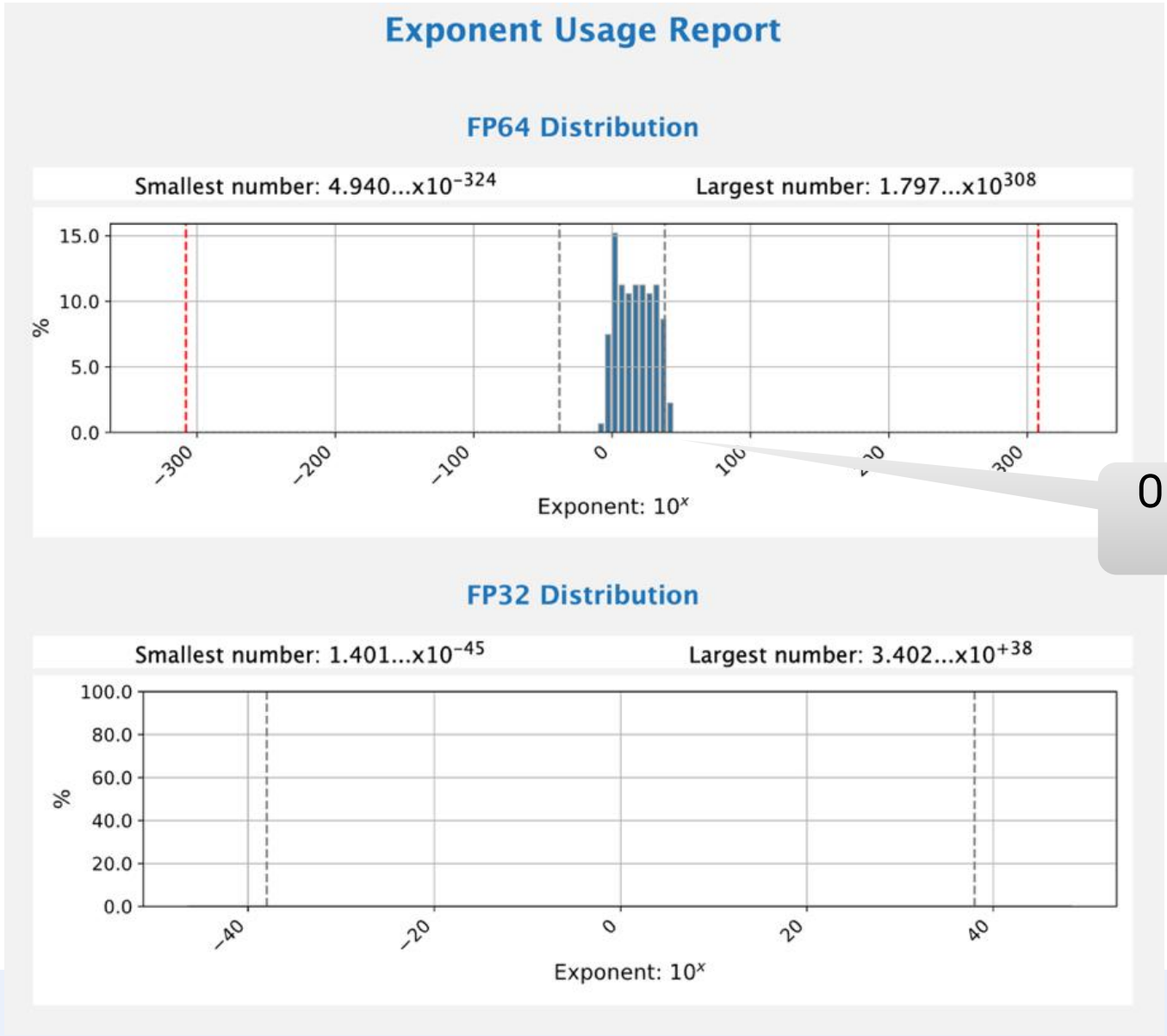
```
uint32_t _EXTRACT_EXPONENT(float x)
{
  uint32_t val;
  memcpy((void *)&val, (void *)&x, sizeof(val));
  val = val << 1;  // get rid of sign bit
  val = val >> 24; // get rid of the mantissa bits
  return val;
}
```

Exponent Usage (Dynamic Range)

- FPChecker builds a **histogram** of exponent values

$$variable = 1.23 \times 10^{200}$$





Out of range for FP32

Relative Rounding Error Analysis

(work in progress for next release…)

- Why is it important to understand *relative rounding error*?
  - **Error amplification**
    - **Example**: iterative solvers error can compound
  - **Catastrophic cancelation**
    - Number truncation in lower precision formats is more dangerous
  - **Stability**

```
>>> 0.1 + 0.1
0.2
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
```

- Compiler-level instrumentation
- Analyze whole-program relative error propagation

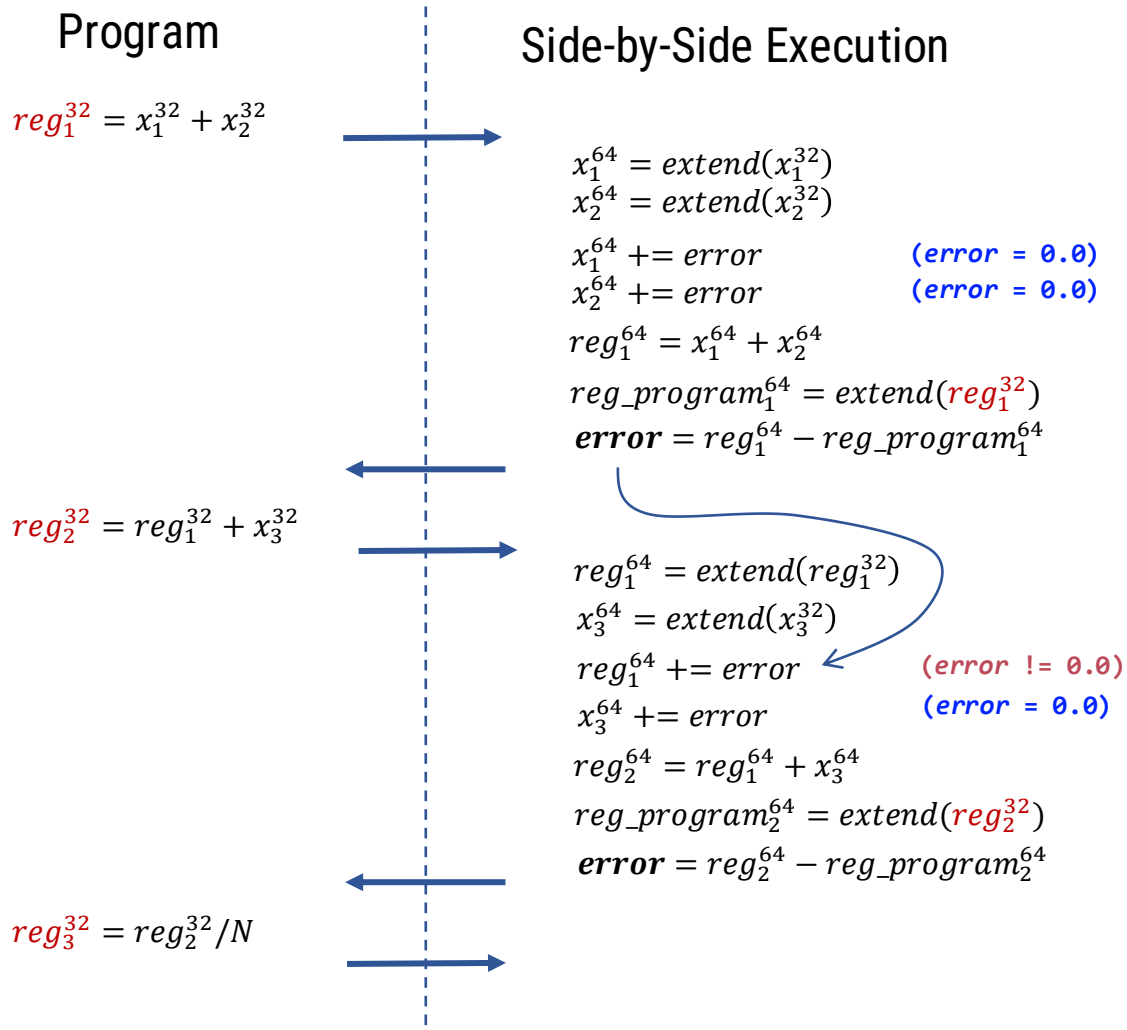
Side-by-Side Execution in Higher Precision

Example (average): $y = \frac{x_1 + x_2 + x_3}{N}$

```
1 void average(float *array, int N, float &average)
2 {
3     float temp = 0.0f;
4     for (int i = 0; i < N; ++i)
5     {
6         temp += array[i];
7     }
8     average = sum_f / N;
9 }
```

```
1 void average(float *array, int N, float &average)
2 {
3     float temp = 0.0f;                          0.000000
4     for (int i = 0; i < N; ++i)
5     {
6         temp += array[i];                       7.12845e-7
7     }
8     average = sum_f / N;                        7.12345e-9
9 }
```

Relative Rounding Error



FPChecker Report

File /Users/lagunaperalt1/projects/fpchecker/FPChecker/tutorial/example_5/fpc-report/index.html

FPChecker

Report Title: ./cg 100 1e-6

Events | Rounding Error

File:

		Rounding Error	Relative Erro
1	#include		
2	...		
37	...		
38	{		
39	guess = 0.5f * (guess + x / guess);	2.7208841e-01	9.9994[illegible]5e-01
40	}		
41	...		
50	...		
51	{		
52	result += v1[i] * v2[i];	1.8510156e-02	1.0000000e+00
53	}		
54	...		
73	...		
74	// A[i][j] is stored at index i * n + j		
75	result[i] += A[i * n + j] * v[j];	1.2704145e-02	1.0000000e+00
76	}		
77	...		
88	...		
89	{		
90	result[i] = v1[i] - alpha * v2[i];	-8.8529801e-05	9.9999998e-01
91	}		
92	...		

Line: 39
Relative error: 9.99e-01

Mixed Precision Use Case

```
int main() {

    // --- REGION 1: Well-Behaved (Keep in FP32)
    float region1_sum = 0.0f;
    for (int i = 0; i < iterations; ++i) {
        region1_sum += small_val;
    }

    // --- REGION 2: Numerically Unstable (Upgrade to FP64)
    float region2_sum_fp32 = large_val;
    for (int i = 0; i < iterations; ++i) {
        region2_sum_fp32 += small_val;
    }

    double region2_sum_fp64 = (double)large_val;
    for (int i = 0; i < iterations; ++i) {
        region2_sum_fp64 += (double)small_val;
    }

    // Output results for the presentation

    return 0;
}
```

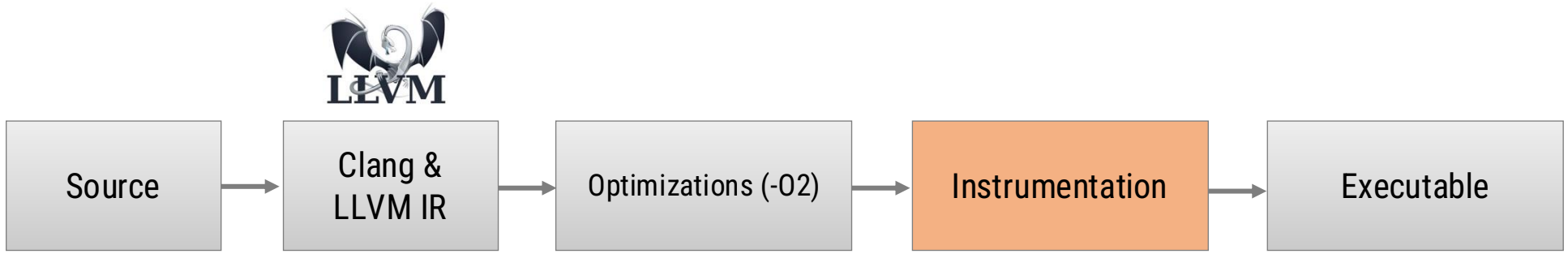
Region 1

- Relative error is low
- Close to 10^{-7} in float precision
- Keep the computation in FP32
- Well-behaved calculation that is not losing significant digits.

Region 2

- Relative error is high
- Close to 10^{-5} in float precision
- Upgrade to FP64 precision
- Numerical instability that float cannot handle.

Compiler (LLVM) Instrumentation Process



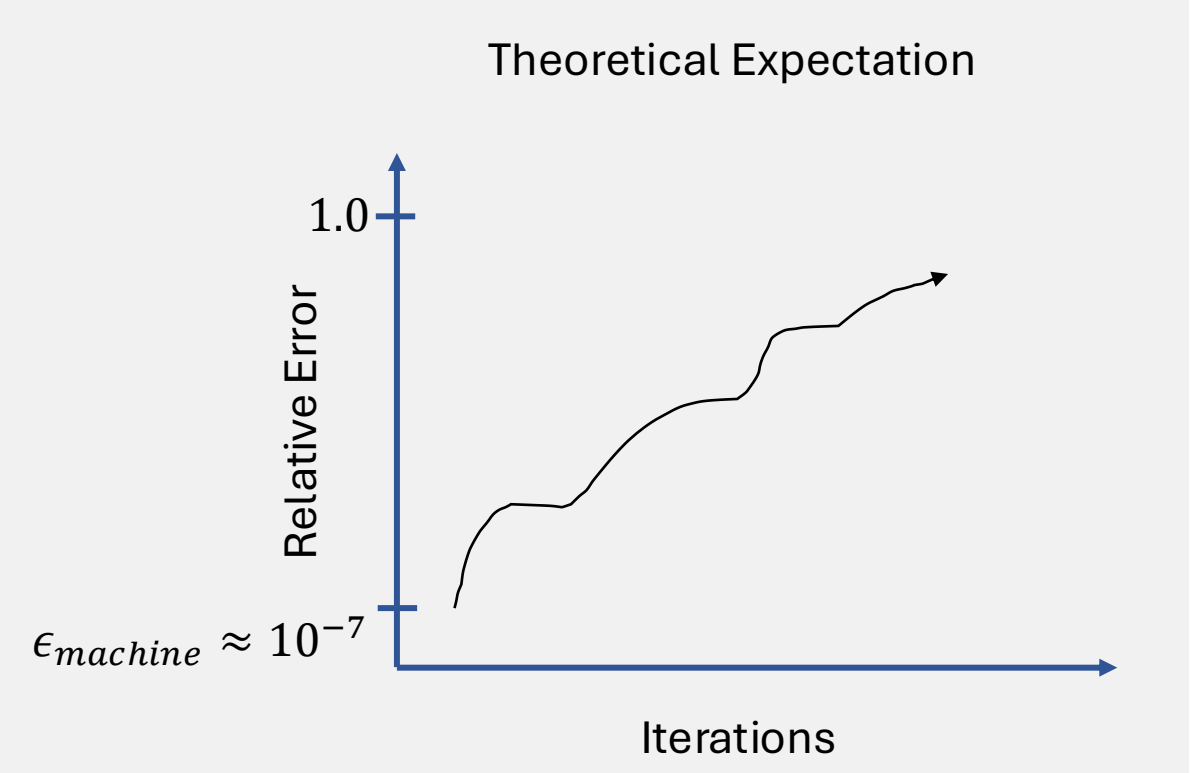
1. Use `clang++-fpchecker` wrapper in your Makefile
2. Compile the code
3. Run executable
4. Create report

Case Study (Example 1): *Catastrophic Cancellation in CG*

- Solution x_k approaches the true solution x
- The two vectors in the subtraction become increasingly nearly equal in magnitude
- **Catastrophic cancellation** may occur

$$r_{k+1} = r_k - \alpha(Ap_k)$$

Become nearly equal



We use the tool to demonstrate and identify this behavior in CG (line of code)

Review of Catastrophic Cancellation

$x - y$

x: 0.123456
y: 0.123444

True Difference: 0.000012

Floating-Point Rounding

x: 0.1235
y: 0.1234

Computed Difference: 0.0001

$$Relative\ Error = \frac{|Computed\ - True|}{|True|}$$

$$= \frac{|0.0001\ - 0.000012|}{0.000012} \approx 7.33$$

- Error introduced is **not bounded** by machine epsilon
- Can occur in different parts of the code
- Tricky to diagnose without proper tools

Catastrophic Cancellation Leads and Loss of Orthogonality in CG

Residual update in CG

$$r_{k+1} = r_k - \alpha(Ap_k)$$

In exact math, this must hold:

$$r_{k+1}^T r_k = 0$$

- Must be true due to Krylov space
- The entire set of residuals are mutually orthogonal

Let's assume for a given iteration k:

$$r_k = \begin{matrix} r_1 \\ r_2 \\ \vdots \end{matrix} \qquad \alpha(Ap_k) = C = \begin{matrix} c_1 \\ c_2 \\ \vdots \end{matrix}$$

$$r_{k+1}^T r_k = [r_1 - c_1, r_2 - c_2, \cdots] \cdot \begin{matrix} r_1 \\ r_2 \\ \vdots \end{matrix}$$

$$= (r_1 - c_1)r_1 + (r_2 - c_2)r_2 + \cdots = 0$$

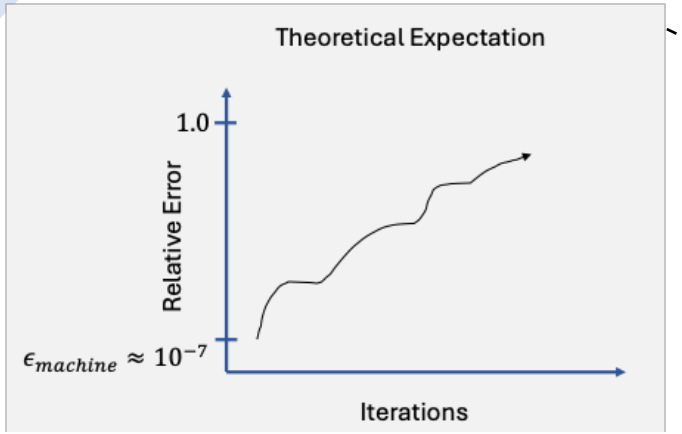
With cancellation, we introduce an error *e*

$$r_{k+1}^T r_k = [r_1 - c_1 + e_1, r_2 - c_2 + e_2, \cdots] \cdot \begin{matrix} r_1 \\ r_2 \\ \vdots \end{matrix}$$

$$= (r_1 - c_1 + e_1)r_1 + (r_2 - c_2 + e_2)r_2 + \cdots = 0$$

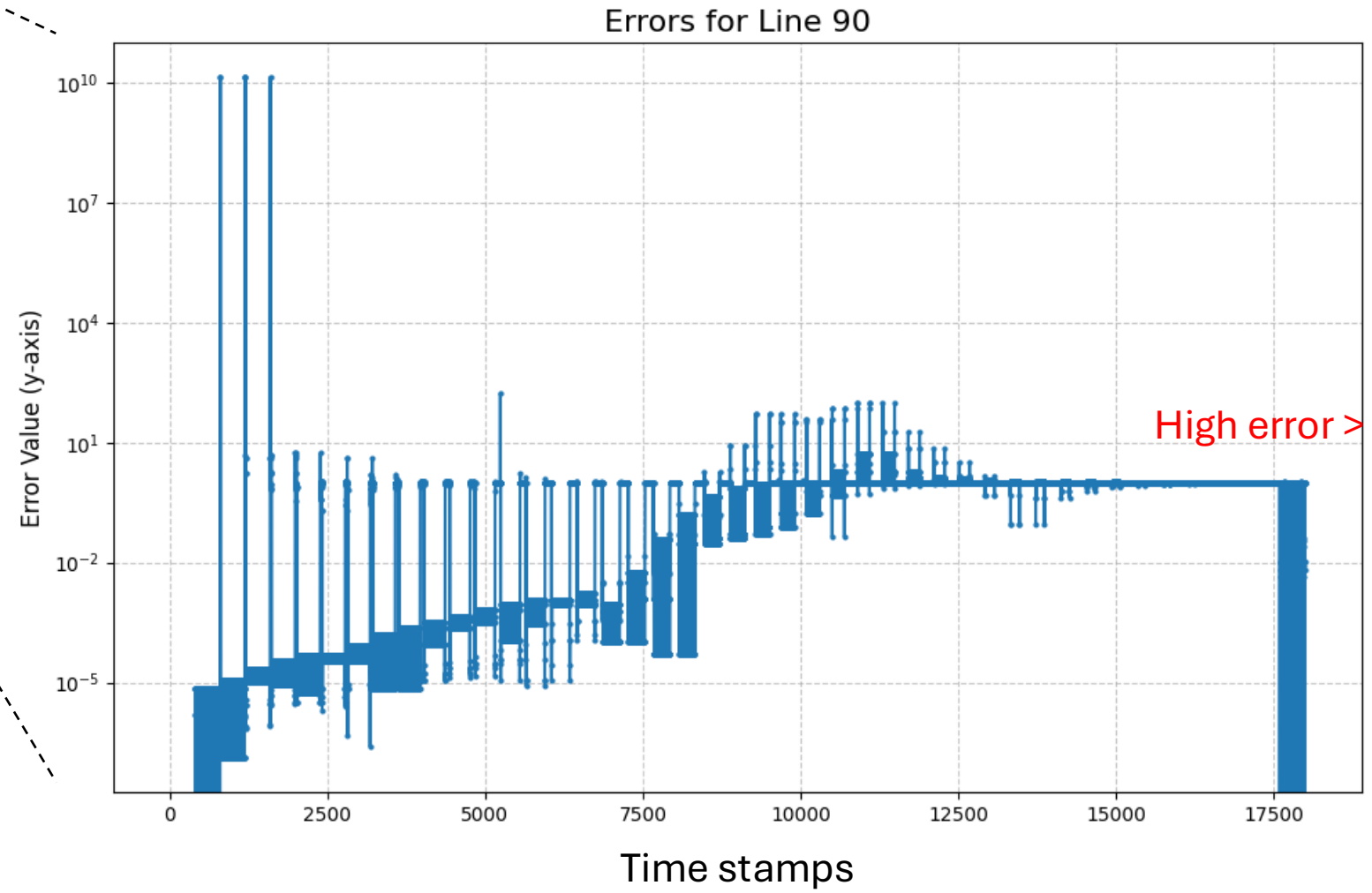
Large error, not bounded by machine epsilon ϵ

Example of the Tool's Output for CG

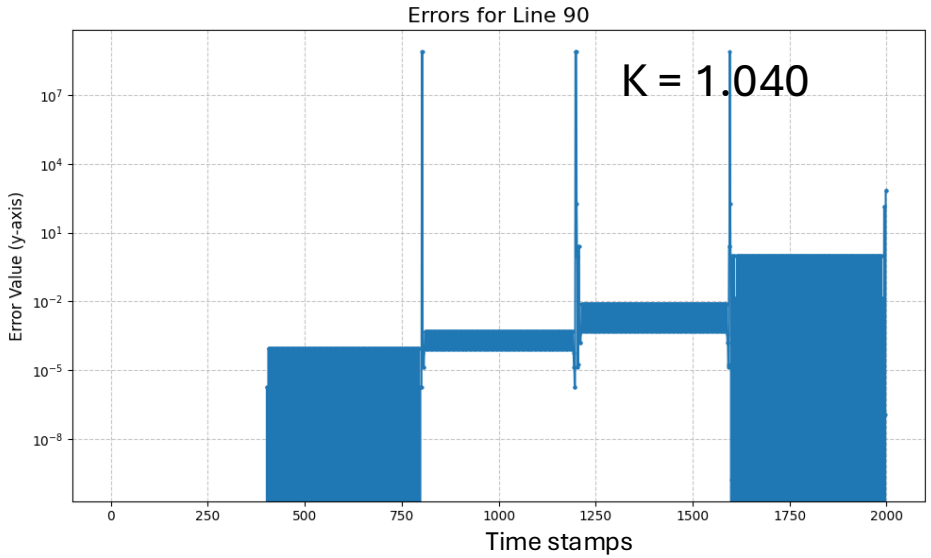


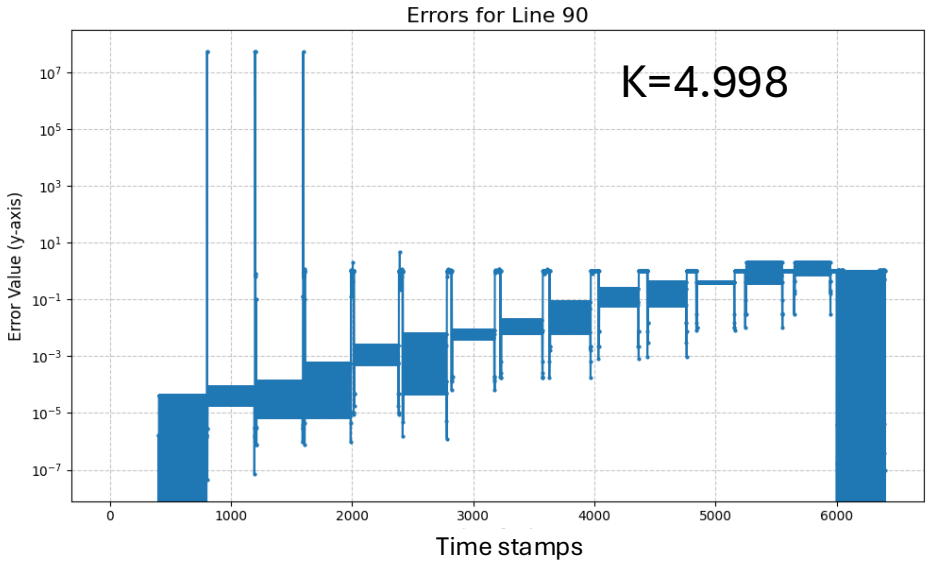
Line 90:

$$r_{k+1} = r_k - \alpha(Ap_k)$$

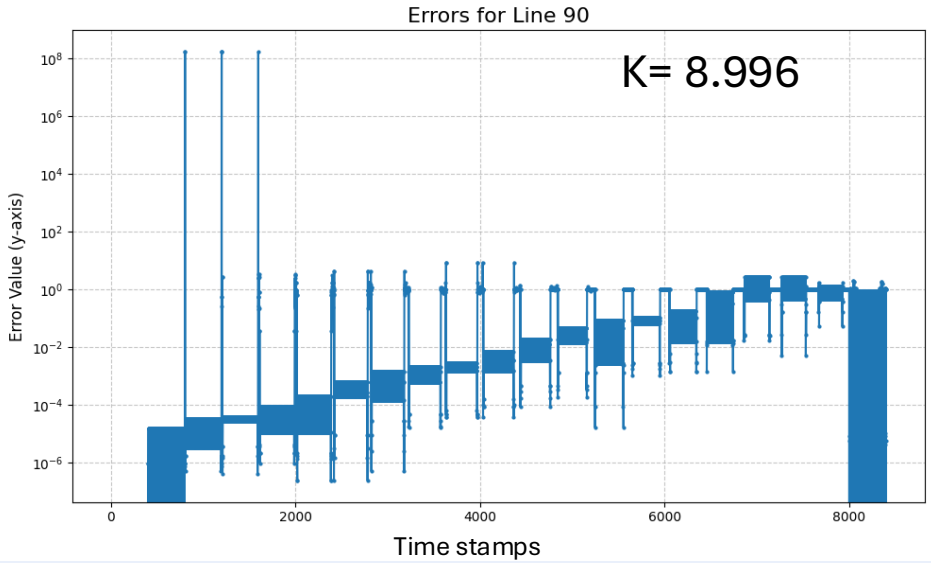


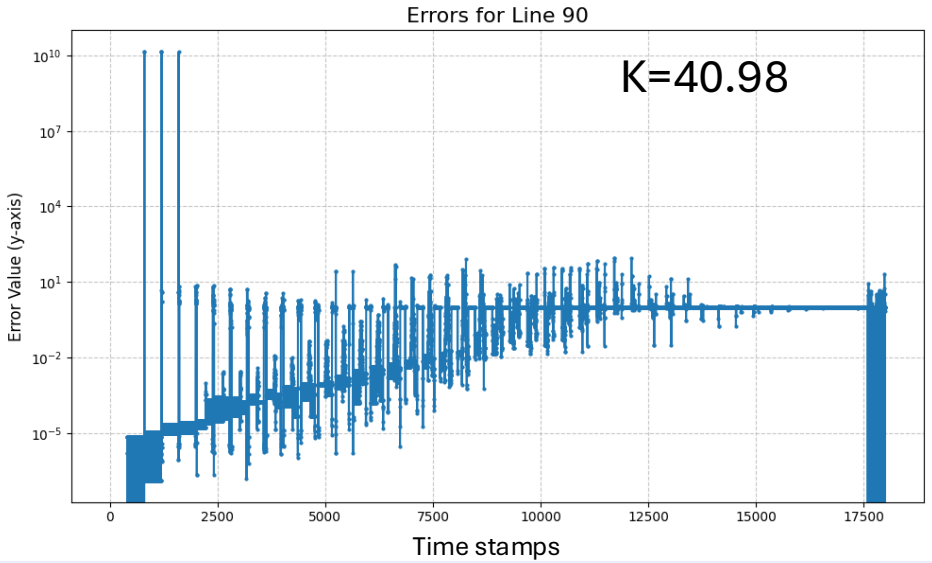
Similar Results as We Vary the Condition Number





K= condition number



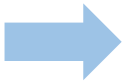


Example 2:
Created Mixed-Precision Based on Tool's Report

Toy example:
- Small matrix 1000x1000
- Serial CG

Regions of code with high relative error (e.g., 1.0)

- Matrix-vector multiply: $A\ p_k$
- Vector Add-Multiply: $x_{k+1} = x_k + \alpha_k p_k$
- Dot products
- Division: $\alpha_k = \frac{r_k^T r_k}{p_k^T A p_k}$



- Converted them into FP64
- Converge in checked in FP64

```
if (relative_residual < tolerance) {
    ...
}
```

- Moved Matrix-vector multiply to FP32 to avoid many casting operations

Source: https://github.com/llnl/FPChecker/blob/v0.6/tutorial/example_5/cg_mixed.cpp

Example 2's Performance Results

Setting:
Max Iterations: 5000
Tolerance: 1e-06

FP32

```
Converged in 791 iterations.
Average time per iteration: 9.521683e-04 sec.
Execution time: 7.551423e-01 sec.
Final Residual Norm (||Ax - b||): 2.759115e-01
```

Mixed

```
Converged in 500 iterations.
Average time per iteration: 9.491665e-04 sec.
Execution time: 4.781714e-01 sec.
Final Residual Norm (||Ax - b||): 1.156815e-02
```

FP64

```
Converged in 500 iterations.
Average time per iteration: 1.031628e-03 sec.
Execution time: 5.179753e-01 sec.
Final Residual Norm (||Ax - b||): 1.907349e-06
```

Summary

- Tools to help applications transition into lower-precision
  - FP32 and other numerical formats
  - Dynamic range analysis to mitigate underflow/overflow risks
  - Accumulated relative error analysis
- Tools are easy to use with the clang/LLVM compiler
- FPChecker:
  - Version 0.5: dynamic range analysis
  - Version 0.6: relative error analysis (to be released in Jun-Jul/26)

Contact:
Ignacio Laguna
Email: ilaguna@llnl.gov