Compensated algorithms in floating point arithmetic: accuracy, validation, performances.

Nicolas Louvet

Directeur de thèse:

Philippe Langlois

Université de Perpignan Via Domitia

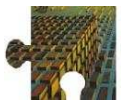
Laboratoire ELIAUS

Équipe DALI

UPVD

Université de Perpignan Via Domitia

ELIAUS



DALI

Dépôts, Architectures et Logiciels Informatiques

Introduction

Sources of errors when computing the solution of a scientific problem in floating point arithmetic:

- mathematical model,
- truncation errors,
- data uncertainties,
- rounding errors.

Introduction

Sources of errors when computing the solution of a scientific problem in floating point arithmetic:

- mathematical model,
- truncation errors,
- data uncertainties,
- rounding errors.

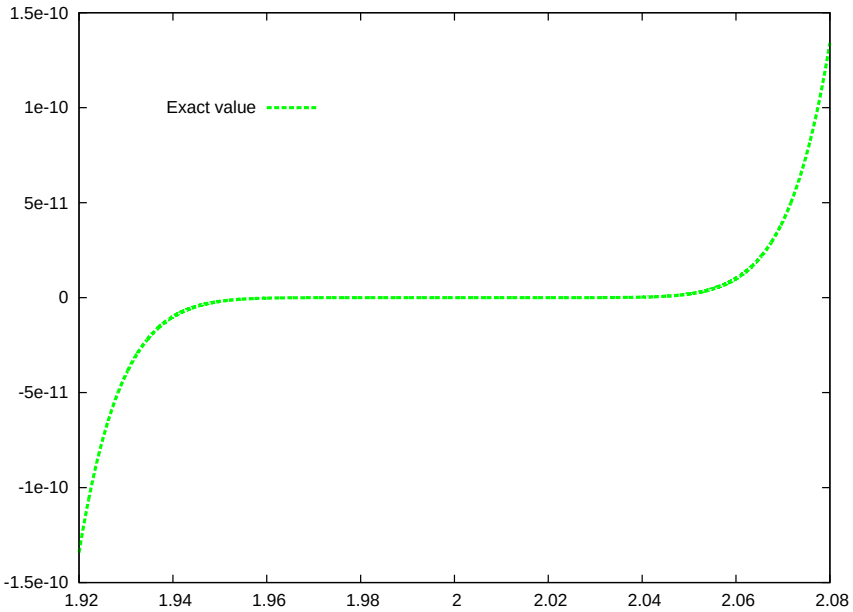
Rounding errors may totally corrupt a floating point computation:

- accumulation of billions of floating point operations,
- intrinsic difficulty to solve the problem accurately.

Example: polynomial evaluation

Evaluation of univariate polynomials with floating point coefficients:

- the evaluation of a polynomial suffers from rounding errors
- example : in the neighborhood of a multiple root

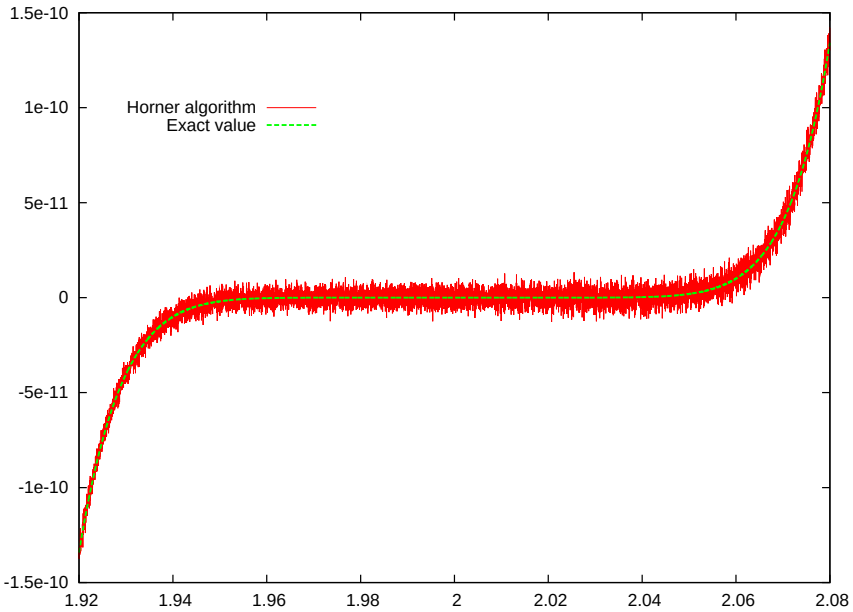


$p(x) = (x-2)^9$ in expanded form
near the multiple root $x = 2$

Example: polynomial evaluation

Evaluation of univariate polynomials with floating point coefficients:

- the evaluation of a polynomial suffers from rounding errors
- example : in the neighborhood of a multiple root



$p(x) = (x-2)^9$ in expanded form near the multiple root $x = 2$ evaluated with the Horner algorithm in IEEE double precision.

Motivation

How to improve and validate the accuracy of a floating point computation, without large computing time overheads ?

- Two case studies:
  - polynomial evaluation which occurs in many fields of scientific computing,
  - triangular system solving which is one of the basic algorithms in numerical linear algebra.
- Our main tool to improve the accuracy: compensation of the rounding errors.

Motivation

1. Improve the accuracy

   What to do when the best precision available in hardware is not sufficient?

   Increasing the working precision / compensated algorithm.

Motivation

1. Improve the accuracy

   What to do when the best precision available in hardware is not sufficient?

   Increasing the working precision / compensated algorithm.

2. Validate the computed result

   How to guarantee the quality of the compensated result?

   Validated algorithm = computed result + an *a posteriori* error bound.

Motivation

1. Improve the accuracy

   What to do when the best precision available in hardware is not sufficient?

   Increasing the working precision / compensated algorithm.

2. Validate the computed result

   How to guarantee the quality of the compensated result?

   Validated algorithm = computed result + an *a posteriori* error bound.

3. Maintain good performances

   How to explain good practical performances of compensated algorithms?

   Number of floating point operations / instruction-level parallelism.

Outline

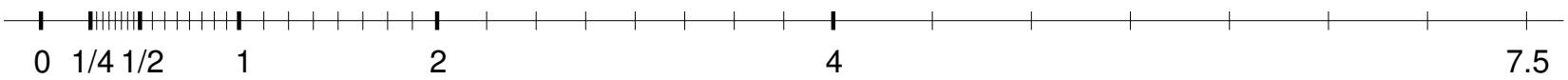
Floating point numbers

- $x \in \mathbb{F}$ is either 0, or a rational of the form

$$x = \pm \underbrace{1.x_1 x_2 \ldots x_{p-1}}_{p \text{ bit mantissa}} \times 2^e,$$

  with $x_i \in \{0, 1\}$, and e an exponent s.t. $e_{\min} \leq e \leq e_{\max}$.

- Example with $p = 4$, $e_{\min} = -2$ and $e_{\max} = 2$:

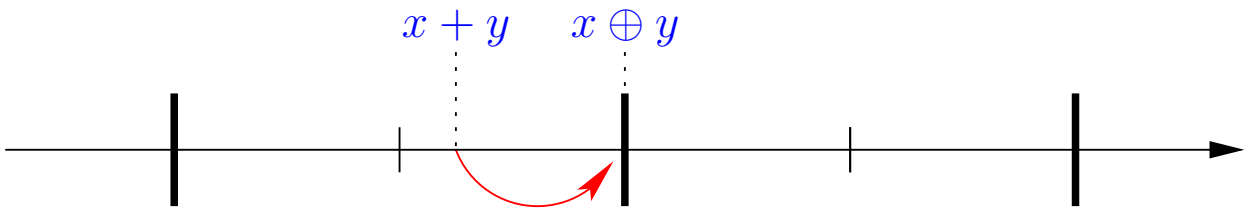


- $\mathbb{F}$ approximates $\mathbb{R}$: how are defined the arithmetic operations on $\mathbb{F}$?

Floating point arithmetic

Let $a, b \in \mathbb{F}$ and $\circ \in \{+, -, \times, /\}$ an arithmetic operation.

- $\mathrm{fl}(x \circ y)$ = the exact $x \circ y$ rounded to the nearest floating point value.



Every arithmetic operation may suffer from a rounding error.

- Standard model of floating point arithmetic :

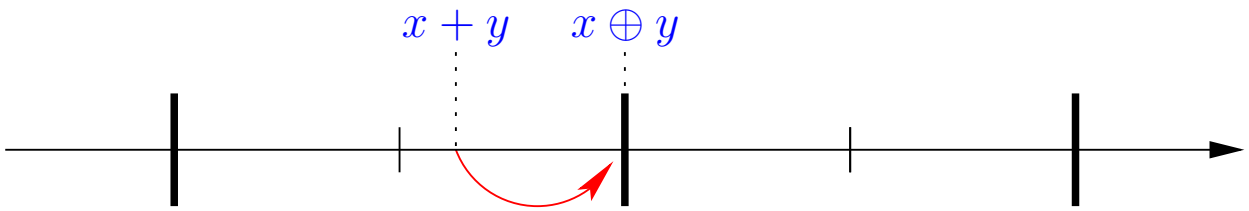
$$\mathrm{fl}(a \circ b) = (1 + \varepsilon)(a \circ b), \quad \text{with} \quad |\varepsilon| \leq \mathbf{u}.$$

Working precision $\mathbf{u} = 2^{-p}$ (in rounding to the nearest rounding mode).

Floating point arithmetic

Let $a, b \in \mathbb{F}$ and $\circ \in \{+, -, \times, /\}$ an arithmetic operation.

- $\mathrm{fl}(x \circ y)$ = the exact $x \circ y$ rounded to the nearest floating point value.



  Every arithmetic operation may suffer from a rounding error.

- Standard model of floating point arithmetic :

$$\mathrm{fl}(a \circ b) = (1 + \varepsilon)(a \circ b), \quad \text{with} \quad |\varepsilon| \leq \mathbf{u}.$$

  Working precision $\mathbf{u} = 2^{-p}$ (in rounding to the nearest rounding mode).

- In this talk: IEEE-754 binary fp arithmetic, rounding to the nearest, no underflow nor overflow.

How to improve the accuracy?

- Condition number measures the difficulty to solve the problem accurately.
  The "Rule of thumb" for backward stable algorithms:

$$\text{accuracy} \lesssim \text{condition number} \times \mathbf{u}.$$

How to improve the accuracy?

- Condition number measures the difficulty to solve the problem accurately.
  The "Rule of thumb" for backward stable algorithms:

$$\text{accuracy} \lesssim \text{condition number} \times \mathbf{u}.$$

- Classic solution to improve the accuracy: increasing the working precision $\mathbf{u}$.
  Hardware:
  - extended precision available in x87 fpu units.

  Software:
  - arbitrary precision library (working precision fixed by the programmer):
    MP, MPFUN/ARPREC, MPFR.
  - fixed length expansions libraries:
    double-double, quad-double (Briggs, Bailey *et al.*).

How to improve the accuracy?

- Condition number measures the difficulty to solve the problem accurately.
  The "Rule of thumb" for backward stable algorithms:

$$\text{accuracy} \lesssim \text{condition number} \times \mathbf{u}.$$

- Classic solution to improve the accuracy: increasing the working precision $\mathbf{u}$.
  Hardware:
  - extended precision available in x87 fpu units.

  Software:
  - arbitrary precision library (working precision fixed by the programmer):
    MP, MPFUN/ARPREC, MPFR.
  - fixed length expansions libraries:
    double-double, quad-double (Briggs, Bailey *et al.*).

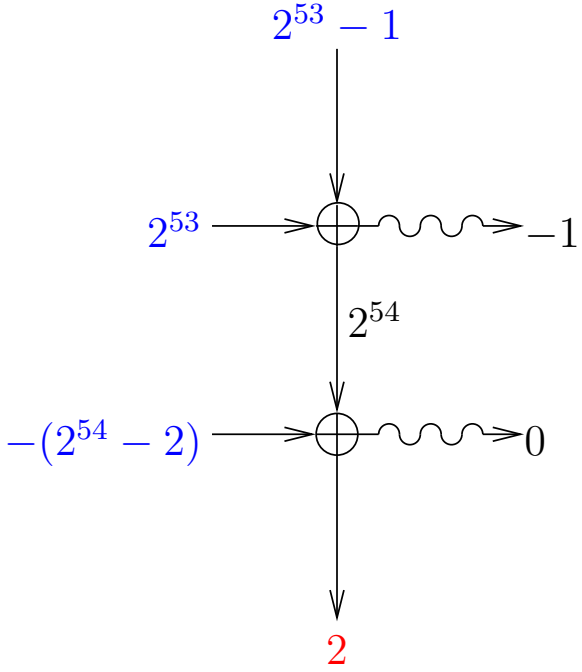
- Compensated algorithms: correction of the generated rounding errors.

Example: compensated summation

IEEE double precision numbers: $x_1 = 2^{53} - 1$, $x_2 = 2^{53}$ and $x_3 = -(2^{54} - 2)$.

Exact sum: $x_1 + x_2 + x_3 = 1$.

Classic summation



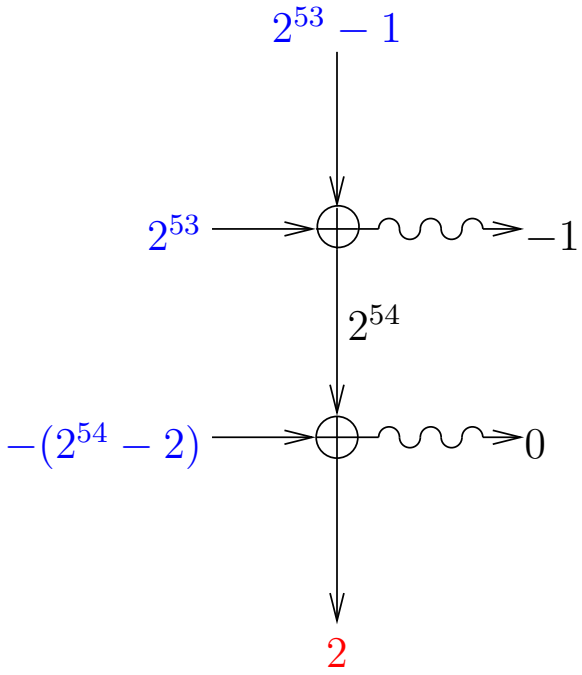
Relative error $= 1$

Example: compensated summation

IEEE double precision numbers: $x_1 = 2^{53} - 1$, $x_2 = 2^{53}$ and $x_3 = -(2^{54} - 2)$.

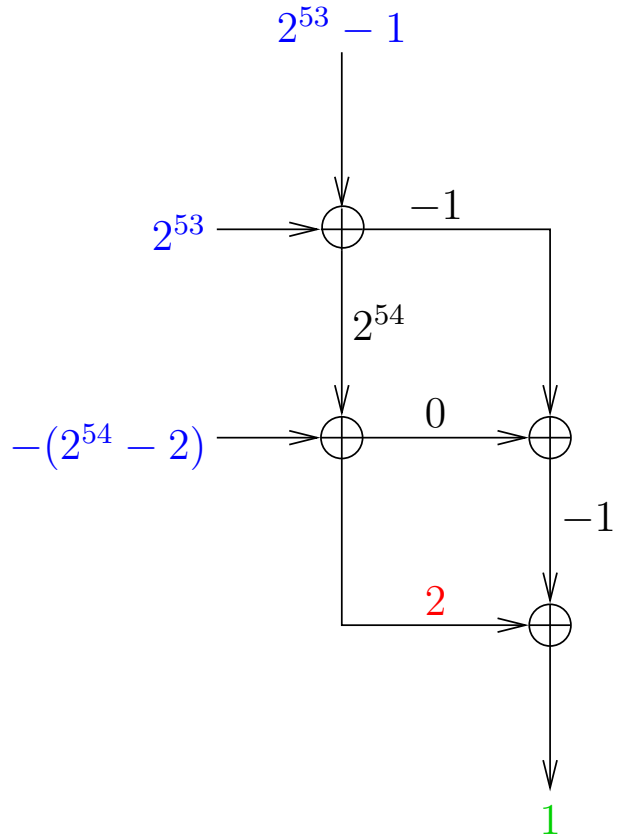
Exact sum: $x_1 + x_2 + x_3 = 1$.

Classic summation



Relative error = 1

Compensation of the rounding errors



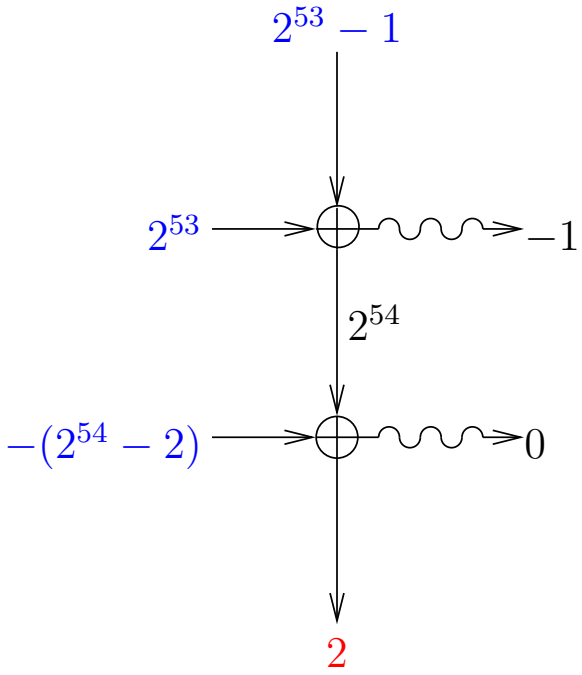
The exact result is computed

Example: compensated summation

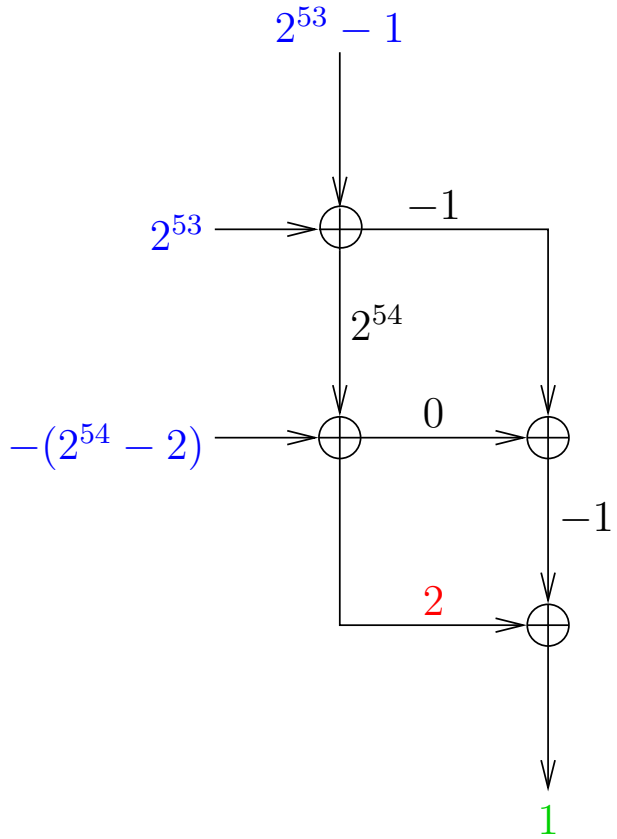
IEEE double precision numbers: $x_1 = 2^{53} - 1$, $x_2 = 2^{53}$ and $x_3 = -(2^{54} - 2)$.

Exact sum: $x_1 + x_2 + x_3 = 1$.

Classic summation



Compensation of the rounding errors



Relative error $= 1$

The exact result is computed

The rounding errors are computed thanks to *error-free transformations*.

Error-free transformations (EFT)

Error-Free Transformations are algorithms to compute the rounding errors at the current working precision.

$+$	$(x, y) = \text{2Sum}(a, b)$ such that $x = a \oplus b$ and $a + b = x + y$	6 flop	Knuth (74)
$\times$	$(x, y) = \text{2Prod}(a, b)$ such that $x = a \otimes b$ and $a \times b = x + y$	17 flop	Dekker (71)

with $a, b, x, y \in \mathbb{F}$.

Algorithm (Knuth)

function [x,y] = 2Sum(a,b)

$x = a \oplus b$

$z = x \ominus a$

$y = (a \ominus (x \ominus z)) \oplus (b \ominus z)$

Error-free transformations (EFT)

Error-Free Transformations are algorithms to compute the rounding errors at the current working precision.

$+$	$(x, y) = \text{2Sum}(a, b)$ such that $x = a \oplus b$ and $a + b = x + y$	6 flop	Knuth (74)
$\times$	$(x, y) = \text{2Prod}(a, b)$ such that $x = a \otimes b$ and $a \times b = x + y$	17 flop	Dekker (71)

with $a, b, x, y \in \mathbb{F}$.

Algorithm (Knuth)

function [x,y] = 2Sum(a,b)

$x = a \oplus b$

$z = x \ominus a$

$y = (a \ominus (x \ominus z)) \oplus (b \ominus z)$

Compensated summation algorithms:

- Kahan, Møller (1965),
- Pichat (1972),
- Neumanier (1974),
- Priest (1992),
- Ogita-Rump-Oishi (2005).

Outline

Accuracy of the Horner algorithm

We consider the polynomial

$$p(x) = \sum_{i=0}^{n} a_i x^i,$$

with $a_i \in \mathbb{F}$, $x \in \mathbb{F}$

Algorithm

function $r_0 = \text{Horner}(p, x)$

$r_n = a_n$

for $i = n-1 : -1 : 0$

$\quad r_i = r_{i+1} \otimes x \oplus a_i$

end

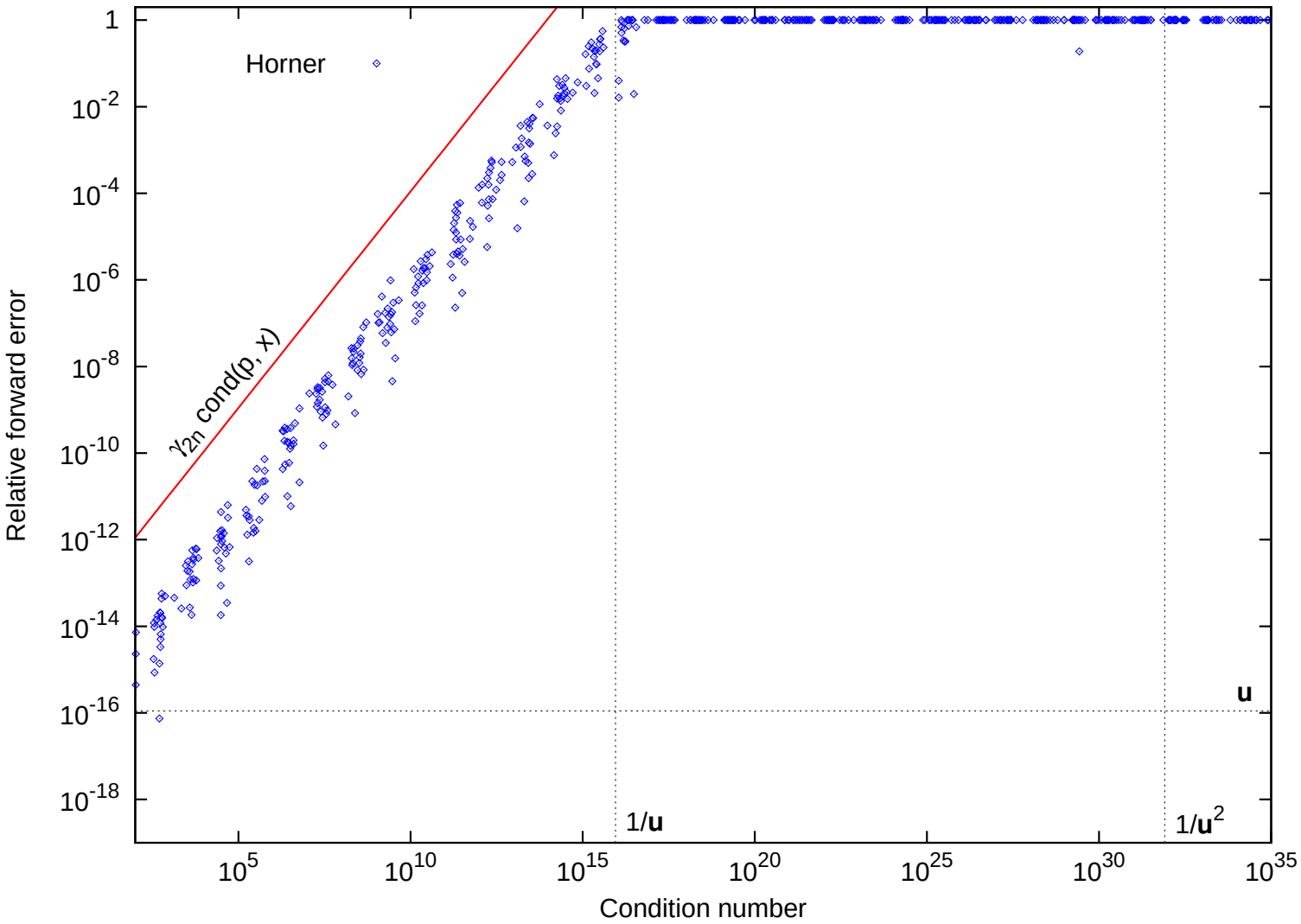
Relative accuracy of the evaluation with the Horner algorithm:

$$\frac{|\text{Horner}(p,x) - p(x)|}{|p(x)|} \leq \underbrace{\gamma_{2n}}_{\approx 2n\mathbf{u}} \text{cond}(p,x).$$

$\text{cond}(p,x)$ denotes the condition number of the evaluation:

$$\text{cond}(p,x) = \frac{\sum |a_i x^i|}{|p(x)|} \geq 1.$$

Accuracy $\lesssim$ condition number of the problem $\times$ **u**



How can we obtain more accuracy for polynomial evaluation?

EFT for the Horner algorithm

Consider $p(x) = \sum_{i=0}^{n} a_i x^i$ of degree n, $a_i, x \in \mathbb{F}$.

Algorithm (Horner)

function $r_0 = \text{Horner}(p, x)$

$r_n = a_n$

for $i = n-1 : -1 : 0$

$p_i = r_{i+1} \otimes x$ % error $\pi_i \in \mathbb{F}$

$r_i = p_i \oplus a_i$ % error $\sigma_i \in \mathbb{F}$

end

Let us define two polynomials p_π and p_σ such that:

$$p_\pi(x) = \sum_{i=0}^{n-1} \pi_i x^i \quad \text{and} \quad p_\sigma(x) = \sum_{i=0}^{n-1} \sigma_i x^i$$

EFT for the Horner algorithm

Consider $p(x) = \sum_{i=0}^{n} a_i x^i$ of degree n, $a_i, x \in \mathbb{F}$.

Algorithm (Horner)

function $r_0 = \text{Horner}(p, x)$

$r_n = a_n$

for $i = n-1 : -1 : 0$

$p_i = r_{i+1} \otimes x$ % error $\pi_i \in \mathbb{F}$

$r_i = p_i \oplus a_i$ % error $\sigma_i \in \mathbb{F}$

end

Let us define two polynomials p_π and p_σ such that:

$$p_\pi(x) = \sum_{i=0}^{n-1} \pi_i x^i \quad \text{and} \quad p_\sigma(x) = \sum_{i=0}^{n-1} \sigma_i x^i$$

Theorem (EFT for Horner algorithm)

$$\underbrace{p(x)}_{\text{exact value}} = \underbrace{\text{Horner}(p, x)}_{\in \mathbb{F}} + \underbrace{(p_\pi + p_\sigma)(x)}_{\text{forward error}}.$$

EFT for the Horner algorithm

Consider $p(x) = \sum_{i=0}^{n} a_i x^i$ of degree n, $a_i, x \in \mathbb{F}$.

Algorithm (Horner)

function $r_0 = \text{Horner}(p, x)$

$r_n = a_n$

for $i = n-1 : -1 : 0$

$p_i = r_{i+1} \otimes x$ % error $\pi_i \in \mathbb{F}$

$r_i = p_i \oplus a_i$ % error $\sigma_i \in \mathbb{F}$

end

Algorithm (EFT for Horner)

function $[r_0, p_\pi, p_\sigma] = \text{EFTHorner}(p, x)$

$r_n = a_n$

for $i = n-1 : -1 : 0$

$[p_i, \pi_i] = \text{2Prod}(r_{i+1}, x)$

$[r_i, \sigma_i] = \text{2Sum}(p_i, a_i)$

$p_\pi[i] = \pi_i \quad p_\sigma[i] = \sigma_i$

end

Theorem (EFT for Horner algorithm)

$$\underbrace{p(x)}_{\text{exact value}} = \underbrace{\text{Horner}(p, x)}_{\in \mathbb{F}} + \underbrace{(p_\pi + p_\sigma)(x)}_{\text{forward error}}.$$

Compensated Horner algorithm

$(p_\pi + p_\sigma)(x)$ is exactly the forward error affecting Horner(p, x).

$\Rightarrow$ we compute an approximate of $(p_\pi + p_\sigma)(x)$ as a correcting term.

Algorithm (Compensated Horner algorithm)

function $\overline{r} = \text{CompHorner}(p, x)$

$[\widehat{r}, p_\pi, p_\sigma] = \text{EFTHorner}(p, x)$ % $\widehat{r} = \text{Horner}(p, x)$

$\widehat{c} = \text{Horner}(p_\pi \oplus p_\sigma, x)$

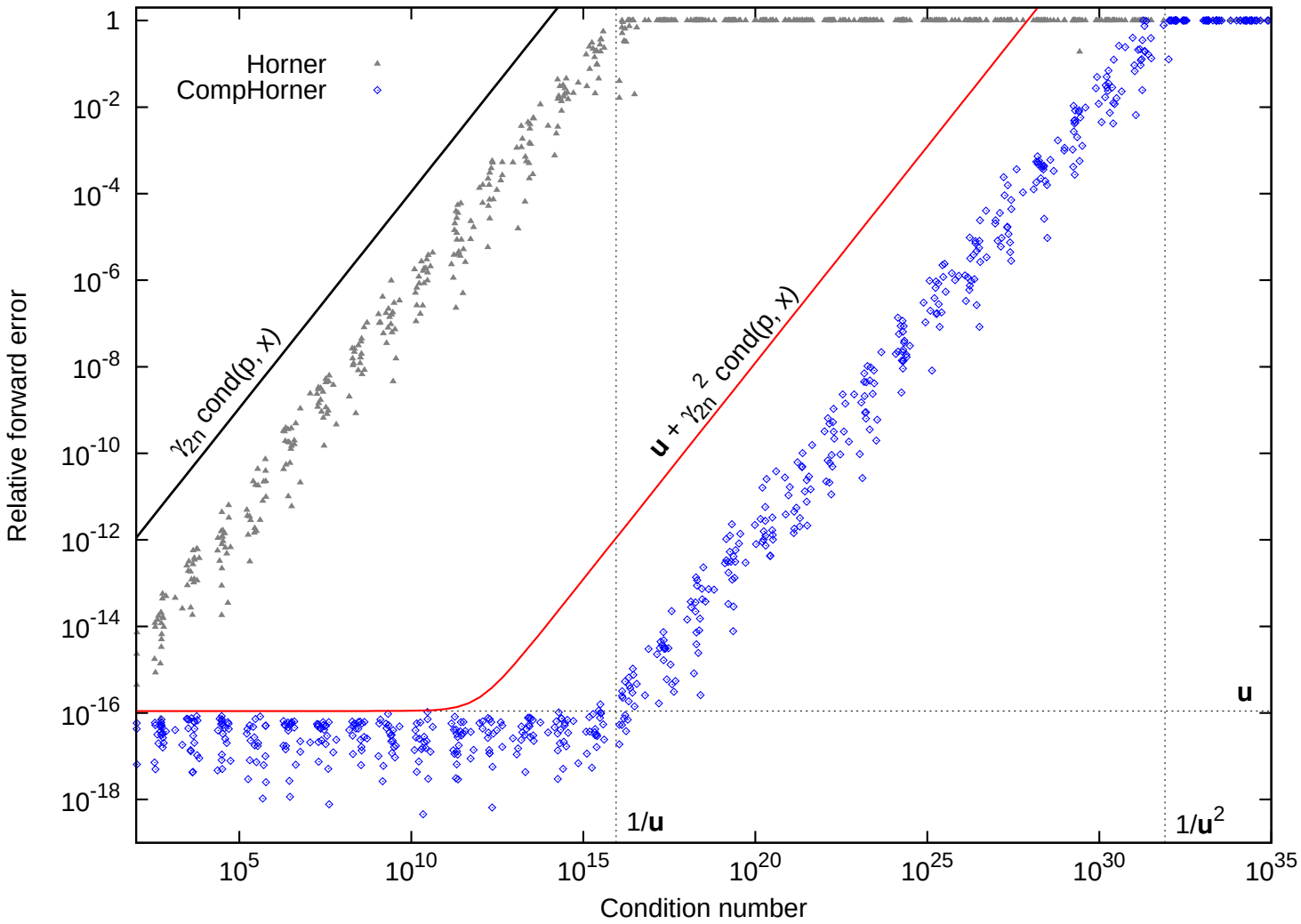
$\overline{r} = \widehat{r} \oplus \widehat{c}$

Theorem

Given p a polynomial with floating point coefficients, and $x \in \mathbb{F}$,

$$\frac{|\text{CompHorner}(p, x) - p(x)|}{|p(x)|} \leq \mathbf{u} + \underbrace{\gamma_{2n}^2}_{\approx (2n\mathbf{u})^2} \text{cond}(p, x).$$

Accuracy of the result $\lesssim \mathbf{u} +$ condition number $\times \mathbf{u}^2$.



The compensated Horner algorithm is as accurate as the classic Horner algorithm performed in twice the working precision, with a final rounding.

Outline

Validation of the compensated Horner algorithm

Consider a polynomial p of degree n with floating point coefficients, and $x \in \mathbb{F}$.

Algorithm

function $\overline{r} = \text{CompHorner}(p, x)$

$[\widehat{r}, p_\pi, p_\sigma] = \text{EFTHorner}(p, x)$

$\widehat{c} = \text{Horner}(p_\pi \oplus p_\sigma, x)$

$\overline{r} = \widehat{r} \oplus \widehat{c}$

A priori error bound for the compensated evaluation:

$$|\text{CompHorner}(p, x) - p(x)| \leq \mathbf{u}|p(x)| + \underbrace{\gamma_{2n}^2}_{\approx (2n\mathbf{u})^2} \widetilde{p}(x).$$

Problem: This *a priori* error bound

- can not be computed at running time, as $|p(x)|$ is "unknown";
- is pessimistic compared to the actual error.

Validated version of CompHorner

Consider a polynomial p of degree n with floating point coefficients, and $x \in \mathbb{F}$.

Algorithm

function $\overline{r} = \text{CompHorner}(p, x)$

$[\widehat{r}, p_\pi, p_\sigma] = \text{EFTHorner}(p, x)$

$\widehat{c} = \text{Horner}(p_\pi \oplus p_\sigma, x)$

$\overline{r} = \widehat{r} \oplus \widehat{c}$ % Rounding error $\delta = \widehat{r} + \widehat{c} - \overline{r} \in \mathbb{F}$.

Since EFTHorner is as error-free transformation, we have:

$$\underbrace{|\text{CompHorner}(p, x) - p(x)|}_{\substack{\text{error in the} \\ \text{compensated result}}} \leq |\delta| + \underbrace{|\widehat{c} - c|}_{\substack{\text{error in the} \\ \text{correcting term}}}.$$

Validated version of CompHorner

Consider a polynomial p of degree n with floating point coefficients, and $x \in \mathbb{F}$.

Algorithm

function $[\overline{r}, \beta] = \text{CompHornerBound}(p, x)$

if $2(n+1)\mathbf{u} \geq 1$, error('Validation impossible'), end

$[\widehat{r}, p_\pi, p_\sigma] = \text{EFTHorner}(p, x)$

$\widehat{c} = \text{Horner}(p_\pi \oplus p_\sigma, x)$

$[\overline{r}, \delta] = \text{2Sum}(\widehat{r}, \widehat{c})$ % Exact computation of δ

$\alpha = \left(\widehat{\gamma}_{2n-1} \otimes \text{Horner}(|p_\pi \oplus p_\sigma|, |x|)\right) \oslash (1 - 2(n+1)\mathbf{u})$

$\beta = (|\delta| \oplus \alpha) \oslash (1 - 2\mathbf{u})$

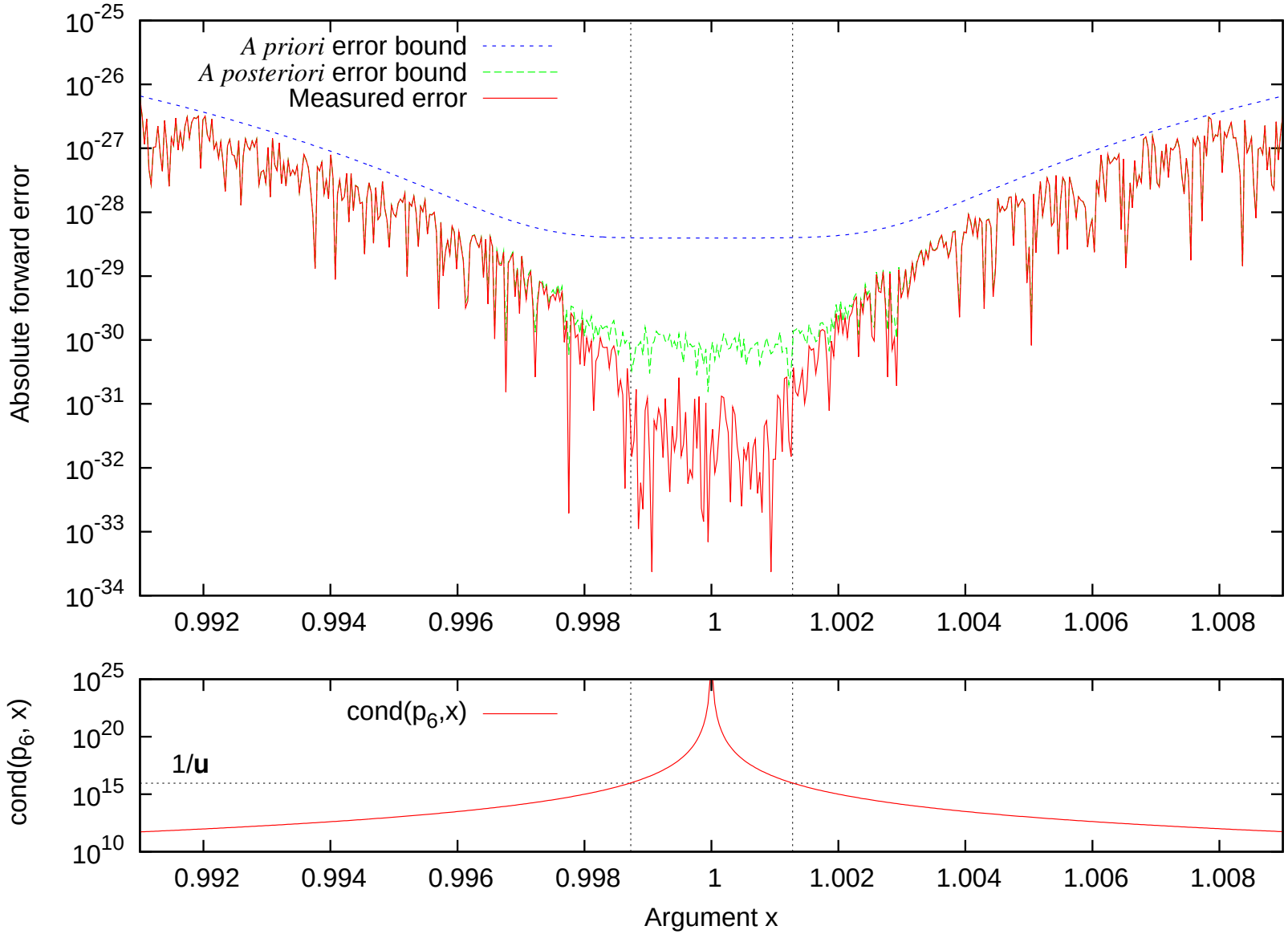
Theorem

Together with the compensated evaluation, $\text{CompHornerBound}(p, x)$ *computes an a posteriori error bound* β *s.t.*

$$|\text{CompHorner}(p, x) - p(x)| \leq \beta.$$

Sharpness of the *a posteriori* error bound

Evaluation of $p_6(x) = (1-x)^6$ in expanded form in the neighborhood of $x = 1$.



Conclusion

- Compensated Horner algorithm[1]:
  - as accurate as the Horner algorithm performed in doubled working precision,
  - very efficient compared to the double-double alternative.
- Validated version of the compensated Horner algorithm:
  - error bound computed using basic fp arithmetic, in RTN rounding mode.
  - runs at most 1.5 times slower than the non validated algorithm.

[1] Ph. Langlois & NL. How to ensure a faithful polynomial evaluation with the compensated Horner algorithm. IEEE ARITH 18, June 2007.

Conclusion

- Compensated Horner algorithm[1]:
  - as accurate as the Horner algorithm performed in doubled working precision,
  - very efficient compared to the double-double alternative.
- Validated version of the compensated Horner algorithm:
  - error bound computed using basic fp arithmetic, in RTN rounding mode.
  - runs at most 1.5 times slower than the non validated algorithm.

Other results:

- Faithful polynomial evaluation with the compensated Horner algorithm:
  - an *a priori* upper bound on the condition number to ensure faithful rounding,
  - an *a posteriori* test to check if the computed result is faithfully rounded.
- Study of the influence of underflow on *a priori* /*a posteriori* error bounds.

[1] Ph. Langlois & NL. How to ensure a faithful polynomial evaluation with the compensated Horner algorithm. IEEE ARITH 18, June 2007.

Outline

Overhead to obtain more accuracy

- We compare:
  - CompHorner = Compensated Horner algorithm
  - DDHorner = Horner algorithm + double-double (Bailey's library)

  Both provide the same output accuracy.
- Practical overheads compared to the classic Horner algorithm[1]:

		$\frac{\text{CompHorner}}{\text{Horner}}$	$\frac{\text{DDHorner}}{\text{Horner}}$	$\frac{\text{DDHorner}}{\text{CompHorner}}$
Pentium 4, 3.00 GHz	GCC 4.1.1	2.8	8.6	3.0
(x87 fp unit)	ICC 9.1	2.7	9.0	3.4
Athlon 64, 2.00 GHz	GCC 4.1.2	3.2	8.7	2.7
Itanium 2, 1.4 GHz	GCC 4.1.1	2.8	6.7	2.4
	ICC 9.1	1.5	5.9	3.9
		2 – 4	**6 – 9**	**2 – 4**

CompHorner runs a least two times faster than DDHorner.

[1] Average ratios for polynomials of degree 5 to 200; wp = IEEE-754 double precision

Motivation

- Floating point operations (flop) counts are commonly used to compare the performances of numerical algorithms.
- Flop counts for CompHorner and DDHorner are very similar.

	CompHorner	DDHorner
Flop count	$\mathbf{22n+5}$	$\mathbf{28n+4}$

- But CompHorner runs at least two times faster than DDHorner.
- Flop counts do not explain the performances of CompHorner compared to DDHorner[1]:

How to explain the practical performances of our compensated algorithm?

[1] The same property is identified but unexplained for compensated summation and dot product by Ogita, Rump and Oishi (05).

Instruction-level parallelism (ILP)

- Modern processors are designed to exploit the parallelism available among the instructions of a program, *i.e.* the instruction-level parallelism (ILP).
- Hennessy & Patterson, *Computer Architecture – A Quantitative Approach*:

  > *All processors since about 1985 [. . .], use pipelining to overlap the execution of instructions and improve performance. This potential overlap among instruction is called instruction-level parallelism since the instruction can be evaluated in parallel.*

- A wide range of techniques have been developed to exploit ILP, *e.g.* pipelining and superscalar architectures.

More ILP implies better performances on modern processors.

How to quantify instruction-level parallelism in a program?

Compute or estimate its Instruction Per Cycle (IPC) on an ideal processor.

[1] As defined by Hennessy & Patterson.

How to quantify instruction-level parallelism in a program?

Compute or estimate its Instruction Per Cycle (IPC) on an ideal processor.

- Ideal processor[1] = all the "artificial" constraints on the ILP are removed:
  - all but true data dependencies are removed;
  - an unlimited number of instructions can be executed in the same clock cycle;
  - any instruction is executed in one clock cycle;
  - memory accesses are perfect.

[1] As defined by Hennessy & Patterson.

How to quantify instruction-level parallelism in a program?

Compute or estimate its Instruction Per Cycle (IPC) on an ideal processor.

- Ideal processor[1] = all the "artificial" constraints on the ILP are removed:
  - all but true data dependencies are removed;
  - an unlimited number of instructions can be executed in the same clock cycle;
  - any instruction is executed in one clock cycle;
  - memory accesses are perfect.
- IPC = average number of instructions executed in one clock cycle:

$$\text{IPC} = \frac{\text{Total number of instructions}}{\text{Total latency of the program}}.$$

[1] As defined by Hennessy & Patterson.

How to quantify instruction-level parallelism in a program?

Compute or estimate its Instruction Per Cycle (IPC) on an ideal processor.

- Ideal processor[1] = all the "artificial" constraints on the ILP are removed:
  - all but true data dependencies are removed;
  - an unlimited number of instructions can be executed in the same clock cycle;
  - any instruction is executed in one clock cycle;
  - memory accesses are perfect.
- IPC = average number of instructions executed in one clock cycle:

$$\text{IPC} = \frac{\text{Total number of instructions}}{\text{Total latency of the program}}.$$

- Ideal IPC = IPC on the ideal processor.

The ideal IPC of a given program only depends on the data dependencies it contains.

[1] As defined by Hennessy & Patterson.

Ideal IPC of CompHorner (1/2)

C implementation of CompHorner

```
double CompHorner(double *P, int n, double x) {
  double p, r, c, pi, sig;
  double x_hi, x_lo, hi, lo, t;
  int i;
  /* Split(x_hi, x_lo, x) */
  t = x * _splitter_;
  x_hi = t - (t - x); x_lo = x - x_hi;
  r = P[n]; c = 0.0;
  for(i=n-1; i>=0; i--) {
    /* TwoProd(p, pi, s, x); */
    p = r * x;
    t = r * _splitter_;
    hi = t - (t - r);
    lo = r - hi;
    pi = (((hi*x_hi - p) + hi*x_lo)
          + lo*x_hi) + lo*x_lo;
    /* TwoSum(s, sigma, p, P[i]); */
    r = p + P[i];
    t = r - p;
    sig = (p - (r - t)) + (P[i] - t);
    /* Computation of the error term */
    c = c * x + (pi+sig);
  }
  return(r+c);
}
```

- To evaluate a polynomial of degree n:
  - n iterations,
  - $22n + 5$ flop.
- Ideal IPC of CompHorner(p, x):

$$\text{IPC}_{\text{CompHorner}} \approx \frac{22n+5}{\begin{array}{c}\text{Latency for}\\ \text{n iterations}\end{array}}.$$

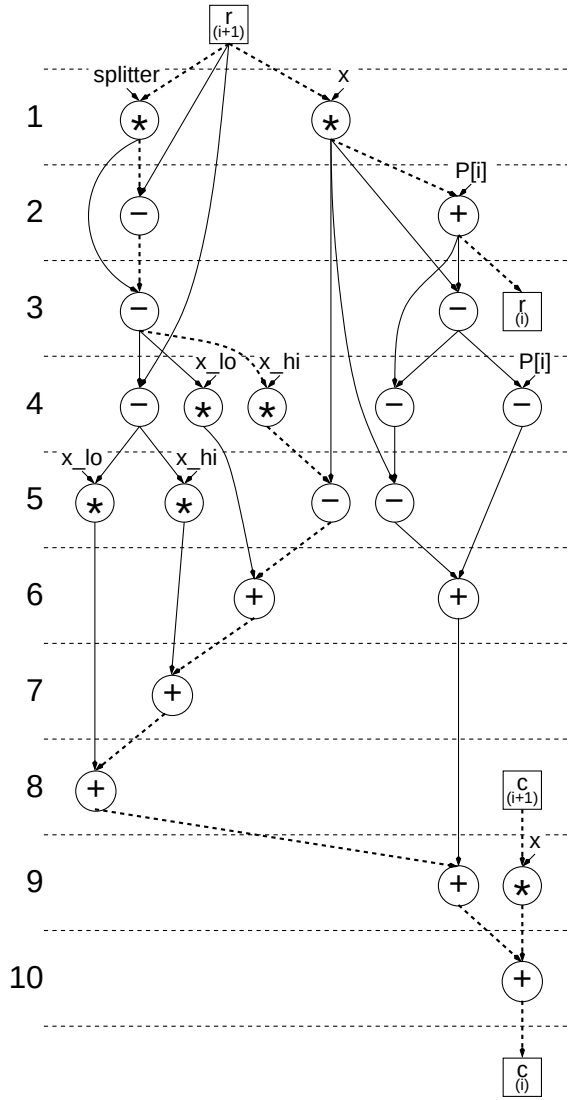
- The total latency is determined by data dependencies.

Ideal IPC of CompHorner (2/2)

- We determine the latency for the execution of n iterations from the data flow graph for one iteration

  $\Longrightarrow$ best scheduling for one iteration.

- Study of this data flow graph shows that:
  - the latency of one iteration is 10 cycles.
  - consecutive iterations overlap by 8 cycles.



Ideal IPC of CompHorner (2/2)

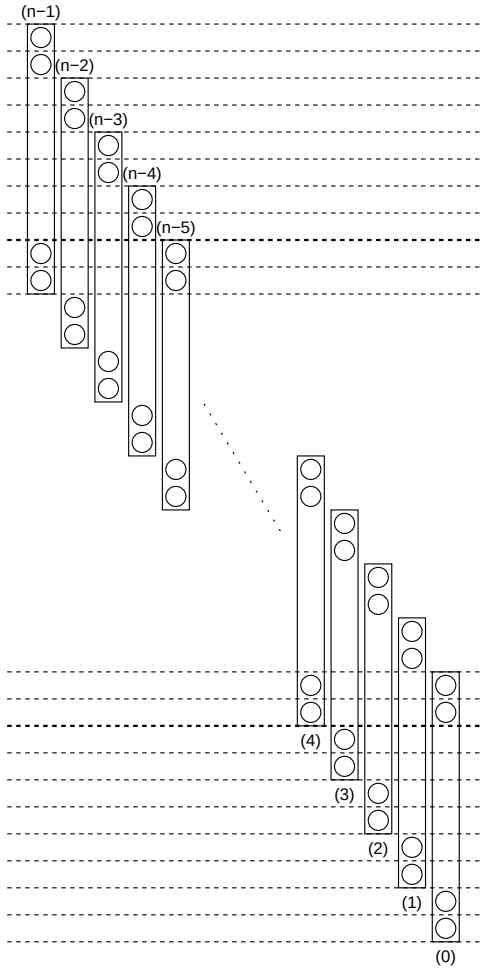
- We determine the latency for the execution of n iterations from the data flow graph for one iteration

  $\Longrightarrow$ best scheduling for one iteration.

- Study of this data flow graph shows that:
  - the latency of one iteration is 10 cycles.
  - consecutive iterations overlap by 8 cycles.

- The latency for n iterations is $2n + 8$, thus

$$\mathrm{IPC}_{\mathrm{CompHorner}} \approx \frac{22n + 5}{2n + 8} \approx 11.$$



Ideal IPC of DDHorner (1/2)

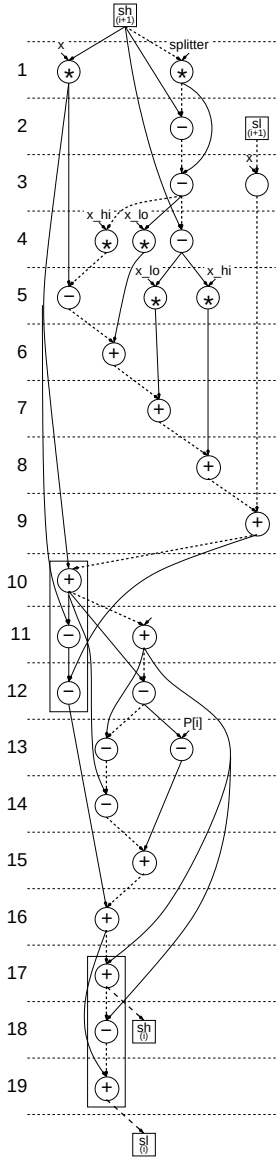
C implementation of DDHorner

```
double DDHorner(double *P, int n, double x) {
  double r_h, r_l, t_h, t_l, x_hi, x_lo, hi, lo, t;
  int i;
  /* Split(x_hi, x_lo, x) */
  t = x * _splitter_;
  x_hi = t - (t - x); x_lo = x - x_hi;
  r_h = P[deg]; r_l = 0.0;
  for(i=deg-1; i>=0; i--) {
    /* (r_h, r_l) = (r_h, r_l) * x */
    t = r_h * _splitter_;
    hi = t - (t - r_h);
    lo = (r_h - hi);
    t_h = r_h * x;
    t_l = (((hi*x_hi-t_h) + hi*x_lo)
          + lo*x_hi) + lo*x_lo;
    t_l += r_l * x;
    r_h = t_h + t_l;
    r_l = (t_h - r_h) + t_l;
    /* (r_h, r_l) = (r_h, r_l) + P[i] */
    t_h = r_h + P[i];
    t = t_h - r_h;
    t_l = ((r_h - (t_h - t)) + (P[i] - t));
    t_l += r_l;
    r_h = t_h + t_l;
    r_l = (t_h - r_h) + t_l;
  }
  return(r_h);
}
```

- To evaluate a polynomial of degree n:
  - n iterations,
  - $28n + 4$ flop.
- Ideal IPC of CompHorner(p, x):

$$\mathrm{IPC}_{\mathrm{CompHorner}} \approx \frac{28n+4}{\begin{array}{c}\text{Latency for}\\ \text{n iterations}\end{array}}.$$

Ideal IPC of DDHorner (2/2)



- With the same reasoning:
  - the latency of one iteration is 19 cycles.
  - consecutive iterations overlap by 2 cycles.
- The latency for n iterations is $17n + 2$, thus

$$\mathrm{IPC}_{\mathrm{DDHorner}} \approx \frac{22n + 5}{17n + 2} \approx 1.65.$$

Conclusion

- On the ideal processor:

$$\mathrm{IPC}_{\mathrm{CompHorner}} \approx 6.8 \times \mathrm{IPC}_{\mathrm{DDHorner}}.$$

- More *instruction-level parallelism* in CompHorner than in DDHorner.

This gives a *qualitative* explanation of the practical performances of CompHorner compared to DDHorner.

Other results[1]:

- This is due to the fact that renormalization steps are avoided in CompHorner.
- The same conclusion holds for other compensated algorithms compared to their double-double counterparts (*e.g.* compensated summation and dot product by Ogita *et al.*)

[1] Ph. Langlois & NL. More Instruction Level Parallelism Explains the Actual Efficiency of Compensated Algorithms. Submitted to IEEE TC.

Outline

Compensated Horner algorithm with a FMA

Fused-Mulitply-and-Add (FMA): given $a, b, c \in \mathbb{F}$

$$\mathrm{FMA}(a, b, c) = a \times b + c \text{ rounded in the current rounding mode.}$$

Motivations:

- How to benefit from FMA within the compensated Horner algorithm?
- Two known results about EFT in presence of a FMA:
  1. FMA is useful to design an efficient EFT for fp multiplication,
  2. an EFT for the FMA has been proposed by Boldo and Muller (2005).

[1] S. Graillat & Ph. Langlois & NL. Improving the compensated Horner scheme with a fused multiply and add. ACM SAC 2006.

[2] Ph. Langlois & NL. Operator dependant compensated algorithms. SCAN 2006.

Compensated Horner algorithm with a FMA

Fused-Mulitply-and-Add (FMA): given $a, b, c \in \mathbb{F}$

$$\mathrm{FMA}(a, b, c) = a \times b + c \text{ rounded in the current rounding mode.}$$

Motivations:

- How to benefit from FMA within the compensated Horner algorithm?
- Two known results about EFT in presence of a FMA:
  1. FMA is useful to design an efficient EFT for fp multiplication,
  2. an EFT for the FMA has been proposed by Boldo and Muller (2005).

Results[1] [2]:

- Two improved versions of the compensated Horner algorithm with a FMA.
- It is more efficient to compensate the rounding errors generated by multiplications.

[1] S. Graillat & Ph. Langlois & NL. Improving the compensated Horner scheme with a fused multiply and add. ACM SAC 2006.

[2] Ph. Langlois & NL. Operator dependant compensated algorithms. SCAN 2006.

CompHornerK: K times the working precision

Motivations:

- Polynomial evaluation with the compensated Horner algorithm,

$$\text{cond}(p, x) \lesssim \mathbf{u}^{-1} \quad \Longrightarrow \quad \text{accuracy of the compensated result} \approx \mathbf{u}.$$

- How to deal with condition numbers larger than $\mathbf{u}^{-1}$?

CompHornerK: K times the working precision

Motivations:

- Polynomial evaluation with the compensated Horner algorithm,

$$\text{cond}(p, x) \lesssim \mathbf{u}^{-1} \quad \Longrightarrow \quad \text{accuracy of the compensated result} \approx \mathbf{u}.$$

- How to deal with condition numbers larger than $\mathbf{u}^{-1}$?

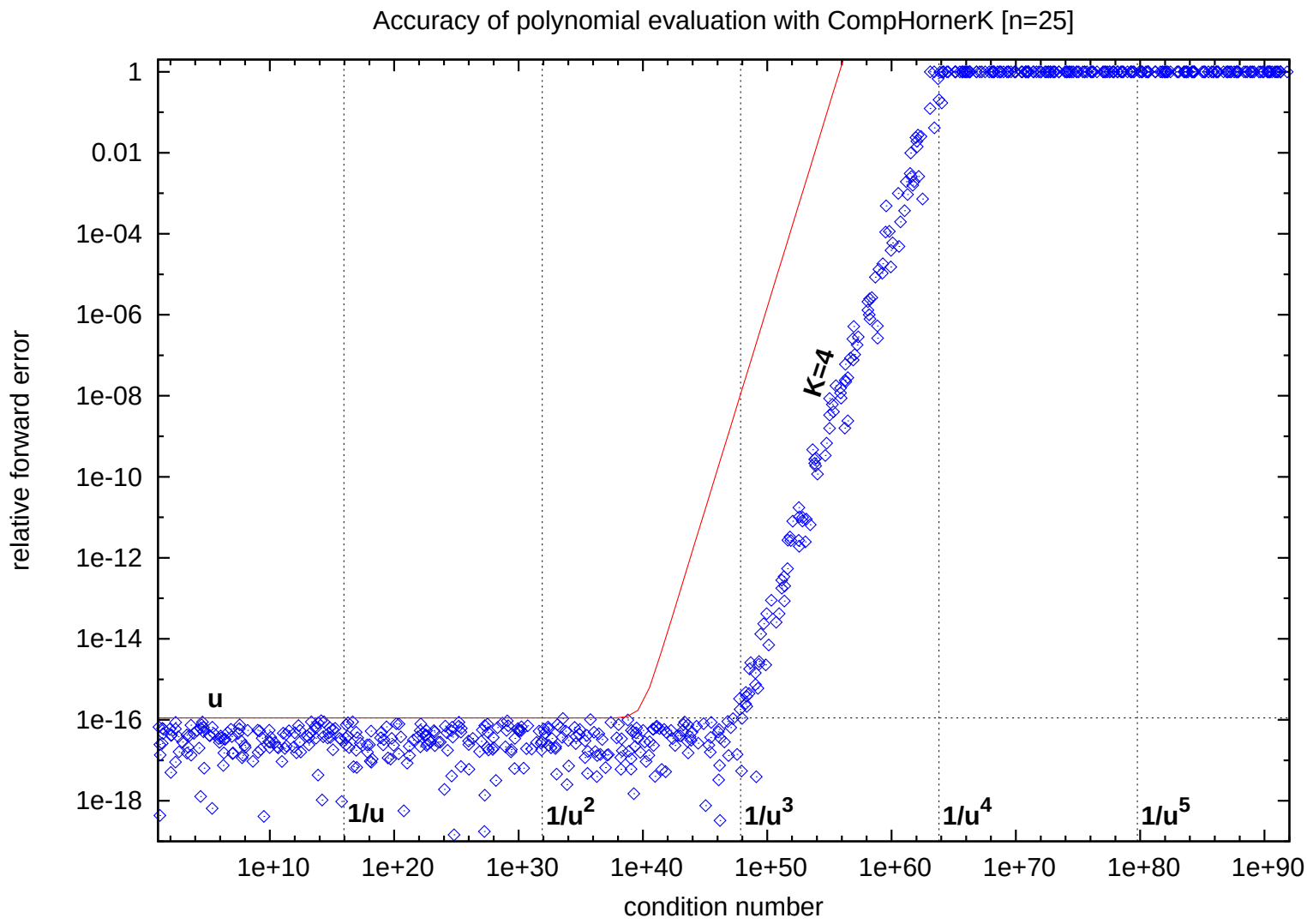
Results:

- CompHornerK: a new compensated algorithm.
  - Recursive application of EFTHorner on $K-1$ levels.
  - Provides K times the working precision,

$$\text{relative accuracy} \lesssim \mathbf{u} + \text{cond}(p, x) \times \mathbf{u}^K.$$

- More efficient than generic solutions (MPFR, quad-double) for $K \leq 4$.
- We also propose a validated version of CompHornerK.

$K = 4$: accuracy $\lesssim \mathbf{u} +$ condition number $\times \mathbf{u}^4$



Triangular linear system and substitution algorithm

Motivations:

- Consider a nonsingular triangular lower matrix $T \in \mathbb{F}^{n\times n}$, and $b^T \in \mathbb{F}^n$

$$\begin{pmatrix} t_{1,1} & & \\ \vdots & \ddots & \\ t_{n,1} & \cdots & t_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}.$$

  Classic substitution algorithm solves $Tx = b$ according to,

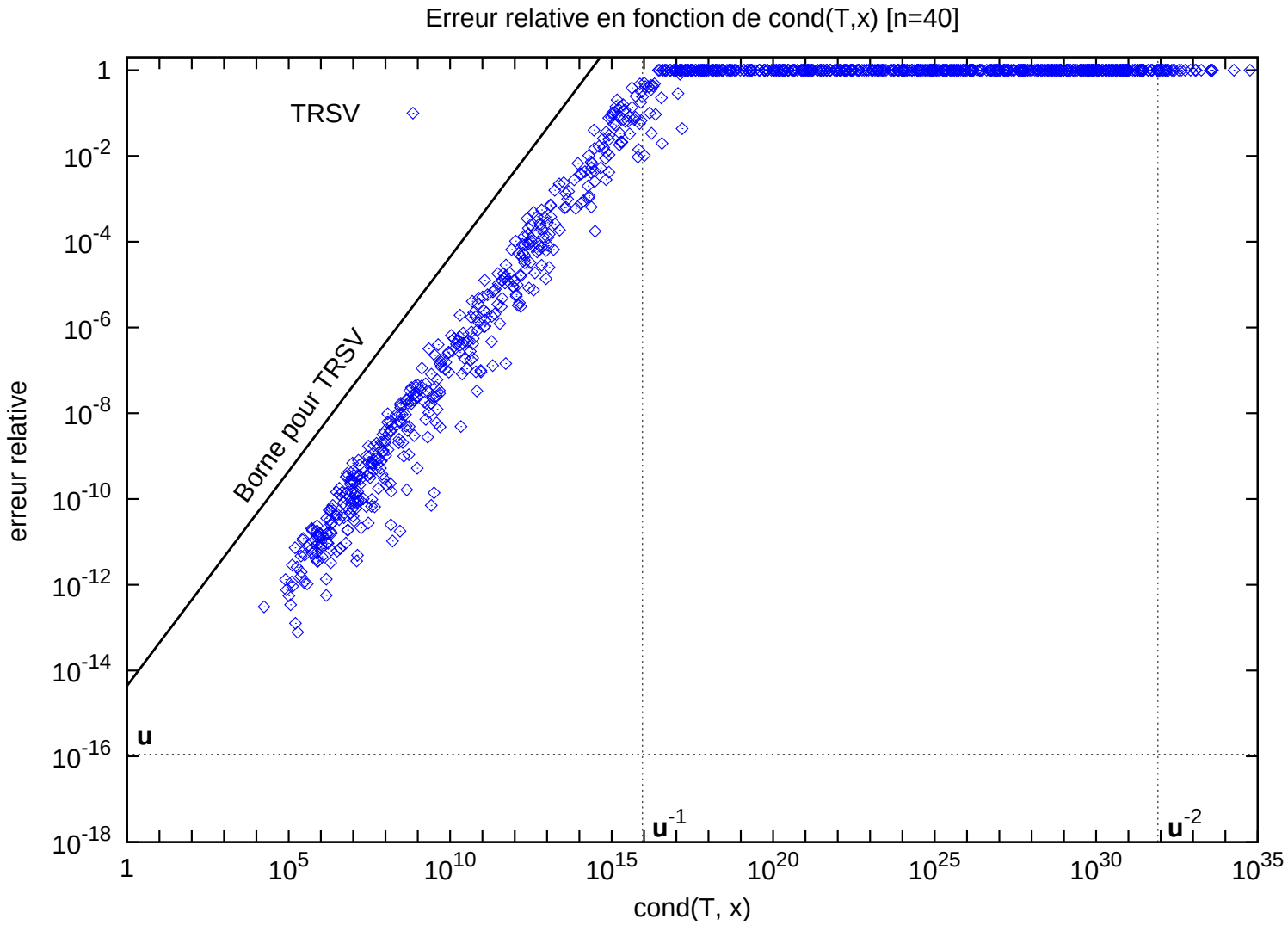
$$x_k = \frac{1}{t_{k,k}} \left(b_k - \sum_{i=1}^{k-1} t_{k,i} x_i \right) \quad \text{for} \quad k = 1, \ldots, n.$$

- Accuracy of the computed solution:

$$\frac{\|\widehat{x} - x\|_\infty}{\|x\|_\infty} \lesssim n\mathbf{u} \times \text{cond}(T, x),$$

  where $\text{cond}(T, x)$ denotes Skeel's condition number.

Accuracy of the solution computed by substitution



How to compensate the rounding errors generated by the substitution algorithm?

Compensated triangular system solving

Results[1]**:**

- A compensated substitution algorithm : CompTRSV.
  - Same practical accuracy
  - and two times faster

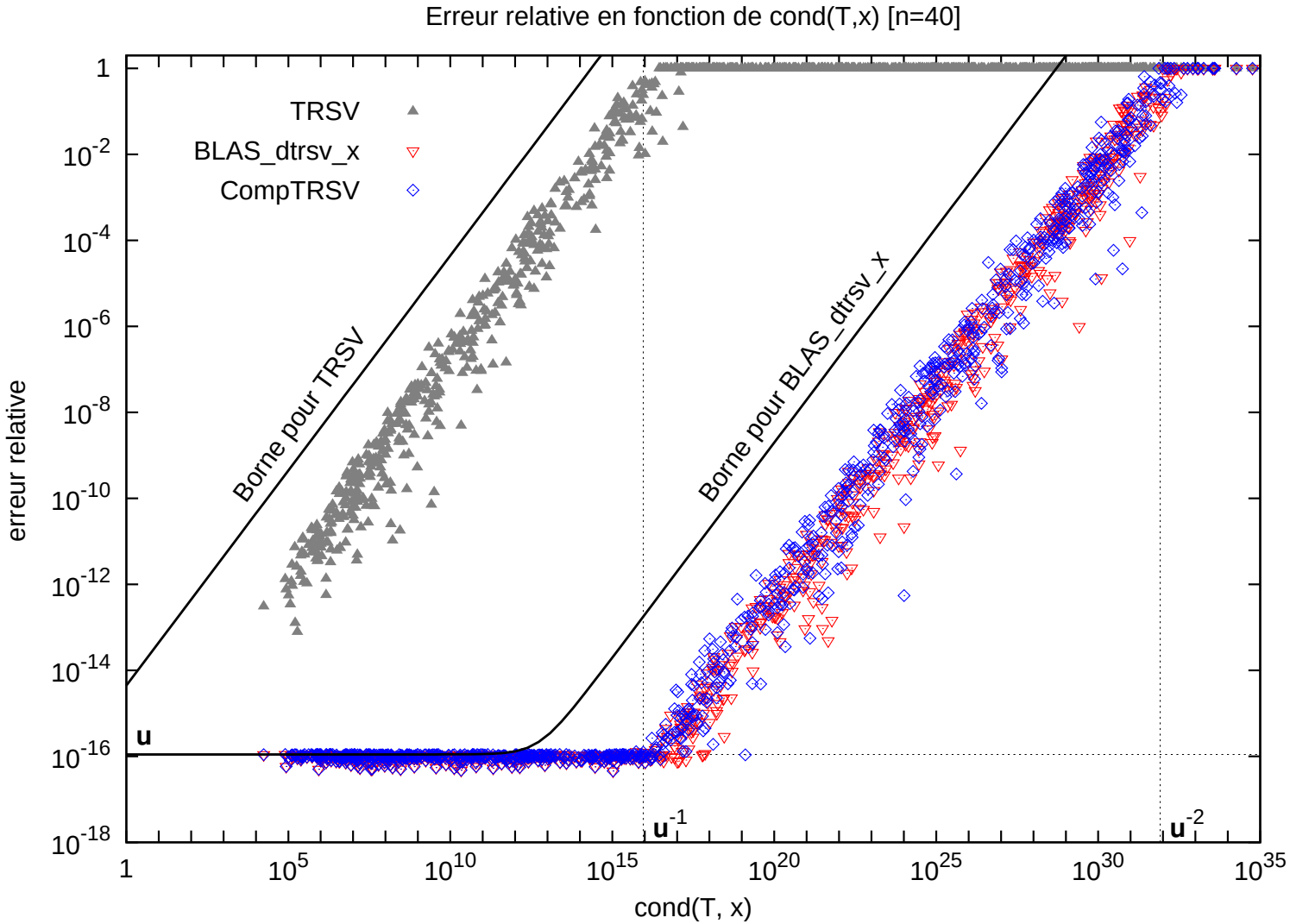
  than substitution with double-double arithmetic from the XBLAS library.

- Algorithmic equivalence between
  - computing a compensated solution to a triangular system
  - and performing one step of iterative refinement to improve the solution computed by classic substitution algorithm.

- *A priori* error analysis of CompTRSV.

[1] Ph. Langlois & NL. Solving triangular systems more accurately and efficiently. 17th IMACS World Congress, 2005.

Practical accuracy w.r.t. cond(T, x)



The compensated solution $\overline{x}$ is in practice as accurate as if it was computed by the substitution algorithm in twice the working precision.

Outline

Improving the accuracy

Summary:

Efficient alternatives to generic solutions (double-double, quad-double, MPFR):

- Compensated Horner algorithm
  - twice the working precision,
  - improved version with a FMA,
  - K times the working precision.
- Compensated triangular system solving.

Future work:

- Generalization of the compensated Horner algorithm:
  - evaluation of multivariate polynomials,
  - more accurate evaluation of the derivatives of a polynomial.
- One step of iterative refinement for general linear systems: *a priori* results ?

Validation of numerical quality

Summary:

- Validated version of the compensated algorithm,
  - *a posteriori* error bound for the compensated Horner algorithm,
  - dynamical test for faithful rounding of the compensated evaluation,
  - *a posteriori* error bound for CompHornerK.
- Good performances: only classic fp arithmetic in round-to-nearest.

Future work:

- Comparison of our methods with state of art methods (interval arithmetic): sharpness of the error bounds *vs.* cost of the computation.
- Validation of the compensated substitution algorithm.

Performances

Summary:

- Performances of compensated algorithms justifies their practical interest: twice faster than generic solutions with the same accuracy (double-double).
- Detailed study of the performances of the compensated Horner algorithm:
  - more *Instruction level parallelism* than with double-double arithmetic.
  - we shown it is due to the absence of "renormalization steps".
- Same conclusions hold for other compensated algorithms.

Future work:

New processors designed for floating point computations:
Cell (IBM, Sony, Toshiba), GPU, fp coprocessors (CSX600 Clear Speed).

- How to implement compensated algorithms efficiently?
- How do they perform on these new architectures?

Compensated algorithms in floating point arithmetic: accuracy, validation, performances.

Nicolas Louvet

Directeur de thèse:

Philippe Langlois

Université de Perpignan Via Domitia

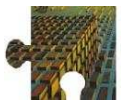
Laboratoire ELIAUS

Équipe DALI

UPVD

Université de Perpignan Via Domitia

ELIAUS



DALI

Dépôts, Architectures et Logiciels Informatiques