# FPU-Supported Running Error Analysis

T. Zahradnický, R. Lórencz

**Abstract**

A-posteriori forward rounding error analyses tend to give sharper error estimates than a-priori ones, as they use actual data quantities. One of such a-posteriori analysis – running error analysis – uses expressions consisting of two parts; one generates the error and the other propagates input errors to the output. This paper suggests replacing the error generating term with an FPU-extracted rounding error estimate, which produces a sharper error bound.

**Keywords:** Analysis of algorithms, a-posteriori error estimates, running error analysis, floating point, numerical stability.

## 1 Introduction

Rounding error analyses are used to discover the stability of numerical algorithms and provide information on whether the computed result is valid. They are typically performed in forward or backward direction, and either a-priori or a-posteriori. Backward error analysis [6] treats all operations as if they were exact but with a perturbed data, and we ask for which input data we have solved our problem. Using backward error analysis is preferred, as each algorithm which is backward stable is automatically numerically stable [5], while this does not hold for forward error analyses. On the other hand, forward error analyses treat operations as sources of errors, and tell us about the size of the solution that our input data corresponds to. Since error analyses performed a-priori can often be difficult if the algorithm being analyzed is complex, a-posteriori analyses may provide a more feasible alternative (MSC 2000 Classification: 65G20, 65G50).

An objective of an a-priori analysis is to find the worst case bound for an algorithm, if it exists, while a-posteriori analyses calculate the error bound concurrently with the evaluation of the result and work with actual data quantities, providing a tighter error bound.

This paper presents a replacement for an error generating term in error bounds calculated with forward a-posteriori error estimates (also known as running error analysis) to obtain sharper error bounds by exploiting the behavior of the computer FPU. We expect the calculation to be portable (conforming strictly to [3]) and assume an Intel x86 compatible FPU which supports double extended precision (80-bit) floating point registers, which are essential for this method.

## 2 Running error analysis and error bounds

Running error analysis [6, 7] is a type of a-posteriori forward error analysis. The algorithm being analyzed is extended to calculate the partial error bound alongside the normal calculation. As the algorithm proceeds, bounds are accumulated, making a total error bound estimate. From here on, a binary double precision floating point arithmetic with a guard digit with round to nearest rounding mode is assumed.

**Definition 1** *Let $\mathcal{F}_t \subset \mathcal{Q}$ be a binary floating point set with a precision of $t$, where $\mathcal{Q}$ denotes the rational number set.*

**Definition 2** *Let $y = \pm m\ 2^{e-t} \in \mathcal{F}_t$ be a binary floating point number, where $m$ stands for mantissa, $e$ for exponent, and $t$ for precision. Let $\mathrm{ex}(y) = e$.*

**Lemma 3** *Let $\widehat{x} = \mathrm{fl}(x)$ be a floating point representation of an exact number $x$, which is obtained by rounding $x$ to the nearest element in $\mathcal{F}_t$. The rounding process can be described as:*

$$\widehat{x} = \mathrm{fl}(x) = x(1+\delta), \qquad |\delta| \le u_t, \tag{1}$$

*where $u_t = 2^{-t}$ is unit roundoff in precision $t$. To conserve some space, and for clarity, $u$ without a subscript means $u_{53}$.*

**Assumption 4** *(Standard model) We assume that operations $\diamondsuit \in \{+, -, \cdot, /\}$ and the square root follow the Standard model [5] and are evaluated with error no greater than $u$:*

$$\begin{aligned} \widehat{s}_{\diamondsuit} &= \mathrm{fl}(\widehat{a} \diamondsuit \widehat{b}) = (\widehat{a} \diamondsuit \widehat{b})/(1+\delta_{\diamondsuit}), & |\delta_{\diamondsuit}| \le u, \\ \widehat{s}_{\sqrt{\ }} &= \mathrm{fl}(\sqrt{\widehat{a}}) \ = \sqrt{\widehat{a}}/(1+\delta_{\sqrt{\ }}), & |\delta_{\sqrt{\ }}| \le u, \end{aligned} \tag{2}$$

*where we use $\widehat{s}$ as a computed result of an operation. The Standard model will also be used in the form of $\widehat{s}_{\diamondsuit} = \mathrm{fl}(\widehat{a} \diamondsuit \widehat{b}) = (\widehat{a} \diamondsuit \widehat{b}) \cdot (1+\delta_{\diamondsuit})$, where useful.*

Table 1 summarizes the error bounds for basic operations and the square root with products of errors being neglected. We also assume that the following holds: $\widehat{s} = s/(1+\delta_s) = s + \sigma$, $\widehat{a} = a/(1+\delta_a) = a + \alpha$, and $\widehat{b} = b/(1+\delta_b) = b + \beta$, where $\alpha$, $\beta$, and $\sigma$ stand for absolute (static) errors.

Table 1: Error bound estimates for basic operations

| Operation | Error bound |
|---|---|
| Addition and subtraction | $\lvert\sigma_{\pm}\rvert \leq u\,\lvert\widehat{a} \pm \widehat{b}\rvert + \alpha + \beta$ |
| Multiplication | $\lvert\sigma_{.}\rvert \leq u\,\lvert\widehat{a}\widehat{b}\rvert + \alpha\lvert\widehat{b}\rvert + \beta\lvert\widehat{a}\rvert$ |
| Division | $\lvert\sigma_{/}\rvert \leq u\left\lvert\frac{\widehat{a}}{\widehat{b}}\right\rvert + \frac{\alpha\lvert\widehat{b}\rvert + \beta\lvert\widehat{a}\rvert}{\widehat{b}^2}$ |
| Square root | $\lvert\sigma_{\surd}\rvert \leq u\left\lvert\sqrt{\widehat{a}}\right\rvert + \frac{\alpha}{2\left\lvert\sqrt{\widehat{a}}\right\rvert}$ |

The right-hand side of each error bound in Tab. 1 contains an error generating term $u|\cdot|$, while the rest of each expression simply propagates input error(s) to the output. When we substitute $u = 2^{-53}$ into $u|\widehat{s}|$, multiplying by $u$ reduces the exponent by 53

$$u|\widehat{s}| = 1,\underbrace{xx\ldots xx}_{52\text{ times}}\cdot 2^{\mathrm{ex}(\widehat{s})-2t}, \qquad (3)$$

leaving the mantissa unchanged if $u|\widehat{s}|$ is normal. The roundoff unit is just the first 1 in (3), while $x$-en are generally nonzero. Using $u|\widehat{s}|$ as a rounding error estimate features two problems:

1. The calculated error bound $u|\widehat{s}|$ can be up to two times larger $\left(u\sum_{i=0}^{t-1}\widehat{s}_i 2^{-i} \lesssim 2u\right)$, assuming that $\widehat{s}_i$ refers to the $i$-th bit of $\widehat{s}$.
2. The error generation term always generates an error even in the case when no physical error was committed.

The expression $u|\widehat{s}|$ tends to give a higher error bound estimate than we would need. However, we can revise this expression if we are interested in obtaining a tighter error bound estimate, and we can do so by exploiting the FPU.

# 3 Analysis

Many computers use a processor with Intel x86 architecture [4], which features an FPU with 8 double extended precision floating point registers. Once a number is loaded into the FPU, regardless of its precision, it gets automatically converted into the double extended precision format. Further calculations are performed in this format but rounded to a precision specified in the floating point control word register (`FCW`). The default settings for `FCW` specify double extended precision, round to the nearest rounding mode and mask out all floating point exceptions. If necessary, they can be changed with an `fstcw` instruction. Once the result gets stored back into a memory location, it is rounded to (or extended if `FCW` specified single precision) a defined precision, depending on the type of store instruction.

[1] Rounding to even as in [3] applies here.

## 3.1 Obtaining a tighter bound from the FPU

When rounding to the nearest element of $\mathcal{F}_t$, the relative error is no worse than $u_t$, and we can extract the static error from the FPU by subtracting the computed value in double extended precision from its rounded value. This idea looks similar to compensated summation [5], which uses an entirely software approach in contrast to our paper. Subtracting the values provides an 11-bit error estimate of the real rounding error, which is no greater than $u$. The entire idea can be written as:

**Theorem 5** *Let $\widehat{S}$ be the result of an operation performed in double extended precision and let $\widehat{s}$ be the result in double precision, which we obtained by rounding $\widehat{S}$ to 53 bits by rounding to the nearest as defined by the IEEE 754 Standard. The absolute rounding error for an operation in floating point can be calculated rather as $\Delta = |\widehat{s} - \widehat{S}|$, where $|\Delta| \leq u|\widehat{s}|$.*

**PROOF.** We can extract 64-bit mantissa intermediate results ($\widehat{S}$) from the FPU before they get rounded to 53 bits ($\widehat{s}$), and calculate $\Delta = |\widehat{s} - \widehat{S}|$. The quantity $\widehat{S}$ is rounded either down to $f_1 \in \mathcal{F}_{53}$ or up to $f_2 \in \mathcal{F}_{53}$, where $f_1,\, f_2 = \{f_1,\, f_2 \in \mathcal{F}_{53} \,:\, |f_2 - f_1|/|f_1| = 2u\}$, $\widehat{S} \in \mathcal{F}_{64} \wedge \widehat{S} \in [f_1,\, f_2]$, $\widehat{s} = \underset{64\to 53}{\mathrm{round}}(\widehat{S}) \Rightarrow \widehat{s} \in \{f_1,\, f_2\}$, where

$$\widehat{s} = \underset{64\to 53}{\mathrm{round}}(\widehat{S}) = \begin{cases} f_2 & \text{if } |\widehat{S} - f_1|/|f_1| > u \\ f_1 \text{ or } f_2{}^{1} & \text{if } |\widehat{S} - f_1|/|f_1| = u \\ f_1 & \text{if } |\widehat{S} - f_1|/|f_1| < u \end{cases}. \qquad (4)$$

The following figure shows the principle:

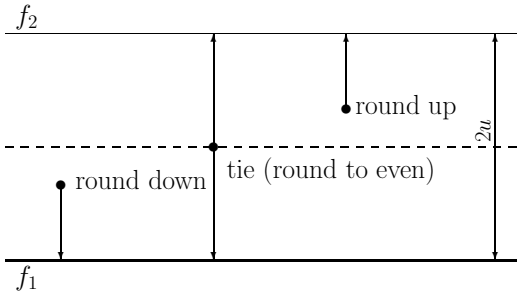


Fig. 1: Rounding a double extended precision number to double precision.

In all cases, the relative error can be computed in the same way as $|\widehat{s} - \widehat{S}|/|\widehat{s}| \leq u$ and is always bounded by $u$. □

**Note 1** *Theorem 5 is proved for positive numbers, but it can be proved for negatives too.*

**Corollary 6** *The expression* $\Delta = |\widehat{s} - \widehat{S}| \leq u|\widehat{s}|$ *provides an* 11*-bit estimate of the static rounding error and since* $|\Delta| \leq u|\widehat{s}|$*, it can be safely substituted for all occurrences of* $u|\widehat{s}|$ *in Tab. 1, providing a tighter error bound.*

## 3.2 Implementation highlights

Our approach is implemented in C++ language as a class `fdouble` with GCC AT&T inline assembly language FPU statements [9]. The `fdouble` class overloads most operators and some standard functions[2] which are beyond the scope of this paper, providing a handy replacement for the `double` data type. The transition from `double` data type to `fdouble` is performed just by changing the name of the data type. The rest is handled by the class. For an illustration of how the class works, the source of a plus operator `fdouble::operator+` and its assembly portion `op_plus` performing the FPU-supported running error analysis with the terms above follows:

```
fdouble fdouble::operator+(fdouble const &b) const
{
    fdouble r;

    op_plus( r.d, r.e, this->d, this->e, b.d, b.e );

    return r;
}
```

The `fdouble::operator+` method calls the `op_plus` function which contains the assembly inline and is common for all other `fdouble::operator+` overloads:

```
inline void op_plus( double& result,
                     double& result_err,
                     register const double a,
                     register const double a_err,
                     register const double b,
                     register const double b_err )
{
    __asm__ __volatile__(
            "fldl %1\n\t"           //  1. load b
            "fldl %2\n\t"           //  2. load a
            "faddp\n\t"             //  3. calc a+b
            "fstl %3\n\t"           //  4. store rounded result
            "fldl %3\n\t"           //  5. load rounded result
            "fsubp\n\t"             //  6. calc difference
            "fabs\n\t"              //  7. calc its absolute value
            "fldl %4\n\t"           //  8. load a_err
            "faddp\n\t"             //  9. calc diff + a_err
            "fldl %5\n\t"           // 10. load b_err
            "faddp"                 // 11. calc diff + a_err + b_err
            :
              "=t"(result_err)      // 12. put result in result_err
            :
              "m"(b),               // 13. %1 = b
              "m"(a),               // 14. %2 = a
              "m"(result),          // 15. %3 = result
              "m"(a_err),           // 16. %4 = a_err
              "m"(b_err)            // 17. %5 = b_err
            );
}
```

[2]Currently only logarithm, power and exponential function, absolute value, remainder and the square root are available.

The `op_plus` first loads both double precision $a$ and $b$ (lines 1 and 2) on the floating point stack and the FPU extends them to the double extended precision. The sum is then calculated (line 3) popping $a$ from the FPU stack and replacing $b$ by the sum. Since the result is still in double extended precision, it has to be stored back into memory in order to get a rounded result according to the Standard that we have to round the result ($\widehat{S}$) after each operation (line 4) and rounded result ($\widehat{s}$) is pushed back onto the floating point stack (line 5). $\widehat{S} - \widehat{s}$ is calculated (line 6) and its absolute value is determined (line 7). Errors are propagated according to Tab. 1 (lines 8 through 11), and the requested absolute error is found at the top of the FPU stack (line 12). Lines 13 through 17 specify input operand constraints for the `__asm__` directive. Other operators and functions are implemented in a similar way.

Traditional running error analysis is implemented alike, and provides class `radouble`. The source code for `radouble::operator+` follows:

```
radouble radouble::operator+(radouble const &b) const
{
    radouble r;

    r.d = d + b.d;
    r.e = fabs(r.d) * ROUNDOFF_UNIT + e + b.e;

    return r;
}
```

Here, we can see that no assembly language inlines are necessary, but it is necessary to provide special compiler flags in order to obtain the same results with the `fdouble` class. These GCC flags are: `-ffloat-store`, which ensures that the result of each floating point operation is stored back into memory and rounded to a defined precision (required by IEEE 754), and the second flag `-mfpmath=387`, which assures that the floating point unit is used. Without the second flag, the optimization could use SSE instructions and provide less accurate results. Readers interested in obtaining the complete source code for all classes should refer to [9] but cite this paper.

## 3.3 Complexity and implementation notes

The traditional running error analysis needs to calculate $u|\widehat{s}|$, and four x86 floating point instructions are necessary[3]. These are: `fld` instruction to load $u$ onto the floating point stack, `fmul` to calculate $u\widehat{s}$, `fabs` to obtain $|u\widehat{s}| = u|\widehat{s}|$, and `fstl` to store the result back into memory and round it to double precision.

Our approach suggests to substitute $|\widehat{s} - \widehat{S}|$ for $u|\widehat{s}|$ and a typical evaluation requires an `fldl` instruction to load $\widehat{s}$ onto the floating point stack, `fsub` to calculate $\widehat{s} - \widehat{S}$, `fabs` to obtain its absolute value $|\widehat{s} - \widehat{S}|$, and `fstl` to store the result into memory while rounding it to double precision.

According to instruction latency tables [1], `fsub` instruction latency is 3 cycles, while `fmul` latency is 5. The following table compares the two approaches from the latency point of view, proving that there is a speed enhancement:

Table 2: Comparing the FPU instructions necessary to evaluate the absolute error

| **Traditional approach** | | **Our approach** | |
|---|---|---|---|
| Instructions | Latency | Instructions | Latency |
| `fldl` | 3 | `fldl` | 3 |
| `fmul` | 5 | `fsub` | 3 |
| `fabs` | 1 | `fabs` | 1 |
| `fstl` | 3 | `fstl` | 3 |
| Total | 12 | Total | 10 |

Our approach not only provides a better error bound estimate, but also at a higher speed and we should expect about 20 per cent speed up.

## 3.4 Case study

The case study compares the presented approach to traditional running error analysis, and also compares it to a forward error bound determined a-priori on the following mathematical identity:

$$y_\infty = \log 2 = \sum_{k=1}^{\infty} \frac{(-1)^{k-1}}{k}. \tag{5}$$

C++ and Mathematica [8] software are used to verify the results. A rounding error analysis of (5) for a double precision floating point arithmetic for $N$ addends is given with the following assumptions:

[3] This approach works only with FPU, not SSE2/SSE3, as SSE does not support double extended precision and we are unable to calculate $|\widehat{S} - \widehat{s}|$.

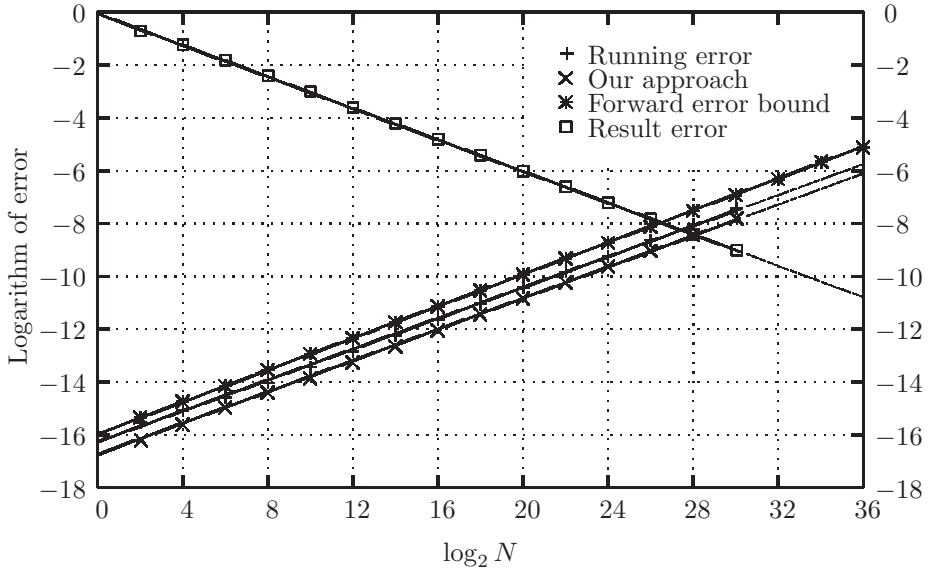


Fig. 2: Calculated value of $\widehat{y}_N$ and its error bounds (forward sum direction)

1. The first two addends (1 and 1/2) of the sum are exact as we use a binary floating point arithmetic,
2. $(-1)^{k-1}$ is evaluated as
$$(-1)^{k-1} = \begin{cases} 1 & \text{when } k \text{ is odd} \\ -1 & \text{otherwise} \end{cases}$$
rather than using `pow` function from `libm` thus the result is always exact,
3. all operations follow the Standard model.

The computed $\widehat{y}_N$ is expressed as:

$$\begin{aligned} \widehat{y}_N &= \left[\cdots\left[\left(1-\frac{1}{2}\right)(1+\delta_1)+\frac{1}{3}(1+\delta_2)\right]\right. \\ &\quad \left.(1+\delta_3)-\cdots\right](1+\delta_{2N-3}) = \qquad (6) \\ &= 1(1+\theta_N)-\frac{1}{2}(1+\theta_N)+\frac{1}{3}(1+\theta_N)- \\ &\quad \ldots+\frac{(-1)^{N-1}}{N}(1+\theta_2) \leq \qquad (7) \\ &\leq \left(1-\frac{1}{2}+\frac{1}{3}-\frac{1}{4}+-\cdots+\frac{(-1)^{N-1}}{N}\right) \\ &\quad (1+\theta_N) = (1+\theta_N)\sum_{k=1}^{N}\frac{(-1)^{k-1}}{k}, \qquad (8) \end{aligned}$$

where $|\delta_i| \leq u$, $\prod_{i=1}^{n}(1+\delta_i)^{\rho_i} = 1+\theta_n$, where $\rho_i = \pm 1$, and $|\theta_n| \leq \frac{nu}{1-nu} = \gamma_n$, and $nu < 1$ [5].

The backward error is immediately visible from equation (7), in which we can consider the sum as an exact sum of perturbed data entries by a relative value certainly bounded by $\gamma_n$. The forward a-priori error bound is then calculated as:

$$|\widehat{y}_N - y_N| \leq \gamma_N \sum_{k=1}^{N}\frac{(-1)^{k-1}}{k}, \qquad (9)$$

and presents the worst case error estimate, which can be far from true error bound. The following section demonstrates this statement.

### 3.4.1 Results

Results are provided for two summation orders. The first case performs the summation in decreasing order of magnitude, where a poor, insufficiently accurate result is quickly visible. The figure 2 depicts this scenario, where $N$ stands for number of iterations and the $y$-axis shows a base 10 logarithm of the order of the error. The first line, marked with $\square$, presents the absolute error, which is $\log_{10}(|\widehat{y}_N - \log 2|)$ and it gets smaller with each further accumulation of the sum. Rounding errors go against this error from bottom to top, and are represented with the remaining three lines. From top to bottom: The $\divideontimes$line presents the worst case, that is, the a-priori forward error bound which is obtained from the right hand side of (9). The second line, marked with $+$, stands for an a-posteriori running error bound. We can see that it is slightly better than the a-priori bound, and the last line ($\times$) presents our approach, which accounts real rounding errors, and that is the reason why the first two iterations have no error (cf. the $\times$ at the top).

We can also observe that if we use an a-priori bound, we will have to terminate the accumulation somewhere after $2^{26}$ iterations, and after $2^{27}$ iterations with the running error bound and approximately $2^{28}$ iterations with our approach. Calculating more iterations beyond the bound does not make sense in this case, because each accumulant is smaller than the error of the result.

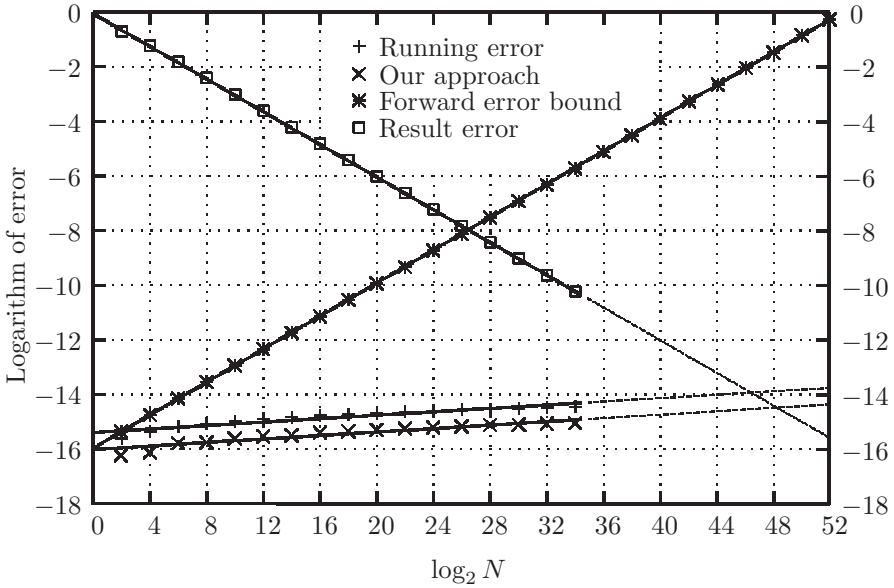


Fig. 3: Calculated value of $\widehat{y}_N$ and its error bounds (reverse sum direction)

The second case performs the sum in the reverse order, i.e. going from numbers of the smallest magnitude towards greater numbers (see figure 3). Note that this scenario demonstrates the claim that an error bound obtained a-priori (i.e. the forward error bound) is often too pessimistic and presents the worst case. Using it would lead to premature termination of the sum evaluation.

One more thing is demonstrated, i.e., that we should accumulate numbers in increasing order of magnitude. If we do not do that, numbers with small magnitude start to contribute less and less to a proportionally huge sum value, and we sooner or later find that further additions do not change the value of the sum. Due to this fact, the harmonic sequence $\sum_{k=1}^{\infty} \frac{1}{k} = \infty$ converges in finite arithmetics, and has a finite sum depending on the type of arithmetic.

The following table presents selected portions of the evaluation time for all two a-posteriori error analyses including the original computation. In addition, the table shows the run time for the `objdouble` class, which wraps the `double` data class that is used after subtracting the `double` column to determine the amount of the C++ object overhead.

The results were obtained with the POSIX `getrusage` API which provides, besides other information, the amount of used time in seconds and microseconds since the start of the process. For measuring purposes, these are internally converted into milliseconds and a difference of two millisecond values performs a measurement without interference with other processes and the operating system. Each value was measured 50 times and the results were statistically evaluated [2].

When we compare the traditional running error analysis run time to a simple evaluation in double precision, we obtain that object-oriented running error analysis with the `radouble` class is approximately 2.81 times slower. This slowdown includes the C++ overhead, consisting of object construction and operator calls. The overhead was measured with the `objdouble` class and the results show that the evaluation with `objdouble` class took 2 times longer than with the `double` data type. Our approach is only 2.35 times slower than the original approach with the `double` data type, and it is 1.19 times faster than the traditional approach. This speed up is what we have been expecting from Tab. 2.

Table 3: Run times for reverse sum evaluations of selected $N$ operations with four data types. These are `double` data type with no error analysis, the `radouble` data type, an object-oriented traditional running error analysis, `fdouble` data type which performs FPU-supported running error analysis, and the `objdouble` class, which wraps `double` data type and is used to measure the C++ overhead. All values are rounded to whole milliseconds

| $\log_2 N$ | `double` | `radouble` | `fdouble` | `objdouble` |
|---|---|---|---|---|
| 16 | 1 | 3 | 3 | 2 |
| 20 | 8 | 50 | 42 | 36 |
| 24 | 285 | 803 | 672 | 573 |
| 28 | 4 568 | 12 849 | 10 777 | 9 178 |
| 32 | 73 120 | 205 726 | 172 538 | 146 847 |

# 4 Conclusions

The traditional running error analysis approach uses the $u|\widehat{s}|$ term to get a rounding error estimate and, as we have shown, this estimate can be up to two times

larger than the actual rounding error. Moreover, this term always generates an error, regardless of whether an error was physically committed.

By exploiting the floating point unit's behavior of the Intel x86 platform, we are able to obtain an 11-bit error estimate by subtracting the rounded result $\widehat{s}$ from its not yet rounded equivalent $\widehat{S}$, and when divided by $\widehat{S}$, this estimate is always less than or equal to the unit roundoff $u$. Our approach is very similar to the traditional approach; it also needs four FPU instructions, but it replaces multiplication by subtraction and that saves 2 CPU cycles per evaluation, obtaining almost 20 per cent speedup over traditional running error analysis.

We encourage the use of running error analysis in all iterative tasks where critical cancellation can occur, such as during evaluation of a numeric derivative. As we have seen in the case study, an error bound determined a-priori can be far from the actual error and, especially with the provided classes, running error analysis provides a very quick and feasible replacement. This, however, costs 2.85 times the evaluation time, but when replaced by FPU-supported running error analysis, the cost is 2.35 while providing a yet tighter bound. With a tighter bound, we are able to calculate more iterations and be sure that the result is still valid.

## Acknowledgement

This research has been partially supported by the Ministry of Education, Youth, and Sport of the Czech Republic, under research program MSM 6840770014, and by the Czech Science Foundation as project No. 201/06/1039.

# References

[1] Fog, A.: *Instruction Tables: Lists of Instruction Latencies, Throughputs and Micro-operation Breakdowns for Intel and AMD CPU's*, Copenhagen University College of Engineering, available online as a part of *Software Optimization Resources*, http://www.agner.org/optimize/, 2008.

[2] Bevington, P., Robinson, D. K.: *Data Reduction and Error Analysis for the Physical Sciences*, McGraw-Hill Science/Engineering/Math, 2002.

[3] IEEE Computer Society Standards Committee. Working group of the Microprocessor Standards Subcommittee and American National Standards Institute: *IEEE Standard for Binary Floating-point Arithmetic*, ser. ANSI/IEEE Std 754-1985. IEEE Computer Society Press, 1985.

[4] Intel Corporation: *Intel® 64 and IA-32 Architectures Software Developer's Manual*, vol. **3A**, System Programming Guide, Part 1, 11/2007.

[5] Higham, N. J.: *Accuracy and Stability of Numerical Algorithms*, 2nd edition, Society for Industrial and Applied Mathematics, 2002.

[6] Wilkinson, J. H.: *The State of the Art in Error Analysis*, NAG Newsletter, vol. **2/85**, 1985.

[7] Wilkinson, J. H.: *Error Analysis Revisited*, IMA Bulletin, vol. **22**, no. 11/12, pp. 192–200, 1986.

[8] Wolfram Research, Inc.: *Mathematica 7*, http://www.wolfram.com/, 2008.

[9] Zahradnický, T.: *Fdouble Class, A Double Data Type Replacement C++ Class with the FPU-Supported Running Error Analysis.* Accessible online at http://service.felk.cvut.cz/anc/zahradt/fdouble.tar.gz, 2007.

**Ing. Tomáš Zahradnický**
was born on 9th March 1979, in Prague, Czech Republic. In 2003 he graduated (MSc) from the Department of Computer Science and Engineering of the Faculty of Electrical Engineering at the Czech Technical University in Prague. Since 2004 he has been a postgraduate student at the same department, where he became an assistant professor in 2007. Since 2009 he has been an assistant professor at the Department of Computer Systems at the Faculty of Information Technologies at the Czech Technical University in Prague. His scientific research focuses on system optimization, and on parameter extraction.

**Doc. Ing. Róbert Lórencz, CSc.**
was born on 10th August 1957 in Prešov, Slovak Republic. In 1981 he graduated (MSc) from the Faculty of Electrical Engineering at the Czech Technical University in Prague. He received his Ph.D. degree in 1990 from the Slovak Academy of Sciences. In 1998 he became an assistant professor at the Department of Computer Science and Engineering at the Faculty of Electrical Engineering at the Czech Technical University in Prague. In 2005 he defended his habilitation thesis and became an associate professor at the same department. Since 2009, he has worked as an associate professor at the Department of Computer Systems at the Faculty of Information Technologies at the Czech Technical University in Prague. His scientific research focuses on arithmetics for numerical algorithms, residual arithmetic, error-free computation and cryptography.

Ing. Tomáš Zahradnický
Doc. Ing. Róbert Lórencz, CSc.
E-mail: zahradt|lorencz@fit.cvut.cz
Department of Computer Systems
Faculty of Information Technologies
Czech Technical University in Prague
Kolejní 550/2, 160 00 Prague, Czech Republic