Computer-assisted proofs in LEAN4

Bachelor's Project Mathematics

November 2025

Student: J. Nijholt

First supervisor: Prof. M. Seri

Second assessor: Dr. A. E. Sterk

Abstract

This thesis explores the use of the proof assistant LEAN4 in the formalization and verification of mathematical proofs. Proof assistants enable mathematicians to express proofs in a fully formal language that can be checked by a computer, ensuring complete logical rigor and eliminating the ambiguities of informal reasoning. We first discuss the theoretical foundations behind LEAN4. This includes the Calculus of Inductive Constructions and the Curry-Howard correspondence. We show how LEAN4 works in practice. We explain common tactics to use in LEAN4, as well as pitfalls that arise as we formalize our proofs. To illustrate the power of LEAN4, we look at the PhysLean library, and discuss some contributions made to this library. Finally, we show the reader how to get involved; we touch upon several open-source projects and educational resources to help readers get involved into the LEAN4 community.

Acknowledgments

First of all, I want to thank my thesis supervisor, Marcello Seri, for his guidance and knowledge throughout the writing of this thesis. His enthusiasm made the process of writing this thesis simply an amazing experience. With this, I also want to thank Alef Sterk, my second supervisor, for assessing my thesis and attending my thesis presentation.

A special thanks also go out to everyone on Zulip that helped me set up and learn about LEAN4. In particular, I want to extend my gratitude to Joseph Tooby-Smith for his guidance, patience, and help with learning and figuring out PhysLean and making my first pull request on GitHub.

Lastly, I want to thank my friends and family for putting up with me when I got frustrated or stuck, especially my friends from the FMF, Gymcord, DitIsPOTS, and my AI Fam and BFFs. In particular, I want to thank Jamara for always being willing to help and be a sounding board, Mikki for helping me figure out how to study and be successful despite my disability, and my mom for her unconditional support throughout my degree.

Contents

1 Introduction

Mathematics is all about writing proofs. An average, good, proof of a mathematical statement consists of a lot of words and implicit assumptions. This proof that we are all used to seeing is also called an informal proof [4, p.48]. Due to their textual nature, the fact that steps are omitted for brevity, and the common abuse of notation for clarity, proofs can be hard to check for correctness. Even the most experienced peer-reviewer can sometimes miss mistakes in a proof. Computers can help us solve this issue.

When we want to use computers to help us check these proofs, we need to develop a language that a computer can understand, that is expressive enough to encode abstract mathematics. This is where the concept of a formal proof come in. A formal proof is a proof that arises totally from the axioms of mathematics. Absolutely no detail can be left out [12]. That being said, a formal proof is not a matter of rigor, it is a matter of detail [4]. All assumptions must be explicitly stated, no steps can be skipped, and notation must be consistent and logical. The details that may be left out in an informal proof are easily inferred from context by a trained mathematician, but not by a computer.

To demonstrate the difference between an informal and a formal proof, let's take a look at the statement that if an integer n is even, then so is n^2. An informal proof could go like this:

Proof. Let n be an even number. Then, there exists an integer k such that $n = 2k$. We then see that $n^2 = (2k)^2 = 4k^2 = 2(2k^2)$. Since n^2 can be written as a product of 2 and some other integer, n^2 is even as well. $\square$

This is a very simple proof, so simple that as mathematicians, we might not even notice the implicit assumptions being made. What assumptions did we miss that makes it so that this proof is not a formal proof? Some unknowns to a computer could be how the square is defined, if our multiplication is commutative, and why can we take out the factor of 2. Furthermore, we implicitly assume that $2k^2$ is an integer. This is not at all obvious to a computer.

Where in the previous text we talked about a computer in general being able to understand proofs, we really mean a proof assistant that is able to understand proofs. A proof assistant is a program that can check whether a formal proof does follow from the basic axioms of mathematics. While advancements with artificial intelligence are made to also have proof assistants generate proofs, automated theorem provers are something different to the proof assistant we focus on.

The first proof assistant is called Automath, developed by Nicolaas de Bruijn in 1967 [6]. Over time, the idea of what a proof assistant could or should be able to do evolved. This led to other frameworks being considered. Over time, as the paradigm shifted, several new proof assistants came onto the market, such as Coq, Isabelle, and LEAN4. Each system works slightly differently behind the scenes and has different syntax. However, all proof assistants work with formal proofs.

The proof assistant this thesis will focus on is LEAN4. There are several reasons we chose LEAN4 over the other proof assistants. First of all, its syntax is fairly user-friendly and easy to learn. Furthermore, the set of basic rules it uses to check the correctness of proofs (also called the kernel) is small, making it quite fast to run. Lastly, it has attracted a lively community, especially after famous mathematicians such as Terence Tao and Peter Scholze took an interest in LEAN4. Especially the amount of community involvement through, for example, the social media platform of the LEAN4 Zulip, makes LEAN4 a great place to learn from and with others.

LEAN4 also has many community projects where scientists work together to prove results and build libraries with said results. The largest of said projects is Mathlib. Mathlib is a library full of mathematical results, and has as an aim to be the largest and most general library of mathematical results. Where LEAN was originally made for computer scientists, the development of Mathlib has made it that LEAN4 is used mostly by mathematicians.

In this thesis, we aim to discover how LEAN4 works, both in theory and in practice. We want to see what the strong points of LEAN4 are, but also what its weak points are. Furthermore, we would like to discover how we can contribute and become part of the community, and how others can do the same.

We first give an overview of LEAN4 in Section 2. Here, we look at why and how LEAN4 works. Specifically, we introduce the theory of filters, a generalization of convergence. We also focus on the weaker points of LEAN4.
Next in Section 3, we focus on one particular community project, namely PhysLean. PhysLean is a library of theorems in (mathematical) physics in LEAN4. We also explain some contributions we have made to the PhysLean library to further demonstrate the use of LEAN4.
Finally in Section 4, we dive deeper into some other community projects, and we suggest ways the average reader can get involved in LEAN4.

2 LEAN4 and Mathlib

2.1 History of LEAN and Mathlib

The first version of LEAN was developed in 2013, primarily by Leonardo de Moura at Microsoft Research [10]. His goal was to create a functional programming language that could also do proof verification, but development was geared towards computer scientists at first. Unlike the proof assistants that existed at the time, LEAN tried to mostly be an easy to use programming environment.

LEAN1 and LEAN2 were still experimental and unstable versions. They mostly served as a way to test new ideas and frameworks. These versions also had support for Homotopy Type Theory; a certain framework for the foundations of mathematics. Support for Homotopy Type Theory was removed after LEAN2 due to the complexity it added to the system maintenance and the speed of the program. More details can be found in Section 2.5.2.

By 2017, LEAN3 was released, the first stable version that could be used by a wider audience. Stability was achieved, amongst others, by improvements to the compiler and the user interface. At the same time, Mathlib [7] was created. Mathlib is a community-maintained library, aiming to be the largest monolithic library to exist. As Mathlib grew, so did the interest of mathematicians in LEAN3, and as a result Mathlib grew again. This positive feedback loop continued, and continues on to this day.

In 2021, LEAN4 was launched. Rather than refining LEAN3, LEAN4 is a re-implementation of LEAN3 into LEAN3. That is, they programmed the core foundations of LEAN3 as a theorem prover in LEAN3 as a functional programming language. This re-implementation was done with the goal to keep as many LEAN3 theorems in Mathlib compatible with LEAN4. LEAN4 is more geared towards the mathematician, who due to Mathlib have become by far the largest user-base of LEAN. Mathlib has continued to grow and evolve alongside LEAN4, which in turn has driven further enhancements to the language itself.

2.2 The theory behind LEAN4

2.2.1 The Calculus of Inductive Constructions

LEAN4 is based on the Calculus of Inductive Constructions, also denoted CIC [10, p. 1]. The calculus of inductive constructions is defined inductively, starting with the ”empty” type *, defining objects over *, and finally using these objects we define contexts.

This is a fairly vague definition. One can think this inductive process in analogy to how the natural numbers can be constructed from the empty set: we start with $0 = \emptyset$, then $1 = \{\emptyset\}$, which iterates to $2 = \{\emptyset, \{\emptyset\}\}$, and so on. 0 would then be the ”empty” type, while objects would be the natural numbers.

In CIC, instead of numbers, we construct sets and objects that serve as labels for types and terms. These objects and contexts are defined through function application, also called a λ-expression. The exact inductive definition is beyond the scope of this thesis, for more details we refer the reader to [9].

The way this calculus of constructions translates to proofs through software, is through something called the Curry-Howard correspondence. The Curry-Howard correspondence states that for any derivation of $\Gamma \to \beta$ in a system of propositional logic we can find a construction of $\Gamma \to \beta$ and conversely [14, p.481]. A derivation is a ”more classic” proof, in our case this is a formal proof. A construction is a λ-expression.

Thus, this Curry-Howard correspondence tells us that if we can find a ”normal” proof, we can translate this into a λ-expression. Then, through the calculus of constructions, this λ-expression can be converted to something a computer can interpret and check for us. Conversely, the computer can then also, after checking, convert this λ-expression back to ”normal” language.

The calculus of construction is a concept in dependent type theory. While dependent type theory and the internal working of LEAN4 fall outside the scope of this thesis, we refer the interested reader to [13].

2.2.2 Currying

One thing we inherit from the λ-expressions that CIC is based on, is that functions cannot have more than one input. This is solved by a process called currying. Currying is named after Haskell Curry, however the process itself was first described by Russian mathematician Moses Schönnfinkel.

To demonstrate, we give a concrete example. Let $\mathcal{F}(X, Y)$ be the space of all function from X to Y. Let $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, defined by $f(x, y) = x^2 + y^2$. If we curry f, we get a function $f_c : \mathbb{R} \to \mathcal{F}(\mathbb{R}, \mathbb{R})$ defined by

$$f_c : x \mapsto (y \mapsto x^2 + y^2).$$

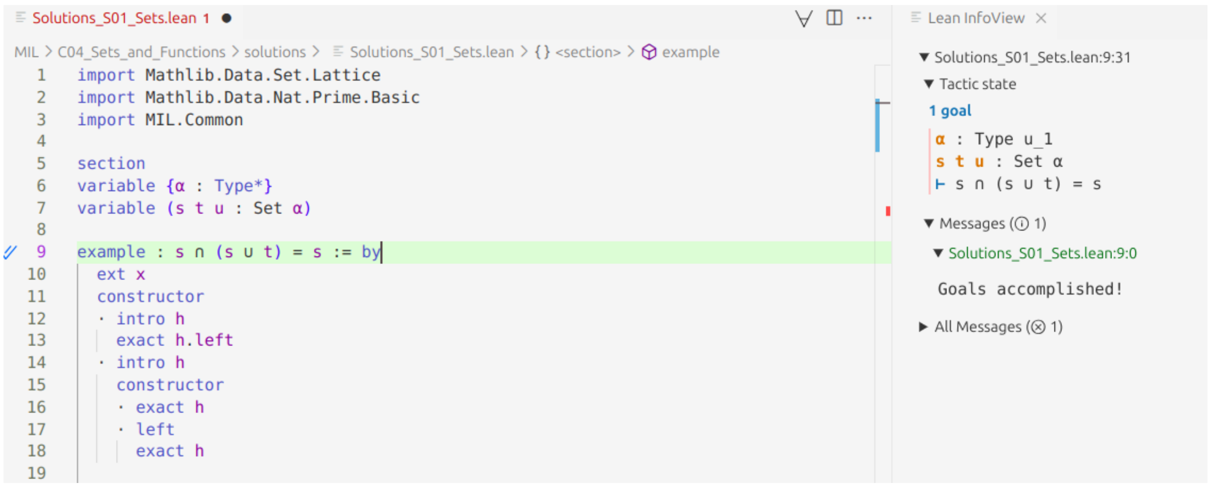
Thus, we have that $f(x, y) = f_c(x)(y)$. While our example shows this for a function with two arguments, we can inductively define this for any finite number of inputs. A function with n inputs would then become n functions that are curried out in succession.

2.3 How to use LEAN4

While LEAN4 is built on deep theoretical foundations, most of this theory remains in the background when one actually uses the system. So, how do we use LEAN4? While many sources have been written with the purpose of teaching LEAN4, here we provide a brief overview of the most common elements appearing in the proofs of this thesis: tactics, which are commands that guide the proof process; terms, which represent mathematical objects; and definitions, which introduce new named concepts.

To introduce these elements, we first need to know how the LEAN4 user interface works. We will demonstrate this by a set-theoretic proof, namely we will show that for sets s, t, we have $s \cap (s \cup t) = s$. When you open a file in LEAN4 and click somewhere in the file, the so-called Lean InfoView opens. In Figure 1, you can see the code open on the left-hand side, and the Lean InfoView on the right. In the InfoView, you will see all your hypotheses, and the statements you want to prove. The final goal you want to prove will be on the bottom, and is denoted with the $\vdash$ symbol.

As you click through the proof, your InfoView changes. In this case, the green bar denotes the line in the code that the InfoView applies to. If we stick to the same proof, but move 2 lines down, we encounter something named ”`constructor`”. This is our first encounter with a tactic. `constructor` is one of the ways that LEAN4 lets you do a proof by case distinction. We will elaborate on this tactic more later. As you can see in Figure 2, in the proof by cases, we have two different cases (namely `h.mp` and `h.mpr`) with both having slightly different goals.


Solutions_S01_Sets.lean 1
MIL > C04_Sets_and_Functions > solutions > Solutions_S01_Sets.lean > {} <section> > example
Lean InfoView

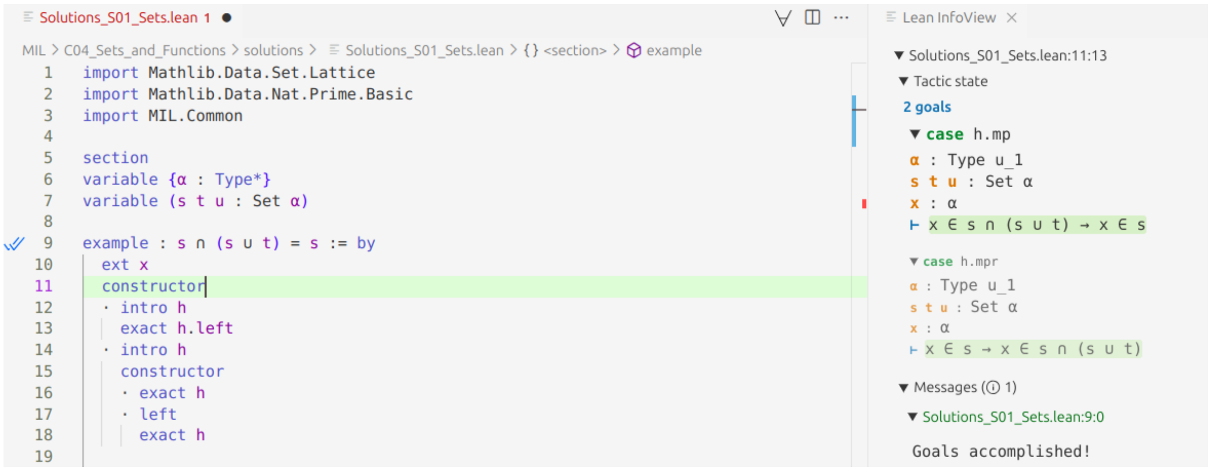
```
import Mathlib.Data.Set.Lattice
import Mathlib.Data.Nat.Prime.Basic
import MIL.Common

section
variable {α : Type*}
variable (s t u : Set α)

example : s ∩ (s ∪ t) = s := by
  ext x
  constructor
  · intro h
    exact h.left
  · intro h
    constructor
    · exact h
    · left
      exact h
```

```
▼ Solutions_S01_Sets.lean:9:31
 ▼ Tactic state
  1 goal
  α : Type u_1
  s t u : Set α
  ⊢ s ∩ (s ∪ t) = s
 ▼ Messages (ⓘ 1)
  ▼ Solutions_S01_Sets.lean:9:0
   Goals accomplished!
▶ All Messages (⊗ 1)
```

Figure 1: when you open LEAN4, the Lean InfoView opens on the right-hand side


Solutions_S01_Sets.lean 1
MIL > C04_Sets_and_Functions > solutions > Solutions_S01_Sets.lean > {} <section> > example
Lean InfoView

```
import Mathlib.Data.Set.Lattice
import Mathlib.Data.Nat.Prime.Basic
import MIL.Common

section
variable {α : Type*}
variable (s t u : Set α)

example : s ∩ (s ∪ t) = s := by
  ext x
  constructor
  · intro h
    exact h.left
  · intro h
    constructor
    · exact h
    · left
      exact h
```

```
▼ Solutions_S01_Sets.lean:11:13
 ▼ Tactic state
  2 goals
  ▼ case h.mp
  α : Type u_1
  s t u : Set α
  x : α
  ⊢ x ∈ s ∩ (s ∪ t) → x ∈ s

  ▼ case h.mpr
  α : Type u_1
  s t u : Set α
  x : α
  ⊢ x ∈ s → x ∈ s ∩ (s ∪ t)
 ▼ Messages (ⓘ 1)
  ▼ Solutions_S01_Sets.lean:9:0
   Goals accomplished!
```

Figure 2: Now, we have two different goals, each with slightly differing hypothesis

Now that we know what LEAN4 visually looks like, we can continue to the actual contents of our proofs. Most of the rest of this subsection is based on the Lean Language Reference [20], with the exact sections hyperlinked.

`noncomputable section` In LEAN4, we can use the existence of an object to create an arbitrary object. This happens through the axiom of choice, denoted `Classical.choice`. In such cases, these arbitrary objects are noncomputable. Note that this means LEAN forces you to be explicit about the use of the axiom of choice, even in finite sets. To make sure LEAN4 does not try to attempt computations with such arbitrary objects, we can open a `noncomputable section`. In this way, LEAN4 knows not to use any computational power and instead it will just let the object exists.

theorem `theorem` is a way to declare the facts you want to prove. They can be given a name that can be referred to later on. An example is

```
theorem add_left_cancel {R : Type*} [Ring R] {a b c : R} (h : a + b = a + c) :
b = c := by
    sorry
```

In this theorem, we state that for any ring R and elements $a, b, c \in \mathbb{R}$, we have that if $a + b = a + c$ (labeled `h`), then $b = c$. Here, you can notice that in our declaration of our variables, we use square, curly, and normal brackets. These all have different meanings.

- The normal, round, brackets denote explicit arguments. These arguments you usually have to explicitly provide when you call the lemma later on. It also lets you add hypotheses (such as `h` above)
- The curly brackets denote implicit arguments. LEAN4 will always try to infer the arguments that are denoted in curly brackets, and you rarely have to provide these yourself.
- Square brackets are for type classes. They denote things such as the structure on your field.

LEAN4 also has a `lemma` attribute. This works the exact same as `theorem` does. The difference in use is similar to what it is in informal mathematics; smaller, intermediate results get tagged as a `lemma`, while larger, more final results get tagged with the `theorem` attribute.

def `def`, as the name suggests, lets you define things. An example of a definition is

```
def deriv (f : 𝕜 → F) (x : 𝕜) :=
    fderiv 𝕜 f x 1
```

Here, we are defining a concept named `deriv`, for any function $f : \mathbb{k} \to F$ and $x \in \mathbb{k}$, $\text{deriv}\, f\, x = \texttt{fderiv}\mathbb{k}\, f\, x\, 1$. `fderiv` is defined in Mathlib, the exact definition goes beyond the scope of this section.

There are also ways to define structures (such as rings, topological spaces, or vector spaces), but these are not defined through `def`. Instead, `structure`, `class`, or `instance` are used for these types of definitions. More information on these last three can be found in the Lean Language Reference [20].

ext `ext` is short for extensionality. Extensionality states that two objects are equal if and only if they have the same properties. For sets, as in our example, this means that they have the same objects.

sorry One tactic you hopefully will not see much in this thesis outside of artificial examples, but is incredibly useful to know, is `sorry`. `sorry` can be used in place of any proof. This way, if one gets stuck on a certain part of the proof, you can replace that part by `sorry` and finish the rest of the proof, before going back to it later.

rfl `rfl` checks for definitional equality. If you have a goal of the form "$a = a$", `rfl` will close the goal, that is, consider the goal as fully proven.

constructor `constructor` enables us to split up any statement that needs two (or more) facts proven into separate goals. The most noteworthy for our purposes is for "if and only if" goals. Using `constructor` splits this goal up into two new goals; one "if" goal, and one "only if" goal.

intro intro lets you introduce hypotheses or objects. The 2 main use cases for this tactic are with implications, and with ”for all” statements For example,

```
-- example with implication
-- ⊢ a → b
intro ha
-- ha : a
-- ⊢ b

-- example with for all
-- ⊢ ∀ (x : E) f x = 0
intro x
-- x : E
-- ⊢ f x = 0
```

have The have tactic lets you give an explicit statement that you may need in a proof. Once this statement is proven, it is added as an hypothesis. It is particularly good for proofs that are small facts that are not large enough to become statements on their own, or facts that are extremely dependent on the local context and would not make sense on their own. Some syntax examples include

```
    have h : x ^ 2 > 0 := by
    sorry
    -- adds hypothesis h stating that x ^ 2 > 0.
    -- A proof for this fact would be put in the place of sorry.
    have : x ^ 2 > 0 := by
    sorry
    -- adds hypothesis this stating that x ^ 2 > 0.
    -- A proof for this fact would be in the place of sorry.
    have h := by exact sq_le x
    -- adds a hypothesis h stating that x ^ 2 \leq x, the statement of sq_le
    have := by exact sq_le x
    -- adds a hypothesis this stating that x ^ 2 \leq x, the statement of sq_le.
```

For the sake of space, we leave out the proofs, and instead use the sorry tactic described in 2.3.

One related tactic is that of haveI. In most cases, have and haveI work the exact same. As such, we do not dive into the differences in this thesis.

let let is quite similar to have, and will let you add hypotheses. However, you cannot unfold a definition introduced with have, while you can do so for definitions introduced with let.

obtain If you have a hypothesis with an ”exists” qualifier (∃), obtain lets you extract an element, together with a specific hypothesis. . For example, if we have the hypothesis $h : \exists x : \mathbb{R}, |x| = x$, we can write

```
x y : ℝ
have h: ∃ x : ℝ, |x| = x
obtain ⟨y, hy⟩ := h
-- hypothesis h disappears, new hypothesis is made stating hy: |y| = y
```

apply `apply` is like a way to reason backwards. For example, take the Heine-Borel theorem, stating that any closed bounded subset C of $\mathbb{R}^n$ is compact. Suppose we want to show that the subset $[0,1] \in \mathbb{R}$ is compact. If we apply Heine-Borel to that goal, we get two new goals coinciding with the hypotheses of the Heine-Borel theorem, namely our two new goals will be that $[0,1]$ is closed, and that $[0,1]$ is bounded. This way, a proof can be split into smaller, more approachable goals. If one of the new goals was already a hypothesis, then LEAN4 will close this goal automatically.

We can also find an example in LEAN4 notation:

```
-- first we state our theorem
theorem sqrt_le_sqrt (h : x ≤ y) : √x ≤ √y := by
  rw [Real.sqrt, Real.sqrt, NNReal.coe_le_coe, NNReal.sqrt_le_sqrt]
  exact toNNReal_le_toNNReal h

-- now our example proof
-- ⊢  √x ≤ √y
apply sqrt_le_sqrt
-- ⊢ x ≤ y
```

exact `exact` closes the goal if the argument you give it matches the goal. Let us take Heine-Borel again. If we have a hypothesis h_1 : $[0,1]$ is closed, and a hypothesis h_2 : $[0,1]$ is bounded, and a goal $[0,1]$ is compact. Then, `exact Heine-Borel h_1 h_2` would close our goal. Another example in LEAN4 notation:

```
-- first we state our theorem
theorem sqrt_le_sqrt (h : x ≤ y) : √x ≤ √y := by
  rw [Real.sqrt, Real.sqrt, NNReal.coe_le_coe, NNReal.sqrt_le_sqrt]
  exact toNNReal_le_toNNReal h

-- now our example proof
-- hxy : x ≤ y
-- ⊢  √x ≤ √y
exact sqrt_le_sqrt hxy
-- Goals accomplished!
```

`apply` and `exact` are quite similar techniques whenever all the hypotheses for your theorem to apply are already proven. However, especially in larger proofs, using one over the other can still give you a certain benefit. For example, `exact` is faster, since it does not automatically run `rfl` and thus does not contain any automation.

refine `refine` behaves very similar to `exact`. The main difference is that `refine` is allowed to have unknowns; those then get turned into new goals.

symm `symm` can use any symmetric relation (such as $=$) and rewrite it; if we have a goal of the form `a = b`, applying `symm` gives us a new goal of `b=a`. Note that `symm` works for all symmetric relations; equality is just the most common symmetric relation. `symm` can also be used in a hypothesis `h`, by using `symm at h`.

rw rw [arg_1,…, arg_n] lets you rewrite the goal using n arguments. After doing so, it applies rfl automatically to try and close the goal. You can also add a ← to rewrite something the other way. For example,

```
-- h1: a = b
-- h2 : b = c
-- ⊢ a * a + c * a = 0
rw [h1]
-- ⊢ b * b + c * a = 0
rw [← h2]
-- ⊢ b * b + b * a = 0
```

Furthermore, rw can be applied at a hypothesis ”hyp” by rw [arg_1, …, arg_n] at hyp.

Sometimes you don't want to rewrite every instance in your goal. In that case, you can use nth_rewrite. For an example of the syntax

```
-- h: a = b
-- ⊢ a * a + c * a = 0
nth_rewrite 2 [h]
-- ⊢ a * b + c * a = 0
```

unfold unfold allows us to replace the definition it is given by its content. It will only do so once, and not recursively. For example,

```
def deriv (f : 𝕜 → F) (x : 𝕜) :=
    fderiv 𝕜 f x 1

example {f : ℝ → ℝ} {x a : ℝ} (h : HasDerivAt f a x) : deriv f x = a := by
    -- ⊢ deriv f x = a
    unfold deriv
    -- ⊢ (fderiv ℝ f x) 1 = a
```

congrFun congrFun lets you rewrite a generalized function to a function with a more specific input. For example

```
have this : f = fun x => 1 / 2 * x
have func : congrFun this t
-- func: f t = 1 / 2 * t
```

calc calc is a way to carry out calculations. It works with most transitive relations, such as $<$, $\leq$, and $=$. It is even possible to define your own transitive relations and use these in a calc block. An example of how to use calc with several transitive relations can be found in the next proof:

```
theorem T
(h1 : a = b)
(h2 : b < c + 1)
(h3 : c ≤ d)
(h4 : e = 1 + d) :
a < e :=
    calc
        a = b       := h1
        b < c + 1   := by rw [h2]
        c + 1 ≤ d + 1  := by rw [h3]
        d + 1 = 1 + d  := by rw [Nat.add_comm]
        1 + d = e      := by rw [h4]
```

where Nat.add_comm is the statement that addition is commutative.

The `calc` tactic can also be combined with `simp` and other tactics, and it is even possible to have nested `calc` blocks.

In this proof, you can see we start each new line with the result of the previous line. While this is a simple proof, we do not always want to repeat the outcome of the previous line. If we do not want to repeat ourselves, we can use underscores to tell LEAN4 to "continue where we left off". This can massively help with readability of more complicated proofs, and thus underscores are used extremely often. Using underscores, the same proof as above could look as follows:

```
theorem T
(h1 : a = b)
(h2 : b < c + 1)
(h3 : c ≤ d)
(h4 : e = 1 + d) :
a < e :=
    calc
        a = b       := h1
        _ < c + 1   := by rw [h2]
        _ ≤ d + 1   := by rw [h3]
        _ = 1 + d   := by rw [Nat.add_comm]
        _ = e       := by rw [h4]
```

`simp` As the name suggests, `simp` allows us to make simplifications. It works by applying lemmas until the expression is in some sense at its most simple form. There are five forms of `simp` we focus on: `simp`, `simp [arg]`, `simp_all [arg]`, `simp only [arg]`, and `simp_all only [arg]`.

- `simp` as a standalone tactic uses a database of lemmas from Mathlib.. We can use `simp at h` to use the database to simplify a hypothesis. The lemmas of this database can be found all throughout Mathlib, tagged as `@simp`.
- `simp [arg]` uses the given arguments to simplify your goal expression, as well as the database that `simp` uses. The arguments can also be hypotheses that you have defined within your proof. Again, `simp [arg] at h` uses the given arguments to simplify the hypothesis labeled `h`.
- `simp_all [arg]` uses both the database and the arguments given to it to simplify both the goal and the hypotheses at the same time.
- `simp only [arg]` uses only the arguments given to simplify, and forgets about the database in Mathlib.
- `simp_all only [arg]` simplifies both the goal and the hypotheses with the arguments given.

The difference between `simp only` and `unfold`, is that `simp only` will unfold the definition recursively, while `unfold` only does so once. This also means that `simp` can produce recursion errors, and thus we should be mindful of this while using `simp`.

`aesop` `aesop` [16] is short for "Automated Extensible Search for Obvious Proofs". `aesop` is a fairly crude algorithm; it tries every rule available, then applies itself to all resulting goals until it terminates. This way, simple proofs can be done automatically. By using `aesop?`, the infoview will print the proof obtained by `aesop`.

other automation There are several other ways to automate in LEAN4. We introduce three here.

- `ring` uses the axioms of rings and theorems that follow after to simplify algebraic expressions in rings. One can use `ring?` to access what theorems were used by the `ring` tactic.
- `ring_nf` is a slightly more powerful version of `ring`. It knows a few more theorems. Once again, `ring_nf?` can be used to access the theorems that were used by `ring_nf`.
- `field_simp` does the same as `ring`, but it uses the axioms of a field (and the theorems that follow from them) instead of for a ring. Here too we can use `field_simp?` to obtain the theorems used by `field_simp`.

Now that we have seen all these different tactics, let's circle back to the proof we started this section with in Figures 1 and 2.

```
example : s ∩ (s ∪ t) = s := by
  ext x
  -- goal changes to x ∈ s ∩ (s ∪ t) ↔ x ∈ s
  constructor
  -- two goals are created
  -- first goal: x ∈ s ∩ (s ∪ t) → x ∈ s
  · intro h
      -- new hypothesis h: x ∈ s ∩ (s ∪ t)
      -- new goal: x ∈ s
    exact h.left
    -- h.left states that x ∈ s
  -- we start a new goal, hypothesis h disappears
  -- second goal: x ∈ s → x ∈ s ∩ (s ∪ t)
  · intro h
      -- new hypothesis h : x ∈ s
      -- new goal: x ∈ s ∩ (s ∪ t)
    constructor
    -- again two goals are created
    -- first goal: x ∈ s
    · exact h
    -- is exactly what h states
    -- second goal : x ∈ s ∪ t
    · left
      -- new goal: x ∈ s
      exact h
      -- exactly what h states.
```

2.4 Generalization in LEAN4

When writing informal proofs, it is often easier to start with a very restricted viewpoint. For example, it is easier to write a proof about a ring than about a field, because a ring has more structure to it. Similarly, it is easier to write proofs about ℝ than about a general manifold, again because ℝ has a lot of added structure. In most cases, the less generalization we have, the more structure there is for us to work with.

When development of Mathlib started, the choice was made to have Mathlib contain theorems in as much generality as possible. That way mathematicians of any background and any level could find use of Mathlib, without the library getting too large. Due to this choice however, the added structure that comes with specificity can become over-restrictive. We look at two instances where generalization actually makes our proving easier.

2.4.1 Limits and filters

The first instance in which generalization is a helpful tool is in the case of convergence and limits. Usually, the first encounter with limits is in an introductory calculus class, when we encounter the ε-δ definition for a limit of a function $f : \mathbb{R} \to \mathbb{R}$.

Definition 2.1 (Limit of a real function)**.** Let $f : \mathbb{R} \to \mathbb{R}$ be a function defined on some open interval that contains the number a, except possibly on a itself. We say that $\lim_{x\to a} f(x) = L$ if, for any $\varepsilon > 0$, there exists a $\delta > 0$ such that if $0 < |x - a| < \delta$, we have $|f(x) - L| < \varepsilon$. [19, p. 25]

After we get comfortable with this definition, we generalize our definition of a limit to a limit and convergence of a real sequence

Definition 2.2 (Limit of a real sequence)**.** The sequence (s_n) of real numbers converges to the real number ℓ if for any $\varepsilon > 0$, there exists an integer N_ε such that $|s_n - \ell| < \varepsilon$ for all $n \geq N_\varepsilon$. [19, p. 21]

Finally, we move on to a topology class and we generalize even further into sequences and converges in a metric space

Definition 2.3 (Limit in a metric space)**.** Let (X, d) be a metric space. Let $B_r(x_0) = \{x \in X : d(x, x_0 < r)\}$ denote the open ball of radius r around a point $x_0 \in X$. A sequence (x_n) in X converges to a point $x \in X$ if for any real $\varepsilon > 0$, there exists an integer N such that $x_n \in B_\varepsilon(x)$ whenever $n \geq N$. [19, p.68]

Outside of these three basic definitions, there are many more ways that convergence is defined in mathematics. We can talk about limits in different directions (left and right for $f : \mathbb{R} \to \mathbb{R}$, and even more directions for function $f : \mathbb{R}^n \to \mathbb{R}^m$), diverging limits (limits to infinity), convergence of sequences of functions, and many more.

If we were to formalize limits in a "traditional" way then, we would get a lot of different definitions, with a lot of very specific use cases. This would be incredibly time- and space consuming. Furthermore, we will find ourselves re-proving theorems for several of these cases. In an informal proof it is easy enough to prove one case and then say the other cases follow analogously. This however no longer works for a formal proof. To avoid these problems, we would like a way to generalize convergence in a way that we can specify it to each of our use cases, while only needing to prove the underlying theorem once. Such a generalization can be found in Bourbaki's theory of filters [5].

Definition 2.4 (Filters)**.** Let X be a set and $\mathfrak{F}$ a set of subsets of X. Let $U, V \in X$. Then, $\mathfrak{F}$ is a *filter* if

1. Whenever U belongs to $\mathfrak{F}$ and U is contained in V, then also V belongs to $\mathfrak{F}$;
2. Any finite intersection of sets of $\mathfrak{F}$ belongs to $\mathfrak{F}$;
3. The empty set does not belong to $\mathfrak{F}$.

The curious reader might now be wondering how a filter is a generalization of a limit. For demonstrational purposes, let us demonstrate how the filter specializes to $f : \mathbb{R} \to \mathbb{R}$, as in Definition 2.2. To do so, we must define a specific filter, namely the *neighborhood filter.*

Definition 2.5 (Neighbourhood Filter)**.** Let $U \subset \mathbb{R}$. The neighborhood filter $\mathcal{N}_a$ around a point $a \in \mathbb{R} \cup \{\pm\infty\}$ is defined as follows:

- Let $a \in \mathbb{R}$. Then, $U \in \mathcal{N}_a$ if and only if there exists a $\delta > 0$ such that $(a - \delta, a + \delta) \subset U$.

- Let $a = \infty$. Then, $U \in \mathcal{N}_a$ if and only if there exists $B \in \mathbb{R}$ such that $(B, \infty) \subset U$.
- Let $a = -\infty$. Then, $U \in \mathcal{N}_a$ if and only if there exists $B \in \mathbb{R}$ such that $(-\infty, B) \subset U$.

In this definition, we claim that $\mathcal{N}_a$ is a filter. To get more familiar with the theory of filters, we explicitly show this fact is true.

Theorem 2.1. The neighborhood filter $\mathcal{N}_a$ as defined in Definition 2.5 is a filter as in Definition 2.4.

Proof. To show so, we check the axioms. We first let $a \in \mathbb{R}$.

- Let $a \in \mathbb{R}$.
  1. Let $U \in \mathfrak{N}_a$. Then, there exists a $\delta > 0$ such that $(a - \delta, a + \delta) \subset U$. Since we know $U \subset V$, also $(a - \delta, a + \delta) \subset V$ and thus $V \in \mathcal{N}_a$.
  2. Take $U_1, U_2 \in \mathcal{N}_a$. Then, there exists $\delta_1, \delta_2 > 0$ such that $(a - \delta_1, a + \delta_1) \subset U_1$ and $(a - \delta_2, a + \delta_2) \subset U_2$. Without loss of generality, let $\delta_1 \leq \delta_2$. This means that $(a - \delta_1, a + \delta_1) \subseteq (a - \delta_2, a + \delta_2)$. Now we consider $U_1 \cap U_2$. We notice that $(a - \delta_1, a + \delta_1) \subset U_1 \cap U_2$. Thus, $U_1 \cap U_2 \in \mathcal{N}_a$.
  3. Suppose $\emptyset \in \mathcal{N}_a$. Then there exists $\delta > 0$ such that $(a - \delta, a + \delta) \in \emptyset$. This is a contradiction. Thus, the empty set does not belong to $\mathcal{N}_a$.
- Let $a = \infty$.
  1. Let $U \in \mathcal{N}_a$. Then, there exists $B \in \mathbb{R}$ such that $(B, \infty) \in U$. Suppose $U \subset V$. Then, $(B, \infty) \subset V$ and thus $V \in \mathcal{N}_a$.
  2. Let $U_1, U_2 \in \mathcal{N}_a$. Then there exists B_1, B_2 such that $(B_1, \infty) \subset U_1$ and $(B_2, \infty) \subset U_2$. Without loss of generality, let $B_1 \geq B_2$. Then $(B_1, \infty) \subset (B_2, \infty)$. Then, $(B_2, \infty) \subset U_1 \cap U_2$ and thus $U_1 \cap U_2 \in \mathcal{N}_a$.
  3. Suppose $\emptyset \in \mathcal{N}_a$. Then there exists $B \in \mathbb{R}$ such that $(B, \infty) \subset \emptyset$. This is a contradiction, hence $\emptyset \notin \mathcal{N}_a$.
- Let $a = -\infty$.
  1. Let $U \in \mathcal{N}_a$. Then, there exists $B \in \mathbb{R}$ such that $(-\infty, B) \in U$. Suppose $U \subset V$. Then, $(-\infty, B) \subset V$ and thus $V \in \mathcal{N}_a$.
  2. Let $U_1, U_2 \in \mathcal{N}_a$. Then there exists B_1, B_2 such that $(-\infty, B_1) \subset U_1$ and $(-\infty, B_2) \subset U_2$. Without loss of generality, let $B_1 \leq B_2$. Then $(-\infty, B_1) \subset (-\infty, B_2)$. Then, $(-\infty, B_2) \subset U_1 \cap U_2$ and thus $U_1 \cap U_2 \in \mathcal{N}_a$.
  3. Suppose $\emptyset \in \mathcal{N}_a$. Then there exists $B \in \mathbb{R}$ such that $(-\infty, B) \subset \emptyset$. This is a contradiction, hence $\emptyset \notin \mathcal{N}_a$.

□

Armed with the tool of the neighborhood filter, let us go back to the definition of a limit as in Definition 2.1. It is fairly easy to see that the condition $0 < |x - a| < \delta$ translates to the requirement that $x \in (a - \delta, a + \delta)$ and thus that there exists a $U_1 \in \mathcal{N}_a$ such that $x \in U_1$. Similarly, we have that $|f(x) - L| < \varepsilon$ if and only if $f(x) \in (L - \varepsilon, L + \varepsilon)$. Thus, there exists an $U_2 \in \mathcal{N}_L$ such that $f(x) \in U_2$. Note that we are working with two different neighborhood filters here, one around a and one around L.

We can now rewrite our definition of a limit using filters:

Theorem 2.2. We have that $\lim_{x \to a} f(x) = L$ if and only if for all $U_1 \in \mathcal{N}_a$ and for every $x \in U_1$ there exists a $U_2 \in \mathcal{N}_L$ such that $f(x) \in U_2$.

Note that in this theorem, L can be a real number, or an infinity. If $L = \pm\infty$, we would usually say the limit diverges. For generality purposes, we would here say that a function diverges if and only if it converges to either infinity.

To round things up, let us look at the code for the limit of a function in the Mathlib library

```
/-- `Filter.Tendsto` is the generic "limit of a function" predicate.
`Tendsto f l₁ l₂` asserts that for every `l₂` neighborhood `a`,
the `f`-preimage of `a` is an `l₁` neighborhood. -/
def Tendsto (f : α → β) (l₁ : Filter α) (l₂ : Filter β) :=
    l₁.map f ≤ l₂
```

This definition is saying that for any function $f : \alpha \to \beta$, any filter l_1 on α, and any filter l_2 on β, for any $A \in l_2$, $f^{-1}(A) \in l_1$. Note that here neighbourhood filters are not even explicitly used. They are simply a particular case of this definition. This really makes this definition the most general that is possible.

By taking different filters, several properties of continuity and limits can be preserved. For more details on this, we refer the reader to [5].

2.4.2 Metric and topological spaces

Another instance where generalization makes writing proofs easier is in the instance of metric spaces and topological spaces. Let us recall the definition of a metric space. We use [19] as our reference.

Definition 2.6 (Metric space)**.** A metric space is a tuple (X, d) where X is a nonempty set and $d : X \times X \to \mathbb{R}$ a function (also called the distance function) such that

- For all $x, y \in X$, $d(x, y) \geq 0$ with equality if and only if $x = y$.
- For all $x, y \in X$, $d(x, y) = d(y, x)$.
- For all $x, y, z \in X$, $d(x, z) \leq d(x, y) + d(y, z)$.

In metric spaces, we can define sets to be open through the following definition

Definition 2.7 (Open sets in metric spaces)**.** A set U is open in X if for every $x \in U$, there exists ε_x such that

$$\{y \in X \mid d(x, y) < \varepsilon_x\} \subseteq U$$

A metric space is a specific class of topological spaces, whose topology is given by the set $\mathcal{T}$ of all metric open sets, as defined in Definition 2.7. Recall the definition of a topological space.

Definition 2.8 (Topological space)**.** A topological space $T = (X, \mathcal{T})$ consists of a nonempty set X with a family $\mathcal{T}$ of set of subsets of X satisfying

- $X, \emptyset \in \mathcal{T}$.
- For all $A, B \in \mathcal{T}$, also $A \cap B \in \mathcal{T}$.
- For any collection of sets $A_i \in \mathcal{T}$, also $\bigcup_i A_i \in \mathcal{T}$

One may ask now how a topological space differs from a filter as in Definition 2.4. While their axioms may appear similar, their functions are entirely different. Filters aim to define ”large” sets to help with convergence, while a topology aims to define ”open” sets and have more applications than filters do.

As students, we first work in Euclidean space, slowly increasing the dimension from 1 to 3. Next, we are introduced to metric spaces and finally, topological spaces come into play. It is generally easiest to work with Euclidean 1-space, and the hardest to work with topological spaces. In LEAN4 this is not true; there, it is easiest to work with topological spaces.

So why is it easier to work in topological spaces in LEAN4? This is mostly because of a concept of functoriality. To understand what this means, we have to dive slightly into category theory. We refer the reader to [18] for more details.

Definition 2.9 (Categories)**.** A *category* is a collection of objects $X, Y, Z, \dots$ and a collection of morphisms $f, g, h, \dots$ such that

- each morphism has specified domain and codomain objects;
- each object X has a designated identity morphism $\mathbb{1}_X : X \to X$;
- for any pair of morphisms $f : X \to Y$ and $g : Y \to Z$, there exists a specified composite morphism $gf : X \to Z$;
- for any $f : X \to Y$, we have $\mathbb{1}_Y f = f = f \mathbb{1}_Y$;
- For any composable triple of morphisms f, g, h, we have $h(gf) = (hg)f$.

So, a category is a set of objects and maps (morphisms) between objects. For example, in the case of the category of topological spaces the objects are the topological spaces themselves, while the morphisms are the homeomorphisms. We can also have maps between categories, named functors.

Definition 2.10 (Functor)**.** Let $\mathcal{C}, \mathcal{D}$ be categories. A *functor* $F : \mathcal{C} \to \mathcal{D}$ consists of an object $F_C \in \mathcal{D}$ for any object $C \in \mathcal{C}$, and a morphism $Ff : F_C \to F_{C'} \in \mathcal{D}$ for each morphism $f : C \to C' \in \mathcal{C}$, satisfying the following two functoriality axioms

- For any composable pair $f, g \in \mathcal{C}$, we have $Fg \cdot Ff = F(g \cdot f)$;
- For each object $C \in \mathcal{C}$, $F(\mathbb{1}_C) = \mathbb{1}_{F_C}$

In other words, a functor maps categories (objects and morphisms) such that the structure of a category is preserved. A functor $F : C \to D$ is called *isomorphic* if there exists a functor $G : D \to C$ such that $FG = \mathbb{1}_D$ and $GF = \mathbb{1}_C$.

While both metric spaces and topological spaces form categories, metric spaces enjoy less functoriality than topological spaces. That is, there exists, in some way, ”less” functors from the category of metric spaces to other categories, than from the category of topological spaces to other spaces.

In particular, there does not exist a functor from the category of topological spaces to the category of metric spaces: while there exists a functor from the category of metric spaces to the category of topological spaces (by associating with each metric space its induced topological space), such a functor is never isomorphic [1, p.33].

Functoriality is something that is extremely useful in LEAN4, since it allows us to transfer properties between categories. This way, we only need one theorem in Mathlib for several different categories, instead of a separate theorem for each category. From this, it follows that for many of the theorems we use in topological spaces, such a theorem might not exist in Mathlib for a metric spaces. Since every metric space is a topological space, it is therefore usually easier to first work with the metric space as a topological space, and only later specify the metric when proving results that are specific to the metric space.

2.5 Is there anything LEAN4 cannot do?

After reading the previous sections, you might be wondering why so many proof assistants exist if one of them can seemingly do it all. There are however small differences between these languages, and they all have strengths and weaknesses. In this section, we will highlight some of the weaknesses of LEAN4.

2.5.1 Computation

One thing LEAN4 is bad at, is computations. In particularly, it is very slow, and terrible at symbolic computations. Computations in LEAN4 are made with the `#eval` command.

With its basis in dependent type theory, computations are based in function evaluation. While they are accurate and rigorous (as long as no sorry is present), their formation takes a long time, and LEAN4 is therefore not the preferred program to do computations with.

If `sorry` is present, `#eval` will not run at all. Even if the presence of `sorry` is indirect (perhaps in a lemma used for your theorem), LEAN4 still recognizes this and refuses to run `#eval`. The presence of `sorry` can lead to runtime instability and crashes. While this can be manually overrun, it is not recommended.

One instance where this difficulty with computation shows up, particularly in speed, is within the natural numbers. In LEAN4, the natural numbers are defined inductively through the *successor* function. That is, they are defined with base step `0`, and inductive step `succ n`, which is equal to $n + 1$. All arithmetic on natural numbers in LEAN4 is done with this definition at its core, meaning any time a natural number n appears anywhere, n function evaluations must be done. Especially as numbers get larger, this time adds up.

2.5.2 Homotopy type theory

Homotopy type theory (HoTT) is, simply said, an extension of the Calculus of Inductive Constructions, introduced in section 2.2.1 . While a precise definition of Homotopy Type Theory is beyond the scope of this thesis, more information on Homotopy Type Theory can be found in [17] and [21].

One of the main differences between HoTT and CIC, and the reason why LEAN4 does not support HoTT, is the notion of proof irrelevance. In LEAN4 (and CIC in general), a proof is simply a means to asserting the correctness of a certain statement. Two proofs of the same statement or proposition are considered to be judgmentally equal. That is, they carry no distinguishable information beyond the truth of the proposition itself. In HoTT , things work differently. There, proofs *can* carry information. Two proofs of the same statements might not be the same, because they take different paths to correctness. In HoTT, a true statement can have many distinct proofs, and studying how these proofs relate to each other is part of the theory itself.

LEAN actually did support HoTT in LEAN2. At the time, LEAN2 contained two different modes, one supporting proof irrelevant reasoning, and one HoTT mode, where proof irrelevance was not present. This second mode was eventually removed in LEAN3, since it had a tendency to give error messages and time-out messages that were not related to mistakes in the proof.

However, not all hope is lost on using HoTT in LEAN. For LEAN3, a library was made that supports HoTT in LEAN [11]. Similar projects have started for LEAN4, one of the most common of such being GroundZero. Some more abstract math needs the framework of HoTT to work and work efficiently. Projects such as GroundZero help bridge the gap between these abstract mathematical concepts and an easy to use proof assistant, increasing the potential user base for LEAN4.

3 PhysLean

3.1 What is PhysLean?

PhysLean, formerly known as HepLean, was started in 2024 by Joseph Tooby-Smith at Cornell university. Where Mathlib is a very developed library full of mathematics, PhysLean is striving to be the equivalent for physics and mathematical physics. Where PhysLean started with high-energy physics, in September 2025 it also contains topics such as quantum mechanics, condensed matter, and quantum field theory. PhysLean currently has over 20 active contributors and consists of nearly 100 000 lines of code, of which over 12 000 are in the mathematics folder.

3.2 Contributions

To show LEAN4 in action, as a part of this thesis we have made several contributions to the PhysLean library. Here, we will discuss these contributions and proofs.

3.2.1 Divergence of a product

The first theorem we formalize is called `divergence_smul`. The `smul` in this theorem name stands for scalar multiplication. It is a well-known fact from any multivariate calculus class that if $f : \mathbb{R}^m \to \mathbb{R}$ and $g : \mathbb{R}^m \to \mathbb{R}^m$, then div $(f \cdot g) = f \cdot$ div $(g) + \nabla f \cdot g$. Since we aim to get our theorems as general as possible, `divergence_smul` shows that this is actually true for any vector space E over any field $\mathbb{K}$, not just $\mathbb{R}^m$ over $\mathbb{R}$. The statement of the theorem is as follows:

Theorem 3.1. *Let $\Bbbk$ a field, and E a normed, additive, commutative group such that E is a finite-dimensional inner product space over $\Bbbk$. Let $f : E \to \Bbbk$ and $g : E \to E$ be functions differentiable at a point $x \in E$. Define $h(x) = f(x) \cdot g(x)$. Then,*

$$\text{div } h|_x = f(x) \cdot \text{div } g|_x + \langle \text{grad } f|_x, g(x) \rangle.$$

Here div $h = \text{tr } (Dh)$, where D is the usual derivative.

Before we start formalizing, it is good practice to know what is going on. To this extent, we write an informal proof first.

Proof. By a simple computation, we see that

$$\begin{aligned}
\text{div } h|_x &\overset{(1)}{=} \text{Tr } (Dh) \\
&\overset{(2)}{=} \text{Tr } (f(x) \cdot Dg|_x + g(x) \cdot Df|_x) \\
&\overset{(3)}{=} f(x) \cdot \text{Tr}(Dg|_x) + \text{Tr}(g(x) \cdot Df|_x) \\
&\overset{(4)}{=} f(x) \cdot \text{div } g|_x + \text{Tr}(g(x) \cdot Df|_x) \\
&\overset{(5)}{=} f(x) \cdot \text{div } g|_x + \text{Tr}(g(x) \cdot \text{grad } f|_x) \\
&\overset{(6)}{=} f(x) \cdot \text{div } g|_x + \langle \text{grad } f|_x, g(x) \rangle.
\end{aligned}$$

where (1) follows by definition of divergence, (2) follows by applying the chain rule to $h = f \cdot g$, (3) follows from linearity of the trace functions, and (4), (5), and (6) follow from the definitions of divergence, gradient, and the inner product, respectively. □

Before we go to the formalized version of this proof, we will note one thing. In the statement of Theorem 3.1, we talk about grad $f|_x$. In PhysLean, this theorem is actually slightly more general. Instead of the gradient, they use the adjoint of the Frechet derivative. In our case, this adjoint Frechet derivative simplifies to the gradient we all know, but this is why in the formalization we will be talking about `adjFDeriv` instead of `grad`.

Now, we are ready to take a look at the formalized version of this proof. Don't worry if it feels cryptic for now; the details will get explained later.

```
noncomputable section
open Module
open scoped InnerProductSpace

variable
{𝕜 : Type*} [RCLike 𝕜]
{E : Type*} [NormedAddCommGroup E] [NormedSpace 𝕜 E]
{F : Type*} [NormedAddCommGroup F] [NormedSpace 𝕜 F]

local notation "⟪" x ", " y "⟫" => inner 𝕜 x y

lemma divergence_smul [InnerProductSpace' 𝕜 E] {f : E → 𝕜} {g : E → E} {x : E}
        (hf : DifferentiableAt 𝕜 f x) (hg : DifferentiableAt 𝕜 g x)
        [FiniteDimensional 𝕜 E] :
        divergence 𝕜 (fun x => f x • g x) x
        = f x * divergence 𝕜 g x + ⟪adjFDeriv 𝕜 f x 1, g x⟫ := by
    unfold divergence
    simp [fderiv_fun_smul hf hg]
    obtain ⟨s, b⟩ := Basis.exists_basis 𝕜 E
    let basis := Classical.choice b
    have s_fin : Fintype s := FiniteDimensional.fintypeBasisIndex basis
    have h_basis : Basis (↑s) 𝕜 E = Basis s.toFinset 𝕜 E := by
        simp only [Set.mem_toFinset]
    rw [h_basis] at basis
    rw [LinearMap.trace_eq_matrix_trace_of_finset (s := s.toFinset) _ basis]
    simp only [Matrix.trace, Matrix.diag, LinearMap.toMatrix]
    simp_all only [Set.mem_toFinset, Finset.univ_eq_attach,
    LinearEquiv.trans_apply, LinearMap.toMatrix'_apply,
    LinearEquiv.arrowCongr_apply, Basis.equivFun_symm_apply, ite_smul,
    one_smul, zero_smul, Finset.sum_ite_eq',
    ↓reduceIte, ContinuousLinearMap.coe_coe,
    ContinuousLinearMap.smulRight_apply, map_smul,
    Basis.equivFun_apply, Pi.smul_apply, smul_eq_mul]
    -- comes from aesop
    rw [adjFDeriv]
    have h₁ : ⟪adjoint 𝕜 (⇑(fderiv 𝕜 f x)) 1, g x⟫ = (fderiv 𝕜 f x)  (g x):= by
        rw [HasAdjoint.adjoint_inner_left]
        · simp_all only [RCLike.inner_apply, map_one, mul_one]
        rfl
        · haveI : CompleteSpace E := FiniteDimensional.complete 𝕜 E
        apply hf.hasAdjFDerivAt.hasAdjoint_fderiv
    rw [h₁]
    have hg_sum : g x = ∑ x_1 ∈ s.toFinset.attach, (basis.repr (g x) x_1) • basis x_1
    := by
        exact Eq.symm (basis.sum_repr (g x))
    calc
        ∑ x_1 ∈ s.toFinset.attach, (fderiv 𝕜 f x) (basis x_1) * (basis.repr (g x)) x_1
        = ∑ x_1 ∈  s.toFinset.attach, (fderiv 𝕜 f x) ((basis.repr (g x) x_1) • basis x_1)
        := by
            refine Finset.sum_congr rfl (fun i hi => ?_)
            calc
                (fderiv 𝕜 f x) (basis i) * (basis.repr (g x) i) =
                (basis.repr (g x) i) * (fderiv 𝕜 f x) (basis i) := by
```

```
54                      exact mul_comm _ _
55                  _ = (fderiv 𝕜 f x) ((basis.repr (g x) i) • basis i) := by
56                      rw [map_smul]
57                      rfl
58                  _ = (fderiv 𝕜 f x) (∑ x_1 ∈ s.toFinset.attach, (basis.repr (g x) x_1)
59                      • basis x_1) := by
60                      rw [map_sum]
61                  _ = (fderiv 𝕜 f x) (g x) := by
62                      rw [hg_sum]
63                      apply congrArg (fderiv 𝕜 f x)
64                      simp only [← hg_sum]
```

So what is happening here? This really is almost a word-for-word translation into LEAN4 of our informal proof. First, we unfold our definition of divergence in line `17`. In lines `18` through `23`, we choose a basis so that we can work on the terms in their matrix form. In this way, we can obtain the inner product. We then use `aesop` in lines `27` through `34` to simplify our expression as much as possible. Finally in lines `46` through `64`, we use a `calc` tactic to manipulate our terms until we reach our goal. Let's dissect this approach.

lines 1 - 4

```
1  noncomputable section
2  open Module
3  open scoped InnerProductSpace
4
```

We first open a noncomputable section as described in Section 2.3. Next, we open the namespaces `Module` and `InnerProductSpace`. This is done to shorten our code a little bit by making all of the theorems in those namespaces immediately available. For example, the theorem `Module.two_smul` can be called upon by simply using `two_smul` when in the `Module` namespace. The `scoped` in front of `InnerProductSpace` means we cannot use everything from the `InnerProductSpace` namespace, but only certain theorems in it (namely, those labeled with `scoped`).

lines 5 - 11

```
5  variable
6  {𝕜 : Type*} [RCLike 𝕜]
7  {E : Type*} [NormedAddCommGroup E] [NormedSpace 𝕜 E]
8  {F : Type*} [NormedAddCommGroup F] [NormedSpace 𝕜 F]
9
10 local notation "⟪" x ", " y "⟫" => inner 𝕜 x y
11
```

Here, we define our variables. We define 𝕜 to be a field, and E and F to be normed additive commutative groups that are also a normed spaces over 𝕜. Furthermore, we introduce some notation: namely we denote by `⟪x, y⟫` the inner product between x and y over 𝕜.

lines 12 - 16

```
12 lemma divergence_smul [InnerProductSpace' 𝕜 E] {f : E → 𝕜} {g : E → E} {x : E}
13         (hf : DifferentiableAt 𝕜 f x) (hg : DifferentiableAt 𝕜 g x)
14         [FiniteDimensional 𝕜 E] :
15         divergence 𝕜 (fun x => f x • g x) x
16         = f x * divergence 𝕜 g x + ⟪adjFDeriv 𝕜 f x 1, g x⟫ := by
```

Then comes the problem statement of Theorem 3.1. After the theorem name, we see that E is an `InnerProductSpace'` over 𝕜 instead of a simple inner product space. `InnerProductSpace'` is defined as a generalization of an inner product space. While we do not need any of the freedom that this change brings, this change fits within the philosophy of LEAN4 to be as general as possible. Next, we define $f : E \to \Bbbk$, $g : E \to E$, and $x \in E$. Next, the hypotheses `hf` and `hg` state that f and g are differentiable at x over 𝕜, respectively.

Then comes the statement we want to prove. One notable thing here is that we write `adjFDeriv 𝕜 f x 1` instead of "simply" writing the gradient. This is due to how PhysLean decided to define the gradient, but these things are definitionally equal. Namely, `adjFDeriv` defines the adjoint of the Frechet derivative, and we have the identity that for linear maps,

```
gradient f x = f' 1 = adjFDeriv 𝕜 f x 1.
```

An advantage to using this adjoint Frechet derivative over the "normal" gradient is to be able to apply compositions theorems; this again aligns with the LEAN4 philosophy of generality.

lines 17 - 18

```
17      unfold divergence
18      simp [fderiv_fun_smul hf hg]
```

Here the actual proof begins. We start by unfolding the definition of divergence in line 17, which states that `divergence` is equal to `(fderiv 𝕜 f x).toLinearMap.trace`. This indeed makes sense, as the divergence can be seen as the trace of the Jacobian. After that, we use `simp` with the theorem `fderiv_fun_smul hf hg` in line 18, which in essence applies the product rule to the case of scalar multiplication, that is, it states

```
fderiv 𝕜 (fun y ⇒ c y • f y) x = c x • fderiv 𝕜 f x + (fderiv 𝕜 c x).smulRight (f x)
```

lines 19 - 24

```
19      obtain ⟨s, b⟩ := Basis.exists_basis 𝕜 E
20      let basis := Classical.choice b
21      have s_fin : Fintype s := FiniteDimensional.fintypeBasisIndex basis
22      have h_basis : Basis (↑s) 𝕜 E = Basis s.toFinset 𝕜 E := by
23          simp only [Set.mem_toFinset]
24      rw [h_basis] at basis
```

Next, we want to choose a basis. We did this, as working with the trace of a matrix is easier than working with the trace of a linear map, at least on paper. Line 19 gives us the existence of a basis `b` with indexing set `s`. Here, technically `b` is the hypothesis that a basis with indexing set `s` exists. We however choose the label `b` instead of `hb`, as we can work with `b` as if it contains the basis itself at a later time.

We then use the axiom of choice to pick a specific basis called `basis`. While we are not working with a set of sets, and thus maybe the use of the axiom of choice seems unnecessary here, this is a peculiarity of Mathlib; it forces us to use the axiom of choice both for singular sets and for sets of sets .

After, we create a hypothesis `s_fin`, stating that the indexing set `s` is finite; this is true, since we are working in a finite-dimensional vector space. Using `s_fin`, we can then also say that our basis is finite, and we do so in the hypothesis `h_basis`. This is a fact we will need later on to convert from linear maps to matrices. The final step we take in line 24 is that we rewrite `h_basis` at our earlier basis hypothesis. This way we have a combined finite basis with a finite indexing set.

lines 25 - 26

```
25      rw [LinearMap.trace_eq_matrix_trace_of_finset (s := s.toFinset) _ basis]
26      simp only [Matrix.trace, Matrix.diag, LinearMap.toMatrix]
```

Now that the basis is picked, in line 25 we are able to rewrite our traces of linear maps into traces of a matrix for easier manipulation. This line was actually quite tricky, as LEAN4 wanted the basis in a very specific form for this theorem to work. After, in line 26, we use some simple facts about matrices, linear maps, and the trace to simplify our goal. Respectively, we use the fact that the trace is an element of `E`, we define what the diagonal elements of a matrix are exactly, and how exactly to go from a linear map to a matrix.

lines 27 - 34

```
27      simp_all only [Set.mem_toFinset, Finset.univ_eq_attach,
28      LinearEquiv.trans_apply, LinearMap.toMatrix'_apply,
29      LinearEquiv.arrowCongr_apply, Basis.equivFun_symm_apply, ite_smul,
30      one_smul, zero_smul, Finset.sum_ite_eq',
31      ↓reduceIte, ContinuousLinearMap.coe_coe,
32      ContinuousLinearMap.smulRight_apply, map_smul,
33      Basis.equivFun_apply, Pi.smul_apply, smul_eq_mul]
34      -- comes from aesop
```

Next, we needed to apply a lot of arithmetic manipulations to our goal. To avoid doing most of this by hand, we employed `aesop`. Since it is not good practice to leave non-terminal (meaning they do not fully solve the goal) automation in the proof, we then asked `aesop` what it did, using `aesop?`, and copied the result of that into our proof. This also highlights the power of `simp`; while we could split up this `simp` into several lines, we would likely end up repeating ourselves, since there is a fair chance these theorems are all used several times in the simplification. This is part of where the real power of this automation comes in; we are able to do in seven lines what manually very well could have taken over 20.

line 35

```
35      rw [adjFDeriv]
```

Next, we simply rewrote the definition of `adjFDeriv` into our proof, namely it replaces the instance of `adjFDeriv` by its definition as the adjoint Frechet derivative.

lines 36 - 41

```
36      have h₁ : ⟪adjoint 𝕜 (⇑(fderiv 𝕜 f x)) 1, g x⟫ = (fderiv 𝕜 f x)  (g x):= by
37          rw [HasAdjoint.adjoint_inner_left]
38          · simp_all only [RCLike.inner_apply, map_one, mul_one]
39          rfl
40          · haveI : CompleteSpace E := FiniteDimensional.complete 𝕜 E
41          apply hf.hasAdjFDerivAt.hasAdjoint_fderiv
```

In these lines, we introduce a new hypothesis involving the inner product. Since we are working with an inner product on a vector space, the inner product can be seen as a dot product. In this hypothesis, we show that that is indeed true. We first want to eliminate the adjoint, which we can quite easily do in an inner product. That is, the adjoint `f'` of `f` is defined through `⟪f' y, x⟫ = ⟪y, f x⟫`. This creates us two goals; in one goal, it wants us to show our hypothesis statement to be true, while in the second goal, it wants us to define which function we take the adjoint of.

- For the first goal, it wants us to show that `⟪1, ?m.128 (g x)⟫ = (fderiv 𝕜 f x) (g x)`, where in this case `?m.128` denotes our to be specified function. We then simplify using `RCLike.inner_apply`, which states that under certain conditions, `⟪x,y⟫ = y * x`, `map_one`, which states that for any linear map, `f 1 = 1`, and `mul_one`, which states that `x * 1 = 1`. We end up with a definitional equality and close the subgoal with `rfl`.
- For the second goal, it wants us to prove that `fderiv 𝕜 f x` has an adjoint, and that that adjoint is given by `adjoint 𝕜 ⇑(fderiv 𝕜 f x)`. We are in luck, because the theorem `hasAdjFDerivAt.hasAdjoint_fderiv` says this is true for any differentiable function in a complete vector space. Since any finite vector space is complete, we make this explicit with `haveI : CompleteSpace E := FiniteDimensional.complete 𝕜 E`. Finally, to tell LEAN4 that `f` is differentiable (which is given in our assumption `hf`), we use dot notation, so we simply add `hf` in front of the theorem that requires it. This then closes the goal.

lines 42 - 45

```
42    rw [h₁]
43    have hg_sum : g x = ∑ x_1 ∈ s.toFinset.attach, (basis.repr (g x) x_1) • basis x_1
44    := by
45       exact Eq.symm (basis.sum_repr (g x))
```

Now we rewrite the hypothesis we just proved in our goal, and we show that for a linear map `g` we can express this as a sum of the product of a basis vector and some element from our field. This is almost exactly what the theorem `basis.sum_repr (g x)`, but it says it the other way around, i.e. it states that

```
∑ x_1 ∈ s.toFinset.attach, (basis.repr (g x) x_1) • basis x_1 = g x,
```

and thus we use `Eq.symm` to take the left-hand side to the right-hand side and vice versa.

lines 46 - 50

```
46    calc
47       ∑ x_1 ∈ s.toFinset.attach, (fderiv 𝕜 f x) (basis x_1) * (basis.repr (g x)) x_1
48       = ∑ x_1 ∈  s.toFinset.attach, (fderiv 𝕜 f x) ((basis.repr (g x) x_1) • basis x_1)
49        := by
50          refine Finset.sum_congr rfl (fun i hi ⇒ ?_)
```

We are getting close to the end of our proof! We now open up a `calc` block for our last part. In this first equality, we simply change from • to ∗, making our next computations easier. We note that line 50 contains some notation we have not seen yet, namely `?_`. This notation means that we want something to go there, but that we would like LEAN4 to figure out what that something is. Sometimes LEAN4 does not know what that something is, in which case it would create a new subgoal. However, in this case it could figure it out, so we can leave the `?_` for brevity of the proof.

lines 51 - 64

```
            calc
                (fderiv 𝕜 f x) (basis i) * (basis.repr (g x) i) =
                (basis.repr (g x) i) * (fderiv 𝕜 f x) (basis i) := by
                    exact mul_comm _ _
                _ = (fderiv 𝕜 f x) ((basis.repr (g x) i) • basis i) := by
                    rw [map_smul]
                    rfl
                _ = (fderiv 𝕜 f x) (∑ x_1 ∈ s.toFinset.attach, (basis.repr (g x) x_1)
                    • basis x_1) := by
                    rw [map_sum]
                _ = (fderiv 𝕜 f x) (g x) := by
                    rw [hg_sum]
                    apply congrArg (fderiv 𝕜 f x)
                    simp only [← hg_sum]
```

We now nest another `calc` block in our first; this way, we can manipulate each element of the sum seperately, instead of having to constantly write the entire sum. In this calc block, we show that

```
(fderiv 𝕜 f x) (basis i) * (basis.repr (g x) i) = (fderiv 𝕜 f x) (g x)
```

through a fairly simple computation, using things such as commutativity of multiplication (`mul_comm`), scalar multiplication of a linear map (`map_smul`), and some hypotheses we stated earlier.

With the computation complete, we have proven our goal!

Before we were able to merge this proof into the PhysLean library, a user by the name of Komyyy finished a different, shorter, proof of this theorem. The proof that was submitted to PhysLean by Komyyy goes as follows:

```
noncomputable section
open Module
open scoped InnerProductSpace

variable
{𝕜 : Type*} [RCLike 𝕜]
{E : Type*} [NormedAddCommGroup E] [NormedSpace 𝕜 E]
{F : Type*} [NormedAddCommGroup F] [NormedSpace 𝕜 F]

local notation "⟪" x ", " y "⟫" => inner 𝕜 x y

lemma divergence_smul [InnerProductSpace' 𝕜 E] {f : E → 𝕜} {g : E → E} {x : E}
        (hf : DifferentiableAt 𝕜 f x) (hg : DifferentiableAt 𝕜 g x)
        [FiniteDimensional 𝕜 E] :
        divergence 𝕜 (fun x => f x • g x) x
        = f x * divergence 𝕜 g x + ⟪adjFDeriv 𝕜 f x 1, g x⟫_𝕜 := by
    haveI : CompleteSpace E := FiniteDimensional.complete 𝕜 E
    simp [divergence, fderiv_fun_smul hf hg,
          hf.hasAdjFDerivAt.hasAdjoint_fderiv.adjoint_inner_left]
```

So, how was Komyyy able to condense my 44 lines of code down to 3? Note that Komyyy used only theorems and definitions that were also in my proof. The biggest difference is that Komyyy realized early on that `E` was a complete space. `simp` was therefore able to work entirely more efficiently, since it could access so many new theorems, as well as the theorem `hf.hasAdjFDerivAt.hasAdjoint_fderiv.adjoint_inner_left`.

Here the real power of LEAN4 and Mathlib is exhibited; by using the appropriate theorems, LEAN4 can fill in many of the gaps that can be simplified from the hypotheses. Being able to exploit LEAN4 in such a way requires extensive familiarity with the library, as well as practice on efficiently searching through it.

This is not to say that one proof is better or worse than the other. The first proof is more explicit, and easier to follow as a novice. At the same time, it is longer and thus will take more memory and runtime. This might not seem like a huge deal for one theorem; however if this is the case for all theorems and statements in a library, these issues pile up and compound very quickly.

They however both get to the same goal. It is a matter of style and objective. Many theorems have several proofs that are available, they all add some sort of insight, even if not all are regularly used. One example in informal mathematics of this same phenomenon is the different proofs available of the Pythagorean theorem. There are proofs of this theorem rooted in fields from geometry to algebra, visual proofs to completely theoretical proofs, existence proofs and constructive proofs, and everything in between.

There has even been been an active discussion within the Mathlib community over what should take precedent, efficiency or readability. Several plans have been made to do both; to have an efficient version, as well as a readable version in the Mathlib library. This could be great for pedagogical purposes; readable proofs will be far less intimidating looking to most novices than the very efficient proofs. However, concerns about upkeep have also been noted. If this were to happen, it would mean the amount of work maintaining the library will effectively double. With the growth rate of the library as it, this brings concerns about sustainability for the Mathlib maintainers.

After this happened, I wanted to go back to my own code and compare the exact differences. However, it was no longer working. Upon further inspection, it turned out some of the theorems we were using had changed, meaning some of our code had become unnecessary. A total of 35 lines of code could be deleted, while the rest did not even need to be edited. The shortened code is as follows.

```
lemma divergence_smul [InnerProductSpace' 𝕜 E] {f : E → 𝕜} {g : E → E} {x : E}
    (hf : DifferentiableAt 𝕜 f x) (hg : DifferentiableAt 𝕜 g x)
    [FiniteDimensional 𝕜 E] :
    divergence 𝕜 (fun x ⇒ f x • g x) x
    = f x * divergence 𝕜 g x + ⟪adjFDeriv 𝕜 f x 1, g x⟫ := by
  unfold divergence
  simp [fderiv_fun_smul hf hg]
  rw [adjFDeriv]
  have h₁ : ⟪adjoint 𝕜 (⇑(fderiv 𝕜 f x)) 1, g x⟫ = (fderiv 𝕜 f x)  (g x):= by
    rw [HasAdjoint.adjoint_inner_left]
    · simp_all only [RCLike.inner_apply, map_one, mul_one]
      rfl
    · haveI : CompleteSpace E := FiniteDimensional.complete 𝕜 E
      apply hf.hasAdjFDerivAt.hasAdjoint_fderiv
  rw [h₁]
```

It was weird to me that a previously checked proof could suddenly be ”wrong”. The main goal of proof assistants is to guarantee absolute correctness. It felt like a contradiction for my previously correct code to suddenly not be correct, so I started to investigate; what made it so that the proof could be shortened?

The major thing that changed, was something to do with the step `simp [fderiv_fun_smul hf hg]`. Where in the original proof using Mathlib version 4.23, we ended up with an expression for the trace, after updating to Mathlib version 4.24, we up with the goal

```
(fderiv 𝕜 f x) (g x) = ⟪adjFDeriv 𝕜 f x 1, g x⟫
```

This avoids any need for a basis, since we are no longer working with the trace and thus don't have to convert to a matrix. Furthermore, the leftover goal is exactly what our hypothesis `h₁` states. This also makes the entire `calc` block that came after unnecessary.

So *why* did this happen? The obvious thing would be if the theorem `fderiv_fun_smul` within Mathlib changed, since this is the last step that stayed the same when comparing the two proofs. However, looking at commit history, this seems to not have happened. It would also be quite strange if that was the case, since `fderiv_fun_smul` does not concern itself with the trace.

The other option is for the change to be within `simp`. As explained in section 2.3, `simp` relies on a database of theorems it uses to simplify expressions. It seems most likely that some theorems concerning the trace were included in this database after we had written the first proof. However, this is something way harder to check, since this database is not centralized; instead attributes are tagged, and we are unsure which attributes to check.

One thing that does become clear however is that due to the fast evolving nature of Mathlib and LEAN4, it is important to continually check the correctness of your proofs that are not yet merged into a project. Furthermore, it is important to submit your contributions as soon as you can. On the one hand it avoids duplicate work when you have a perfectly valid proof ready to go, and reduces the risk that others who are unaware of your work, also submit a proof in the meantime. On the other hand, once your work becomes part of a project, keeping up with your dependencies changing is not your responsibility anymore; instead, it becomes the responsibility of the maintainer(s) of the project that you merged into.

3.2.2 Zero-velocity condition for the trajectory of a harmonic oscillator

The second theorem we formalized is called `trajectory_velocity_eq_zero_iff`. This is a statement regarding the trajectory of a harmonic oscillator. The statement we want to formalize goes as follows.

Theorem 3.2. Let S be a harmonic oscillator described by a mass m and a constant k whose trajectory is given by $x(t)$. Define $\dot{x}(t) = \frac{\partial x}{\partial t}(t)$, and let S have initial conditions $x(0) = x_0$ and $\dot{x}(0) = v_0$. Then

$$\dot{x}(t) = 0 \text{ if and only if } ||x(t)|| = \sqrt{||x_0||^2 + \left(\frac{||v_0||^2}{\omega^2}\right)^2},$$

where $\omega^2 = \frac{k}{m}$.

Again, before we start formalizing, it is good to write an informal proof. Both our informal and formalization will be based on the conservation of energy. To this extent, we will first state the energy equation for a harmonic oscillator and its conservation property.

Theorem 3.3. [15, p.14, 59] The energy for a harmonic oscillator is given by

$$E(t) = \frac{1}{2}m\dot{x}^2(t) + \frac{1}{2}kx^2(t), \tag{1}$$

where $\frac{1}{2}m\dot{x}^2(t)$ represents the kinetic energy, while $\frac{1}{2}kx^2(t)$ represents the potential energy. Furthermore, the energy is conserved. That is, for any times t_1 and t_2, $E(t_1) = E(t_2)$.

Again, before we start formalizing, let us prove Theorem 3.2 informally.

Proof. ($\Rightarrow$) Suppose $\dot{x}(t) = 0$. Then, $\dot{x}(t) = 0$. Filling this in to our energy equation (1), we see that $E(t) = \frac{1}{2}kx^2(t)$. However, by conservation of energy, also $E(t) = E(0) = \frac{1}{2}mv_0^2 + \frac{1}{2}kx_0^2$. We can then compute

$$\begin{aligned}
E(t) &= E(0) \\
\frac{1}{2}kx^2(t) &= \frac{1}{2}mv_0^2 + \frac{1}{2}kx_0^2 \\
x^2(t) &= \frac{\frac{1}{2}mv_0^2 + \frac{1}{2}kx_0^2}{\frac{1}{2}k} \\
&= \frac{v_0^2}{\omega^2} + x_0^2
\end{aligned}$$

Taking norms on either side gives the desired result.

($\Leftarrow$) Suppose $||x(t)|| = \sqrt{||x_0||^2 + \left(\frac{||v_0||^2}{\omega^2}\right)^2}$. Since here we are working with real numbers, we can state without loss of generality that

$$x(t) = \sqrt{x_0^2 + \left(\frac{v_0^2}{\omega^2}\right)^2}. \tag{2}$$

Furthermore, as before, we know that $E(t) = \frac{1}{2}m\dot{x}^2(t) + \frac{1}{2}kx^2(t) = \frac{1}{2}mv_0^2 + \frac{1}{2}kx_0^2$. Setting them equal and substituting in Equation (2), we get

$$\begin{aligned}
\frac{1}{2}m\dot{x}^2(t) + \frac{1}{2}kx^2(t) &= \frac{1}{2}mv_0^2 + \frac{1}{2}kx_0^2 \\
\frac{1}{2}m\dot{x}^2(t) + \frac{1}{2}k\left(\sqrt{x_0^2 + \left(\frac{v_0}{\omega}\right)^2}.\right)^2 &= \frac{1}{2}mv_0^2 + \frac{1}{2}kx_0^2 \\
\frac{1}{2}m\dot{x}^2(t) + \frac{1}{2}kx_0^2 + \frac{1}{2}k\frac{v_0^2}{\omega^2} &= \frac{1}{2}mv_0^2 + \frac{1}{2}kx_0^2 \\
\frac{1}{2}m\dot{x}^2(t) &= \frac{1}{2}mv_0^2 - \frac{1}{2}k\frac{v_0^2}{\omega^2} \\
\dot{x}^2(t) &= v_0^2 - \frac{k}{m}\frac{v_0^2}{\omega^2} \\
&= v_0^2 - \omega^2\frac{v_0^2}{\omega^2} = 0.
\end{aligned}$$

Since $\dot{x}^2(t) = 0$, also $\dot{x}(t) = 0$. $\square$

Now let us look at my formalization. It was merged into the PhysLean library on October 23rd, 2025.

```
variable (S : HarmonicOscillator)

lemma trajectory_velocity_eq_zero_iff (IC : InitialConditions) (t : Time) :
    ∂ₜ (IC.trajectory S) t = 0 ↔
    ‖(IC.trajectory S) t‖ = √(‖IC.x₀‖^2 + (‖IC.v₀‖/S.ω)^2) := by
  have := by exact energy_eq S (trajectory S IC)
  have h_energy_t := congrFun this t
  simp [kineticEnergy_eq, potentialEnergy_eq] at h_energy_t
  rw [real_inner_self_eq_norm_sq (trajectory S IC t)] at h_energy_t
  have := by exact trajectory_energy S IC
  have h_init := congrFun this t
  have h_ω := by exact ω_sq S
  constructor
  · intro h_partial
    rw [h_partial, inner_zero_left, mul_zero, zero_add] at h_energy_t
    have h₁ : ‖trajectory S IC t‖ ^ 2 = S.energy (trajectory S IC) t * 2 * (1 / S.k)
    := by
      simp [h_energy_t]
      field_simp
    symm
    refine (sqrt_eq_iff_mul_self_eq ?_ ?_).mpr ?_
    · apply add_nonneg <;> apply sq_nonneg
    · apply norm_nonneg
    rw [← pow_two]
    rw [h₁, h_init]
    ring_nf
    rw [mul_assoc]
    rw [mul_inv_cancel₀]
    · rw [mul_one, inv_eq_one_div S.k, mul_assoc]
      rw [mul_one_div S.m S.k, ← inverse_ω_sq]
      ring
    · exact k_neq_zero S
  · intro h_norm
    apply norm_eq_zero.mp
    rw [real_inner_self_eq_norm_sq (∂ₜ (trajectory S IC) t)] at h_energy_t
    have energies : S.energy (trajectory S IC) t = S.energy (trajectory S IC) t
    := by rfl
    nth_rewrite 1 [h_energy_t] at energies
    nth_rewrite 1 [h_init] at energies
    rw [h_norm] at energies
    have h₁ : S.m * ‖∂ₜ (trajectory S IC) t‖ ^ 2
            + S.k * (√(‖IC.x₀‖ ^ 2 + (‖IC.v₀‖ / S.ω) ^ 2) ^ 2)
            = S.m * ‖IC.v₀‖ ^ 2 + S.k * ‖IC.x₀‖ ^ 2 := by
      calc
        S.m * ‖∂ₜ (trajectory S IC) t‖ ^ 2
        + S.k * (√(‖IC.x₀‖ ^ 2 + (‖IC.v₀‖ / S.ω) ^ 2) ^ 2)
            = 2 * (2⁻¹ * S.m * ‖∂ₜ (trajectory S IC) t‖ ^ 2
            + 2⁻¹ * (S.k * √(‖IC.x₀‖ ^ 2 + (‖IC.v₀‖ / S.ω) ^ 2) ^ 2)) := by
          simp [mul_add]
          rw [← mul_assoc, ← mul_assoc]
          rw [mul_inv_cancel_of_invertible 2, one_mul]
      _ = 2 * (1 / 2 * (S.m * ‖IC.v₀‖ ^ 2 + S.k * ‖IC.x₀‖ ^ 2)) := by rw [energies]
      _ = S.m * ‖IC.v₀‖ ^ 2 + S.k * ‖IC.x₀‖ ^ 2 := by simp
    have h₂ : S.m * ‖∂ₜ (trajectory S IC) t‖ ^ 2 + S.k * (‖IC.x₀‖ ^ 2 + (‖IC.v₀‖ / S.ω) ^ 2)
        = S.m * ‖IC.v₀‖ ^ 2 + S.k * ‖IC.x₀‖ ^ 2 := by
      rw [← h₁, sq_sqrt ?_]
      apply add_nonneg
      apply sq_nonneg
      apply sq_nonneg
    have h₃: ‖∂ₜ (trajectory S IC) t‖ ^ 2 = ‖IC.v₀‖ ^ 2 - (S.k / S.m) * (‖IC.v₀‖ / S.ω) ^ 2
    := by
      calc
        ‖∂ₜ (trajectory S IC) t‖ ^ 2 = (1 / S.m) * (S.m * ‖∂ₜ (trajectory S IC) t‖ ^ 2
64        + S.k * (‖IC.x₀‖ ^ 2 + (‖IC.v₀‖ / S.ω) ^ 2) - S.k * (‖IC.x₀‖ ^ 2
65        + (‖IC.v₀‖ / S.ω) ^ 2)) := by simp
66        _ = (1 / S.m) * (S.m * ‖IC.v₀‖ ^ 2 + S.k * ‖IC.x₀‖ ^ 2
67          - S.k * (‖IC.x₀‖ ^ 2 + (‖IC.v₀‖ / S.ω) ^ 2)) := by rw [h₂]
68        _ = (1 / S.m) * (S.m * ‖IC.v₀‖ ^ 2 + S.k * ‖IC.x₀‖ ^ 2
69          - S.k * ‖IC.x₀‖ ^ 2 - S.k * (‖IC.v₀‖ / S.ω) ^ 2) := by
70          rw [mul_add S.k (‖IC.x₀‖ ^ 2) ((‖IC.v₀‖ /S.ω) ^2)]
71          rw [←sub_sub_sub_eq (S.m * ‖IC.v₀‖ ^ 2) (S.k * ‖IC.x₀‖ ^ 2)
72          (S.k * (‖IC.v₀‖ / S.ω) ^ 2) (S.k * ‖IC.x₀‖ ^ 2)]
73          simp only [one_div, sub_sub_sub_cancel_right, add_sub_cancel_right]
74        _ = (1 / S.m) * (S.m * ‖IC.v₀‖ ^ 2 - S.k * (‖IC.v₀‖ / S.ω) ^ 2) := by simp
75        _ = (1 / S.m) * (S.m * ‖IC.v₀‖ ^ 2) - (1 / S.m) * (S.k * (‖IC.v₀‖ / S.ω) ^ 2) := by
76          rw [mul_sub (1 / S.m) (S.m * ‖IC.v₀‖ ^ 2) (S.k * (‖IC.v₀‖ / S.ω) ^ 2)]
77        _ = ‖IC.v₀‖ ^ 2 - (S.k / S.m) * (‖IC.v₀‖ / S.ω) ^ 2 := by
78          simp only [one_div, ne_eq, m_neq_zero, not_false_eq_true, inv_mul_cancel_left₀,
79            sub_right_inj]
80          rw [← mul_assoc, inv_mul_eq_div S.m S.k]
81    rw [← ω_sq, div_pow ‖IC.v₀‖ S.ω 2] at h₃
82    rw [mul_div_cancel₀ (‖IC.v₀‖ ^ 2) ?_] at h₃
83    rw [sub_self (‖IC.v₀‖ ^ 2)] at h₃
84    rw [sq_eq_zero_iff] at h₃
85    exact h₃
86    rw [pow_ne_zero_iff ?_]
87    apply ω_neq_zero
88    exact Ne.symm (Nat.zero_ne_add_one 1)
```

With the goal to let the reader become more familiar with LEAN4, let us dissect this proof as well. As in `divergence_smul`, we cut up the code into smaller pieces (albeit some larger than others) and analyze each piece to see what is happening.

lines 1 - 5

```
1  variable (S : HarmonicOscillator)
2
3  lemma trajectory_velocity_eq_zero_iff (IC : InitialConditions) (t : Time) :
4      ∂ₜ (IC.trajectory S) t = 0 ↔
5      ‖(IC.trajectory S) t‖ = √(‖IC.x₀‖^2 + (‖IC.v₀‖/S.ω)^2) := by
```

We start with the problem statement. We have initial conditions `IC.x₀` and `IC.v₀` and a time `t`. Then, we want to prove that the velocity of the trajectory `∂ₜ (IC.trajectory S) t` is equal to zero if and only if `‖(IC.trajectory S) t‖ = √(‖IC.x₀‖^2 + (‖IC.v₀‖/S.ω)^2)`. Thus, this is exactly the statement as in Theorem 3.2.

lines 6 - 12

```
6    have := by exact energy_eq S (trajectory S IC)
7    have h_energy_t := congrFun this t
8    simp [kineticEnergy_eq, potentialEnergy_eq] at h_energy_t
9    rw [real_inner_self_eq_norm_sq (trajectory S IC t)] at h_energy_t
10   have := by exact trajectory_energy S IC
11   have h_init := congrFun this t
12   have h_ω := by exact ω_sq S
```

We now define a few key concepts that we will need in both directions (the ”if” direction and the ”only if” direction) of our proof. In line 6, we obtain that the total energy of our system is the sum of the kinetic and potential energy as in Theorem 3.3. In line 7, we then specify this energy at our time `t`. In line 8, we rewrite the definitions for the kinetic and potential energy.

Since these are defined with norms, in line 9 we rewrite the inner product to a norm, and we end up with

```
h_energy_t : S.energy (trajectory S IC) t
             = 2⁻¹ * S.m * inner ℝ (∂ₜ (trajectory S IC) t) (∂ₜ (trajectory S IC) t)
             + 2⁻¹ * (S.k * ‖trajectory S IC t‖ ^ 2).
```

In lines 10 and 11 we do something similar, but with the energy at the initial condition to obtain

```
S.energy (trajectory S IC) t = 1 / 2 * (S.m * ‖IC.v₀‖ ^ 2 + S.k * ‖IC.x₀‖ ^ 2).
```

Finally, in line 12, we state that `S.ω ^ 2 = S.k / S.m`.

line 13

```
13   constructor
```

Now that we have some basic facts covered, we split our "if and only if" proof into an "if" proof and an "only if" proof. Our first goal becomes

```
∂ₜ (trajectory S IC) t = 0 → ‖trajectory S IC t‖ = √(‖IC.x₀‖ ^ 2 + (‖IC.v₀‖ / S.ω) ^ 2),
```

while our second goal becomes

```
‖trajectory S IC t‖ = √(‖IC.x₀‖ ^ 2 + (‖IC.v₀‖ / S.ω) ^ 2) → ∂ₜ (trajectory S IC) t = 0.
```

lines 14 - 15

```
14   · intro h_partial
15     rw [h_partial, inner_zero_left, mul_zero, zero_add] at h_energy_t
```

We now start with our first goal, the "if" case. We intro the antecedent (the part before the implication) with `h_partial`, and then subsitute this into our energy equation `h_energy_t`. We are left with

```
h_energy_t : S.energy (trajectory S IC) t = 2⁻¹ * S.m * inner ℝ 0 0
                                          + 2⁻¹ * (S.k * ‖trajectory S IC t‖ ^ 2)
```

Since we now have an inner product between 0 and 0 in `h_energy_t`, we can simplify and get rid of the inner product term all together.

lines 16 - 18

```
16     have h₁ : ‖trajectory S IC t‖ ^ 2 = S.energy (trajectory S IC) t * 2 * (1 / S.k)
17     := by
18       simp [h_energy_t]
19       field_simp
```

We now use our energy equation to rewrite the norm of the trajectory. This sets us up to apply energy conservation.

lines 20 - 24

```
20    symm
21    refine (sqrt_eq_iff_mul_self_eq ?_ ?_).mpr ?_
22    · apply add_nonneg <;> apply sq_nonneg
23    · apply norm_nonneg
24    rw [← pow_two]
```

Here, we remove the square root from our goal. After the `refine` in line 21, we end up with two goals. In the first bulletpoint we prove the first goal,

```
0 ≤ ‖IC.x₀‖ ^ 2 + (‖IC.v₀‖ / S.ω) ^ 2.
```

After applying `add_nonneg` in line 22 two new goals appear, namely that `0 ≤ ‖IC.x₀‖ ^ 2`, and that `0 ≤ (‖IC.v₀‖ / S.ω) ^ 2`. Since both of these goals have the same proof, namely `apply sq_nonneg`, we opt to not make new bulletpoints for these goals. Instead, we use the syntax `<;>` to tell LEAN4 to use `apply sq_nonneg` in both goals. In the second bulletpoint we give a proof of the second goal,

```
0 ≤ ‖trajectory S IC t‖.
```

After all these lines, we end up with the goal

```
‖IC.x₀‖ ^ 2 + (‖IC.v₀‖ / S.ω) ^ 2 = ‖trajectory S IC t‖ ^ 2.
```

line 25

```
25    rw [h₁, h_init]
```

Now we actually apply energy conservation. When we rewrite `h₁`, we replace

```
‖trajectory S IC t‖ ^ 2
```

with

```
S.energy (trajectory S IC) t * 2 * (1 / S.k),
```

which then gets replaced to

```
1 / 2 * (S.m * ‖IC.v₀‖ ^ 2 + S.k * ‖IC.x₀‖ ^ 2) * 2 * (1 / S.k)
```

by `h_init`.

lines 26 - 32

```
26    ring_nf
27    rw [mul_assoc]
28    rw [mul_inv_cancel₀]
29    · rw [mul_one, inv_eq_one_div S.k, mul_assoc]
30      rw [mul_one_div S.m S.k, ← inverse_ω_sq]
31      ring
32    · exact k_neq_zero S
```

Now all we need to do is some arithmetic to show that indeed

```
‖IC.x₀‖ ^ 2 + (‖IC.v₀‖ / S.ω) ^ 2
= 1 / 2 * (S.m * ‖IC.v₀‖ ^ 2 + S.k * ‖IC.x₀‖ ^ 2) * 2 * (1 / S.k)
```

This is what the code above does. The curious reader might be wondering why `mul_inv_cancel₀` gives us two goals. The first goal has us complete the actual arithmetic we need to do. The second goal asks us to assert that `S.k ≠ 0` , which is needed for the theorem `mul_inv_cancel₀` to actually be applied.

lines 33 - 34

```
33    · intro h_norm
34      apply norm_eq_zero.mp
```

We move on to the ”only if” part of the proof. We again first introduce the antecedent, and we call it `h_norm`. We are then left with the goal `∂ₜ (trajectory S IC) t = 0`. From the start of our proof we had an idea that we would end up with the norm of our goal, and we know that $||x|| = 0$ if and only if $x = 0$. Thus, we already rewrite our goal to include that norm.

lines 35 - 40

```
35      rw [real_inner_self_eq_norm_sq (∂ₜ (trajectory S IC) t)] at h_energy_t
36      have energies : S.energy (trajectory S IC) t = S.energy (trajectory S IC) t
37      := by rfl
38      nth_rewrite 1 [h_energy_t] at energies
39      nth_rewrite 1 [h_init] at energies
40      rw [h_norm] at energies
```

We then again convert the norms to inner products. Next, in line 36, we state the conservation of energy. In line 38, we rewrite the first instance of energy to the general energy equation, while in line 39 we rewrite the (now also first, but the original second) instance to the energy at the initial conditions. Finally in this block, we use our antecedent to obtain

```
      2⁻¹ * S.m * ‖∂ₜ (trajectory S IC) t‖ ^ 2
      + 2⁻¹ * (S.k * √(‖IC.x₀‖ ^ 2 + (‖IC.v₀‖ / S.ω) ^ 2) ^ 2)
  =   1 / 2 * (S.m * ‖IC.v₀‖ ^ 2 + S.k * ‖IC.x₀‖ ^ 2)
```

at our hypothesis of conservation of energy.

So why do we start from $E(t) = E(t)$, and don't keep manipulating the goal as we have done prior? Simply said, it is less work. This entire proof could very well work ”in reverse”, manipulating the goal until equality of energies is obtained. However, every time you want to introduce a new term, you need to be explicit in what this term should look like. By removing terms as we are doing now, we can let LEAN4 fill in more of the gaps, and the code will be shorter and more readable as a result.

lines 41 - 52

```
41      have h₁ : S.m * ‖∂ₜ (trajectory S IC) t‖ ^ 2
42      + S.k * (√(‖IC.x₀‖ ^ 2 + (‖IC.v₀‖ / S.ω) ^ 2) ^ 2)
43              = S.m * ‖IC.v₀‖ ^ 2 + S.k * ‖IC.x₀‖ ^ 2 := by
44        calc
45          S.m * ‖∂ₜ (trajectory S IC) t‖ ^ 2 + S.k * (√(‖IC.x₀‖ ^ 2 + (‖IC.v₀‖ / S.ω) ^ 2) ^ 2)
46              = 2 * (2⁻¹ * S.m * ‖∂ₜ (trajectory S IC) t‖ ^ 2
47              + 2⁻¹ * (S.k * √(‖IC.x₀‖ ^ 2 + (‖IC.v₀‖ / S.ω) ^ 2) ^ 2)) := by
48            simp [mul_add]
49            rw [← mul_assoc, ← mul_assoc]
50            rw [mul_inv_cancel_of_invertible 2, one_mul]
51        _ = 2 * (1 / 2 * (S.m * ‖IC.v₀‖ ^ 2 + S.k * ‖IC.x₀‖ ^ 2)) := by rw [energies]
52        _ = S.m * ‖IC.v₀‖ ^ 2 + S.k * ‖IC.x₀‖ ^ 2 := by simp
```

Here, we carry out a fairly straight-forward calulation of the fact that

```
 S.m * ‖∂ₜ (trajectory S IC) t‖ ^ 2 + S.k * (√(‖IC.x₀‖ ^ 2 + (‖IC.v₀‖ / S.ω) ^ 2) ^ 2) .
= S.m * ‖IC.v₀‖ ^ 2 + S.k * ‖IC.x₀‖ ^ 2 .
```

lines 53 - 59

```
53      have h₂ : S.m * ‖∂ₜ (trajectory S IC) t‖ ^ 2
54      + S.k * (‖IC.x₀‖ ^ 2 + (‖IC.v₀‖ / S.ω) ^ 2)
55        = S.m * ‖IC.v₀‖ ^ 2 + S.k * ‖IC.x₀‖ ^ 2 := by
56       rw [← h₁, sq_sqrt ?_]
57       apply add_nonneg
58       apply sq_nonneg
59       apply sq_nonneg
```

Here, we carry out a very similar computation as in lines 41 through 52. We have now removed the square roots from that computation.

lines 60 - 80

```
60      have h₃: ‖∂ₜ (trajectory S IC) t‖ ^ 2 = ‖IC.v₀‖ ^ 2 - (S.k / S.m) * (‖IC.v₀‖ / S.ω) ^ 2
61      := by
62       calc
63        ‖∂ₜ (trajectory S IC) t‖ ^ 2 = (1 / S.m) * (S.m * ‖∂ₜ (trajectory S IC) t‖ ^ 2
64        + S.k * (‖IC.x₀‖ ^ 2 + (‖IC.v₀‖ / S.ω) ^ 2) - S.k * (‖IC.x₀‖ ^ 2
65        + (‖IC.v₀‖ / S.ω) ^ 2)) := by simp
66        _ = (1 / S.m) * (S.m * ‖IC.v₀‖ ^ 2 + S.k * ‖IC.x₀‖ ^ 2
67         - S.k * (‖IC.x₀‖ ^ 2 + (‖IC.v₀‖ / S.ω) ^ 2)) := by rw [h₂]
68        _ = (1 / S.m) * (S.m * ‖IC.v₀‖ ^ 2 + S.k * ‖IC.x₀‖ ^ 2
69         - S.k * ‖IC.x₀‖ ^ 2 - S.k * (‖IC.v₀‖ / S.ω) ^ 2) := by
70         rw [mul_add S.k (‖IC.x₀‖ ^ 2) ((‖IC.v₀‖ /S.ω) ^2)]
71         rw [←sub_sub_sub_eq (S.m * ‖IC.v₀‖ ^ 2) (S.k * ‖IC.x₀‖ ^ 2)
72         (S.k * (‖IC.v₀‖ / S.ω) ^ 2) (S.k * ‖IC.x₀‖ ^ 2)]
73         simp only [one_div, sub_sub_sub_cancel_right, add_sub_cancel_right]
74        _ = (1 / S.m) * (S.m * ‖IC.v₀‖ ^ 2 - S.k * (‖IC.v₀‖ / S.ω) ^ 2) := by simp
75        _ = (1 / S.m) * (S.m * ‖IC.v₀‖ ^ 2) - (1 / S.m) * (S.k * (‖IC.v₀‖ / S.ω) ^ 2) := by
76         rw [mul_sub (1 / S.m) (S.m * ‖IC.v₀‖ ^ 2) (S.k * (‖IC.v₀‖ / S.ω) ^ 2)]
77        _ = ‖IC.v₀‖ ^ 2 - (S.k / S.m) * (‖IC.v₀‖ / S.ω) ^ 2 := by
78         simp only [one_div, ne_eq, m_neq_zero, not_false_eq_true, inv_mul_cancel_left₀,
79          sub_right_inj]
80         rw [← mul_assoc, inv_mul_eq_div S.m S.k]
```

We now carry out a long computation to obtain a new fact, namely we want to state that

`h₃: ‖∂ₜ (trajectory S IC) t‖ ^ 2 = ‖IC.v₀‖ ^ 2 - (S.k / S.m) * (‖IC.v₀‖ / S.ω) ^ 2.`

While most of this all is a simple computation, we note two theorems on line 78 that we might not expect; namely, we see `not_false_eq_true` and `ne_eq`. These appear, because we need to show some goals that are of the form `x ≠ 0`. With the way negation is defined in LEAN4, this is definitionally equal (via `ne_eq`) to `¬(x = 0)`. To then later get rid of the negation symbol `¬`, we need `not_false_eq_true`.

lines 81 - 85

```
81      rw [← ω_sq, div_pow ‖IC.v₀‖ S.ω 2] at h₃
82      rw [mul_div_cancel₀ (‖IC.v₀‖ ^ 2) ?_] at h₃
83      rw [sub_self (‖IC.v₀‖ ^ 2)] at h₃
84      rw [sq_eq_zero_iff] at h₃
85      exact h₃
```

We now manipulate `h₃` to be of the form `‖∂ₜ (trajectory S IC) t‖ = 0`, so we can close our main goal. However, in manipulating `h₃`, we did create some extra goals, so our proof is not quite yet done.

lines 86 - 88

```
86      rw [pow_ne_zero_iff ?_]
87      apply ω_neq_zero
88      exact Ne.symm (Nat.zero_ne_add_one 1)
```

Finally, we close the remaining goals. These goals are artifacts of small details we did not prove when using previous theorems. In lines 86 and 87 we close the goal `S.ω ^ 2 ≠ 0`, while in line 88 we close the goal that `2 ≠ 0`.

We see that while this proof was long in terms of the number of lines of code, most of this proof actually consisted of quite straightforward symbol manipulation. More precisely, most effort went into guiding the system in how to perform these manipulations.

4 How to get involved

Hopefully by now, you are convinced of the power and future of theorem provers, and want to join in. In this section, we aim to show the ways to get involved with LEAN4.

4.1 Resources

The first step in getting involved in LEAN4 is to install LEAN4. After all, the best way to learn is to do. The most common way to run LEAN4 is on VScode. Instructions on how to install LEAN4 in VSCode can be found here.

Now that LEAN4 is installed, it is good to get used to the syntax of LEAN4. Several books and other resources with this purpose exist, the two most common ones for learning how to use LEAN4 as a proof assistant are ”Mathematics in LEAN” [3] and ”Theorem proving in LEAN” [2]. The main difference between these two books is their approach: ”Mathematics in LEAN” very practically shows you how to use tactics and you will be writing quite complicated proofs fairly quickly. ”Theorem proving in LEAN” focuses more on why everything works as it does, and first tried to get a wide knowledge of everything LEAN4 has to offer before going into depth and complicated proofs.

Another often recommended book is ”Functional programming in LEAN” [8], which is moreso geared towards programmers and not as much to mathematicians. It can be a good source to refer back to regardless.

The final source you should not miss is the actual documentation. This can be found in the LEAN language reference [20], as well as in the Mathlib and LEAN4 github pages.

Now that you know all about how LEAN4 works, you might want to write your own proofs. Mathlib does have certain naming conventions, but it can still be difficult to find the proper theorem names. Luckily several search engines have been built for this exact purpose, such as ”Loogle”, ”LeanSearch”, and ”LeanExplore”.

If you want a slightly less formal introduction to LEAN4, the LEAN game server is an excellent introduction without requiring installation. In particular, the Natural Number Game is a good introduction to the world of proof assistants in general.

4.2 Zulip and community projects

Mathematics is rarely a solo project. Most papers that are being written and most mathematics that is being done in general is done by two or more people. As much as this is already true in informal mathematics, it holds even more true in formal mathematics, as these forms of mathematics are often online and thus convenient to work on with multiple people.

One large, recent example of a large collaboration happened in 2023 under the direction of Terence Tao. After finding an informal proof of the polynomial Freiman-Ruzsa conjecture, Tao set out to formalize this in LEAN4. In only three weeks, over 30 contributers wrote over 10 000 lines of code.

Several community projects are happening right now as well that you could contribute to. The aforementioned Mathlib and PhysLean are also examples of community projects. Some other active community projects are Terrence Tao’s formalization of his Analysis 1 book, the π-base, which is a database of topological spaces, or the formalization of Fermats last theorem

Projects like this are generally coordinated through Zulip. Zulip is an open-source chat software. The LEAN4 Zulip has over 13 000 members and is an incredibly active and welcoming place for both new LEAN4 users and experienced LEAN4 veterans. On here, people ask and answer questions, coordinate community projects, plan both physical and online meetups, and much more.

5 Conclusion

At this point, LEAN4 has become one of the most popular proof assistants out there for mathematicians, and in this thesis we showed why it is on the list of popular proof assistants. We explored LEAN4 itself, as well as several community projects. In particular we explore PhysLean, to which I also made a contribution, as discussed in Section 3.2.2.

When I first started on this project, I was mostly very curious. I had not really heard of proof assistants before starting, but was really excited to learn something so new to me. The learning curve was quite steep at first, but as I found my footing in what felt like this new side of mathematics, I found a deep appreciation for the rigor and structure that these proof assistants bring to the table.

Of course there were times of frustration. The syntax of LEAN4 can be quite finnicky, especially in parts of the Mathlib library that are used less. In particular, I got quite stuck on defining a basis when it came to the proof of the divergence. However, there is no feeling like when you are finally able to resolve such a thing. It is extremely rewarding to be able to work through these roadblocks and succeed. As discussed in Section 2.3, some of the automation tools, like `aesop` and `apply?` were very useful in resolving these struggles. Most of all though, the entire community has been an extremely important aid in going through struggles, and getting acquainted with the syntax and the process.

I also believe that learning about these proof assistants made me do my ”normal” mathematics in a different way. I look at assumptions much more critical; I do not always assume I need all my hypotheses anymore. I am a lot more explicit about assumptions I am making, and more clear in my wording and notation. I believe proof assistants have many elements such as these that most mathematicians can learn from to elevate their proofs to the next level.

LEAN4 was not the only thing I learned during the writing of this thesis. I also learned how to manage my workflow using git, how to cooperate on github, and how to use linters, just to name a few things. Many of these things I will likely be using the rest of my career moving forward.

Looking forward, the future of proof assistants is looking bright in my eyes. The development of large language models are already influencing how proofs are written and generated. LEAN4 and other proof assistants can complement, or in the future maybe even replace, these models by implementing (semi-) automated theorem proving. Several projects developing this use of LEAN4 already exists.

While I do not see artificial intelligence fully replacing mathematicians, I do see the job of the mathematician shifting towards communication. While it would be very impressive if new proofs can be generated, I still see a human telling the computer what to prove, and also translating the work the proof assistant did to something that is easily readable. Formal proofs are great for verifiability, but not so much for readability. Furthermore, I also do not see an automated proof assistant being able to communicate why a certain theorem is useful or relevant. These things still seem like an art best left for humans.

A thousand years ago, proofs were shared and published on paper. A hundred years ago, typewriters took over the industry standard for publishing. Fifty years ago, we again had a shift to LaTeX,and computers as the industry standard for publishing. Nobody can look into the future, but to me it seems entirely possible that in fifty more years proof assistants will be the industry standard for publishing and communicating mathematics. Papers could include links to github repositories with proofs verified, while papers can moreso focus on the main ideas, relevance, and beauty that proofs can contain. In this way, proof assistants could become an integral part of every mathematicians toolkit.

References

[1] J. Adámek, H. Horst, and G.E. Strecker. *ABSTRACT AND CONCRETE CATEGORIES, THE JOY OF CATS.* URL: `http://www.tac.mta.ca/tac/reprints/articles/17/tr17.pdf`.

[2] J. Avigad, L. De Moura, and S. Kong. *Theorem proving in Lean.* URL: `https://leanprover.github.io/theorem_proving_in_lean4/`.

[3] J. Avigad and P. Massot. *Mathematics in lean.* URL: `https://leanprover-community.github.io/mathematics_in_lean/`.

[4] J. Barwise et al. *Language, proof and logic.* CSLI publications Stanford, USA, 2011. ISBN: 978-1-57586-632-1.

[5] N. Bourbaki. *General topology: chapters 1–4.* Springer Science & Business Media, 2013. ISBN: 978-3540642411.

[6] N.G. de Bruijn. "Verification of Mathematical Proofs by a Computer: A preparatory study for a project Automath". In: *Studies in Logic and the Foundations of Mathematics.* Vol. 133. Elsevier, 1994.

[7] K. Buzzard et al. *mathlib4.* GitHub repository. URL: `https://github.com/leanprover-community/mathlib4`.

[8] D.T. Christiansen. *Functional programming in Lean.* URL: `https://lean-lang.%20org/functionalprogramminginlean`.

[9] T. Coquand and G. Huet. "The calculus of constructions". In: *Information and Computation* 76.2 (1988). URL: `https://doi.org/10.1016/0890-5401(88)90005-3`.

[10] L. De Moura and S. Ullrich. "The Lean 4 Theorem Prover and Programming Language". In: *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings.* Springer-Verlag, 2021. ISBN: 978-3-030-79875-8. URL: `https://doi.org/10.1007/978-3-030-79876-5_37`.

[11] F. van Doorn, J. von Raumer, and U. Buchholtz. "Homotopy Type Theory in Lean". In: *Interactive Theorem Proving.* Springer International Publishing, 2017. ISBN: 978-3-319-66107-0.

[12] T.C. Hales. "Formal proof". In: *Notices of the AMS* 55.11 (2008). URL: `https://cse-robotics.engr.tamu.edu/dshell/cs625/Hales-Formal_Proof.pdf`.

[13] M. Hofmann. "Syntax and Semantics of Dependent Types". In: *Semantics and Logics of Computation.* Publications of the Newton Institute. Cambridge University Press, 1997.

[14] W. A. Howard. "The formulae-as-types notion of construction". In: *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980). URL: `https://www.dcc.fc.up.pt/~acm/howard2.pdf`.

[15] L.D. Landau and E.M. Lifshitz. *Mechanics.* CUP Archive, 1960. ISBN: 0 7506 2896 0.

[16] J. Limperg and A.H. From. "Aesop: White-Box Best-First Proof Search for Lean". In: *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs.* Association for Computing Machinery, 2023. ISBN: 9798400700262. URL: `https://doi.org/10.1145/3573105.3575671`.

[17] A.R. Matei. "Type Theory and its Homotopical Interpretation". BSc thesis. Rijksuniversiteit Groningen, 2025.

[18] E. Riehl. *Category theory in context.* Courier Dover Publications, 2017. ISBN: 978-0486809038.

[19] W. A. Sutherland. *Introduction to metric and topological spaces.* Oxford University Press, 2009. ISBN: 978-0199563081.

[20] The Lean Developers. *The Lean Language Reference*. Version 4.24.0. 2025. URL: https://lean-lang.org/doc/reference/latest/.

[21] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013.