Third International Workshop on Numerical Software Verification

Formal verification of numerical programs: from C annotated programs to Coq proofs

Sylvie Boldo

INRIA Saclay - Île-de-France

July 15th, 2010

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

INRIA

centre de recherche SACLAY - ÎLE-DE-FRANCE

Thanks to

- the organizers !

Thanks to

- the organizers !
- all collaborators of these works
  - F. Clément
  - J.-C. Filliâtre
  - G. Melquiond
  - T. Nguyen

Motivations

- Numerical Software Verification

Motivations

- Numerical Software Verification

$\Rightarrow$ software with floating-point computations

Floating-point number

This is only a string of bits.

11100011010010011110000111000000

Floating-point number

This is only a string of bits.

11100011010010011110000111000000

We interpret it depending on the respective values of s (sign), e (exponent) and f (fraction).

1 11000110 10010011110000111000000

1	11000110	10010011110000111000000
s	e	f

Floating-point number

We associate a real value :

$$\boxed{1} \quad \boxed{11000110} \quad \boxed{10010011110000111000000}$$

$$\begin{array}{ccc} s & e & f \\ \downarrow & \downarrow & \downarrow \\ (-1)^s \times & 2^{e-B} \quad \times & 1 \bullet f \end{array}$$

$$(-1)^1 \times \; 2^{198-127} \; \times 1.10010011110000111000000_2$$

$$-2^{54} \times 206727 \approx -3.724 \times 10^{21}$$

Floating-point number

We associate a real value :

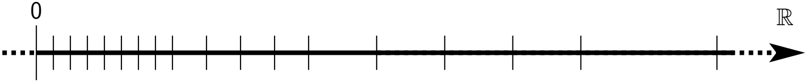
$$\begin{array}{cccc} \boxed{1} & \boxed{11000110} & & \boxed{10010011110000111000000} \\ s & e & & f \\ \downarrow & \downarrow & & \downarrow \\ (-1)^s \times & 2^{e-B} & \times & 1 \bullet f \end{array}$$

$$(-1)^1 \times \; 2^{198-127} \; \times 1.10010011110000111000000_2$$

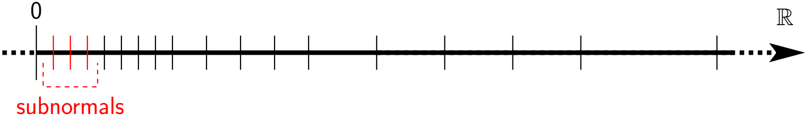
$$-2^{54} \times 206727 \approx -3.724 \times 10^{21}$$

except for the special values of e : ± 0, $\pm\infty$, NaN, subnormals.

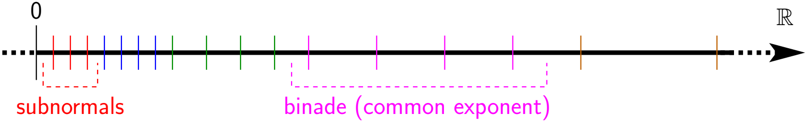
Floating-point number repartition



Floating-point number repartition



Floating-point number repartition



Floating-point operations

Thanks to the IEEE-754 standard, the computed results of $+, -, \times, /, \sqrt{}$ should be the same as if they were first computed with infinite precision and then rounded.

$\Rightarrow$ computations with 3 more bits (see J. Coonen)

Floating-point operations

Thanks to the IEEE-754 standard, the computed results of $+, -, \times, /, \sqrt{}$ should be the same as if they were first computed with infinite precision and then rounded.

$\Rightarrow$ computations with 3 more bits (see J. Coonen)

$\Rightarrow$ mathematical properties such that :
when a real value fits exactly in a floating-point number in a given format, then it is exactly computed.

Motivations

- Numerical Software Verification

Motivations

- Numerical Software Verification

- Critical C code $\hookrightarrow$ formal proof

  $\Rightarrow$ high guarantee

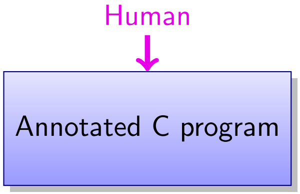
Related work

- static analyzers
  - Astrée
  - Fluctuat
- specification languages
  - JML
- formal proofs about floating-point arithmetic
  - trigonometric functions (HOL Light)
  - verification of the FPU (ACL2)

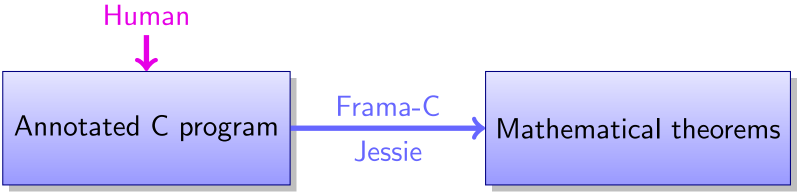
Motivations

C program

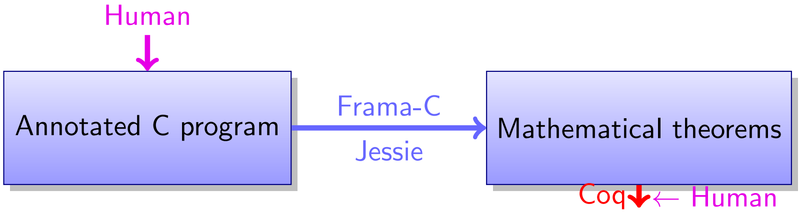
Motivations



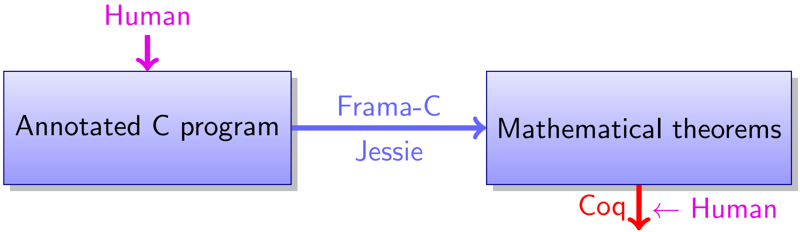
Motivations



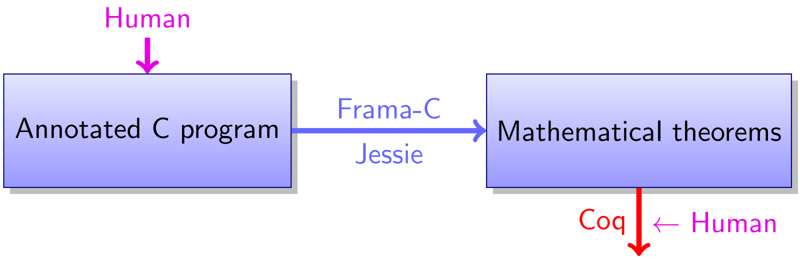
Motivations



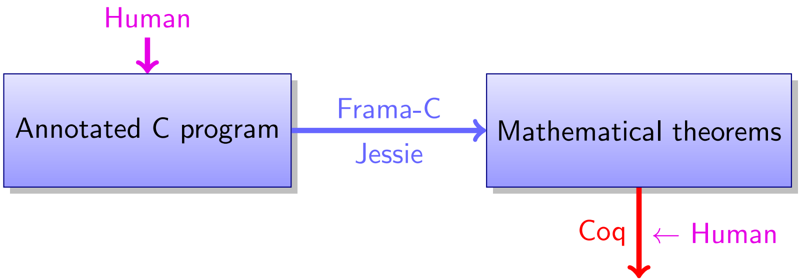
Motivations



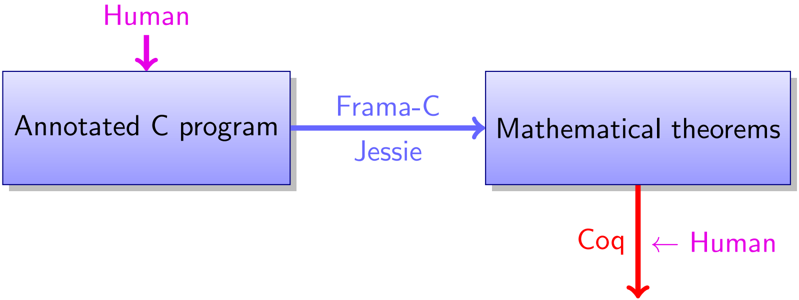
Motivations



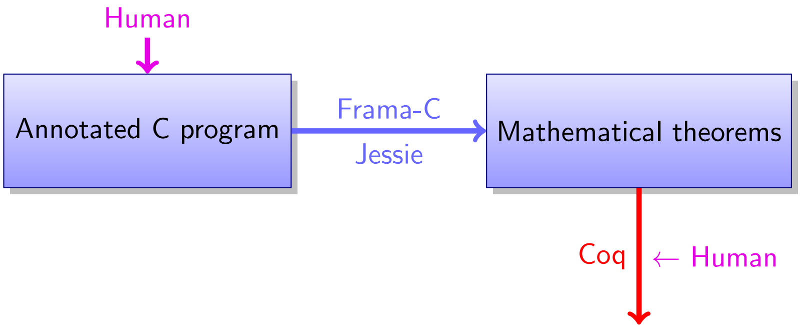
Motivations



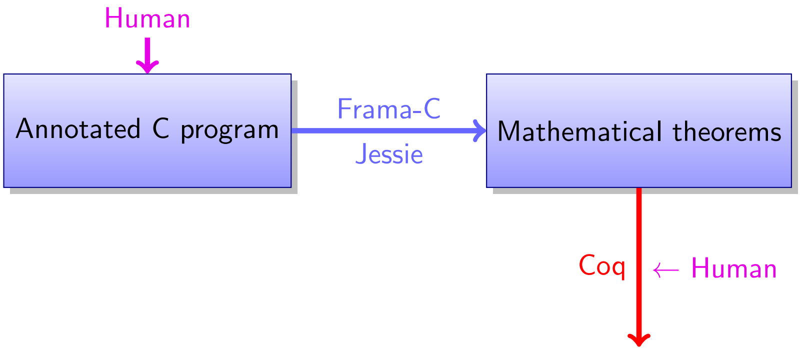
Motivations



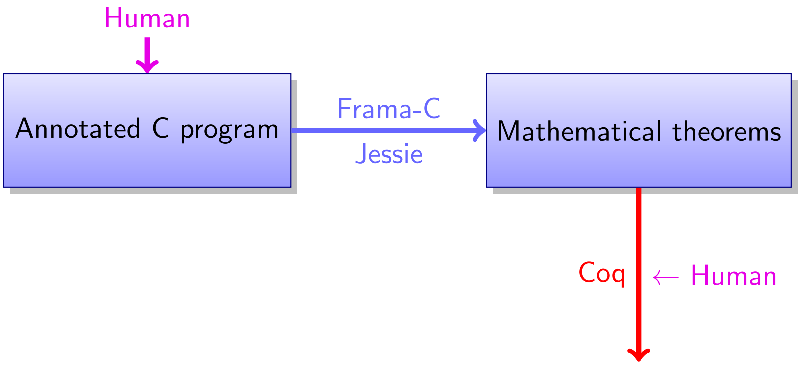
Motivations



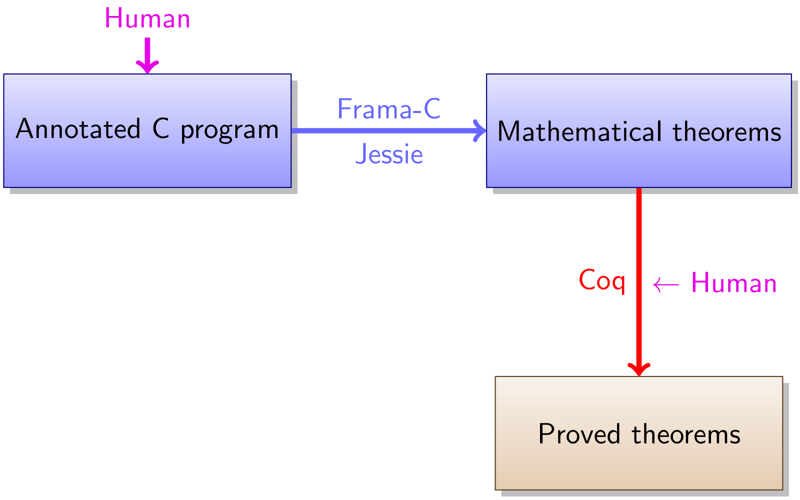
Motivations



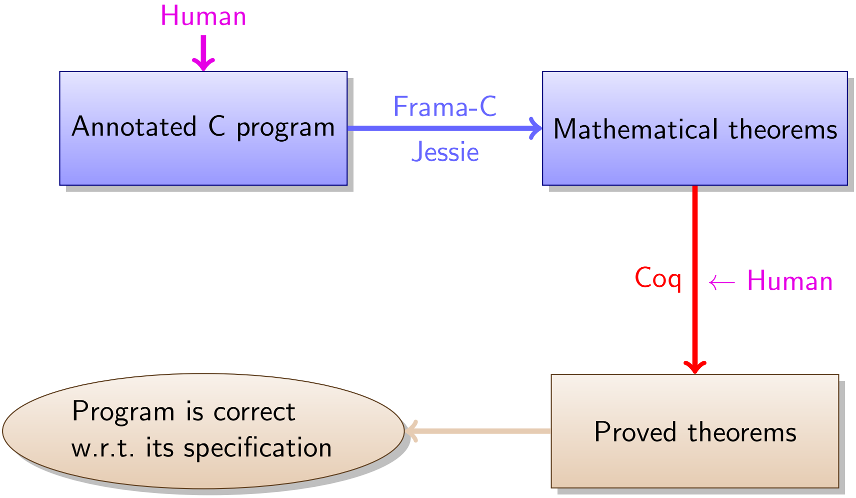
Motivations



Motivations



Motivations



Plan

1 Motivations

2 Tools
- Formal proof
- Frama-C/Jessie/Why

3 Examples

4 Conclusions

Formal proof

Certified formal proof

The proof is checked in its deep details until the computer agrees with it.

We often use formal proof checkers, meaning programs that only **check** a proof (they may also generate easy demonstrations).

Therefore the checker is a very short program (de Bruijn criteria : the correctness of the system as a whole depends on the correctness of a very small "kernel").

The Coq proof assistant (`http://coq.inria.fr`)

- Based on the Curry-Howard isomorphism.
  (equivalence between proofs and λ-terms)
- Few automations.
- Comprehensive libraries, including on $\mathbb{Z}$ and $\mathbb{R}$.
- Coq kernel mechanically checks each step of each proof.
- The method is to apply successively tactics (theorem application, rewriting, simplifications...) to transform or reduce the goal down to the hypotheses.
- The proof is handled starting from the conclusion.

Coq formalization (by L. Théry)

Float = pair of signed integers (mantissa, exponent)

$$(n, e) \in \mathbb{Z}^2$$

Coq formalization (by L. Théry)

Float = pair of signed integers (mantissa, exponent)
associated to a real value.

$$(n, e) \in \mathbb{Z}^2 \hookrightarrow n \times \beta^e \in \mathbb{R}$$

Coq formalization (by L. Théry)

Float = pair of signed integers (mantissa, exponent)
associated to a real value.

$$(n, e) \in \mathbb{Z}^2 \;\hookrightarrow\; n \times \beta^e \in \mathbb{R}$$

1.00010_2 E 4	$\mapsto$	$(100010_2, -1)_2$	$\hookrightarrow$	17
IEEE-754		significant of 754R		real value

$\Rightarrow$ normal floats, subnormal floats, ~~overflow~~.

Many floats may represent the same real value, but we can exhibit a canonical representation.

Example using Coq 8.2

```
Theorem Rle_Fexp_eq_Zle :
 forall x y :float, (x <= y)%R ->
   Fexp x = Fexp y -> (Fnum x <= Fnum y)%Z.
intros x y H' H'0.
apply le_IZR.
apply (Rle_monotony_contra_exp radix)
   with (z := Fexp x); auto with real arith.
pattern (Fexp x) at 2 in |- *; rewrite H'0; auto.
Qed.
```

With keywords, stating of the theorem, tactics and names of used theorems.

Example using Coq 8.2

```
Theorem Rle_Fexp_eq_Zle :
 forall x y :float, (x <= y)%R ->
   Fexp x = Fexp y -> (Fnum x <= Fnum y)%Z.
intros x y H' H'0.
apply le_IZR.
apply (Rle_monotony_contra_exp radix)
   with (z := Fexp x); auto with real arith.
pattern (Fexp x) at 2 in |- *; rewrite H'0; auto.
Qed.
```

With keywords, stating of the theorem, tactics and names of used theorems.

Theorem (Rle_Fexp_eq_Zle)

If two floats $x = (n_x, e_x)$ and $y = (n_y, e_y)$ verifies $x \leq y$, and $e_x = e_y$, then $n_x \leq n_y$.

Plan

1. Motivations
2. Tools
   - Formal proof
   - Frama-C/Jessie/Why
3. Examples
4. Conclusions

Frama-C/Jessie/Why

- Frama-C is a framework dedicated to the analysis of the source code of software written in C.

Frama-C/Jessie/Why

- Frama-C is a framework dedicated to the analysis of the source code of software written in C.
- Available plugins :

Frama-C/Jessie/Why

- Frama-C is a framework dedicated to the analysis of the source code of software written in C.
- Available plugins :
  - value analysis

Frama-C/Jessie/Why

- Frama-C is a framework dedicated to the analysis of the source code of software written in C.
- Available plugins :
  - value analysis
  - Jessie, the deductive verification plug-in
    (based on weakest precondition computation techniques)

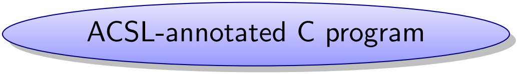
Frama-C/Jessie/Why

- Frama-C is a framework dedicated to the analysis of the source code of software written in C.
- Available plugins :
  - value analysis
  - Jessie, the deductive verification plug-in
    (based on weakest precondition computation techniques)
  - ...

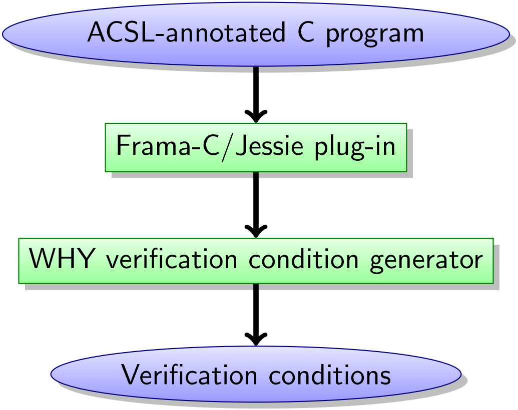
Frama-C/Jessie/Why

- Frama-C is a framework dedicated to the analysis of the source code of software written in C.
- Available plugins :
  - value analysis
  - Jessie, the deductive verification plug-in (based on weakest precondition computation techniques)
  - ...
- Free softwares in CAML available at `http://frama-c.com/` and `http://why.lri.fr/`.

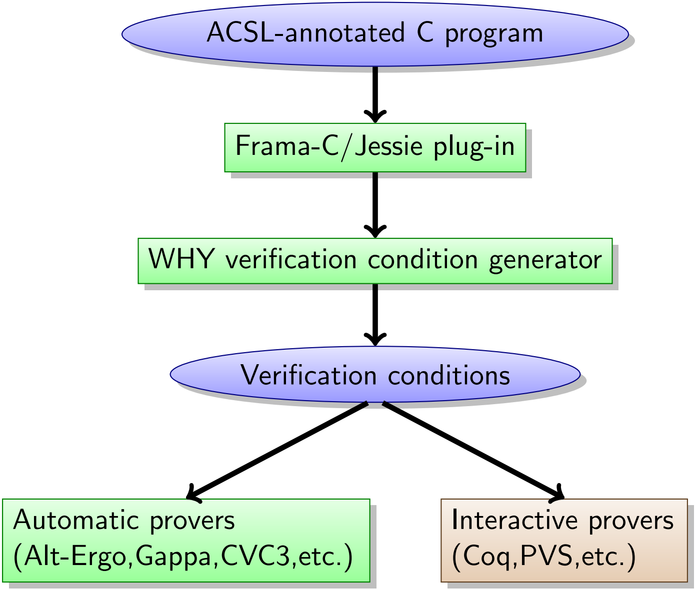
Frama-C/Jessie/Why



Frama-C/Jessie/Why



Frama-C/Jessie/Why



ACSL

- ANSI/ISO C Specification Language

ACSL

- ANSI/ISO C Specification Language
- behavioral specification language for C programs

ACSL

- ANSI/ISO C Specification Language
- behavioral specification language for C programs
- pre-conditions and post-conditions to functions
  (and which variables are modified).

ACSL

- ANSI/ISO C Specification Language
- behavioral specification language for C programs
- pre-conditions and post-conditions to functions
  (and which variables are modified).
- variants and invariants of the loops.

ACSL

- ANSI/ISO C Specification Language
- behavioral specification language for C programs
- pre-conditions and post-conditions to functions
  (and which variables are modified).
- variants and invariants of the loops.
- assertions

ACSL

- ANSI/ISO C Specification Language
- behavioral specification language for C programs
- pre-conditions and post-conditions to functions (and which variables are modified).
- variants and invariants of the loops.
- assertions
- In annotations, all computations are exact.

ACSL and floating-point numbers

A floating-point number is a triple :

- the floating-point number, really computed by the program,
  $x \to x_f$ floating-point part

ACSL and floating-point numbers

A floating-point number is a triple :

- the floating-point number, really computed by the program,
  $x \to x_f$ floating-point part
- the value that would have been obtained with exact computations,
  $x \to x_e$ exact part

ACSL and floating-point numbers

A floating-point number is a triple :

- the floating-point number, really computed by the program,
  $x \to x_f$ floating-point part
- the value that would have been obtained with exact computations,
  $x \to x_e$ exact part
- the value that we ideally wanted to compute
  $x \to x_m$ model part

ACSL and floating-point numbers

A floating-point number is a triple :

- the floating-point number, really computed by the program,
  $x \to x_f$ floating-point part `1+x+x*x/2`
- the value that would have been obtained with exact computations,
  $x \to x_e$ exact part $1 + x + \frac{x^2}{2}$
- the value that we ideally wanted to compute
  $x \to x_m$ model part $\exp(x)$

ACSL and floating-point numbers

A floating-point number is a triple :

- the floating-point number, really computed by the program,
  $x \to x_f$ floating-point part `1+x+x*x/2`
- the value that would have been obtained with exact computations,
  $x \to x_e$ exact part $1 + x + \frac{x^2}{2}$
- the value that we ideally wanted to compute
  $x \to x_m$ model part $\exp(x)$

$\Rightarrow$ easy to split into method error and rounding error

ACSL and floating-point numbers

A floating-point number is a triple :

- the floating-point number, really computed by the program,
  $x \to x_f$ floating-point part `1+x+x*x/2`
- the value that would have been obtained with exact computations,
  $x \to x_e$ exact part $1 + x + \frac{x^2}{2}$
- the value that we ideally wanted to compute
  $x \to x_m$ model part $\exp(x)$

$\Rightarrow$ easy to split into method error and rounding error

For a float `f`, we have macros such as `\rounding_error(f)` and `\exact(f)`, while `f` (as a real) is its floating-point value.

Pragmas

Several pragmas corresponding to different formalization for floating-point numbers.

- `defensive` (default pragma) : IEEE roundings occur. We prove that no exceptional behavior may happen (Overflow, NaN creation...)

Pragmas

Several pragmas corresponding to different formalization for floating-point numbers.

- `defensive` (default pragma) : IEEE roundings occur. We prove that no exceptional behavior may happen (Overflow, NaN creation...)
- `math` : all computations are exact.

Pragmas

Several pragmas corresponding to different formalization for floating-point numbers.

- `defensive` (default pragma) : IEEE roundings occur. We prove that no exceptional behavior may happen (Overflow, NaN creation...)
- `math` : all computations are exact.
- `full` : IEEE roundings occur. Exceptional behaviors may happen.

Pragmas

Several pragmas corresponding to different formalization for floating-point numbers.

- `defensive` (default pragma) : IEEE roundings occur. We prove that no exceptional behavior may happen (Overflow, NaN creation...)
- `math` : all computations are exact.
- `full` : IEEE roundings occur. Exceptional behaviors may happen.
- `multi-rounding` : we may have any hardware and compiler (80-bit extended registers, FMA)

Plan

1 Motivations

2 Tools
- Formal proof
- Frama-C/Jessie/Why

3 Examples

4 Conclusions

Examples

- All examples use Frama-C Boron and Why 2.26.

Examples

- All examples use Frama-C Boron and Why 2.26.
- All proof obligations are proved using Coq.
  (except 2 inequalities in the last example).

Examples

- All examples use Frama-C Boron and Why 2.26.
- All proof obligations are proved using Coq.
  (except 2 inequalities in the last example).
- Code & proofs available on
  `http://www.lri.fr/~sboldo/research.html`.

Sterbenz

Theorem (Sterbenz)

If x and y are FP numbers in a given precision such that

$$\frac{y}{2} \leq x \leq 2y,$$

then $x - y$ fits in a FP number in the same precision and is therefore computed without error.

Sterbenz – program

```
/*@ requires  y/2. <= x <= 2.*y;
  @ ensures   \result == x-y;
  @*/

float Sterbenz(float x, float y) {
  return x-y;
}
```

Sterbenz – program

Exact subtraction

```
/*@ requires  y/2. <= x <= 2.*y;
  @ ensures   \result == x-y;
  @*/

float Sterbenz(float x, float y) {
  return x-y;
}
```

Sterbenz – program

1 PO : exact subtraction

```
/*@ requires  y/2. <= x <= 2.*y;
  @ ensures   \result == x-y;
  @*/

float Sterbenz(float x, float y) {
  return x-y;
}
```

Sterbenz – program

1 PO : exact subtraction

```
/*@ requires  y/2. <= x <= 2.*y;
  @ ensures   \result == x-y;
  @*/

float Sterbenz(float x, float y) {
  return x-y;
}
```

1 PO : no overflow

Veltkamp/Dekker

Theorem (Veltkamp/Dekker)

Provided no Overflow and no Underflow occur, there is an algorithm computing the exact error of the multiplication using only FP operations.

Veltkamp/Dekker

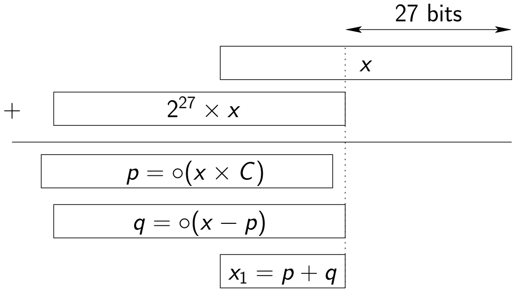
Theorem (Veltkamp/Dekker)

Provided no Overflow and no Underflow occur, there is an algorithm computing the exact error of the multiplication using only FP operations.

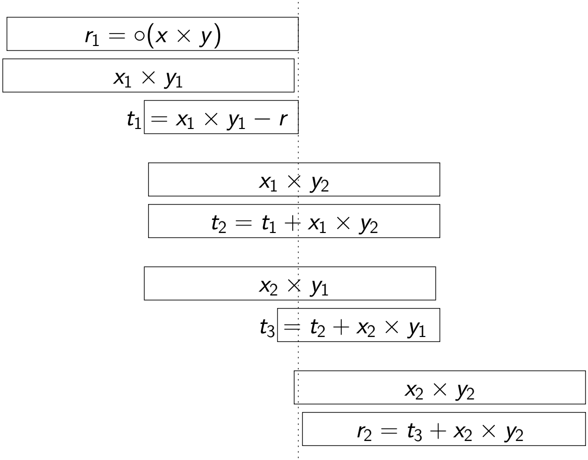
Idea :
split your floats in 2, multiply all the parts, add them in the correct order.

Veltkamp : how to split a floating-point number

Let $C = 2^{27} + 1$ for `double` precision numbers.



Dekker : how to get the error of the multiplication



Veltkamp/Dekker – program

```
/*@ requires xy == \round_double(\NearestEven,x*y) &&
  @          \abs(x) <= 0x1.p995 &&
  @          \abs(y) <= 0x1.p995 &&
  @          \abs(x*y) <=  0x1.p1021;
  @ ensures  ((x*y == 0 || 0x1.p-969 <= \abs(x*y))
  @                 ==> x*y == xy+\result);
  @*/
double Dekker(double x, double y, double xy) {

  double C,px,qx,hx,py,qy,hy,tx,ty,r2;
  int i;
  [...]
  /*@ assert C == \pow(2.,27) + 1. */

  px=x*C; qx=x-px; hx=px+qx; tx=x-hx;

  py=y*C; qy=y-py; hy=py+qy; ty=y-hy;

  r2=-xy+hx*hy;
  r2+=hx*ty;
  r2+=hy*tx;
  r2+=tx*ty;
  return r2;
}
```

Veltkamp/Dekker – program

```
/*@ requires xy == \round_double(\NearestEven,x*y) &&
  @          \abs(x) <= 0x1.p995 &&
  @          \abs(y) <= 0x1.p995 &&
  @          \abs(x*y) <=  0x1.p1021;
  @ ensures  ((x*y == 0 || 0x1.p-969 <= \abs(x*y))
  @                  ==> x*y == xy+\result);
  @*/
double Dekker(double x, double y, double xy) {

  double C,px,qx,hx,py,qy,hy,tx,ty,r2;
  int i;
  [...]
  /*@ assert C == \pow(2.,27) + 1. */

  px=x*C; qx=x-px; hx=px+qx; tx=x-hx;

  py=y*C; qy=y-py; hy=py+qy; ty=y-hy;

  r2=-xy+hx*hy;
  r2+=hx*ty;
  r2+=hy*tx;
  r2+=tx*ty;
  return r2;
}
```

Split x and y

Veltkamp/Dekker – program

```
/*@ requires xy == \round_double(\NearestEven,x*y) &&
  @          \abs(x) <= 0x1.p995 &&
  @          \abs(y) <= 0x1.p995 &&
  @          \abs(x*y) <=  0x1.p1021;
  @ ensures  ((x*y == 0 || 0x1.p-969 <= \abs(x*y))
  @                 ==> x*y == xy+\result);
  @*/
double Dekker(double x, double y, double xy) {

  double C,px,qx,hx,py,qy,hy,tx,ty,r2;
  int i;
  [...]
  /*@ assert C == \pow(2.,27) + 1. */

  px=x*C; qx=x-px; hx=px+qx; tx=x-hx;

  py=y*C; qy=y-py; hy=py+qy; ty=y-hy;

  r2=-xy+hx*hy;
  r2+=hx*ty;
  r2+=hy*tx;
  r2+=tx*ty;
  return r2;
}
```

Multiply all halves and add all the results

Veltkamp/Dekker – program

```
/*@ requires xy == \round_double(\NearestEven,x*y) &&
  @          \abs(x) <= 0x1.p995 &&
  @          \abs(y) <= 0x1.p995 &&
  @          \abs(x*y) <=  0x1.p1021;
  @ ensures  ((x*y == 0 || 0x1.p-969 <= \abs(x*y))
  @                  ==> x*y == xy+\result);
  @*/
double Dekker(double x, double y, double xy) {

  double C,px,qx,hx,py,qy,hy,tx,ty,r2;
  int i;
  [...]
  /*@ assert C == \pow(2.,27) + 1. */

  px=x*C; qx=x-px; hx=px+qx; tx=x-hx;

  py=y*C; qy=y-py; hy=py+qy; ty=y-hy;

  r2=-xy+hx*hy;
  r2+=hx*ty;
  r2+=hy*tx;
  r2+=tx*ty;
  return r2;
}
```

$xy = \circ(xy)$

Veltkamp/Dekker – program

```
/*@ requires xy == \round_double(\NearestEven,x*y) &&
  @           \abs(x) <= 0x1.p995 &&
  @           \abs(y) <= 0x1.p995 &&
  @           \abs(x*y) <=  0x1.p1021;
  @ ensures  ((x*y == 0 || 0x1.p-969 <= \abs(x*y))
  @                 ==> x*y == xy+\result);
  @*/
double Dekker(double x, double y, double xy) {

  double C,px,qx,hx,py,qy,hy,tx,ty,r2;
  int i;
  [...]
  /*@ assert C == \pow(2.,27) + 1. */

  px=x*C; qx=x-px; hx=px+qx; tx=x-hx;

  py=y*C; qy=y-py; hy=py+qy; ty=y-hy;

  r2=-xy+hx*hy;
  r2+=hx*ty;
  r2+=hy*tx;
  r2+=tx*ty;
  return r2;
}
```

Overflow

Veltkamp/Dekker – program

```
/*@ requires xy == \round_double(\NearestEven,x*y) &&
  @          \abs(x) <= 0x1.p995 &&
  @          \abs(y) <= 0x1.p995 &&
  @          \abs(x*y) <=  0x1.p1021;
  @ ensures  ((x*y == 0 || 0x1.p-969 <= \abs(x*y))
  @                ==> x*y == xy+\result);
  @*/
double Dekker(double x, double y, double xy) {

  double C,px,qx,hx,py,qy,hy,tx,ty,r2;
  int i;
  [...]
  /*@ assert C == \pow(2.,27) + 1. */

  px=x*C; qx=x-px; hx=px+qx; tx=x-hx;

  py=y*C; qy=y-py; hy=py+qy; ty=y-hy;

  r2=-xy+hx*hy;
  r2+=hx*ty;
  r2+=hy*tx;
  r2+=tx*ty;
  return r2;
}
```

If no Underflow

Veltkamp/Dekker – program

```
/*@ requires xy == \round_double(\NearestEven,x*y) &&
  @          \abs(x) <= 0x1.p995 &&
  @          \abs(y) <= 0x1.p995 &&
  @          \abs(x*y) <=  0x1.p1021;
  @ ensures  ((x*y == 0 || 0x1.p-969 <= \abs(x*y))
  @                  ==> x*y == xy+\result);
  @*/
double Dekker(double x, double y, double xy) {

  double C,px,qx,hx,py,qy,hy,tx,ty,r2;
  int i;
  [...]
  /*@ assert C == \pow(2.,27) + 1. */

  px=x*C; qx=x-px; hx=px+qx; tx=x-hx;

  py=y*C; qy=y-py; hy=py+qy; ty=y-hy;

  r2=-xy+hx*hy;
  r2+=hx*ty;
  r2+=hy*tx;
  r2+=tx*ty;
  return r2;
}
```

Exact error of $\otimes$

Accurate discriminant

It is pretty hard to compute $b^2 - ac$ accurately.

Accurate discriminant

It is pretty hard to compute $b^2 - ac$ accurately.

Theorem (Kahan)

Provided no Overflow and no Underflow occur, there is an algorithm computing the $b^2 - a * c$ within 2 ulps.

Accurate discriminant – program

```
/*@ requires
  @     (b==0.    || 0x1.p-916 <= \abs(b*b)) &&
  @     (a*c==0.  || 0x1.p-916 <= \abs(a*c)) &&
  @     \abs(b) <= 0x1.p510 &&
  @     \abs(a) <= 0x1.p995 && \abs(c) <= 0x1.p995 &&
  @     \abs(a*c) <= 0x1.p1021;
  @ ensures \result==0.
  @       || \abs(\result-(b*b-a*c)) <= 2.*ulp(\result);
  @ */

double discriminant(double a, double b, double c) {
  double p,q,d,dp,dq;
  p=b*b;
  q=a*c;

  if (p+q <= 3*fabs(p-q))
    d=p-q;
  else {
    dp=Dekker(b,b,p);
    dq=Dekker(a,c,q);
    d=(p-q)+(dp-dq);
  }
  return d;
}
```

Accurate discriminant – program

```
/*@ requires
  @     (b==0.   || 0x1.p-916 <= \abs(b*b)) &&
  @     (a*c==0. || 0x1.p-916 <= \abs(a*c)) &&
  @     \abs(b) <= 0x1.p510 &&
  @     \abs(a) <= 0x1.p995 && \abs(c) <= 0x1.p995 &&
  @     \abs(a*c) <= 0x1.p1021;
  @ ensures \result==0.
  @       || \abs(\result-(b*b-a*c)) <= 2.*ulp(\result);
  @ */

double discriminant(double a, double b, double c) {
  dou
  p=b*b;
  q=a*c;

  if (p+q <= 3*fabs(p-q))
    d=p-q;
  else {
    dp=Dekker(b,b,p);
    dq=Dekker(a,c,q);
    d=(p-q)+(dp-dq);
  }
  return d;
}
```

Test whether $ac \approx b^2$

Accurate discriminant – program

```
/*@ requires
  @      (b==0.   || 0x1.p-916 <= \abs(b*b)) &&
  @      (a*c==0. || 0x1.p-916 <= \abs(a*c)) &&
  @      \abs(b) <= 0x1.p510 &&
  @      \abs(a) <= 0x1.p995 && \abs(c) <= 0x1.p995 &&
  @      \abs(a*c) <= 0x1.p1021;
  @ ensures \result==0.
  @       || \abs(\result-(b*b-a*c)) <= 2.*ulp(\result);
  @ */

double discriminant(double a, double b, double c) {
  dou
  p=b*b;
  q=a*c;

  if (p+q <= 3*fabs(p-q))
    d=p-q;
  else {
    dp=Dekker(b,b,p);
    dq=Dekker(a,c,q);
    d=(p-q)+(dp-dq);
  }
  return d;
}
```

Test whether $ac \approx b^2$

If $ac \not\approx b^2$, compute naively

Accurate discriminant – program

```
/*@ requires
  @     (b==0.   || 0x1.p-916 <= \abs(b*b)) &&
  @     (a*c==0. || 0x1.p-916 <= \abs(a*c)) &&
  @     \abs(b) <= 0x1.p510 &&
  @     \abs(a) <= 0x1.p995 && \abs(c) <= 0x1.p995 &&
  @     \abs(a*c) <= 0x1.p1021;
  @ ensures \result==0.
  @      || \abs(\result-(b*b-a*c)) <= 2.*ulp(\result);
  @ */

double discriminant(double a, double b, double c) {
  dou
  p=b*b;
  q=a*c;

  if (p+q <= 3*fabs(p-q))
    d=p-q;
  else {
    dp=Dekker(b,b,p);
    dq=Dekker(a,c,q);
    d=(p-q)+(dp-dq);
  }
  return d;
}
```

Test whether $ac \approx b^2$

If $ac \approx b^2$, compute accurately using errors of the multiplications

Accurate discriminant – program

```
/*@ requires
  @      (b==0.   || 0x1.p-916 <= \abs(b*b)) &&
  @      (a*c==0. || 0x1.p-916 <= \abs(a*c)) &&
  @      \abs(b) <= 0x1.p510 &&
  @      \abs(a) <= 0x1.p995 && \abs(c) <= 0x1.p995 &&
  @      \abs(a*c) <= 0x1.p1021;
  @ ensures \result==0.
  @       || \abs(\result-(b*b-a*c)) <= 2.*ulp(\result);
  @ */

double discriminant(double a, double b, double c) {
  double p,q,d,dp,dq;
  p=b*b;
  q=a*c;

  if (p+q <= 3*fabs(p-q))
    d=p-q;
  else {
    dp=Dekker(b,b,p);
    dq=Dekker(a,c,q);
    d=(p-q)+(dp-dq);
  }
  return d;
}
```

Underflow

Accurate discriminant – program

```
/*@ requires
  @      (b==0.   || 0x1.p-916 <= \abs(b*b)) &&
  @      (a*c==0. || 0x1.p-916 <= \abs(a*c)) &&
  @      \abs(b) <= 0x1.p510 &&
  @      \abs(a) <= 0x1.p995 && \abs(c) <= 0x1.p995 &&
  @      \abs(a*c) <= 0x1.p1021;
  @ ensures \result==0.
  @       || \abs(\result-(b*b-a*c)) <= 2.*ulp(\result);
  @ */

double discriminant(double a, double b, double c) {
  double p,q,d,dp,dq;
  p=b*b;
  q=a*c;

  if (p+q <= 3*fabs(p-q))
    d=p-q;
  else {
    dp=Dekker(b,b,p);
    dq=Dekker(a,c,q);
    d=(p-q)+(dp-dq);
  }
  return d;
}
```

Overflow

Accurate discriminant – program

```
/*@ requires
  @      (b==0.   || 0x1.p-916 <= \abs(b*b)) &&
  @      (a*c==0. || 0x1.p-916 <= \abs(a*c)) &&
  @      \abs(b) <= 0x1.p510 &&
  @      \abs(a) <= 0x1.p995 && \abs(c) <= 0x1.p995 &&
  @      \abs(a*c) <= 0x1.p1021;
  @ ensures \result==0.
  @      || \abs(\result-(b*b-a*c)) <= 2.*ulp(\result);
  @ */
```

2 ulps

```
double discriminant(double a, double b, double c) {
  double p,q,d,dp,dq;
  p=b*b;
  q=a*c;

  if (p+q <= 3*fabs(p-q))
    d=p-q;
  else {
    dp=Dekker(b,b,p);
    dq=Dekker(a,c,q);
    d=(p-q)+(dp-dq);
  }
  return d;
}
```

Accurate discriminant – program

```
/*@ requires
  @     (b==0.   || 0x1.p-916 <= \abs(b*b)) &&
  @     (a*c==0. || 0x1.p-916 <= \abs(a*c)) &&
  @     \abs(b) <= 0x1.p510 &&
  @     \abs(a) <= 0x1.p995 && \abs(c) <= 0x1.p995 &&
  @     \abs(a*c) <= 0x1.p1021;
  @ ensures \result==0.
  @      || \abs(\result-(b*b-a*c)) <= 2.*ulp(\result);
  @ */

double discriminant(double a, double b, double c) {
  double p,q,d,dp,dq;
  p=b*b;
  q=a*c;

  if (p+
    d=p-q;
  else {
    dp=Dekker(b,b,p);
    dq=Dekker(a,c,q);
    d=(p-q)+(dp-dq);
  }
  return d;
}
```

Function calls

⇒ pre-conditions to prove

⇒ post-conditions guaranteed

Accurate discriminant – program

```
/*@ requires
  @      (b==0.   || 0x1.p-916 <= \abs(b*b)) &&
  @      (a*c==0. || 0x1.p-916 <= \abs(a*c)) &&
  @      \abs(b) <= 0x1.p510 &&
  @      \abs(a) <= 0x1.p995 && \abs(c) <= 0x1.p995 &&
  @      \abs(a*c) <= 0x1.p1021;
  @ ensures \result==0.
  @        || \abs(\result-(b*b-a*c)) <= 2.*ulp(\result);
  @ */
```

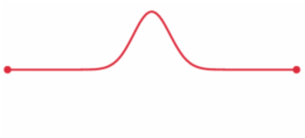
```
double                              le b, double c) {
  doub
  p=b*b;
  q=a*c;

  if (p+q <= 3*fabs(p-q))
    d=p-q;
  else {
    dp=Dekker(b,b,p);
    dq=Dekker(a,c,q);
    d=(p-q)+(dp-dq);
  }
  return d;
}
```

In initial proof,
test assumed correct

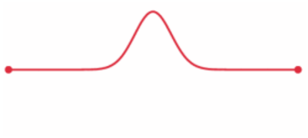
⇒ Additional proof
when test is incorrect

Wave equation resolution scheme

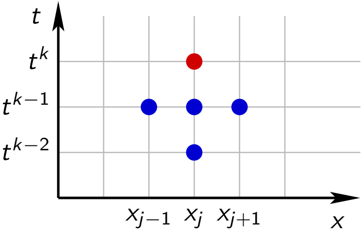


$$\frac{\partial^2 u(x,t)}{\partial t^2} - c^2 \frac{\partial^2 u(x,t)}{\partial x^2} = s(x,t)$$

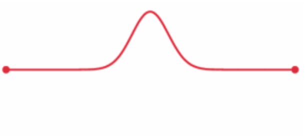
Wave equation resolution scheme



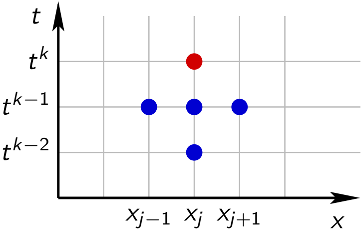
$$\frac{\partial^2 u(x,t)}{\partial t^2} - c^2 \frac{\partial^2 u(x,t)}{\partial x^2} = s(x,t) \quad \hookrightarrow$$



Wave equation resolution scheme



$$\frac{\partial^2 u(x,t)}{\partial t^2} - c^2 \frac{\partial^2 u(x,t)}{\partial x^2} = s(x,t) \quad \hookrightarrow$$



$$\frac{u_j^k - 2u_j^{k-1} + u_j^{k-2}}{\Delta t^2} - c^2 \frac{u_{j+1}^{k-1} - 2u_j^{k-1} + u_{j-1}^{k-1}}{\Delta x^2} = s_j^{k-1}$$

Wave equation resolution scheme – program

```
double **forward_prop(int ni, int nk, double dx, double dt,
     double v, double xs, double l) {
  double **p; int i, k; double a1, a, dp;

  a1 =  dt/dx*v; a  = a1*a1;

  [...] // initializations of  p[...][0] and  p[...][1]

  /* propagation = time loop */
  /*@ loop invariant 1 <= k <= nk && analytic_error(p,ni,ni,k,a);
    @ loop variant nk-k; */
  for (k=1; k<nk; k++) {
    p[0][k+1] = 0.;

    /* time iteration = space loop */
    /*@ loop invariant 1 <= i <= ni && analytic_error(p,ni,i-1,k+1,a)
      @ loop variant ni-i; */
    for (i=1; i<ni; i++) {
      dp = p[i+1][k] - 2.*p[i][k] + p[i-1][k];
      p[i][k+1] = 2.*p[i][k] - p[i][k-1] + a*dp;
    }
    p[ni][k+1] = 0.;
  }
  return p;
}
```

Wave equation resolution scheme – program

```
double **forward_prop(int ni, int nk, double dx, double dt,
     double v, double xs, double l) {
  double **p; int i, k; double a1, a, dp;

  a1 =  dt/dx*v; a  = a1*a1;

  [...] // initializations of  p[...][0] and  p[...][1]

  /* propagation = time loop */
  /*@ loop invariant 1 <= k <= nk && analytic_error(p,ni,ni,k,a);
    @ loop variant nk-k; */
  for (k=1; k<nk; k++) {
    p[0][k+1] = 0.;

    /* time iteration = space loop */
    /*@ loop invariant 1 <= i <= ni && analytic_error(p,ni,i-1,k+1,a)
      @ loop variant ni-i; */
    for (i=1; i<ni; i++) {
      dp = p[i+1][k] - 2.*p[i][k] + p[i-1][k];
      p[i][k+1] = 2.*p[i][k] - p[i][k-1] + a*dp;
    }
    p[ni][k+1] = 0.;
  }
  return p;
}
```

Time loop

Space loop

Wave equation resolution scheme – program

```
double **forward_prop(int ni, int nk, double dx, double dt,
    double v, double xs, double l) {
  double **p; int i, k; double a1, a, dp;

  a1 = dt/dx*v; a = a1*a1;

  [...] // initializations of p[...][0] and p[...][1]

  /* propagation = time loop */
  /*@ loop invariant 1 <= k <= nk && analytic_error(p,ni,ni,k,a);
    @ loop variant nk-k; */
  for (k=1; k<nk; k++) {
    p[0][k+1] = 0.;

    /* time iteration = space loop */
    /*@ loop invariant 1 <= i <= ni && analytic_error(p,ni,i-1,k+1,a)
      @ loop variant ni-i; */
    for (i=1; i<ni; i++) {
      dp = p[i+1][k] - 2.*p[i][k] + p[i-1][k];
      p[i][k+1] = 2.*p[i][k] - p[i][k-1] + a*dp;
    }
    p[ni][k+1] = 0.;
  }
  return p;
}
```

Loop invariant (outer, on the time loop)

Time loop

Loop invariant (inner, on the space loop)

Space loop

Wave equation resolution scheme – program

```
double **forward_prop(int ni, int nk, double dx, double dt,
     double v, double xs, double l) {
  double **p; int i, k; double a1, a, dp;

  a1 =  dt/dx*v; a  = a1*a1;

  [...] // initializations of  p[...][0] and  p[...][1]

  /* propagation = time loop */
  /*@ loop invariant 1 <= k <= nk && analytic_error(p,ni,ni,k,a);
    @ loop variant nk-k; */
  for (k=1; k<nk; k++) {
    p[0][k+1] = 0.;

    /* time iteration = space loop */
    /*@ loop invariant 1 <= i <= ni && analytic_error(p,ni,i-1,k+1,a)
      @ loop variant ni-i; */
    for (i=1; i<ni; i++) {
      dp = p[i+1][k] - 2.*p[i][k] + p[i-1][k];
      p[i][k+1] = 2.*p[i][k] - p[i][k-1] + a*dp;
    }
    p[ni][k+1] = 0.;
  }
  return p;
}
```

Main computations

Wave equation resolution scheme – program

```
double **forward_prop(int ni, int nk, double dx, double dt,
    double v, double xs, double l) {
  double **p; int i, k; double a1, a, dp;

  a1 = dt/dx*v; a = a1*a1;

  [...] // initializations of p[...][0] and p[...][1]

  /* propagation = time loop */
  /*@ loop invariant 1 <= k <= nk && analytic_error(p,ni,ni,k,a);
    @ loop variant nk-k; */
  for (k=1; k<nk; k++) {
    p[0][k+1] = 0.;

    /* time iteration = space
    /*@ loop invariant 1 <= i <= ni && analytic_error(p,ni,i-1,k+1,a)
      @ loop variant ni-i; */
    for (i=1; i<ni; i++) {
      dp = p[i+1][k] - 2.*p[i][k] + p[i-1][k];
      p[i][k+1] = 2.*p[i][k] - p[i][k-1] + a*dp;
    }
    p[ni][k+1] = 0.;
  }
  return p;
}
```

Accumulation of rounding errors

Main computations

Wave equation resolution scheme – rounding error

Interval arithmetic ⇒ p_i^k has error $2^k 2^{-53}$.

Wave equation resolution scheme – rounding error

Interval arithmetic $\Rightarrow \quad p_i^k$ has error $2^k 2^{-53}$.

We define ε_i^k as the signed rounding error made at step (i, k).

Wave equation resolution scheme – rounding error

Interval arithmetic $\Rightarrow$ $\quad p_i^k$ has error $2^k 2^{-53}$.

We define ε_i^k as the signed rounding error made at step (i, k).

The predicate `analytic_error(x,t)` is defined in Coq as :
For all steps (i, k) that are under (x, t),

- $|\varepsilon_i^k| \leq 78 \times 2^{-52}$
- $p_i^k - exact(p_i^k) = \sum_{l=0}^{k} \sum_{j=-l}^{l} \alpha_j^l \, \varepsilon_{i+j}^{k-l}$, $\qquad$ with known α_j^l

Wave equation resolution scheme – rounding error

Interval arithmetic $\Rightarrow \quad p_i^k$ has error $2^k 2^{-53}$.

We define ε_i^k as the signed rounding error made at step (i, k).

The predicate `analytic_error(x,t)` is defined in Coq as :
For all steps (i, k) that are under (x, t),

- $|\varepsilon_i^k| \leq 78 \times 2^{-52}$
- $p_i^k - exact(p_i^k) = \sum_{l=0}^{k} \sum_{j=-l}^{l} \alpha_j^l \, \varepsilon_{i+j}^{k-l}, \qquad$ with known α_j^l

$$\left| p_i^k - exact\left(p_i^k\right) \right| \leq 85 \times 2^{-53} \times (k+1) \times (k+2)$$

Wave equation resolution scheme – proof

- 33 proof obligations for the behavior
  (assertions, loop invariants, post-conditions. . .)

Wave equation resolution scheme – proof

- 33 proof obligations for the behavior
  (assertions, loop invariants, post-conditions. . .)
- 84 proof obligations for the safety
  (loop variants, Overflow, pointer dereferencing. . .)

Wave equation resolution scheme – proof

- 33 proof obligations for the behavior
  (assertions, loop invariants, post-conditions. . .)
- 84 proof obligations for the safety
  (loop variants, Overflow, pointer dereferencing. . .)
- 2 admits corresponding to the boundedness of the $exact(p_i^k)$
  (by scheme properties)

Wave equation resolution scheme – proof

- 33 proof obligations for the behavior
  (assertions, loop invariants, post-conditions...)
- 84 proof obligations for the safety
  (loop variants, Overflow, pointer dereferencing...)
- 2 admits corresponding to the boundedness of the $exact(p_i^k)$
  (by scheme properties)
- 26000 lines of Coq (including less than 3700 lines of proof)

  (Note that the method error proof was presented at ITP on July 11th)

Plan

1. Motivations
2. Tools
   - Formal proof
   - Frama-C/Jessie/Why
3. Examples
4. Conclusions

Conclusion : advantages

- Very high guarantee

Conclusion : advantages

- Very high guarantee
- not only rounding errors :

Conclusion : advantages

- Very high guarantee
- not only rounding errors :
  - all other errors such as pointer dereferencing or division by zero

Conclusion : advantages

- Very high guarantee
- not only rounding errors :
  - all other errors such as pointer dereferencing or division by zero
  - link with mathematical properties

Conclusion : advantages

- Very high guarantee
- not only rounding errors :
  - all other errors such as pointer dereferencing or division by zero
  - link with mathematical properties
  - any property can be checked

Conclusion : advantages

- Very high guarantee
- not only rounding errors :
  - all other errors such as pointer dereferencing or division by zero
  - link with mathematical properties
  - any property can be checked
- expressive annotation language (as expressive as Coq)
  $\Rightarrow$ exactly the specification you want

Conclusion : limits (1/2)

- long and tedious

Conclusion : limits (1/2)

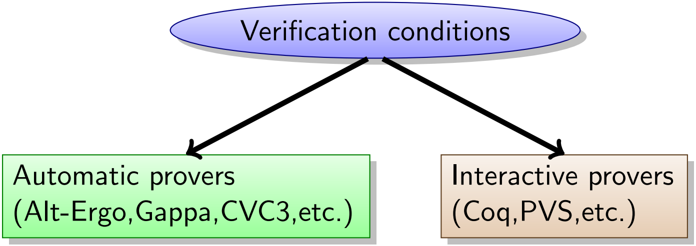
- long and tedious $\Rightarrow$ automations !

Conclusion : limits (1/2)

- long and tedious $\Rightarrow$ automations !
- for example Gappa (G. Melquiond)

Conclusion : limits (1/2)

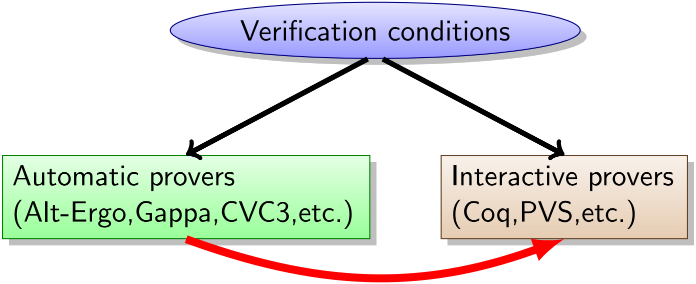
- long and tedious ⇒ automations !
- for example Gappa (G. Melquiond)



⇒ Use automatic provers to prove part of the verification conditions

Conclusion : limits (1/2)

- long and tedious ⇒ automations !
- for example Gappa (G. Melquiond)



⇒ Use automatic provers to prove part of the verification conditions
⇒ Use Gappa inside Coq to ease proofs

Conclusion : limits (2/2)

- We assume all `double` operations are direct 64-bit roundings.

Conclusion : limits (2/2)

- We assume all `double` operations are direct 64-bit roundings.
- On recent processors, we have x86 extended registers (80-bit long) and FMA ($\circ(ax + b)$ with one single rounding).

Conclusion : limits (2/2)

- We assume all `double` operations are direct 64-bit roundings.
- On recent processors, we have x86 extended registers (80-bit long) and FMA ($\circ(ax + b)$ with one single rounding).
- How does we know how the program was compiled and what will be the result ?

Conclusion : limits (2/2)

- We assume all `double` operations are direct 64-bit roundings.
- On recent processors, we have x86 extended registers (80-bit long) and FMA ($\circ(ax + b)$ with one single rounding).
- How does we know how the program was compiled and what will be the result ?
- Solution 1 : cover all cases.
  The result of an operation is a real near the correct result (it covers, 64-bit, 80-bit, double roundings and all uses of FMA)
  pragma `multi-rounding`
  Less precise, but always correct !

Conclusion : limits (2/2)

- We assume all `double` operations are direct 64-bit roundings.
- On recent processors, we have x86 extended registers (80-bit long) and FMA ($\circ(ax + b)$ with one single rounding).
- How does we know how the program was compiled and what will be the result ?

- Solution 1 : cover all cases.
  The result of an operation is a real near the correct result (it covers, 64-bit, 80-bit, double roundings and all uses of FMA)
  pragma `multi-rounding`
  Less precise, but always correct !

- Solution 2 : look into the assembly. . .

Perspectives

- How to find correct specifications ?

Perspectives

- How to find correct specifications ?

  $\Rightarrow$ use other tools...

Perspectives

- How to find correct specifications ?

  $\Rightarrow$ use other tools...

- What about the Coq library ?

Perspectives

- How to find correct specifications ?

  ⇒ use other tools. . .

- What about the Coq library ?

PFF
(Boldo, Théry,
Rideau. . .)

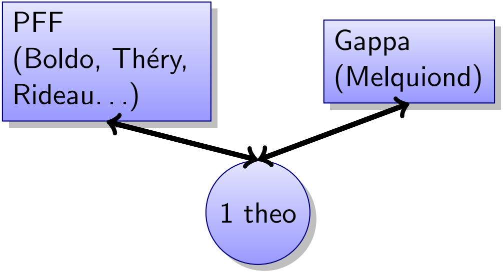
Gappa
(Melquiond)

Perspectives

- How to find correct specifications ?

  ⇒ use other tools. . .

- What about the Coq library ?

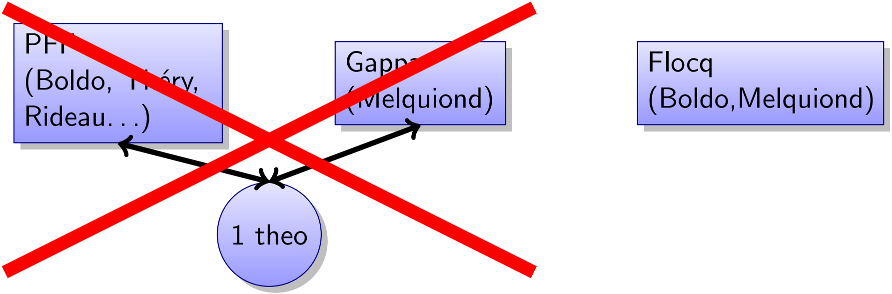


Perspectives

- How to find correct specifications ?

  ⇒ use other tools. . .

- What about the Coq library ?

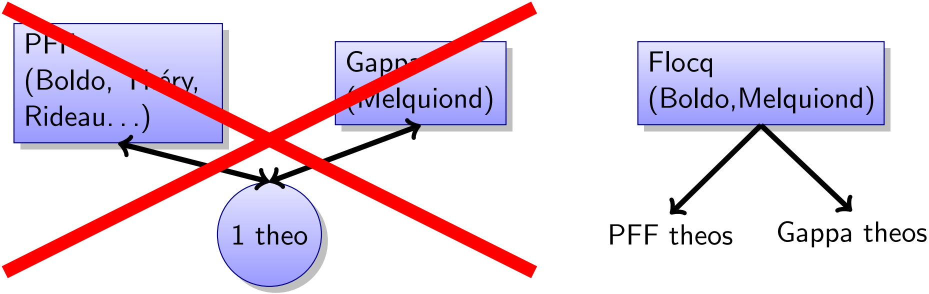


Perspectives

- How to find correct specifications ?

  $\Rightarrow$ use other tools. . .

- What about the Coq library ?



Thank you for your attention

- **Tools :**
  - `http://frama-c.com/`
  - `http://why.lri.fr/`
  - `http://coq.inria.fr/`
- **Code & proofs :**
  - `http://www.lri.fr/~sboldo/research.html`.
- **Formal proofs about scientific computations :**
  - `http://fost.saclay.inria.fr/`