DES ASSISTANTS DE PREUVES POUR L'ANALYSE RÉELLE ET NUMÉRIQUE; UN FOCUS SUR ROCQ: DE LA RECHERCHE À L'ENSEIGNEMENT ET VICE VERSA

par

Micaela Mayero

Résumé. — Les outils de preuves formelles, issus des recherches en logique, ont pris de l'ampleur ces dernières décennies. Nous présenterons succinctement ces outils, leurs évolutions ainsi que leurs fondements théoriques.
Les preuves formelles sont utilisées et développées à la fois dans le domaine de l'enseignement, principalement en mathématiques et en informatique, et de la recherche, et ces dernières années, parfois de manière coordonnée.
Nous présenterons ces deux aspects en faisant un focus sur l'analyse réelle et l'analyse numérique en Rocq.
Nous nous appuierons régulièrement sur des exemples.

Table des matières

Mots clefs. — Assistants de preuve, Rocq, formalisation des mathématiques, formalisation de l'analyse réelle, analyse numérique, bibliothèque formelle, enseignement des mathématiques.

PARTIE I. INTRODUCTION AUX PREUVES FORMELLES ET À LA FORMALISATION

Cette partie se veut une introduction aux outils de preuves formelles. Nous commencerons par revenir sur leur origine, en les replaçant dans l'histoire de la logique, discipline dont ils sont directement issues. En effet, les preuves formelles reposent sur des systèmes logiques, construits pour éliminer toute ambiguïté du raisonnement mathématique. Nous présenterons les grandes étapes de l'émergence de la logique comme fondement des mathématiques et nous verrons que ces outils, considérés maintenant comme faisant partie de l'informatique, sont relativement récents à l'échelle de l'histoire des mathématiques. Cette perspective historique mettra en lumière le décalage entre la pratique classique des mathématiques et leur formalisation complète.

Dans un second temps, nous aborderons la description de l'assistant d'aide à la preuve Rocq, qui sera accompagnée d'exemples concrets, simples, inspirés de contenus issus de l'enseignement.

1. Des origines philosophiques de la logique aux preuves formelles

La logique est la science du raisonnement (Larousse). Née dans la philosophie antique, elle est progressivement devenue une discipline mathématique formelle avant de jouer un rôle central dans les fondements de l'informatique moderne. De ce fait, l'histoire de la logique

est étroitement liée à celle des mathématiques. Elle illustre la transformation du raisonnement en un objet formel pouvant être étudié, manipulé et même vérifié par des machines.

1.1. Les origines antiques de la logique. — Bien avant que la logique ne devienne une science, des civilisations d'Égypte et de Mésopotamie (-2500) développaient des mathématiques pratiques pour l'arpentage ou le commerce, mais sans en théoriser les fondements.

Les premières réflexions sur le raisonnement apparaissent dans plusieurs traditions philosophiques de civilisations anciennes.

Dans la tradition occidentale, c'est en Grèce antique que naît un questionnement sur le raisonnement lui-même où la logique devient une discipline systématique avec Aristote (IV^{e} siècle avant notre ère). Dans l'*Organon*[**Ari50**], Aristote développe la logique syllogistique : une forme de raisonnement déductif où, de deux propositions dites prémisses, on tire une conclusion de manière nécessaire. Son exemple célèbre : "Tous les hommes sont mortels ; or Socrate est un homme ; donc Socrate est mortel." Ce système constitue la première théorie formelle du raisonnement déductif. À partir du III^{e} siècle avant notre ère, les Stoïciens introduisent le terme même de "logique". Avec des penseurs comme Chrysippe de Soles, ils développent une approche complémentaire de celle d'Aristote, une "logique des propositions" qui s'intéresse aux connecteurs logiques comme "si... alors" (implication) et "ou" (disjonction) [**Gou05**].

En Inde, c'est la tradition du Nyāya, avec son exposé le plus ancien constitué par le Nyāya-sūtra[**Vid90**] de Gotama (V^{e} siècle apr. J.-C.- les dates varient entre 200 et 450) qui systématise la logique indienne qui avait été élaborée jusqu'alors. Le syllogisme indien est semblable à celui des Grecs, à cette différence près qu'un terme supplémentaire est ajouté à la fin sous la forme d'un exemple concret.

Pendant plus de mille ans, la logique aristotélicienne est transmise, commentée et enseignée, d'abord dans le monde arabo-musulman (avec des savants comme Avicenne), puis dans les universités européennes à partir du XII^{e} siècle. Les logiciens médiévaux, comme Pierre d'Espagne ou Jean Buridan, affinent la théorie sémantique et étudient des paradoxes, comme celui du Menteur ("Cette phrase est fausse"), hérité de l'Antiquité. Les philosophes scolastiques (par

exemple Guillaume d'Ockham, 1285-1347) approfondissent l'analyse logique, notamment dans l'étude des conséquences logiques et de la structure des propositions.

1.2. La naissance de la logique mathématique. — Une étape importante se produit au XVII^{e} siècle avec le projet de Gottfried Wilhelm Leibniz d'un calcul universel du raisonnement (*calculus ratiocinator*) [**Lei66**]. Il imagine un langage symbolique permettant de résoudre les disputes intellectuelles par le calcul.

Au XIX^{e} siècle, la logique devient véritablement mathématique. George Boole introduit en 1854 une algèbre des propositions logiques dans *An Investigation of the Laws of Thought* [**Boo54**]. Cette algèbre permet de manipuler les propositions logiques comme des objets mathématiques et constitue aujourd'hui la base théorique des circuits numériques.

La transformation décisive intervient avec Gottlob Frege. Dans *Begriffsschrift* (1879) [**Fre79**] (idéographie), il introduit un langage formel capable d'exprimer des raisonnements mathématiques complexes et fonde la logique des prédicats. Il introduit les concepts de quantificateurs ("pour tout", "il existe").

Au début du XX^{e} siècle, Bertrand Russell, Giuseppe Peano, Alfred North Whitehead tentent de montrer que les mathématiques peuvent être fondées entièrement sur la logique (en particulier un petit nombre d'axiomes) dans leur ouvrage *Principia Mathematica* [**RW10**].

1.3. Formalisation des mathématiques, controverses et limites. — David Hilbert propose au début du XX^{e} siècle un programme visant à formaliser toutes les mathématiques dans des systèmes axiomatiques rigoureux [**HA28**]. L'objectif est de démontrer leur cohérence à l'aide de méthodes finitaires. Il lance le programme "formaliste", visant à prouver la cohérence et la complétude des mathématiques.

En 1907 Luitzen Egbertus Jan Brouwer commence à formuler ses idées intuitionnistes dès sa thèse en 1907, intitulée "Over de grondslagen der wiskunde" ("Sur les fondements des mathématiques").

Il s'en suit une controverse Brouwer–Hilbert dans les années 1920, qui marque le choc entre intuitionnisme et formalisme (au sens de Hilbert). Hilbert défend une approche formaliste : les mathématiques

sont un système de symboles régis par des règles logiques précises, indépendantes de toute intuition. Dans ce cadre, la logique classique domine, notamment le principe du tiers exclu (toute proposition est vraie ou fausse). Brouwer, au contraire, fonde l'intuitionnisme, où les objets mathématiques n'existent que s'ils peuvent être construits explicitement. Il rejette donc le tiers exclu en général, car affirmer "P ou non-P" sans preuve constructive n'a pas de sens.

Cette divergence conduit à deux logiques différentes : la logique classique (Hilbert) et la logique intuitionniste (Brouwer). Des mathématiciens comme Arend Heyting formaliseront la logique intuitionniste, tandis que Kurt Gödel explorera les limites du programme de Hilbert. Gödel montrera notamment en 1931, avec ses théorèmes d'incomplétude, que tout système formel suffisamment riche contient des vérités indémontrables **[Göd31]** dans ce même système.

Arend Heyting, qui a été étudiant de Brouwer à l'Université d'Amsterdam, est le principal formaliseur de la pensée de ce dernier : il donne une version rigoureuse de la logique intuitionniste (calcul des prédicats intuitionniste, sémantique, règles de preuve). Autrement dit, il transforme une philosophie mathématique en un système logique manipulable, parallèle à la logique classique de Hilbert.

Par ailleurs et indépendamment, Andreï Kolmogorov propose une lecture sémantique : il voit les énoncés logiques comme des problèmes à résoudre. Dans cette vision (souvent appelée interprétation de Brouwer–Heyting–Kolmogorov, ou BHK), une implication est une méthode transformant une preuve en une autre, une disjonction correspond au choix explicite d'un cas, une existence signifie qu'on peut construire un objet.

Ainsi, Heyting joue un rôle syntaxique et formel (il construit la logique), tandis que Kolmogorov apporte une interprétation sémantique computationnelle avant l'heure. C'est cette lignée qui influencera plus tard la correspondance preuves-programmes en informatique théorique.

Plus tard, Alonzo Church et Alan Turing relieront ces questions à la calculabilité, rapprochant intuitionnisme et informatique (voir 1.4).

Parallèlement, Gerhard Gentzen développe de nouveaux systèmes de preuve comme la déduction naturelle et le calcul des séquents, qui

structurent les démonstrations de manière plus proche du raisonnement mathématique [**Gen35**].

1.4. Logique et naissance de l'informatique. — C'est cette logique mathématique, devenue un outil abstrait et puissant, qui va fournir les fondements théoriques à l'informatique, en particulier lorsque les logiciens chercheront à formaliser la notion d'algorithme.

Au XIX^{e} siècle apparaissent également les premiers projets de machines capables d'automatiser des calculs complexes. Les travaux de Charles Babbage et d'Ada Lovelace constituent une étape intermédiaire entre la mécanisation du calcul et la formalisation logique du raisonnement. La machine analytique de Babbage introduit l'idée d'une machine programmable universelle, tandis que les notes d'Ada Lovelace montrent comment des suites d'instructions (algorithme) peuvent être exécutées automatiquement. Lovelace souligne également que de telles machines pourraient manipuler des symboles et non seulement des nombres, anticipant ainsi l'idée que les machines pourraient traiter des structures abstraites comme celles étudiées en logique. Ces idées préfigurent la conception moderne des programmes et préparent le terrain pour la formalisation des algorithmes par Church et Turing au XX^{e} siècle.

En 1936, Alan Turing introduit sa célèbre machine, un modèle abstrait décrivant les procédures de calcul mécaniques [**Tur36**]. Au même moment, Alonzo Church développe le λ-calcul [**Chu41**], un système formel décrivant les fonctions et les transformations symboliques.

Ces travaux fondent la théorie de la calculabilité et établissent les bases théoriques de l'informatique.

Le développement des premiers ordinateurs dans les années 1940-1950 est directement redevable à l'algèbre de Boole. Celle-ci permet de concevoir des circuits électroniques (les portes logiques) qui réalisent physiquement les opérations logiques (ET, OU, NON) à la base de tout calcul informatique.

En 1958, une vingtaine d'années après le λ-calcul de Church, le premier langage de programmation fonctionnelle [1] est né : Lisp. Il s'agit

1. Rappelons que Fortran (1957) est généralement considéré comme étant le premier langage de haut niveau.

de la première tentative réussie de transformer une théorie mathématique du calcul (λ-calcul) en langage de programmation concret. Suivront par la suite d'autres langages fonctionnels tels que Scheme (1975), Haskel (1990), Caml-Light et OCaml (1990, hérités du ML des années 1970).

1.5. Preuves formelles et assistants de preuve. — En section 1.3 nous avons cité les mots *formalisation des mathématiques*, utilisés au début du XX^{e} siècle, avec les travaux de Frege, Hilbert et Gentzen. Cependant, l'informatique permet d'aller plus loin en développant des systèmes capables de vérifier automatiquement les démonstrations, ce sont les outils de preuves formelles. Depuis les années 1970, le sens du terme formalisation des mathématiques a donc évolué dans cette direction.

Une preuve formelle est une démonstration exprimée entièrement dans un système logique précis, où chaque étape suit explicitement une règle d'inférence. Les outils de preuves formelles sont alors basés sur des familles de logiques, qui peuvent être très différentes, au même titre qu'il existe plusieurs paradigmes pour les langages de programmation. Nous donnerons un panel de ces logiques en section 2.

Les outils de preuves formelles, que ce soit des prouveurs automatiques ou des prouveurs interactifs, n'ont pas uniquement vocation à être utilisés pour formaliser des mathématiques. Par exemple ils servent dans des domaines où les bugs peuvent avoir de graves conséquences, comme l'aéronautique, le ferroviaire, le médical ou les logiciels embarqués en prouvant la correction des programmes, des protocoles ou de compilateur [**BL09**] ou micro-noyaux [**KEH^{+}09**] afin de garantir leur fiabilité avant leur déploiement. Pour cette raison, nous précisions que nous ne nous intéresserons ici qu'aux bibliothèques mathématiques et quelques grands résultats qui ont été ou sont formalisés avec ces prouveurs (voir également la Figure 2).

Le système Automath. — a permis la formalisation d'une partie des éléments d'Euclide, constituant l'un des premiers grand projet de mathématiques formelles [**dB70**].

En Coq/Rocq. — , nous pouvons citer la formalisation et la preuve du théorème des quatre couleurs [**Gon08, GW07**] ou le théorème de Feit–Thompson [**GAA^{+}13**]

Le système PVS. — a été utilisé pour la vérification de systèmes critiques, notamment en aéronautique [**MCDB03**].

La Mizar Mathematical Library. — constitue une des plus vaste bibliothèque de mathématiques formalisées depuis la fin des années 1980 [**Com**].

Agda. — est utilisé pour la formalisation de la théorie des types homotopiques (HoTT) depuis les années 2010 [**Pro13**].

C'est en combinant HOL-Light et Isabelle. — que le projet Flyspeck (2014) a formalisé complètement la conjecture de Kepler [**HAB+15**].

En Isabelle/HOL. — nous pouvons citer la formalisation des théorèmes de Gödel [**Pau21**].

En Lean. — est développée depuis 2020 la bibliothèque Mathlib, une des plus importante à ce jour [**Com17**].

Synthèse. — Comme le résume la Figure 1, l'histoire de la logique montre une évolution importante : discipline philosophique dans l'Antiquité, elle devient une branche fondamentale des mathématiques au XIX^{e} siècle, puis un pilier théorique de l'informatique au XX^{e} siècle.

La Figure 2 présente les systèmes de preuve formelle et les assistants de preuve qui représentent aujourd'hui l'aboutissement de cette évolution : au XXI^{e} siècle le raisonnement mathématique lui-même peut être représenté, analysé et vérifié par des machines, voire autoformalisé et autoprouvé par l'Intelligence Artificielle [**AIT**].

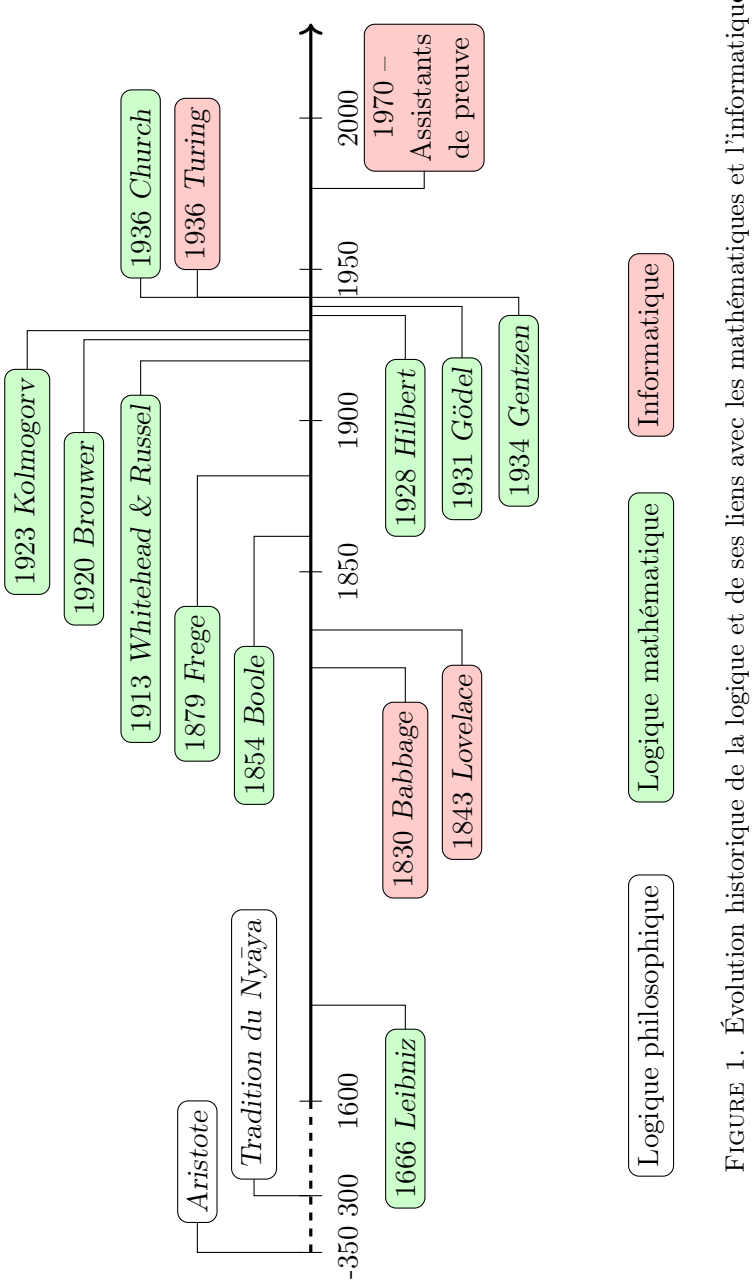

FIGURE 1. Évolution historique de la logique et de ses liens avec les mathématiques et l'informatique

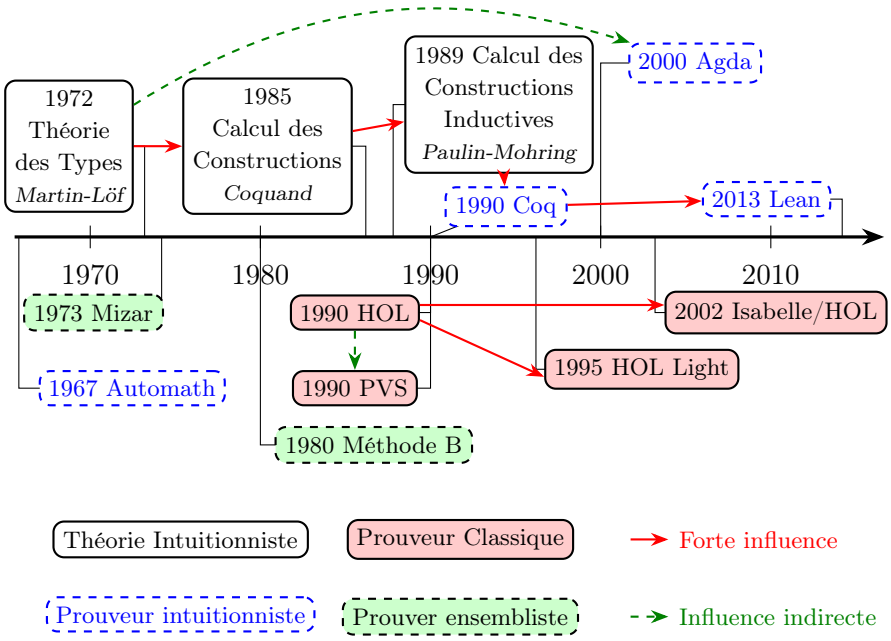

FIGURE 2. Naissance et évolution des outils de preuve formelle interactifs et leur lien avec les théories logiques

2. Les familles de prouveurs intéractifs

Nous aborderons uniquement les familles de prouveurs interactifs. Nous ne parlerons pas, par exemple, des prouveurs automatiques tels que les solveurs SMT (Satisfiability Modulo Theories).

Les prouveurs interactifs (Interactive Theorem Provers, ITP) sont des systèmes informatiques permettant de formaliser et prouver des théorèmes ou propriétés mathématiques avec l'aide d'un utilisateur. Contrairement aux prouveurs automatiques, qui cherchent seuls une preuve, les prouveurs interactifs reposent sur une collaboration entre l'utilisateur et le système : l'utilisateur guide la structure de la preuve, tandis que le système vérifie chaque étape selon une logique formelle.

Ces prouveurs sont basés sur une logique, et ces sont ces logiques qui participent à les différentier, accompagnés d'autres critères tels que leur langage de preuves, leurs bibliothèques, etc.

2.1. Prouveurs basés sur la logique d'ordre supérieur. — Une première famille importante repose sur la logique d'ordre supérieur (Higher-Order Logic, HOL), que nous décrirons en section3. Les systèmes comme HOL-Light et HOL4 utilisent une logique simple fondée sur le λ-calcul typé. Les preuves sont construites à l'aide de tactiques programmées, généralement dans un langage fonctionnel (OCaml pour HOL-Light ou Standard ML pour HOL4). Les tactiques sont des "ordres" donnés par l'utilisateur à l'outil qui indique quelle règle logique appliquer.

Ces prouveurs possèdent des bibliothèques mathématiques couvrant notamment l'arithmétique des entiers, des rationnels et des réels, ainsi que l'analyse et l'algèbre.

Un système proche mais plus général est Isabelle. Isabelle est un cadre générique de prouveurs capable d'implémenter plusieurs logiques, la plus utilisée étant Isabelle/HOL. Il propose un langage structuré appelé Isar (en référence à Mizar pour son langage proche des mathématiques naturelles), qui permet d'écrire des preuves lisibles et proches du style mathématique naturel. Isabelle dispose également d'une vaste bibliothèque, l'*Archive of Formal Proofs*, et d'outils d'automatisation comme *Sledgehammer*, qui permettent de combiner preuve interactive et preuve automatique.

2.2. Prouveurs basés sur la théorie des types dépendants. — Une autre famille majeure de prouveurs repose sur la théorie des types dépendants, inspirée du Calcul des Constructions et du Calcul des Conctructions Inductives. Ce sont également des langages d'ordre supérieur. Les systèmes Coq — récemment renommé Rocq — et Lean appartiennent à cette catégorie. Nous exposerons quelques aspects techniques en section 3.

Dans ces systèmes, les propositions sont représentées comme des types et les preuves comme des programmes, conformément à la correspondance de Curry–De Bruinj-Howard [**How80**]. Cette approche permet d'exprimer des propriétés très riches et de formaliser des structures mathématiques complexes. L'arithmétique est généralement définie de manière constructive (par exemple les entiers de Peano), et de nombreuses bibliothèques couvrent des domaines avancés des mathématiques.

Comme énoncé en section 1.5, Lean se distingue par sa bibliothèque mathlib, qui contient une formalisation étendue de nombreux domaines mathématiques modernes et constitue l'un des plus grands corpus de mathématiques formalisées, avec la bibliothèque de Mizar.

Des systèmes proches incluent également Agda, qui combine langage de programmation et assistant de preuve, ou encore Twelf, utilisé comme cadre logique pour représenter différents systèmes formels.

2.3. Systèmes hybrides et orientés vérification. — Certains systèmes adoptent une approche hybride. Nous pouvons citer PVS, qui repose sur une logique d'ordre supérieur typée enrichie par des sous-types et des prédicats, permettant d'exprimer des contraintes précises dans les types eux-mêmes. PVS intègre fortement des décideurs automatiques et des SMT solvers, ce qui facilite la preuve de propriétés impliquant des contraintes arithmétiques ou logiques complexes. Il est largement utilisé pour la vérification de systèmes critiques dans l'industrie.

Un autre système est ACL2, qui repose sur une logique du premier ordre adaptée à la vérification de programmes écrits en Lisp. ACL2 combine preuve interactive et automatisation, et possède une forte orientation vers la vérification matérielle et logicielle.

2.4. Prouveurs basés sur la théorie des ensembles. — Certains prouveurs sont fondés directement sur la théorie des ensembles, qui constitue la base classique des mathématiques modernes. Par exemple Mizar, introduit par Andrzej Trybulec, repose sur une logique classique fortement inspirée de la théorie des ensembles et utilise un langage de preuve déclaratif proche de l'écriture mathématique traditionnelle. Sa bibliothèque, la Mizar Mathematical Library, contient plusieurs milliers de résultats couvrant l'arithmétique, l'algèbre, l'analyse et la théorie des ensembles.

Un autre exemple est Metamath, qui permet de formaliser des mathématiques directement dans un cadre axiomatique basé sur la théorie des ensembles ZFC (Zermelo-Fraenkel avec Axiome du Choix). Metamath se distingue par un noyau extrêmement minimaliste et par une base de connaissances mathématiques très large.

La méthode B, introduite par Jean-Raymond Abrial, constitue une approche importante dans le domaine de la vérification formelle industrielle. Elle repose sur la théorie des ensembles, la logique du premier ordre et l'arithmétique entière. Les systèmes sont développés par raffinement successif, chaque étape étant accompagnée d'obligations de preuve vérifiées par des outils comme Atelier B ou Rodin Platform. Cette approche a été utilisée dans plusieurs systèmes critiques, notamment dans les systèmes ferroviaires (ligne 14 du métro parisien issue du projet meteor, par exemple).

2.5. Synthèse. — Les prouveurs interactifs peuvent ainsi être regroupés selon leur fondement logique principal :

- Logique d'ordre supérieur : HOL-Light, HOL4, Isabelle/HOL
- Théorie des types dépendants, CCI : Coq/Rocq, Lean, Agda
- Logiques hybrides avec automatisation : PVS, ACL2
- Approche basée sur la théorie des ensembles : Mizar, Metamath, Isabelle/ZF, Méthode B

Notons que dans cette classification faite selon un fondement logique principal, lorsqu'un prouveur est classé dans une catégorie, cela ne signifie pas qu'il ne possède aucune des caractéristiques d'autres catégories. Par exemple, Rocq possède des tactiques d'automatisation et sa logique est également d'ordre supérieur.

3. Focus sur Rocq

"The Rocq Prover follows from over 40 years of research in Dependent Type Theory and Interactive Theorem Proving. It is based on the Predicative, Polymorphic and Cumulative Calculus of Inductive Constructions (PCUIC)." (2)

Il ne s'agit pas ici de faire un cours sur le λ-calcul, ni sur la théorie des types de Martin-Löf, si sur le Calcul des Constructions Inductives (CCI), ni sur PCUIC. Nous survolerons les quelques notions

2. https://rocq-prover.org/why
Rocq est le fruit de plus de 40 ans de recherche dans le domaine de la théorie des types dépendants et de la démonstration interactive de théorèmes. Il repose sur le calcul prédicatif, polymorphe et cumulatif des constructions inductives (PCUIC).

nécessaires pour comprendre la logique sous-jacente de Rocq et mieux appréhender son utilisation.

3.1. Le Calcul des Constructions Inductives. — Ce calcul **[CP90, Wer94, PM96]** étend le Calcul des Constructions (CoC) **[Coq85, CH88]** en y ajoutant des types inductifs.

De ce fait, le CCI unifie dans un seul système le λ-calcul typé, la déduction naturelle, la logique intuitionniste d'ordre supérieur, les types dépendants, types inductifs, l'isomorphisme de Curry-De Bruijn-Howard et une hiérarchie de types (non présentée dans ce support). Nous allons présenter succinctement chacune de ces notions en les illustrant avec des exemples.

3.1.1. *Le λ-calcul typé.* — "Le λ-calcul, on aurait pu l'appeler flèche-calcul" disait parfois Gilles Dowek lors de ses cours. Ce formalisme vise à formaliser les mathématiques sur la notion de fonction (mathématiques), plutôt que sur celle d'ensemble. Cette formalisation, datant des années 30, fut la source, 20 ans plus tard, du premier langage fonctionnel de programmation, Lisp en 1958. On parle de λ-calcul typé lorsque l'on ajoute des informations de type. Ces informations de type permettent d'obtenir la propriété de normalisation forte (toutes les séquences de réductions se réduisent à une même forme normale, voir ci-après pour la notion de réduction), nécessaire à la cohérence logique.

Le λ-calcul est composé de variables, d'applications et d'abstractions. Nous parlons alors de λ-terme. Un parallèle naturel pour comprendre ce formalisme consiste à le comparer au formalisme mathématique :

	langage naturel	maths	λ-terme
abstraction	$\lambda\ x\cdot\ v$ représente la fonction qui à x associe v, où v contient en général des occurrences de x	$x \to\ v(x)$	$\lambda\ x\cdot\ v$
application	si u est une fonction et si v est son argument, alors $(u\ v)$ est le résultat de l'application à v de la fonction u	$u(v)$	$(u\ v)$

Voici deux exemples concrets :

Soit la fonction f de $\mathbb{N}$ dans $\mathbb{N}$ qui à x et y associe $2x + y$:

$$\begin{aligned} f : \quad & \mathbb{N}^2 \to \mathbb{N} \\ & (x, y) \mapsto 2x + y \end{aligned}$$

Le λ-terme typé correspondant est :

$$\lambda x : \mathbb{N}.\lambda y : \mathbb{N}.2x + y$$

Ce formalisme est plus expressif et opérationnel que le formalisme mathématique usuel, car il décrit non seulement ce qu'est une fonction, mais aussi comment elle se calcule et comment elle se manipule. Il unifie fonction, calcul et syntaxe dans un même système. Il permet de représenter des fonctions partielles, infinies, ou non calculables comme objets syntaxiques, qui font la force des langages fonctionnels.

Reprenons l'exemple ci-dessus dans le langage de programmation OCaml, en mode interactif (`#` représente le prompt) :

```
1  # let f x y=2*x+y;;
2  val f : int -> int -> int = <fun>
3  # let g= f 3;;
4  val g : int -> int = <fun>
5  # g 2;;
6  - : int = 8
```

En ligne `1` nous déclarons/définissons la fonction f(x,y)=2x+y. La ligne `2` est la réponse d'OCaml qui indique que `f` est une fonction de type `int->int->int` i.e. une fonction qui prend 2 entiers et qui renvoie un entier. Nous remarquons que ce type correspond exactement à l'expression donnée en λ-calcul. Nous aurions également pu écrire

```
# let f (x,y)=2*x+y;;
val f : int * int -> int = <fun>
```

Nous voyons que les types `int * int -> int` et `int -> int -> int` sont isomorphes. Cette transformation se nomme *curryfication*.

En ligne `3` nous définissons la fonction partielle `g` qui retourne `f(3)` qui est l'application partielle de `f` appliquée à un seul argument `x=3`. Le type de `g` est donc bien `int->int`. En ligne `5`, on donne le paramètre `2` à `g`, ce qui équivaut à donner le second paramètre `y` à `f`. Le résultat est donc $f(3, 2) = 8$.

Pour effectuer ce genre de "calculs", le λ-calcul fournit ce que l'on appelle des règles de réduction. Les deux plus connues, simples et essentielles sont les règles de α-conversion et de β-conversion.

L'α-conversion permet d'affirmer que le nommage des variables liées (vl) est indifférent : $\lambda x.\lambda y.2x + y$ est la même fonction, i.e. le même λ-terme, que $\lambda z.\lambda t.2z + t$.

La β-réduction est ainsi définie : l'expression $(\lambda x.s)\ t$ se réduit en $s[t/x]$ (également noté $[x := t]$ ou $[x \leftarrow t]$).

Dit autrement, il s'agit du résultat de l'application d'un λ-terme à une valeur. Ainsi, certaines étapes de raisonnement se réduisent à des calculs effectifs. Cette interaction entre calcul et preuve permet de simplifier de nombreux arguments mathématiques. Par exemple, si nous reprenons l'exemple précédent, $(\lambda x.\lambda y.2x + y)\ 3\ 2$ se réduit en $2x + y[x := 3, y := 2]$ et l'on note $(\lambda x.\lambda y.2x + y)\ 3\ 2 \rightsquigarrow_\beta 8$.

Le λ-calcul permet également très simplement de parler de fonction ayant d'autres fonctions comme paramètres : $\lambda f.\lambda x.f(fx)$

Qui se traduit ainsi en OCaml :

```
1 # let appliquer_deux_fois f x = f (f x);;
2 val appliquer_deux_fois : ('a -> 'a) -> 'a -> 'a = <fun>
3 # let resultat = appliquer_deux_fois
                  (fun x -> x * 2) 3;;
4 val resultat : int = 12
5 # let resultat = appliquer_deux_fois
                   (fun x -> x *. 2.) 3.;;
6 val resultat : float = 12.
```

En ligne 1 nous déclarons/définissons la fonction `appliquer_deux_fois` qui prend deux paramètres : `f` et `x`. Nous remarquons que cette définition de fonction n'indique pas explicitement de type. L'algorithme d'inférence de type permet de déduire les types. La fonction `f` prend l'argument `x` de type que OCaml note `'a`, elle retourne une valeur de même type puisque c'est aussi l'argument de `f` une seconde fois. La ligne 2 indique donc le type `('a -> 'a) -> 'a -> 'a`. On parle alors de type polymorphe. La ligne 3 calcule le résultat en appliquant à la fonction $x \rightarrow x * 2$ de type `int -> int` et à 3 de type `int`. Le résultat est bien `12`. A la ligne 5, on applique la même fonction mais de type `float -> float` (notation `2.`) etc.

En résumé, tous ces exemples illustrent le λ-calcul comme un formalisme où une fonction est conçue comme une règle de calcul, c'est-à-dire un programme, pouvant être manipulée comme une donnée.

3.1.2. *La déduction naturelle.* — La déduction naturelle est une méthode formelle utilisée en logique pour démontrer des propositions à partir de règles d'inférence. Son objectif principal est de montrer qu'une conclusion découle nécessairement d'un ensemble de prémisses.

Le principe fondamental est le suivant : à chaque connecteur logique correspond une paire de règles. Les règles d'*introduction* expliquent comment construire une preuve d'une formule comportant ce connecteur, tandis que les règles d'*élimination* décrivent comment exploiter une telle formule.

La Figure 3 énonce les règles de la déduction naturelle. Elles se lisent de bas en haut. Par exemple si nous prenons la règle $\wedge i$, la règle nous indique que pour montrer $A \wedge B$ dans un environnement Γ (i.e. contenant des hypothèses), alors il faut à la fois montrer A dans ce même environnement Γ et B dans ce même environnement. Montrer A dans un environnement Γ se note $\Gamma \vdash A$ et se lit "Γ thèse A".

En utilisant ces règles, nous pouvons maintenant montrer des propriétés. Les tactiques de Rocq sont construites sur de telles règles. Montrons par exemple que $\vdash \neg\neg A \Rightarrow A$. Nous rappelons que cette propriété est une propriété classique, la preuve utilise donc le tiers exclus :

$$\dfrac{\dfrac{\dfrac{}{\neg\neg A \vdash A \vee \neg A}\,EM \quad \dfrac{}{\neg\neg A, A \vdash A}\,Ax \quad \dfrac{\dfrac{\dfrac{}{\neg\neg A, \neg A \vdash \neg\neg A}\,Ax \quad \dfrac{}{\neg\neg A, \neg A \vdash \neg A}\,Ax}{\neg\neg A, \neg A \vdash \bot}\,\neg e}{\neg\neg A, \neg A \vdash A}\,\bot e}{\neg\neg A \vdash A}\,\vee e}{\vdash \neg\neg A \Rightarrow A}\,\Rightarrow i$$

Les tactiques de base de Rocq sont proches des règles de la déduction naturelle. Par exemple, la tactique `intro` correspond à la règle $\Rightarrow i$, la tactique `assumption` correspond à la règle Ax etc. La plupart des tactiques appliquent plusieurs règles.

En section 3.2, nous reverrons la même preuve faite en Rocq.

Axiome	Classique	Coupure
$\frac{A \in \Gamma}{\Gamma \vdash A}\, Ax$	$\frac{}{\Gamma \vdash A \vee \neg A}\, EM$	$\frac{\Gamma, A \vdash B \quad \Gamma \vdash A}{\Gamma \vdash B}\, Cut$

	Introduction	Elimination
$\bot$		$\frac{\Gamma \vdash \bot}{\Gamma \vdash C}\, \bot\, e$
$\neg$	$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\, \neg i$	$\frac{\Gamma \vdash \neg A \quad \Gamma \vdash A}{\Gamma \vdash C}\, \neg e$
$\wedge$	$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\, \wedge i$	$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\, \wedge eg \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\, \wedge ed$
$\vee$	$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\, \vee ig \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\, \vee id$	$\frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\, \vee e$
$\Rightarrow$	$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\, \Rightarrow i$	$\frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\, \Rightarrow e$
$\forall$	$\frac{\Gamma \vdash P \quad x \notin vl(\Gamma)}{\Gamma \vdash \forall x, P}\, \forall i$	$\frac{\Gamma \vdash \forall x, P}{\Gamma \vdash P[x \leftarrow t]}\, \forall e$
$\exists$	$\frac{\Gamma \vdash P[x \leftarrow t]}{\Gamma \vdash \exists x, P}\, \exists i$	$\frac{\Gamma \vdash \exists x, P \quad \Gamma, P \vdash C \quad x \notin vl(\Gamma, C)}{\Gamma \vdash C}\, \exists e$

	Hypothèses
$\bot$	$\frac{}{\Gamma, \bot \vdash C}\, \bot\, h$
$\neg$	$\frac{\Gamma, \neg A \vdash A}{\Gamma, \neg A \vdash C}\, \neg h$
$\wedge$	$\frac{\Gamma, A, B \vdash C}{\Gamma, A \wedge B \vdash C}\, \wedge h$
$\vee$	$\frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \vee B \vdash C}\, \vee h$
$\Rightarrow$	$\frac{\Gamma, B \vdash C \quad \Gamma, A \Rightarrow B \vdash A}{\Gamma, A \Rightarrow B \vdash C}\, \Rightarrow h$
$\forall$	$\frac{\Gamma, (\forall x, P), P[x \leftarrow t] \vdash C}{\Gamma, (\forall x, P) \vdash C}\, \forall h$
$\exists$	$\frac{\Gamma, P \vdash C \quad x \notin vl(\Gamma, C)}{\Gamma, (\exists x, P) \vdash C}\, \exists h$

FIGURE 3. Les règles de la déduction naturelle

3.1.3. *La logique d'ordre supérieur.* — La logique d'ordre supérieur (HOL, pour Higher-Order Logic) étend la logique du premier ordre en autorisant la quantification non seulement sur des éléments d'un domaine, mais aussi sur des fonctions, des relations, et plus généralement sur des objets d'ordre supérieur. Autrement dit, on peut manipuler des prédicats comme des objets mathématiques à part entière.

En logique du premier ordre, une formule comme $\forall x \exists y, P(x, y)$ exprime une propriété sur des éléments x et y d'un domaine, où P est une relation ou propriété fixée. On peut écrire par exemple : $\exists P, \forall x, \exists y, P(x, y)$ où cette fois P est une variable : on quantifie sur les relations ou propositions ou fonctions elles-mêmes.

Un exemple classique est l'union et l'inclusion d'ensembles : on peut écrire et montrer trivialement que $\forall E, \forall F, E \subseteq E \cup F$. Nous reviendrons sur cet exemple en section 3.2.

Un autre point important est que les fonctions sont des objets primitifs. On peut écrire : $\forall f : \mathbb{N} \to \mathbb{N}, ; \exists n; f(n) \geq n$

Ici, f est une variable fonctionnelle, ce qui n'est pas possible en logique du premier ordre.

Cela rapproche fortement la logique d'ordre supérieur du langage usuel des mathématiques, où l'on manipule constamment des applications, des opérateurs, des foncteurs, etc.

En résumé, la logique d'ordre supérieur permet de formaliser directement le langage des mathématiques, en traitant les prédicats et les fonctions comme des objets à part entière.

3.1.4. *La logique intuitionniste.* — La logique *intuitionniste* réfute un axiome majeur de la logique *classique*. Il s'agit de l'axiome du tiers exclu :

$$\forall P, P \vee \neg P$$

Par exemple, la proposition "tout nombre réel est soit rationnel, soit irrationnel", trivialement vraie en logique classique, n'est pas prouvable en logique intuitionniste.

L'axiome du tiers exclu établit que la proposition "ne pas être faux" est équivalente à la proposition "être vrai" ($\forall P, \neg\neg P \leftrightarrow P$), ce qui n'est pas le cas en logique intuitionniste [3]

Une autre différence réside dans le fait que la logique intuitionniste n'accepte une preuve d'existence d'un objet mathématique, satisfaisant une propriété, que si elle fournit une méthode effective (un programme par exemple) de construction de l'objet en question.

3.1.5. *Les types inductifs.* — ils permettent de définir récursivement des types de données infinies ; c'est le plus petit ensemble fermé par un certain nombre de constructeurs. Par exemple, en Rocq, nous pouvons définir le types des entiers naturels de la manière suivante :

```
Inductive nat : Set :=
  | O : nat
```

3. Notons que dans cette équivalence, c'est l'implication $\to$ qui est fausse, $\leftarrow$ étant valide en logique classique et intuitionniste.

```
| S : nat → nat.
```

Le constructeur `O` représente zéro et `S` est le constructeur successeur. Les entiers naturels sont alors exactement les termes obtenus par applications finies de ces constructeurs. On appelle les entiers ainsi définis les *entiers de Peano* [4], en référence à l'arithmétique de Peano **[Pea89]**. Par exemple, le nombre 5 s'écrira (`S (S (S (S (S O))))`).

À toute définition inductive est associé un schéma d'induction. Pour les entiers, on retrouve le principe de récurrence usuel :

```
nat_ind
     : ∀ P : nat → Prop,
       P 0 →
       (∀  n : nat, P n → P (S n)) →
       ∀ n : nat, P n
```

Celui-ci permet de démontrer qu'une propriété $P(n)$ est vraie pour tout entier naturel n en montrant :

— que $P(0)$ est vraie ;

— que si $P(n)$ alors $P(S(n))$ pour tout n.

Il est alors possible de définir des fonctions inductives telle que l'addition par exemple :

```
Fixpoint add n m :=
  match n with
  | O ⇒ m
  | S p ⇒ S (add p m)
  end
```

Les règles de réduction [5] permettent alors de "calculer" (réduire) un résultat : si nous notons 3=(`S (S (S O))`), 2=(`S (S O)`) et ainsi de suite, 3 + 2 se réduira en 5 par (`add 3 2`) ⇝ (`S (add 2 2)`) ⇝ (`S (S (add 1 2))`) ⇝ (`S (S (S (add 0 2))))`) ⇝ (`S (S (S 2))`) c'est-à-dire (`S (S (S (S (S O)))))`) (=5).

Cette interaction étroite entre calcul et preuve rend possible des preuves très concises, car certaines étapes sont prises en charge très

4. Pour expliquer la sémantique, on nomme parfois les entiers de Peano les *entiers bâtons*, en remplaçant le symbole `S` par | et `O` par rien. 5 s'écrira alors | | | | |.

5. voir https://rocq-prover.org/doc/V9.1.1/refman/language/core/conversion.html pour l'ensemble des règles de réduction disponibles en Rocq.

simplement par la réduction. Notons néanmoins que ces "calculs" ne sont pas particulièrement efficaces (rapides). Par exemple, il existe d'autres manières de définir certains nombres. La notion de calcul n'est pas l'objectif de cet exemple centré sur les définitions inductives.

Grâce au mécanisme inductif, il est également possible de définir des prédicats inductifs. Par exemple, la propriété "être pair" peut être définie par les règles suivantes :

— 0 est pair ;
— si n est pair alors $S(S(n))$ est pair.

Dans ce cas, une preuve que n est pair correspond à un terme construit à partir de ces règles.

3.1.6. *Les types dépendants.* — L'un des constructeurs fondamentaux est le produit dépendant :

$$\Pi x : A.\, B(x),$$

qui généralise à la fois les fonctions et le quantificateur universel. Un terme de ce type est une fonction qui, à tout élément x de type A, associe un terme de type $B(x)$. Dans l'interprétation logique, cela correspond à une preuve de la proposition `∀ x:A, B(x)`.

Notons que lorsque B ne dépend pas de x, on retrouve l'implication $A \to B$.

Un exemple de type dépendant est celui des listes de taille n, ou le type de ces listes dépend d'un autre argument de type entier. Voyons cela en Rocq :

```
Inductive list_n {A: Type} : nat → Type :=
  | niln: list_n 0
  | consn (n: nat) (_: A) (t: list_n n): list_n (S n).
```

`{A:Type}` indique que cet argument est implicite, i.e. qui peut être déduit (inféré) à partir du contexte. `niln` est une liste à 0 élément et (`consn n _ t`) est une liste à n + 1 éléments de type `A` si t est une liste à n éléments. Ainsi, (`list_n A n`) définit les listes à n éléments de type `A`.

Nous pouvons maintenant définir, le type liste d'entiers de taille 0. Notons que nous devons lui préciser ici explicitement (symbole @) le type des éléments de la liste car le contexte ne permet pas de l'inférer.

```
Definition list_0:= @niln nat.
```

```
Print list_0.
(*
list_0 = niln
     : list_n 0
*)
```

Nous pouvons voir en commentaire, `(*...*)`, la définition, accompagnée de son type, fournie par Rocq.

La liste de taille 1 contenant l'entier 2 ainsi que celle de taille 2 contenant les entiers 3 et 2 seront ainsi définies :

```
Definition list_1_with_2:= (consn 0 2 list_0).
Print list_1_with_2.
(*
list_1_with_2 = consn 0 2 list_0
     : list_n 1
*)

Definition list_2_with_2_3:= (consn 1 3 list_1_with_2).
Print list_2_with_2_3.
```

Notons que nous pouvons très bien définir directement une liste. Par exemple, la liste de taille 5 contenant les entiers [15 ;14 ;13 ;12 ;11] ou celle contenant les booléens [true ;false ;false ;true ;true], toutes deux de taille 5 et de types respectifs (`list_n nat` 5) et (`list_n bool` 5). Les commandes `Check` nous donnent leur type :

```
Check (consn 4 15 (consn 3 14 (consn 2 13 (consn 1 12 (consn 0 11 niln))))).
(*
consn 4 15
  (consn 3 14 (consn 2 13 (consn 1 12 (consn 0 11 niln))))
     : list_n 5
*)

Check (consn 4 true
  (consn 3 false (consn 2 false (consn 1 true (consn 0 true niln))))).
(*
consn 4 true
  (consn 3 false
     (consn 2 false (consn 1 true (consn 0 true niln))))
```

```
    : list_n 5
*)
```

Ou, en montrant explicitement tous les types :

```
(*
@consn bool 4 true
  (@consn bool 3 false
    (@consn bool 2 false
      (@consn bool 1 true
        (@consn bool 0 true (@niln bool)))))
    : @list_n bool 5
*)
```

De la même manière, il existe la somme dépendante, notée

$$\Sigma\, x : A.\, B(x)$$

dont l'interprétation logique correspond à une preuve de la proposition `∃ x:A, B(x)`.

3.1.7. *L'isomorphisme de Curry-de Bruijn-Howard.* — Il s'agit de la correspondance entre preuves et programmes, établissant une analogie entre logique et programmation. Les propositions sont interprétées comme des types, les preuves sont des termes de ces types. De ce fait, démontrer une proposition P revient donc à construire un terme bien typé p de type P. La vérification d'une preuve revient alors à vérifier le type.

Grâce à cet isomorphisme, il est alors possible d'obtenir une programme à partir d'un λ-terme de preuve. En Rocq, on appelle ce principe l'extraction. Ce mécanisme permet donc de transformer des preuves et des programmes écrits dans le cadre du calcul des constructions en code exécutable dans des langages fonctionnels comme OCaml, Haskell ou Scheme. Lors de l'extraction, les parties purement logiques (c'est-à-dire les preuves sans contenu calculatoire) sont effacées, tandis que les parties computationnelles (construites au sens de la logique intuitionniste [6]), sont conservées et traduites

6. Nous ne parlons pas de la réalisabilité qui fournit une interprétation sémantique des preuves permettant d'en extraire un contenu calculatoire, y compris dans des cadres non strictement intuitionnistes, en associant à chaque énoncé un ensemble de programmes qui en réalisent effectivement la preuve.

dans le langage cible. Cela permet d'obtenir du code certifié correct par construction à partir du λ-terme de preuve.

3.1.8. *Synthèse.* — En résumé, le Calcul des Constructions Inductives est une théorie des types riche qui unifie logique et programmation. Grâce à la correspondance de Curry–De Bruijn-Howard et aux types inductifs, il permet de représenter simultanément des objets mathématiques, des propositions et leurs preuves dans un même système formel, constituant ainsi la base théorique de plusieurs assistants de preuve.

3.2. Rocq en action. — Dans cette section d'initiation à Rocq, nous allons voir comment les recherches sur ces théories logiques sont maintenant utilisées, à travers ces assistants d'aide à la preuve, pour l'enseignement des mathématiques. Nous reprendrons des exemples issus de différents cours destinés à des élèves de lycée et de cycle L en Double-Licence maths-info [**KLRM$^+$22, BCH$^+$24, KMR24**]. L'objectif est de montrer comment un prouveur tel que Rocq peut avantageusement permettre d'aider les étudiants à formaliser leurs démonstrations mathématiques "papier-crayon".

Dans la section 3, nous avons parlé de types inductifs (les entiers naturels par exemple). Mais nous n'avons pas parlé de type de type, de hiérarchie d'univers, de sortes. Ces notions qui font partie des la théorie de types de Martin-Löf, forment la base du système de type de Rocq et elles garantissent la cohérence logique du système. Néanmoins, il est tout à fait possible d'utiliser Rocq sans en connaître toute sa théorie.

3.2.1. *Au lycée.* — Nous nous intéressons aux fonctions affines. Ces exemples, sous forme de travaux pratiques, ont été fait par des élèves de classe de seconde en début d'année (octobre). Le fichier complet se trouve en Annexe A. Dans cette partie, nous ne nous intéressons pas aux nombres réels ni comment ils sont formalisés, nous verrons cela dans la Partie II.

Nous commençons par définir une fonction affine :

```
(** Soient a et b deux réels fixés mais quelconques. *)
Variables (a b : R).
```

```
(** Soit f la fonction (affine) suivante : *)
Definition f (x : R) : R := a * x + b.
```

On s'intéresse ensuite aux fonctions constantes :

```
(** On commence par un cas assez simple :
   si a = 0, alors f est constante. *)

Definition constante (g : R → R) : Prop := ∀ x y : R, g x = g y.
```

Mais avant de se lancer dans la preuve, il faut dire un peu comment on travaille dans $\mathbb{R}$. On explique aux élèves que l'on a un ensemble de règles de réécritures qu'on peut utiliser pour remplacer des termes par d'autres dans le but (ou dans les hypothèses). Ils ont vu dans des exercices précédents quelques tactiques telles que `intros`,`rewrite`,`unfold`,`reflexivity`,`apply`. Nous pouvons d'ailleurs faire quelques correspondances avec les règles de la déduction naturelles énoncées à la section 3.1.2.

`assumption`	Ax
`intro,intros`	$\Rightarrow i$, $\forall i$, $\neg i$
`apply,destruct`	$\Rightarrow e$, $\forall e$, $\neg e$
`split`	$\wedge i$
`left,right`	$\vee i$
`exists`	$\exists i$
`exfalso`	$\bot\ e$
...	

Voici des exemples d'égalités, qui font partie de la bibliothèque standard des nombres réels de Rocq, que l'on peut utiliser, suivi d'un exemple et d'un exercice :

```
(** La commande Check permet d'afficher un théorème de nom donné *)
Check Rmult_0_r.
Check Rmult_0_l.
Check Rplus_0_r.
Check Rplus_0_l.
(* l est pour "left", à gauche et r pour "right", à droite *)

Theorem exemple_rewrite (x : R) :
  (0 * 2 + x) + 3 * 0 = x.
```

```
Proof.
  (* D'après Rmult_0_r, 3 * 0 peut être remplacé par 0. *)
  rewrite Rmult_0_r.
  (* D'après Rmult_0_l, 0 * 2 peut être remplacé par 0. *)
  rewrite Rmult_0_l.
  (* D'après Rplus_0_l, 0 + x peut être remplacé par x. *)
  rewrite Rplus_0_l.
  (* D'après Rplus_0_r, x + 0 peut être remplacé par x. *)
  rewrite Rplus_0_r.
  (* Les deux membres de l'égalité sont les mêmes, la preuve est finie. *)
  reflexivity.
Qed.

(* À vous *)
Theorem exercice_rewrite (x y : R) :
  ((0 + (y * 0)) + 0 * x) + x = x.
Proof.
  (* Début Solution *)
  rewrite Rplus_0_l, Rmult_0_l, Rmult_0_r, Rplus_0_r, Rplus_0_l.
  reflexivity.
Qed.
  (* Fin Solution *)
```

Les commentaires `(*Début Solution *)` et `(*Fin Solution *)` sont des délimiteurs pour générer automatiquement les sujets pour étudiants et un corrigé possible pour enseignants.

Nous pouvons maintenant montrer le lemme énoncé plus haut :

```
(* Maintenant les fonctions affines constantes.
   Conseil : unfold constante et unfold f pour remplacer
   par les définitions. *)
Theorem a_0_affine_constante: a = 0 → constante f.
Proof.
  (* Début Solution *)
  unfold f, constante.
  intros H x y.
  rewrite H.
  rewrite Rmult_0_l.
  rewrite Rmult_0_l.
```

```
  rewrite Rplus_0_l.
  reflexivity.
Qed.
  (* Fin Solution *)
```

Nous donnons ensuite aux élèves la preuve Rocq commentée de la réciproque.

De la même façon, nous pouvons définir la notions de croissance stricte d'une fonction et quelques propriétés sur les fonctions affines croissantes :

```
(** On passe au cas a > 0. Alors f est strictement croissante. *)

Definition strict_croissante (f : R → R) :=
          ∀ x y : R, x < y → f x < f y.

(** Pour prouver le théorème suivant, vous aurez besoin
    de certains des résultats suivants : *)
Check Rplus_lt_compat_r.
Check Rplus_lt_compat_l.
Check Rmult_lt_compat_r.
Check Rmult_lt_compat_l.

Theorem a_sup_0_affine_croissante :
  a > 0 → strict_croissante f.
Proof.
  (* Début Solution *)
  unfold strict_croissante, f.
  intros H x y H2.
  apply Rplus_lt_compat_r.
  apply Rmult_lt_compat_l.
  exact H.
  exact H2.
Qed.
  (* Fin Solution *)

(** Plus difficile, à faire seulement si vous avez
    bien compris la preuve de affine_constante_a_0 :
```

```
   Théorèmes que vous pouvez utiliser : *)

Check Rplus_lt_reg_r.
Check Rplus_lt_reg_l.
Check Rlt_0_1.

Theorem affine_croissante_a_sup_0 :
  strict_croissante f → 0 < a.
Proof.
  (* Début Solution *)
  unfold strict_croissante, f.
  intros H.
  specialize (H 0 1).
  apply (Rplus_lt_reg_r b).
  rewrite Rmult_0_r in H.
  rewrite Rmult_1_r in H.
  apply H.
  exact Rlt_0_1.
Qed.
  (* Fin Solution *)
```

Ces exemples très simples sont particulièrement adaptés à l'enseignement. D'après nos retours, ils rendent même l'apprentissage plus ludique.

3.2.2. *En cycle L.* — En section 3.1.3 nous avons donné quelques exemple pour illustrer la logique d'ordre supérieure. Nous les retrouvons ici, issus d'un cours de 1er semestre de L1 double licence maths-info. Nous définissons l'union ensembliste et l'inclusion puis nous montrons que pour tout E, F deux ensembles alors $E \subseteq E \cup F$:

```
Definition union {A: Type} (E F : A → Prop) :=
          fun (a : A) ⇒ (E a ∨ F a).

Definition incl {A : Type} (E F : A → Prop) :=
          ∀ (a : A), (E a → F a).

Lemma E_incl_EuF {A : Type} (E F : A → Prop):
      incl E (union E F).
```

```
Proof.
unfold incl, union.
intros a Ha.
now left.
Qed.
```

où `now left` est une notation pour `left;easy`. Cela revient à appliquer la tactique `left` (on choisit la partie gauche d'une disjonction) puis `easy` qui est une tactique automatique qui résout le but, ici on aurait pu écrire `left;assumption`.

Comme nous l'avons vu, l'ordre supérieur permet également de quantifier sur des fonctions :

```
From Stdlib Require Arith.

Lemma fafunction : ∀ f:nat → nat, ∃ n:nat, f(n)>=n.
intros f.
exists 0.
apply PeanoNat.Nat.le_0_l.
Qed.

Search (∀ n:nat, 0 <= n).
```

Notons que lorsque l'on ne connaît pas le nom du lemme à appliquer, il est possible de le rechercher grâce à la commande `Search`.

Nous reprenons maintenant la preuve du lemme classique $\forall P, \neg\neg P \to P$:

```
From Stdlib Require Classical.

Lemma NNPP: ∀ P: Prop, ¬ ¬ P → P.
Proof.
intros P HP.
destruct (Classical_Prop.classic P).
assumption.
exfalso.
apply HP.
assumption.
Qed.
```

En première année de cycle L, la notion de preuve par récurrence peut ne pas être totalement intégrée. Les exercices suivants permettent de faire un rappel de la démonstration par récurrence sur les entiers naturels. Nous allons montrer les deux propriétés suivantes : l'associativité de l'addition et la distributivité. Nous commençons par rappeler la définition de l'addition (vue en section 3.1.1) et, de manière similaire, de la multiplication dans Rocq :

```
Print Nat.add.
(*
Nat.add =
fix add (n m : nat) {struct n} : nat :=
  match n with
  | 0 ⇒ m
  | S p ⇒ S (add p m)
  end
     : nat → nat → nat
*)

Print Nat.mul.
(*
Nat.mul =
fix mul (n m : nat) {struct n} : nat :=
  match n with
  | 0 ⇒ 0
  | S p ⇒ m + mul p m
  end
     : nat → nat → nat
*)
```

Comme nous l'avons vu en section 3.1.1, le Calcul des Constructions inductive apporte un certain nombre de règles de réduction. En particulier, chaque cas (i.e. chaque cas de filtrage de la fonction définie inductivement) pourra se réduite selon sa règle, grâce à la tactique `simpl`. C'est ainsi que, par exemple, lorsque nous devons montrer que

$$\forall m, p : \mathbb{N}, 0 + (m + p) = m + p$$

l'application de la tactique `simpl` effectue une suite de réductions jusqu'à obtenir :

$$\forall m, p : \mathbb{N}, m + p = m + p$$

Notons que si nous avions $\forall m, p : \mathbb{N}, (m + p) + 0 = m + p$, la réduction n'aurait pas pu avoir lieu.

Après les réductions, pour le cas récursif, il reste à appliquer l'hypothèse de récurrence, nommée `IHn`.

Nous avons donc les deux preuves suivante :

```
Theorem plus_associatif : ∀ (n m p : nat), n + (m + p) = (n + m) + p.
Proof.
  (* remarquez que Coq affiche naturellement plus comme étant
  associatif à gauche *)
  (* Début Solution *)
  induction n as [|n IHn].
  reflexivity.
  simpl. intros m p. rewrite IHn. reflexivity.
Qed.
  (* Fin Solution *)

Theorem mul_distributive : ∀ n m p : nat, (n + m) * p = n * p + m * p.
Proof.
  (* Début Solution *)
  intros n.
  induction n.
  simpl. reflexivity.
  intros m p.
  simpl.
  rewrite IHn.
  rewrite plus_associatif.
  reflexivity.
Qed.
    (* Fin Solution *)
```

Il existe des écritures bien plus compactes de ces preuves. Par exemple le point-virgule (`tac1;tac2`) permet d'appliquer `tac2` à tous les sous-buts de `tac1`. Les tirets permettent de structurer la preuve par sous-buts traités. La preuve du lemme précédente peut alors s'écrire :

```
Theorem mul_distributive_s : ∀ n m p : nat, (n + m) * p = n * p + m * p.
Proof.
  intros n; induction n; simpl.
  – reflexivity.
  – intros m p; rewrite IHn; rewrite plus_associatif; reflexivity.
Qed.
```

3.2.3. *Synthèse.* — Les assistants de preuves formelles sont de plus en plus utilisés, également en enseignement. Leur apport est double : d'une part, ils permettent de renforcer la rigueur des apprentissages en mathématiques et en informatique, en obligeant les élèves à expliciter chaque étape du raisonnement, ce qui limite les approximations et les erreurs implicites. D'autre part, ils favorisent une meilleure compréhension des concepts abstraits en rendant visibles les structures logiques sous-jacentes. En retour, l'enseignement influence lui aussi l'évolution de ces outils de preuves formelles : les besoins pédagogiques conduisent à concevoir des interfaces plus accessibles, des langages plus intuitifs, des approches adaptées, des réflexions quant aux niveaux d'abstraction adaptés aux différents utilisateurs visés, afin de démocratiser leur usage au-delà des spécialistes. Cette interaction réciproque contribue aujourd'hui à rapprocher recherche et enseignement.

Nous verrons, dans la partie II, après avoir décrit certains choix de formalisation concernant les nombres réels, des lemmes et des preuves d'analyse réelle. Nous verrons également que ces réflexions et cette réciprocité s'applique également à de nombreux autres domaines de recherche, par exemple à l'analyse numérique, pour les numériciens.

PARTIE II. L'ANALYSE RÉELLE ET NUMÉRIQUE EN ROCQ : RECHERCHE ET ENSEIGNEMENT

Dans cette seconde partie, nous nous focaliserons sur le cas des nombres réels, un sujet à part entière dans les prouveurs. Nous mettrons en évidence les difficultés de formalisation dans les prouveurs. Nous verrons d'ailleurs que ces développements sont récents au regard d'autres structures mathématiques.

Nous présenterons ensuite la bibliothèque standard Reals de Rocq, en détaillant ses choix de formalisation. Des exemples d'analyse réelle, issus de l'enseignement, serviront de fil conducteur. Ils permettront d'illustrer concrètement l'utilisation de ces outils dans un cadre familier aux mathématiciens.

Enfin, nous évoquerons brièvement la formalisation d'exemples non triviaux d'analyse numérique destinuée aux numériciens.

4. Des formalisations des réels

Les mots "*n*ombres réels" sont souvent employés comme terme générique afin de représenter différentes notions selon le milieu scientifique qui les utilise. En effet, loin des formalisations mathématiques, qui sont déjà nombreuses, nous trouvons les nouvelles formalisations introduites par la science de l'informatique [7].

4.1. Quels nombres réels ? — Comme "formalisations des nombres réels", nous pouvons citer, entre autres, les définitions mathématiques théoriques [**Bou66**], les travaux dans la domaine de l'informatique sur l'arithmétique exacte [**Vui90, MM94, EPE96, O'C08, KS11**], les nombres flottants dans les langages de programmation, les formalisations dans les systèmes formels [**Dut96, Got00, May01, GWZ00, Har98, How86**] et dans le domaine du calcul formel [**Rio91**].

7. Cette section s'inspire de descriptions issues de [**May01, May12**]

Dans les langages de programmation, qui servent à faire des calculs, on trouve les nombres à virgule flottante (ou nombres flottants). Ces nombres ne sont pas des nombres réels, mais en sont une approximation plus ou moins satisfaisante. En effet, on dénombre, en plus des résultats erronés, certaines propriétés élémentaires du corps des réels non respectées. Les deux exemples qui vont suivre illustrent ces deux problèmes. Pour plus de détails nous pourrons consulter **[MM94, Vui90, Mul89, MBdD+10]**. La suite $(a_n)_{n\in\mathbb{N}}$ suivante, due à Jean-Michel Muller, est un exemple, parmi d'autres, des problèmes liés aux erreurs d'arrondi :

$$a_0 = \frac{11}{2},\ a_1 = \frac{61}{11},\ a_{n+1} = 111 - \frac{1130 - 3000/a_{n-1}}{a_n}$$

Bien que la notion de calcul dans un langage de programmation n'est en aucun cas une démonstration, on peut tout de même espérer retrouver un résultat "assez proche" de celui donné par une preuve mathématique (informelle ou formelle). Par récurrence on montre que

$$a_n = \frac{6^{n+1} + 5^{n+1}}{6^n + 5^n}$$

dont la limite à l'infini vaut 6.
En effectuant les calculs, en définissant une fonction récursive, en double précision (dans OCaml), on obtient les valeurs suivantes :

n	2	14	15	16	17	18	19	21	25	26
a_n	5.59	15.41	67.47	97.14	99.82	99.98	99.99	99.99	100.	100.

D'après ces résultats on aurait tendance à croire que cette suite converge vers 100 et non vers 6.

L'exemple qui suit va nous permettre de mieux comprendre pourquoi nous ne pouvons utiliser les nombres flottants tels qu'ils sont définis dans ces langages de programmation. En particulier, la propriété d'associativité de l'addition n'est pas respectée. Les calculs sont, ici encore, effectués en OCaml.

$$\begin{cases} A = (1000 + -1000) + 7.501 = 7.501 \\ B = 1000 + (-1000 + 7.501) = 7.500999999999997635 \end{cases}$$

$$\begin{cases} A = (10e16 + -10e16) + 7.501 = 7.501 \\ B = 10e16 + (-10e16 + 7.501) = 0 \end{cases}$$

$$\begin{cases} A = (10e15 + -10e15) + 7.501 = 7.501 \\ B = 10e15 + (-10e15 + 7.501) = 8 \end{cases}$$

Ces dernières décennies ont également vue progresser les bibliothèques basées sur les nombres à virgule flottante. Les erreurs d'arrondi sont étudiées (et même prouvées) avec une extrême attention, ce qui garantit certaines propriétés importantes. Par exemple, nous pouvons citer la bibliothèque MPFR [**FHL+07**] qui est une bibliothèque C de flottants multi-précision avec arrondi correct.

Dans les systèmes de calcul formel, nous trouvons plusieurs formes de définition pour les nombres réels. En plus des flottants en précision arbitraire (où il est possible de choisir une précision fixe, 1000 chiffres décimaux, par exemple, dans Maple [**Rio91**]), il existe une couche supplémentaire formelle. La donnée $sin(1)$ peut être vue comme la valeur formelle ou comme sa valeur flottante 0.8414709848. Néanmoins, même si ces systèmes sont plus valides que les langages de programmation pour corroborer certaines propriétés, ils ne permettent pas de les vérifier de façon sûre. En effet, le contre-exemple cité précédemment concernant la suite a_n reste valable, c'est-à-dire que la suite continue également de converger vers 100 dans les systèmes de calcul formel.

Pour cette raison, il est impératif d'avoir de "vrais" nombres réels si l'on veut pouvoir faire des preuves qui doivent utiliser les propriétés de corps des nombres réels.

Avant d'entrer dans les détails de choix de formalisation dans les prouveurs, il convient de citer les principales différentes notions de nombres réels :

— réels exacts,
— réels non standards,
— réels classiques,
— réels intuitionnistes

ainsi que de leurs représentations numériques comme

— les nombres flottants,
— l'arithmétique d'intervalle.

4.2. Formalisations mathématiques. — Dans cette section, nous nous intéresserons principalement aux nombres réels classiques, même si nous décrirons brièvement la problématique des réels intuitionnistes.

Il existe plusieurs manières de définir le corps de nombres réels. Nous pouvons soit les construire, soit les axiomatiser.

Les deux constructions les plus usuelles, que nous ne décrirons pas ici, sont la méthode de Cantor et les coupures de Dedekind. Ces méthodes peuvent être adaptées presque directement à la logique intuitionniste [**Bis67**].

En ce qui concerne la méthode axiomatique, cela revient à poser un ensemble minimal d'axiomes permettant de spécifier l'ensemble des réels. Des axiomatisations classiques ou intuitionnistes sont possibles. Dans ce dernier cas, moins usuel, le problème principal vient de l'égalité : de manière générale, il n'est pas possible de décider si deux nombres réels sont égaux ou non. En pratique, cela conduit à introduire des relations d'équivalence plus faibles, telles que l'*apartness* et l'*equivalence*, comme des inégalités approchées, ou à travailler avec des représentations permettant de contourner cette indécidabilité. L'idée est que si nous ne savons pas si deux réels sont égaux, il est par contre généralement possible de savoir s'ils ne le sont pas. Si nous voyons le symbole d'apartness (noté @) comme un $\neq$, nous pouvons alors poser l'axiome de négation de la non réflexivité :

$$\forall x, \neg(@\ x\ x)$$

De la même manière,

$$\forall x, y, (@\ x\ y) \Rightarrow (@\ y\ x)$$

Nous avons également

$$\forall x, y, (@\ x\ y) \Rightarrow \forall z, (@\ x\ z) \vee (@\ z\ y)$$

ainsi que l'équivalence

$$\forall x, y, \neg(@\ x\ y) \Leftrightarrow (\equiv\ x\ y)$$

4.3. Quelques exemples de formalisations dans les prouveurs. — Nous commencerons par un bref historique avant de

nous intéresser ici aux formalisation des nombres réels dans Mizar, HOL-Light, PVS et Lean. Une description plus détaillée se trouve en [**BLM16**]. La cas de Rocq sera décrit plus précisément en section 5.

Les débuts des nombres réels dans les prouveurs sont détaillés dans [**May01**] :

— 1985 : construction intuitionniste par D.J. Howe dans Nuprl ;
— 1991 : axiomatisation classique par C. Jones dans Lego ;
— 1995 : construction classique par J. Harrison dans HOL ;
— 1996,2000 : axiomatisation classique par B. Dutertre et A. Gottliebsen dans PVS ;
— 1997 : axiomatisation classique par M. Mayero dans Coq ;
— 1998 : construction classique par A. Trybulec dans Mizar
— 1999 : réels non standard par R. Gamboa dans ACL2 ;
— 2000 : axiomatisation intuitionniste par M. Niqui dans Coq ;
— 2000 : réels non standard par J. Fleuriot dans Isabelle.
— 2018 : construction classique par M. Carneiro dans Lean

4.3.1. *Mizar.* — Nous rappelons que Mizar est basé sur la théorie des ensembles de Tarski-Grothendieck (une extension de Zermelo-Fraenkel). Bien qu'il s'agisse à l'origine d'une axiomatisation, la formalisation des nombres réels dans Mizar a ensuite été remplacée par une construction fondée sur les coupures de Dedekind (Trybulec 1998). En voici un extrait :

```
func DEDEKIND_CUTS -> Subset-Family of RAT+ equals
{ A where A is Subset of RAT+ :
for r being Element of RAT+ str in A holds
( ( for s being Element of RAT+ st s <=' r holds s in A ) &
ex s being Element of RAT+ st ( s in A & r < s ) )
} \ { RAT+};
```

ce qui signifie que :

A est une coupure de Dedekind si c'est un sous-ensemble strict de $\mathbb{Q}^+ \cup \{0\}$ tel que $\forall r \in A$, $(\forall s \in \mathbb{Q}^+ \cup \{0\}, s \leqslant r \Rightarrow s \in A) \wedge (\exists s \in \mathbb{Q}^+ \cup \{0\}, r < s \wedge s \in A)$

Et nous avons alors par exemple l'addition :

```
func A + B -> Element of DEDEKIND_CUTS equals
{ ( r + s ) where r , s is Element of RAT+ : ( r in A & s in B ) };
```

4.3.2. *HOL-Light.* — Nous rappelons que HOL-Light est basée sur la logique classique d'ordre supérieur avec les axiomes de l'infini, de l'extensionalité et du choix sous la forme de l'opérateur ϵ de Hilbert.

Plutôt que d'utiliser des suites complètes de nombres rationnels, les nombres réels sont basés sur des suites quasi-additives de nombres naturels.

Le prédicat `is_nadd` caractérisant une telle séquence `x` est donné ci-dessous, avec `dist` la fonction renvoyant la distance entre deux nombres naturels.

```
let is_nadd = new definition
‘is_nadd x <=> (?B. !m n. dist (m*x(n), n*x(m))<=B*(m+n))‘;;

let nadd_le = new definition
‘x <<= y <=> ?B. !n. x(n) <= y(n)+B‘;;
```

En d'autres termes, l'existence d'une borne supérieure minimale pour tout ensemble non vide et borné de suites quasi-additives ; ce théorème est ensuite étendu pour démontrer la complétude des nombres réels ; l'addition de deux suites quasi-additives s'effectue point par point ; la multiplication correspond à la composition des suites ; l'ensemble de la droite réelle est construit comme une succession de types de quotients.

4.3.3. *PVS.* — Nous rappelons que la logique de PVS repose sur la logique classique d'ordre supérieur.

Grâce à la prise en charge du sous-typage, PVS adopte une approche descendante. Il part d'un type numérique qui est un sur-ensemble de tous les nombres. Cet ensemble englobe les entiers (issus de Lisp), les rationnels, les nombres réels. Voici un extrait des nombres réels de PVS.

```
number : NONEMPTYTYPE
number_field : NONEMPTYTYPE FROM number
n0x : VAR nonzero_number
inverse_mult : AXIOM n0x * (1/ n0x ) = 1

real : NONEMPTYTYPE FROM number_field
```

```
nonzero_real : NONEMPTYTYPE = { r : real | r /= 0} CONTAINING 1
nzreal_is_nznum : JUDGEMENT nonzero_real SUBTYPE OF nonzero_number
x , y : VAR real
posreal_add_closed : POSTULATE x > 0 AND y > 0 IMPLIES x + y > 0
trichotomy : POSTULATE x > 0 OR x = 0 OR 0 > x
```

POSTULAT signifie que les procédures de décision de PVS permettent de démontrer ces propriétés : c'est la combinaison des axiomes explicites issus des théories et des axiomes implicites issus des procédures de décision qui définit ce que sont les nombres réels dans PVS.

4.3.4. *Lean.* — Nous rappelons que la logique de Lean repose sur le calcul des constructions inductives, étendu aux types quotients.

Le type des nombres réels est construit comme classes d'équivalence de suites de Cauchy de nombres rationnels.

```
structure real := of_cauchy ::
(cauchy : cau_seq.completion.Cauchy (abs : Q → Q))

def Cauchy := @quotient (cau_seq _ abv) cau_seq.equiv

instance equiv : setoid (cau_seq B abv) :=...
```

Nous rappelons également que Mathlib est une bibliothèque unifiée de mathématiques formalisées. Elle se distingue par des fondements à typage dépendant, et est, de par sa conception, entièrement dédiée aux mathématiques classiques.

5. Les réels standard en Rocq

La bibliothèque standard des réels (Reals) de Coq, développée entre 1998 et 2000 par Mayero [**May01**] est une axiomatisation classique comprenant 17 axiomes. Les principales caractéristiques de cette approche sont que les opérations sont modélisées comme des fonctions totales et que l'égalité de deux nombres réels est décidable, ce qui reflète son aspect classique.

EN 2000, la bibliothèque C-CoRN, proposée par Niqui et Geuvers [**GN02**], adopte au contraire une approche constructive des nombres réels. Elle met l'accent sur des définitions effectives et

calculables. Nous avons décrit en section4 la notion d'apartnesse et d'équivalence.

En 2013, Coquelicot, développée par Boldo, Lelay et Melquiond [**BLM15**], étend de manière conservative la bibliothèque standard des réels. Elle vise à être plus conviviale et moderne pour l'utilisateur, notamment dans l'écriture des formules. Cette bibliothèque privilégie l'usage de fonctions totales plutôt que des types dépendants, une hiérarchie algébrique et une abstraction sous forme de filtres. Cela simplifie l'expression des limites, dérivées, intégrales et séries de puissances.

En 2020, la bibliothèque Reals est mise à jour par Semeria [**Sem20**] et introduit une évolution importante, dont l'idée apparaît dans [**May12**]. Elle propose une partie commune reliant les approches constructive et classique.

Il existe une autre bibliothèque, mathcomp-analysis. Les réels sont intégrés dans une hiérarchie de structures définie dans un fichier faisant un lien entre l'univers de MathComp et Reals. Cette approche favorise des preuves courtes, modulaires et réutilisables, mais demande une maîtrise plus avancée de la hiérarchie des structures de MathComp qui s'adaptent difficilement à l'enseignement ou aux utilisateurs novices.

Dans la suite, nous présenterons le socle classique de Reals, qui constitue la base sur laquelle reposent les autres développements.

Cette bibliothèque est une axiomatisation basée sur le fait que $\mathbb{R}$ est un corps ordonné archimédien et complet. Comme pour toute axiomatisation, des constantes sont nécessaires :

$0 : \mathtt{R}$
$1 : \mathtt{R}$
$+ : \mathtt{R} \to \mathtt{R} \to \mathtt{R}$
$* : \mathtt{R} \to \mathtt{R} \to \mathtt{R}$
$- : \mathtt{R} \to \mathtt{R}$
$/ : \mathtt{R} \to \mathtt{R}$
$< : \mathtt{R} \to \mathtt{R} \to \mathtt{Prop}$
$= : \mathtt{R} \to \mathtt{R} \to \mathtt{Prop}$

Les autres constantes ($>$, $\leqslant$, $\geq$, $-$, $/$) sont définies à partir des précédentes :

```
r1 > r2 : r2 < r1
r1 <= r2 : r1 < r2 ∨ r1 = r2
r1 >= r2 : r1 > r2 ∨ r1 = r2
r1 − r2 : r1 + − r2
r1 / r2 : r1 * / r2
```

Les axiomes sont les suivants :

```
1 (*commutativité de l'addition:*) ∀ r1,r2:ℝ, r1+r2=r2+r1
2 (*associativité de l'addition:*)
  ∀ r1,r2,r3:ℝ, (r1+r2)+r3=r1+(r2+r3)
3 (*opposé:*) ∀ r:ℝ,r+ −r=0
4 (*élément neutre de l'addition:*) ∀ r:ℝ,0+r=r
5 (*commutativité de la multiplication:*)
  ∀ r1,r2:ℝ, r1*r2=r2* r1
6 (*associativité de la multiplication:*)
  ∀ r1,r2,r3:ℝ, (r1*r2)*r3=r1*(r2*r3)
7 (*inverse:*) ∀ r:ℝ, si r\neq 0 alors \frac{r}{r}=1
8 (*élément neutre de la multiplication:*) ∀ r:ℝ,1*r=r
9 (*anneau: *) ¬ (1=0)
10 (*distributivité: *)
   ∀ r1,r2,r3:ℝ,r1*(r2+r3)=r1*r2+r1*r3
11 (*ordre total: *)
   ∀ r1,r2:ℝ,r1<r2 \mbox{ ou } r1=r2 \mbox{ ou } r1>r2
12 (*asymétrie de lt: *)
   ∀ r1,r2:ℝ, si r1<r2 alors ¬ (r2< r1)
13 (*transitivité de lt: *)
   ∀ r1,r2,r3:ℝ, si r1<r2 ∧ r2<r3 alors r1<r3
14 (*compatibilité pour l'addition: *) ∀ r,r1,r2:ℝ, si r1<r2
   alors r+r1<r+r2
15 (*compatibilité pour mult: *) ∀ r,r1,r2:ℝ,
   si 0<r ∧ r1<r2 alors r*r1<r*r2
16 (*Archimédien: *) ∀ r:ℝ,∃ n:\N, r<n ∧ n−r<= 1
17 (*complet: *) tout ensemble E borné et non vide de ℝ
   admet une borne supérieure
```

La plupart de ces propriétés sont usuelles d'un point de vue strictement mathématique, mais du point de vue de la théorie des types et de Rocq en particulier, nous allons en détailler quelques unes.

Le symbole et l'axiome pour l'inverse /. — : Nous avons fait le choix de définir la fonction inverse comme une $f$onction totale, i.e. de type R→ R. Nous aurions pu choisir de la définir en tant que fonction partielle, i.e. ∀ x:R, x≠ 0 → R. Dans ce dernier cas, à chaque écriture du symbole inverse, une preuve du fait que l'inverse doit être différent de zéro est alors nécessaire. Par contre, avec une fonction inverse totale, il est possible d'écrire $\frac{1}{0}$, ce qui n'est pas un problème. En effet, Il est possible de donner une valeur arbitraire à l'interprétation de $\frac{1}{0}$ tout en préservant la cohérence grâce à l'axiome 7, qui, lui, requiert l'hypothèse que l'inverse doit être différent de zéro pour autoriser la simplification. L'équivalence entre les deux méthodes est montrée dans [**May01**].

L'axiome d'ordre total. — :

En Rocq il existe plusieurs méthodes pour énoncer la totalité de l'ordre car il y a 2 manières d'écrire une disjonction. Néanmoins, nous n'allons pas les décrire ici, il faudrait pour cela rentrer un peu plus dans les détails de la théorie des types. Ce qu'il est important de comprendre, c'est que l'axiome 11 est un axiome qui permet de décider de l'égalité de deux réels, qui est cet un axiome de la logique classique, étant donné que l'égalité des réels n'est pas décidable. Á titre de comparaison, la même propriété dans les entiers naturels est intuitionniste puisque l'égalité y est décidable.

L'axiome d'Archimed. — :

Nous notons que la seconde partie de l'axiome est plus forte que l'axiome d'Archimède usuel. Ce choix est historique, il permettait juste de définir les parties entière et fractionnaire directement [**May00**]. Cet énoncé est équivalent à l'énoncé usuel ∀ r:R, IZR (up r) > r et la preuve [**May01**] est triviale en utilisant le fait qu'un ensemble ordonné et minoré admet un plus petit élément.

A partir de ces 17 axiomes, nous sommes maintenant en mesure de prouver n'importe quel théorème nécessitant des nombres réels. Nous allons en voir quelques aperçus dans les sections suivantes.

6. Enseigner l'analyse avec Rocq

Nous présentons ici quelques théorèmes d'analyse sur les suites et les coefficients binomiaux. Il s'agit de feuilles d'exercices destinées aux étudiants de cycle L [**KLRM+22, KMR24, BCH+24**].

Ces feuilles se veulent quasiment auto-suffisantes, les étudiants pouvant les suivre pas à pas en autonomie.

Nous énonçons et montrons ici pas à pas aux étudiants l'unicité de la limite d'une suite convergente, en ayant pris soin avant d'expliquer la définition de la limite (voir Annexe B).

```
Theorem UL_sequence:
  ∀ (Un : nat → R) (l1 l2 : R), Un_cv Un l1 → Un_cv Un l2 → l1 = l2.
Proof.
  unfold Un_cv.
  intros Un l1 l2 Hl1 Hl2.
  (* On va montrer que la distance entre l1 et l2 est aussi
  petite qu'on veut.*)
  apply small_dist_equal.
  (* Soit eps > 0. *)
  intros eps Heps.
  assert (Heps2 : eps / 2 > 0) by lra. (* eps / 2 > 0 *)
  (* Soit n1 tel que pour tout n >= n1, |Un - l1| < eps / 2. *)
  destruct (Hl1 (eps / 2) Heps2) as [n1 Hn1].
  (* Soit n2 tel que pour tout n >= n2, |Un - l2| < eps / 2. *)
  destruct (Hl2 (eps / 2) Heps2) as [n2 Hn2].
  (* Soit n3 = max(n1, n2). *)
  pose (n3 := (max n1 n2)).
  (* Alors, |Un3 - l1| < eps / 2 *)
  assert (Hdistl1: R_dist (Un n3) l1 < eps / 2). {
    apply Hn1.
    apply Nat.le_max_l. (* max(n1, n2) >= n1 *)
  }
  (* et |Un3 - l2| < eps / 2 *)
  assert (Hdistl2: R_dist (Un n3) l2 < eps / 2). {
    apply Hn2.
    apply Nat.le_max_r. (* max(n1, n2) >= n2 *)
  }
```

```
  (* Il suffit de montrer (par transitivité de <) que
    |l1 - l2| < |l1 - Un3| + |Un3 - l2| et
    |l1 - Un3| + |Un3 - l2| < eps *)
  apply Rle_lt_trans with
    (r2 := R_dist (Un n3) l1 + R_dist (Un n3) l2).
    - (* La première inégalité est l'inégalité triangulaire. *)
    rewrite (R_dist_sym (Un n3)).
    apply R_dist_tri.
    - (* La seconde en sommant les deux inégalités précédentes. *)
    replace eps with (eps/2 + eps/2) by lra.
    now apply Rplus_lt_compat.
Qed.
```

Les étudiants peuvent maintenant montrer d'autres propriétés :

```
Theorem CV_plus (An Bn : nat → R) (l1 l2 : R) :
  let Cn := (λ n ⇒ An n + Bn n) in
  Un_cv An l1 → Un_cv Bn l2 → Un_cv Cn (l1 + l2).
Proof.
  (* Début Solution *)
  unfold Un_cv.
  intros HA HB eps Heps.
  assert (Heps2 : eps / 2 > 0) by lra.
  destruct (HA (eps / 2) Heps2) as [n1 Hn1].
  destruct (HB (eps / 2) Heps2) as [n2 Hn2].
  pose (n3 := (max n1 n2)).
  exists n3.
  intros n Hn.
  assert (Hdistl1: R_dist (An n) l1 < eps / 2). {
    apply Hn1.
    apply Nat.le_trans with (m:=n3).
    - apply Nat.le_max_l.
    - assumption.
  }
  assert (Hdistl2: R_dist (Bn n) l2 < eps / 2). {
    apply Hn2.
    apply Nat.le_trans with (m:=n3).
    - apply Nat.le_max_r.
    - assumption.
```

```
  }
  unfold R_dist.
  replace (An n + Bn n − (l1 + l2)) with ((An n − l1) + (Bn n − l2)) by lra.
  apply Rle_lt_trans with
    (r2 := Rabs (An n − l1) + Rabs (Bn n − l2)).
  − apply Rabs_triang.
  − replace eps with (eps/2 + eps/2) by lra.
    now apply Rplus_lt_compat.
Qed.
(* Fin Solution *)
```

On va maintenant vers un résultat très important : celui qui énonce que les suites convergences sont bornées.

```
(*fr Faire la preuve du lemme suivant et remplacer "Admitted" par "Qed". *)
(*en Fill the proof of next lemma and replace "Admitted" by "Qed". *)
Lemma binom_ex01b n : binom n 5 = 17 * binom n 4 → n < 5 ∨ n = 89.
Proof.
(*fr Indication : discuter suivant que n < 5 ou pas. *)
(*en Hint: discuss wether n < 5 or not. *)
(* Begin solution for the teacher. *)
destruct (nat_lt_le_dec n 5) as [Hn | Hn].
+ intros _; left. assumption.
+ assert (Hn' : 4 <= n).
    apply le_trans with 5. apply le_succ_diag_r. assumption.
  rewrite ← (mul_cancel_l _ _ 5). 2: discriminate.
  rewrite binom_mul_S_r. rewrite mul_assoc. rewrite mul_cancel_r.
  2: apply binom_neq_0; assumption.
  intros H. apply nat_sub_add_eq_l in H. 2: assumption.
  right. subst. reflexivity.
Qed.
(* End solution for the teacher. *)
```

Indépendamment du cours de L1, nous avons développé, dans la cadre du défi Inria LiberAbaci, d'autres feuilles d'exercices destinées au cycle. Par contre, il est à noté que celles-ci n'ont, à notre connaissance, pas encore été utilisées. D'autre part, plusieurs points sont à noter :

— ces feuilles d'exercice peuvent être utilisées également pas des non francophones ;
— elles incluent plusieurs niveaux d'aides i.e. que l'enseignant peut très simplement choisir le niveau d'indication qu'il souhaite fournir.

```
(*fr Faire la preuve du lemme suivant et remplacer
  "Admitted" par "Qed". *)
(*en Fill the proof of next lemma and replace "Admitted" by "Qed". *)
Lemma binom_sum n : sum_range 0 n (λ k ⇒ binom n k) = 2 ^ n.
Proof.
(*fr Indications :
 - Transformer 2 en une somme.
 - Utiliser la formule du binôme.
 - Réduire l'égalité de somme à l'égalité des fonctions. *)
(*en Hints:
 - Transform 2 into a sum.
 - Use binomial theorem.
 - Reduce equality on sums to equality on functions. *)

(* Begin solution for the teacher. *)
(* Without automation. *)
replace 2 with (1+1) by reflexivity.
rewrite binom_formula.
apply sum_range_ext. intros k Hk.
rewrite 2 pow_1_l.
rewrite 2 mul_1_r. reflexivity.
Qed.
(* End solution for the teacher. *)
```

Ces exercices s'appuient sur une bibliothèque de théorèmes prouvés, qui peut être appelée, ou pas, également suivant le niveau de difficulté souhaité.

```
(*fr Formule de la gouttière. *) (*en Hockey stick identity. *)
(*fr Faire la preuve du lemme suivant et remplacer
  "Admitted" par "Qed". *)
(*en Fill the proof of next lemma and replace "Admitted" by "Qed". *)
Lemma binom_hockey_stick n k :
```

```
  sum_range k n (λ i ⇒ binom i k) = binom (S n) (S k).
Proof.
(*fr1 Indications :
 - Commencer par énoncer une version équivalente en remplaçant
   "sum_range k n" par "sum_n k p", où  p représente le nombre de
   termes sommés, c'est-à -dire n-k+1.
   Donc "binom (S n) (S k)" devient "binom (k + p) (S k)" et il faut prouver
   "∀ p k, sum_n k p (λ i ⇒ binom i k) = binom (k + p) (S k)".
 - Prouver cette version par induction sur p, en utilisant la
   formule de Pascal pour le cas successeur.
 - Puis faire apparaître "sum_n", utiliser le résultat prouvé
   précédemment, et discuter suivant que "S n < k" ou pas. *)
(*en1 Hints:
 - First state an equivalent version by replacing "sum_range k n"
   by "sum_n k p", where p stands for the number of added terms, ie n-k+1.
   Thus "binom (S n) (S k)" becomes "binom (k + p) (S k)" and we have to prove
   "∀ p k, sum_n k p (λ i ⇒ binom i k) = binom (k + p) (S k)".
 - Prove this version by induction on p. Use Pascal's rule for the
   successor case.
 - Then, make "sum_n" visible, use the result proved previously, and discuss
   whether "S n < k" or not. *)

(*fr2 Indications :
 - Commencer par énoncer une version équivalente en remplaçant
   "sum_range k n" par "sum_n k p", où  p représente le nombre de
   termes sommés, c'est-à -dire n-k+1, donc "binom (S n) (S k)" devient
   "binom (k + p) (S k)".
 - Prouver cette version par induction sur p (penser à la formule de
   Pascal).
 - Utiliser le résultat prouvé précédemment, et discuter suivant que
   "S n < k" ou pas. *)
(*en2 Hints:
 - First state an equivalent version by replacing "sum_range k n"
   by "sum_n k p", where p stands for the number of added terms, ie
   n-k+1, thus "binom (S n) (S k)" becomes "binom (k + p) (S k)".
 - Prove this version by induction on p (think about Pascal's rule).
 - Then, use the result proved previously, and discuss whether
```

```
  "S n < k" or not. *)

(*fr3 Indications :
 - Commencer par énoncer une version équivalente en remplaçant
   "sum_range k n" par "sum_n k p", où  p représente le nombre de
   termes sommés.
 - Prouver cette version par induction sur p, puis utiliser ce
   résultat intermédiaire. *)
(*en3 Hints:
 - First state an equivalent version by replacing "sum_range k n"
   by "sum_n k p", where p stands for the number of added terms.
 - Prove this version by induction on p, then use this intermediary result. *)

(* Begin solution for the teacher. *)
assert (H : ∀ p k, sum_n k p (λ  i ⇒ binom i k) = binom (k + p) (S k)).
intros p. induction p.
— intros. simpl. rewrite binom_eq_0.
  reflexivity.
  rewrite add_0_r. apply lt_succ_diag_r.
— intros. simpl. rewrite IHp.
  rewrite add_succ_r. rewrite ← binom_pascal. apply add_comm.
(* *)
— unfold sum_range; rewrite H. destruct (nat_lt_le_dec (S n) k) as [H1 | H1].
  + rewrite 2 binom_eq_0. (* lia solves next subgoals. *)
    reflexivity.
    apply lt_trans with k. assumption. apply lt_succ_diag_r.
    rewrite (nat_sub_0_lt _ _ H1). rewrite add_0_r. apply lt_succ_diag_r.
  + f_equal. (* lia solves the next subgoal. *)
    rewrite (add_sub_assoc _ _ _ H1).
    rewrite add_comm, add_sub. reflexivity.
Qed.
(* End solution for the teacher. *)
```

7. Formalisation de l'analyse numérique : les éléments finis

Cette section est une section dédiée à des activités de recherche autour de la formalisation de l'analyse numérique en Rocq. Ces résultats sont le fruit de plusieurs années d'amélioration des outils de preuve formelle et de Rocq en particulier. Nous présenterons ici, sans toutefois rentrer dans les détails des preuves mathématiques ni Rocq, l'état des lieux [8]

Notre objectif à long terme est d'offrir davantage de garanties pour les programmes numériques simulant des systèmes physiques. Ceux-ci sont en effet largement utilisés, dans le domaine de la santé, du génie civil, de l'aéronautique, de la climatologie, de l'astrophysique, etc. Dans ces travaux nous abordons l'une des méthodes les plus courantes pour résoudre les équations aux dérivées partielles (EDP), à savoir la méthode des éléments finis (MEF). Nous nous concentrons sur la formalisation des éléments finis (EF), et plus particulièrement sur la construction de l'EF la plus couramment utilisée, l'EF de Lagrange simpliciaux. Il s'agit d'une première étape nécessaire vers une preuve formelle de l'ensemble de la méthode des éléments finis.

De manière informelle, la méthode des éléments finis est utilisée pour approcher la solution des équations différentielles partielles. Ces équations servent à modéliser un large éventail de problèmes, notamment la mécanique des fluides (équations de Navier-Stokes), l'électromagnétisme (équations de Maxwell), l'équation de la chaleur (Fourier), la mécanique (élasticité), la mécanique quantique (Schrödinger et Heisenberg), la prévision météorologique et l'aéronautique, entre autres.

Les solutions exactes de ces EDP ne sont généralement pas calculables. Pour pouvoir calculer une solution approximative, l'idée consiste à discrétiser, c'est-à-dire à découper le problème à résoudre en petits morceaux. Pour cela, nous créons un maillage constitué d'éléments géométriques « simples » couvrant la surface ou le volume total, sur lesquels nous savons comment effectuer les calculs (4). Nous passons alors d'un problème continu à un problème discret. Ces éléments « de calcul » sont appelés « éléments finis ». Pour effectuer

8. Cette présentation est en partie issue de [**BCM+25**]. Il s'agit d'un travail commun avec plusieurs co-auteurs.

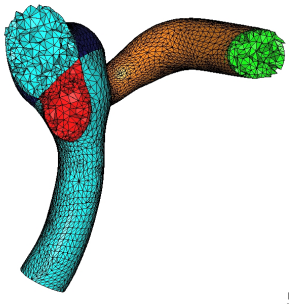

FIGURE 4. Maillage 3D d'un anévrisme cérébral (les couleurs servent uniquement à distinguer les zones).

des calculs approximatifs sur ces différents éléments, un changement de système de coordonnées est nécessaire. Cela implique de changer la représentation de l'espace d'un « élément courant » à l'« élément de référence », où les calculs sont plus simples (voir la figure 5) ; par exemple en 2D, on peut prendre le triangle rectangle unité comme élément de référence. Le lien entre les éléments de référence et actuels est également utilisé pour établir les propriétés d'approximation des EF [**EG21a**, Chap. 11]. La technique consistant à utiliser les éléments finis pour approcher la solution du problème global est la « méthode des éléments finis ».

Par exemple, la figure 4 représente une application de la méthode des éléments finis au domaine de la santé. Il s'agit d'un maillage 3D d'un anévrisme cérébral utilisé pour calculer le débit sanguin à l'intérieur de l'artère (à l'aide de l'équation de Navier-Stokes) et la mécanique des parois, avec interaction fluide-structure entre la paroi et le sang ; voir par exemple [**FGM08**].

Une EF n'est pas seulement une forme géométrique, c'est un triplet (K, P, Σ). En termes simples, K représente la géométrie, P représente l'ensemble des polynômes d'approximation, et Σ est un moyen de caractériser les polynômes, par exemple par les valeurs prises en certains nœuds (les points bleus de la figure 5). De plus, une EF "correcte" doit satisfaire la propriété d'unisolvence, ce qui signifie que la caractérisation ci-dessus est unique. Il existe plusieurs types de EF, tels que Lagrange, Raviart-Thomas, le choix dépendant des équations à approcher.

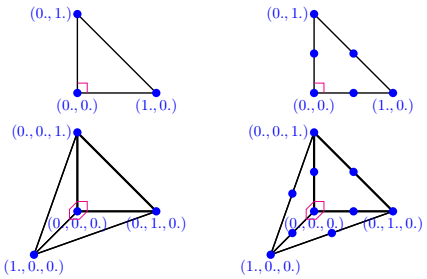

FIGURE 5. Référence : Éléments finis de Lagrange ${}^{Lag}\mathbb{P}_k^d$ en dimension $d = 2$ (en haut) et $d = 3$ (en bas), avec des degrés d'approximation $k = 1$ (à gauche) et $k = 2$ (à droite). Les éléments géométriques sont des d-simples. Les nœuds (représentés par des points bleus) ne sont que les sommets lorsque $k = 1$, et lorsque $k = 2$, des nœuds sont ajoutés au milieu des arêtes. Leurs coordonnées étant des nombres réels, elles sont notées avec un suffixe point, comme dans « 0. » et « 1. ».

Des représentations de l'équation de Lagrange simpliciale sont données dans la figure 5 pour les dimensions spatiales $d = 2$ et 3, et pour les polynômes affines ($k = 1$) et quadratiques ($k = 2$).

D'un point de vue logiciel, il existe de nombreux programmes permettant d'effectuer des calculs d'approximation basés sur la méthode des éléments finis, et de nombreuses bibliothèques d'éléments finis sont disponibles, telles que FreeFEM++,[9] FELiScE,[10] et XLiFE++.[11] Comme dans tout programme, des erreurs sont possibles, qu'elles soient inattendues (bogues) ou attendues, telles que des erreurs de méthode, ainsi que des erreurs dues aux calculs effectués par le processeur, en particulier celles liées à l'arithmétique en virgule flottante. Pour rendre ces programmes sûrs, notre objectif est de prouver formellement en Rocq la correction de leur implémentation, tant en termes d'erreurs de méthode que d'erreurs en virgule flottante. Pour prouver formellement la correction d'un tel programme, nous

9. https://freefem.org
10. https://gitlab.inria.fr/felisce/felisce/
11. https://xlifepp.pages.math.cnrs.fr/

devons commencer par formaliser et prouver les propriétés mathématiques nécessaires dans un assistant de preuve. De plus, depuis les années 2000, ces méthodes nous ont permis de formaliser et de prouver un programme de résolution d'équations d'ondes en une dimension [**BCF+13**], ainsi qu'un certain nombre de propriétés d'analyse réelle nécessaires à notre objectif de démontrer des programmes numériques critiques, tels que le théorème de Lax–Milgram [**BCF+17**], l'intégrale de Lebesgue [**BCF+22, BCM+23**] et l' intégrale de Bochner [**BCL22**]. Ces formalisations sont des pré-requis à la formalisation de la méthode des éléments finis (MEF).

Il convient également de noter que notre formalisation s'adresse aussi bien aux experts qu'aux non-experts en Rocq, y compris aux numériciens et aux étudiants [**BCH+24, KMR24**]. Nous accordons une attention particulière à la lisibilité des énoncés, ce qui guide certaines de nos décisions concernant la formalisation et l'utilisation des bibliothèques existantes. Par "lisibilité", nous entendons avant tout la syntaxe des énoncés des lemmes, ainsi que la recherche d'un juste équilibre en termes d'abstraction des concepts mathématiques utilisés.

Pour donner une idée de l'ampleur du travail, ces travaux de preuves de théorèmes liés aux EF représentent environ une centaine de fichiers, plus de 7 000 énoncés (tels que des lemmes, des définitions et des structures canoniques), pour environ 45 kloc [12] de Rocq.

Tout cela est disponible à l'adresse suivante :

https://lipn.univ-paris13.fr/rocq-num-analysis/tree/2.2.0/

sous la forme des paquets Opam rocq-num-analysis-subset, [13] rocq-num-analysis-algebra, [14] et rocq-num-analysis-fem. [15] Les dépendances respectives de ces paquets sont illustrées dans la figure 6.

Cette formalisation comprend à la fois des composants génériques et des composants dédiés. Parmi les composants génériques pouvant être réutilisés pour d'autres projets, on trouve les familles finies (c'est-à-dire les vecteurs d'une taille donnée), les espaces affines, les sous-structures et les restrictions de fonctions (qui appartiennent aux

12. loc signifie "lines of code"
13. https://rocq-prover.org/p/rocq-num-analysis-subset/2.2.0
14. https://rocq-prover.org/p/rocq-num-analysis-algebra/2.2.0
15. https://rocq-prover.org/p/rocq-num-analysis-fem/2.2.0

paquets rocq-num-analysis-subset et rocq-num-analysis-algebra), mais aussi les multi-indices. Les composants dédiés sont la définition de ce qu'est une EF avec un `Record` précis comprenant à la fois des valeurs et des exigences, et l' instanciation de cet enregistrement dans le cas (courant) de la EF de Lagrange simpliciale.

Ces travaux sur les preuves formelles autour des EF ne sont que le commencement. Il existe plusieurs directions de poursuite possibles. Une approche simple consiste à considérer les équations de Lagrange pour les parallélépipèdes et leurs propriétés. Cela nécessitera de répartir une partie de notre développement entre les équations de Lagrange pour les simplexes et celles pour les parallélépipèdes. Les équations de Lagrange pour les parallélépipèdes sont souvent définies comme la tensorisation des équations de Lagrange en 1D. La preuve d'unisolvence serait plus simple et se ferait uniquement par induction sur la dimension. Ce qui est plus délicat, c'est le fait que la transformation géométrique n'est généralement plus affine, voir [**EG21a**, Sec. 13.5].

Une autre perspective importante et complexe consiste à traiter les éléments de surface, c'est-à-dire les éléments finis de dimension l appartenant à $\mathbb{R}^d$ avec $l < d$ (généralement $l = d - 1$). Ils peuvent être utilisés pour appliquer les conditions aux limites de l'équation différentielle partielle, mais aussi pour le traitement des fractures [**MJR05**], ou pour les méthodes des éléments finis aux limites, voir par exemple [**JR16**] et les références qui y sont citées. Ces éléments de frontière ne s'inscrivent pas dans notre formalisation actuelle. Il serait également intéressant d'étendre l'enregistrement `EF` aux éléments finis plus complexes tels que Raviart-Thomas (EF), voir [**EG21b**, Chap. 14], qui nécessite un développement important du calcul intégral, du calcul différentiel et des composantes géométriques (telles que les intégrales de surface, les normales et les transformations géométriques non affines).

Plus que d'autres éléments finis, nous avons plusieurs perspectives à long terme décrites dans la Figure 6. Nous souhaitons prouver formellement la méthode des éléments finis, c'est-à-dire l'algorithme complet permettant de résoudre une équation différentielle partielle (EDP) où l'EDP est résolue sur chaque élément fini, puis les résultats assemblés.

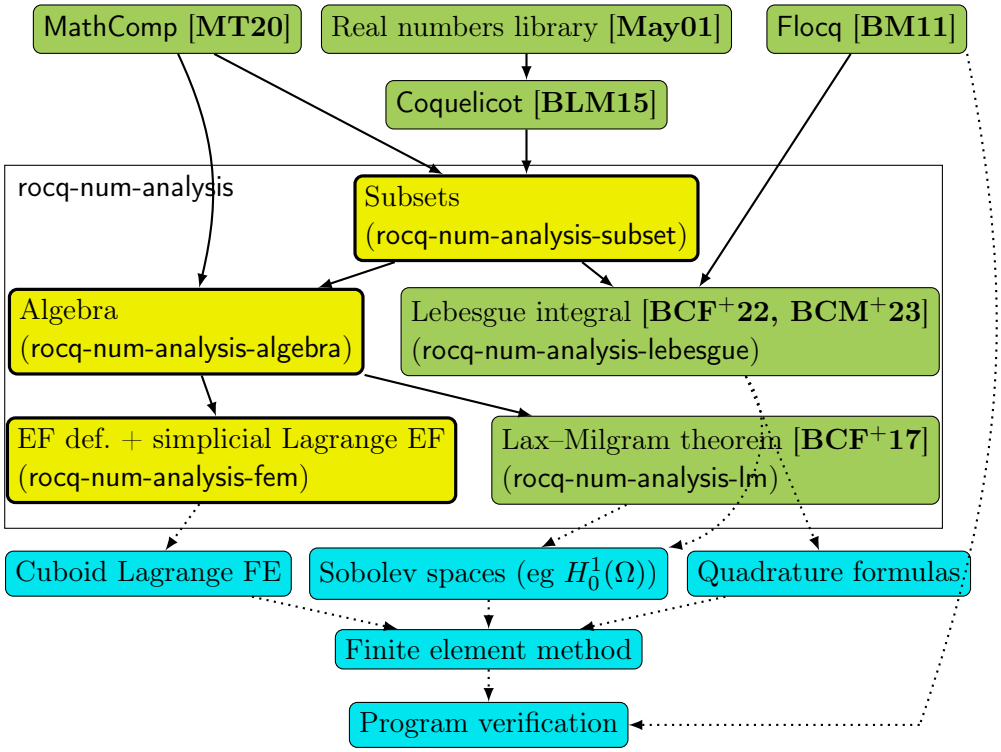

FIGURE 6. Graphique des dépendances, incluant la littérature (en vert), nos contributions présentées dans cet article (en jaune) et les perspectives (en bleu). Les flèches pleines représentent les dépendances actuelles entre les bibliothèques, tandis que les flèches pointillées indiquent les dépendances prévues.

Cette procédure dite d'assemblage est une étape nécessaire à formaliser. Une autre étape consiste à borner l'erreur d'approximation, ce qui nécessite de formaliser les espaces de Sobolev (et de prouver qu'ils sont des espaces de Hilbert). L'objectif final est de prouver formellement la correction d'une bibliothèque (ou d'une partie de bibliothèque) implémentant la méthode des éléments finis, telle que XLiFE++.[16] Il s'agira d'un autre type de preuves : plus que prouver la méthode des éléments finis en tant qu'algorithme mathématique, il faudra prouver qu'elle est conforme au programme en C. Un point spécifique de

16. https://xlifepp.pages.math.cnrs.fr/

la littérature consiste à s'assurer que les erreurs en virgule flottante n'altèrent pas trop les résultats, en s'appuyant sur des démonstrateurs automatiques ou sur la bibliothèque Flocq [**BM11**].

8. Conclusion

Depuis plus de trois décennies, les preuves formelles connaissent une expansion importante, portée par de nouvelles théories logiques et les progrès de l'informatique. La formalisation des mathématiques est devenue un sujet d'actualité majeur, aussi bien dans l'enseignement que dans la recherche. En mathématiques, elle ouvre la voie à une meilleure structuration des connaissances. En informatique, elle joue un rôle central dans la vérification des logiciels et des systèmes critiques. Ses applications dépassent également ces domaines, notamment en médecine, où la fiabilité des modèles et des algorithmes est essentielle. Pourtant, cet essor soulève toujours des défis, comme la complexité des outils ou leur accessibilité. Nous n'en avons pas parlé dans cette présentation, mais l'avènement de l'intelligence artificielle est en train de marquer un nouveau tournant. Elle pourrait accélérer la formalisation, l'automatisation des preuves ainsi que la découverte de nouvelles preuves ou théorèmes. N'omettons pas toutefois d'intégrer des réflexions sur les enjeux éthiques, notamment en matière de fiabilité, de transparence et de responsabilité.

Références

[AIT] « Conference on artificial intelligence and theorem proving » – https ://aitp-conference.org/.

[Ari50] Aristotle – *Organon*, Various editions, -350.

[BCF+13] S. Boldo, F. Clément, J.-C. Filliâtre, M. Mayero, G. Melquiond & P. Weis – « Wave equation numerical resolution : a comprehensive mechanized proof of a C program », *J. Autom. Reason.* **50** (2013), no. 4, p. 423–456.

[BCF+17] S. Boldo, F. Clément, F. Faissole, V. Martin & M. Mayero – « A Coq formal proof of the Lax–Milgram theorem », in *Proc. of the 6th ACM SIGPLAN Internat. Conf. on Certified Programs and Proofs (CPP 2017)* (New York), Association for Computing Machinery, 2017, p. 79–89.

[BCF+22] S. Boldo, F. Clément, F. Faissole, V. Martin & M. Mayero – « A Coq formalization of Lebesgue integration of nonnegative functions », *J. Autom. Reason.* **66** (2022), no. 2, p. 175–213.

[BCH+24] S. Boldo, F. Clément, D. Hamelin, M. Mayero & P. Rousselin – « Teaching divisibility and binomials with Coq », in *Proc. of the 13th Internat. Workshop on Theorem proving components for Educational software (ThEdu 2024)* (J. Narboux, W. Neuper & P. Quaresma, éds.), EPTCS, vol. 419, 2024, p. 124–139.

[BCL22] S. Boldo, F. Clément & L. Leclerc – « A Coq formalization of the Bochner integral », Research Report RR-9456, Inria, 2022.

[BCM+23] S. Boldo, F. Clément, V. Martin, M. Mayero & H. Mouhcine – « A Coq formalization of Lebesgue induction principle and Tonelli's theorem », in *Proc. of the 25th Internat. Symp. on Formal Methods (FM 2023)* (Cham) (M. Chechik, J. Katoen & M. Leucker, éds.), vol. 14000, Springer, 2023, p. 39–55.

[BCM+25] ———, « A Rocq formalization of simplicial Lagrange finite elements », Research Report RR-9590, Inria, 2025.

[Bis67] E. Bishop – « Foundations of constructive analysis », in *New York : Academic Press*, 1967.

[BL09] S. Blazy & X. Leroy – « Mechanized Semantics for the Clight Subset of the C Language », *J. Autom. Reason.* **43** (2009), no. 3, p. 263–288.

[BLM15] S. Boldo, C. Lelay & G. Melquiond – « Coquelicot : A user-friendly library of real analysis for Coq », *Math. Comput. Sci.* **9** (2015), no. 1, p. 41–62.

[BLM16] S. Boldo, C. Lelay & G. Melquiond – « Formalization of real analysis : A survey of proof assistants and libraries », *Math. Struct. Comput. Sci.* **26** (2016), no. 7, p. 1196–1233.

[BM11] S. Boldo & G. Melquiond – « Flocq : A unified library for proving floating-point algorithms in Coq », in *Proc. of the IEEE 20th Symposium on Computer Arithmetic (ARITH 2020)* (Los Alamitos), IEEE, 2011, p. 243–252.

[Boo54] G. Boole – *An investigation of the laws of thought*, Macmillan, 1854.

[Bou66] N. Bourbaki – *Éléments de mathématique. Théorie des ensembles.*, Hermann, 1966.

[CH88] T. Coquand & G. Huet – « The Calculus of Constructions », in *Information and Computation*, vol. 76, 1988, p. 95–120.

[Chu41] A. Church – *The Calculi of Lambda-Conversion*, Annals of Mathematics Studies, vol. 6, Princeton University Press, 1941.

[Com] T. M. Community – « Mizar Mathematical Library », https://mizar.uwb.edu.pl/library/.

[Com17] T. L. Community – « mathlib : The Lean Mathematical Library », 2017, https://github.com/leanprover-community/mathlib.

[Coq85] T. Coquand – « Une théorie des constructions », Thèse, Université Paris 7, 1985.

[CP90] T. Coquand & C. Paulin – « Inductively defined types », in *Proceedings of the International Conference on Computer Logic (COLOG-88)*, Lecture Notes in Computer Science, vol. 417, Springer, 1990, p. 50–66.

[dB70] N. G. de Bruijn – « The Mathematical Language Automath », *Nederl. Akad. Wetensch. Proc. Ser. A* **73** (1970), p. 29–61.

[Dut96] B. Dutertre – « Elements of mathematical analysis in PVS », in *Proc. TPHOL 96*, vol. 1125, Springer LNCS, 1996.

[EG21a] A. Ern & J.-L. Guermond – *Finite elements i. Approximation and interpolation*, Texts in Applied Mathematics, vol. 72, Springer, Cham, 2021.

[EG21b] ———, *Finite elements i. Approximation and interpolation*, Texts in Applied Mathematics, vol. 72, Springer, Cham, 2021.

[EPE96] A. Edalat, P. Potts & M. Escardo – « Semantics of Exact Real Arithmetic », in *Proc. LICS 97*, Springer LNCS, 1996.

[FGM08] M. A. Fernández, J.-F. Gerbeau & V. Martin – « Numerical simulation of blood flows through a porous interface », *M2AN Math. Model. Numer. Anal.* **42** (2008), no. 6, p. 961–990.

[FHL+07] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier & P. Zimmermann – « MPFR : A multiple-precision binary floating-point library with correct rounding », *ACM Trans. Math. Softw.* **33** (2007), no. 2.

[Fre79] G. Frege – *Begriffsschrift*, Louis Nebert, 1879.

[GAA+13] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. L. Roux, A. Mahboubi, R. O'Connor, S. O. Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi & L. Théry – « A Machine-Checked Proof of the Odd Order Theorem », in *Interactive Theorem Proving (ITP) Proceedings* (S. Blazy, C. Paulin-Mohring & D. Pichardie, éds.), Lecture Notes in Computer Science, Springer, 2013, p. 163–179.

[Gen35] G. Gentzen – « Investigations into logical deduction », *Mathematische Zeitschrift* **39** (1935), p. 176–210.

[GN02] H. Geuvers & M. Niqui – « Constructive reals in Coq : Axioms and categoricity », in *Proc. of the Internat. Workshop on Types for Proofs and Programs (TYPES 2000)* (Berlin - Heidelberg) (P. Callaghan, Z. Luo, J. McKinna & R. Pollack, éds.), vol. 2277, Springer, 2002, p. 79–95.

[Göd31] K. Gödel – « Über formal unentscheidbare sätze der principia mathematica und verwandter systeme », *Monatshefte für Mathematik* **38** (1931), p. 173–198.

[Gon08] G. Gonthier – « Formal proof – the four color theorem », *Notices of the AMS* **55** (2008), no. 11, p. 1382–1393.

[Got00] H. Gottliebsen – « Transcendental Functions and Continuity Checking in PVS », in *Proc. TPHOL 2000*, Springer LNCS, Septembre 2000.

[Gou05] J.-B. Gourinat – « La logique : une création de la grèce antique », *Pour la science* (2005), Dossier "La logique, fil d'Ariane du raisonnement".

[GW07] G. Gonthier & B. Werner – « Le théorème des quatre couleurs : ingénierie d'une preuve formelle », (2007).

[GWZ00] H. Geuvers, F. Wiedijk & J. Zwanenburg – « A constructive proof of the fundamental theorem of algebra without using the rationals », in *TYPES*, 2000, p. 96–111.

[HA28] D. Hilbert & W. Ackermann – *Grundzüge der theoretischen logik*, Springer, 1928.

[HAB+15] T. C. Hales, M. Adams, G. Bauer, D. T. Dang, J. Harrison, T. L. Hoang, C. Kaliszyk, V. Magron, S. McLaughlin, T. T. Nguyen, T. Q. Nguyen, T. Nipkow, S. Obua, J. Pleso, J. M. Rute, A. Solovyev, A. H. T. Ta, T. N. Tran, D. T. Trieu, J. Urban, K. K. Vu & R. Zumkeller – « A formal proof of the Kepler conjecture », *CoRR* **abs/1501.02155** (2015).

[Har98] J. Harrison – *Theorem Proving with the Real Numbers*, Springer-Verlag, 1998.

[How80] W. A. Howard – « The formulae-as-types notion of construction », in *To H. B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism* (J. P. Seldin & J. R. Hindley, éds.), Academic Press, 1980, p. 479–490.

[How86] D. J. Howe – « Implementing Analysis », Thèse, Cornell University, 1986.

[JR16] P. Joly & J. Rodríguez – « Mathematical aspects of variational boundary integral equations for time dependent wave propagation », *J. Integral Equ. Appl.* **29** (2016), no. 1, p. 137–187.

[KEH+09] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. A. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch & S. Winwood – « seL4 : formal verification of an OS kernel. », in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (J. N. Matthews & T. E. Anderson, éds.), ACM, 2009, p. 207–220.

[KLRM+22] M. Kerjean, F. Le Roux, P. Massot, M. Mayero, Z. Mesnil, S. Modeste, J. Narboux & P. Rousselin – « Utilisation des assistants de preuves pour l'enseignement en L1 », *La Gazette de la Société mathématique de France* **174** (2022).

[KMR24] M. Kerjean, M. Mayero & P. Rousselin – « Maths with Coq in L1, a pedagogical experiment », in *Proc. of the 13th Internat. Workshop on Theorem proving components for Educational software (ThEdu 2024)* (J. Narboux, W. Neuper & P. Quaresma, éds.), EPTCS, vol. 419, 2024, p. 112–123.

[KS11] R. Krebbers & B. Spitters – « Computer certified efficient exact reals in coq », in *Calculemus/MKM*, 2011, p. 90–106.

[Lei66] G. W. Leibniz – *Logical papers*, Oxford University Press, 1966.

[May00] M. Mayero – « The Three Gap Theorem (steinhauss conjecture) », in *Proceedings of TYPES'99*, vol. 1956, Springer-Verlag LNCS, 2000, p. 162–173.

[May01] ———, « Formalisation et automatisation de preuves en analyses réelle et numérique [Formalization and proof automation in real and numerical analysis] », Thèse de Doctorat, Université Paris VI, 2001, In French.

[May12] ———, « Problèmes critiques et preuves formelles », Hdr, Université Paris 13, 2012, In French.

[MBdD+10] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé & S. Torres – *Handbook of floating-point arithmetic*, Birkhäuser, 2010.

[MCDB03] C. A. Muñoz, V. Carreño, G. Dowek & R. W. Butler – « Formal verification of conflict detection algorithms », *Int. J. Softw. Tools Technol. Transf.* **4** (2003), no. 3, p. 371–380.

[MJR05] V. Martin, J. Jaffré & J. E. Roberts – « Modeling fractures and barriers as interfaces for flow in porous media », *SIAM J. Sci. Comput.* **26** (2005), no. 5, p. 1667–1691.

[MM94] V. Ménissier-Morain – « Arithmétique exacte. conception, algorithme et performances d'une implémentation informatique en précision arbitraire. », Thèse de Doctorat, Université Paris VII, Décembre 1994.

[MT20] A. Mahboubi & E. Tassi – *Mathematical components*, Zenodo, Genève, 2020.

[Mul89] J.-M. Muller – *Arithmétique des ordinateurs*, Masson, 1989.

[O'C08] R. O'Connor – « Certified exact transcendental real number computation in coq », in *TPHOLs*, 2008, p. 246–261.

[Pau21] L. C. Paulson – « A Mechanised Proof of gödel's Incompleteness Theorems using Nominal Isabelle », *CoRR* **abs/2104.13792** (2021).

[Pea89] G. Peano – *Arithmetices principia, nova methodo exposita*, 1889.

[PM96] C. Paulin-Mohring – « Définitions inductives en théorie des types [Inductive definitions in type theory] », Habilitation à diriger les recherches, Université Claude Bernard Lyon I, 1996, In French.

[Pro13] T. U. F. Program – « Homotopy Type Theory : Univalent Foundations of Mathematics », (2013).

[Rio91] R. Rioboo – « Quelques aspects du calcul exact avec les nombres réels », Thèse de Doctorat, Université Paris 6, Février 1991.

[RW10] B. Russell & A. N. Whitehead – *Principia mathematica*, Cambridge University Press, 1910.

[Sem20] V. Semeria – « Nombres réels dans Coq [Real numbers in Coq] », in *Actes des 31es Journées Francophones des Langages Applicatifs (JFLA 2020)* (Z. Dargaye & Y. Regis-Gianas, éds.), IRIF, 2020, In French, p. 104–111.

[Tur36] A. Turing – « On computable numbers, with an application to the entscheidungsproblem », *Proceedings of the London Mathematical Society* **42** (1936), p. 230–265.

[Vid90] S. C. Vidyabhusana – *The nyāya sutrās of gotama [archive]*, Motilal Banarsidass Publ., 1990, ISBN 9788120807488.

[Vui90] J. Vuillemin – « Exact Real Computer Arithmetic with Continued Fractions », in *IEEE Transactions on computers 39*, vol. 8, 1990, p. 1087–1105.

[Wer94] B. Werner – « Une théorie des constructions inductives », Thèse, Université Paris 7, 1994.

Appendice A. En lycée

```
(* begin hide *)
(*
  This file is part of the tutoring for 1st year high-school
  students.

Copyright (C) Kerjean, Mayero, Rousselin

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 3 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
COPYING file for more details.
*)
(* end hide *)

Require Import Reals.

Section Variations.
Open Scope R_scope.
Set Printing Parentheses.
(** On travaille maintenant avec les nombres réels et les
   fonctions affines. *)

(** Soient a et b deux réels fixés mais quelconques. *)
Variables (a b : R).

(** Soit f la fonction (affine) suivante : *)
Definition f (x : R) : R := a * x + b.


(** On commence par un cas assez simple : si a = 0, alors f est
   constante. *)

Definition constante (g : R → R) : Prop :=
  ∀ x y : R, g x = g y.
```

```
(** Mais avant de se lancer dans la preuve, il faut dire un
   peu comment on travaille dans R.

   On a un ensemble de règles de réécritures qu'on peut utiliser
   pour remplacer des termes par d'autres dans le but (ou dans
   les hypothèses).

   En voici quelques-unes, suivi d'un exemple et d'un exercice.
*)

(** La commande Check permet d'afficher un théorème de nom donné *)
Check Rmult_0_r.
Check Rmult_0_l.
Check Rplus_0_r.
Check Rplus_0_l.
(* l est pour "left", à gauche et r pour "right", à droite *)

Theorem exemple_rewrite (x : R) :
  (0 * 2 + x) + 3 * 0 = x.
Proof.
  (* D'après Rmult_0_r, 3 * 0 peut être remplacé par 0. *)
  rewrite Rmult_0_r.
  (* D'après Rmult_0_l, 0 * 2 peut être remplacé par 0. *)
  rewrite Rmult_0_l.
  (* D'après Rplus_0_l, 0 + x peut être remplacé par x. *)
  rewrite Rplus_0_l.
  (* D'après Rplus_0_r, x + 0 peut être remplacé par x. *)
  rewrite Rplus_0_r.
  (* Les deux membres de l'égalité sont les mêmes, la preuve est
     finie. *)
  reflexivity.
Qed.

(* À vous *)
Theorem exercice_rewrite (x y : R) :
  ((0 + (y * 0)) + 0 * x) + x = x.
Proof.
  (* Début Solution *)
  rewrite Rplus_0_l, Rmult_0_l, Rmult_0_r, Rplus_0_r, Rplus_0_l.
  reflexivity.
```

```
Qed.
  (* Fin Solution *)

(** On peut aussi utiliser rewrite avec des hypothèses. *)
Theorem exercice_rewrite2 (x : R) :
  a = 0 → a + a = a * a.
Proof.
  (* On suppose (hypothèse (H)) que a = 0. *)
  intros H.
  (* D'après (H), on peut remplacer a par 0 dans le but. *)
  rewrite H.
  (* À vous de terminer la preuve. *)
  (* Début Solution *)
  rewrite Rplus_0_l, Rmult_0_r.
  reflexivity.
Qed.
  (* Fin Solution *)

(* Maintenant les fonctions affines constantes.
   Conseil : unfold constante et unfold f pour remplacer par les
   définitions. *)
Theorem a_0_affine_constante :
  a = 0 → constante f.
Proof.
  (* Début Solution *)
  unfold f, constante.
  intros H x y.
  rewrite H.
  rewrite Rmult_0_l.
  rewrite Rmult_0_l.
  rewrite Rplus_0_l.
  reflexivity.
Qed.
  (* Fin Solution *)

(* La réciproque est vraie : si f est constante, alors a = 0.
   Mais la preuve utilise des tactiques pas encore vues, alors on
   vous la donne avec les explications mathématiques en commentaires.
*)
Theorem affine_constante_a_0 :
  constante f → a = 0.
```

```
Proof.
  unfold constante, f.
  (* On suppose que f est constante, donc que :
     (H) : pour tous x, y dans R, a * x + b = a * y + b *)
  intros H.
  (* Comme c'est vrai pour tous les x et y de R, on peut l'écrire
     en particulier pour x = 1 et y = 0. *)
  specialize (H 1 0).
  (* On a donc
     (H) : a * 1 + b = a * 0 + b
     On peut remplacer a * 1 par a. *)
  rewrite Rmult_1_r in H.
  (* Puis a * 0 par 0. *)
  rewrite Rmult_0_r in H.
  (* Enfin, comme on peut soustraire de chaque côté dans une égalité,
     on a a + b = 0 + b → a = 0, donc pour prouver que a = 0, il
     suffit de prouver a + b = 0 + b. *)
  apply (Rplus_eq_reg_r b).
  (* Ce qui est exactement (H) *)
  exact H.
Qed.

(** On passe au cas a > 0. Alors f est strictement croissante. *)

Definition strict_croissante (f : R → R) :=
  ∀ x y : R, x < y → f x < f y.

(** Pour prouver le théorème suivant, vous aurez besoin de certains
    des résultats suivants : *)
Check Rplus_lt_compat_r.
Check Rplus_lt_compat_l.
Check Rmult_lt_compat_r.
Check Rmult_lt_compat_l.

Theorem a_sup_0_affine_croissante :
  a > 0 → strict_croissante f.
Proof.
  (* Début Solution *)
  unfold strict_croissante, f.
  intros H x y H2.
  apply Rplus_lt_compat_r.
```

```
  apply Rmult_lt_compat_l.
  exact H.
  exact H2.
Qed.
  (* Fin Solution *)

(** Plus difficile, à faire seulement si vous avez bien compris la
    preuve de affine_constante_a_0 :

    Théorèmes que vous pouvez utiliser : *)
Check Rplus_lt_reg_r.
Check Rplus_lt_reg_l.
Check Rlt_0_1.

Theorem affine_croissante_a_sup_0 :
  strict_croissante f → 0 < a.
Proof.
  (* Début Solution *)
  unfold strict_croissante, f.
  intros H.
  specialize (H 0 1).
  apply (Rplus_lt_reg_r b).
  rewrite Rmult_0_r in H.
  rewrite Rmult_1_r in H.
  apply H.
  exact Rlt_0_1.
Qed.
  (* Fin Solution *)

Definition strict_decroissante (f : R → R) :=
  ∀ x y : R, x < y → f x > f y.

(** Pour prouver le théorème suivant, vous aurez besoin de ce
    théorème qui dit qu'en multipliant une inégalité par un nombre
    négatif, l'inégalité est renversée : *)
Check Rmult_lt_gt_compat_neg_l.

Theorem a_inf_0_affine_decroissante :
  a < 0 → strict_decroissante f.
Proof.
  (* Début Solution *)
```

```
  unfold strict_decroissante, f.
  intros H x y H2.
  apply Rplus_lt_compat_r.
  apply Rmult_lt_gt_compat_neg_l.
  exact H.
  exact H2.
Qed.
  (* Fin Solution *)

(** Vraiment si vous en voulez encore, vous pouvez essayer de
    prouver le théorème qui va suivre.
    Indice : x > y est juste une notation pour y < x. *)

(** Exemple pour illustrer l'indice : *)
Theorem indice_sup_inf (x y : R) : x < y → y > x.
Proof.
  intros H.
  exact H.
Qed.

(** Attention ce qui suit est vraiment compliqué. Indice: vous
    pouvez faire un "rewrite _ in _ " **)
Theorem affine_decroissante_a_inf_0 :
  strict_decroissante f → a < 0.
Proof.
  (* Début Solution *)
  unfold strict_decroissante, f.
  intros H.
  specialize (H 0 1).
  rewrite Rmult_0_r in H.
  rewrite Rmult_1_r in H.
  apply (Rplus_lt_reg_r b).
  apply H.
  exact Rlt_0_1.
Qed.
  (* Fin Solution *)
End Variations.
```

Appendice B. En L1

```
(* begin hide *)
(*
  This file is part of a first-year (L1) course in the DL (dual-degree
  program) in mathematics and computer science. 2022.

Copyright (C) Kerjean, Mayero, Rousselin

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 3 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
COPYING file for more details.
*)
(* end hide *)


  (** *Nombres réels : les suites convergentes *)
Require Import Unicode.Utf8 Rbase Rfunctions SeqSeries Lra Lia PeanoNat.
Import Nat.
Open Scope nat_scope.
Open Scope R_scope.

(** ** Première partie : égalité en analyse

   En analyse, prouver que deux choses sont égales s'obtient souvent par
   des moyens détournés qui peuvent surprendre le débutant.
*)

(** Voici le résultat sans doute le plus fréquemment utilisé en
   analyse (mais souvent implicitement) : pour prouver qu'un réel est nul, il
   suffit de montrer que sa valeur absolue (c'est-à -dire sa distance à
0) est
   inférieure à tout réel strictement positif.

   Si vous voulez, vous pouvez (mais ce n'est pas obligé) utiliser :
*)
Check Rabs_pos_lt.
```

```
Lemma small_zero: ∀ x, (∀ eps, eps > 0 → (Rabs x) < eps) → x = 0.
Proof.
  (* Début Solution *)
  intros x H.
  destruct (Req_dec x 0) as [Hx | Hx].
  – assumption.
  – exfalso. apply (Rlt_irrefl (Rabs x)).
    apply H.
    now apply Rabs_pos_lt.
Qed.
  (* Fin Solution *)

(** Deux réels sont égaux lorsque leur distance peut être rendue aussi petite
    qu'on veut. Pensez à [unfold R_dist, Rminus].
*)
Lemma small_dist_equal: ∀ x y,
  (∀ eps, eps > 0 → (R_dist x y) < eps) → x = y.
Proof.
  (* Début Solution *)
  unfold R_dist.
  intros x y H.
  apply Rminus_diag_uniq.
  now apply small_zero.
Qed.
  (* Fin Solution *)

(** ** Deuxième partie : les suites convergentes (1) *)

(** Une suite réelle n'est rien d'autre qu'une fonction de nat vers R.
    La définition de la convergence d'une suite vers une limite l (finie)
    par Weierstrass a fait entrer l'analyse dans son ère moderne.
    Dans cette bibliothèque, cette propriété se note Un_cv.
*)

Print Un_cv.

(** Il s'agit bien de la définition usuelle : une suite Un converge vers une
    limite l si la distance entre Un et l devient aussi
    petite que l'on veut à partir d'un certain rang. *)

(** En guise d'exemple, nous allons donner la preuve de l'unicité d'une telle
```

```
   limite, lorsqu'elle existe.

   Mais avant, nous devons parler de l'ordre sur les entiers naturels car
   ils sont aussi de la partie.
*)

(** Nous ne pouvons pas définir l'ordre <= sur nat dans ce cours pour des
   raisons pédagogiques. Par contre nous pouvons l'utiliser et utiliser les
   théorèmes déjà  prouvés dessus. *)

(* Il y a une grande quantité de théorèmes déjà  prouvés dans la bibliothèque : *)
Search "<="%nat inside PeanoNat. (* Le %nat sert à définir la portée
        du symbole <= car ici, nous sommes dans celles des réels. *)

(* En particulier, <= est bien une relation d'ordre. *)
Check le_trans. (* le pour "lesser than or equal to", transitivité *)
Check le_refl. (* réflexivité *)
Check Nat.le_antisymm. (* antisymétrie *)

(** Pour des inégalités "évidentes" dans nat, il y a la tactique automatique
   très puissante [lia]. Un exemple (très utile) : *)
Lemma le_succ_r (n m : nat) : (n <= (S m) → n <= m ∨ n = (S m))%nat.
Proof.
  intro H. lia.
Qed.

(** Enfin, on a souvent à utiliser les lemmes sur le maximum de deux entiers. *)
Check max_l.
Check max_r.
Check le_max_l.
Check le_max_r.
Check max_lub_l.
Check max_lub_r.
Search max%nat.

(* On s'attaque maintenant à l'unicité de la limite : *)
Theorem UL_sequence (Un : nat → R) (l1 l2 : R) :
  Un_cv Un l1 → Un_cv Un l2 → l1 = l2.
Proof.
  unfold Un_cv.
  intros Hl1 Hl2.
```

```
  (* On va montrer que la distance entre l1 et l2 est aussi petite qu'on veut.*)
  apply small_dist_equal.
  (* Soit eps > 0. *)
  intros eps Heps.
  (* Soit n1 tel que pour tout n >= n1, |Un - l1| < eps / 2. *)
  destruct (Hl1 (eps / 2)) as [n1 Hn1]. lra.
  (* Soit n2 tel que pour tout n >= n2, |Un - l2| < eps / 2. *)
  destruct (Hl2 (eps / 2)) as [n2 Hn2]. lra.
  (* Soit n3 = max(n1, n2). *)
  remember (max n1 n2) as n3 eqn:n3_max.
  (* Il suffit de montrer (par transitivité de l'ordre) que
    |l1 - l2| <= |l1 - Un3| + |Un3 - l2| et
    |l1 - Un3| + |Un3 - l2| < eps *)
  apply (Rle_lt_trans _ (R_dist (Un n3) l1 + R_dist (Un n3) l2) _).
  - (* La première inégalité est l'inégalité triangulaire. *)
    rewrite (R_dist_sym (Un n3)).
    apply R_dist_tri.
  - (* La seconde est la somme des inégalités Hn1 et Hn2 appliquées à
n3. *)
    replace eps with (eps/2 + eps/2) by lra.
    apply Rplus_lt_compat.
    + apply Hn1. lia.
    + apply Hn2. lia.
Qed.

(** Remarques sur l'importante preuve précédentes
    - On essaie de déléguer au maximum les preuves évidentes annexes à
[lra]
      pour les réels et [lia] pour les naturels.
    - La tactique
      [replace terme1 with terme2]
      remplace un terme par un autre dans le but. Coq va nous demander ensuite
      une preuve de l'égalité [terme1 = terme2].
      Ici, on la donne directement après le mot-clé [by] car [lra] sait le
      faire.
      Il y a bien sû r une variante
      [replace terme1 with terme2 in hypothèse]
      qui permet de remplacer un terme par un autre dans une hypothèse.
    - La tactique [remember] permet d'introduire dans le contexte un nouvel
      objet (ou même un théorème, une hypothèse).
      Ici on aurait pu s'en passer et simplement écrire [exists (max n1 n2)]
```

```
     mais ça correspondait mieux à la démonstration mathématique.
   - Le coeur de l'activité de l'analyste est de manipuler des inégalités. Ici
     on a utilisé (et vous utiliserez encore beaucoup) :
     Rplus_lt_compat
     Rle_lt_trans
     Rdist_tri
*)

(** À vous. En vous inspirant de la preuve de UL_sequence, prouvez
   que la somme de deux suites convergentes converge vers la somme des limites.

   Il est indispensable que la preuve mathématique soit déjà  claire pour vous.
 *)

Theorem CV_plus (An Bn : nat → R) (l1 l2 : R) :
  Un_cv An l1 → Un_cv Bn l2 → Un_cv (λ  n ⇒ An n + Bn n) (l1 + l2).
Proof.
  (* Début Solution *)
  intros HA HB eps Heps.
  destruct (HA (eps / 2)) as [n1 Hn1]. lra.
  destruct (HB (eps / 2)) as [n2 Hn2]. lra.
  remember (max n1 n2) as n3 eqn:def_n3.
  exists n3.
  intros n Hn.
  replace eps with (eps/2 + eps/2) by lra.
  apply (Rle_lt_trans _ ((R_dist (An n) l1) + (R_dist (Bn n) l2))).
    apply R_dist_plus.
  apply Rplus_lt_compat.
  + apply Hn1. lia.
  + apply Hn2. lia.
Qed.
  (* Fin Solution *)

(** Maintenant, on prouve qu'en multipliant une suite convergente vers l par une
   constante K, on obtient une suite qui converge vers K * l.
   Attention, il y a deux cas K = 0 ou non.
   Faire un [destruct Req_dec K 0] pour séparer ces deux cas.
   Quelques théorèmes, qui peuvent être utiles :
*)

Check Req_dec.
```

```
Check R_dist_eq.
Check Rabs_pos_lt.
Check Rdiv_lt_0_compat.
Check Rmult_lt_compat_l.
Check R_dist_mult_l.

(** Assurez-vous, avant de vous lancer dans la preuve formelle de savoir le
    faire en maths. *)
Lemma CV_mult_const_l (Un : nat → R) (K : R) (l : R) :
  Un_cv Un l → Un_cv (λ n ⇒ K * Un n) (K * l).
Proof.
  (* Début Solution *)
  intros H.
  destruct (Req_dec K 0) as [H0 | Hn0].
  - exists 0%nat. intros n _.
    now rewrite H0, 2Rmult_0_l, R_dist_eq.
  - intros eps Heps.
    apply Rabs_no_R0 in Hn0 as HRabs_n0.
    apply Rabs_pos_lt in Hn0.
    replace eps with ((Rabs K) * (eps / (Rabs K))) by now field.
    destruct (H (eps / (Rabs K))) as [N HN].
      apply Rdiv_lt_0_compat; assumption.
    exists N; intros n Hn.
    rewrite R_dist_mult_l.
    apply Rmult_lt_compat_l; try assumption.
    now apply HN.
Qed.
  (* Fin Solution *)

(** Résultat intermédiaire facile pour prouver la convergence de Un * K en
    utilisant le théorème précédent. *)
Theorem CV_eq_compat_l (Un Vn : nat → R) (l : R) :
  (∀ n, Vn n = Un n) → Un_cv Un l → Un_cv Vn l.
Proof.
  (* Début Solution *)
  intros H un_to_l eps Heps.
  destruct (un_to_l eps Heps) as [N HN].
  now exists N; intros n Hn; rewrite H; apply HN.
Qed.
  (* Fin Solution *)
```

```
Lemma CV_mult_const_r (Un : nat → R) (K : R) (l : R) :
  Un_cv Un l → Un_cv (λ n ⇒ Un n * K) (l * K).
Proof.
  (* Début Solution *)
  intros H.
  rewrite Rmult_comm.
  apply (CV_eq_compat_l (λ n ⇒ K * Un n)).
    now intros n; rewrite Rmult_comm.
  now apply CV_mult_const_l.
Qed.
  (* Fin Solution *)

(** ** Partie 3 : Un exemple de suite qui diverge vers l'infini *)

(** Il n'y a pas que les suites convergentes dans la vie. Il y a aussi
    - des suites qui divergent vers + l'infini;
    - des suites qui divergent vers - l'infini;
    - des suites qui divergent tout court (n'admettent ni limite finie ni limite
      infinie).

    On va ici prouver la suite définie par Un = n pour tout n diverge vers
    l'infini.
    D'abord la définition. Dans cette bibliothèque, cela s'écrit [cv_infty].
*)
Print cv_infty.

(** En fait, cette définition n'est pas très pratique. On va plutôt utiliser *)
Definition cv_infty' (Un : nat → R) :=
  ∀ A : R, A > 0 →
    ∃ N : nat, ∀ n : nat, (n >= N)%nat → Un n > A.

(** Avec la preuve (à  lire) que les deux propositions sont équivalentes *)

Theorem cv_infty_cv_infty' (Un : nat → R) :
  cv_infty Un ↔ cv_infty' Un.
Proof.
  (* On va prouver les deux implications. *)
  split; intros H.
  - (* Celle de gauche à droite est évidente. *)
    intros A _; apply H.
  - intros A.
```

```
    (* Pour la seconde, on raisonne par cas selon le signe de A *)
    destruct (total_order_T A 0) as [H' | H'].
    + (* On suppose A <= 0.
         L'hypothèse (H) fournit N tel que (Un n) > 1 dès que n >= N. *)
      destruct (H 1) as [N HN].
        lra.
      (* On peut choisir ce N. *)
      exists N; intros n Hn.
      (* Soit n >= N. Alors Un > 1 > A car A <= 0. *)
      apply (Rgt_trans _ 1).
      * apply HN, Hn.
      * destruct H'; lra.
    + (* Si A > 0, le résultat est évident. *)
      now apply H.
Qed.

(** En mathématiques, on fait souvent (et pour de bonnes raisons) l'abus que
    l'ensemble des naturels est inclus dans l'ensemble des réels. En coq, ce
    n'est pas possible : nat et R sont deux types différents.
    La fonction INR (pour injection de N dans R) permet de passer du naturel n
    au réel 1 + 1 + ... + 1 (n fois). *)
Print INR.

(** Nous allons prouver que la suite INR diverge vers l'infini. Pour ceci, nous
    aurons besoin de la propriété suivante des réels : ils forment un corps
    archimédien : *)
Axiom archimed' :
  ∀ eps A : R, eps > 0 → A > 0 → ∃ n : nat, (INR n) * eps > A.
(** Cet axiome dit en substance que
    - pour un eps > 0 aussi petit qu'on veut;
    - pour un A > 0 aussi grand qu'on veut;
    On peut toujours trouver un entier naturel n tel que n * eps est plus grand
    que A.

    Remarque : l'axiome [archimed] de cette bibliothèque est différent, car plus
    puissant, mais plus difficile, dans un premier temps, à
utiliser.
*)

(** À vous maintenant, vous aurez à
    [destruct (archimed' preuve1 preuve2 val_eps val_A)]
```

```
    avec :
    - val_eps : ce que vous avez choisi comme eps (n'allez pas chercher trop
    loin...)
    - val_A : ce que vous avez choisi comme A
    - preuve1 : une preuve que votre eps > 0
    - preuve2 : une preuve que votre A > 0

    Enfin, vous aurez sans doute besoin de résultats sur INR.
*)

Search "INR".
Check le_INR.
Theorem n_to_infty :
  let Un n := INR n in cv_infty Un.
Proof.
  (* Début Solution *)
  apply cv_infty_cv_infty'; intros A HA.
  destruct (archimed' 1 A) as [N HN].
    lra.
    assumption.
  exists N.
  intros n Hn.
  apply (Rlt_le_trans _ (INR N)).
  + lra.
  + now apply le_INR.
Qed.
  (* Fin Solution *)

(** Une suite qui tend vers l'infini est toujours positive à
partir d'un certain
    rang. *)
Theorem cv_infty_pos (Un : nat → R) :
  cv_infty Un → ∃ N, ∀ n, (n >= N)%nat → Un n > 0.
Proof.
  (* Début Solution *)
  intros H.
  destruct (H 0) as [N HN].
  now exists N.
Qed.
  (* Fin Solution *)
(** Deux théorèmes pratiques qui ne sont pas dans la bibliothèque : *)
```

```
Check Rinv_lt_contravar.
Theorem Rinv_lt_contravar' (x y : R) :
  x > 0 → y > 0 → x < y → / y < / x.
Proof.
  (* Début Solution *)
  intros xpos ypos. apply Rinv_lt_contravar.
  now apply Rmult_gt_0_compat.
Qed.
  (* Fin Solution *)

Theorem Rinv_left_lt (x y : R) :
  y > 0 → x > / y → / x < y.
Proof.
  (* Début Solution *)
  intros H H'.
  assert (yinv_pos : / y > 0) by now apply Rinv_0_lt_compat.
  rewrite ← Rinv_inv.
  apply Rinv_lt_contravar'; try easy.
  now apply (Rgt_trans _ (/ y)).
Qed.
  (* Fin Solution *)

(** Maintenant on peut prouver que l'inverse d'une suite qui diverge vers
    l'infini converge vers 0. *)
Theorem cv_infty_cv_0 (Un : nat → R) :
  cv_infty Un → Un_cv (λ n : nat, / Un n) 0.
Proof.
  (* Début Solution *)
  intros H eps Heps.
  destruct (H (/ eps)) as [N HN].
  destruct (cv_infty_pos _ H) as [N' HN'].
  exists (max N N').
  intros n Hn; unfold R_dist.
  rewrite Rminus_0_r.
  rewrite Rabs_right; cycle 1.
    left. apply Rinv_0_lt_compat. now apply HN', (max_lub_r N).
  apply Rinv_left_lt; try easy.
  apply HN.
  now apply (max_lub_l N N' n).
Qed.
  (* Fin Solution *)
```

```
(** En particulier, la suite (λ  n ⇒ 1 / (INR n)) tend vers 0. *)

Theorem inv_n_to_0 : Un_cv (λ  n ⇒ / (INR n)) 0.
Proof.
  (* Début Solution *)
  apply cv_infty_cv_0, n_to_infty.
Qed.
  (* Fin Solution *)

(** ** Partie 4 : Majorant, minorant, borne supérieure et borne inférieure *)

(** Rappel : en coq, une partie de R est une fonction de R vers Prop.
    Un majorant (en anglais upper bound) est une valeur qui est supérieure ou
    égale à tous les éléments de l'ensemble. *)
Print is_upper_bound. (* en français : est majorant de *)
(** On rappelle qu'il faut ici comprendre (E x) comme "x appartient à
E", donc
    [is_upper_bound E m] dit que pour tout x appartenant à
E, x <= m. *)

(** Un minorant d'une partie est défini de la même manière. *)
Definition is_lower_bound (E : R → Prop) (l : R) :=
  ∀ x : R, E x → l <= x.

(** On va montrer que l'ensembles des inverses des entiers naturels est minoré
    par 0. *)
Definition ens_inverses (x : R) := ∃ n : nat, x = / ((INR n) + 1).

Check pos_INR.
Theorem inverses_min : is_lower_bound ens_inverses 0.
Proof.
  (* Début Solution *)
  unfold is_lower_bound, ens_inverses.
  intros x [n H]. rewrite H.
  unfold Rle. left.
  apply Rinv_0_lt_compat.
  apply Rplus_le_lt_0_compat.
  - apply pos_INR.
  - apply Rlt_0_1.
Qed.
```

```
  (* Fin Solution *)

(** La borne supérieure d'un ensemble, lorsqu'elle existe est le plus petit
    majorant de cet ensemble (en anglais least upper bound), ici, [is_lub]. *)
Print is_lub.

(** Même chose pour la borne inférieure (en anglais greatest lower bound). *)
Definition is_glb (E : R → Prop) (l : R) :=
  is_lower_bound E l ∧ (∀ b : R, is_lower_bound E b → b <= l).

(** À vous, maintenant, prouvez que la borne inférieure des inverses des entiers
    naturels est 0. *)
Theorem inverses_glb : is_glb ens_inverses 0.
Proof.
  (* Début Solution *)
  unfold is_glb, ens_inverses.
  split.
  - apply inverses_min.
  - unfold is_lower_bound; intros b H.
    destruct (Rle_lt_dec b 0).
    + assumption.
    + exfalso.
      destruct (archimed' 1 (/ b)) as [N HN].
        apply Rlt_0_1. now apply Rinv_0_lt_compat.
      apply (Rle_not_lt (/ (INR N + 1)) b).
      * apply H. now exists N.
      * apply Rinv_left_lt; lra.
Qed.
  (* Fin Solution *)

(** ** Partie 5 : Compléments sur les suites convergentes. *)

(** On va d'abord vers un résultat très important : les suites convergences
    sont bornées. Pour ce faire on a besoin de l'opération suivante : *)
Fixpoint running_max (Un : nat → R) (n : nat) :=
  match n with
    0%nat ⇒ (Un 0%nat)
  | S n ⇒ Rmax (running_max Un n) (Un (S n))
  end.
(** En français, running_max Un n est le maximum des n + 1 premiers termes de la
    suite Un. *)
```

```
Lemma Un_le_running_max (Un : nat → R) (n : nat) :
  ∀ i, (i <= n)%nat → Un i <= (running_max Un n).
Proof.
(* N'oubliez pas que sur les entiers nat, l'induction existe *)
  (* Début Solution *)
  induction n as [|n' IHn'].
  – intros i Hi.
    simpl. apply Req_le_sym. f_equal. lia.
  – intros i Hi.
    simpl.
    apply le_succ_r in Hi as [ilen' | i_eq_Sn'].
    + apply (Rle_trans _ (running_max Un n')).
      * now apply IHn'.
      * now apply Rmax_l.
    + rewrite i_eq_Sn'; apply Rmax_r.
Qed.
  (* Fin Solution *)

(** Voici une définition possible de "suite bornée" : *)
Definition bounded (Un : nat → R) :=
  ∃ m, ∀ n, Rabs (Un n) <= m.

(** On peut prendre le majorant aussi grand qu'on veut *)
Theorem bounded_choose_ub (A : R) (Un : nat → R) :
  bounded Un → (∃ m, m > A ∧ ∀ n, Rabs (Un n) <= m).
Proof.
  (* Début Solution *)
  intros [m H].
  destruct (Rlt_or_le A m) as [Altm | Alem].
  – now exists m.
  – exists (A + 1).
    assert (ineq: A < A + 1) by now apply Rlt_plus_1.
    split; try easy.
    intros n.
    left. apply (Rle_lt_trans _ m); try easy.
    now apply (Rle_lt_trans _ A).
Qed.
  (* Fin Solution *)

(** Le suites convergentes sont bornées. *)
```

```
Theorem CV_impl_bounded (Un : nat → R) (l : R) :
  Un_cv Un l → bounded Un.
Proof.
  (* Début Solution *)
  intros H; unfold bounded.
  destruct (H 1) as [n0 Hn0]. lra.
  exists (Rmax (Rabs l + 1) (running_max (λ  n ⇒ Rabs (Un n)) n0)).
  intros n.
  destruct (le_gt_cases n n0) as [H1 | H2].
  - apply (Rle_trans _ (running_max (λ  n ⇒ Rabs (Un n)) n0)).
    apply (Un_le_running_max (λ  n ⇒ Rabs (Un n)) n0 n); try easy.
    apply Rmax_r.
  - apply Rle_trans with (r2 := Rabs l + 1).
    + apply (Rle_trans _ (Rabs (Un n − l) + Rabs l)).
      * replace (Un n) with ((Un n − l) + l) at 1 by lra.
        apply Rabs_triang.
      * unfold R_dist in Hn0.
        rewrite Rplus_comm.
        apply Rplus_le_compat_l. left. apply Hn0. lia.
    + apply Rmax_l.
Qed.
  (* Fin Solution *)

Lemma CV_l_CV_0 (Un : nat → R) (l : R) :
  Un_cv Un l ↔ Un_cv (λ  n ⇒ (Un n − l)) 0.
Proof.
  (* Début Solution *)
  split; intros H eps Heps; destruct (H eps Heps) as [N HN];
    unfold R_dist in * |- *; exists N; intros n Hn.
  - now rewrite Rminus_0_r; apply HN.
  - now rewrite ← (Rminus_0_r (Un n −l)) ; apply HN.
Qed.
  (* Fin Solution *)

(** On voudrait montrer que le produit de deux suites convergentes est une suite
    convergente.

    On commence par le cas où  l'une des suites converge vers 0 et l'autre
    est bornée. *)

Theorem CV_mult_0 (An Bn : nat → R) :
```

```
  let Cn (n:nat) := (An n * Bn n) in
  bounded An → Un_cv Bn 0 → Un_cv Cn 0.
Proof.
  (* Début Solution *)
  unfold Un_cv, R_dist.
  intros HA. apply (bounded_choose_ub 0) in HA as [m [mpos boundedA]].
  intros HB eps Heps.
  destruct (HB (eps / m)) as [nB HnB].
  apply Rdiv_lt_0_compat; try easy.
  exists nB. intros n Hn.
  rewrite Rminus_0_r, Rabs_mult.
  apply Rle_lt_trans with (r2 := m * Rabs (Bn n)).
  - apply Rmult_le_compat_r.
      apply Rabs_pos.
    apply boundedA.
  - replace eps with (m * (eps / m)).
    + apply Rmult_lt_compat_l; try easy.
      rewrite ← (Rminus_0_r (Bn n)). (* moche *)
      now apply HnB.
    + unfold Rdiv; rewrite Rmult_comm, Rmult_assoc, Rinv_l, Rmult_1_r.
      reflexivity. now apply Rgt_not_eq.
Qed.
  (* Fin Solution *)

Theorem CV_mult (An Bn : nat → R) (a b : R) :
  let Cn (n : nat) := (An n * Bn n) in
  Un_cv An a → Un_cv Bn b → Un_cv Cn (a * b).
Proof.
  (* Début Solution *)
  intros Cn HA HB.
  apply (CV_eq_compat_l (λ n ⇒ An n * (Bn n - b) + (An n) * b)).
    intros n; unfold Cn; ring.
  rewrite ← (Rplus_0_l (a * b)); apply CV_plus.
  - apply CV_mult_0.
    + now apply (CV_impl_bounded An a).
    + now apply (CV_l_CV_0 Bn b).
  - now apply CV_mult_const_r.
Qed.
(* Fin Solution *)
```

Mars 2026

Micaela Mayero, Laboratoire d'Informatique de Paris Nord (LIPN - UMR 7030), Université Sorbonne Paris Nord (USPN)
E-mail : mayero@lipn.univ-paris13.fr
Url : https://www-lipn.univ-paris13.fr/~mayero/