The Heine-Borel Theorem for Metric Spaces

Project for the course “Topology in Lean”

Samuele Biscaro

03 January 2026

Abstract

In this report we describe how we formalized using Lean the Heine Borel theorem. Rather than relying on the general topology library of mathlib, we work with a custom topology library developed during the MAT740 course. The report tries to focus on the interaction between mathematical structure and proof handling in Lean.

§1 Introduction

In metric spaces, the *Heine-Borel theorem* is one of the most classical results, characterizing compactness in terms of completeness and total boundedness.

Theorem

A metric space X is compact if and only if it is complete and totally bounded.

This equivalence is often presented using sequences, relying on subsequence extraction and diagonal arguments, see for example the proof given in [1, Theorem 5.10.13]. While effective in informal mathematics, such proofs can become very hard to handle when formalized in a theorem prover.

An alternative approach is provided by filters, which generalize sequences, offering a uniform language for “closeness” and other topological notions. For example compactness can be expressed very simply as “every ultrafilter converges”.

The report is organized as follows. We begin by briefly introducing Lean and its interactive proof style, focusing on the aspects necessary to understand the formalization. We then describe briefly the proof strategy, discussing in the end the main difficulties.

§2 Theorem proving in Lean

In Lean, mathematical statements are expressed as types, and proofs are terms inhabiting these types. While the underlying logical foundations are rigorous, the user can interact with Lean through a high-level proof language that resembles structured mathematical reasoning.

§2.1 Proofs Structure and Tactics

A typical Lean proof proceeds by transforming a goal into simpler subgoals until all of them are discharged. At any point in a proof, Lean displays the current goal and a list of available hypotheses. The user applies tactics, which are commands that manipulate the goal or the hypothesis, such as

introducing assumptions, applying lemmas, or breaking down complex expressions. Lean checks each step for correctness, ensuring that no logical gaps remain.

Most proofs in this project are written using Lean's tactic mode, which supports a structured, step-by-step style. Common tactics include:

- introducing hypotheses (`intro`);
- applying known results (`apply`);
- rewriting expressions using equalities (`simp`, `rw`).

Rather than aiming for maximal formality, the proofs emphasize clarity and traceability, making the logical structure of the arguments visible in the code. For example we can see this in the following snippet of code from the file `HeineBorel.lean` (lines 11-54), we can almost directly read the proof from the code, thanks also to the help of the comments.

Code

```
theorem Compact_iff_Complete_and_totallyBounded {X : Type u} [MetricSpace X] :
    Compact (Set.univ : Set X) ↔ Complete X ∧ totallyBounded X := by
  -- Switch to the new definitions
  rw [← filterCompact_iff_Compact]
  rw [complete_iff_filterComplete]
  rw [totallyBounded_iff_filterTotallyBounded]
  constructor
  case mp ⇒
    ...
  case mpr ⇒
    -- Complete ∧ totallyBounded → Compact
    -- Take an ultrafilter F ...
    intro ⟨hC, hB⟩ F hF
    rw [← MyFilter.ultra_iff_prime] at hF
    -- ... F is a Cauchy filter since X is totally bounded ...
    specialize hB F hF
    -- ... but then F converges since X is complete
    obtain ⟨l, hl⟩ := hC F hB
    use l
```

§2.2 An example

Consider the following definition in Lean of a Cauchy sequence:

Code

```
def CauchySequ {X : Type u} [MetricSpace X] (x : ℕ → X) : Prop :=
    ∀ ε : ℝ, 0 < ε → ∃ N : ℕ, ∀ m n : ℕ, N ≤ m → N ≤ n → dist (x m) (x n) < ε
```

Also this definition closely follows the standard mathematical formulation, but its syntax reflects several important features of Lean. For example the expression `{X : Type u}` declares a type parameter that is implicit. The curly braces `{}` indicate that Lean should attempt to infer `X` automatically from context, rather than requiring it to be supplied explicitly each time the definition is used. The square brackets means more or less the same, and they are saying that Lean should infer a metric structure on `X`.

By contrast, the sequence itself is written as `(x : ℕ → X)` using parentheses. This indicates that x is an explicit argument to the definition: whenever `CauchySequ` is applied, the sequence `x` must be provided.

The colon `:` is used in Lean to declare the type of an expression. For example, `(x : ℕ → X)` declares that `x` is a function from natural numbers to `X`, and the final `: Prop` indicates that `CauchySequ x` returns a proposition.

The symbol `:=` is used to define an object by giving its value. In this case, it specifies what it means to be a Cauchy sequence. Informally, one can read this as "CauchySequ is defined to be the proposition that ...".

The right-hand side of the definition is a proposition built from universal quantifiers `∀`, existential quantifiers `∃`, and implications `→` (but could also make use of other logical connectors). In Lean, these are all constructors for types in the universe `Prop`. For example, an implication `P → Q` is itself a type whose inhabitants are functions taking proofs of `P` to proofs of `Q`. As a result, the entire definition is a precise object that Lean can manipulate and check for correctness. In practice, most of the time one can simply not concern about this, and just treat the formalized proposition as its informal counterpart. Even if, to be completely fair, having an understanding of this can shorten a lot many proof, but at the cost of decreasing readability.

§3 Proving the Heine–Borel Theorem in Lean

§3.1 Filters

Mathematically, the definition of a filter is the following.

Definition

A filter $\mathcal{F}$ on a set X is a nonempty collection of subsets of X satisfying the following properties:

1. $X \in \mathcal{F}$;
2. if $A \in \mathcal{F}$ and $A \subseteq B$, then also $B \in \mathcal{F}$;
3. if $A, B \in \mathcal{F}$, then also $A \cap B \in \mathcal{F}$.

Filters arise naturally in topology, for example given a sequence $\{x_n\}$ we are often interested in which sets are eventually visited, and the collection $\mathcal{F}$ of all of these sets is indeed a filter, the eventuality filter of $\{x_n\}$. Saying that $x_n \to l$ can be simply restated as $\mathcal{N}(l) \subseteq \mathcal{F}$, where $\mathcal{N}(l)$ is the set of all the neighborhoods of l. This abstraction turns out to be very powerful in topology. We defined them in lean using the following structure.

Code

```
structure Filter (X : Type u) where
  Sets : Set (Set X)
  univ_Sets : Set.univ ∈ Sets
  upward_Sets {A B} : A ∈ Sets → A ⊆ B → B ∈ Sets
  inter_Sets {A B} : A ∈ Sets → B ∈ Sets → A ∩ B ∈ Sets
```

The parameter `X : Type u` represents the underlying space. The field `Sets` is a set of subsets of X and the remaining fields correspond exactly to the filter axioms.

Remark

Each of these properties is stated as a function producing evidence that a certain set belongs to the filter, making them directly usable in proofs. For example if `hA` and `hB` are proofs of, respectively, `A ∈ F.Sets` and `B ∈ F.Sets`, then `F.inter_Sets hA hB` is a proof of `A ∩ B ∈ Sets`.

Within this framework convergence can be expressed in a very general way, note that in the following we can talk explicitly of convergence of a filter, not only of a sequence.

Code

```
def filter_convergence [Topology X] (F : Filter X) (x : X) : Prop :=
    neighborhoods x ≤ F.Sets
```

Compactness, completeness and total boundedness can be also expressed talking only about filters. These filter-based counterparts of classical notions are developed in the project files and form the foundation for the proof of Heine-Borel. We already know that compactness is equivalent to filter compactness.

Definition

A topological space X is *filter compact* if every ultrafilter $\mathcal{U}$ on X converges.

§3.2 Completeness

In the file `CompleteSpaces.lean` we introduce the definition of filter completeness.

Code

```
def CauchyFilt {X : Type u} [MetricSpace X] (F : MyFilter.Filter X) : Prop :=
    MyFilter.properFilter F ∧
      ∀ ε : ℝ, 0 < ε → ∃ A : Set X,
        A ∈ F ∧ ∀ (x : X) (y : X), x ∈ A → y ∈ A → dist x y < ε

def CauchyComplete (X : Type u) [MetricSpace X] : Prop :=
    ∀ (F : MyFilter.Filter X), CauchyFilt F → Convergent F
```

And, besides some other properties of Cauchy filters, we prove

Code

```
lemma complete_iff_filterComplete {X : Type u} [MetricSpace X] :
    Complete X ↔ filterComplete X
```

§3.3 Total boundedness

Recall that

Definition

A metric space X is *totally bounded* if for every $\varepsilon > 0$ there exist a finite collection of balls of radius ε covering X.

In the file `BoundedSpaces.lean` we define

Code

```
def filterTotallyBounded (X : Type u) [MetricSpace X] : Prop :=
    ∀ (F : MyFilter.Filter X), MyFilter.ultraFilter F → CauchyFilt F
```

and prove

Code

```
lemma totallyBounded_iff_filterTotallyBounded {X : Type u} [MetricSpace X] :
    totallyBounded X ↔ filterTotallyBounded X
```

The last two lemmas tell us that the filter based definitions are equivalent to the classical one, so the theorem that we want to prove becomes:

Theorem

Let X be a metric space, then X is filter compact if and only if X is filter complete and filter totally bounded.

The theorem itself turns out to be not difficult to prove thanks to the choice of the right setting (filters) and the right lemmas. The proof can be found in the file `Heine Borel.lean`.

§4 Conclusion

Formalizing the Heine–Borel theorem and the equivalence of compactness with completeness and total boundedness revealed several challenges. Many arguments that are typically summarized in informal proofs with phrases like "and so on" had to be made fully explicit in Lean. In particular, reasoning about finite subcovers, unions of indexed families of sets, and manipulations of ε-balls required careful bookkeeping of indices and types. A simple construction like extracting a Cauchy sequence from a Cauchy filter translated in more than 200 lines of code.

Using filters proved to be very natural. The main design choice in the project was to try to isolate general lemmas and prove them independently before applying them to the main theorems. This approach was intended to mirror standard mathematical practice, where auxiliary results are established once and then reused, rather than reproved in each argument. While this was not always possible, especially when dealing with highly specific Lean goals, it helped keep the structure of the proofs clearer and made the main arguments closer to their informal counterparts. When successful, this strategy reduced repetition and made the final proofs easier to read and maintain.

From this project, I learned how Lean enforces rigor at every step, making informal reasoning fully explicit. In the future, I would consider creating more general lemmas before starting a proof to reduce repetition (there is a lot of it in my project). Overall, the project illustrated the balance between formal

precision and mathematical intuition. Not being able to use the full power of mathlib showed the importance of carefully structuring definitions and intermediate lemmas to keep proofs manageable.

§5 AI Usage Declaration

I used AI (ChatGPT) to assist with phrasing, structuring the report, and suggesting approaches in Lean, such as handling types and identifying applicable lemmas. All formal proofs and code were independently verified and reflect my own reasoning and design choices.

Bibliography

[1] L. Ambrosio, C. Mantegazza, and F. Ricci, *Complementi di matematica*. 2021.

[2] L. d. 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*, Berlin, Heidelberg: Springer-Verlag, 2021, pp. 625–635. doi: 10.1007/978-3-030-79876-5_37.

[3] P. J. Cameron, *Sets, logic and categories*, 1998th ed. in Springer Undergraduate Mathematics Series. Guildford, England: Springer, 2012.

[4] J. L. Kelley, *General Topology*, 1st ed. in Graduate Texts in Mathematics. New York, NY: Springer, 1975.

[5] S. G. Krantz, *Handbook of logic and proof techniques for computer science*. New York, NY: Springer, 2012.

[6] J. Munkres, *Topology (Classic Version)*, 2nd ed. Upper Saddle River, NJ: Pearson, 2017.

[7] B. Poizat, *A course in model theory*, 2000th ed. in Universitext. New York, NY: Springer, 2000.

[8] G. Takeuti and W. M. Zaring, *Axiomatic set theory*, 1973rd ed. in Graduate Texts in Mathematics. New York, NY: Springer, 1973.