Formal verification of tricky numerical computations

Sylvie Boldo

Inria

September 25th, 2014

(joint work with Clément, Filliâtre, Mayero, Melquiond, Weis)
Motivations

- Scientific Computing, Computer Arithmetic and Validated Numerics
Motivations

- Scientific Computing, Computer Arithmetic and Validated Numerics
Motivations

- **Scientific Computing, Computer Arithmetic and **Validated **Numerics**

- **My personal challenge:**

  - Consider small critical programs, where complex properties about floating-point arithmetic are involved.
  - How can we get a high guarantee?
    - Formal verification
  - Convince people of what is formally verified!
Motivations

- Scientific Computing, Computer Arithmetic and Validated Numerics

- My personal challenge: CORRECTNESS
Motivations

- Scientific Computing, Computer Arithmetic and Validated Numerics

- My personal challenge: CORRECTNESS

- consider small critical programs, where complex properties about floating-point arithmetic are involved
Motivations

- Scientific Computing, Computer Arithmetic and Validated Numerics

- My personal challenge: **CORRECTNESS**

- consider small critical programs, where complex properties about floating-point arithmetic are involved

- How can we get a high guarantee?

  - formal verification
Motivations

- Scientific Computing, Computer Arithmetic and Validated Numerics

- My personal challenge: **CORRECTNESS**

  - consider small critical programs, where complex properties about floating-point arithmetic are involved

  - How can we get a high guarantee?

    - formal verification

  - Convince people of what is formally verified!
Outline

1 Introduction

2 Tools
   - Frama-C/Jessie/Why
   - ACSL
   - Proof assistant: Coq

3 Examples
   - Sterbenz
   - Error of the multiplication
   - Accurate discriminant
   - Area of a triangle
   - 1-D Wave equation discretization

4 Conclusion
The used toolchain: Frama-C/Jessie/Why

Annotated C program
The used toolchain: Frama-C/Jessie/Why

Annotated C program

Frama-C/Jessie plug-in

WHY verification condition generator

Verification conditions
The used toolchain: Frama-C/Jessie/Why

1. Annotated C program
2. Frama-C/Jessie plug-in
3. WHY verification condition generator
4. Verification conditions
   - Automatic provers (Alt-Ergo, Gappa, CVC3, etc.)
   - Interactive provers (Coq, PVS, etc.)
Outline

1 Introduction

2 Tools
   • Frama-C/Jessie/Why
   • ACSL
   • Proof assistant: Coq

3 Examples
   • Sterbenz
   • Error of the multiplication
   • Accurate discriminant
   • Area of a triangle
   • 1-D Wave equation discretization

4 Conclusion
Annotation language: ACSL

(how a bug differs from a rounding error)
Annotation language: ACSL

(how a bug differs from a rounding error)

- ANSI/ISO C Specification Language
Annotation language: ACSL

(how a bug differs from a rounding error)

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

(how a bug differs from a rounding error)

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

(how a bug differs from a rounding error)

- 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.
Annotation language: ACSL

(how a bug differs from a rounding error)

- 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
Annotation language: ACSL

(how a bug differs from a rounding error)

- 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.
Annotation language: ACSL

(how a bug differs from a rounding error)

- 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.

⇒ For the programmer, the specification is easy to understand.
A floating-point number is a triple:

- the **floating-point number**, really computed by the program, \( x \rightarrow x_f \) floating-point part.
A floating-point number is a triple:

- the **floating-point number**, really computed by the program,
  \[ x \rightarrow x_f \] floating-point part

- the **value that would have been obtained with exact computations**, 
  \[ x \rightarrow x_e \] exact part
A floating-point number is a triple:

- the floating-point number, really computed by the program, \( x \rightarrow x_f \) floating-point part
- the value that would have been obtained with exact computations, \( x \rightarrow x_e \) exact part
- the value that we ideally wanted to compute \( x \rightarrow x_m \) model part
A floating-point number is a triple:

- The **floating-point number**, really computed by the program,
  \[ x \rightarrow x_f \text{ floating-point part} \]
  \[ 1 + x + x^2 / 2 \]

- The **value that would have been obtained with exact computations**, 
  \[ x \rightarrow x_e \text{ exact part} \]
  \[ 1 + x + \frac{x^2}{2} \]

- The **value that we ideally wanted to compute**
  \[ x \rightarrow x_m \text{ model part} \]
  \[ \exp(x) \]
A floating-point number is a triple:

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

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

1 Introduction

2 Tools
   - Frama-C/Jessie/Why
   - ACSL
   - Proof assistant: Coq

3 Examples
   - Sterbenz
   - Error of the multiplication
   - Accurate discriminant
   - Area of a triangle
   - 1-D Wave equation discretization

4 Conclusion
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 $\lambda$-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.
A FP format is only characterized by a function $\varphi : \mathbb{Z} \rightarrow \mathbb{Z}$. 
A FP format is only characterized by a function $\varphi : \mathbb{Z} \rightarrow \mathbb{Z}$.

For $x \in \mathbb{R}$, we compute $e$ such that $\beta^{e-1} \leq |x| < \beta^e$. Then $x$ is in the format iff

$$x = \left\lfloor x \beta^{-\varphi(e)} \right\rfloor \beta^{\varphi(e)}$$

In other words: if it can be written with exponent $\varphi(e)$. 
Usual Formats

**Definition (FIX)**

Fixed-point format with exponent $e_{\text{min}}$: $\varphi(e) = e_{\text{min}}$. 
Usual Formats

Definition (FIX)

Fixed-point format with exponent $e_{\min}$: $\varphi(e) = e_{\min}$.

Definition (FL*)

Floating-point format with precision $p$:
- unbounded (FLX): $\varphi(e) = e - p$,
Usual Formats

Definition (FIX)

Fixed-point format with exponent \( e_{\text{min}} \): \( \varphi(e) = e_{\text{min}} \).

Definition (FL*)

Floating-point format with precision \( p \):
- unbounded (FLX): \( \varphi(e) = e - p \),
- bounded with subnormal numbers (FLT): \( \varphi(e) = \max(e - p, e_{\text{min}}) \),
Usual Formats

**Definition (FIX)**
Fixed-point format with exponent $e_{\text{min}}$: $\varphi(e) = e_{\text{min}}$.

**Definition (FL*)**
Floating-point format with precision $p$:
- unbounded (FLX): $\varphi(e) = e - p$,
- bounded with subnormal numbers (FLT): $\varphi(e) = \max(e - p, e_{\text{min}})$,
- bounded without subnormal numbers (FTZ).

A random $\varphi$ may not allow to define a rounding: we have a valid predicate for being a reasonable $\varphi$. 
Usual Floating-Point Formats

\[ \varphi(e) \]

\[ e_{\text{min}} + \frac{p - 1}{p} \]

\[ e_{\text{min}} \]

\[ e_{\text{min}} + p - 1 \]

\[ e \]

\[ p \]

FTZ

FTX

FLT

Formal verification of numerical computations
Example of Coq theorem

**Theorem (round_NE_abs)**

Let $\varphi$ be a format, such that the rounding to nearest, ties to even ($\circ$) can be defined. For all $x \in \mathbb{R}$, $\circ(|x|) = |\circ(x)|$.  

Lemma round_NE_abs: forall x : R, round beta fexp ZnearestE (Rabs x) = Rabs (round beta fexp ZnearestE x).

Proof with auto with typeclass instances.

intros x; apply sym eq.
unfold Rabs at 2.
destruct (Rcase abs x) as [Hx|Hx].
rewrite round NE opp.
apply Rabs left1.
rewrite <- (round 0 beta fexp ZnearestE).
apply round le...
now apply Rlt le.
apply Rabs pos eq.
rewrite <- (round 0 beta fexp ZnearestE).
apply round le...
now apply Rge le.
Qed.

With the stating of the theorem, the tactics, and the name of theorems.
Example of Coq theorem

**Theorem (round\_NE\_abs)**

Let $\varphi$ be a format, such that the rounding to nearest, ties to even ($\circ$) can be defined. For all $x \in \mathbb{R}$, $\circ(|x|) = |\circ(x)|$.

**Lemma round\_NE\_abs:**

forall x : R, 
round beta fexp ZnearestE (Rabs x) = Rabs (round beta fexp ZnearestE x).
Example of Coq theorem

**Theorem (round\_NE\_abs)**

Let $\varphi$ be a format, such that the rounding to nearest, ties to even ($\circ$) can be defined. For all $x \in \mathbb{R}$, $\circ(|x|) = |\circ(x)|$.

**Lemma round\_NE\_abs**: forall $x : R$,

$\text{round beta fexp ZnearestE (Rabs x)} = \text{Rabs (round beta fexp ZnearestE x)}$.

Proof with auto with typeclass_instances.

intros x; apply sym_eq.
unfold Rabs at 2.
destruct (Rcase abs x) as [Hx|Hx].
rewrite round\_NE\_opp.
apply Rabs_left1.
rewrite <- (round\_0 beta fexp ZnearestE).
apply round\_le...
now apply Rlt\_le.
apply Rabs_pos_eq.
rewrite <- (round\_0 beta fexp ZnearestE).
apply round\_le...
now apply Rge\_le.
Qed.

With the stating of the theorem, the tactics, and the name of theorems.
More about Flocq

Flocq: 16,000 lines of Coq, 700 theorems,
- any radix, any format,
- both axiomatic and computable definitions of rounding,
- effective arithmetic operators,
- numerous theorems.
More about Flocq

**Flocq**: 16,000 lines of Coq, 700 theorems,
- any radix, any format,
- both axiomatic and computable definitions of rounding,
- effective arithmetic operators,
- numerous theorems.

**Applications**:
- Frama-C/Jessie  
  C code certifier
- CompCert  
  certified C compiler

http://flocq.gforge.inria.fr/
More about Flocq

Flocq: 16 000 lines of Coq, 700 theorems,
- any radix, any format,
- both axiomatic and computable definitions of rounding,
- effective arithmetic operators,
- numerous theorems.

Applications:
- **Frama-C/Jessie**  
  C code certifier
- **CompCert**  
  certified C compiler

http://flocq.gforge.inria.fr/
Methodology for the verification of C programs

C Program

The program is correct with respect to its specifications.
Methodology for the verification of C programs

- Annotated C Program
  (specification, invariant)

- Theorem statements

- Proof tools:
  - Frama-C
  - Jessie
  - Automatic provers (Alt-Ergo, Gappa, Z3)

- Proved Theorems

The program is correct with respect to its specifications.
Methodology for the verification of C programs

Human

Annotated C Program (specification, invariant)

Frama-C

Theorem statements

Human

Coq

The program is correct with respect to its specifications

Sylvie Boldo (Inria)
Formal verification of numerical computations
September 25th, 2014
Methodology for the verification of C programs

Annotated C Program (specification, invariant)

Human

Frama-C

Theorem statements

Jessie

Automatic provers (Alt-Ergo, Gappa, Z3)

Coq ← Human

The program is correct with respect to its specifications
Methodology for the verification of C programs

Annotated C Program (specification, invariant) → Frama-C

Theorem statements → Coq

Automatic provers (Alt-Ergo, Gappa, Z3)

Human

Coq ← Human

The program is correct with respect to its specifications
Methodology for the verification of C programs

Human

Annotated C Program (specification, invariant)

Frama-C
Jessie

Theorem statements

Automatic provers (Alt-Ergo, Gappa, Z3)

Coq ← Human

The program is correct with respect to its specifications
Methodology for the verification of C programs

Annotated C Program (specification, invariant) → Theorem statements

Human → Annotated C Program (specification, invariant)

Frama-C → Theorem statements

Jessie

Automatic provers (Alt-Ergo, Gappa, Z3)

Coq ← Human

The program is correct with respect to its specifications

Sylvie Boldo (Inria)  Formal verification of numerical computations  September 25th, 2014  16 / 52
Methodology for the verification of C programs

- Human
  - Annotated C Program (specification, invariant)
    - Frama-C
      - Automatic provers (Alt-Ergo, Gappa, Z3)
    - Jessie
  - Theorem statements
    - Coq
      - Human

The program is correct with respect to its specifications.
Methodology for the verification of C programs

- Annotated C Program (specification, invariant)
  - Frama-C
  - Jessie

  → Theorem statements
    - Automatic provers (Alt-Ergo, Gappa, Z3)
      - Coq

  ↓ Human

The program is correct with respect to its specifications.
Methodology for the verification of C programs

Human

Annotated C Program (specification, invariant)

Frama-C

Theorem statements

Jessie

Automatic provers (Alt-Ergo, Gappa, Z3)

Coq

→ Human
Methodology for the verification of C programs

- Annotated C Program (specification, invariant)
- Theorem statements

Human → Frama-C

Jessie

Automatic provers (Alt-Ergo, Gappa, Z3)

Coq ← Human
Methodology for the verification of C programs

- Annotated C Program (specification, invariant)
- Theorem statements
- Coq
- Human

Proved Theorems

Automatic provers (Alt-Ergo, Gappa, Z3)
Methodology for the verification of C programs

Human

Annotated C Program (specification, invariant)

Frama-C

Theorem statements

Jessie

Automatic provers (Alt-Ergo, Gappa, Z3)

Coq

The program is correct with respect to its specifications

Proved Theorems

Human
Outline

1 Introduction

2 Tools
   • Frama-C/Jessie/Why
   • ACSL
   • Proof assistant: Coq

3 Examples
   • Sterbenz
   • Error of the multiplication
   • Accurate discriminant
   • Area of a triangle
   • 1-D Wave equation discretization

4 Conclusion
Examples

- All examples use Frama-C Neon, Why 2.34 and Why3 0.83.
Examples

- All examples use Frama-C Neon, Why 2.34 and Why3 0.83.
- Non-automatic proof obligations are proved using Coq 8.4pl4.
Examples

- All examples use Frama-C Neon, Why 2.34 and Why3 0.83.
- Non-automatic proof obligations are proved using Coq 8.4pl4.
- Overflow is considered a runtime error.
Examples

- All examples use Frama-C Neon, Why 2.34 and Why3 0.83.
- Non-automatic proof obligations are proved using Coq 8.4pl4.
- Overflow is considered a runtime error.
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.
/*@ requires y/2. <= x <= 2.*y; @ ensures \result == x-y; @*/

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

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

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

Exact subtraction
<table>
<thead>
<tr>
<th>Proof obligations</th>
<th>CVC3</th>
<th>Coq</th>
</tr>
</thead>
<tbody>
<tr>
<td>VC for behavior</td>
<td></td>
<td>2.34</td>
</tr>
<tr>
<td>VC for safety</td>
<td>0.23</td>
<td></td>
</tr>
</tbody>
</table>

Sylvie Boldo (Inria)
Formal verification of numerical computations
September 25th, 2014
Outline

1 Introduction

2 Tools
  - Frama-C/Jessie/Why
  - ACSL
  - Proof assistant: Coq

3 Examples
  - Sterbenz
  - Error of the multiplication
  - Accurate discriminant
  - Area of a triangle
  - 1-D Wave equation discretization

4 Conclusion
Also known as Error-Free-Transformation for the multiplication.

**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.*
Also known as Error-Free-Transformation for the multiplication.

**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/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);
 @*/

define Dekker(double x, double y, double xy) {

double C, px, qx, hx, py, qy, hy, tx, ty, r2;
C=0x8000001p0;
/*@ assert C == 0x1p27+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) && 
@ \hspace*{1em} \abs(x) <= 0x1.p995 && 
@ \hspace*{1em} \abs(y) <= 0x1.p995 && 
@ \hspace*{1em} \abs(x*y) <= 0x1.p1021; 
@ ensures (\hspace*{1em} (x*y == 0 || 0x1.p-969 <= \abs(x*y)) 
@ \hspace*{1em} ==> x*y == xy+\result); 
@*/

double Dekker(double x, double y, double xy) {

double C, px, qx, hx, py, qy, hy, tx, ty, r2;
C=0x8000001p0;
/*@ assert C == 0x1p27+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;
}
/*@ requires xy == round_double(NearestEven,x*y) && */
/*@ abs(x) <= 0x1.p995 && */
/*@ abs(y) <= 0x1.p995 && */
.Fprintf(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;
  C=0x8000001p0;
  //@ assert C == 0x1p27+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;
}
/*@ 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;
C=0x8000001p0;
/*@ assert C == 0x1p27+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;
}
```c
/*@ 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))
@ \implies x*y == xy+\result);
@*/

double Dekker(double x, double y, double xy) {

double C, px, qx, hx, py, qy, hy, tx, ty, r2;
C=0x8000001p0;
/*@ assert C == 0x1p27+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;
}
```
/*@ 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;
  C=0x8000001p0;
 /*@ assert C == 0x1p27+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;
    C=0x8000001p0;
   /*@ assert C == 0x1p27+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;
}
<table>
<thead>
<tr>
<th>Proof obligations</th>
<th>Coq Nb lines</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Previous Coq proof (spec + proof)</td>
<td>2639</td>
<td></td>
</tr>
<tr>
<td>VC for behavior</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. assertion</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2. postcondition</td>
<td>238</td>
<td></td>
</tr>
<tr>
<td>VC for safety</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1-9. FP overflow</td>
<td>1 or 2</td>
<td></td>
</tr>
<tr>
<td>10. FP overflow</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>11. FP overflow</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>12. FP overflow</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>13. FP overflow</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>14. FP overflow</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>15. FP overflow</td>
<td>83</td>
<td></td>
</tr>
<tr>
<td>16. FP overflow</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>17. FP overflow</td>
<td>94</td>
<td></td>
</tr>
<tr>
<td><strong>Total (1,248 lines spec VC excluded)</strong></td>
<td><strong>3351</strong></td>
<td><strong>9 min 02</strong></td>
</tr>
</tbody>
</table>
Outline

1. Introduction

2. Tools
   - Frama-C/Jessie/Why
   - ACSL
   - Proof assistant: Coq

3. Examples
   - Sterbenz
   - Error of the multiplication
   - Accurate discriminant
   - Area of a triangle
   - 1-D Wave equation discretization

4. Conclusion
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 $b^2 - a \cdot c$ within 2 ulps.*
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 $b^2 - a \times c$ within 2 ulps.*

**Idea:**
Test whether there is cancellation. If not, then use the naive algorithm. Else, compute the errors of the multiplication, and add everything in the correct order.
Accurate discriminant – program

/*@ requires
@  (b==0. || \@abs(b*b)) &&
@  (a*c==0. || \@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

```c
/*@ 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.
@ \result-(b*b-a*c)) <= 2.*ulp(result);
@ */

double discriminant(double a, double b, double c) {
    double 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 of cancellation

When $p \geq q$, it roughly corresponds to $p \geq 2q$
Accurate discriminant – program

```c
/*@ 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;
  q=a;

  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

```c
/*@ 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;
}
```

Compute the multiplication errors
Accurate discriminant – program

```c
/*@ 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;
}
```

Add everything, $p-q$ being correct.

As $\frac{q}{2} \lesssim p \lesssim 2q$
Accurate discriminant – program

```c
/*@ 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) {
    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

```c
/*@ 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

```c
/*@ 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;
}
```

Test of cancellation
When $p \geq q$, it roughly corresponds to $p \geq 2q$.

Naive algorithm
Compute the multiplication errors, add everything, $p-q$ being correct.

As $q^2 \ll p \ll 2q$.

Function calls
⇒ pre-conditions to prove
⇒ post-conditions guaranteed
Accurate discriminant – program

```c
/*@ 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;
}
```

In initial proof, test assumed correct

⇒ Additional proof when test is incorrect
Accurate discriminant – proof

<table>
<thead>
<tr>
<th>Proof obligations</th>
<th>Coq</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Previous Coq proof (spec + proof)</td>
<td>3390</td>
<td></td>
</tr>
<tr>
<td>VC for theory realization</td>
<td>88</td>
<td></td>
</tr>
<tr>
<td>Behavior</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. postcondition</td>
<td>61</td>
<td></td>
</tr>
<tr>
<td>2. postcondition</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>Safety</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. floating-point overflow</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2. floating-point overflow</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3. floating-point overflow</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4. floating-point overflow</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5. floating-point overflow</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6. precondition for call</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>7. precondition for call</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>8. precondition for call</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>9-13. precondition for call</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>14. floating-point overflow</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>15. floating-point overflow</td>
<td>45</td>
<td></td>
</tr>
</tbody>
</table>

**Total (1,146 lines spec VC excluded)** 3655 5 min 47
1 Introduction

2 Tools
- Frama-C/Jessie/Why
- ACSL
- Proof assistant: Coq

3 Examples
- Sterbenz
- Error of the multiplication
- Accurate discriminant
- Area of a triangle
- 1-D Wave equation discretization

4 Conclusion
Heron's formula: \[ \triangle = \sqrt{s(s-a)(s-b)(s-c)} \]
where \( s = \frac{a+b+c}{2} \).

Kahan's formula, for \( c \leq b \leq a \):
\[ \triangle = \frac{1}{4} \sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a-(b-c))} \].

[Goldberg, 1991] Area \( \triangle \) is accurate to within a few units in their last digits.
[Goldberg, 1991] The rounding error of area \( \triangle \) is at most \( 11\varepsilon \), provided \( \varepsilon < 0.005 \) and subtraction and square roots are accurate.
Heron’s formula: $ \Delta = \sqrt{s \ (s-a) \ (s-b) \ (s-c)}$ where $s = \frac{a+b+c}{2}$.

Kahan’s formula, for $c \leq b \leq a$:

$$ \Delta = \frac{1}{4} \sqrt{(a+(b+c)) \ (c-(a-b)) \ (c+(a-b)) \ (a+(b-c))}. $$
Heron’s formula: $\Delta = \sqrt{s \,(s - a) \,(s - b) \,(s - c)}$ where $s = \frac{a+b+c}{2}$.

Kahan’s formula, for $c \leq b \leq a$:

$$\Delta = \frac{1}{4} \sqrt{(a+(b+c)) \,(c-(a-b)) \,(c+(a-b)) \,(a+(b-c))}.$$
Heron’s formula: \( \Delta = \sqrt{s (s - a) (s - b) (s - c)} \) where \( s = \frac{a+b+c}{2} \).

Kahan’s formula, for \( c \leq b \leq a \):

\[
\Delta = \frac{1}{4} \sqrt{(a + (b + c)) (c - (a - b)) (c + (a - b)) (a + (b - c))}.
\]

[Kahan, Miscalculating Area and Angles of a Needle-like Triangle]

Area \( \Delta \) is accurate to within a few units in their last digits.

[Goldberg, 1991]

The rounding error of area \( \Delta \) is at most 11 \( \varepsilon \), provided \( \varepsilon < 0.005 \) and subtraction and square roots are accurate.
Triangle area

Theorem (err_Δ_flx_radix2)

With an unbounded exponent range, \( \beta = 2 \), and \( \varepsilon \leq \frac{1}{100} \), the rounding error of area \( \Delta \) is at most \( 4.75\varepsilon + 33\varepsilon^2 \).
Theorem (err$_\Delta$$_{\text{flx\_radix2}}$)

With an unbounded exponent range, $\beta = 2$, and $\epsilon \leq \frac{1}{100}$, The rounding error of area $\Delta$ is at most $4.75\epsilon + 33\epsilon^2$.

For underflow:

- detect afterwards if a subnormal appeared in the computation
- order the intermediate variables, and multiply the biggest first:
  
  \[ 0 \leq c \ominus (a \ominus b) \leq c \oplus (a \ominus b) \leq a \oplus (b \ominus c) \leq a \oplus (b \oplus c) \]
Triangle area

Theorem (err_Δ_flt_radix2)

With an unbounded exponent range, $\beta = 2$, and $\epsilon \leq \frac{1}{100}$, the rounding error of area $\Delta$ is at most $4.75\epsilon + 33\epsilon^2$.

For underflow:

- detect afterwards if a subnormal appeared in the computation
- order the intermediate variables, and multiply the biggest first:
  
  $$0 \leq c \oplus (a \ominus b) \leq c \ominus (a \ominus b) \leq a \ominus (b \ominus c) \leq a \oplus (b \ominus c)$$

Theorem (err_Δ_flt_radix2)

We assume that $\beta = 2$, that $\epsilon \leq \frac{1}{100}$, and that $2\left[\frac{E_i + p - 1}{2}\right]^{-2} < \Delta$. The rounding error of area $\Delta$ (computed in the given order) is at most $4.75\epsilon + 33\epsilon^2$. 
Triangle area

**Theorem (err_\Delta_flx_radix2)**

With an unbounded exponent range, $\beta = 2$, and $\varepsilon \leq \frac{1}{100}$, the rounding error of area $\Delta$ is at most $4.75\varepsilon + 33\varepsilon^2$.

**For underflow:**

- detect afterwards if a subnormal appeared in the computation
- order the intermediate variables, and multiply the biggest first:

$$0 \leq c \ominus (a \ominus b) \leq c \oplus (a \ominus b) \leq a \oplus (b \ominus c) \leq a \oplus (b \oplus c)$$

**Theorem (err_\Delta_flt_radix2)**

We assume that $\beta = 2$, that $\varepsilon \leq \frac{1}{100}$, and that $2 \left\lceil \frac{E_i + p - 1}{2} \right\rceil^{-2} < \Delta$. The rounding error of area $\Delta$ (computed in the given order) is at most $4.75\varepsilon + 33\varepsilon^2$.

(and $5.75\varepsilon$ in radix 10 as multiplying by $\frac{1}{4}$ is not exact).
Triangle area – program

/*@ requires 0 <= x;
 @ ensures \result==\round\_double(\Nearest\_Even,\sqrt(x));
 @*/
double sqrt(double x);

/*@ logic real S(real a, real b, real c) =
 @ \let s = (a+b+c)/2;
 @ \sqrt(s*(s-a)*(s-b)*(s-c));
 @ */

/*@ requires 0 <= c <= b <= a && a <= b + c && a <= 0x1p255;
 @ ensures 0x1p-513 < \result
 @ ==> \abs(\result-S(a,b,c))
 @ <= (4.75*0x1p-53 + 33*0x1p-106)*S(a,b,c);
 @ */
double triangle (double a, double b, double c) {
    return (0x1p-2*sqrt((a+(b+c))*(a+(b-c))*(c+(a-b))*(c-(a-b))));
}
Triangle area – program

/*@ requires 0 <= x;
@ ensures \result == \round\_double(\Nearest\_Even,\sqrt(x));
@*/
double sqrt(double x);

/*@ logic real S(real a, real b, real c) =
@ \let s = (a+b+c)/2;
@ \sqrt(s*(s-a)*(s-b)*(s-c));
@ */

/*@ requires 0 <= c <= b <= a && a <= b + c && a <= 0x1p255;
@ ensures 0x1p-513 < \result
@ ==> \abs(\result-S(a,b,c))
@ <= (4.75*0x1p-53 + 33*0x1p-106)*S(a,b,c);
@ */

double triangle (double a, double b, double c) {
    return (0x1p-2*sqrt((a+(b+c))*(a+(b-c))*(c+(a-b))*(c-(a-b))));
}
Triangle area – program

/*@ requires 0 <= x;
 @ ensures \result==\round_double(\NearestEven,\sqrt(x));
 @*/
double sqrt(double x);

/*@ logic real S(real a, real b, real c) =
 @ \let s = (a+b+c)/2;
 @ \sqrt(s*(s-a)*(s-b)*(s-c));
 @ */

/*@ requires 0 <= c <= b <= a && a <= b + c && a <= 0x1p255;
 @ ensures 0x1p-513 < \result
 @ ==> \abs(\result-S(a,b,c))
 @ <= (4.75*0x1p-53 + 33*0x1p-106)*S(a,b,c);
 @ */

double triangle (double a, double b, double c) {
    return (0x1p-2*sqrt((a+(b+c))*(a+(b-c))*(c+(a-b))*(c-(a-b))));
}
Triangle area — program

/*@ requires 0 <= x;
   @ ensures \result == \round_double(\NearestEven,\sqrt(x));
   @*/

double sqrt(double x);

/*@ logic real S(real a, real b, real c) =
   @ \let s = (a+b+c)/2;
   @ \sqrt(s*(s-a)*(s-b)*(s-c));
   @ */

/*@ requires 0 <= c <= b <= a && a <= b + c && a <= 0x1p255;
@ ensures 0x1p-513 < \result
@ ==> \abs(\result-S(a,b,c))
@ <= (4.75*0x1p-53 + 33*0x1p-106)*S(a,b,c);
@ */

double triangle (double a, double b, double c) {
    return (0x1p-2*sqrt((a+(b+c))*(a+(b-c))*(c+(a-b))*(c-(a-b))));
}
Triangle area – program

/*@ requires 0 <= x;
@ ensures \result == \round_double(\text{NearestEven},\sqrt(x));
@*/
double sqrt(double x);

/*@ logic real S(real a, real b, real c) =
@ \let s = (a+b+c)/2;
@ \sqrt(s*(s-a)*(s-b)*(s-c));
@ */

double triangle(double a, double b, double c) {
    return (0x1p−2*sqrt((a+(b+c))*(a+(b−c))*(c+(a−b))*(c−(a−b))));
}
Triangle area – program

/*@ requires 0 <= x;
   @ ensures \result == \texttt{round\_double(\texttt{NearestEven},\sqrt{x))};
   @*/
double sqrt(double x);

/*@ logic real S(real a, real b, real c) =
   @ \texttt{let s = (a+b+c)/2;}
   @ \texttt{sqrt(s*(s-a)*(s-b)*(s-c));}
   @ */

/*@ requires 0 <= c <= b <= a && a <= b + c && a <= 0x1p255;
   @ ensures 0x1p-513 < \result
   @ ==> \abs(\result - S(a,b,c))
   @ <= (4.75*0x1p-53 + 33*0x1p-106)*S(a,b,c);
   @ */

double triangle (double a, double b, double c) {
    return (0x1p-2*sqrt((a+(b+c))*(a+(b-c))*(c+(a-b))*(c-(a-b))));
}
Triangle area – program

```c
/*@ requires 0 <= x;
 @ ensures \result==\round_double(\NearestEven,\sqrt(x));
 @*/
double sqrt(double x);

/*@ logic real S(real a, real b, real c) =
 @ \let s = (a+b+c)/2;
 @ \sqrt(s*(s-a)*(s-b)*(s-c));
 @ */

/*@ requires 0 <= c <= b <= a && a <= b + c && a <= 0x1p255;
 @ ensures 0x1p-513 < \result
 @ ==> \abs(\result-S(a,b,c))
 @ <= (4.75*0x1p-53 + 33*0x1p-106)*S(a,b,c);
 @ */

double triangle (double a, double b, double c) {
  return (0x1p−2*sqrt((a+(b+c))*(a+(b−c))*(c+(a−b))*(c−(a−b))));
}
```

Sylvie Boldo (Inria)  Formal verification of numerical computations  September 25th, 2014  33 / 52
Triangle area – program

```c
/*@ requires 0 <= x;
 @ ensures \result == \round_double(\NearestEven, \sqrt(x));
 @*/
double sqrt(double x);

/*@ logic real S(real a, real b, real c) =
 @ \let s = (a+b+c)/2;
 @ \sqrt(s*(s-a)*(s-b)*(s-c));
 @ */

/*@ requires 0 <= c <= b <= a && a <= b + c && a <= 0x1p255;
 @ ensures 0x1p-513 < \result
 @ ==> \abs(\result - S(a,b,c))
 @ <= (4.75*0x1p-53 + 33*0x1p-106)*S(a,b,c);
 @ */

double triangle (double a, double b, double c) {
    return (0x1p-2*sqrt((a+(b+c))*(a+(b-c))*(c+(a-b))*(c-(a-b))));
}
```
Triangle area – program

/*@ requires 0 <= x;
@ ensures \result==\round_double(\NearestEven,\sqrt(x));
@*/
double sqrt(double x);

/*@ logic real S(real a, real b, real c) =
@ \let s = (a+b+c)/2;
@ \sqrt(s*(s-a)*(s-b)*(s-c));
@ */

/*@ requires 0 <= c <= b <= a && a <= b + c && a <= 0x1p255;
@ ensures 0x1p-513 < \result
@ ==> \abs(\result-S(a,b,c))
@ <= (4.75*0x1p-53 + 33*0x1p-106)*S(a,b,c);
@ */

double triangle (double a, double b, double c) {
    return (0x1p-2*sqrt((a+(b+c))*(a+(b-c))*(c+(a-b))*(c-(a-b))));
}
## Triangle area – proof

<table>
<thead>
<tr>
<th>Proof obligations</th>
<th>Gappa</th>
<th>Coq</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Previous Coq proof</strong></td>
<td>18.89</td>
<td>2091</td>
</tr>
<tr>
<td><strong>Behavior</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. postcondition</td>
<td>16.00</td>
<td>82</td>
</tr>
<tr>
<td><strong>Safety</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. FP overflow</td>
<td>0.02</td>
<td></td>
</tr>
<tr>
<td>2. FP overflow</td>
<td>0.03</td>
<td></td>
</tr>
<tr>
<td>3. FP overflow</td>
<td>0.03</td>
<td></td>
</tr>
<tr>
<td>4. FP overflow</td>
<td>0.03</td>
<td></td>
</tr>
<tr>
<td>5. FP overflow</td>
<td>0.03</td>
<td></td>
</tr>
<tr>
<td>6. FP overflow</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>7. FP overflow</td>
<td>0.02</td>
<td></td>
</tr>
<tr>
<td>8. FP overflow</td>
<td>0.01</td>
<td></td>
</tr>
<tr>
<td>9. FP overflow</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>10. FP overflow</td>
<td>0.02</td>
<td></td>
</tr>
<tr>
<td>11. FP overflow</td>
<td>0.02</td>
<td></td>
</tr>
<tr>
<td>12. precondition for call</td>
<td>13.22</td>
<td>13</td>
</tr>
<tr>
<td>13. FP overflow</td>
<td>0.03</td>
<td></td>
</tr>
<tr>
<td>14. FP overflow</td>
<td>0.04</td>
<td></td>
</tr>
</tbody>
</table>
Triangle area – proof

```plaintext
Triangle area – proof

```
Outline

1 Introduction

2 Tools
- Frama-C/Jessie/Why
- ACSL
- Proof assistant: Coq

3 Examples
- Sterbenz
- Error of the multiplication
- Accurate discriminant
- Area of a triangle
- 1-D Wave equation discretization

4 Conclusion
The wave equation

Looking for $u : \mathbb{R}^2 \rightarrow \mathbb{R}$ regular enough such that:

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

with given values for the initial position $u_0(x)$ and the initial velocity $u_1(x)$.

$\Rightarrow$ rope oscillation, sound, radar, oil prospection...
We want \( u_j^k \approx u(j \Delta x, k \Delta t) \).

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

And other horrible formulas to initialize \( u_j^0 \) and \( u_j^1 \).
We want $u_j^k \approx u(j\Delta x, k\Delta t)$.

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

And other horrible formulas to initialize $u_j^0$ and $u_j^1$.

Three-point scheme: $u_j^k$ depends on $u_{j-1}^{k-1}$, $u_j^{k-1}$, $u_{j+1}^{k-1}$ and $u_j^{k-2}$. 
We want $u_j^k \approx u(j\Delta x, k\Delta t)$.

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

And other horrible formulas to initialize $u_j^0$ and $u_j^1$.

Three-point scheme: $u_j^k$ depends on $u_{j-1}^{k-1}$, $u_j^{k-1}$, $u_{j+1}^{k-1}$ and $u_j^{k-2}$.

Not really tricky computer arithmetic!
// initialization of p[i][0] and p[i][1]
for (k=1; k<nk; k++) {
    p[0][k+1] = 0.;
    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.;
}

Two different errors:
round-off errors
due to floating-point roundings
method errors
the scheme only approximates the exact solution
Program

// initialization of p[i][0] and p[i][1]
for (k=1; k<nk; k++) {
    p[0][k+1] = 0.;
    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.;

Two different errors:

- **round-off errors**
  due to floating-point roundings
- **method errors**
  the scheme only approximates the exact solution
Rounding error

Remainder:

\[ dp = p[i+1][k] - 2 \times p[i][k] + p[i-1][k]; \]
\[ p[i][k+1] = 2 \times p[i][k] - p[i][k-1] + a \times dp; \]

If we use a naive technique to bound the rounding errors, we get
Rounding error

Remainder:

\[ dp = p[i+1][k] - 2 \cdot p[i][k] + p[i-1][k]; \]
\[ p[i][k+1] = 2 \cdot p[i][k] - p[i][k-1] + a \cdot dp; \]

If we use a naive technique to bound the rounding errors, we get

\[ |p_i^k - exact(p_i^k)| \leq O \left( 2^k 2^{-53} \right) \]
Rounding error

Remainder:

\[ dp = p[i+1][k] - 2 \cdot p[i][k] + p[i-1][k]; \]
\[ p[i][k+1] = 2 \cdot p[i][k] - p[i][k-1] + a \cdot dp; \]

If we use a naive technique to bound the rounding errors, we get

\[ |p_i^k - exact(p_i^k)| \leq O\left(2^k 2^{-53}\right) \]

This is too much because the errors do compensate.
Definition of $\varepsilon_{i}^{k}$

Remainder:

\[
\begin{align*}
    dp &= p[i+1][k] - 2 \cdot p[i][k] + p[i-1][k]; \\
    p[i][k+1] &= 2 \cdot p[i][k] - p[i][k-1] + a \cdot dp;
\end{align*}
\]

Let $\varepsilon_{i}^{k+1}$ be the rounding error made during these two lines of computations.

We assume $a$, $p_{i-1}^{k}$, $p_{i}^{k}$, $p_{i+1}^{k}$ and $p_{i}^{k-1}$ are exact and we look into the rounding error of these two lines. It is called $\varepsilon_{i}^{k+1}$. 
Definition of $\varepsilon^k_i$

Remainder:

\[
dp = p[i+1][k] - 2 \times p[i][k] + p[i-1][k]; \\
p[i][k+1] = 2 \times p[i][k] - p[i][k-1] + a \times dp;
\]

Let $\varepsilon^{k+1}_i$ be the rounding error made during these two lines of computations.

We assume $a$, $p^k_{i-1}$, $p^k_i$, $p^k_{i+1}$ and $p^k_{i-1}$ are exact and we look into the rounding error of these two lines. It is called $\varepsilon^{k+1}_i$.

We know (from initializations) that the model values of the $|p^m_n|$ are bounded by 1. We assume that the floating-point values of the $|p^m_n|$ are bounded by 2.
Definition of $\varepsilon^k_i$

Remainder:

\[
dp = p[i+1][k] - 2.0 * p[i][k] + p[i-1][k];
\]
\[
p[i][k+1] = 2.0 * p[i][k] - p[i][k-1] + a * dp;
\]

Let $\varepsilon^{k+1}_i$ be the rounding error made during these two lines of computations.

We assume $a$, $p_{i-1}^k$, $p_i^k$, $p_{i+1}^k$ and $p_{i-1}^{k-1}$ are exact and we look into the rounding error of these two lines. It is called $\varepsilon^{k+1}_i$.

We know (from initializations) that the model values of the $|p_n^m|$ are bounded by 1. We assume that the floating-point values of the $|p_n^m|$ are bounded by 2.

\[
|\varepsilon_n^m| \leq 78 \times 2^{-52}
\]
Rounding error

\[ p^k_i - \text{exact}(p^k_i) = \sum_{l=0}^{k} \sum_{j=-l}^{l} \alpha^l_j \varepsilon^{k-l}_{i+j} \]

We have an analytical expression of the rounding error with known constants \( \alpha^k_i \).
Rounding error

\[
p_i^k - \text{exact}(p_i^k) = \sum_{l=0}^{k} \sum_{j=-l}^{l} \alpha_j \varepsilon_{i+j}^{k-l}
\]

1. We have an **analytical expression** of the rounding error with known constants \(\alpha_i^k\).
2. It is not that complicated!
   (we cannot get rid of the pyramidal double summation)
Rounding error

\[ p_i^k - \text{exact}(p_i^k) = \sum_{l=0}^{k} \sum_{j=-l}^{l} \alpha_j \varepsilon_{i+j}^k \]

1. We have an **analytical expression** of the rounding error with known constants \( \alpha_i^k \).

2. It is not that complicated!
   (we cannot get rid of the pyramidal double summation)

3. The rounding error is bounded by \( \bigcirc(k^2 2^{-53}) \):

\[ \left| p_i^k - \text{exact}(p_i^k) \right| \leq 78 \times 2^{-53} \times (k + 1) \times (k + 2) \]
We measure that \( u \) and \( u_j^k \) are close when \( (\Delta x, \Delta t) \to 0 \).

We define \( e_j^k \overset{\text{def}}{=} \bar{u}_j^k - u_j^k \): convergence error
where \( \bar{u}_j^k \) is the value of \( u \) at the \((j, k)\) point of the grid.
We measure that $u$ and $u_j^k$ are close when $(\Delta x, \Delta t) \to 0$.

We define $e_j^k \overset{\text{def}}{=} \bar{u}_j^k - u_j^k$: convergence error

where $\bar{u}_j^k$ is the value of $u$ at the $(j, k)$ point of the grid.

We want to bound $\| e_h^{k\Delta t}(t) \|_{\Delta x}$: the average of the convergence error on all points of the grid at a given time $k_{\Delta t}(t) = \left\lfloor \frac{t}{\Delta t} \right\rfloor \Delta t$. 
We measure that $u$ and $u_j^k$ are close when $(\Delta x, \Delta t) \to 0$.

We define $e_j^k \overset{\text{def}}{=} \bar{u}_j^k - u_j^k$: convergence error
where $\bar{u}_j^k$ is the value of $u$ at the $(j, k)$ point of the grid.

We want to bound $\left\| e_h^{k\Delta t(t)} \right\|_{\Delta x}$: the average of the convergence error on all points of the grid at a given time $k_{\Delta t}(t) = \left\lfloor \frac{t}{\Delta t} \right\rfloor \Delta t$.

We want to prove:

$$\left\| e_h^{k\Delta t(t)} \right\|_{\Delta x} = O_{[0,t_{\text{max}}]}(\Delta x^2 + \Delta t^2)$$
We proved that:

\[
\left\| e^{kh(t)} \right\|_{\Delta x} = O(t \in [0, t_{\text{max}}], (\Delta x, \Delta t) \to 0, 0 < \Delta x \land 0 < \Delta t \land \zeta \leq c \frac{\Delta t}{\Delta x} \leq 1 - \xi)
\]

\( (\Delta x^2 + \Delta t^2). \)

(This is out of the scope of this talk.)
Extraction of the big O constants

The preceding result is a uniform big O defined by:

\[ \exists \alpha, C > 0, \quad \forall x, \Delta x, \quad \| \Delta x \| \leq \alpha \Rightarrow |f(x, \Delta x)| \leq C \cdot |g(\Delta x)|. \]
Extraction of the big O constants

The preceding result is a uniform big O defined by:

$$\exists \alpha, C > 0, \quad \forall x, \Delta x, \quad \|\Delta x\| \leq \alpha \Rightarrow |f(x, \Delta x)| \leq C \cdot |g(\Delta x)|.$$  

Let \((\alpha_3, C_3)\) be the constants for the order-3 Taylor development of the exact solution and \((\alpha_4, C_4)\) for order-4. The initial support is \([\chi_1; \chi_2]\).
Extraction of the big O constants

The preceding result is a uniform big O defined by:

$$\exists \alpha, C > 0, \forall x, \Delta x, \| \Delta x \| \leq \alpha \Rightarrow |f(x, \Delta x)| \leq C \cdot |g(\Delta x)|.$$  

Let \((\alpha_3, C_3)\) be the constants for the order-3 Taylor development of the exact solution and \((\alpha_4, C_4)\) for order-4. The initial support is \([\chi_1; \chi_2]\).

\[
\begin{align*}
\alpha &= \min(\alpha_3, \alpha_4, 1, t_{\text{max}}) \\
\alpha_1 &= \max(1, 2 \cdot C_4 \cdot (c^2 + 1), C_3 \cdot (1 + c^2/2) + 1) \\
\alpha_2 &= \alpha_1^2 \left( |\chi_2| - |\chi_1| + 2 \cdot c \cdot t_{\text{max}} \cdot \left(1 + \frac{1}{\xi}\right) + 3 \right) \\
\alpha_3 &= \frac{1}{\sqrt{2}} \left( C_3 \cdot (1 + c^2/2) + 1 \right) \cdot (\chi_2 - \chi_1 + 1 + (2 \cdot c + 4)) \\
C &= \frac{\sqrt{2}}{\sqrt{2\xi - \xi^2}} \cdot 2 \cdot t_{\text{max}} \cdot s_3 \\
\end{align*}
\]
Program verification

- 154 lines of annotations for 32 lines of C
- **150 verification conditions:**
  - 44 about the behavior
  - 106 about the safety (runtime errors)
Program verification

- 154 lines of annotations for 32 lines of C
- **150 verification conditions:**
  - 44 about the behavior
  - 106 about the safety (runtime errors)

<table>
<thead>
<tr>
<th>Prover</th>
<th>Behavior VC</th>
<th>Safety VC</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alt-Ergo</td>
<td>18</td>
<td>80</td>
<td>98</td>
</tr>
<tr>
<td>CVC3</td>
<td>18</td>
<td>89</td>
<td>107</td>
</tr>
<tr>
<td>Gappa</td>
<td>2</td>
<td>20</td>
<td>22</td>
</tr>
<tr>
<td>Z3</td>
<td>21</td>
<td>63</td>
<td>84</td>
</tr>
<tr>
<td><strong>Automatically proved</strong></td>
<td><strong>23</strong></td>
<td><strong>94</strong></td>
<td><strong>117</strong></td>
</tr>
<tr>
<td><strong>Coq</strong></td>
<td><strong>21</strong></td>
<td><strong>12</strong></td>
<td><strong>33</strong></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>44</strong></td>
<td><strong>106</strong></td>
<td><strong>150</strong></td>
</tr>
</tbody>
</table>
Program verification

- About 90% of the safety goals (matrix access, Overflow, and so on) are proved automatically.
- 33 theorems are interactively proved using Coq for a total of about 15,000 lines of Coq and 30 minutes of compilation.

<table>
<thead>
<tr>
<th>Type of proofs</th>
<th>Nb spec lines</th>
<th>Nb lines</th>
<th>Compilation time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convergence</td>
<td>991</td>
<td>5275</td>
<td>42 s</td>
</tr>
<tr>
<td>Round-off + runtime errors</td>
<td>7737</td>
<td>13175</td>
<td>32 min</td>
</tr>
</tbody>
</table>
Outline

1 Introduction

2 Tools
   - Frama-C/Jessie/Why
   - ACSL
   - Proof assistant: Coq

3 Examples
   - Sterbenz
   - Error of the multiplication
   - Accurate discriminant
   - Area of a triangle
   - 1-D Wave equation discretization

4 Conclusion
always a Coq proof, generic wrt precision and minimal exponent (and often radix)
always a Coq proof, generic wrt precision and minimal exponent (and often radix)
but also an annotated C program that handles exceptional behavior (e. g. Overflow, division by zero)
Conclusion

- always a Coq proof, generic wrt precision and minimal exponent (and often radix)
- but also an annotated C program that handles exceptional behavior (e.g. Overflow, division by zero)
- formal proofs are required because algorithms are tricky
always a Coq proof, generic wrt precision and minimal exponent (and often radix)

but also an annotated C program that handles exceptional behavior (e. g. Overflow, division by zero)

formal proofs are required because algorithms are tricky

formal proofs are possible because algorithms are small
Conclusion

- always a Coq proof, generic wrt precision and minimal exponent (and often radix)
- but also an annotated C program that handles exceptional behavior (e.g. Overflow, division by zero)
- formal proofs are required because algorithms are tricky
- formal proofs are possible because algorithms are small
- (Have you seen long tricky algorithms?)
Conclusion

- always a Coq proof, generic wrt precision and minimal exponent (and often radix)
- but also an annotated C program that handles exceptional behavior (e.g. Overflow, division by zero)
- formal proofs are required because algorithms are tricky
- formal proofs are possible because algorithms are small
- (Have you seen long tricky algorithms?)

- not applicable on big (naive) industrial algorithms
Conclusion

- Very high guarantee
Conclusion

- Very high guarantee
- not only rounding errors:
Conclusion

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

- Very high guarantee

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

- 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

- 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
We assume all double operations are direct 64-bits roundings.
Limits: compilation

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

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

$\Rightarrow$ several possible results!
We assume all double operations are direct 64-bits roundings.

On recent processors, we have x86 extended registers (80-bits long) and FMA ($\circ(ax + b)$ with one single rounding).

$\Rightarrow$ several possible results!

Solution 1: cover all cases.
We assume all double operations are direct 64-bits roundings.

On recent processors, we have x86 extended registers (80-bits long) and FMA ($\circ(ax + b)$ with one single rounding).

⇒ several possible results!

Solution 1: cover all cases.

only use forward analysis with a slightly larger bound (it covers, 64-bit, 80-bit, double roundings and all uses of FMA)
We assume all double operations are direct 64-bits roundings.

On recent processors, we have x86 extended registers (80-bits long) and FMA \((\circ(ax + b))\) with one single rounding).

⇒ several possible results!

Solution 1: cover all cases.

only use forward analysis with a slightly larger bound (it covers, 64-bit, 80-bit, double roundings and all uses of FMA)

Solution 2: look into the assembly, and prove what is compiled.
Limits: compilation

- We assume all double operations are direct 64-bits roundings.
- On recent processors, we have x86 extended registers (80-bits long) and FMA \((a \times + b)\) with one single rounding.
  \[\Rightarrow\] several possible results!

- Solution 1: cover all cases.
  only use forward analysis with a slightly larger bound (it covers, 64-bit, 80-bit, double roundings and all uses of FMA)

- Solution 2: look into the assembly, and prove what is compiled.

- Solution 3: use a certified compiler, then compilation is specified.
a better handling of exceptional behaviors
Perspectives

- a better handling of exceptional behaviors
- prove and generalize well-known facts/algorithms/programs from the computer arithmetic community
• a better handling of exceptional behaviors
• prove and generalize well-known facts/algorithms/programs from the computer arithmetic community
  ⇒ basic blocks to build upon
Perspectives

- a better handling of exceptional behaviors
- prove and generalize well-known facts/algorithms/programs from the computer arithmetic community
  ⇒ basic blocks to build upon

- prove libraries with computational contents
  (e.g. computational geometry)
Perspectives

- a better handling of exceptional behaviors
- prove and generalize well-known facts/algorithms/programs from the computer arithmetic community
  \[ \Rightarrow \text{basic blocks to build upon} \]

- prove libraries with computational contents (e.g. computational geometry)
- go deeper into numerical analysis
Perspectives

- a better handling of exceptional behaviors
- prove and generalize well-known facts/algorithms/programs from the computer arithmetic community
  \[\Rightarrow\] basic blocks to build upon
- prove libraries with computational contents
  (e.g. computational geometry)
  \[\Rightarrow\] go deeper into numerical analysis
  \[\Rightarrow\] e.g. finite elements
Perspectives

- a better handling of exceptional behaviors
- prove and generalize well-known facts/algorithms/programs from the computer arithmetic community
  ⇒ basic blocks to build upon

- prove libraries with computational contents
  (e.g. computational geometry)

- go deeper into numerical analysis
  ⇒ e.g. finite elements
  ⇒ e.g. stability
This is not a slide.
Big O = big pain

Usually, the big O uses one variable and $f(x) = O_{∥x∥→0}(g(x))$ means:

$$\exists \alpha, C > 0, \ \forall x \in \mathbb{R}^n, \ ||x|| \leq \alpha \Rightarrow |f(x)| \leq C \cdot |g(x)|.$$
Big O = big pain

Usually, the big O uses one variable and $f(x) = O_{\|x\| \to 0}(g(x))$ means

$$\exists \alpha, C > 0, \forall x \in \mathbb{R}^n, \|x\| \leq \alpha \Rightarrow |f(x)| \leq C \cdot |g(x)|.$$ 

Here 2 variables: $\Delta x$ (grid sizes, tends to 0), and $x$ (time and space). (Think about Taylor expansions)
Big O = big pain

Usually, the big O uses one variable and \( f(x) = O_{\|x\| \to 0}(g(x)) \) means

\[
\exists \alpha, C > 0, \forall x \in \mathbb{R}^n, \|x\| \leq \alpha \Rightarrow |f(x)| \leq C \cdot |g(x)|.
\]

Here 2 variables: \( \Delta x \) (grid sizes, tends to 0), and \( x \) (time and space). (Think about Taylor expansions)

\[
\forall x, \exists \alpha, C > 0, \forall \Delta x \in \mathbb{R}^2, \|\Delta x\| \leq \alpha \Rightarrow |f(x, \Delta x)| \leq C \cdot |g(\Delta x)|
\]
does not work.
We used a uniform big O:

$$\exists \alpha, C > 0, \forall x, \Delta x, \|\Delta x\| \leq \alpha \Rightarrow |f(x, \Delta x)| \leq C \cdot |g(\Delta x)|.$$  

where variables $x$ and $\Delta x$ are restricted to subsets of $\mathbb{R}^2$.  
(for example such that $\Delta t > 0$)  
⇒ Taylor expansions
Proof idea 1/3: consistency

The truncation error is defined as how much the exact solution solves the numerical scheme:

$$\varepsilon_{j}^{k-1} = \frac{\bar{u}_{j}^{k} - 2\bar{u}_{j}^{k-1} + \bar{u}_{j}^{k-2}}{\Delta t^2} - c^2 \frac{\bar{u}_{j+1}^{k-1} - 2\bar{u}_{j}^{k-1} + \bar{u}_{j-1}^{k-1}}{\Delta x^2} - s_{j}^{k-1}$$
Proof idea 1/3: consistency

The truncation error is defined as how much the exact solution solves the numerical scheme:

\[ \varepsilon_{j}^{k-1} = \frac{\bar{u}_{j}^{k} - 2\bar{u}_{j}^{k-1} + \bar{u}_{j}^{k-2}}{\Delta t^{2}} - c^{2} \frac{\bar{u}_{j+1}^{k-1} - 2\bar{u}_{j}^{k-1} + \bar{u}_{j-1}^{k-1}}{\Delta x^{2}} - s_{j}^{k-1} \]

The consistency is the boundedness of the truncation error:

\[ \left\| \varepsilon_{\Delta t}^{(t)} \right\|_{\Delta x} = O_{[0, t_{\text{max}}]}(\Delta x^{2} + \Delta t^{2}) \]

By Taylor series and many computations.
Proof idea 2/3: stability

We define a discrete energy by

\[ E_h(c)(u_h)^{k+\frac{1}{2}} \overset{\text{def}}{=} \frac{1}{2} \left\| \frac{u_{h}^{k+1} - u_{h}^{k}}{\Delta t} \right\|_{\Delta x}^2 + \frac{1}{2} \left\langle u_{h}^{k}, u_{h}^{k+1} \right\rangle_{A_h(c)} \]

\text{kinetic energy} \quad \text{potential energy}

\[ \left\langle v_h, w_h \right\rangle_{A_h(c)} \overset{\text{def}}{=} \left\langle A_h(c) v_h, w_h \right\rangle_{\Delta x} \quad \text{and} \quad (A_h(c) v_h)_j \overset{\text{def}}{=} -c^2 \frac{v_{j+1} - 2v_j + v_{j-1}}{\Delta x^2}. \]
Proof idea 2/3: stability

We define a discrete energy by

\[ E_h(c)(u_h)^{k+\frac{1}{2}} \overset{\text{def}}{=} \frac{1}{2} \left\| \frac{u_h^{k+1} - u_h^k}{\Delta t} \right\|_{\Delta x}^2 + \frac{1}{2} \left\langle u_h^k, u_h^{k+1} \right\rangle_{A_h(c)} \]

\[ \langle v_h, w_h \rangle_{A_h(c)} \overset{\text{def}}{=} \langle A_h(c) v_h, w_h \rangle_{\Delta x} \text{ and } (A_h(c) v_h)_j \overset{\text{def}}{=} -c^2 \frac{v_{j+1} - 2v_j + v_{j-1}}{\Delta x^2}. \]

Note that this energy is constant if \( f = 0 \).
We prove an overestimation and an underestimation of this energy.
⇒ \( u_h \) does not diverge.
Proof idea 3/3: convergence

The convergence error is solution of the same discrete scheme with inputs

\[ u_{0,j} = 0, \quad u_{1,j} = \frac{e^1_j}{\Delta t}, \quad \text{and} \quad s^k_j = \varepsilon^{k+1}_j. \]

+ proofs about the initializations.
Proof idea 3/3: convergence

The convergence error is solution of the same discrete scheme with inputs

\[ u_{0,j} = 0, \quad u_{1,j} = \frac{e^1_j}{\Delta t}, \quad \text{and} \quad s^k_j = \varepsilon^{k+1}_j. \]

+ proofs about the initializations.

All these proofs require the existence of \( \zeta \) and \( \xi \) in \( ]0, 1[ \) with \( \zeta \leq 1 - \xi \) and we require that \( \zeta \leq \frac{c \Delta t}{\Delta x} \leq 1 - \xi \) (CFL conditions).