A BEGINNER'S GUIDE ON HOW TO AUTOFORMALIZE MATHEMATICS IN LEAN

LUKAS LIEHR, DAVID MUÑOZ-LAHOZ, AND MITCHELL A. TAYLOR

This article is intended as a practical guide for mathematicians who want to formalize their papers using LLMs. It will explain how to set up Lean on your computer, discuss how to read Lean statements, present precise instructions on how to use LLMs to formalize mathematics, and give several tips and tricks that we have learned through trial and error over the last nine months.

Before starting, it is important to emphasize two points. First, AI is changing extremely rapidly, and much of the material mentioned here will surely go out of date within a few months[1]. That, of course, does not mean that it is not worth learning now. Indeed, when we first started autoformalizing our papers in February/March we had to design or invoke specific workflows, skills, and MCP. Although this is now largely unnecessary, learning exactly how autoformalization worked was extremely instructive and made us much better at adapting to future technology. Moreover, some of the skills mentioned below, such as learning how to *read* Lean, will likely become increasingly useful in the future.

Second, it is important to make one's goals clear. The advice below is aimed at the working mathematician who wants to formally verify the correctness of results that they care about; it is *not* aimed at someone whose goal is to produce beautiful, curated Lean code and contribute to Mathlib. However, Lean is very fun, and we do hope that curiosity and an initial push to learn Lean will eventually bring people from the first camp to the second.

1. Outline of the tutorial

The tutorial begins in Section 2 by explaining how to set up Lean on your computer. This section mainly refers to the second author's guide [12], which explains the process in detail. In Section 3 we discuss the importance of learning how to *read* Lean statements. We also give several examples explaining the syntax of Lean as well as a comparison between an AI-generated Lean statement and a human curated one. Finally, we provide useful references for those who want to delve deeper into formal verification and clarify some points regarding what Lean does and does not certify.

Section 4 discusses how one may use LLMs to autoformalize papers. We begin with an explanation of coding agents and servers, as well as a brief description of what the LLM is doing behind the scenes when formalizing a result. Then, in Section 4.4, we present a basic set of prompts that can be used to formalize any paper which, modulo Mathlib, is self-contained and less than 150 pages. After this, we discuss how to improve the workflow using skills, how to verify that the result is properly formalized, and how to nicely present the code. We also offer various tips and tricks that can be used to formalize longer papers and produce better code. Finally, Section 5 discusses the differences between building a resuable library and verifying a "one-off" result. Some basic advice for formalizing high-quality libraries is also provided.

Date: November 20, 2026.

[1]We do plan to maintain this guide, and post updated versions with new "best practices".

2. Setting up Lean

In order to get our community up to speed on Lean verification, the second author created the tutorials [12]. These tutorials carefully explain how to install Lean, as well as related tools such as VS Code. They are intended for beginners with no prior experience with computers, and explain every step of the process in complete detail.

The tutorials also contain information on the Banach lattice Lean library [15], coding agents, GitHub, and the basics of how Lean works.

3. Reading versus writing Lean

A Lean certificate says that a conclusion follows from the assumptions. It does not say that the statement means what you think it means. This is why learning how to *read* Lean is important, so that you can ensure that the theorem in your paper and the Lean theorem agree. Notably, learning how to read Lean is significantly less time-consuming than writing proofs in Lean.

Our first recommendation for those interested in learning Lean is to play the Natural Number Game and to join the Lean Zulip chat. The former offers a glimpse of how to use Lean by playing an interactive game and the latter will introduce you to a wonderful community of people interested in formal verification. After this, one can proceed in various directions, some of which we outline below.

3.1. **Writing Lean.** There are many resources for learning how to write mathematics in Lean, as well as for learning how Lean itself works. For mathematicians, the book *Mathematics in Lean* by Jeremy Avigad and Patrick Massot [4] is an excellent resource. Another excellent resource more devoted to computer scientists and those interested in the theory of Lean is *Theorem Proving in Lean* 4 by Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich [3][2].

3.2. **Reading Lean.** Writing proofs in Lean is a slow process that requires a lot of skill. Indeed, since one cannot skip any details, one needs to be very efficient, learn to employ tactics, and carefully think about reusability. Traditionally, this has limited the rate at which Lean theorems could be proven. However, now that LLMs can write code, it is possible to produce "one-off" verifications of extremely deep results essentially autonomously. For this purpose, one is less interested in the quality of the code, as its only purpose is to compile the theorem[3]. What the author must do is ensure that the theorem that is being compiled is the intended one. This leads to the importance of knowing how to *read* Lean and knowing how to determine whether a Lean verification was performed correctly.

Since autoformalization of research papers has only recently become practical, there is not yet a canonical resource designed to teach the working mathematician how to autoformalize. Best practices also change weekly, so a static reference will quickly become outdated. One nice reference on how to read Lean is [8]. The tutorials [12] are what we used to get our community up to speed on Lean, and additional references such as [11] have recently been created.

Another method that we used to teach people in our community how to read Lean was to play a game where we asked an LLM to convert a theorem in natural language into Lean code, but in different ways. In one case, we asked it to import the statement directly from Mathlib, but in the other we asked it to incorporate a subtle error that made the translation unfaithful. We

[2]And for those interested in the fascinating history behind Lean, we suggest *The Proof in the Code: How a Truth Machine Is Transforming Math and AI* [10] by Kevin Hartnett.

[3]This is in contrast to building a reusable library, where maintaining code quality is paramount.

then tasked ourselves with identifying the correct translation and the error. This was a fun and effective way for our group to get up to speed with reading Lean. We also had many discussions on formal verification and gave multiple talks and mini-courses; we refer the reader to [19] for more information on how we were able to get our community to embrace Lean.

3.2.1. *Examples of what a theorem looks like in Lean.* In Lean, everything has a type. Types are a primitive notion, in the spirit of sets. For example, 1 can have type $\mathbb{N}$, expressed as $(1 : \mathbb{N})$. The function $f(x) := x^2$ can have type $\mathbb{R} \to \mathbb{R}$ or $\mathbb{N} \to \mathbb{N}$, depending on the context, which in Lean must be made explicit. Theorems are defined as follows, with named variables and hypotheses

Statements in Lean

```
theorem TheoremName -- Two dashes start a comment, (this is a comment)
    {variable1 : Type} (variable2 : Type) (variable3 : Type) :
    -- The colon separates the hypotheses and the conclusion
    conclusion
```

We will illustrate the structure of Lean theorems via several examples. Using the web editor `live.lean-lang.org`, the reader may run Lean together with Mathlib in the browser without installation. Each example can be pasted into the editor, provided that the file starts with the line `import Mathlib`. The editor will then give information on whether the syntax is correct and tell you if the code compiles. Note that mathematical symbols can be written in Lean in a similar fashion as in LaTeX. As with LaTeX, the best way to develop a familiarity with the language is to try writing things yourself.

3.2.2. *Example 1.* We begin with the standard inequality between the geometric and arithmetic means of two nonnegative real numbers. As a first step, we state the theorem in Lean without providing a proof.

Example 1

```
theorem sqrt_mul_le_half_add (a b : ℝ) (ha : 0 ≤ a) (hb : 0 ≤ b) :
    √(a * b) ≤ (a + b) / 2 := by
  sorry
```

The keyword `theorem` introduces a theorem, followed by its name, which in this case is given by `sqrt_mul_le_half_add`. The variables and hypotheses are then specified in parentheses. The term `(a b : ℝ)` introduces two real numbers a and b, while `(ha : 0 ≤ a)` introduces the hypothesis that $a \geq 0$. Here, `ha` is the name of the hypothesis and the proof can later deal with it like a variable. Similarly, `hb` denotes the hypothesis $b \geq 0$. The colon following the two hypotheses `(ha : 0 ≤ a) (hb : 0 ≤ b)` separates them from the actual statement in Lean which is `√(a * b) ≤ (a + b) / 2`. This statement together with the two hypotheses translates directly into the familiar mathematical assertion that for all $a, b \in \mathbb{R}$ with $a, b \geq 0$, we have $\sqrt{ab} \leq \frac{a+b}{2}$.

Everything following the symbol `:=` belongs to the proof. In the example above, the proof is replaced by `sorry`, a placeholder indicating that a proof has not yet been provided. Lean accepts such a declaration but issues a warning. It therefore checks that the statement is syntactically and mathematically well-formed, without establishing that the theorem is true. Consequently, when the goal is to verify that a mathematical theorem has been faithfully translated into Lean, it is sufficient to inspect the part before `sorry`, without considering the proof.

To illustrate what a proof in Lean looks like, we now present the same theorem statement, replacing `sorry` with a complete proof.

Example 1 (with proof)

```
theorem sqrt_mul_le_half_add (a b : ℝ) (ha : 0 ≤ a) (hb : 0 ≤ b) :
    √(a * b) ≤ (a + b) / 2 := by
      rw [Real.sqrt_le_left (by positivity)]
      nlinarith [sq_nonneg (a - b)]
```

The statement remains unchanged and only the placeholder `sorry` has been replaced by two lines of Lean code. The proof is verified by the Lean kernel.

3.2.3. *Example 2.* Quantifiers, implications, and equivalences are expressed in Lean much as they are in ordinary mathematical notation. The following theorem states that a function $f : \mathbb{R} \to \mathbb{R}$ is continuous if and only if it satisfies the ε-δ criterion at every $x \in \mathbb{R}$.

Example 2

```
theorem continuous_iff_eps_delta (f : ℝ → ℝ) :
    Continuous f ↔
    ∀ x, ∀ ε > 0, ∃ δ > 0, ∀ y, |y - x| < δ → |f y - f x| < ε := by
  sorry
```

The expression `f : ℝ → ℝ` introduces a function from $\mathbb{R}$ to $\mathbb{R}$. Function applications are written without parentheses, so `f y` denotes $f(y)$. The arrow $\to$ is used both for function types, as in $\mathbb{R} \to \mathbb{R}$, and for logical implications. The symbol $\leftrightarrow$ denotes logical equivalence, while $\forall$ and $\exists$ denote the universal and existential quantifiers, respectively.

Unlike the previous example, the above statement also uses a notion that is not explicitly defined in the code: `Continuous`. This is a definition provided by Mathlib, expressing continuity for maps between arbitrary topological spaces. Thus, to verify that the Lean statement agrees with the intended mathematical theorem, the reader must understand the meaning of `Continuous`. In a Lean editor, hovering over a name displays information about its type and, when available, its documentation, providing a convenient way to inspect notions imported from Mathlib. The standard editor for Lean is VS Code with the Lean 4 extension. If one moves the mouse over a name in this editor, then a small window displays the relevant information. Moreover, one can easily open the Lean file in which the name is defined. This is a practical way to inspect the notions that are imported from Mathlib.

3.2.4. *Example 3.* Theorems about abstract spaces use two further kinds of brackets. As an example, let us consider the Cauchy-Schwarz inequality in a complex inner product space.

Example 3

```
theorem cauchy_schwarz {H : Type*} [NormedAddCommGroup H]
    [InnerProductSpace ℂ H] (x y : H) : ‖inner ℂ x y‖ ≤ ‖x‖ * ‖y‖ := by
  sorry
```

The above statement contains four brackets, and we read them one by one. The bracket `{H : Type*}` says that H is a type. At this point, the type H has no further structure. The two square brackets equip H with a structure, which Mathlib builds in layers. The bracket `[NormedAddCommGroup H]` says that H is a normed Abelian group, and `[InnerProductSpace ℂ H]` adds a complex inner product that induces the norm. Hence, the first three brackets are the Lean version of the sentence "let H be a complex inner product space". Note that adding the bracket `[CompleteSpace H]` would turn H into a Hilbert space. Finally, `(x y : H)` introduces two vectors

$x, y \in H$. Notice that without the information in the square brackets, Lean rejects the statement, since the expression `{H : Type*}` has neither a norm nor an inner product.

The three kinds of brackets are not so relevant for stating the theorem itself but make a difference when the theorem is applied. In order to see this, we apply the theorem in the setting when $H = \mathbb{C}$.

Application of Example 3

```
example (z w : ℂ) : ‖inner ℂ z w‖ ≤ ‖z‖ * ‖w‖ :=
  cauchy_schwarz z w
```

Only the arguments introduced in round brackets, namely z and w, are supplied explicitly. Arguments in curly brackets are inferred by Lean from the available information. Here, Lean determines that $H = \mathbb{C}$ because z and w are complex numbers. Arguments in square brackets are supplied automatically through Lean's typeclass inference mechanism. In this case, Lean finds the normed group and inner product structures on $\mathbb{C}$ already defined in Mathlib. If we replaced `{H : Type*}` with `(H : Type*)` in the theorem declaration, the mathematical statement would remain unchanged, but H would become an explicit argument. The application would then read `cauchy_schwarz ℂ z w`. Thus, curly brackets are particularly useful for arguments that can be inferred from the remaining ones.

Finally, we remark that `inner ℂ x y` denotes the inner product $\langle x, y\rangle$, which in Mathlib is linear in the second argument and conjugate linear in the first.

3.3. **Can't I just get an LLM to state my theorems?** If we are using LLMs to write the Lean proofs, can't we also get them to write the statements of our main results in Lean? The short answer is yes. If you ask a frontier LLM to convert a natural language statement into the same statement written in Lean, it will do it correctly with high probability. The probability will also increase if you send multiple agents and try different frontier LLMs. However, there are several reasons why we believe that it is important to be able to certify the translation yourself. The first is that one should know why one's own results are correctly formalized. The second is that LLMs still make mistakes, and the third is that even if the conversion is faithful, it is often *ugly*. Indeed, for likely the same reasons that current LLMs have issues with mathematical exposition, they also have issues writing Lean code that is palatable to humans. In our experience, the code that they write is usually about twice as long as it should be (which is much better than it used to be!) and is not optimized for human understanding. Thus, being able to curate the formal *statement* of the theorem allows one to better communicate to others why it is faithful.

3.4. **AI-generated versus manually curated Lean statements.** Here, we will illustrate the difference between AI-generated Lean statements and manually curated ones through a concrete example from basis theory. The result is derived from our recent paper on Schauder bases of translates [9]. The starting point was the observation that certain arguments in [9] extend to a general group theoretical setting. For this demonstration, we asked GPT-6 Astra to identify and extract the parts of the original paper needed to establish this more general result. We then asked an LLM to translate the mathematical statement into Lean. After this, we made a carefully curated Lean statement by hand. Although both statements are correct, we will see that there are many choices involved when writing a theorem in Lean and that these choices determine how mathematically natural and readable the statement is.

3.4.1. *Target theorem.* The theorem that we wish to formalize is stated in the opening paragraph of the corresponding PDF paper as follows:

> Let $U = (U_t)_{t\in\mathbb{R}}$ be a strongly continuous one-parameter group of unitary operators on a separable infinite-dimensional complex Hilbert space H. A vector $v \in H$ is *cyclic* if the linear span of $\{U_t v : t \in \mathbb{R}\}$ is dense in H. Moreover, a nonzero vector $w \in H$ is a *common eigenvector* of U if for every $t \in \mathbb{R}$ we have $U_t w \in \mathbb{C}w$. The group U has *simple continuous spectrum* if it has a cyclic vector and no common eigenvector. Recall that a sequence $(b_n)_{n=1}^{\infty} \subset H$ is a Schauder basis if every vector in H has a unique norm-convergent expansion with respect to $(b_n)_{n=1}^{\infty}$.
>
> **Theorem.** *The group U has simple continuous spectrum if and only if there are $g \in H$ and distinct $t_n \in \mathbb{R}$ such that $(U_{t_n} g)_{n=1}^{\infty}$ is a Schauder basis of H.*

Our goal is to state this theorem in Lean in a self-contained way using Mathlib as the only external dependency. To do this, we will create a file called `Showcase.lean`. This file will import only modules from Mathlib and contain all additional definitions needed to state the theorem that are not already available in Mathlib.

The first step is to identify the mathematical notions required to formulate the theorem. Any notion not already available in Mathlib must be explicitly defined in Lean. To illustrate this, we restate the above paragraph and theorem, highlighting in green the notions that require additional definitions. Moreover, we highlight in yellow the definitions that are already available in Mathlib and are thus accessible through the appropriate imports.

> Let $U = (U_t)_{t\in\mathbb{R}}$ be a strongly continuous one-parameter group of unitary operators on a separable infinite-dimensional complex Hilbert space H. A vector $v \in H$ is *cyclic* if the linear span of $\{U_t v : t \in \mathbb{R}\}$ is dense in H. Moreover, a nonzero vector $w \in H$ is a *common eigenvector* of U if for every $t \in \mathbb{R}$ we have $U_t w \in \mathbb{C}w$. The group U has *simple continuous spectrum* if it has a cyclic vector and no common eigenvector. Recall that a sequence $(b_n)_{n=1}^{\infty} \subset H$ is a Schauder basis if every vector in H has a unique norm-convergent expansion with respect to $(b_n)_{n=1}^{\infty}$.

The four notions highlighted in green are defined in the text but not in Mathlib. To incorporate them into `Showcase.lean`, we must translate these mathematical definitions into precise Lean definitions. For clarity, let us collect them in the following list:

(1) A *strongly continuous one-parameter group of unitary operators* is a family $U = (U_t)_{t\in\mathbb{R}}$ of unitary operators on H such that $U_{s+t} = U_s U_t$ for all $s, t \in \mathbb{R}$ and such that the map $t \mapsto U_t x$ is continuous for every $x \in H$.
(2) A vector $v \in H$ is *cyclic* if the linear span of $\{U_t v : t \in \mathbb{R}\}$ is dense in H.
(3) A nonzero vector $w \in H$ is a *common eigenvector* of U if $U_t w \in \mathbb{C}w$ for every $t \in \mathbb{R}$.
(4) The group U has *simple continuous spectrum* if it has a cyclic vector and no common eigenvector.

3.4.2. *Creating and curating the Showcase file.* We began by asking an AI coding agent to generate the file `Showcase.lean` directly from the mathematical paper, without providing any further guidance on how the theorem or its underlying notions should be formalized. Based on this, we will examine how the agent translates a mathematical statement into Lean, including its choices of definitions and Mathlib terminology. For this initial attempt, we used Opus 5.5 in Extra Thinking mode with the following prompt.

```
Consider the file paper.tex in the current folder. Create a file called
Showcase.lean which contains the statement in Lean of the main theorem of
paper.tex (Theorem 1.1 of the compiled TEX file), without a proof (use sorry
as a placeholder). The file Showcase.lean should compile and be
self-contained on the basis of Mathlib, in the sense that it imports only
modules from Mathlib. In particular, any definitions needed to state the
theorem should be included in Showcase.lean.
```

The agent produced the following `Showcase.lean` file. For readability, we have removed a lengthy introductory comment block and shortened the remaining comments to a single line each. No changes were made to the Lean code itself, which is reproduced exactly as generated by Opus. As before, green highlights notions defined within the file, while yellow indicates notions imported from Mathlib.

Showcase.lean — LLM generated

```
import Mathlib

open Filter Topology

namespace SchauderUnitaryOrbits

variable {H : Type*} [NormedAddCommGroup H] [InnerProductSpace ℂ H]

/-- Every vector of `H` has a unique norm-convergent expansion with respect to `b`. -/
def IsSchauderBasis (b : ℕ → H) : Prop :=
  ∀ x : H, ∃! c : ℕ → ℂ,
    Tendsto (fun N : ℕ => ∑ n ∈ Finset.range N, c n • b n) atTop (𝓝 x)

/-- The linear span of the orbit `{U t v : t ∈ ℝ}` is dense in `H`. -/
def IsCyclicVector (U : ℝ → (H ≃ₗᵢ[ℂ] H)) (v : H) : Prop :=
  Dense ((Submodule.span ℂ (Set.range fun t : ℝ => U t v) : Submodule ℂ H) : Set H)

/-- `w ≠ 0` and `U t w ∈ ℂ w` for every `t ∈ ℝ`. -/
def IsCommonEigenvector (U : ℝ → (H ≃ₗᵢ[ℂ] H)) (w : H) : Prop :=
  w ≠ 0 ∧ ∀ t : ℝ, ∃ c : ℂ, U t w = c • w

/-- `U` has a cyclic vector and no common eigenvector. -/
def HasSimpleContinuousSpectrum (U : ℝ → (H ≃ₗᵢ[ℂ] H)) : Prop :=
  (∃ v : H, IsCyclicVector U v) ∧ ∀ w : H, ¬ IsCommonEigenvector U w

/-- Theorem 1.1 of the paper. -/
theorem hasSimpleContinuousSpectrum_iff_exists_schauderBasis_of_orbit
    [CompleteSpace H] [TopologicalSpace.SeparableSpace H]
    (hH : ¬ FiniteDimensional ℂ H)
    (U : ℝ → (H ≃ₗᵢ[ℂ] H))
    (hgroup : ∀ s t : ℝ, ∀ x : H, U (s + t) x = U s (U t x))
    (hcont : ∀ x : H, Continuous fun t : ℝ => U t x) :
    HasSimpleContinuousSpectrum U ↔
      ∃ (g : H) (τ : ℕ → ℝ), Function.Injective τ ∧
        IsSchauderBasis (fun n : ℕ => U (τ n) g) := by
  sorry

end SchauderUnitaryOrbits
```

The generated file compiles and provides a correct Lean formalization of the theorem, in the sense that its statement is mathematically equivalent to the original one. However, this equivalence is not immediately apparent from the code. The assumptions on the Hilbert space are distributed across five separate hypotheses, while the one-parameter group U is not introduced as a mathematical structure but instead specified through the hypotheses `hgroup` and `hcont` in the theorem statement. Moreover, the notion of a Schauder basis is defined from scratch, even though it is already available in Mathlib.

We therefore proceed to curate the generated file, with the goal of obtaining a Lean statement that closely mirrors the original mathematical formulation and is readily understandable to mathematicians familiar with the underlying concepts.

ShowcaseCurated.lean curated

```
import Mathlib

open Function Submodule TopologicalSpace

/-- A complex Hilbert space is a complex inner product space which is complete. -/
class abbrev ComplexHilbertSpace (H : Type*) :=
  NormedAddCommGroup H, InnerProductSpace ℂ H, CompleteSpace H

/-- A unitary operator on H is a surjective linear isometry H → H. -/
abbrev UnitaryOperator (H : Type*) [ComplexHilbertSpace H] := H ≃ₗᵢ[ℂ] H

-- Throughout, U = (U t) is a family of unitary operators on a complex Hilbert space H.
variable {H : Type*} [ComplexHilbertSpace H] (U : ℝ → UnitaryOperator H)

/-- U is a strongly continuous one-parameter group of unitary operators. -/
def IsStronglyContinuousGroup : Prop :=
  (∀ s t x, U (s + t) x = U s (U t x)) ∧ ∀ x, Continuous fun t ↦ U t x

/-- A vector v ∈ H is cyclic if the linear span of {U t v : t ∈ ℝ} is dense in H. -/
def IsCyclicVector (v : H) : Prop :=
  Dense (span ℂ {U t v | t : ℝ} : Set H)

/-- A nonzero vector w ∈ H is a common eigenvector if U t w ∈ ℂ w for every t ∈ ℝ. -/
def IsCommonEigenvector (w : H) : Prop :=
  w ≠ 0 ∧ ∀ t, U t w ∈ ℂ • w

/-- Simple continuous spectrum: U has a cyclic vector and no common eigenvector. -/
def HasSimpleContinuousSpectrum : Prop :=
  (∃ v, IsCyclicVector U v) ∧ ¬ ∃ w, IsCommonEigenvector U w

/-- U has simple continuous spectrum if and only if some (U (t n) g) is a Schauder basis. -/
theorem MainResult
    [SeparableSpace H] (hH : ¬ FiniteDimensional ℂ H) (hU : IsStronglyContinuousGroup U) :
    HasSimpleContinuousSpectrum U ↔
      ∃ (g : H) (t : ℕ → ℝ), Injective t ∧ ∃ b : SchauderBasis ℂ H, ∀ n, b n = U (t n) g := by
  sorry
```

3.4.3. *Differences between the LLM generated and the curated file.* The curated file differs from the generated one in various aspects. First, the LLM-generated file follows the Bourbaki style of introducing each mathematical notion through a separate definition and subsequently referring to it by name. For instance, Mathlib describes a Hilbert space through the three assumptions

`NormedAddCommGroup`, `InnerProductSpace`, and `CompleteSpace`. In the LLM-generated file, completeness appears as a separate hypothesis of the main theorem rather than being incorporated into the definition of the Hilbert space itself. To make this more readable to analysts with little Lean knowledge, the curated file instead combines all three assumptions into a single definition, `ComplexHilbertSpace`, so that completeness is built into the notion of a Hilbert space from the beginning:

curated
```
class abbrev ComplexHilbertSpace (H : Type*) :=
  NormedAddCommGroup H, InnerProductSpace ℂ H, CompleteSpace H
```

Furthermore, the generated file repeats the type of U in each definition, whereas the curated file introduces U once at the beginning, as in the original mathematical statement.

LLM generated
```
def IsCyclicVector (U : ℝ → (H ≃ℓi[ℂ] H)) (v : H) : Prop :=
```
curated
```
def IsCyclicVector (v : H) : Prop :=
```

Similarly, the group law and strong continuity are no longer stated as separate hypotheses of the theorem. Instead, they are incorporated into the definition `IsStronglyContinuousGroup`. This substantially simplifies the theorem's hypotheses.

LLM generated
```
[CompleteSpace H] [TopologicalSpace.SeparableSpace H]
(hH : ¬ FiniteDimensional ℂ H)
(U : ℝ → (H ≃ℓi[ℂ] H))
(hgroup : ∀ s t : ℝ, ∀ x : H, U (s + t) x = U s (U t x))
(hcont : ∀ x : H, Continuous fun t : ℝ => U t x)
```
curated
```
[SeparableSpace H] (hH : ¬ FiniteDimensional ℂ H) (hU : IsStronglyContinuousGroup U)
```

The curated version expresses the assumptions directly, stating that H is separable and infinite-dimensional, and that U is a strongly continuous one-parameter group.

We also remark that mathematical notions often admit equivalent formulations that differ considerably in readability when expressed in Lean. For example, the condition that a vector v is cyclic is expressed as follows in the two files.

LLM generated
```
Dense ((Submodule.span ℂ (Set.range fun t : ℝ => U t v) : Submodule ℂ H) : Set H)
```
curated
```
Dense (span ℂ {U t v | t : ℝ} : Set H)
```

The LLM-generated version is relatively difficult to read, as it introduces an auxiliary function, takes its range, and includes two explicit type annotations. By contrast, the curated version expresses the orbit of v directly as a set, closely following the original mathematical formulation. Similarly, the condition that w is a common eigenvector takes the following forms:

LLM generated
```
w ≠ 0 ∧ ∀ t : ℝ, ∃ c : ℂ, U t w = c • w
```
curated
```
w ≠ 0 ∧ ∀ t, U t w ∈ ℂ • w
```

Here, `ℂ • w` denotes the linear span of w. Thus, the curated version directly translates the mathematical condition $U_t w \in \mathbb{C}w$. Finally, the definition of simple continuous spectrum is expressed as follows:

LLM generated
```
(∃ v : H, IsCyclicVector U v) ∧ ∀ w : H, ¬ IsCommonEigenvector U w
```
curated
```
(∃ v, IsCyclicVector U v) ∧ ¬ ∃ w, IsCommonEigenvector U w
```

The two formulations are logically equivalent, but the curated version mirrors the mathematical statement more closely, stating that there exists a cyclic vector and that there exists no common eigenvector.

Note also that the curated file makes use of notions already available in Mathlib rather than redefining them. For example, the LLM-generated file introduces its own definition of a Schauder basis, despite this notion being available in Mathlib, as indicated by the yellow highlighting above.

LLM generated

```
def IsSchauderBasis (b : ℕ → H) : Prop :=
  ∀ x : H, ∃! c : ℕ → ℂ,
    Tendsto (fun N : ℕ => ∑ n ∈ Finset.range N, c n • b n) atTop (𝓝 x)
```

Understanding this definition requires the reader to be familiar with how Mathlib expresses convergence of sequences. Instead of introducing a new definition, the curated file uses Mathlib's existing `SchauderBasis` structure. The assertion that $(U_{t_n} g)_{n=1}^{\infty}$ is a Schauder basis then becomes

curated

```
∃ b : SchauderBasis ℂ H, ∀ n, b n = U (t n) g
```

The structure `SchauderBasis` consists of the basis vectors together with continuous coefficient functionals. In a Banach space, this is equivalent to the definition recalled above, since the coefficient functionals of a Schauder basis are automatically continuous.

The theorem in the curated file is approximately half as long as its LLM-generated counterpart, and the descriptive but cumbersome original theorem name, consisting of 61 characters, is replaced by the concise name `MainResult`. Moreover, each mathematical notion is expressed in just one or two lines of Lean, making it possible to compare the formal definitions directly with their counterparts in the original mathematical statement.

Finally, we note that `Showcase.lean` serves as the reference input to a comparator, a program that takes the Showcase file and a second file containing a proof, and checks that the latter proves exactly the statement specified in the Showcase file and that the proof is accepted by the Lean kernel. However, the comparator does not verify that the formal statement faithfully represents the theorem in the original paper, nor does it determine whether two different Lean formulations, such as the LLM-generated and curated versions, are mathematically equivalent. Thus, the correspondence between the Showcase file and the original mathematical statement must be checked by a human. This is precisely why careful curation of the Showcase file is essential as it makes this verification as transparent as possible.

Of course, a similar tradeoff in quality is made not only in the formal statement but throughout the entire proof. However, the proof will likely be tens or hundreds of thousands of lines of code, and since it is not meant to be reused, we do not expect the reader to go through it. One can, however, build a skill to write Lean code according to certain rules (just as one can do for deslopifying AI-generated papers). This will substantially increase the quality of both the formal statement and its proof. Nonetheless, these skills are currently far from perfect and usually result in code which is of lower quality than that produced by humans.

3.5. **Does Lean verify that the proof is correct too?** When formalizing a theorem in Lean, one typically begins with the theorem and a proof of the theorem in natural language. Then one attempts to convert this material into Lean code. However, the faithfulness of the conversion of the proof depends heavily on how much care you take and on the existing infrastructure. For example, if you just give the LLM the paper and ask it to formalize it, there is typically no guarantee that the resulting Lean proof will agree with the natural language proof. Indeed, it is very common for

the agents to deviate from the natural language proof [7], as their objective is mainly to compile the theorem.

One reason why Lean proofs differ from natural language proofs is that the body of established mathematical knowledge is significantly larger than Mathlib. In traditional mathematics, the goal of a proof is to reduce a statement outside of the borders of mathematical knowledge to a combination of known results and logical implications. However, the goal of a formal proof is instead to reduce a statement to Mathlib. Often, the path created by using the natural language proof to reach known mathematical knowledge and then formalizing that known knowledge is much longer than finding a direct path to Mathlib. For example, it could be that a theorem is 10 pages out of reach if one invokes a well-known result with a hundred page proof, and 30 pages out of reach if one finds a proof avoiding this fact. Although humans will often prefer to take the former route in their papers, agents will usually prefer to take the latter when formalizing the result.

There are several ways to make it more likely that the natural language proofs and Lean proofs correspond. The first is to build additional infrastructure in the form of a library. This then closes the gap between the frontiers of informal and formal mathematics, and reduces the incentive for the agents to find an alternative proof[4]. Having built the Banach lattice Lean library, we now formalize not only the main results in our papers but also every lemma.

A second way is to limit the autonomy of the LLMs. In March, the LLMs would frequently hallucinate, and we essentially converted our papers into Lean code one quarter page at a time. This then ensured that the formal proof largely followed the natural language proof. However, agents can now run autonomously for hours and are increasingly able to discover shortcuts. Thus, without further instructions, it is likely that the formal proof will deviate from the natural language one. However, one can always ask the LLM to convert the Lean code back to natural language, and compare it with the original natural language proof.

4. Formalizing papers

As mentioned above, LLMs have gotten remarkably good at coding. If your paper, after being enlarged so that it is self-contained modulo Mathlib, is 150 pages or less, you can now formalize it in one day.

Here, we will discuss a workflow that can be used to verify such papers. The goal will be to check for correctness and not to optimize the quality of the code used in the proof. However, we do recommend optimizing the *statement* of the formal theorem, as doing so is important for communicating the result.

Most of the time, the majority of the work is not formalizing the new mathematics in the paper, but getting the LLMs to write all of the prerequisites. The process of making the paper self-contained modulo Mathlib or other formal libraries is quite likely to lead to a substantial increase in the length of the proof, and hence the difficulty of formalizing the result. At the end of this section we will comment on how to Lean verify longer proofs, and discuss various tips for making autoformalization easier. In the next section, we will discuss reusable libraries, which will make both stating and proving theorems in Lean much more efficient.

[4]Alternative proofs often have value, and our hope is that in the future we will be able to use Lean to carefully build multiple dependency graphs showing different ways to reduce a theorem to the axioms of math. This could be an interesting way to identify to what extent results truly depend on each other.

This section discusses many things, including coding harnesses, servers, skills, and what is going on behind the scenes when asking an LLM to formalize a proof. The actual set of prompts used to formalize a paper is now remarkably simple; the reader interested mainly in these can skip to Section 4.4.

4.1. **Harnesses.** To begin, we first need the right tools. For Lean verification, the appropriate tool is a good coding harness. OpenAI's version is called Codex, Anthropic's is called Claude Code, and Google's is called Antigravity. These are agentic tools, which, unlike the usual browser versions of ChatGPT, Claude, or Gemini, have been specifically optimized for coding as well as long horizon tasks. At the moment, our personal favorite is Claude Code, but this changes often.

All of these harnesses are available as part of the usual paid subscription to ChatGPT, Claude, or Gemini, respectively. They can be downloaded as part of the app, or run via the terminal. Through these tools, the user can specify a task to be completed, attach a folder with relevant files, and then adjust the number of agents they send, their subobjectives, and their levels of speed and intelligence. One can have the model run until the task is complete, until it uses a specified percentage of your weekly tokens, or until it encounters a serious issue. Moreover, one can request status updates or provide additional guidance to the model throughout the whole process.

4.2. **Servers.** When doing mathematics, it is perfectly fine to work directly in the browser or the coding harness on your personal computer. Moreover, most of the preparatory steps of the Lean verification can also be done this way. However, the compilation of the Lean code – although possible on a standard computer – is best done on a server. Indeed, if you work directly on your computer, it will get very hot, be slow, and may use a significant fraction of your CPU and RAM. Also, every time you close your laptop, the coding agents will stop working.

Our fix to this was to rent a server with a good CPU and plenty of RAM, and then to SSH on to that computer[5]. This allows us to use our laptops (or even our phones!) to control the server. Consequently, we can run Lean continuously (as the server is always on) and also safely give the LLMs full access to the server without giving them full access to our personal computers.

4.3. **An explanation of what's going on behind the scenes.** The purpose of this subsection is to give the reader a broad idea of what the LLM is doing behind the scenes when formalizing a paper. We will break the process into several steps.

Step 0: Prepare your natural language proof for formalization. A preliminary step is to obtain a self-contained proof of your theorem. To achieve this, simply give the tex file containing the theorem and its proof to one of the coding agents. After this, ask the model to produce a new tex file that contains all of the details of the proof and is entirely self-contained modulo Mathlib. You may want to do this several times, to make sure that the LLMs are not being lazy. Once the length of the proof stagnates, you can move on to the next step. When autoformalizing, it is also good to remove the introduction and motivation from the tex file so that there are no "distractions". The tex file should contain only the statement of the theorem, all necessary definitions, and a complete proof.

Step 1: Identify the theorem to be formalized and convert it into Lean code. The next step is to convert the statement of your main theorem into an equivalent statement written in Lean code. To do this, it is important to clearly state the main theorem in natural language, together with all relevant definitions, hypotheses, quantifiers, and conclusions. You want to make it so that the LLM just has to do a simple "language conversion" from informal to formal mathematics, and does not have to guess what you meant the theorem to be. This will produce a new file that you will call

[5]If you don't know how to do this, a coding agent can help with the setup.

`Showcase.lean`[6] which contains the formal theorem with a `sorry` inserted as the proof. This file should also contain the necessary imports (e.g. Mathlib and whatever other libraries you are using) and all definitions not already in these libraries. The file should compile and be easy to compare with the natural language theorem.

Step 2: Audit and rewrite the Showcase file. Send multiple agents (preferably from different frontier LLMs) to examine the Showcase file to ensure that the conversion is faithful. After the LLMs are confident, check the Showcase file yourself. It is also good practice to clean up this file and write it nicely so that your reader can easily understand it. See Section 3.2.1 for an example.

Step 3: Organize the proof for Lean. You now want to ask the LLM to refactor the natural language proof so that it is easier to convert into Lean code. If you call the original self-contained article paper.tex, the goal is to create a new article refactored.tex that is better suited for formal verification. Such a file will begin with all definitions, followed by labeled lemmas and propositions. The hypotheses of each intermediate result and how each result follows from the preceding ones should be made completely explicit. This is a preliminary step before creating the blueprint.

Step 4: Find the shortest proof path to Mathlib and make a blueprint. The next goal is to optimize the path from your main theorem to Mathlib. The file refactored.tex contains one such path, but it may not be optimal, so it is worth having the LLMs look for more efficient routes. The LLMs can start by searching Mathlib for relevant infrastructure, take a selected lemma in refactored.tex and work backwards to more efficiently reduce it to Mathlib, introduce an alternative strategy to remove some unnecessary lemmas, or split results into simpler claims that are easier to prove. The goal is to find a path to Mathlib which requires producing as little new Lean code as possible. This process will result in a file, which you may call BLUEPRINT.md, which is organized as a directed acyclic graph connecting Mathlib to the theorem you wish to prove. The blueprint should list each selected lemma with its paper label and Lean name, a division into files with dependencies and proofs, and all Mathlib results that will be needed.

Producing a good blueprint is the most important step of the process. If done well, the rest is simply a "language conversion". In particular, it is important to have a complete and detailed blueprint before writing any Lean code, as any simultaneous attempt to produce mathematical infrastructure and convert into Lean code will result in a messy formalization and higher token usage.

Step 5: Choose the remote workspace. All of the above steps were performed in natural language, and can be done on your personal computer. However, we will soon move to the stage of compiling Lean code. Although this can be done on your personal computer, we recommend moving to a more powerful server. This has multiple benefits. First, servers are always on, so the agents will work even after you close your computer. Second, servers are more powerful, so the compilation will be faster. Finally, having a separate computer is more convenient to sandbox the LLMs and the code execution, so that the agents can run autonomously with full permissions on an isolated environment.

Step 6: Create a Lean scaffold. It is now convenient to have the agents convert the definitions and the statements of the intermediate theorems/lemmas into Lean code, with each proof given by `sorry`. If the main theorem compiles, then it follows that the intermediate results imply the main

[6]This is what we have been calling it in our papers, but the community has since agreed on it being called the Challenge file rather than the Showcase file, so it is probably better to call it `Challenge.lean`.

theorem. The task then is to efficiently verify each of these intermediate theorems[7]. To do this, one can split the job into multiple largely independent files, with appropriate temporary axioms introduced. For example, the split could look like

- `Lean_Code/Definitions.lean`
- `Lean_Code/Preliminaries.lean`
- `Lean_Code/BridgeLemmas.lean`
- `Lean_Code/MainTheorem.lean`
- `Lean_Code/AXIOM_STATUS.md`

Moreover, one can create a ledger listing all of the `sorry`'s and axioms that need to be removed.

For short projects, the split into files can be done autonomously. However, for large projects, it is extremely important to do this carefully. Indeed, every LLM has a "context window" which, roughly speaking, is the size of its brain. Once the context window is filled, the LLM must compress the material and can only keep what it deems to be the main points. Modern LLMs have a context window of about one million tokens, which is quite large[8]. However, Lean blueprints are often larger than this. Hence, the LLM may only "see" part of the project at one time, and therefore can hallucinate. By splitting the project nicely, one can send different agents at different parts of the project, leaving ample room in their context windows not only to digest the material presented to them, but to think and execute their tasks.

Step 7: Translate the project into Lean code. Now, we are ready to convert all of the mathematics into Lean code. For short projects, this can be done easily, even with cheaper agents, as it is essentially just a language conversion task. However, for longer projects, it is good to use a powerful agent as an "orchestrator" that spawns multiple subagents of equal or lower intelligence. Since the project has been nicely split into files, one can run many agents in parallel. Moreover, one can use the ledger to track progress. If the blueprint is built correctly, this process should go smoothly. However, if it is done poorly, the project can stall. In this case, one must identify the step of the natural language proof that was not formalizable and revise the blueprint.

Step 8: Final checks. Check again that the formal statement matches the intended mathematics and that the theorem compiles with only the three standard axioms of the kernel (see Section 4.5 for more details). At this point, you can also run a "clean up" workflow, i.e., ask the LLMs to go through the code and look for simplifications. Often, the blueprint is not sufficiently detailed (especially if its size exceeds the context window of the LLM), so the LLMs may have had to produce some additional mathematical infrastructure. A final inspection can often identify dead code and significantly reduce the complexity of the proof. Since this step is now as simple as prompting "go through the Lean code, look for simplifications, and delete dead code", it is certainly worth doing. You can also ask the LLM to refactor the code in any stylistic way you prefer, and to produce a nice visualization of the proof via a dependency graph. Benchmarks such as LeanLean and skills such as cLEAN may be useful for improving Lean code, but we have not personally tested them or any other related benchmarks or skills.

4.4. **A practical workflow.** Here, we present one workflow for formalizing a paper. We, of course, do not claim that it is optimal and encourage the reader to try other workflows such as [1]. The only thing that we guarantee is that this workflow will succeed if your paper is under 150 pages self-contained (and correct!).

[7]If one wants to be safe, one can audit each of these intermediate results with several agents, checking that they are indeed true and manageable. All intermediate lemmas should have relatively short proofs, so that it is easy for the LLM to convert the proof from natural language into Lean code.

[8]It used to be much smaller, which is why in the past using the same chat for too long often decreased mathematical performance, as not only the task, but all preliminary material and thinking material had to fit in the context window.

4.4.1. *Prompt 1.* The first step is to rename your paper to "paper.tex", label the main theorem as "thm:main", and put the tex in a folder. You will also want to create a Showcase file with the equivalent Lean statement of your main result (as explained in Section 3.4) and put it in the folder. After this, you may go to your favorite coding agent, give them access to that folder, and send the following prompt.

```
Audit the LaTeX file paper.tex, whose goal is to prove the theorem with the
label thm:main. The Lean formulation of this theorem is provided in
Showcase.lean, but do not write or modify any Lean code unless explicitly
instructed.

The objective is to make the proof of thm:main fully self-contained, without
relying on external mathematical results whose proofs are not included in
paper.tex.

Examine the proof and identify every dependency on external mathematical
inputs including results that are used implicitly without an explicit
citation.

For each such dependency, develop a complete, rigorous proof and incorporate
it into paper.tex. Keep these additions minimal: include only the results and
arguments that are genuinely necessary. Do not merely restate external
results or replace citations with unproved assertions. All non-elementary
mathematical results needed for thm:main must be established within the
document. In particular, you can only rely on results already proven in
Mathlib, and you should scan Mathlib for such results.

After updating the file, perform an independent audit with a subagent to
verify that all external dependencies have been resolved. If any dependency
cannot be resolved, report it explicitly.
```

The above prompt can be sent with any of GPT 6.2 Astra or Sol, Fable 5.2, or Opus 5.5. You may choose the Extra High or Max reasoning mode but need not use "Ultracode".

4.4.2. *Prompt 2.* Now that we have a self-contained document, we will audit it for formalization. Nine months ago, the instructions had to be extremely precise, but nowadays the frontier LLMs will know what you mean even if you do not provide too many details. A standard prompt is below; it can be sent in the same chat as the previous prompt.

```
    Audit the proof of thm:main in paper.tex for its suitability for
formalization in Lean 4.

The objective is to determine whether the proof can be formalized using
existing Mathlib infrastructure, without developing substantial new
mathematical infrastructure.

In particular, investigate the following questions:

1. Lean-friendly structure: Is the proof divided into sufficiently explicit
definitions, lemmas, and propositions to allow a systematic formalization in
Lean? Identify any steps that should be separated into intermediate results
or reformulated.
```

```
2. Mathlib availability: Does the proof rely only on mathematical concepts,
definitions, and results that are already available in Mathlib? Check the
actual Mathlib library rather than assuming that standard mathematical
results have been formalized (download Mathlib if it is not available).

3. Missing infrastructure: Identify any concepts or arguments that would
require new definitions, substantial auxiliary theories, or mathematical
infrastructure not currently available in Mathlib. Investigate whether these
dependencies can be avoided by using existing Mathlib results or by modifying
the proof strategy.

4. Recommendations: Propose concrete changes to the mathematical proof that
would make its formalization easier, prioritizing approaches that reuse
existing Mathlib infrastructure.

Record all findings in detail in AUDIT.md.
```

4.4.3. *Prompt 3.* With the above information, we can refactor the tex to make it better suited for Lean formalization. Note that since the LLM has access to the folder with the original files, it will automatically deposit all of its findings there.

```
    Based on the findings in AUDIT.md, refactor the mathematical proof in
paper.tex into a Lean-friendly document named refactored.tex.

The objective is to produce a rigorous, self-contained mathematical proof
that can later serve as the foundation for a Lean 4 formalization, relying as
much as possible on existing Mathlib infrastructure.

Instructions:

Refactor the proof. Organize the argument into clearly stated definitions,
lemmas, propositions, and theorems. Formulate and prove nontrivial
intermediate claims separately, avoiding large proof steps, implicit
arguments, and hidden assumptions.

Incorporate the audit findings. Address the obstacles identified in AUDIT.md
by adapting arguments, introducing suitable intermediate results, or choosing
alternative proof strategies that better align with Mathlib.

Preserve mathematical rigor. Provide complete proofs of all results, with
explicit assumptions and logically justified steps. Preserve the statement of
thm:main and avoid unnecessary changes to the mathematical content.

Ensure a clear structure. Organize the document into coherent sections and
subsections, with consistent notation, descriptive theorem labels, and
explicit references to previously established results.
```

4.4.4. *Prompt 4.* Finally, we can make the refactored tex file into a blueprint that the LLM will then try to convert into code. For this and the next step, one should turn on "goal mode", which tells the LLM not to stop until it has completed the objective. In Codex, one can do this by clicking the button on the bottom left of the chat, and in Claude one can start the prompts below with /goal.

```
    Based on refactored.tex, create a formalization blueprint in
BLUEPRINT.md, with the goal of proving the main Lean statement in
Showcase.lean using Lean 4 and Mathlib.

Instructions:

Construct the blueprint. Translate the mathematical proof into precise,
independently formalizable definitions, lemmas, and theorems. Specify their
dependencies, detailed proof strategies, and relevant Mathlib declarations,
verifying their actual availability.

Establish the dependency graph. Provide a complete, acyclic dependency
structure from Mathlib to the main theorem. Ensure every step is sufficiently
fine-grained for autoformalization, without hidden arguments or substantial
missing infrastructure.

Design the Lean file structure. Propose a division into Lean files. Organize
the formalization so that it can proceed incrementally toward the main
theorem.

Independent verification and iterative refinement:

Once the blueprint is complete, launch an independent subagent to audit
whether the dependency graph is acyclic, the path from Mathlib to the main
theorem is complete, and every intermediate step is detailed enough.

If issues are found, revise BLUEPRINT.md. Then launch a fresh, independent
subagent to audit the updated blueprint from scratch.
```

4.4.5. *Prompt 5.* All of the above steps have produced natural language documents. We are now ready to start converting the blueprint into code. For this, we recommend working on a server. One can also now switch to a lower tier model – the critical parts were creating the blueprint and Showcase file, the rest is essentially language conversion.

```
    Based on BLUEPRINT.md, carry out the complete Lean 4 autoformalization,
with the goal of proving the main theorem in Showcase.lean.

Follow the blueprint systematically, using existing Mathlib infrastructure
wherever possible. You may use up to three xhigh subagents, distributing
tasks to maximize efficiency and enable parallel work.

The final theorem must compile without sorry, admit, or additional axioms,
relying only on Lean's three standard axioms.
```

4.4.6. *Outcome.* We applied the five prompts described above to the note on Schauder bases in unitary orbits described in Section 3.4.1. The starting files were `paper.tex` and `Showcase.lean`, to which we applied the five prompts. The resulting files are available in the repository

`https://github.com/lukasliehr/Schauder-Basis-Orbits`.

The repository contains five folders, named Prompt 1 through Prompt 5, each containing the state of the Lean project after the corresponding prompt was executed. In particular, Prompt 5 successfully completed the Lean verification of the main theorem. In Section 4.7 we describe how to create a GitHub repository and good practices in presenting the Lean formalization.

4.5. **Did it work?** So now your LLM has completed the autoformalization process, great! But how can you be certain that the theorem is correctly verified? This topic has already been previously discussed on xenaproject in the article ...and the proof is checked in Lean. For most practical purposes, one simply has to compile the code and print the axioms. There should be no `sorry`, and the only axioms should be Propext, Quot.sound, and Classical.choice. There are also a few precautionary checks to make before being completely certain that the process is complete; we refer the reader to ...and the proof is checked in Lean for an explanation.

We remark that "partial formalizations" relying on additional axioms can be dangerous, as it is surprisingly easy to write a useless statement in Lean. In fact, one way to gain confidence that the statement was correctly translated is to observe that the formal proof actually followed the blueprint – this indicates that the agents were unable to find a loophole to avoid executing the intended task.

4.6. **Skills.** The above workflow can be built into a "skill", i.e., a reusable set of instructions. This gives context and "memory" to the LLM and essentially reduces the above workflow to pressing a button. However, we thought that it would be instructive to explain the inner workings of what is actually happening. A very nice set of skills that we recommend for formalizing papers is [2]; another nice set of skills that can be used to help the LLMs follow stylistic guidelines and produce better code is Lean skills GitHub.

4.7. **Presenting the result.** After completing the Lean verification, it is standard practice to make it available in a GitHub repository and include a link to the repository in the paper. The initial setup consists of creating a GitHub account, installing Git on your computer, and connecting Git to the account, as described in the GitHub setup guide. Once an empty public repository has been created on the GitHub website, the remaining steps can be carried out by a coding agent. With the Lean project folder open in Codex or Claude Code, it suffices to provide the agent with the repository URL and ask it to push the contents of the folder to the repository. The formalization is then accessible at that URL.

A natural starting point for the reader is the `Showcase.lean` file described in Section 3.4. Placing this file in the top-level folder of the repository makes the formal statement of the main theorem easy to find. It provides a self-contained formulation of the main results of the paper, allowing the reader to compare the Lean formalization with the corresponding theorem in the paper.

The `README.md` file provides an overview of the repository, including its structure and the main theorems being formalized. For example, the repository accompanying a paper on energy minimization for eight points on the sphere contains a `Showcase.lean` file, and its README also includes a geometric visualization. It is good practice to curate the README by hand, as it is what tells the reader about the repository.

In the final section of our papers, we also like to explain why the statement in the Showcase file and the statement in our paper agree. This involves stating the theorem in natural language and in Lean, explaining what the syntax is, what definitions are already in Mathlib, and which are defined by us. We then carefully explain how we defined the missing terms in Lean, as in Section 3.4. We believe that including such a section in the paper is good practice, as it helps to encourage the community to read and present Lean. In particular, we refer the reader to [17, 18, 20, 21] for instructive examples of research-level Lean translations.

In our recent papers on Banach lattices, we have also experimented with a different format, where all lemmas are formally verified. In this case, we did not present an appendix on the translation,

as we assumed the reader to be familiar with the Banach lattice Lean library. See, for example, [5, 14] for this presentation style.

4.8. **Anecdotes on the quality of a formalization.** The quality of the formalization one produces depends on many factors. The most important is the quality of the blueprint. However, the model and the amount of careful human intervention also play an important role. For example, in our paper [17], the original blueprint suggested by GPT-5.5 required the formalization of several deep facts in algebraic geometry and resulted in 155k lines of code. After this was completed, we asked Fable 5 to carefully scan the code and look for simplifications. This ended up reducing the code to around 22k lines. Finally, a careful human-led analysis shortened the code by another factor of two while at the same time proving a more general theorem.

This is a general phenomenon. Indeed, at the moment, if asked to do the same task, current LLMs are significantly faster than human experts but produce lower quality code. It will be interesting to see how the reusability of the code improves over time. Of course, for the purpose of formalizing a one-off theorem, this is perhaps not so important. However, maintaining code quality is crucial when building libraries. From our experiments, building reasonable formal libraries on the scale of several millions of lines of code can now be done in a matter of weeks. However, much more experimentation will be needed to test the quality of these libraries and to identify the extent to which they can be built on, improved, and merged.

4.9. **Caveats.** As mentioned above, there are several caveats to brute-force formal verification. However, it is worth emphasizing them again. The first is that the code may not be reusable. Indeed, the purpose of the above workflow is to formally verify a statement, not to build a reusable library. The latter requires more planning, and we will outline some thoughts below. The second related issue is that the proof in Lean may not follow the proof in your paper. In fact, it is extremely common for agents to fix small mistakes in the formalization process or to find new routes that they find easier to implement. This also means that intermediate definitions and lemmas may not faithfully represent those in the paper.

Finally, and most importantly, the author is responsible for checking that the formal theorem is stated precisely. Although Lean can check a proof, it cannot check that the definitions and statements express the intended mathematics. Moreover, the author is also responsible for stating the formal theorem in a way that is amenable to the understanding of other mathematicians.

4.10. **Palomar.** If one has successfully produced a Lean verification of a theorem, one can deposit it on the website Palomar, which is a public repository for Lean proofs.

A submission to Palomar needs the URL of a public GitHub repository and its full commit hash, which is an identification of the exact version of the code to be checked. In the usual setup, `Challenge.lean` states the result, much like the Showcase file described above, and `Solution.lean` gives the proof. A file called `comparator.json` tells Palomar which statements to check. The project also needs a file called `formalization.yaml`. It records things such as who worked on the formalization, how AI was used, and what review of the code took place. It also describes any known differences between the Lean statement and the theorem in the paper. Palomar's submission guide includes a template for this file.

4.11. **Long proofs.** If the agents repeatedly get stuck at a certain step, the blueprint may be missing a mathematical argument. A useful first step is to ask the agent to state the exact claim it is trying to prove, list the facts available to it, and identify the step in the blueprint that is supposed to connect them. This often reveals an unstated lemma or a missing hypothesis. It is also worth checking whether the agent is still following the intended proof strategy. If several agents have

pursued different approaches, comparing the first points at which they diverged from the blueprint can help identify an issue in the blueprint. Once the blueprint has been fixed (e.g. a missing argument has been worked out and clarified), the blueprint can be updated and the formalization resumed from that point. With extremely long proofs, it is not uncommon for agents to get confused, so it is also good practice to thoroughly audit the blueprint and divide the project into multiple independent components that different agents can analyze.

4.12. **Some "best" practices for the Lean verification of papers.** The best advice we can offer for autoformalization is to always stay on top of current technology, be willing to adapt, and to find friends with whom to enjoy the process. There are very few things we can say here that will surely remain true in one year. In fact, there are many interesting strategies that are already completely out of date. For example, it used to be optimal to use a specific combination of Fable and Opus agents so that the five hour (as well as the weekly token rates) of both models would expire at exactly the same time. However, nowadays Opus 5.5 is more than enough for Lean verification. Nevertheless, this practice still has a useful analog in modern times, where the strong agent is used as an orchestrator and for checking the Lean statements, and cheap agents such as Sonnet 5.5 are used for the routine parts of the formal verification.

One practical workflow in Codex is to use Astra at reasoning effort `xhigh` or higher to prepare the blueprint, and then use Sol or Luna at `xhigh` to formalize the proofs. Astra can also serve as an orchestrator, launching Sol subagents to handle independent tasks. A similar setup is possible in Claude Code, with Fable as the main agent and Opus as subagents. One can also use the Codex CLI to send a prompt to Astra, Sol, Luna, or Terra using Fable or Opus.

Each agent has a limited context window, so it helps to assign each agent a limited amount of work. Lean also uses the term *heartbeat* for a unit of computational work. If a proof reaches Lean's heartbeat limit, splitting it into smaller lemmas can help identify the computationally expensive step.

For work on a server, the first step is to clone the project there, install the Lean version specified in `lean-toolchain`, and check that `lake build` succeeds. For instance, Armstrong [1] describes connecting to a server over SSH and starting the coding agent in a `tmux` session, which continues running even after the local computer disconnects. Another option is to run the agent locally and ask it to execute all Lean compilations on the server over SSH.

For a formalization that spans several sessions, Codex's goal mode gives the agent a clear stopping condition. It is also useful to specify a second stopping condition based on the weekly usage quota. When about 2% of the quota remains, the agent should stop assigning new tasks, leave the project in a state that builds successfully, and write a short `HANDOFF.md`. Recording the last successful build, the first unfinished statement, and the next step in its proof is usually enough for a fresh session to resume the work.

If one is feeling particularly ambitious, one can also use "fast mode" to speed up the process, with a significant increase in token costs. One can also configure Codex to run up to 128 agents at the same time. One should also be sure to work with a model with a $\sim$1 million token context window, and reconfigure Codex if this is not the case. If working on a server, it is also extremely useful to change the permission settings to "full access".

4.13. **Autoformalization systems.** We remark that there are numerous systems and platforms devoted specifically to autoformalization, including Archon, Aristotle, AxiomProver, Lea, Numina, OpenGauss, Prove2Me, Trellis, and more. Although we have not used any of these systems, they are certainly worth looking into. Generally speaking, methods to improve an LLMs performance

on any particular task include building a custom harness, post-training the LLM on that task, or doing extensive benchmarking.

5. Formalizing libraries

To use Lean most effectively in your field, it is nice to have a reusable library. The process of building a reusable library is very different from performing a one-off verification of a research paper. Indeed, one now wishes to choose definitions that work across several applications, separate reusable lemmas from arguments specific to one proof, make results easy to find through consistent names, and have a clear organization. Moreover, one also needs to maintain the code as dependencies change, and document how the pieces fit together.

5.1. **Mathlib and TauCeti.** Mathlib is the core library for mathematics in Lean. It is what we use to reduce graduate level mathematics to the axioms and to state our advanced theorems. Our trust in this process stems from the fact that Mathlib is thoughtfully designed and human curated. Indeed, more important than the size of the library is the care that the Mathlib community took in formalizing objects in an intuitive way. This is what makes it possible to quickly and reliably understand formal definitions and to use the library to state more advanced formal theorems.

TauCeti is a more recent library that balances the tradeoff between code quality and scale. The blueprints are human curated, but the proofs are largely written by AI. It is incubated by the Mathlib Initiative and the Lean FRO, and it maintains a very high standard for AI-generated code. In particular, the standards are much higher than what is obtained by doing the "one-off" verifications described above. Another Mathlib initiative is Formal Frontier, which is a project building a pipeline converting source texts into interlinked formal libraries.

5.2. **The Banach lattice Lean library.** The Banach lattice Lean library is our domain-specific library for Banach lattices and related fields. The level of automation of this library sits in-between Mathlib and TauCeti. The code is largely written by LLMs, but with close human supervision of the definitions, statements, docstrings and, more generally, of the implementation. Pull requests are critically reviewed by humans, and changes are continually made to make the code understandable and coherent. In other words, it is a codebase largely written by LLMs, but understandable by humans.

The goal of the library is to serve as a research tool. It is this balance between human understanding and automation that makes the tool useful. On the one hand, automation accelerates the process of writing code so that current research can be formalized in a practical way (as demonstrated in several new research papers [5, 6, 13, 14, 16]). On the other hand, retaining human understanding of the library allows for a better understanding of the field; in fact, the formalization process has served to yield new insights into the theory of Banach lattices (some early ones are reported in [15]). Generally speaking, the project is research-driven rather than an attempt to formalize a specific monograph, and it grows by adding coherent, reusable prerequisites for concrete research-level applications.

An interesting fact about building a good library is that it not only makes theorems easier to prove, but easier to state. Indeed, the more trusted definitions one has in the library, the less ad hoc definitions one has to make when stating a new theorem. In some sense, definitions are the most fundamental part of a library, and the theorems are mainly there to be useful. Indeed, if the definitions are not faithful, then it will be impossible to correctly state a new research-level result. However, if a theorem in the library is not faithfully stated, it will either not compile or not

serve its purpose when proving new research-level theorems[9]. Thus, unlike theorems in a preprint which are at the forefront of knowledge and are not (yet) being repeatedly reused, theorems in a library have a "self-refereeing" mechanism. Thus, focusing on building a trusted "Definitionslib" is perhaps even more important than proving as many formal theorems as possible.

5.3. **Some "best" practices for library building.** The following are some tips and best practices to get started with library building in a way that adds to human understanding.

- **Understand the math.** Before starting the formalization of some part of the theory, you should first have a deep understanding of it yourself. This will serve two purposes. First, it will allow you to know whether the content to be added is within reach of the current state of the library, or whether other parts of the theory should go first. This will also help in organizing the library properly. Second, it will give you judgment and taste, which is the most important skill to make it human-readable; in particular, you will be able to write much more informative and useful documentation. The only way to write documentation that goes beyond a simple translation of what the Lean code says is by having a global understanding of how a result or a construction sits in the theory.
- **Understand Lean and Mathlib.** Similarly to the above, this is the only way for you to be certain that a particular implementation is natural and understandable by other Lean users. In particular, try to stick to Mathlib's naming conventions, code, and documentation styles.
- **Pay attention to docstrings.** As of today, LLMs are particularly bad at writing informative and useful docstrings which go beyond a simple translation of the Lean code. This can be good enough for intermediate and technical lemmas, but not for non-obvious definitions and theorems, and certainly not for file docstrings. The latter are particularly important to be able to navigate the library in an effective way. It is not uncommon for LLMs to provide details about proofs in file docstrings, instead of providing a global overview of the file.

Of course, one can also build libraries by sacrificing human understanding much further, and having LLMs do most of the work. We have experimented in putting together libraries on the order of a few million lines in this way, and it is effective. However, at the moment, it does result in a lower quality library, both in terms of code quality and organization.

We expect that as LLMs improve further, they will be able to produce good formal libraries. On the other hand, a deep level of human understanding is gained when formalizing a result yourself, and a broad overview of the geometry of your field becomes visible when blueprinting and designing an LLM-assisted library. Thus, there will always be tradeoffs between levels of human understanding, trust, speed, and scale.

References

[1] Scott Armstrong. Autoformalization is now very easy. *https://www.scottnarmstrong.com/2026/10/autoformalization-is-now-very-easy/*, 2026.
[2] Scott Armstrong. Lean Autoformalization Skills. *https://github.com/scottnarmstrong/LeanAutoformalizationSkills*, 2026.
[3] Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich. Theorem proving in Lean 4. Online textbook. `https://lean-lang.org/theorem_proving_in_lean4/`.
[4] Jeremy Avigad and Patrick Massot. Mathematics in Lean. Online textbook. `https://leanprover-community.github.io/mathematics_in_lean/`.
[5] Antonio Avilés, Christian Rosendal, Mitchell A. Taylor, and Pedro Tradacete. Coordinate systems and maximal order complexity in Banach lattices. *arXiv preprint arXiv:2610.33462*, 2026.

[9]There is also a third, unlikely, option that you accidentally discover a new, useful theorem by misformalizing a known result.

[6] Antonio Avilés, Mitchell A. Taylor, and Pedro Tradacete. Order closure, order adherence and Fatou norms. *arxiv preprint, arXiv:2609.06689*, 2026.
[7] Alexander Bastounis, Fabian Circelli, and Anders C. Hansen. Navier-Stokes lost in translation: Why Lean verification of AI autoformalisation does not guarantee correct natural language proofs. *arXiv preprint arXiv:2610.08144*, 2026.
[8] Martin Dvořák. How to read Lean. *https://github.com/madvorak/read-lean*, 2024.
[9] Daniel Freeman, Lukas Liehr, and Mitchell A. Taylor. Structured coordinate systems in $L^p(\mathbb{R})$. *arXiv preprint arXiv:2610.18709*, 2026.
[10] Kevin Hartnett. *The Proof in the Code: How a Truth Machine Is Transforming Math and AI*. Farrar, Straus and Giroux, 2026.
[11] Marc Masdeu. Learn Lean without sorries. *https://mmasdeu.github.io/leancraftgame/*, 2026.
[12] David Muñoz-Lahoz. Autoformalize your Math. *https://lean.functionalanalysismadrid.com/*, 2026.
[13] David Muñoz-Lahoz. Wickstead's conjecture on positive projections and non-representable Banach lattice algebras. *arxiv preprint, arXiv:2604.14697*, 2026.
[14] David Muñoz-Lahoz, Mitchell A. Taylor, and Pedro Tradacete. Aliprantis's questions on locally solid topologies. *arXiv preprint arXiv:2609.29498*, 2026.
[15] David Muñoz-Lahoz. The Banach lattice Lean library. *arXiv preprint arXiv:2608.07388*, 2026.
[16] Raúl E. Pariente-Martín and Pedro Tradacete. Order continuity of induced homomorphisms between free Banach lattices. *arxiv preprint, arXiv:2611.00451*, 2026.
[17] Jaume de Dios Pont, Josef Greilhuber, Lukas Liehr, and Mitchell A. Taylor. Phase retrieval from a uniformly discrete point set. *arXiv preprint arXiv:2609.28341*, 2026.
[18] Jaume de Dios Pont, Karlheinz Gröchenig, Lukas Liehr, Irina Shafkulovska, and Mitchell A. Taylor. Gabor Frames of Totally Positive Functions: A Complete Characterization. *arXiv preprint arXiv:2608.04992*, 2026.
[19] Jaume de Dios Pont, Lukas Liehr, David Muñoz-Lahoz, Mitchell A. Taylor, and Pedro Tradacete. Banach lattices and phase retrieval: A case study for the use of AI in mathematics. *arXiv preprint arXiv:2608.07396*, 2026.
[20] Jaume de Dios Pont, Lukas Liehr, and Mitchell A. Taylor. Cantor measures with odd base do not admit Fourier frames. *arXiv preprint arXiv:2607.08656*, 2026.
[21] Jaume de Dios Pont, Lukas Liehr, and Mitchell A. Taylor. On the existence problem of regular Gabor frames. *arXiv preprint arXiv:2606.26052*, 2026.

Department of Mathematics, Bar-Ilan University, Ramat-Gan 5290002, Israel
Email address: lukas.liehr@biu.ac.il

Instituto de Ciencias Matemáticas. Universidad Autónoma de Madrid, Campus de Cantoblanco 28049 Madrid, Spain
Email address: david.munnozl@uam.es

Mathematical Institute, University of Oxford, Andrew Wiles Building, Radcliffe Observatory Quarter, Woodstock Road, Oxford, OX2 6GG, United Kingdom.
Email address: mitchell.taylor@maths.ox.ac.uk