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

# TABLE DES MATIÈRES

# PARTIE I 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 droit 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 cohérentes : l'ordre des arguments, la nécessité ou non de telle ou 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 constituer une librairie suffisante pour que l'utilisation de L∃∀N devienne accessible à tous niveaux. Cet objectif est néanmoins sujet à discussion, puisque l'utilisation, même avec une librairie complète, de L∃∀N pour la recherche fondamentale est assez corsée.

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, ou même parfois triviaux.

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. Comme nous n'apporteront pas de nouveaux éléments mathématiques aux connaissances acquises lors du premier semestre, le lecteur ne se trouvera pas étonné de trouver dans ce rapport un contenu mathématique restreint.

# PARTIE II
# 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.

## 1 Les tactiques de résolution – exemple

Voici un énoncé simple que nous allons étudier dans un premier temps.

**Code lean : Inégalités de réels**

```
example (a b c  : ℝ) (hc : 0 ≤ c) (hab : a ≤ b) : a*c ≤ b*c :=
begin
  rw ← sub_nonneg,
  have h_fact : b*c - a*c = (b - a)*c,
  { ring },
  rw h_fact,
  apply mul_nonneg,
  { rw sub_nonneg,
    exact hab },
  { exact hc },
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.

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.

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 1 goal
2 a b c : ℝ
3 hc: 0 ≤ c
4 hab: a ≤ b
5 ⊢ 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**<br>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 $P \to R$ en $Q \to R$.

Une variante est nécessaire dans notre cas, puisque l'on veut réécrire l'implication inverse. Pour cela, il suffit d'ajouter ← après le `rw`. On écrit donc `rw ← 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 :

```
⊢ 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 :

```
2 goals

abc: ℝ
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**<br>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**<br>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` :<br>— `simp` utilise certains lemmes de la librairie,<br>— `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.

## 2 Les tactiques de travail

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**<br>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 corresponde à donner l'application, prouver son additivité et prouver son homogénéité (cf plus loin). |
|---|---|---|

Nous verrons l'utilité de cette tactique plus loin.

Lorsque nous travaillons sur une preuve, qui n'est donc pas finie, L∃∀N annonce à chaque ligne des erreurs liées à la fin de la preuve. Afin d'éviter cette surcharge d'informations, nous pouvons utiliser le mot magique `sorry`. Lors de la recherche et de la rédaction d'une preuve, on va par habitude écrire `sorry` à la fin, où sur chaque élément de preuve de sorte que L∃∀N accepte notre lemme et que l'on puisse l'utiliser par la suite, sans avoir encore fini la preuve.

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`.

Nous avons qualifié ces tactiques de "tactiques de travail" car, dans le format de preuve présent dans la librairie `mathlib`, c'est-à-dire un format condensé, on ne retrouve pas ces tactiques. Elles sont pourtant indispensables lors de la recherche et la création des preuves, comme vous pourrez l'observer lors de la présentation de ce rapport.

Une bonne façon d'apprendre à utiliser L∃∀N est de résoudre des exemples. Dans un premier temps, il est bienvenu de découvrir L∃∀N à l'aide du jeu "Natural Number Game" (disponible ICI) qui propose la découverte de L∃∀N via les démonstrations des lemmes les plus évidents sur les entiers naturels. On y découvre aussi de quelle façon sont implémentés les entiers naturels, avec les axiomes de Peano.

Ensuite, pour continuer l'apprentissage, on peut renvoyer aux tutoriels de Partick Massot, disponibles sur Github ICI, qui proposent de coder quelques résultats d'analyse du premier cycle, concernant les limites et les suites.

N'ayez pas peur, l'apprentissage prends du temps, il faut s'exercer sur le logiciel avant d'être assez à l'aise pour s'attaquer à des résultats plus complexes. Acquérir des méthodes de recherche ainsi qu'une certaine connaissance de la librairie requiert du temps.

En ce qui concerne la librairie `mathlib`, avec un peu d'habitude, on peut anticiper le nom du lemme recherché, puisqu'ils sont tous nommés avec la même méthode.

Ajouter paragraphe Type et propositions, fonctions envoie preuves sur preuves

# PARTIE III

# L'ANALYSE COMPLEXE PAR LEAN

Nous allons mettre en pratique les connaissances acquises via les tutoriels et jeux préliminaires dans une adaptation sur L∃∀N de certains résultats d'analyse complexe.

Le travail permettant d'arriver à l'écriture complète d'une preuve prenant du temps, nous nous concentrerons dans ce rapport sur les biens connues équations de Cauchy-Riemann, dont nous rappelons l'énoncé ci-dessous :

**Équations de Cauchy-Riemann**

Soit $f$ une fonction définie au voisinage de $z_0 = (x_0, y_0)$. Les assertions suivantes sont équivalentes.

- ▷ **1)** La fonction $f$ est $\mathbb{C}-$dérivable en $z_0$.
- ▷ **2)** La fonction $f$ est $\mathbb{R}^2-$différentiable en $(x_0, y_0)$ et $f$ satisfait l'équation de Cauchy-Riemann (complexe)
$$\frac{\partial f}{\partial x}(x_0, y_0) + i\frac{\partial f}{\partial y}(x_0, y_0) = 0.$$
De façon équivalente, si $f = P + iQ$, $P$ et $Q$ satisfont les équations de Cauchy-Riemann (réelles)
$$\frac{\partial P}{\partial x}(x_0, y_0) = \frac{\partial Q}{\partial y}(x_0, y_0)$$
et
$$\frac{\partial P}{\partial y}(x_0, y_0) = -\frac{\partial Q}{\partial x}(x_0, y_0)$$
- ▷ **3)** La fonction $f$ est $\mathbb{R}^2-$différentiable en $(x_0, y_0)$ et sa différentielle est $\mathbb{C}-$linéaire.

## 1 Posons un 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

```
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 :

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

Pour finir, ajoutons le code

```
noncomputable theory
```

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

## 2 Le premier lemme

Commençons notre étude de l'analyse complexe par un premier résultat, qui va nous amener aux équations de Cauchy-Riemann. En voici l'énoncé mathématique :

**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`. Dans notre situation, `has_deriv_at f f' z` signifie que la fonction $f$ est dérivable en $z \in \mathbb{C}$ de dérivée $f'$.

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) :
```

Notons 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 :=.

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 et est accepté par L∃∀N .

### 2.A — Un premier essai

Nous avons tout d'abord défini `realify f` comme étant la composition de `f` avec les fonctions qui vont de $\mathbb{C}$ dans $\mathbb{R}^2$ et réciproquement. Avec la fonction `f` en argument, nous obtenons le code suivant :

```
def realify (f : ℂ → ℂ ) : (ℝ × ℝ → ℝ × ℝ) := equiv_real_prod.to_fun ∘ f ∘
  equiv_real_prod.inv_fun
```

On notera l'utilisation ici des attributs `to_fun` et `inv_fun` qui permettent l'utilisation d'un sens ou de l'autre d'une bijection. Naturellement, L∃∀N comprends le sens direct d'une bijection, c'est-à-dire `to_fun`. Il est aussi possible de remplacer `.inv_fun` par `.symm`.

Une autre nouveauté dans cette ligne de code est la commande `def`. Comme son nom l'indique, elle permet de définir un objet, notamment des fonctions. On écrit donc `def` puis le nom de la fonction, on ajoute : et on écrit le type de l'objet, ici une fonction de $\mathbb{R}^2$ dans $\mathbb{R}^2$. Enfin, après `:=`, on donne la définition.

Nous avons maintenant notre fonction qui transforme une fonction sur $\mathbb{C}$ en une fonction sur $\mathbb{R}^2$. Il nous faut maintenant notre multiplication. Nous allons la définir comme une application linéaire et continue de $\mathbb{R}^2$ dans $\mathbb{R}^2$, ce qui nous donne :

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

Notons que le caractère linéaire et continue est donné par la syntaxe → L[ℝ]. Nous n'allons pas pouvoir ici juste dire qu'il s'agit de la multiplication, puisqu'il nous faudra aussi prouver la linéarité et la continuité. Pour observer tout ceci, nous allons utiliser `refine`.

```
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 :

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 (nnorm 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 (nnorm z * 2) (realify (has_mul.mul z)),
      exact lipschitz_with.continuous this,
```

Il reste maintenant à prouver que notre fonction est Lipschitzienne. C'est malheureusement quelque chose qui n'est pas simple. Après de nombreuses recherches, voici le code nécessaire pour prouver cette caractérisation lipschitzienne :

**Code lean : La multiplication est lipschitzienne**

```
def rabs : ℝ → ℝ := abs

```

```
lemma l1 (z w : ℂ) : (realify (has_mul.mul z) (equiv_real_prod w)) = equiv_real_prod.to_fun (z * w) := rfl

lemma l2 (z1 z2 : ℂ) : equiv_real_prod (z1 - z2) = equiv_real_prod z1 - equiv_real_prod z2 := by { refl }
lemma l3 (z1 z2 : ℂ) : equiv_real_prod (z1 - z2) = equiv_real_prod.to_fun z1 - equiv_real_prod.to_fun z2 := by { refl }

lemma l6 (z : ℂ) : nnnorm (equiv_real_prod z) ≤ nnnorm z := by {
  rw prod.nnnorm_def, simp [nnnorm, norm], split,
  apply abs_re_le_abs, apply abs_im_le_abs
}

lemma l7 (z : ℂ) : nnnorm z ≤ 2 * nnnorm (equiv_real_prod z) := by {
  simp [nnnorm,norm], rw ← subtype.coe_le_coe, simp,
  transitivity rabs z.re + rabs z.im,
  exact complex.abs_le_abs_re_add_abs_im z,
  rw two_mul, apply add_le_add, apply le_max_left, apply le_max_right,
}

lemma l5 {z xy : ℂ} : ‖equiv_real_prod (z * xy)‖₊ ≤ nnnorm z * 2 * ‖equiv_real_prod xy‖₊ :=
begin
  transitivity nnnorm (z * xy), apply l6,
  simp_rw mul_assoc,
  simp,
  apply mul_le_mul le_rfl,
  apply l7, apply zero_le, apply zero_le,
end

lemma l8 (z : ℂ) : lipschitz_with (nnnorm z * 2) (realify (has_mul.mul z)) :=
begin
  rw [lipschitz_with], intros x y,
  set x' := equiv_real_prod.inv_fun x with h1,
  set y' := equiv_real_prod.inv_fun y with h1,
  have : x = equiv_real_prod x' := by { simp [x'] }, rw this, rw l1,
  have : y = equiv_real_prod y' := by { simp [y'] }, rw this, rw l1,
  rw [edist_eq_coe_nnnorm_sub,edist_eq_coe_nnnorm_sub,←l2,←l3],
  rw ← mul_sub,
  set xy := x' - y' with hxy, rw ←hxy,
  rw ← ennreal.coe_mul,
  apply ennreal.coe_le_coe.mpr,
  exact l5
end
```

Et ainsi, on peut conclure la définition de `multiplication` par un simple `exact l8 z`.

Pour cette preuve, il a nécessité un grand nombre de lemmes intermédiaires lors de la recherche. Il est possible de réduire grandement la rédaction de ces lemmes grâce aux raccourcis d'écriture de L∃∀N .

Nous ne l'avons pas fait ici car, en tentant la preuve de notre `cauchy_riemann_step_1`, nous nous sommes aperçu que cette définition de multiplication n'est pas la plus simple à manipuler. Nous l'avons donc laissée de côté, ou plutôt, nous avons adapté nos énoncés.

### 2.B — Un simple changement

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
```

Ce sont les fonctions que nous avons vu depuis le début avec la particularité qu'elles sont directement fournies avec la preuve de leur continuité et de leur linéarité (la seule différence est le petit $_l$ après le nom).

Re-définissons maintenant le `realify` avec ces fonctions.

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

Au final, nous avons donné la même définition à `realify`, à la différence près des petites fonctions `C_to_R2` et `R2_to_C` qui sont légèrement différentes des précédentes. Cette petite différence va nous être bien utile par la suite.

On peut d'ailleurs voir que la définition de la multiplication du paragraphe précédent est maintenant beaucoup plus aisée avec ces deux fonctions `C_to_R2` et `R2_to_C`.

**Code lean : La multiplication par un complexe dans $\mathbb{R}^2$ est $\mathbb{R}$−linéaire et continue**

```
def mul_exe (z : ℂ) : (ℝ × ℝ →L[ℝ] ℝ × ℝ) := by {
  refine ⟨_,_⟩, -- on recommence les refine, comme avant
  { refine ⟨_,_,_⟩,
    -- on veut la multiplication par z, mais de ℝ² dans ℝ²
    { exact realify (λ w, w * z) },
    -- LEAN prouve l'éadditivit avec la tactique ring
    { intros, simp [realify], ring},
    -- on continue avec l'ééého mognit, encore une fois avec ring,
    { intros, simp[realify, C_to_R2], split ; ring },
  },
  -- on simplifie et on applique la èrgle de écontinuit sur la composition
  simp [realify], apply continuous.comp,
  -- C_to_R2 est continue
  { exact C_to_R2.continuous },
  -- encore la écontinuit de la composition
  apply continuous.comp,
  -- la multiplication à droite est continue
  { exact continuous_mul_right z },
  -- R2_to_C est continue
  { exact R2_to_C.continuous },
}
```

En revanche, ce n'est toujours pas cette multiplication que nous allons utiliser car elle ne permet pas de conclure sur notre énoncé `cauchy_riemann_step_1`.

Nous allons plutôt utiliser la multiplication par un complexe comme une application de ℂ dans ℂ qui serait continue et ℝ−linéaire. Pour cela, définissons un premier élément : la multiplication par un complexe de ℂ dans ℂ est ℂ−linéaire et continue. Le code est sensiblement le même que celui ci-dessus :

**Code lean : La multiplication par un complexe dans ℂ est ℂ−linéaire et continue**

```
def multiplication (w : ℂ) : ℂ →L[ℂ] ℂ :=
begin
  refine ⟨_,_⟩, -- on demande a LEAN de generer les objectifs de la definition
  {
    refine ⟨_,_,_⟩, -- encore une fois
    exact λ z, w * z, -- LEAN veut une application de ℂ dans ℂ, on lui donne la multiplication par w ∈ ℂ
    exact mul_add w, -- on trouve ensuite la propriete de linearite (on utilise notamment library_search)
    intros, simp, ring, -- on termine avec des tactiques simples
  },
  simp, -- on demande à LEAN de simplifier pour y voir clair
  exact continuous_mul_left w, -- encore un library_search pour trouver la propriete dans mathlib
end
```

On notera que certaines étapes se résolvent de façon beaucoup plus simple, avec l'aide de la librairie et de la tactique `library_search`.

Le code ci-dessus représente la méthode de recherche des preuves. Dans la librairie notamment, les résultats ne sont pas présentés ainsi, mais en version raccourcie. Ici, nous pouvons raccourcir la preuve en utilisant les symboles ⟨⟩ comme ceux utilisés avec `refine`. Plus précisément, on va remplacer les _ utilisés avec `refine` par les éléments de preuve correspondants :

**Code lean : La définition précédente – en version condensée**

```
def multiply (w : ℂ) :  ℂ →L[ℂ] ℂ :=
⟨⟨λ z, w * z, mul_add w, by { intros, simp, ring }⟩, continuous_mul_left w⟩
```

La suite consiste à prouver que cette multiplication est également ℝ−linéaire, ou plutôt de définir une multiplication de ℂ dans ℂ qui est ℝ−linéaire à partir de `multiply`. Pour cela, nous allons utiliser une propriété qui implique la continuité et la linéarité sur le corps des scalaires à partir de celles sur l'espace vectoriel, qui s'appelle `continuous_linear_map.restrict_scalars`. Autrement dit, on écrit :

**Code lean : La multiplication restreinte au corps des scalaires**

```
def real_multiply (f' : ℂ) : ℂ →L[ℝ] ℂ :=
continuous_linear_map.restrict_scalars ℝ (multiply f')
```

Nous allons maintenant pouvoir revenir à notre `cauchy_riemann_step_1`. Mais, avant de le prouver, nous allons devoir adapter l'énoncé avec les nouvelles définitions que nous avons prises. En effet, l'argument de dérivée que nous avions était simplement `multiply f'` qui devait être une application de $\mathbb{R}^2$

dans $\mathbb{R}^2$, $\mathbb{R}$–linéaire et continue. Ici, nous n'avons plus qu'une multiplication dans $\mathbb{C}$ qui est $\mathbb{R}$–linéaire et continue.

Nous allons donc devoir transformer cette application. À première vue, nous pourrions simplement appliquer `realify` à notre fonction. Or, la définition de `realify` n'assure pas la continuité de la composition de fonctions, même si celle-ci est vraie. Nous allons donc définir une version linéaire et continue de `realify` :

```
def realifyₗ (f : ℂ →L[ℝ] ℂ) : ℝ × ℝ →L[ℝ] ℝ × ℝ := C_to_R2 ∘L f ∘L R2_to_C
```

et enfin donner notre énoncé :

**Code lean : Nouvel énoncé**

```
lemma cauchy_riemann_step_1 {f : ℂ → ℂ} {z : ℂ} (f' : ℂ) (hf :
    has_deriv_at f f' z) :
  has_fderiv_at (realify f) (realifyₗ (real_multiply f')) (C_to_R2 z) :=
```

Il nous reste alors la preuve, qui se déroule de façon conventionnelle, avec pour principal outil le théorème `has_fderiv_at.comp` qui n'est rien d'autre que le théorème de dérivation en chaîne. Voici la preuve :

**Code lean : Preuve de la première étape de Cauchy-Riemann**

```
lemma cauchy_riemann_step_1 {f : ℂ → ℂ} {z : ℂ} (f' : ℂ) (hf :
    has_deriv_at f f' z) :
  has_fderiv_at (realify f) (realifyₗ (real_multiply f')) (C_to_R2 z) :=
begin
  -- On donne la preuve que R2_to_C est l'inverse à gauche de C_to_R2
  have zz : function.left_inverse R2_to_C C_to_R2 :=
    complex.equiv_real_prod.left_inv,
  -- on applique la èrgle de édrivation en îchane
  apply has_fderiv_at.comp,
  -- la preuve que C_to_R2 soit édiffrentiable de édiffrentielle elle-êmme
    est :
  { apply C_to_R2.has_fderiv_at},
  -- on applique encore une fois la èrgle de la îchane
  {apply has_fderiv_at.comp,
  -- on demande à LEAN de comparer l'èhypohtse hf restreinte à ℝ avec le
    goal
  -- LEAN donne alors à prouver les édiffrences
  { convert has_fderiv_at.restrict_scalars ℝ hf.has_fderiv_at,
     -- on simplifie le goal : on édveloppe real_multiply, puis multiply
     { simp [real_multiply, multiply,
    continuous_linear_map.restrict_scalars],
        -- deux applications sont égales si elles sont égales en tout point
        apply linear_map.ext, intro x, simp, apply mul_comm
     }, -- il reste maintenant à appliquer l'èhypothse zz que nous avions
    émontre
     { apply zz},},
  -- et voici la preuve que R2_to_C est édiffrentiable de édiffrentielle
    elle-êmme
  {apply R2_to_C.has_fderiv_at, }, },
```

```
end
```

Enfin, la dernière étape, qui n'est pas obligatoire, mais qui correspond au travail mené pour la librairie, est de rendre la preuve plus courte. Elle sera uniquement plus courte en nombre de lignes, car le contenu et la méthode seront les mêmes. Voici ce que l'on obtient :

**Code lean : Preuve précédente – version condensée**

```
lemma cauchy_riemann_step_1 {f : ℂ → ℂ} {z : ℂ} (f' : ℂ) (hf :
    has_deriv_at f f' z) :
  has_fderiv_at (realify f) (realifyₗ (real_multiply f')) (C_to_R2 z) :=
begin
  refine C_to_R2.has_fderiv_at.comp _ (has_fderiv_at.comp _ _
    R2_to_C.has_fderiv_at),
  have zz : function.left_inverse R2_to_C C_to_R2 :=
    complex.equiv_real_prod.left_inv,
  rw zz z,
  convert has_fderiv_at.restrict_scalars ℝ hf.has_fderiv_at,
  simp [real_multiply, multiply, continuous_linear_map.restrict_scalars],
  apply linear_map.ext, intro z, simp, apply mul_comm,
end
```

## 3 Notre deuxième lemme

L'objectif est d'exprimer clairement les relations de Cauchy-Riemann avec les dérivées partielles. Pour cela, nous allons devoir parler de la matrice de la notre application de multiplication.

Commençons par définir la forme générale d'une matrice de multiplication. Pour définir une matrice, on définit un objet mathématique du type `matrix`. On doit donner aussi la taille. Pour cela, on va utiliser les types de la forme `fin n` : c'est le sous-type de $\mathbb{N}$ composé des entiers strictement inférieurs à $n$. Comme on travaille sur $\mathbb{R}^2$, on veut des matrices de tailles $2 \times 2$ à coefficients dans $\mathbb{R}$. Ensuite, pour définir une matrice, la syntaxe est sensiblement la même que la plupart des langages de programmation. Voici la définition :

**Code lean : Matrice de multiplication**

```
def mulmatrix (a b : ℝ) : matrix (fin 2) (fin 2) ℝ :=
![![a,  -b],
  ![b,  a]]
```

Nous allons donc discuter de la matrice de la multiplication par $f'$, c'est-à-dire la matrice avec partie réelle et imaginaire de $f'$ :

$$F = \begin{bmatrix} \mathrm{Re}(f') & -\mathrm{Im}(f') \\ \mathrm{Im}(f') & \mathrm{Re}(f') \end{bmatrix}$$

Cette matrice correspond donc, d'après notre définition, à `mulmatrix (f'.re) (f'.im)`. On veut donc montrer que notre application `real_multiply f'` dans sa version réalifiée, s'exprime par la matrice $F$ ci-dessus. Pour cela, on va plutôt montrer que l'application linéaire qui vient de la matrice $F$ est la même que `real_multiply f'`.

Les outils dont nous auront besoin sont les suivants :

— `matrix.to_lin'` est un lemme qui donne l'équivalence entre une matrice et une application linéaire. Ainsi, `matrix.to_lin' (mulmatrix (f'.re ) (f'.im))` est une application linéaire de $(\text{fin}\, 2 \to \mathbb{R})$ dans $(\text{fin}\, 2 \to \mathbb{R})$.

— `fin_two_arrow_equiv` qui est l'équivalence entre le type $(\text{fin}\, 2 \to \alpha)$ et $\alpha \times \alpha$. Alors, `fin_two_arrow_equiv` $\mathbb{R}$ est l'équivalence entre $(\text{fin}\, 2 \to \mathbb{R})$ et $\mathbb{R} \times \mathbb{R}$.

Nous voulons donc prouver cet énoncé :

La preuve est finalement assez simple, elle parait même être triviale. Du côté de L∃∀N , nous utilisons tout d'abord le fait que nos deux fonctions sont égales si elles sont égales sur tout élément de $\mathbb{R} \times \mathbb{R}$, puis plusieurs simplifications successives et enfin une résolution évidente avec `ring`. Voici la preuve :

**Code lean : Preuve Cauchy-Riemann step 2**

```
lemma cauchy_riemann_step_2 (f' : ℂ) :
  (fin_two_arrow_equiv ℝ) ∘ matrix.to_lin' (mulmatrix (f'.re) (f'.im)) ∘
   (fin_two_arrow_equiv ℝ).symm
    = realify_l (real_multiply f') :=
begin
  funext, -- deux fonctions sont les êmmes si elles sont les êmmes sur tout
   éélment de l'ensemble
  simp [realify_l, C_to_R2, R2_to_C, mulmatrix, real_multiply, multiply],
  split ; ring,
end
```