A survey of multiple precision computation using floating-point arithmetic

Fourth International Workshop on Taylor Methods

Christoph Quirin Lauter

Laboratoire de l'Informatique et du Parallélisme
École Normale Supérieure de Lyon

Boca Raton, December 16 - 19

Motivation

Motivation

Exact floating-point arithmetic

Double-double, triple-double and expansion arithmetic

Project by Arénaire at ENS de Lyon

`crlibm`[1]: correctly rounded elementary function library

---

[1] `http://lipforge.ens-lyon.fr/www/crlibm/`

Project by Arénaire at ENS de Lyon

`crlibm`[1]: correctly rounded elementary function library

- Elementary functions as in an usual `libm`:
  - `exp`
  - `sin`
  - `cos`
  - ...

---

[1] `http://lipforge.ens-lyon.fr/www/crlibm/`

Project by Arénaire at ENS de Lyon

`crlibm`[1]: correctly rounded elementary function library

- Elementary functions as in an usual `libm`:
  - `exp`
  - `sin`
  - `cos`
  - ...
- Evaluating elementary functions means evaluating polynomials

---

[1] `http://lipforge.ens-lyon.fr/www/crlibm/`

Project by Arénaire at ENS de Lyon

`crlibm`[1]: correctly rounded elementary function library

- Elementary functions as in an usual `libm`:
  - `exp`
  - `sin`
  - `cos`
  - ...
- Evaluating elementary functions means evaluating polynomials
- Correct rounding requires high accuracy and complete proofs

---

[1] `http://lipforge.ens-lyon.fr/www/crlibm/`

Need for more precision

- IEEE 754 double precision offers 53 bits of precision
- In `crlibm`, we need an accuracy of "120 correct bits"
- In Taylor models, no use of high order polynomials if the remainder grows too fast

Need for more precision

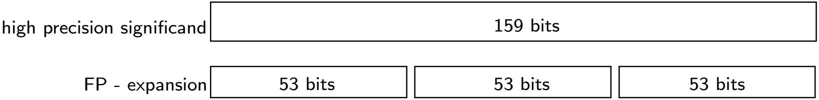
- IEEE 754 double precision offers 53 bits of precision
- In `crlibm`, we need an accuracy of “120 correct bits”
- In Taylor models, no use of high order polynomials if the remainder grows too fast
- First approach:
  - Use an integer based fixed high precision floating-point library
  - Necessity to leave the floating-point pipeline
  - High impact on performance (factor 100)

Need for more precision

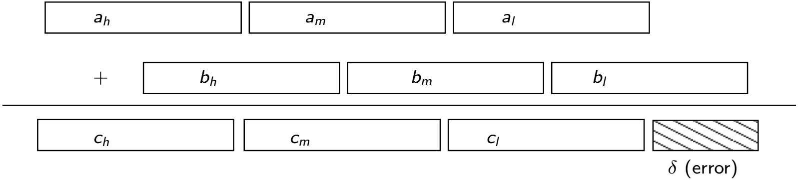
- IEEE 754 double precision offers 53 bits of precision
- In `crlibm`, we need an accuracy of "120 correct bits"
- In Taylor models, no use of high order polynomials if the remainder grows too fast
- First approach:
  - Use an integer based fixed high precision floating-point library
  - Necessity to leave the floating-point pipeline
  - High impact on performance (factor 100)
- Second approach:
  - Emulate higher precision in floating-point
  - Reusage of already computed floating-point values possible
  - No conversions, fill completely floating-point pipeline
  - Speed-up by at least a factor 10 w.r.t. the first approach
  - Same quality of certification possible

Higher precision in floating-point

- Floating-point expansions:

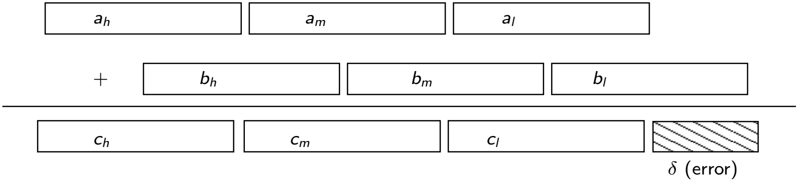


- Operations on expansions: for example addition:

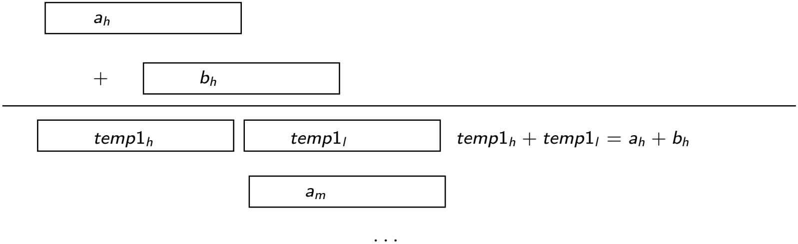


Need for exact floating-point arithmetic

- We want to implement:



- Single step:



Exact floating-point arithmetic

Motivation

Exact floating-point arithmetic

Double-double, triple-double and expansion arithmetic

Exact floating-point arithmetic?

- Floating-point arithmetic can produce round-off error

  $$a \otimes b = a \cdot b \cdot (1 + \varepsilon)$$

  where $|\varepsilon| \leq 2^{-p}$

Exact floating-point arithmetic?

- Floating-point arithmetic can produce round-off error

$$a \otimes b = a \cdot b \cdot (1 + \varepsilon)$$

  where $|\varepsilon| \leq 2^{-p}$

- A floating-point operation is called exact if its result is the mathematical one

$$a \otimes b = a \cdot b$$

$$\varepsilon = 0$$

Exact floating-point arithmetic?

- Floating-point arithmetic can produce round-off error

  $$a \otimes b = a \cdot b \cdot (1 + \varepsilon)$$

  where $|\varepsilon| \leq 2^{-p}$
- A floating-point operation is called exact if its result is the mathematical one

  $$a \otimes b = a \cdot b$$

  $$\varepsilon = 0$$

- However: floating-point arithmetic is often exact:
  - Floating-point numbers are scaled integers
  - If no integer overflow occurs, operations are exact on integers
  - Just factorize the scale (where possible)

    $$a \otimes b = 2^{E_a} \cdot m_a \otimes 2^{E_b} \cdot m_b = 2^{E_a + E_b} \cdot \circ \left(m_a \cdot m_b \right)$$

    where $\circ$ is the rounding operator satisfying

    $$\forall x \in \mathbb{F} \ . \ \circ \left(x \right) = x$$

Disclaimer

If tomorrow, you want to implement what I am going to show in the next slides, remember that...

- Code here is in `C` and that `Fortran` behaves differently
  - Implicit parentheses are elsewhere but our exact FP arithmetic requires the indicated operation order
  - Typing of mixed precision expressions is different
  - "Optimizations" the compiler is allowed to do are different
- Declaring variables as `double x,y,z;` does not imply usage of IEEE 754 double precision on most systems
- Round-to-nearest rounding mode required by some exact arithmetic sequences, in particular for exact multiplication
- Special care is needed for subnormals, underflow and overflow

Sterbenz' lemma

Let be $a, b \in \mathbb{F}$ such that

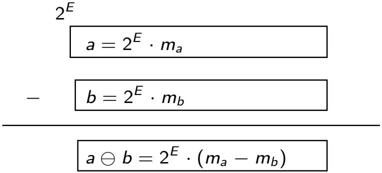
$$sgn(a) = sgn(b)$$

and

$$\frac{1}{2} \cdot |a| \leq |b| \leq 2 \cdot |a|$$

Thus

$$a \ominus b = a - b$$



2^E

$\boxed{a = 2^E \cdot m_a}$

$- \quad \boxed{b = 2^E \cdot m_b}$

$\boxed{a \ominus b = 2^E \cdot (m_a - m_b)}$

Sterbenz' lemma

Let be $a, b \in \mathbb{F}$ such that

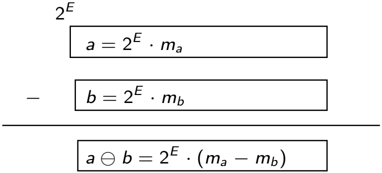
$$sgn(a) = sgn(b)$$

and

$$\frac{1}{2} \cdot |a| \leq |b| \leq 2 \cdot |a|$$

Thus

$$a \ominus b = a - b$$



2^E

$$\boxed{a = 2^E \cdot m_a}$$

$$- \quad \boxed{b = 2^E \cdot m_b}$$

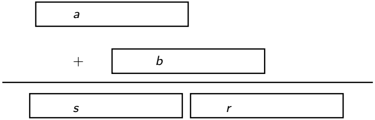
$$\boxed{a \ominus b = 2^E \cdot (m_a - m_b)}$$

- At the base of most extended precision addition algorithms
- Independent of the rounding mode
- Proof intuition: factor the scale of both scaled integers that are a and b

Fast2Sum

Let round-to-nearest the current rounding mode in IEEE 754.
Let $a, b \in \mathbb{F}$ such that $|a| \geq |b|$.
Let be $s, r \in \mathbb{F}$ computed by

```
s = a + b;
t = s - a;
r = b - t;
```



Thus

$$s + r = a + b$$

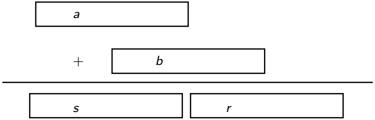
and

$$|r| \leq \mathrm{ulp}(s)$$

Fast2Sum

Let round-to-nearest the current rounding mode in IEEE 754.
Let $a, b \in \mathbb{F}$ such that $|a| \geq |b|$.
Let be $s, r \in \mathbb{F}$ computed by

```
s = a + b;
t = s - a;
r = b - t;
```



Thus

$$s + r = a + b$$

and

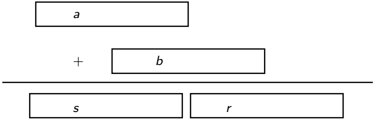
$$|r| \leq \mathrm{ulp}(s)$$

Proof intuition: apply Sterbenz' lemma

Fast2Sum

Let round-to-nearest the current rounding mode in IEEE 754.
Let $a, b \in \mathbb{F}$ such that $|a| \geq |b|$.
Let be $s, r \in \mathbb{F}$ computed by

```
s = a + b;
t = s - a;
r = b - t;
```



Thus

$$s + r = a + b$$

and

$$|r| \leq \mathrm{ulp}(s)$$

Proof intuition: apply Sterbenz' lemma
Meaning of s and r: s is a approximate sum, r the absolute error

2Sum

Let round-to-nearest the current rounding mode in IEEE 754.

Let $a, b \in \mathbb{F}$.

Let be $s, r \in \mathbb{F}$ computed by

```
s = a + b;
if (fabs(a) >= fabs(b)) {
  t = s - a;
  r = b - t;
} else {
  t = s - b;
  r = a - t;
}
```

Thus

$$s + r = a + b$$

and

$$|r| \leq \mathrm{ulp}(s)$$

Branches ?

There are branches!

Branches are expensive on current pipelined processors!

2Sum - avoiding branches

Let round-to-nearest the current rounding mode in IEEE 754.

Let $a, b \in \mathbb{F}$.

Let be $s, r \in \mathbb{F}$ computed by

```
s = a + b;
t1 = s - a;
t2 = s - b;
d1 = b - t1;
d2 = a - t2;
r = d1 + d2;
```

Thus

$$s + r = a + b$$

and

$$|r| \leq \mathrm{ulp}(s)$$

2Sum - avoiding branches

Let round-to-nearest the current rounding mode in IEEE 754.
Let $a, b \in \mathbb{F}$.
Let be $s, r \in \mathbb{F}$ computed by

```
s = a + b;
t1 = s - a;
t2 = s - b;
d1 = b - t1;
d2 = a - t2;
r = d1 + d2;
```

Thus

$$s + r = a + b$$

and

$$|r| \leq \mathrm{ulp}(s)$$

⇒ Up to 10% performance gain w.r.t. branching version !

Round-to-nearest ?

Round-to-nearest mode required?

I am doing interval arithmetic and I do not like to change the rounding-mode !

Fast2Sum - any rounding mode

Let $a, b \in \mathbb{F}$ such that $|a| \geq |b|$.
Let be $s, r \in \mathbb{F}$ computed by

```
s = a + b;
e = s - a;
g = s - e;
h = g - a;
f = b - h;
r = f - e;
if (r + e != f) {
   s = a;
   r = b;
}
```

Thus

$$s + r = a + b$$

and

$$|r| \leq \mathrm{ulp}(s)$$

Multiplication - Introduction

- Addition: $s + r = a + b$

Multiplication - Introduction

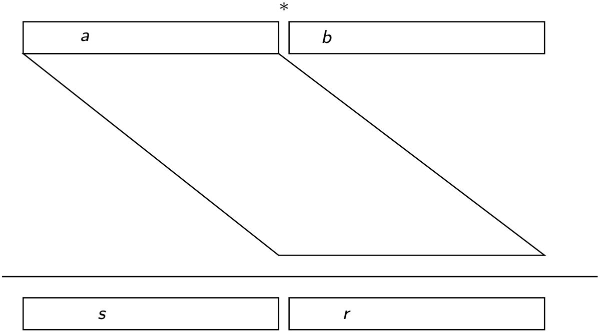
- Addition: $s + r = a + b$
- Multiplication - similarly: $s + r = a \cdot b$

Multiplication - Introduction

- Addition: $s + r = a + b$
- Multiplication - similarly: $s + r = a \cdot b$
- Is this possible ?

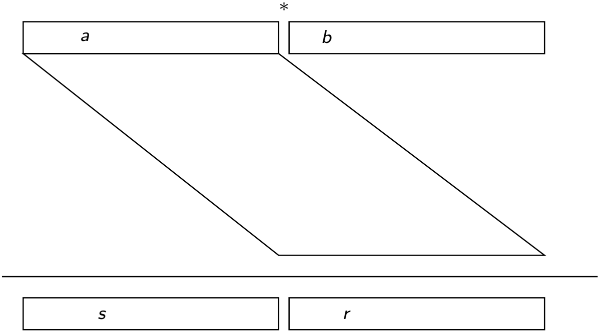
Multiplication - Introduction

- Addition: $s + r = a + b$
- Multiplication - similarly: $s + r = a \cdot b$
- Is this possible ?



Multiplication - Introduction

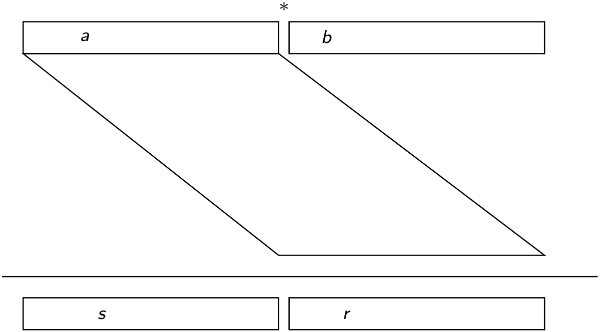
- Addition: $s + r = a + b$
- Multiplication - similarly: $s + r = a \cdot b$
- Is this possible ?



- The significand of $a \cdot b$ holds on a sum of two FP-numbers $s + r$

Multiplication - Introduction

- Addition: $s + r = a + b$
- Multiplication - similarly: $s + r = a \cdot b$
- Is this possible ?



- The significand of $a \cdot b$ holds on a sum of two FP-numbers $s + r$
- How do we compute s and r?

Multiplication - The easy way

Suppose that the system supports a fused-multiply-and-add (FMA) operation: $\texttt{FMA}(a, b, c) = \circ\left(a \cdot b + c\right)$.

Let be $a, b \in \mathbb{F}$.

Let be $s, r \in \mathbb{F}$ computed by

```
s = a * b;
r = FMA(a,b,−s);  // r = ∘(a·b − s)
```

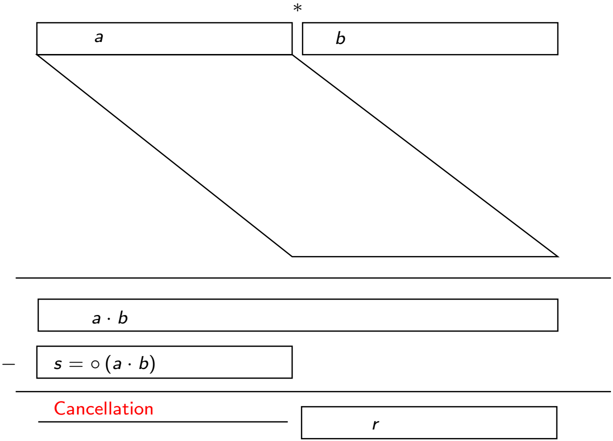
Thus

$$s + r = a \cdot b$$

and

$$|r| \leq \mathrm{ulp}(s)$$

Multiplication - Graphical “proof”



Multiplication - without FMA

- Let be $a, b \in \mathbb{F}_p$ on p bits
- We want $s + r = a \cdot b$

Multiplication - without FMA

- Let be $a, b \in \mathbb{F}_p$ on p bits
- We want $s + r = a \cdot b$
- Let be $a_h + a_l = a$ and $b_h + b_l = b$
- Clearly $a \cdot b = a_h \cdot b_h + a_h \cdot b_l + a_l \cdot b_h + a_l \cdot b_l$

Multiplication - without FMA

- Let be $a, b \in \mathbb{F}_p$ on p bits
- We want $s + r = a \cdot b$
- Let be $a_h + a_l = a$ and $b_h + b_l = b$
- Clearly $a \cdot b = a_h \cdot b_h + a_h \cdot b_l + a_l \cdot b_h + a_l \cdot b_l$
- If a_h, a_l, b_h, b_l are written on at most p' bits, all products hold on $2 \cdot p'$ bits.

Multiplication - without FMA

- Let be $a, b \in \mathbb{F}_p$ on p bits
- We want $s + r = a \cdot b$
- Let be $a_h + a_l = a$ and $b_h + b_l = b$
- Clearly $a \cdot b = a_h \cdot b_h + a_h \cdot b_l + a_l \cdot b_h + a_l \cdot b_l$
- If a_h, a_l, b_h, b_l are written on at most p' bits, all products hold on $2 \cdot p'$ bits.
- If $2 \cdot p' \leq p$ we can write:

$$a \cdot b = a_h \otimes b_h + a_h \otimes b_l + a_l \otimes b_h + a_l \otimes b_l$$

Multiplication - without FMA

- Let be $a, b \in \mathbb{F}_p$ on p bits
- We want $s + r = a \cdot b$
- Let be $a_h + a_l = a$ and $b_h + b_l = b$
- Clearly $a \cdot b = a_h \cdot b_h + a_h \cdot b_l + a_l \cdot b_h + a_l \cdot b_l$
- If a_h, a_l, b_h, b_l are written on at most p' bits, all products hold on $2 \cdot p'$ bits.
- If $2 \cdot p' \leq p$ we can write:

$$a \cdot b = a_h \otimes b_h + a_h \otimes b_l + a_l \otimes b_h + a_l \otimes b_l$$

- Since $a \cdot b$ holds on at most $2 \cdot p$ bits, there will be sufficient cancellation in the summation of the products producing $s + r$ $\Rightarrow$ Use here the exact 2Sum presented before.

Multiplication - without FMA

- Let be $a, b \in \mathbb{F}_p$ on p bits
- We want $s + r = a \cdot b$
- Let be $a_h + a_l = a$ and $b_h + b_l = b$
- Clearly $a \cdot b = a_h \cdot b_h + a_h \cdot b_l + a_l \cdot b_h + a_l \cdot b_l$
- If a_h, a_l, b_h, b_l are written on at most p' bits, all products hold on $2 \cdot p'$ bits.
- If $2 \cdot p' \leq p$ we can write:

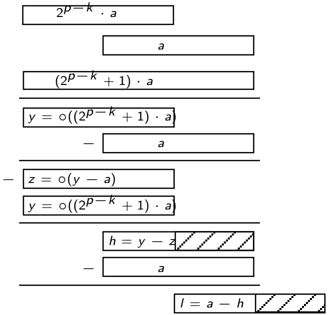
$$a \cdot b = a_h \otimes b_h + a_h \otimes b_l + a_l \otimes b_h + a_l \otimes b_l$$

- Since $a \cdot b$ holds on at most $2 \cdot p$ bits, there will be sufficient cancellation in the summation of the products producing $s + r$ $\Rightarrow$ Use here the exact 2Sum presented before.
- How can we compute $a_h + a_l = a$?

Cut into halves

Let round-to-nearest the current rounding mode in IEEE 754.
Let $a \in \mathbb{F}_p$ with precision p.
Let be $h, l \in \mathbb{F}_p$ computed by

```
c = $2^{p-k} + 1$ ;
y = c * a ;
z = y - a ;
h = y - z ;
l = a - h ;
```



Thus

$$h + l = a$$

and h has at least k trailing zeros and l has at least $p - k + 1$ trailing zeros.

Other exact operations

- Division and square root
- One can express only the backward error

  $$s = f(a - \delta)$$

  instead of

  $$s + \delta = f(a)$$

  as for addition and multiplication
- Division:

  $$d = \frac{a - r}{b}$$

  where $d = a \oslash b \in \mathbb{F}$ and $r \in F$
- Square root:

  $$s = \sqrt{a - r}$$

  where $s = \circ\left(\sqrt{a}\right)$ and $r \in \mathbb{F}$
- We can implement division and square root on expansions even with backward errors

Double-double, triple-double and expansion arithmetic

Motivation

Exact floating-point arithmetic

Double-double, triple-double and expansion arithmetic

Vocabulary

- Represent high precision numbers as unevaluated sums of floating-point numbers

$$x = \sum_{i=1}^{n} x_i$$

- Suppose native precision to be IEEE 754 double precision
  - $n = 2$: "double-double" – ≈ 102 bits of accuracy
  - $n = 3$: "triple-double" – ≈ 150 bits of accuracy
  - $n = 4$: Bailey: "quad-double"
  - any n: expansions

Operations on expansions

Operations on expansions:

- Addition – Use 2Sum algorithm for carries
- Multiplication – Partial products using 2Mult, sum up using 2Sum
- Division – Euclid's division using an exact backward error sequence or Newton's method
- Square root – Newton's method
- Renormalization – use 2Sums and tests for bringing expansions to a non-overlapping form

Operations on expansions

Operations on expansions:

- Addition – Use 2Sum algorithm for carries
- Multiplication – Partial products using 2Mult, sum up using 2Sum
- Division – Euclid's division using an exact backward error sequence or Newton's method
- Square root – Newton's method
- Renormalization – use 2Sums and tests for bringing expansions to a non-overlapping form

Cost:

- No conversions between floating-point and integer ⇒ double-double and triple-double is much faster
- Expansions are inefficient: the exponents are redundant information
- Floating-point arithmetic has some bizarre behaviours: ⇒ general expansions seem to be more expensive than integer based methods because of a high number of tests

Double-double and triple-double in `crlibm`

- Full implementation of double-double
    - Versions for 2Sum and 2Mult optimized for different processors (FMA, FABS, . . .)
    - All combinations double + double, double-double + double etc.
    - Accuracy proof for each operator; proof can already be formally verified (Gappa)

Double-double and triple-double in `crlibm`

- Full implementation of double-double
  - Versions for 2Sum and 2Mult optimized for different processors (FMA, FABS, ...)
  - All combinations double + double, double-double + double etc.
  - Accuracy proof for each operator; proof can already be formally verified (Gappa)
- Almost complete implementation of triple-double
  - Based on double-double
  - Almost all combinations double, double-double or triple-double in operand or result
  - Accuracy proof of each operator
  - Approach for avoiding renormalizations whilst being rigurous
  - No branches on common machines
  - Correct (IEEE 754) rounding to double implemented

Double-double and triple-double in `crlibm`

- Full implementation of double-double
  - Versions for 2Sum and 2Mult optimized for different processors (FMA, FABS, ...)
  - All combinations double + double, double-double + double etc.
  - Accuracy proof for each operator; proof can already be formally verified (Gappa)
- Almost complete implementation of triple-double
  - Based on double-double
  - Almost all combinations double, double-double or triple-double in operand or result
  - Accuracy proof of each operator
  - Approach for avoiding renormalizations whilst being rigurous
  - No branches on common machines
  - Correct (IEEE 754) rounding to double implemented
- Automatic routines for generating double, double-double and triple-double code for evaluating complete polynomials in Horner's scheme with formal proof generation

Speed-ups

Logarithm - evaluate polynomials of degree about $12 - 20$

Library	cycles
`MPFR` - integer based multiprec.	12942
`crlibm` portable using integer based multiprec.	2748
`crlibm` portable using triple-double	266

Exponential - evaluate polynomials of degree about $7 - 15$

Library	cycles
`MPFR` - integer based multiprec.	4908
`crlibm` portable using integer based multiprec.	1976
`crlibm` portable using triple-double	258

Conclusion

- Presentation of exact floating-point arithmetic
- Overview over general techniques for expansions
- Double-double and triple-double are quite efficient
  - No branches needed
  - No conversions needed
  - Speed-up of a factor of about 10
- Rigourous proofs are possible (Gappa)
- General expansion algorithms known but rarely implemented