Compensated Algorithms and Error-Free Transformations

Floating-point arithmetic and error analysis (AFAE)

Sorbonne Université

Contents

1 Introduction to Backward Error Analysis

The fundamental approach to understanding numerical errors uses **backward error analysis**:

- **Forward error**: Measures the difference between computed result and exact result
- **Backward error**: Identifies the computed result as the exact solution to a perturbed problem
- **Condition number**: Measures sensitivity of the solution to perturbations in the data

The key relationship is:

$$\text{forward error} \lesssim \text{condition number} \times \text{backward error} \tag{1}$$

When the backward error is approximately u (machine precision), the algorithm is called **backward stable**.

1.1 Standard Model of Floating-Point Arithmetic

For basic operations, we have:

$$a \circ b = \text{fl}(a \circ b) = (a \circ b)(1 + \delta) \tag{2}$$

where $|\delta| \leq u$ and $u = 2^{-p}$ is the unit roundoff.

2 Fundamental Theorems and Results

2.1 Lemma 1: Error Accumulation

Lemma 1 (Error Accumulation)

If $|\delta_i| \leq u$ for $i = 1 : n$ and $nu < 1$, then:

$$\prod_{i=1}^{n} (1 + \delta_i)^{\rho_i} = 1 + \delta_n \tag{3}$$

where $|\delta_n| \leq \gamma_n = \frac{nu}{1-nu}$.

This lemma is fundamental for analyzing error accumulation in sequences of floating-point operations.

2.2 Convergence of Newton's Method

Under appropriate assumptions about the derivative accuracy and assuming $u \cdot \text{cond}_{\text{root}}(p, x) \leq 1/8$, Newton's method converges until reaching the accuracy limit determined by the condition number and the quality of function/derivative evaluation.

3 Practical Considerations

3.1 When to Use Compensated Algorithms

Compensated algorithms are beneficial when:

- The condition number of the problem is large (near $1/u$ or larger)
- High accuracy is required without switching to higher precision
- Performance is acceptable (typically 2–4× slower than standard algorithms)

3.2 Computational Costs

Summary of operation counts:

Algorithm	Cost (flops)
TwoSum	6
FastTwoSum	3
TwoProduct (without FMA)	17
TwoProduct (with FMA)	2
CompSum (for n terms)	$\approx 13n$
CompHorner (degree n)	$\approx 13n$

Table 1: Operation counts for compensated algorithms

Key Result

The compensated algorithms achieve accuracy improvements of roughly $1/u$ (going from γ_n to $u+\gamma_n^2$) while only requiring 3–4 times more operations than standard algorithms, making them much more efficient than switching to double-double or quad precision arithmetic.

4 Error-Free Transformations (EFT)

EFTs are fundamental building blocks that allow us to compute the exact rounding error from floating-point operations.

4.1 TwoSum Algorithm

Computes $x + y = s + e$ where both s and e are floating-point numbers:

Algorithm 1 TwoSum

1: **function** TwoSum(a, b)
2: $x \leftarrow a \oplus b$
3: $z \leftarrow x \ominus a$
4: $y \leftarrow (a \ominus (x \ominus z)) \oplus (b \ominus z)$
5: **return** $[x, y]$
6: **end function**

Properties:

- $a + b = x + y$ exactly
- $|y| \leq u|x|$ and $|y| \leq u|a + b|$
- Cost: 6 floating-point operations

4.2 FastTwoSum Algorithm

When $|a| \geq |b|$, only 3 operations are needed.

Algorithm 2 FastTwoSum

1: $s \leftarrow \text{fl}(a + b)$
2: $z \leftarrow \text{fl}(s - a)$
3: $t \leftarrow \text{fl}(b - z)$
4: **return** $[s, t]$

With round-to-nearest: $s + t = a + b$ exactly, where t captures the rounding error.

4.2.1 FastTwoSum with Directed Rounding

Directed Rounding Issues

FastTwoSum requires **round-to-nearest** mode. The algorithm fails with directed rounding modes.

Counterexample with Rounding toward $+\infty$

Values:

- $a = 1$
- $b = 2^{-53}$
- Check: $|a| \geq |b|$ ✓

Step 1: $s \leftarrow \text{fl}_\uparrow(1 + 2^{-53})$

- Exact value: $1 + 2^{-53}$ (not representable)
- Rounded up to: $s = 1 + 2^{-52}$ (next representable number)

Step 2: $z \leftarrow \text{fl}_\uparrow(s - a) = \text{fl}_\uparrow(2^{-52}) = 2^{-52}$
Step 3: $t \leftarrow \text{fl}_\uparrow(b - z) = \text{fl}_\uparrow(2^{-53} - 2^{-52}) = -2^{-53}$
Verification:

$$s + t = (1 + 2^{-52}) + (-2^{-53}) = 1 + 2^{-53} = a + b \quad \checkmark \tag{4}$$

The **true rounding error** from step 1 should be:

$$\text{error}_{\text{true}} = s - (a + b) = (1 + 2^{-52}) - (1 + 2^{-53}) = 2^{-53} \tag{5}$$

But FastTwoSum computes $t = -2^{-53}$, which has the **wrong sign**.
While $s + t = a + b$ still holds, t does **not** represent the actual rounding error of the addition. The algorithm captures the difference needed to reconstruct $a + b$, but not the rounding error itself.

Analysis:

- **With round-to-nearest**:
  - Rounding errors are symmetric ($\pm u/2$)
  - Error cancellations work correctly
  - t represents the true rounding error
- **With rounding toward $+\infty$**:

- All operations systematically round upward
- Creates directional bias that compounds across operations
- The subtraction steps (2–3) cannot correctly recover the rounding error
- t becomes a correction term but not the actual error

Note

FastTwoSum requires **round-to-nearest** mode. Directed rounding breaks the error-free transformation property because systematic rounding bias prevents exact error recovery. Different algorithms are needed for directed rounding modes.

4.3 TwoProduct Algorithm

Computes $a \times b = x + y$ exactly.

4.3.1 Without FMA (17 operations)

Algorithm 3 TwoProduct (without FMA)

function TwoProduct(a, b)
  $x \leftarrow a \otimes b$
  $[a_1, a_2] \leftarrow$ Split(a)
  $[b_1, b_2] \leftarrow$ Split(b)
  $y \leftarrow a_2 \otimes b_2 \ominus (((x \ominus a_1 \otimes b_1) \ominus a_2 \otimes b_1) \ominus a_1 \otimes b_2)$
  return $[x, y]$
end function

4.3.2 With FMA (2 operations)

Algorithm 4 TwoProduct (with FMA)

function TwoProduct(a, b)
  $x \leftarrow a \otimes b$
  $y \leftarrow \text{FMA}(a, b, -x)$
  return $[x, y]$
end function

Performance

With FMA (Fused Multiply-Add), TwoProduct requires only 2 operations instead of 17, making compensated algorithms dramatically more efficient on modern processors.

5 Summation Algorithms

5.1 Classic Recursive Sum

The standard algorithm has error bound:

$$|\text{res} - s| \leq \gamma_{n-1} \sum_{i=1}^{n} |p_i| \tag{6}$$

where $\gamma_n = \frac{nu}{1-nu}$.

5.2 Kahan's Compensated Summation

Improves accuracy significantly with error bound:

$$|\text{res} - s| \leq (2u + O(nu^2)) \sum_{i=1}^{n} |p_i| \tag{7}$$

The algorithm maintains a running compensation term that captures lost precision.

5.3 Compensated Summation (Ogita-Rump-Oishi)

Achieves near-optimal accuracy with error bound:

$$|\text{res} - s| \leq u|s| + \gamma_{n-1}^2 S \tag{8}$$

where $S = \sum |p_i|$.

Algorithm 5 CompSum (Compensated Summation)

```
function CompSum(p)
    π_1 ← p_1; σ_1 ← 0
    for i = 2 to n do
        [π_i, q_i] ← TwoSum(π_{i−1}, p_i)
        σ_i ← σ_{i−1} ⊕ q_i
    end for
    res ← π_n ⊕ σ_n
    return res
end function
```

The algorithm uses EFT to capture all rounding errors, then sums them separately.

Accuracy Improvement

CompSum reduces the error from $O(nu)$ to $O(u + n^2u^2)$, achieving near-optimal accuracy. For typical values with $n \approx 10^6$ and $u \approx 10^{-16}$:

- Classic sum: error $\approx 10^{-10}$
- CompSum: error $\approx 10^{-16}$

6 Polynomial Evaluation

6.1 Horner's Scheme

The classic method with error bound:

$$\frac{|p(x) - \text{Horner}(p, x)|}{|p(x)|} \leq \gamma_{2n} \cdot \text{cond}(p, x) \tag{9}$$

where the condition number is:

$$\text{cond}(p, x) = \frac{\sum_{i=0}^{n} |a_i||x|^i}{|\sum_{i=0}^{n} a_i x^i|} = \frac{\tilde{p}(|x|)}{|p(x)|} \tag{10}$$

6.2 Compensated Horner Scheme (CompHorner)

Achieves much better accuracy with error bound:

$$\frac{|\text{CompHorner}(p,x) - p(x)|}{|p(x)|} \leq u + \gamma_{2n}^2 \cdot \text{cond}(p,x) \tag{11}$$

Algorithm 6 CompHorner (Compensated Horner)

```
function CompHorner(p, x)
    [h, p_π, p_σ] ← EFTHorner(p, x)
    c ← Horner(p_π ⊕ p_σ, x)
    res ← h ⊕ c
    return res
end function
```

The algorithm uses TwoProduct and TwoSum to capture rounding errors during Horner evaluation, then evaluates the error polynomial.

CompHorner Performance

About 3 times slower than standard Horner, but 3 times faster than double-double arithmetic while achieving similar accuracy.

7 Root Finding with Newton's Method

7.1 Condition Number for Root Finding

For a simple root x of polynomial p:

$$\text{cond}_{\text{root}}(p,x) = \frac{\tilde{p}(|x|)}{|x||p'(x)|} \tag{12}$$

7.2 Classic Newton's Method

Error bound:

$$\frac{|x_{i+1} - x|}{|x|} \approx \gamma_{2n} \cdot \text{cond}_{\text{root}}(p,x) \tag{13}$$

7.3 Accurate Newton's Method

Using CompHorner for function evaluation:

$$\frac{|x_{i+1} - x|}{|x|} \approx u + \gamma_{2n}^2 \cdot \text{cond}_{\text{root}}(p,x) \tag{14}$$

7.4 New Accurate Newton's Method

Using both CompHorner and CompHD (compensated Horner derivative):

$$\frac{|x_{i+1} - x|}{|x|} \approx Ku + D\gamma_{2n}^2 \cdot \text{cond}_{\text{root}}(p,x) \tag{15}$$

where K and D are small constants.

Best Possible Accuracy

This achieves the best possible accuracy, with both the residual and derivative computed accurately. The method maintains convergence even for extremely ill-conditioned root-finding problems (condition numbers up to 10^{30}).

8 Summary Tables

8.1 Error Bounds Comparison

Algorithm	Standard Error	Compensated Error
Summation	$\gamma_{n-1}\sum \lvert p_i\rvert$	$u\lvert s\rvert + \gamma_{n-1}^2 S$
Dot Product	$\gamma_n \lvert x\rvert_1 \lvert y\rvert_1$	$u\lvert x\cdot y\rvert + \gamma_n^2 \lvert x\rvert_1 \lvert y\rvert_1$
Horner	$\gamma_{2n}\,\mathrm{cond}(p,x)\lvert p(x)\rvert$	$(u+\gamma_{2n}^2\,\mathrm{cond}(p,x))\lvert p(x)\rvert$
Newton (root)	$\gamma_{2n}\,\mathrm{cond}_{\mathrm{root}}(p,x)$	$u+\gamma_{2n}^2\,\mathrm{cond}_{\mathrm{root}}(p,x)$

Table 2: Comparison of error bounds for standard vs compensated algorithms

8.2 Accuracy Gain

The compensated algorithms typically improve accuracy by a factor of:

$$\text{Improvement factor} \approx \frac{\gamma_n}{u} = \frac{nu}{(1-nu)u} \approx n \tag{16}$$

For $n = 1000$ and $u = 2^{-53}$:

- Standard error: $\approx 1000u \approx 10^{-13}$
- Compensated error: $\approx u \approx 10^{-16}$
- Improvement: $\approx 1000\times$

8.3 Cost vs Accuracy Trade-off

Approach	Cost	Error	Efficiency
Standard (binary64)	$1\times$	nu	Baseline
Compensated	3–$4\times$	$u + n^2u^2$	High
Double-double	10–$20\times$	u_{dd}	Medium
Quad precision	50–$100\times$	u_{quad}	Low

Table 3: Performance and accuracy comparison of different approaches

Practical Recommendation

Compensated algorithms offer the best cost-accuracy trade-off when:

- Condition number is 10^3 to 10^{10}
- Standard precision is insufficient
- Higher precision arithmetic is too expensive

9 Key Takeaways

1. **Error-Free Transformations** enable exact capture of rounding errors using only standard floating-point arithmetic.
2. **Compensated algorithms** achieve near-optimal accuracy by:
   - Using EFT to compute exact rounding errors
   - Accumulating and correcting for these errors
   - Maintaining the error term in a separate accumulator
3. **Cost is reasonable**: 3–4× slowdown for $1/u$ accuracy improvement.
4. **Hardware support** (FMA instruction) reduces cost dramatically for TwoProduct.
5. **Applications**: Essential for ill-conditioned problems where standard algorithms fail but full multi-precision arithmetic is too expensive.
6. **Limitation**: FastTwoSum requires round-to-nearest mode; directed rounding breaks error-free properties.
7. **Theory**: The γ_n term appears consistently in error analysis and characterizes error growth in recursive algorithms.

Slides summarized with Claude by Giulia Lionetti- Sorbonne Université