Formal Verification of a Floating-Point Elementary Function

Guillaume Melquiond

Inria Saclay–Île-de-France & LRI, Université Paris Sud, CNRS

2015-06-25

Approximating the Exponential with FP Numbers

Goal: a binary64 code that approximates exp within a few ulps.

Approximating the Exponential with FP Numbers

Goal: a binary64 code that approximates exp within a few ulps.

- Architectures only support basic operations such as $+$, $\times$, $\div$.
  - So one needs a polynomial/rational approximation of exp.
  - Effective domain: $[-710; 710]$.
  - No sane approximation on such a large domain.

Approximating the Exponential with FP Numbers

Goal: a binary64 code that approximates exp within a few ulps.

- Architectures only support basic operations such as $+$, $\times$, $\div$.
  - So one needs a polynomial/rational approximation of exp.
  - Effective domain: $[-710; 710]$.
  - No sane approximation on such a large domain.
- Cody & Waite's code (1980):
  - Clever argument reduction to $[-0.35; 0.35]$.
  - Degree-5 rational approximation of exp, suitably factored.
  - Trivial reconstruction.

Approximating the Exponential with FP Numbers

Goal: a binary64 code that approximates exp within a few ulps.

- Architectures only support basic operations such as $+$, $\times$, $\div$.
  - So one needs a polynomial/rational approximation of exp.
  - Effective domain: $[-710; 710]$.
  - No sane approximation on such a large domain.

- Cody & Waite's code (1980):
  - Clever argument reduction to $[-0.35; 0.35]$.
  - Degree-5 rational approximation of exp, suitably factored.
  - Trivial reconstruction.

Correctness condition: the relative error between `cw_exp(x)` and the mathematical value $\exp x$ is less than 2^{-51}.

Outline

Outline

Algorithm Overview and Error Analysis

$$\exp x \quad =$$

Algorithm Overview and Error Analysis

$$\exp x \quad = \quad \exp(x - k \cdot \log 2) \cdot 2^k \quad \text{with } k = \lfloor x / \log 2 \rceil_{\simeq}$$

Algorithm Overview and Error Analysis

$$\begin{aligned}\exp x &= \exp(x - k \cdot \log 2) \cdot 2^k \quad \text{with } k = \lfloor x / \log 2 \rceil_{\simeq} \\ &= \exp t \cdot \exp(-\varepsilon_t) \cdot 2^k \quad \text{with } t \simeq x - k \cdot \log 2\end{aligned}$$

Algorithm Overview and Error Analysis

$$\begin{aligned}
\exp x &= \exp(x - k \cdot \log 2) \cdot 2^k \quad \text{with } k = \lfloor x / \log 2 \rceil_{\simeq} \\
&= \exp t \cdot \exp(-\varepsilon_t) \cdot 2^k \quad \text{with } t \simeq x - k \cdot \log 2 \\
&= f(t) \cdot (1 + \varepsilon_f)^{-1} \cdot \exp(-\varepsilon_t) \cdot 2^k \quad \text{with } f \simeq \exp
\end{aligned}$$

Algorithm Overview and Error Analysis

$$\begin{aligned}
\exp x &= \exp(x - k \cdot \log 2) \cdot 2^k \quad \text{with } k = \lfloor x / \log 2 \rceil_{\simeq} \\
&= \exp t \cdot \exp(-\varepsilon_t) \cdot 2^k \quad \text{with } t \simeq x - k \cdot \log 2 \\
&= f(t) \cdot (1 + \varepsilon_f)^{-1} \cdot \exp(-\varepsilon_t) \cdot 2^k \quad \text{with } f \simeq \exp \\
&= \tilde{f}(t) \cdot (1 + \varepsilon_{\tilde{f}})^{-1} \cdot (1 + \varepsilon_f)^{-1} \cdot \exp(-\varepsilon_t) \cdot 2^k
\end{aligned}$$

Algorithm Overview and Error Analysis

$$\begin{aligned}
\exp x &= \exp(x - k \cdot \log 2) \cdot 2^k \quad \text{with } k = \lfloor x / \log 2 \rceil_{\simeq} \\
&= \exp t \cdot \exp(-\varepsilon_t) \cdot 2^k \quad \text{with } t \simeq x - k \cdot \log 2 \\
&= f(t) \cdot (1 + \varepsilon_f)^{-1} \cdot \exp(-\varepsilon_t) \cdot 2^k \quad \text{with } f \simeq \exp \\
&= \tilde{f}(t) \cdot (1 + \varepsilon_{\tilde{f}})^{-1} \cdot (1 + \varepsilon_f)^{-1} \cdot \exp(-\varepsilon_t) \cdot 2^k
\end{aligned}$$

So $\tilde{f}(t) \cdot 2^k$ approximates $\exp x$ with a relative error $\approx \varepsilon_{\tilde{f}} + \varepsilon_f + \varepsilon_t$.

Algorithm Overview and Error Analysis

$$
\begin{aligned}
\exp x &= \exp(x - k \cdot \log 2) \cdot 2^k \quad \text{with } k = \lfloor x / \log 2 \rceil_{\simeq} \\
&= \exp t \cdot \exp(-\varepsilon_t) \cdot 2^k \quad \text{with } t \simeq x - k \cdot \log 2 \\
&= f(t) \cdot (1 + \varepsilon_f)^{-1} \cdot \exp(-\varepsilon_t) \cdot 2^k \quad \text{with } f \simeq \exp \\
&= \tilde{f}(t) \cdot (1 + \varepsilon_{\tilde{f}})^{-1} \cdot (1 + \varepsilon_f)^{-1} \cdot \exp(-\varepsilon_t) \cdot 2^k
\end{aligned}
$$

So $\tilde{f}(t) \cdot 2^k$ approximates $\exp x$ with a relative error $\approx \varepsilon_{\tilde{f}} + \varepsilon_f + \varepsilon_t$.

Goal: design the function and bound the following expressions

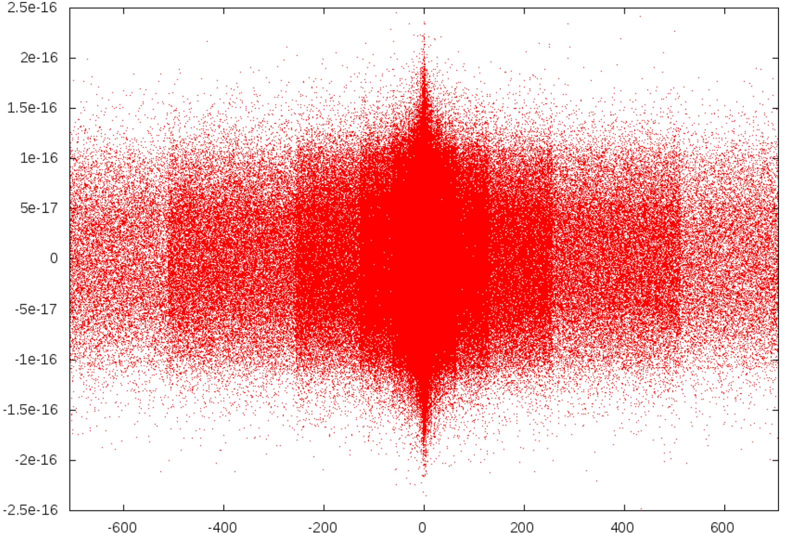
- reduced argument t, (f depends on the range of t)
- argument reduction error $\varepsilon_t = t - (x - k \cdot \log 2)$,
- relative method error $\varepsilon_f = f(t) / \exp t - 1$,
- relative round-off error $\varepsilon_{\tilde{f}} = \tilde{f}(t) / f(t) - 1$.

C Implementation

```
double cw_exp(double x)
{
  if (fabs(x) > 710.) return x < 0. ? 0. : INFINITY;
  double Log2h = 0xb.17217f7d1c00p-4;
  double Log2l = 0xf.79abc9e3b398p-48;
  double InvLog2 = 0x1.71547652b82fep0;
  double p1 = 0x1.c70e46fb3f692p-8;
  double p2 = 0x1.152a46f58dc1cp-16;
  double q1 = 0xe.38c738a128d98p-8;
  double q2 = 0x2.07f32dfbc7012p-12;

  double k = nearbyint(x * InvLog2);
  double t = x - k * Log2h - k * Log2l;
  double t2 = t * t;
  double p = 0.25 + t2 * (p1 + t2 * p2);
  double q = 0.5  + t2 * (q1 + t2 * q2);
  double f = t * (p / (q - t * p)) + 0.5;
  return ldexp(f, (int)k + 1);
}
```

Total Relative Error



Formal Proofs and Interval Arithmetic

Issue

Algorithms are intricate, so correctness proofs are error-prone.

Formal Proofs and Interval Arithmetic

Issue

Algorithms are intricate, so correctness proofs are error-prone.

Solution

Verify the algorithms using a formal system.

Formal Proofs and Interval Arithmetic

Issue

Algorithms are intricate, so correctness proofs are error-prone.

Solution

Verify the algorithms using a formal system.

Issue

Formal proofs are tedious, time-consuming, and reserved to experts.

Formal Proofs and Interval Arithmetic

Issue

Algorithms are intricate, so correctness proofs are error-prone.

Solution

Verify the algorithms using a formal system.

Issue

Formal proofs are tedious, time-consuming, and reserved to experts.

Solution

Let the proof assistant perform (parts of) the proof automatically.

Formal Proofs and Interval Arithmetic

Issue

Algorithms are intricate, so correctness proofs are error-prone.

Solution

Verify the algorithms using a formal system.

Issue

Formal proofs are tedious, time-consuming, and reserved to experts.

Solution

Let the proof assistant perform (parts of) the proof automatically.

Issue

How do you automate proofs on real and FP numbers?

Formal Proofs and Interval Arithmetic

Issue

Algorithms are intricate, so correctness proofs are error-prone.

Solution

Verify the algorithms using a formal system.

Issue

Formal proofs are tedious, time-consuming, and reserved to experts.

Solution

Let the proof assistant perform (parts of) the proof automatically.

Issue

How do you automate proofs on real and FP numbers?

Solution

Use reliable numerical methods.

Outline

Coq: a Proof Assistant

Support

- typed lambda-calculus with inductive types,
- proof verification using a "small" kernel,
- proof assistance using tactic-based backward reasoning.

Coq: a Proof Assistant

Stating and proving $\frac{ab}{ac} = \frac{b}{c}$

```
Lemma Rdiv_compat_r : (* stating the theorem *)
  forall a b c : R,
  a <> 0 -> c <> 0 -> (a*b) / (a*c) = b/c.
Proof. (* building the proof using tactics *)
  intros.
  field.
  now split.
Qed. (* verifying the resulting proof *)
```

Flocq: a Floating-Point Formalization for Coq

Support

- multi-radix (2, 10, exotic),
- multi-format (fixed-point, floating-point, exotic).
- axiomatic rounding operators (no overflow),
- computable IEEE-754 operators, including $\div$ and $\sqrt{\cdot}$,
- comprehensive library of generic theorems.

Flocq: a Floating-Point Formalization for Coq

Axiomatizing the binary64 addition

```
Definition add (x y : R) : R :=
  round radix2 (FLT_exp (-1074) 53) ZnearestE (x + y).
```

add(x, y) is the real number

- the closest to the exact sum $x + y$,
- when rounding to nearest, tie breaking to even,
- in a FP format with 53 β-digits and a minimal value β^{-1074},
- with radix $\beta = 2$.

Coq Implementation and Specification

Flocq-based implementation

```
Definition cw_exp (x : R) : R :=
  let k := nearbyint (mul x InvLog2) in
  let t := sub (sub x (mul k Log2h)) (mul k Log2l) in
  let t2:= mul t t in
  let p := add p0 (mul t2 (add p1 (mul t2 p2))) in
  let q := add q0 (mul t2 (add q1 (mul t2 q2))) in
  let f:= add (mul t (div p (sub q (mul t p)))) 1/2 in
  pow2 (Zfloor k + 1) * f.
```

Specification

```
Theorem exp_correct :
  forall x : R,
  generic_format radix2 (FLT_exp (-1074) 53) x ->
  Rabs x <= 710 ->
  Rabs ((cw_exp x - exp x) / exp x) <= 1 * pow2 (-51).
```

Intermediate Lemmas

```
Lemma argument_reduction :
  forall x : R,
  generic_format radix2 (FLT_exp (-1074) 53) x ->
  Rabs x <= 710 ->
  let k := nearbyint (mul x InvLog2) in
  let t := sub (sub x (mul k Log2h)) (mul k Log2l) in
  Rabs t <= 355 / 1024 /\
  Rabs (t - (x - k * ln 2)) <= 65537 * pow2 (-71).
```

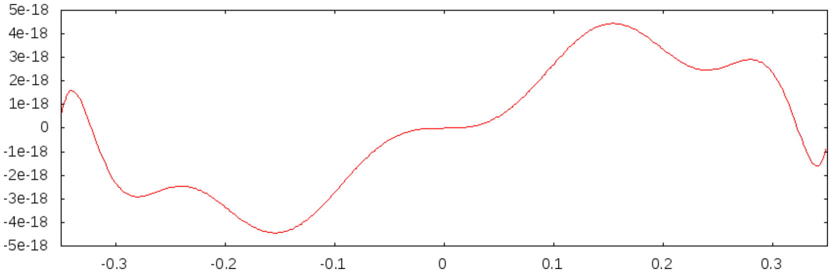
```
Lemma method_error :
  forall t : R,
  let t2 := t * t in
  let p := p0 + t2 * (p1 + t2 * p2) in
  let q := q0 + t2 * (q1 + t2 * q2) in
  let f := 2 * (t * (p / (q - t * p)) + 1/2) in
  Rabs t <= 355 / 1024 ->
  Rabs ((f - exp t) / exp t) <= 23 * pow2 (-62).
```

Outline

1. Introduction: Cody & Waite's exponential
2. Formalizing floating-point algorithms: Coq & Flocq
3. Bounding method errors: Coq.Interval
   - Tool description
   - Bounding the method error
4. Bounding round-off errors: Gappa
5. Conclusion

Intermediate Lemma: Method Error

```
Lemma method_error :
  forall t : R,
  let t2 := t * t in
  let p := p0 + t2 * (p1 + t2 * p2) in
  let q := q0 + t2 * (q1 + t2 * q2) in
  let f := 2 * (t * (p / (q - t * p)) + 1/2) in
  Rabs t <= 355 / 1024 ->
  Rabs ((f - exp t) / exp t) <= 23 * pow2 (-62).
```



Automatic Proof using Coq.Interval

Support

Quantifier-free formulas of enclosures of expressions using

- basic arithmetic operators: $+$, $-$, $\times$, $\div$, $\sqrt{\cdot}$,
- elementary functions: cos, sin, tan, arctan, exp, log.

Automatic Proof using Coq.Interval

Support

Quantifier-free formulas of enclosures of expressions using

- basic arithmetic operators: $+$, $-$, $\times$, $\div$, $\sqrt{\cdot}$,
- elementary functions: cos, sin, tan, arctan, exp, log.

Approach

Fully formalized in Coq:

- efficient multi-precision FP arithmetic,
- interval arithmetic with univariate Taylor models,
- reflexive tactic.

Bounding Errors Automatically

Naive interval arithmetic cannot compute tight bounds for

$$\frac{f(t) - \exp t}{\exp t} \in \frac{[0.7, 1.5] - [0.7, 1.5]}{[0.7, 1.5]} = \frac{[-0.8, 0.8]}{[0.7, 1.5]} \subseteq [-1.2, 1.2]$$

due to the dependency effect.

Bounding Errors Automatically

Naive interval arithmetic cannot compute tight bounds for

$$\frac{f(t) - \exp t}{\exp t} \in \frac{[0.7, 1.5] - [0.7, 1.5]}{[0.7, 1.5]} = \frac{[-0.8, 0.8]}{[0.7, 1.5]} \subseteq [-1.2, 1.2]$$

due to the dependency effect.

But one can automatically compute a polynomial P and an interval Δ such that

$$\frac{f(t) - \exp t}{\exp t} = P(t) + \delta(t) \quad \text{with } \delta(t) \in \Delta$$

and then use naive interval arithmetic to compute tight bounds for

$$P(t) + \delta(t) \in [-23 \cdot 2^{-62}, 23 \cdot 2^{-62}].$$

Outline

1. Introduction: Cody & Waite's exponential
2. Formalizing floating-point algorithms: Coq & Flocq
3. Bounding method errors: Coq.Interval
4. Bounding round-off errors: Gappa
   - Tool description
   - Round-off error
   - Argument reduction
   - User hints
5. Conclusion

Relative Round-off Error

```
double cw_exp(double x) {
  ...
  //@ assert \abs(t) <= 355. / 1024.;
  double t2 = t * t;
  double p = 0.25 + t2 * (p1 + t2 * p2);
  double q = 0.5  + t2 * (q1 + t2 * q2);
  double f = t * (p / (q - t * p)) + 0.5;
  //@ assert \abs((f - \exp(t)) / \exp(t)) <= ...;
  ...
}
```

Automatic Proof using Gappa

Support

Quantifier-free formulas of enclosures of expressions using

- binary floating-/fixed-point rounding operators,
- basic arithmetic operators: $+$, $-$, $\times$, $\div$, $\sqrt{\cdot}$.

Automatic Proof using Gappa

Support

Quantifier-free formulas of enclosures of expressions using

- binary floating-/fixed-point rounding operators,
- basic arithmetic operators: $+$, $-$, $\times$, $\div$, $\sqrt{\cdot}$.

Approach

1. symbolic proof search of relevant theorems,
2. numerical application of selected instances,
3. proof minimization and output.

Automatic Proof using Gappa

Support

Quantifier-free formulas of enclosures of expressions using

- binary floating-/fixed-point rounding operators,
- basic arithmetic operators: $+$, $-$, $\times$, $\div$, $\sqrt{\cdot}$.

Approach

1. symbolic proof search of relevant theorems,
2. numerical application of selected instances,
3. proof minimization and output.

Database of $\approx$ 150 theorems

- naive interval arithmetic,
- rewriting of errors between structurally-similar expressions.

Errors Between Structurally-similar Expressions

Let us suppose that $\tilde{u}$ and u are close, and $\tilde{v}$ and v too.
How to bound

$$\tilde{u} \cdot \tilde{v} - u \cdot v?$$

Errors Between Structurally-similar Expressions

Let us suppose that $\tilde{u}$ and u are close, and $\tilde{v}$ and v too.
How to bound

$$\tilde{u} \cdot \tilde{v} - u \cdot v?$$

Not by naive interval arithmetic due to the dependency effect.

Errors Between Structurally-similar Expressions

Let us suppose that $\tilde{u}$ and u are close, and $\tilde{v}$ and v too.
How to bound

$$\tilde{u} \cdot \tilde{v} - u \cdot v?$$

Not by naive interval arithmetic due to the dependency effect.

But it works by rewriting

$$\tilde{u} \cdot \tilde{v} - u \cdot v = (\tilde{u} - u) \cdot v + u \cdot (\tilde{v} - v) + (\tilde{u} - u) \cdot (\tilde{v} - v)$$

and then by naive interval arithmetic.

Bounding the Relative Round-off Error using Gappa

First Try

```
t2 double= t * t;
p double= 0.25 + t2 * (p1 + t2 * p2);
q double= 0.5  + t2 * (q1 + t2 * q2);
f double= t * (p / (q - t * p)) + 0.5;

Mt2 = t * t;
Mp = 0.25 + Mt2 * (p1 + Mt2 * p2);
Mq = 0.5  + Mt2 * (q1 + Mt2 * q2);
Mf = t * (Mp / (Mq - t * Mp)) + 0.5;

{ |t| <= 355b-10 -> f -/ Mf in ? }
```

Argument Reduction

How to compute $x - k \cdot \log 2$?

Naive implementation

```
double k = nearbyint(x * 0x1.71547652b82fep0);
double t = x - k * 0xb.17217f7d1cf78p-4;
```

For $x = 700$, we get $k = 1010$ and $\varepsilon_t \simeq 2^{-44.2}$.

Argument Reduction

How to compute $x - k \cdot \log 2$?

Naive implementation

```
double k = nearbyint(x * 0x1.71547652b82fep0);
double t = x - k * 0xb.17217f7d1cf78p-4;
```

For $x = 700$, we get $k = 1010$ and $\varepsilon_t \simeq 2^{-44.2}$.

Cody & Waite's trick

```
double k = nearbyint(x * 0x1.71547652b82fep0);
double Log2h = 0xb.17217f7d1cp-4; // 42 bits out of 53
double Log2l = 0xf.79abc9e3b398p-48;
double t = (x - k * Log2h) - k * Log2l;
```

For $x = 700$, we get $k = 1010$ and $\varepsilon_t \simeq 2^{-58.1}$.

Bounding Errors Automatically (1/2)

Gappa cannot compute tight bounds for

$$x - \lfloor x \cdot \texttt{InvLog2} \rceil \cdot \texttt{Log2h}$$

due to the dependency effect inherent to interval arithmetic.

Bounding Errors Automatically (1/2)

Gappa cannot compute tight bounds for

$$x - \lfloor x \cdot \texttt{InvLog2} \rceil \cdot \texttt{Log2h}$$

due to the dependency effect inherent to interval arithmetic.

But it can compute tight bounds for

$$(x \cdot \texttt{InvLog2}) \cdot \texttt{InvLog2}^{-1} - \lfloor x \cdot \texttt{InvLog2} \rceil \cdot \texttt{Log2h}$$

since it is an error between two structurally-similar expressions.

Bounding Errors Automatically (1/2)

Gappa cannot compute tight bounds for

$$x - \lfloor x \cdot \mathtt{InvLog2} \rceil \cdot \mathtt{Log2h}$$

due to the dependency effect inherent to interval arithmetic.

But it can compute tight bounds for

$$(x \cdot \mathtt{InvLog2}) \cdot \mathtt{InvLog2}^{-1} - \lfloor x \cdot \mathtt{InvLog2} \rceil \cdot \mathtt{Log2h}$$

since it is an error between two structurally-similar expressions.

User hint: $x = (x \cdot \mathtt{InvLog2}) \cdot \mathtt{InvLog2}^{-1}$.

Bounding Errors Automatically (2/2)

Gappa cannot compute tight bounds for

$$((x - k \cdot \texttt{Log2h}) - k \cdot \texttt{Log2l}) - (x - k \cdot \log 2)$$

due to the dependency effect and the use of log.

Bounding Errors Automatically (2/2)

Gappa cannot compute tight bounds for

$$((x - k \cdot \texttt{Log2h}) - k \cdot \texttt{Log2l}) - (x - k \cdot \log 2)$$

due to the dependency effect and the use of log.

But it can compute tight bounds for

$$((x - k \cdot \texttt{Log2h}) - k \cdot \texttt{Log2l}) - ((x - k \cdot \texttt{Log2h}) - k \cdot \mu)$$

since it is an error between two structurally-similar expressions, as long as the user gives some bounds on $\mu = \log 2 - \texttt{Log2h}$.

Bounding Errors Automatically (2/2)

Gappa cannot compute tight bounds for

$$((x - k \cdot \texttt{Log2h}) - k \cdot \texttt{Log2l}) - (x - k \cdot \log 2)$$

due to the dependency effect and the use of log.

But it can compute tight bounds for

$$((x - k \cdot \texttt{Log2h}) - k \cdot \texttt{Log2l}) - ((x - k \cdot \texttt{Log2h}) - k \cdot \mu)$$

since it is an error between two structurally-similar expressions, as long as the user gives some bounds on $\mu = \log 2 - \texttt{Log2h}$.

User hints: $x - k \cdot \log 2 = x - k \cdot \texttt{Log2h} - k \cdot (\log 2 - \texttt{Log2h})$ and $\texttt{Log2l} - (\log 2 - \texttt{Log2h}) \in [-2^{-102}, 0]$.

Outline

Proof Summary

- Relative method error:
  - multi-precision interval arithmetic using Taylor models,
  - fully automated proof.

Proof Summary

- Relative method error:
  - multi-precision interval arithmetic using Taylor models,
  - fully automated proof.
- Relative round-off error:
  - naive interval arithmetic + forward error analysis,
  - fully automated proof.

Proof Summary

- Relative method error:
  - multi-precision interval arithmetic using Taylor models,
  - fully automated proof.
- Relative round-off error:
  - naive interval arithmetic + forward error analysis,
  - fully automated proof.
- Argument reduction (tricky code):
  - naive interval arithmetic + forward error analysis,
  - partly automated proof, user interactions:
    - a case analysis for excluding $x \simeq 0$,
    - two trivial identities, (developer knowledge)
    - some bounds on $\log 2$ using interval arithmetic.

Proof Summary

- Relative method error:
  - multi-precision interval arithmetic using Taylor models,
  - fully automated proof.
- Relative round-off error:
  - naive interval arithmetic + forward error analysis,
  - fully automated proof.
- Argument reduction (tricky code):
  - naive interval arithmetic + forward error analysis,
  - partly automated proof, user interactions:
    - a case analysis for excluding $x \simeq 0$,
    - two trivial identities, (developer knowledge)
    - some bounds on $\log 2$ using interval arithmetic.
- Result reconstruction and total error:
  - straightforward manual proof + interval arithmetic.

Lies, Lies, and More Lies

The theorem is formally proved, but what does it actually state?

Lies, Lies, and More Lies

The theorem is formally proved, but what does it actually state?

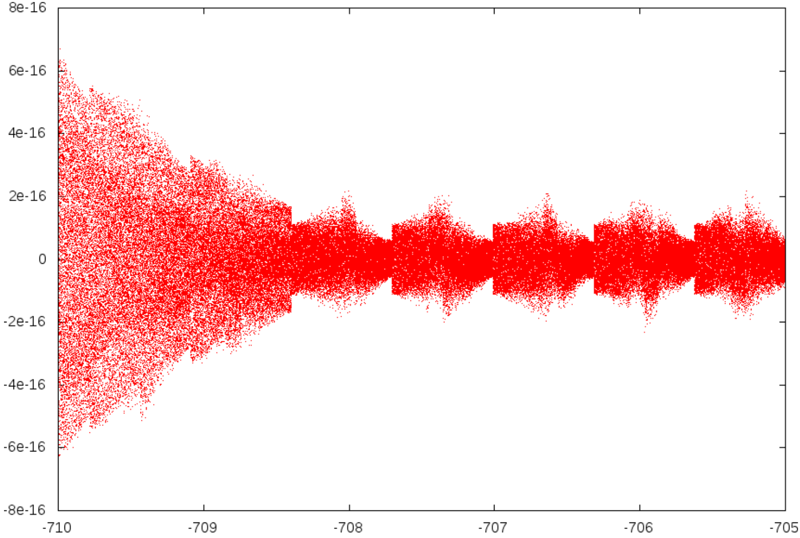
- Flocq's abstract formats have no upper bound
  $\Rightarrow$ need for additional proofs to ensure no overflow occurs.

Lies, Lies, and More Lies

The theorem is formally proved, but what does it actually state?

- Flocq's abstract formats have no upper bound
  $\Rightarrow$ need for additional proofs to ensure no overflow occurs.
- Result reconstruction is proved over real numbers
  $\Rightarrow$ on subnormal numbers, the relative error explodes.

Total Relative Error (Subnormal Results)



Questions?

Thanks to dedicated automations, formally proving the correctness of floating-point algorithms is now accessible to non-specialists.

Flocq:	`http://flocq.gforge.inria.fr/`
Gappa:	`http://gappa.gforge.inria.fr/`
Coq.Interval:	`http://coq-interval.gforge.inria.fr/`