Flottants primitifs dans Coq

Érik Martin-Dorel, Pierre Roux

Flottants primitifs dans Coq

Érik Martin-Dorel[1], Pierre Roux[2]

[1] Lab. IRIT, Université de Toulouse, CNRS, Université Paul Sabatier, Toulouse, France
[2] ONERA / DTIS, Université de Toulouse, Toulouse, France

Résumé

Certaines preuves mathématiques font intervenir d'important calculs, par exemple le théorème des quatre couleurs, le théorème de Hales sur l'empilement optimal des sphères (ex-conjecture de Kepler) ou l'arithmétique d'intervalles. Pour les calculs numériques, l'arithmétique à virgule flottante est très appréciée pour son efficacité malgré l'introduction d'erreurs d'arrondi. Des garanties formelles peuvent toutefois être obtenues en se basant sur le standard IEEE 754, qui spécifie précisément l'arithmétique à virgule flottante, et un assistant de preuve tel que Coq, qui offre des fonctionnalités de calcul efficace.

Coq propose de longue date une interface aux entiers machine mais jusqu'à récemment, l'arithmétique flottante était au mieux émulée grâce à ces entiers plutôt que d'utiliser les flottants machine, au prix d'un ralentissement de deux à trois ordres de grandeur. Cette proposition de démonstration vise à présenter l'interface vers les flottants machines disponibles dans Coq depuis sa version 8.11. Après une rapide présentation des choix techniques effectués, on présentera l'interface retenue, et en particulier les questions de saisie et d'affichage de constantes. Enfin sera évoquée l'utilisation de cette fonctionnalité dans les bibliothèques ValidSDP et Coq.Interval.

1 Motivation

La preuve de certains résultats mathématiques peut faire appel à des calculs numériques de telle sorte que la confiance en ces preuves nécessite une confiance en les calculs numériques eux-mêmes. Ainsi, pour être capable d'effectuer efficacement ce genre de preuves dans un assistant de preuve comme Coq, l'outil doit avoir des capacités de calcul numérique efficace.

L'arithmétique à virgule flottante est largement utilisée, en particulier pour son efficacité grâce à son implémentation matérielle. Toutefois, elle ne donne généralement pas des résultats exacts puisqu'elle introduit des erreurs d'arrondi. Des preuves rigoureuses peuvent néanmoins être obtenues en bornant ces erreurs d'arrondi. Il y a ainsi un intérêt évident à disposer d'un accès correct et efficace aux opérations flottantes du processeur au sein de Coq.

On donne ci-dessous quelques exemples de preuves mettant en jeu des calculs flottants. Considérons la preuve qu'un nombre réel donné $a \in \mathbb{R}$ est positif. On peut exhiber un autre nombre réel r tel que $a = r^2$ et appliquer le lemme garantissant que tout carré d'un nombre réel est positif. Typiquement, on peut utiliser la racine carré $\sqrt{a}$. Une méthode similaire peut être appliquée pour prouver qu'une matrice $A \in \mathbb{R}^{n\times n}$ est semi-définie positive [1] puisqu'on peut exhiber une matrice R tel que [2] $A = R^T R$. Une telle matrice peut être calculée en utilisant un algorithme appelé la décomposition de Cholesky, présenté sur la Figure 1. L'algorithme réussit, en n'effectuant ni une racine négative ni une division par zéro, dès que A est définie positive [3].

Quand elle est évaluée en arithmétique à virgule flottante, l'égalité exacte $A = R^T R$ est perdue, mais il reste possible de borner l'accumulation des erreurs d'arrondi lors de la décomposition de Cholesky de telle sorte qu'on obtient le Théorème 1.1 suivant, modulo quelques préconditions qui sont typiquement vérifiées en pratique, mais omises ici par souci de concision (ce résultat ayant été formellement prouvé en Coq dans un travail antérieur [Rou16], et disponible dans la bibliothèque libValidSDP [4]).

$R := 0$;
for j **from** 1 **to** n **do**
  for i **from** 1 **to** $j-1$ **do**
    $R_{i,j} := \left(A_{i,j} - \Sigma_{k=1}^{i-1} R_{k,i} R_{k,j}\right) / R_{i,i}$;
  end for
  $R_{j,j} := \sqrt{M_{j,j} - \Sigma_{k=1}^{j-1} {R_{k,j}}^2}$;
end for

Figure 1: Décomposition de Cholesky : pour $A \in \mathbb{R}^{n\times n}$, tente de calculer R tel que $A = R^T R$.

1. Une matrice $A \in \mathbb{R}^{n\times n}$ est dite semi-définie positive si pour tout $x \in \mathbb{R}^n$, $x^T A\, x \geq 0$.
2. Puisque, si $A = R^T R$, on a $x^T A\, x = x^T \left(R^T R\right) x = (Rx)^T (Rx) = \|Rx\|^2 \geq 0$.
3. Une matrice $A \in \mathbb{R}^{n\times n}$ est dite définie positive si pour tout $x \in \mathbb{R}^n \setminus \{0\}$, $x^T A\, x > 0$.
4. https://github.com/validsdp/validsdp/blob/v0.7.0/libvalidsdp/cholesky.v#L646

Theorem 1.1 (Corollaire 2.4 in [Rum06]). *Pour $A \in \mathbb{R}^{n\times n}$, en définissant $c := \frac{(n+1)\epsilon}{1-2(n+1)\epsilon}\,\mathrm{tr}(A) + 4n\,(2(n+1) + \max_i A_{i,i})\,\eta$, si la décomposition de Cholesky flottante réussit sur $A - c\,I$, alors A est définie positive. ϵ et η sont des constantes dépendantes du format flottant utilisé.*

Ainsi, une implémentation efficace de l'arithmétique à virgule flottante dans un assistant de preuve mène à des preuves efficaces de positivité de matrices. Cela peut avoir de multiples applications, comme prouver que des polynômes sont positifs en les exprimant comme des sommes de carrés [MR17], ce qui peut être utilisé dans une preuve de la conjecture de Kepler [MAGW15].

L'arithmétique d'intervalles constitue un autre exemple de preuve mettant en jeu des calculs numériques. Des intervalles enveloppant peuvent être facilement calculés en arithmétique à vigule flottante en utilisant les arrondis dirigés, vers $\pm\infty$. La bibliothèque Coq.Interval [MDM16] implémente l'arithmétique d'intervalles et peut bénéficier de calculs flottants efficaces.

Plus généralement, de nombreux résultats sur les méthodes numériques rigoureuses [Rum10] pourraient avoir des implémentations formelles efficaces si une arithmétique flottante efficace est disponible dans les assistants de preuve.

Coq dispose d'un support intégré pour le calcul, qui peut être utilisé dans les preuves et de récents progrès ont été faits pour fournir des entiers machines 63 bits. Coq dispose maintenant depuis sa version 8.11 d'un interface similaire vers les flottants machine « double-précision ». [5] Pour de plus amples détails sur l'implémentation, l'extension de la base de confiance et une évaluation des performances, on pourra se référer au papier original [BMR19].

2 Démonstration

On effectuera une démonstration de cette nouvelle fonctionnalité en commençant par présenter la lecture et l'affichage de constantes (qui ont été améliorés depuis [BMR19]) et les compromis associés entre lisibilité de l'écriture décimale pour l'utilisateur humain et exactitude de l'écriture binaire [6]. On présentera ensuite les différentes opérations disponibles, tant les opérations de calcul flottant que de conversion avec les entiers, puis on étudiera la spécification de ces opérations, en particulier le lien avec la bibliothèque Flocq [BM11]. Enfin, deux exemples d'utilisation seront brièvement présentés : les bibliothèques Coq.Interval et ValidSDP [7].

Références

[BM11] Sylvie Boldo and Guillaume Melquiond. Flocq : A unified library for proving floating-point algorithms in coq. In Elisardo Antelo, David Hough, and Paolo Ienne, editors, *ARITH 2011, Tübingen, Germany, 25-27 July 2011*, pages 243–252. IEEE Computer Society, 2011. doi:10.1109/ARITH.2011.40.

[BMR19] Guillaume Bertholon, Érik Martin-Dorel, and Pierre Roux. Primitive Floats in Coq. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 7 :1–7 :20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ITP.2019.7.

[MAGW15] Victor Magron, Xavier Allamigeon, Stéphane Gaubert, and Benjamin Werner. Formal proofs for nonlinear optimization. *Journal of Formalized Reasoning*, 8(1) :1–24, 2015. URL : http://jfr.unibo.it/article/view/4319.

[MDM16] Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *Journal of Automated Reasoning*, 57(3) :187–217, October 2016. doi:10.1007/s10817-015-9350-4.

[MR17] Érik Martin-Dorel and Pierre Roux. A reflexive tactic for polynomial positivity using numerical solvers and floating-point computations. In Yves Bertot and Viktor Vafeiadis, editors, *CPP 2017, Paris, France, January 16-17, 2017*, pages 90–99. ACM, 2017. doi:10.1145/3018610.3018622.

[Rou16] Pierre Roux. Formal Proofs of Rounding Error Bounds - With Application to an Automatic Positive Definiteness Check. *J. Autom. Reasoning*, 57(2) :135–156, 2016. doi:10.1007/s10817-015-9339-z.

[Rum06] Siegfried M. Rump. Verification of positive definiteness. *BIT Numerical Mathematics*, 46 :433–452, 2006.

[Rum10] Siegfried M. Rump. Verification methods : Rigorous results using floating-point arithmetic. *Acta Numerica*, 19 :287–449, 2010.

5. https://github.com/coq/coq/pull/9867

6. Certains nombres décimaux n'ont pas de représentation binaire finie et si toute valeur binaire à une représentation décimale, elle peut être trop grande pour être utilisable en pratique.

7. Preuves de positivité de matrice suivant le principe exposé ci dessus.

A Fichier Coq de la démonstration

```
(** Démonstration des flottants primitifs

   (nécessite Coq >= 8.11 (démo réalisée avec master) *)

Require Import ZArith Int63.

(** * Un module de la librairie standard *)

(** On importe le module Floats de la librairie standard de Coq *)
Require Import Floats.

(** on obtient un type des flottants primitifs *)
Check float.

(** ** Constantes *)

(** les constantes sont interprétées comme
    le flottant le plus proche *)
Open Scope float_scope.

Check 0.5.

(** avec warning si un arrondi est nécessaire *)
Check 0.1.

(** l'impression se fait avec 17 décimales, ce qui garantit
    que (parse o print) est injectif *)

(** néanmoins, ça peut donner des choses un peu étranges *)
Goal 0.99999999999999995 = 1.
Proof. reflexivity. Qed.

(** on peut voir la valeur exacte en hexadécimal *)
Set Printing All.
Check 0.1.
Unset Printing All.

(** on a trois valeurs spéciales *)
Check nan.
Check infinity.
Check neg_infinity.

(** et le 0 est signé *)
Check 0.
Check -0.

Goal 0 = -0.
Proof. Fail reflexivity. Abort.

(** mais pas les NaNs (nan est unique) *)
Goal nan = -nan.
Proof. reflexivity. Qed.

(** ** Opérations primitives *)

(** *** Opérations arithmétiques *)

(** on dispose des opération arithmétiques standard
    (toutes en arrondi au plus proche (tie to even)) *)
```

```
Eval compute in - 0.5.
Eval compute in abs (- 0.5).
Eval compute in 1 + 0.5.
Eval compute in 1 - 0.5.
Eval compute in 0.5 * 3.
Eval compute in 3 / 2.
Eval compute in 1 / 0.
Eval compute in 1 / (-0).
Eval compute in sqrt 2.
Eval compute in sqrt (-2).

(** *** Comparaison *)

Eval compute in 0.5 ?= 1.
Eval compute in 0.5 ?= 0.5.
Eval compute in infinity ?= 0.5.
Eval compute in 0 ?= (-0).
Eval compute in 1 ?= nan.

(** et les fonctions booléennes *)
Eval compute in 0.5 <=? 1.
Eval compute in 1 <? 1.
Eval compute in 1 =? 1.

(** *** "Destructeurs" *)

(** - classe *)
Eval compute in classify 0.5.
Print float_class.

(** - mantisse, exposant (int63 non signé, donc décalé de 2101) *)
Check frshiftexp.
Definition m_e := Eval compute in frshiftexp 15.
Print m_e.
Eval compute in 0.9375 * 16. (** 16 = 2^4 = 2^(2105 - 2101) *)

(** - mantisse comme un int63 *)
Eval compute in normfr_mantissa 0.9375.

(** - quelques fonctions pratiques (implémentées en Coq) *)
Eval compute in is_nan 0.
Eval compute in is_zero 0.
Eval compute in is_infinity 0.
Eval compute in get_sign (-0).
Eval compute in frexp 15. (** frshiftexp dans Z au lieu de int63,
                              sans décalage *)
Eval compute in ulp 15. (** Unit in Last Place *)

(** *** "Constructeurs" *)

(** - à partir d'un entier *)
Eval compute in of_int63 15.

(** - décalage d'exposant (décalé de 2101) *)
Check ldshiftexp.
Eval compute in ldshiftexp 10 2103. (** 10 * 2^(2103 - 2101) = 10 * 2^2 *)
Eval compute in ldshiftexp (fst m_e) (snd m_e).

(** - ldexp : ldexp sur Z au lieu de int63, sans décalage *)
Eval compute in ldexp 10 2.
```

```
(** *** Prédécesseur et successeur *)

(** pour implémenter l'arithmétique d'intervalle *)
Eval compute in next_up 1.
Eval compute in next_down 1.
Eval compute in next_up neg_infinity.
Eval compute in next_down infinity.

(* et si le temps le permet, remarque sur
   quelques valeurs flottantes remarquables *)
Eval compute in next_up 0.
Eval compute in classify (next_up 0).
Eval compute in next_up 1 - 1.
Eval compute in classify (next_down infinity).

(** tout est implémenté aussi avec vm_compute et native_compute *)
Eval vm_compute in 1 + 0.5.
Eval native_compute in 1 + 0.5.

(** ** Spécification *)

(** 1. On dispose d'un type spec_float *)
Print spec_float.

(** 2. de fonctions float <-> spec_float *)
Check Prim2SF.
Check SF2Prim.

(** 3. d'une implémentation de chaque fonction *)
Print SFopp.

(** 4. et d'un axiome de correction *)
Check opp_spec.
Check add_spec.

(** Cette spécification est extraite de la librairie Flocq.
    C'est un extrait minimal (seulement 400 lignes)
    mais peu commode. En pratique on utilisera Flocq (>= 3.3)
    qui fait le lien avec les réels de la librairie standard. *)
Require Import Reals Flocq.IEEE754.BinarySingleNaN Flocq.IEEE754.PrimFloat.

Check Prim2B.
Check B2R.
Check add_equiv.
Check Bplus_correct.

(** * Exemples d'utilisation *)

(** ** Librairie Coq.Interval: https://gitlab.inria.fr/coqinterval/interval *)

Require Import Coquelicot.Coquelicot Interval.Tactic.

Lemma Rump_Tucker :
  Rabs (RInt (fun x => sin (x + exp x)) 0 8 - 0.3474) <= 0.1.
Proof.
(** flottants primitifs : 2.3 s *)
Time integral with (i_degree 6, i_fuel 1000).
Undo.
(** flottants émulés (avec bigZ) : 36 s *)
Time integral with (i_degree 6, i_fuel 1000, i_prec 53).
Qed.
```

```
(** ** Librairie ValidSDP: https://github.com/validsdp/validsdp *)

(** une "grosse" matrice *)
Require Import matrice.
Check A.

(** de taille 120x120 *)
Eval compute in length A.

(** on prouve qu'elle est semi-définie positive *)

Require Import ValidSDP.posdef_check.

(** d'abord avec des flottants émulés (bigZ) *)
Lemma with_bigint : posdef_seqF A.
Time posdef_check.
Qed.

Print Assumptions with_bigint.

(* Int63.mul : Int63.int -> Int63.int -> Int63.int *)
(* Int63.mul_spec : ∀x y : Int63.int, Int63.to_Z (Int63.mul x y)
   = BinInt.Z.modulo (BinInt.Z.mul (Int63.to_Z x) (Int63.to_Z y)) Int63.wB *)

(** puis avec des flottants primitifs *)
Lemma with_prim_float : posdef_seqF A.
Time primitive_posdef_check.
Qed.

Print Assumptions with_prim_float.

(* mul_spec : ∀x y : float, Prim2SF (x * y)
   = SF64mul (Prim2SF x) (Prim2SF y) *)
```