

Vérification de bout en bout d'une fonction mathématique

Paul Geneau de Lamarlière^{1, 2}

JFLA 2025, 30 janvier

¹ MITSUBISHI ELECTRIC R&D CENTRE EUROPE

² UNIVERSITE PARIS-SACLAY, CNRS, ENS PARIS-SACLAY, INRIA, LMF

Bibliothèques mathématiques

Une bibliothèque mathématique fournit des approximations flottantes de fonctions mathématiques (exp, cos, sin, ...).

Utilisation

- Ingénierie
- Prédications météorologiques
- ...

Vérification automatique d'encadrements d'expressions réelles avec CoqInterval

```
Goal 2487 <= exp (1900 / 243) <= 2488.
```

```
Proof. interval. Qed. (* Quasi-instantane *)
```

```
Goal Rabs (RInt (fun x => sin (x + exp x)) 0 8 - 0.347400) < 0.000001.
```

```
Proof. integral with (i_fuel 10000). Qed. (* Quelques secondes *)
```

Vérification automatique d'encadrements d'expressions réelles avec CoqInterval

```
Goal 2487 <= exp (1900 / 243) <= 2488.  
Proof. interval. Qed. (* Quasi-instantane *)  
  
Goal Rabs (RInt (fun x => sin (x + exp x)) 0 8 - 0.347400) < 0.000001.  
Proof. integral with (i_fuel 10000). Qed. (* Quelques secondes *)
```

Difficulté

On veut que les fonctions qui calculent des encadrements de fonctions mathématiques soient :

- rapides
- précises
- formellement vérifiées

Fonction puissance $\text{pow} : (x, y) \mapsto x^y$

- Hubrecht, Jeannerod, Zimmermann, Rideau, Théry.
Towards a correctly-rounded and fast power function in binary64 arithmetic. 2024.

Fonction puissance $\text{pow} : (x, y) \mapsto x^y$

- Hubrecht, Jeannerod, Zimmermann, Rideau, Théry.
Towards a correctly-rounded and fast power function in binary64 arithmetic. 2024.

Difficulté

Correction : $\forall x, y, |\text{pow}(x, y) - x^y| \leq \delta \cdot x^y$

- La preuve est intrinsèquement longue.
- Preuve partielle en Coq : >2000 lignes

bounded as follows: $E_0 = |\epsilon_7| \leq 2^{-55}$, $E_1 \leq v'|w_\ell| + |\epsilon_6| < (2^{-1.58058} + 2^{-54}) \cdot 2^{-68} + 2^{-70} < 2^{-68.775}$, $\tilde{E}_3 \leq u'|w_\ell| + |\epsilon_3| + |\epsilon_5| < (2^{-2.31707} + 2^{-55}) \cdot 2^{-68} + 2^{-54} + 2^{-54} < 2^{-52.999}$, $\tilde{E}_5 \leq t|w_\ell| + |\epsilon_2| + |\epsilon_4| < (2^{-2.8022} + 2^{-55}) \cdot 2^{-68} + 2^{-55} + 2^{-55} < 2^{-53.999}$ and $\tilde{E}_7 = |\epsilon_1| < 2^{-55}$. All this then leads

(la preuve complète représente environ 10 pages)

Fonction puissance $\text{pow} : (x, y) \mapsto x^y$

- Hubrecht, Jeannerod, Zimmermann, Rideau, Théry.
Towards a correctly-rounded and fast power function in binary64 arithmetic. 2024.

Difficulté

Correction : $\forall x, y, |\text{pow}(x, y) - x^y| \leq \delta \cdot x^y$

- La preuve est intrinsèquement longue.
- Preuve partielle en Coq : >2000 lignes

bounded as follows: $E_0 = |\epsilon_7| \leq 2^{-68}$, $E_1 \leq v'|w_\ell| + |\epsilon_6| < (2^{-1.58058} + 2^{-54}) \cdot 2^{-68} + 2^{-70} < 2^{-68.775}$, $\tilde{E}_3 \leq u'|w_\ell| + |\epsilon_3| + |\epsilon_5| < (2^{-2.31707} + 2^{-55}) \cdot 2^{-68} + 2^{-54} + 2^{-54} < 2^{-52.999}$, $\tilde{E}_5 \leq t|w_\ell| + |\epsilon_2| + |\epsilon_4| < (2^{-2.8022} + 2^{-55}) \cdot 2^{-68} + 2^{-55} + 2^{-55} < 2^{-53.999}$ and $\tilde{E}_7 = |\epsilon_1| < 2^{-55}$. All this then leads

(la preuve complète représente environ 10 pages)

On ne peut pas simplifier les preuves, mais on peut faciliter leur construction.

- Exponentielle rapide pour CoqInterval :
 - Calcule avec les flottants natifs
 - Utilise les méthodes de l'état de l'art
- Méthodologie pour la vérification de fonctions de bibliothèques mathématiques :
 - Gestion des comportements exceptionnels
 - Support pour des astuces connues en arithmétique flottante
 - Preuve Coq/Rocq concise (tactiques dédiées)

Méthode d'approximation

- Réduction d'argument : $\exp(x) = \exp(x - k \cdot \ln 2/64) \cdot 2^{k/64}$
 - Entier le plus proche k calculé comme $f + 0x1.8p52 - 0x1.8p52$ (partie entière rapide)
 - Tableau pré-calculé de taille 64 contenant $2^{i/64}$ pour $0 \leq i \leq 63$
- Polynôme de degré 5 à coefficients flottants

```
1 Definition iexp x :=
2   if x < -0x1.74385446d71c4p9 then (0, 0x1.p-1074) else
3   if x > 0x1.62e42fefa39efp9 then (0x1.fffffffffff2ap1023, infinity) else
4   let k' := x * invLog2_64 + 0x1.8p52 in (* partie entière rapide *)
5   let k := k' - 0x1.8p52 in (* *)
6   let t := (x - k * log2_64h) - k * log2_64l in
7   let y := t * (p1 + t * (p2 + t * (... + t * p5))) in (* approximation polynomiale *)
8   let ki := (int_of_float k' - 0x1800000000000000)%uint63 in
9   let p0 := cst.(ki land 63) in
10  let d := 0x1.25p-57 in (* compensation *)
11  let lb := p0 + (p0 * y - d) in let ub := p0 + (p0 * y + d) in
12  next_down (ldexp lb (ki asr 6)), next_up (ldexp ub (ki asr 6))
```

Correction de iexp

Soient x un flottant double et $(\underline{y}, \overline{y})$ le résultat de $\text{iexp}(x)$. Si x est fini, alors :

$$\underline{y} \leq \exp x \leq \overline{y}$$

La preuve papier est longue et complexe.

Ce que nous aimerions faciliter

- Gestion des comportements exceptionnels
- Utilisation d'astuces connues en arithmétique flottante (ex. partie entière rapide)
- Schémas de preuve récurrents

Difficulté

Les fonctions des bibliothèques mathématiques calculent des expressions flottantes, mais nous aimerions raisonner sur des expressions réelles.

La norme IEEE 754

“Every operation shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result.”

Pour a et b finis, $a \oplus b$ est fini $\implies a \oplus b = \circ(a + b)$

Langage d'expressions abstraites et interprétations

- $e ::= c \mid x \mid e + e \mid e \times e \mid \dots$
- $\begin{cases} \llbracket e_1 + e_2 \rrbracket_{\text{flt}} := \llbracket e_1 \rrbracket_{\text{flt}} \oplus \llbracket e_2 \rrbracket_{\text{flt}} & \text{(flottants)} \\ \llbracket e_1 + e_2 \rrbracket_{\text{rnd}} := \circ(\llbracket e_1 \rrbracket_{\text{rnd}} + \llbracket e_2 \rrbracket_{\text{rnd}}) & \text{(réels arrondis)} \end{cases}$

Expressions qui se "comportent bien"

$$WB(u/v) \triangleq \left\{ \begin{array}{ll} WB(u) \wedge WB(v) & \\ \wedge \llbracket v \rrbracket_{\text{rnd}} \neq 0 & \text{(pas de division par 0)} \\ \wedge |\circ(\llbracket u \rrbracket_{\text{rnd}} / \llbracket v \rrbracket_{\text{rnd}})| < 2^{1024} & \text{(pas de dépassement de capacité)} \end{array} \right.$$

Théorème (correspondance)

$$WB(e) \Rightarrow \llbracket e \rrbracket_{\text{flt}} \text{ est fini} \wedge \llbracket e \rrbracket_{\text{flt}} = \llbracket e \rrbracket_{\text{rnd}}$$

Preuve automatisée de $WB(e)$

Tactique dédiée `simplify_wb` pour prouver automatiquement un maximum de clauses de $WB(e)$ en utilisant l'arithmétique d'intervalle.

Exemple: calcul de l'entier le plus proche

Un court algorithme pour calculer l'entier le plus proche d'un flottant :

```
4 let k' := x *. invLog2_64 +. 0x1.8p52 in
5 let k  := k' -. 0x1.8p52 in
```

Notre méthodologie ne permet pas de simplifier la preuve de fonctions qui utilisent cette astuce.

Exemple: calcul de l'entier le plus proche

Un court algorithme pour calculer l'entier le plus proche d'un flottant :

```
4 let k' := x *. invLog2_64 +. 0x1.8p52 in
5 let k  := k' -. 0x1.8p52 in
```

Notre méthodologie ne permet pas de simplifier la preuve de fonctions qui utilisent cette astuce.

Solution: ajout d'une macro-opération au langage

- $$\begin{cases} \llbracket \text{FastNearbyint } e \rrbracket_{\text{flt}} &:= \llbracket e \rrbracket_{\text{flt}} \oplus 3 \cdot 2^{51} \ominus 3 \cdot 2^{51} \\ \llbracket \text{FastNearbyint } e \rrbracket_{\text{rnd}} &:= \text{nearbyint}(\llbracket e \rrbracket_{\text{rnd}}) \end{cases}$$
- $\text{WB}(\text{FastNearbyint } e) = \text{WB}(e) \wedge \llbracket e \rrbracket_{\text{rnd}} \leq 2^{51}$

Observation

On fait souvent, dans cet ordre :

- énoncer une propriété sur une expression flottante $\llbracket e \rrbracket_{\text{ft}}$
- appliquer le théorème de correspondance avec e
- prouver $\text{WB}(e)$ automatiquement avec `simplify_wb`

→ Tactique dédiée `assert_float`.

Exemple

```
set (k0 := FastNearbyint (Op MUL (Var 0) (BinFl InvLog2_64))).  
change (_ - 0x1.8p52)%float with (evalPrim k0 (x, tt)).  
assert_float (fun (k : R) => -68736 <= k <= 65536). (* prouve WB(k0) *)  
{ rewrite round_FIX_IZR. interval. }
```

Bilan

- Intervalle de largeur 2 ulps dans 98% des cas (sinon 3 ulps)
- 840 000 évaluations par seconde

Difficile de faire mieux

- Calculer en Coq/Rocq est intrinsèquement lent
- L'algorithme traduit en C a des performances comparables à l'état de l'art

Quelques chiffres

- 450 lignes de preuve
- Tactique `assert_float` utilisée 8 fois

Preuve disponible dans CoqInterval : <https://coqinterval.gitlabpages.inria.fr/>

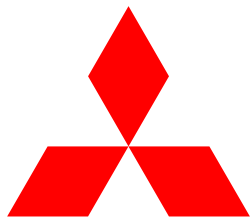
Ce qu'il faut retenir

```
assert_float (fun y => Rabs (1 + y - exp t) <= 11 * 2(-62)) .  
{ interval with (i_taylor t, i_bisect t, i_prec 80). }
```

Ce petit script Coq/Rocq prouve :

- l'absence de comportements exceptionnels dans l'approximation polynomiale
- que l'erreur totale est bornée par $11 \cdot 2^{-62}$

- Utiliser un langage plus générique pour faciliter l'ajout de nouvelles (macro-)opérations
- Réécrire le code des autres fonctions de CoqInterval.
- Vérifier des fonctions de la bibliothèque standard d'autres langages



**MITSUBISHI
ELECTRIC**

Changes for the Better