Travail d'études et de recherche

Assistant de preuves et formalisation

Enki SOUILLOT

Encadré par Vincent BEFFARA

Université Grenoble Alpes — M1 Mathématiques Générales — Année Universitaire 2021-2022

1 Introduction

Ce rapport a pour but de donner les bases de l'utilisation du langage de formalisation mathématique L∃∀N , à l'aide d'exemples plus ou moins simples de niveau Master.

Mais tout d'abord, qu'est-ce que L∃∀N ? L∃∀N (ou plutôt le "L∃∀N Theorem Prover" est un langage de formalisation des théorèmes et preuves mathématiques développé par *Leonardo de Moura* au sein de *Microsoft Research*.

L'objectif de L∃∀N est la vérification des preuves mathématiques, notamment via l'application pure des règles de logique élémentaires.

Mais pourquoi vérifier informatiquement nos preuves, même lorsque celles-ci nous semblent justes de part en part ? Le raisonnement mathématique repose sur la rigueur et la logique, il ne laisse pas de place à l'approximatif. L'intuition du mathématicien lui donne les idées d'une preuve, mais rendre cette preuve rigoureuse est parfois un travail de longue haleine. L∃∀N répond à ce problème puisque son objectif est de répondre à une simple question à chaque étape : "Ai-je le droite de faire ceci ?".

L'application des règles de logique entre plusieurs énoncés, basés sur un système cohérent d'axiomes, permet à L∃∀N de vérifier la véracité des preuves mais également de les rendre plus cohérente : l'ordre des arguments, la nécessité ou non de telle hypothèse, etc...

L∃∀N est donc un outil à la résolution des problèmes mathématiques, mais il ne remplacera jamais l'esprit acéré du mathématicien devant son tableau noir.

Depuis quelques années, la communauté L∃∀N , constituée de chercheurs en mathématiques et de passionnés, œuvre dans le but de constitué une librairie suffisante pour que l'utilisation de L∃∀N devienne accessible à tous niveaux.

La librairie `mathlib` rassemble un grand nombre de définitions, lemmes et théorèmes avec leurs preuves, sur des domaines très variés allant de la topologie à l'algèbre des modules passant par la théorie de la mesure. L'objectif est de formaliser les résultats "undergraduate", c'est-à-dire jusqu'au niveau Master principalement.

Toutes ces preuves existantes peuvent être utilisées directement dans les preuves que nous faisons, afin de ne pas avoir à re-démontrer certains résultats préliminaires.

Ce rapport présentera, dans un premier temps, l'utilisation basique du logiciel en prenant exemple sur des résultats simples, puis quelques résultats d'analyse complexe du premier semestre de Master, dont nous détailleront les preuves.

2 LEAN et ses tactiques

Le langage L∃∀N a un grand intérêt lors de l'écriture des preuves. En effet, à chaque étape, L∃∀N affiche le *contexte local* et l'*objectif*.

Le *contexte locale* réuni toute les données existantes à l'instant T, que ce soit les variables introduites dans l'énoncé du théorème ou dans la preuve, mais aussi les hypothèses sur ces variables. L'*objectif* est, au début, l'énoncé à prouver. À chaque ligne, celui-ci s'actualise afin de donner exactement les éléments qu'il reste à prouver.

Cette configuration permet de réflechir à la preuve directement sur le logiciel, mais aussi de comprendre plus facilement les erreurs que nous aurions pu faire.

Pour résoudre un objectif, c'est-à-dire faire une preuve, il nous faudra utiliser des tactiques et des lemmes. Les tactiques sont des outils permettant le raisonnement mathématique pur, par exemple remplacer dans une équation une variable par une autre dont on a prouvé qu'elles sont égales. Les lemmes sont des énoncés que l'on a prouvé précédemment ou qui se trouvent dans la librairie. Nous y feront appel, en les adaptant au contexte local.

La combinaison de ces deux éléments permet de réaliser les preuves, de la plus simple à la plus complexe. Parfois, pour résoudre un objectif complexe, nous serons amenés à créer des lemmes intermédiaires, ou encore à avoir plusieurs objectifs dans la même preuve.

Un autre élément important se trouve dans la syntaxe des preuves. On trouvera toujours une virgule après une commande, elle est indispensable pour que L∃∀N interprète la tactique.

Par exemple, voici un énoncé simple que nous allons étudier dans un premier temps.

Code lean : Inégalités de réels

```
1  example (a b c  : ℝ) (hc : 0 ≤ c) (hab : a ≤ b) : a*c ≤ b*c :=
2  begin
3    rw ← sub_nonneg,
4    have h_fact : b*c - a*c = (b - a)*c,
5    { ring },
6    rw h_fact,
7    apply mul_nonneg,
8    { rw sub_nonneg,
9      exact hab },
10   { exact hc },
11 end
```

À la première lecture, tout ceci doit vous sembler quelque peu incompréhensible. Prenons les choses une par une.

Tout d'abord, la ligne 1 : c'est l'énoncé. Le mot clef `example` annonce à L∃∀N un énoncé que l'on ne souhaite pas garder en mémoire pour pouvoir le réutiliser, à l'instar de `lemma` que nous verrons plus tard.

La syntaxe pour ce mot clef est la suivante : `example (variables) (hypothèses) : résultat :=`.

Ici, on déclare 3 variables `a, b, c` qui sont des réels, ou plutôt qui sont "de type" réel. On donne ensuite deux hypothèses :

— Une appelée `hc` qui dit que 0 est plus petit que `c` ;

— l'autre, `hab`, donne `a` plus petit ou égal à `b`.

Enfin, on annonce le résultat que l'on souhaite prouver : $ac \leq bc$. Ce résultat est, a priori, très simple. Il va cependant nécessité quelques tactiques pour le prouver.

La preuve du résultat se trouve après les symboles := et entre les mots `begin` et `end`. Si nous plaçons notre curseur juste après le `begin`, voici ce que L∃∀N affiche :

```
1 goal
a b c : ℝ
hc: 0 ≤ c
hab: a ≤ b
⊢ a * c ≤ b * c
```

On observe sur les lignes 2,3 et 4 le *contexte local* avec les variables et les hypothèses données dans l'énoncé, puis en ligne 5, après le symbole ⊢, l'*objectif* en cours.

Commençons maintenant la preuve. Pour cela nous aurons besoin de quelques lemmes existant déjà dans la librairie mathlib :

```
sub_nonneg : 0 ≤ b - a ↔ a ≤ b,
mul_nonneg : (0 ≤ a → 0 ≤ b) → 0 ≤ a * b
```

La première étape va être de transformer l'objectif $ac \leq bc$ en $0 \leq bc - ac$. Nous allons donc utiliser le lemme `sub_nonneg` et la tactique `rewrite`

Tactique	**Rewrite – rw** La tactique `rw` réécrit l'objectif en cours à l'aide d'une égalité. On peut la traduire par "On remplace".	Si l'objectif est `a = c` et une hypothèse `h : a = b`, alors écrire `rw h` donne l'objectif `b = c`.

La tactique `rw` fonctionne également avec les équivalences : si `h` : P $\leftrightarrow$ Q est une hypothèse, alors `rw h` transforme P $\rightarrow$ R en Q $\rightarrow$ R.

Une variante est nécessaire dans notre cas, puisque l'on veut réécrire l'implication inverse. Pour cela, il suffit d'ajouter $\leftarrow$ après le `rw`. On écrit donc `rw` $\leftarrow$ `sub_nonneg`, et voici ce que L∃∀N nous dit :

```
1 a b c : ℝ
2 hc : 0 ≤ c
3 hab : a ≤ b
4 ⊢ 0 ≤ b * c - a * c
```

Passons à la ligne 4. Nous voulons maintenant factoriser le membre de droite de notre égalité. Pour cela, nous allons avoir recours à la tactique `have` :

Tactique	**Have** Elle a pour effet de créer une nouvelle hypothèse, sous condition d'en donner une preuve dans le contexte local. La syntaxe est la suivante : `have 'nom' : 'hypothèse'.`	Écrire `have h : a = b` donne deux objectifs : – Prouver que `a = b` avec le contexte local ; – l'objectif initial dont le contexte local est enrichi avec l'hypothèse `h`.

Ici, on va donc créer l'hypothèse nommée `h_fact` qui donne la factorisation : c'est la ligne 4.

Il nous faut ensuite donner la preuve de cette factorisation. Bien heureusement, nous n'avons pas à reprendre toutes les mathématiques de base, il nous suffit d'utiliser le raccourcis `ring`. Celui-ci va tout simplement résoudre les objectifs avec les règles de calculs propres aux anneaux (comme le nom l'indique), comme dans notre exemple.

On peut finalement réécrire cette nouvelle hypothèse dans l'objectif par un simple `rw h_fact,`, et on obtient l'objectif suivant :

```
1 ⊢ 0 ≤ (b - a) * c
```

Nous avons maintenant quelque chose de la même forme que la conclusion du lemme `mul_nonneg` et nous voudrions pouvoir revenir à ses arguments, "prendre la flèche dans l'autre sens". Pour cela, nous allons introduire la tactique `apply`.

Tactique	**Apply** `Apply` cherche une ressemblance entre l'objectif et la conclusion du lemme. Il change ensuite l'objectif en cours en demandant les arguments du lemme en question.	Si on a `lemme_1 : A → B` et un objectif de la forme B, écrire `apply lemme_1` change l'objectif par A.

Ainsi, appliquons le lemme `mul_nonneg` à l'objectif, nous obtenons les objectifs suivants :

```
1 2 goals
2
3 abc: ℝ
4 hc: 0 ≤ c
```

```
hab: a ≤ b
h_fact: b * c - a * c = (b - a) * c
⊢ 0 ≤ b - a

abc: ℝ
hc: 0 ≤ c
hab: a ≤ b
h_fact: b * c - a * c = (b - a) * c
⊢ 0 ≤ c
```

Concernant le premier objectif, celui-ci ressemble fortement à l'hypothèse `hab`. Pour avoir une ressemblance exacte, il nous faut appliquer notre lemme `sub_nonneg` avec la tactique `rw` : en effet, ici on veut remplacer notre objectif $0 \leq b - a$ par $a \leq b$. On peut ensuite conclure à l'aide de la tactique `exact`

Tactique	**Exact** Comme son nom l'indique, cette tactique agit uniquement quand l'objectif est exactement le même qu'une hypothèse ou qu'un lemme connu. Elle permet de conclure la démonstration.	Si on a une hypothèse `h : a = b` et que l'objectif est `a = b`, alors `exact h` permet de conclure.

Ici, on conclut donc le premier objectif.
Pour le second, c'est exactement l'hypothèse `hc`, donc on conclut avec la tactique `exact`.

Nous avons donc réussi à prouver que multiplier par un nombre positif ne change pas le sens d'une inégalité. C'est évidemment un résultat très simple que nous n'aurons plus jamais besoin de démontrer, puisqu'il est dans la librairie sous le nom de `mul_mono_nonneg`.
On peut d'ailleurs écrire comme seule démonstration :

```
exact mul_mono_nonneg hc hab,
```

Il s'agit du lemme correspondant que l'on applique aux hypothèses du contexte local.

Il reste nombre d'autres tactiques que l'on peut utiliser, nous ne les énuméreront pas toutes ici. Voici simplement les quelques autres tactiques dont nous aurons besoin par la suite.

Pour la première, L∃∀N connait déjà nombre de choses en mathématiques, et pour faire appel à ces connaissances, on peut demander à L∃∀N de simplifier l'objectif :

Tactique	**Simplify — simp** L∃∀N va simplifier les énoncés de l'objectif à l'aide des lemmes qu'il connait. On peut également lui demander de simplifier certaines définitions propres à notre code ou encore de s'aider des hypothèses.	Il existe plusieurs syntaxe pour `simp` : — `simp` utilise certains lemmes de la librairie, — `simp[blabla]` utilise certains lemmes et les hypothèses blabla qui lui sont données.

C'est une tactique bien utile pour simplifier les objectifs sans avoir à rechercher tous les lemmes correspondants dans la librairie.

Parlons maintenant d'une tactique plus complexe mais pourtant bien utile pour les preuves longues : `refine`. Nous l'utiliseront à de nombreuses reprises dans la suite, prenons donc le temps de l'expliquer ici.

Tactique	**Refine** On utilise cette tactique pour séparer un objectif complexe en autant d'objectifs qu'il y a d'arguments. Il suffit d'indiquer autant de _ que d'arguments et chaque argument devient un objectif.	Par exemple, si on veut une application linéaire de $\mathbb{R}$ dans $\mathbb{R}$, c'est-à-dire un objectif $\mathbb{R} \to_l \mathbb{R}$, écrire `refine ⟨_,_,_⟩` permet d'avoir 3 objectifs (3 _) qui correspond à donner l'application, prouver son additivité et prouver son homogénéité (cf plus loin).

Nous verrons l'utilité de cette tactique plus loin. Enfin, terminons par une tactique qui permet de chercher s'il existe un résultat semblable à l'objectif dans la librairie, qui permettrait de conclure : `library_search`.

3 L'analyse complexe par LEAN

Afin de rendre les connaissances acquises via les tutoriels d'utilisation de L∃∀N , nous allons coder les résultats et les preuves du cours d'analyse complexe de M1. En raison de la longueur du travail nécessaire pour une seule preuve, nous n'iront pas jusqu'au bout du cours.

3.A — Cadre

Tout d'abord, posons le cadre. Heureusement, la librairie mathlib possède déjà une version de $\mathbb{C}$. Les nombres complexes sont présentés comme des paires de réels, la partie réelle et la partie imaginaire. Autrement dit, le code $\boxed{z : \mathbb{C}}$ signifie $\boxed{x\,y : \mathbb{R}, z = (x, y)}$.

Cette définition va nous être bien utile par la suite. Pour y faire appel dans notre fichier L∃∀N , il nous suffit d'écrire

```
1  import analysis.complex.basic
```

Une des conséquences évidentes est donc que $\mathbb{C}$ est en bijection avec $\mathbb{R} \times \mathbb{R}$. Ceci est donné par la fonction `complex.equiv_real_prod`. Par exemple, si $z : \mathbb{C}$, alors `complex.equiv_real_prod z` est un élément de $\mathbb{R} \times \mathbb{R}$. En revanche, si $x : \mathbb{R} \times \mathbb{R}$, alors `complex.equiv_real_prod.symm x` est un nombre complexe. En effet, `complex.equiv_real_prod` est une fonction de $\mathbb{C}$ dans $\mathbb{R} \times \mathbb{R}$ qui est bijective, dont on peut donc prendre la réciproque via la commande `.symm`

Une autre propriété est que cette fonction qui lie $\mathbb{C}$ et $\mathbb{R} \times \mathbb{R}$ est linéaire continue. Pour cela on utilise la fonction `complex.equiv_real_prod`$_l$ dont la linéarité et la continuité sont prouvées.

Pour continuer, nous aurons besoin des fonctions holomorphes, qui sont en réalité des fonctions dérivables de $\mathbb{C}$. Il se trouve que la dérivation à été définie, tout comme la différentiabilité (qui sont fortement liée l'une à l'autre...) dans la librairie. Ces notions, comme toutes les autres dans la librairie, sont définis de la manière la plus générale possible, ce qui signifie que nous pouvons utiliser la dérivation dans $\mathbb{C}$ par la même commande que celle dans $\mathbb{R}$, ce qui est très pratique.

Afin d'utiliser ces définitions, importons le bon fichier :

```
1  import analysis.calculus.deriv
```

Pour finir, ajoutons le code

```
1  noncomputable theory
```

afin d'éviter la plupart des problèmes qui pourrait survenir.

3.B — Cauchy-Riemann, étape 1 — énoncé

Commençons par ce premier résultat, les équations de Cauchy-Riemann. En revanche, nous n'allons pas pouvoir le prouver sous la forme la plus connue, c'est-à-dire par les dérivées partielles, puisque

celles-ci n'existent pas directement dans la librairie. Nous allons passer par une version quelque peu différente. Voici le premier résultat à prouver.

Holomorphe $\Longrightarrow$ différentiable sur $\mathbb{R}^2$

Soit $f : \mathbb{C} \to \mathbb{C}$ une fonction holomorphe ($\mathbb{C}$–dérivable) en $z \in \mathbb{C}$. Alors f, en tant qu'application de $\mathbb{R}^2$ dans $\mathbb{R}^2$ est différentiable en $z \in \mathbb{R}^2$ de différentielle la multiplication par $f'(z)$.

La première étape est de réussir à écrire un énoncé accepté par L∃∀N . On a ici 3 variables : la fonction f, le point z et le point $f'(z)$ que l'on notera f'. On travaille ici en un unique point, il n'y a pas de nécessité de considérer la fonction dérivée $f' : \mathbb{C} \to \mathbb{C}$.

On a ensuite une hypothèse de départ : f est $\mathbb{C}$–dérivable en z de dérivée f' (encore une fois, f' est un point qui est par définition la dérivée de f en z). Le théorème qui dit que f est $\mathbb{C}$–dérivable en un point porte le nom de `has_deriv_at`. On peut donc commencer par écrire :

```
lemma cauchy_riemann_step_1 {f : ℂ → ℂ} {z : ℂ} (f' : ℂ) (hf : has_deriv_at
f f' z) :
```

Parlons maintenant de l'énoncé à prouver. On veut voir f comme une fonction de $\mathbb{R}^2$. Pour cela, nous allons écrire `realify f`, puis nous définirons la fonction `realify` plus loin. Le théorème qui dit qu'une fonction est différentiable est `has_fderiv_at`. On peut appeler `multiply` la fonction de multiplication que nous définirons plus loin. On a alors :

```
lemma cauchy_riemann_step_1 {f : ℂ → ℂ} {z : ℂ} (f' : ℂ) (hf : has_deriv_at
f f' z) :

has_fderiv_at (realify f) (multiply f') (complex.equiv_real_prod z) :=
```

Sous réserve de définir `realify` et `multiply`, cet énoncé tient la route.

On notera d'ailleurs la syntaxe de la commande `lemma` : en premier le nom que l'on donne afin d'y faire référence plus tard. Ensuite, entre parenthèse ou accolades, les variables et les hypothèses, puis `:`, l'énoncé et on termine par `:=`.

3.C – Les lemmes intermédiaires

Pour commencer et afin de simplifier l'utilisation, re-définissons les fonctions de $\mathbb{C}$ dans $\mathbb{R}^2$:

```
def C_to_R2 : ℂ →L[ℝ] ℝ × ℝ := complex.equiv_real_prodₗ -- l'application de
ℂ dans ℝ²
def R2_to_C : ℝ × ℝ →L[ℝ] ℂ := complex.equiv_real_prodₗ.symm -- sa reciproque
```

Expliquons cette première ligne. La commande `def` annonce la définition d'un objet mathématique. Ici, son nom est `C_to_R2`. Après les `:`, on donne le type de notre objet, ici une fonction de $\mathbb{C}$ dans $\mathbb{R}^2$ qui est $\mathbb{R}$–linéaire et continue. C'est le fait d'écrire L[ℝ] qui donne la linéarité et la continuité.

Enfin, après les `:=`, on donne la définition de l'objet.

Maintenant, passons à notre première propriété dont nous avons parlé : `realify`. On veut donc, à partir d'une fonction de $\mathbb{C}$ dans $\mathbb{C}$, obtenir une fonction de $\mathbb{R}^2$ dans $\mathbb{R}^2$.

On va donc utiliser une nouvelle fois la commande `def` avec cette fois une variable en argument : la fonction $f : \mathbb{C} \to \mathbb{C}$. La fonction `realify` est en fait la composée de f avec les deux fonctions que nous avons définies juste avant, dans le bon sens.

```
def realify (f : ℂ → ℂ) : ℝ × ℝ → ℝ × ℝ := C_to_R2 ∘ f ∘ R2_to_C
```

3.D – Un premier essai de multiplication

Il nous faut ensuite la multiplication complexe, vu comme une application linéaire continue de $\mathbb{R}^2$ dans $\mathbb{R}^2$, que l'on va appeler `multiply`.On utilise donc une fois de plus la commande `def`.

La définition est donc simple à donner :

```
def multiply (z : ℂ) : (ℝ × ℝ →L[ℝ] ℝ × ℝ) :=
```

Ici, on utilise la syntaxe := car il nous faut définir ce que fait cette fonction, et pourquoi elle est $\mathbb{R}-$linéaire et continue. On va écrire ces éléments après la commande `by{`.

Dans un premier temps, on utilise la commande `refine` afin que L∃∀N nous donne les éléments attendus :

```
def multiply (z : ℂ) : (ℝ × ℝ →L[ℝ] ℝ × ℝ) := by {
  refine ⟨_,_⟩,
```

On a deux _, donc deux choses à donner à L∃∀N :

```
2 goals
z: ℂ
⊢ ℝ × ℝ →ₗ[ℝ] ℝ × ℝ
z: ℂ
⊢ auto_param (continuous ?m_1.to_fun) (name.mk_string "continuity'"
    (name.mk_string "interactive" (name.mk_string "tactic" name.anonymous)))
```

Premièrement, L∃∀N attends une fonction $\mathbb{R}-$linéaire de $\mathbb{R}^2$ dans $\mathbb{R}^2$. Puis, le second objectif est la continuité de cette fonction (dit dans des termes bien complexes).

On utilise une nouvelle fois `refine` pour avoir le détail du premier objectif :

```
def multiply (z : ℂ) : (ℝ × ℝ →L[ℝ] ℝ × ℝ) := by {
  refine ⟨_,_⟩,
   { refine ⟨_,_,_⟩,
```

L'utilisation des accolades permet de se concentrer sur un seul objectif à la fois. Voici ce que nous dit L∃∀N :

```
3 goals
z: ℂ
⊢ ℝ × ℝ → ℝ × ℝ
z: ℂ
⊢ ∀ (x y : ℝ × ℝ), ?m_1 (x + y) = ?m_1 x + ?m_1 y
z: ℂ
⊢ ∀ (r : ℝ) (x : ℝ × ℝ), ?m_1 (r · x) = ⇑(ring_hom.id ℝ) r · ?m_1 x
```

Voici donc ce qu'est une application linéaire : c'est la donnée d'une application, la preuve de son additivité et celle de son homogénéité.

On donne donc l'application : il s'agit de la réalification de la multiplication dans $\mathbb{C}$ par z, puis les preuves qui sont assez élémentaires puisque L∃∀N sait le faire :

```
def multiply (z : ℂ) : (ℝ × ℝ →L[ℝ] ℝ × ℝ) := by {
  refine ⟨_,_⟩,
   { refine ⟨_,_,_⟩,
      { exact realify (λ w, z * w) },
      { intros, simp [realify], split; ring },
      { intros, simp [realify], split; ring } },
```

Il nous reste maintenant à prouver la continuité de cette application. Cela peut paraître élémentaire, mais essayons de le faire avec L∃∀N . Tout d'abord, on utilise `simp` afin de comprendre l'objectif, on obtient :

```
1 goal
z: ℂ
⊢ continuous (realify (has_mul.mul z))
```

Ici, `has_mul.mul` est le nom donné par L∃∀N pour la fonction que nous avons défini juste avant. Afin de prouver la continuité, on va utiliser la caractérisation de Lipschitz, donnée par `lipshitz_with`, et prouver que notre application est lipschitzienne avec une constante de Lipschitz $K = 2\|z\|$. Pour cela, on utilise la commande `suffices` qui traduit "il suffit de" afin d'amener un nouvel objectif qui permettras de conclure. Voici le code :

```
   def multiplication (z : ℂ) : (ℝ × ℝ →L[ℝ] ℝ × ℝ) := by {
  refine ⟨_,_⟩,
  { refine ⟨_,_,_⟩,
    { exact realify (λ w, z * w) },
    { intros, simp [realify], split, ring, ring },
    { intros, simp [realify], split; ring } },
  { simp,
    suffices : lipschitz_with (nnnorm z * 2) (realify (has_mul.mul z)),
```

et le résultat donné par L∃∀N :

```
2 goals
z: ℂ
this: lipschitz_with (‖z‖₊ * 2) (realify (has_mul.mul z))
⊢ continuous (realify (has_mul.mul z))
z: ℂ
⊢ lipschitz_with (‖z‖₊ * 2) (realify (has_mul.mul z))
```

Il nous faut donc, dans un premier temps, prouver que savoir la fonction Lipschitzienne permet de conclure à sa continuité. Pour cela, on utilise un lemme de la librairie, `lispchitz_with.continuous`, qui dit exactement ce qu'il nous faut. On écrit alors :

```
def multiplication (z : ℂ) : (ℝ × ℝ →L[ℝ] ℝ × ℝ) := by {
  refine ⟨_,_⟩,
  { refine ⟨_,_,_⟩,
    { exact realify (λ w, z * w) },
    { intros, simp [realify], split, ring, ring },
    { intros, simp [realify], split; ring } },
  { simp,
    suffices : lipschitz_with (nnnorm z * 2) (realify (has_mul.mul z)),
      exact lipschitz_with.continuous this,
```

Il reste maintenant à prouver que notre fonction est Lipschitzienne.

3.E – La multiplication que nous allons utiliser