Formalising Mathematics - Coursework 3

Banach-Steinhaus Theorem and Extensions

CID: 01871147

March 30, 2023

Introduction

As the final part of my coursework I decided to formalise the Banach-Steinhaus theorem which I've learned about this year when taking the functional analysis course: MATH60030: Functional Analysis. Given that the final part of the coursework needed to cover a theorem from the 3rd year of undergraduate mathematics, I had to pick a theorem from Functional Analysis as it is the only other 3rd year mathematics module that I do apart from Formalising Mathematics. However that constraint wasn't a big issue for me as I thoroughly enjoyed taking the module during the Autumn Term 2022/2023.

What is more, I deliberately chose the past two projects so that the experience that I gained by working on those was very helpful when working on this project. My first project covered the intermediate value theorem and was a great way to gain experience proving theorems in mathematical analysis which mainly focus on manipulating inequalities and being able to estimate and bound expressions. The second project - Vitali's theorem allowed me to learn how to properly use the mathlib library and further developed my toolkit for proving claims in analysis.

This report documents the process of formalising Banach-Steinhaus theorem using the Lean programming language. It is a functional language which can also be used as an interactive theorem prover. I used Lean together with the mathlib library which contains many fundamental theorems and identities that are useful when building more complex proofs.

In what follows we'll first state and prove the theorem. Then I will explain the methodology that I followed in order to formalise the theorem. I will also describe the interesting challenges that needed to be overcome when formalising the proof. The last section contains conclusions which I made after working on the three sizeable projects in Lean and describe the future things that I will be working on.

Proof of the Banach-Steinhaus Theorem

Before I document the process of formalising, let us first state the theorem and observe the proof which I followed when translating the theorem into Lean.

Theorem (Banach-Steinhaus Theorem)**.** *Let X, Y be normed vector spaces over $\mathbb{R}$, assume that X is complete. Consider the following family of continuous linear operators: $(A_\lambda)_{\lambda\in\Lambda} \subset \mathcal{L}(X,Y)$ Assume that this family is bounded pointwise i.e. (with each upper bound being dependent on x)*

$$\forall x \in X \ \sup_{\lambda\in\Lambda} \|A_\lambda x\|_Y < \infty.$$

Then $(A_\lambda)_{\lambda\in\Lambda}$ is bounded uniformly i.e.

$$\sup_{\lambda\in\Lambda} \|A_\lambda\|_{\mathcal{L}(X,Y)} < \infty.$$

Before we prove the theorem let us first reformulate it in terms of upper bounds instead of requiring that suprema are finite. The reasoning behind doing this is that the statement of the theorem below is equivalent to the one above, however it is much more convenient to work with in Lean. That is because bounding a specific term is easier than working with the finiteness of suprema. In fact the two alternative ways of formulating the theorem are present in the mathlib library, however the original statement that can be seen above is considered to be an alternative definition.

Theorem (Banach-Steinhaus Theorem (Alternative formulation))**.** *Let X, Y be normed vector spaces over $\mathbb{R}$, assume that X is complete. Consider the following family of linear operators: $(A_\lambda)_{\lambda\in\Lambda} \subset \mathcal{L}(X,Y)$ Assume that this family is bounded pointwise i.e.*

$$\forall x \in X, \ \exists \ K_x \in \mathbb{R} \text{ such that } \forall \lambda \in \Lambda \ \|A_\lambda x\|_Y \leq K_x.$$

Then $(A_\lambda)_{\lambda\in\Lambda}$ is bounded uniformly i.e.

$$\exists K' \in \mathbb{R} \text{ such that } \forall \lambda \in \Lambda \ \|A_\lambda\|_{\mathcal{L}(X,Y)} \leq K'.$$

In the later part of the report we'll see how in Lean we can use the proof of the alternative version to easily deduce the theorem in it's initial form.

Before we do that however, let us consider the following lemma which is a consequence of the Baire Category Theorem and allows us to deduce uniform boundedness of a family of continuous functions which are bounded pointwise.

Lemma. *Let (X, d) be a complete metric space. Let $(f_\lambda)_{\lambda\in\Lambda}$ be a family of continuous functions*

$$f_\lambda : X \to \mathbb{R}$$

(note that f_λ don't need to be linear). If the family above is bounded pointwise i.e. :

$$\forall x \in X,\ \exists\ K_x \in \mathbb{R} \text{ such that } \forall \lambda \in \Lambda\ |f_\lambda(x)| \leq K_x.$$

Then there exists an open ball $B_r(x_0)$ with $x_0 \in X$ and $0 < r \in \mathbb{R}$. Such that $(f_\lambda)_{\lambda\in\Lambda}$ is uniformly bounded on it, that is:

$$\exists K' \in \mathbb{R} \text{ such that } \forall x \in B_r(x_0)\ \forall \lambda \in \Lambda\ |f_\lambda(x)| \leq K'.$$

We'll first prove the lemma by appeal to the Baire category theorem and then use it to prove the main statement of the theorem.

Let (X, d) be a complete metric space and $(f_\lambda)_{\lambda\in\Lambda}$ be a uniformly bounded family of continuous functions. Define a family of closed sets $(A_k)_{k\in\mathbb{N}}$, where

$$A_k = \{x \in X \mid \forall \lambda \in \Lambda\ |f_\lambda(x)| \leq k\}.$$

Now in order to apply Baire category theorem we need to show that all A_k are closed and that

$$X = \bigcup_{k=1}^{\infty} A_k. \tag{1}$$

First let us prove that all sets in that family are closed. Note that for all $\lambda \in \Lambda$ the map $x \mapsto |f_\lambda(x)|$ is continuous by composition. That is because the absolute value function is continuous and we have also assumed that all f_λ are continuous. Therefore if we rewrite each of A_k as follows, we'll get:

$$A_k = \{x \in X \mid \forall \lambda \in \Lambda\ |f_\lambda(x)| \leq k\} = \bigcap_{\lambda\in\Lambda} \{x \in X \mid |f_\lambda(x)| \leq k\}.$$

Now if we denote the map $g_\lambda := x \mapsto |f_\lambda(x)|$, then each of the sets of the form $\{x \in X \mid |f_\lambda(x)| \leq k\}$ is the pre-image of g_λ of $[0, k]$. Consequently it is the pre-image of a continuous function of a closed set, hence it is closed. Therefore, each of the A_k is an intersection of closed sets, and thus it is also closed. Hence we have established that all A_k are closed. Now in order to show that their union covers all of X, we only need to show the forward inclusion i.e.

$$X \subseteq \bigcup_{k=1}^{\infty} A_k. \tag{2}$$

and the equality of the two sets above follows immediately from antisymmetry of set inclusion, because each of A_k is already a subset of X and therefore the backward inclusion follows from the fact that a union of subsets is still a subset.

In order to show the forward direction, let $x \in X$ be arbitrary. Now by our assumption of pointwise boundedness of the collection $(f_\lambda)_{\lambda\in\Lambda}$, for that particular x we can find a K_x such that

$$\forall \lambda \in \Lambda \; |f_\lambda(x)| \leq K_x.$$

But now we can pick a sufficiently large $k \in \mathbb{N}$ such that $k \geq K_x$ and then we deduce:

$$\forall \lambda \in \Lambda \; |f_\lambda(x)| \leq K_x \leq k.$$

And so now it follows that $x \in \bigcap_{\lambda\in\Lambda}\{x \in X \mid |f_\lambda(x)| \leq k\} = A_k$. Which in turn implies that $x \in \bigcup_{k=1}^{\infty} A_k$. And since x was arbitrary, we can deduce that (2) holds which combined with antisymmetry and the fact that the union is a subset of X yields equation (1).

Now since X is nonempty and complete, and given that $X = \bigcup_{k=1}^{\infty} A_k$, where each A_k is closed, by Baire category theorem we deduce that there exists a $k_0 \in \mathbb{N}$ such that the interior of A_{k_0} is non-empty. Hence, because of the fact that $\text{int}(A_{k_0})$ is open and non-empty, we can pick x_0 inside of it and $r > 0$ such that

$$B_r(x_0) \subseteq A_{k_0}.$$

Thus, by definition of A_{k_0} we deduce that:

$$\forall x \in B_r(x_0) \; \forall \lambda \in \Lambda \; |f_\lambda(x)| \leq k_0.$$

And so we have found our $K' := k_0$ whose existence we needed to show to prove the lemma. ■

Having proved the lemma, we can now move on to proving the main theorem. Let X, Y be normed spaces, assume X is complete. Let $(A_\lambda)_{\lambda\in\Lambda} \subset \mathcal{L}(X, Y)$ be a family of linear operators which is pointwise bounded. In order to prove that $(A_\lambda)_{\lambda\in\Lambda}$ is in fact uniformly bounded, let us first define a family of continuous functions to which we can then apply the previous lemma. For $\lambda \in \Lambda$ define a map $f_\lambda : X \to \mathbb{R}$ as

$$\forall x \in X \; f_\lambda(x) = \|A_\lambda(x)\|_Y.$$

Now note that the norm $y \mapsto \|y\|_Y$ is continuous and by our assumption A_λ is a continuous linear map. Therefore, by composition we deduce that for all $\lambda \in \Lambda$ f_λ is continuous and therefore we can apply the lemma to get a ball $B = B_r(x_0)$ with $r > 0$ and $x_0 \in X$, such that there exists $K' \in \mathbb{R}$ satisfying:

$$\forall x \in B_r(x_0) \; \forall \lambda \in \Lambda \; |f_\lambda(x)| \leq K'. \tag{3}$$

Our objective is to show that

$$\exists K'' \in \mathbb{R} \text{ such that } \forall \lambda \in \Lambda \ \|A_\lambda\|_{\mathcal{L}(X,Y)} \leq K''.$$

Now note that we are able to control the operator norm in a following way:

$$\forall \lambda \in \Lambda \ \forall x \in X \ \|A_\lambda x\|_Y \leq K''\|x\|_X \implies \|A_\lambda\|_{\mathcal{L}(X,Y)} \leq K''.$$

Now fix $K'' := \dfrac{4K'}{r}$ where K' is the constant bounding uniformly $|f_\lambda(s)|$ that we have obtained from the lemma. We want to show that

$$\forall \lambda \in \Lambda, \ \|A_\lambda\|_{\mathcal{L}(X,Y)} \leq K''.$$

In order to do this, let $\lambda \in \Lambda$ arbitrary and consider the following:

$$\forall x \in X \ \|A_\lambda x\|_Y \leq K''\|x\|_X \implies \|A_\lambda\|_{\mathcal{L}(X,Y)} \leq K''.$$

Hence if we need to take $x \in X$ arbitrary and impose a bound

$$\|A_\lambda x\|_Y \leq K''\|x\|_X.$$

In order to achieve this, consider the following chain of manipulations. First note that by adding and subtracting x_0 and multiplying by a scaling factor and its inverse, we get:

$$x = \frac{2\|x\|}{r}\left(x_0 + \frac{r}{2}\frac{x}{\|x\|} - x_0\right).$$

It is important to note that in order for this to work we need X to be a normed vector space over $\mathbb{R}$ (as $r \in \mathbb{R}$). And that is why in my Lean implementation I took X, Y to be vector spaces over $\mathbb{R}$ as opposed to arbitrary fields $\mathbb{F}_1, \mathbb{F}_2$, Now if we substitute the expression above into $\|A_\lambda x\|$ we obtain:

$$\|A_\lambda x\| = \left\|A_\lambda\left(\frac{2\|x\|}{r}\left(x_0 + \frac{r}{2}\frac{x}{\|x\|} - x_0\right)\right)\right\|.$$

By linearity of A_λ we can rearrange the expression above using the following properties: $A_\lambda(\alpha x) = \alpha A_\lambda(x)$ and $A_\lambda(x - y) = A_\lambda(x) - A_\lambda(y)$. Hence, we obtain

$$\left\|A_\lambda\left(\frac{2\|x\|}{r}\left(x_0 + \frac{r}{2}\frac{x}{\|x\|} - x_0\right)\right)\right\|_Y = \left\|\frac{2\|x\|}{r}\left(A_\lambda\left(x_0 + \frac{r}{2}\frac{x}{\|x\|}\right) - A_\lambda(x_0)\right)\right\|_Y.$$

Now, by absolute homogeneity of $\|\cdot\|_Y$, we get:

$$\left\|\frac{2\|x\|}{r}\left(A_\lambda\left(x_0 + \frac{r}{2}\frac{x}{\|x\|}\right) - A_\lambda(x_0)\right)\right\|_Y = \left|\frac{2\|x\|}{r}\right|\left\|\left(A_\lambda\left(x_0 + \frac{r}{2}\frac{x}{\|x\|}\right) - A_\lambda(x_0)\right)\right\|_Y =$$

Observe that both $r, \|x\|$ are non-negative, therefore we may drop the absolute value above and then apply the triangle inequality:

$$\leq \frac{2\|x\|}{r}\left(\left|\left|A_\lambda\left(x_0 + \frac{r}{2}\frac{x}{\|x\|}\right)\right|\right|_Y + ||A_\lambda(x_0)||_Y\right). \tag{4}$$

Now note that both $x_0 + \frac{r}{2}\frac{x}{\|x\|}$ and x_0 belong to $B_r(x_0)$. In case of x_0 it is the case because x_0 is the centre of the ball, whereas when it comes to the first point, let us consider the following to see that it is indeed contained in the ball B:

$$x_0 + \frac{r}{2}\frac{x}{\|x\|} \in B_r(x_0) \iff \left|\left|x_0 - \left(x_0 + \frac{r}{2}\frac{x}{\|x\|}\right)\right|\right| < r.$$

Simplifying the left-hand side of the iff above allows us to see:

$$\left|\left|x_0 - \left(x_0 + \frac{r}{2}\frac{x}{\|x\|}\right)\right|\right| = \left|\left|\frac{r}{2}\frac{x}{\|x\|}\right|\right| = \frac{r}{2}\frac{\|x\|}{\|x\|} = \frac{r}{2} < r.$$

Hence, we deduce that both of the points above belong to the open ball B, and thus we can conclude:

$$\left|\left|A_\lambda\left(x_0 + \frac{r}{2}\frac{x}{\|x\|}\right)\right|\right|_Y = \left|f_\lambda\left(x_0 + \frac{r}{2}\frac{x}{\|x\|}\right)\right| \leq K' \text{ and } ||A_\lambda(x_0)||_Y = |f_\lambda(x_0)| \leq K'.$$

That is because we can use the definition of f_λ in reverse and then apply (3) to impose those bounds.

We can substitute the above findings back into the inequality (4) to continue estimating the bound:

$$\leq \frac{2\|x\|}{r}\left(\left|\left|A_\lambda\left(x_0 + \frac{r}{2}\frac{x}{\|x\|}\right)\right|\right|_Y + ||A_\lambda(x_0)||_Y\right) \leq \frac{2\|x\|}{r}\left(K' + K'\right) = \frac{4K'}{r}\|x\| = K''\|x\|.$$

Since λ and x were arbitrary and our choice of K'' didn't depend on either of them, we can conclude that:

$$\forall \lambda \in \Lambda\ \forall x \in X\ \|A_\lambda x\|_Y \leq K''\|x\|_X.$$

Which in turn implies

$$\forall \lambda \in \Lambda\ \|A_\lambda\|_{\mathcal{L}(X,Y)} \leq K''.$$

And so the family $(A_\lambda)_{\lambda\in\Lambda}$ is uniformly bounded and that concludes the proof. ■

The final step of the proof was to deduce the initial way of formulating the theorem from the alternative one which uses those bounding constants K_x, K'. Informally, it isn't difficult to observe that the two ways of phrasing the theorem are equivalent. However, in Lean showing this requires a certain degree of care and doesn't follow immediately. In the next section I will explain the process of formalising the theorem and discuss unexpected observations that I made while translating the proof above into Lean.

The Process of Formalising

Having already completed two sizeable projects in Lean, I had quite a strong background knowledge and hands-on experience using the programming language in conjunction with the mathlib library. When I started working on the third project, I already had a fixed approach which I was following when formalising any given proof. What I would normally do was first write down the proof by hand on paper, then move on to typesetting it into latex for the report. During that process I was always trying to formulate all transitions in the proof as explicitly as possible so that it is easier to translate everything into Lean later on. After that I would make an attempt at formalising the argument by splitting it into smaller lemmas where applicable. When I encountered a more difficult part of the proof which didn't translate into Lean very well, I would try to reformulate the written argument. After that I would normally refactor the code to remove redundancies and possibly introduce additional notation to make the code more similar to an actual written argument.

However, when working on the final project I decided to follow a different approach. This was because after I received the feedback for my second project, it included a very thorough code review of my submission which indicated the main problems with my code and suggested ways to fix them and refactor the code so that it is more professional and could possibly be included in an actual library. I found this advice very helpful and at this point I wanted to thank Prof. Kevin Buzzard for devoting so much time to analysing my code and suggesting ways to improve it. I must admit that this was by far the most useful piece of advice thich I have received throughout the whole duration of the module. The reason being that it was directly referring to my code, so it allowed me to observe inefficient patterns in my cod and helped me change my habits to closer resemble professional workflow. Below you can see a list of the most common problems that one could find in my submission for the second part of the coursework:

1. Introducing redundant one-liner lemmas which could be inlined in the main body of the proof.
2. Using inefficient `simp` tactic in the middle of the proof instead of only closing the goals with it.
3. Adding redundant `dsimp` or `rw` lines to only change the visual output in the goal state.
4. Introducing unnecessary intermediate hypotheses for simple claims which are in the library (e.g. `one_ne_top`) instead of using function composition in the `exact` tactic to close the goal.
5. Not supplying immediate arguments into `apply` or `rw` tactics which causes multiple goals to appear and requires indenting the code.

In the following section I will show how I applied these suggestions when writing the code for this project.

When writing the proof for the first time, I have changed my approach towards adding auxiliary lemmas. In the past what I would do is sometimes when the proof was getting a bit too cluttered and I wasn't able to easily read it as if it was hand-written, I would introduce a new lemma with a descriptive name to "hide complexity". This time however, I prioritised optimising the code for performance and so I only introduced an additional lemma if it was some generalised claim or a seemingly obvious algebraic manipulation which was tricky to prove inline using Lean. An example of that can be seen below.

```
theorem uniform_bounded_of_cont_of_bounded_pointwise {ι : Type*} [complete_space X]
{f : ι → (X → ℝ)} (h_cont : ∀ (i : ι), continuous (f i)) (h : ∀ x, ∃ K, ∀ (i : ι), ∥f i x∥ ≤ K) :
∃ (x₀ : X) (r : ℝ), 0 < r ∧  ∃ K' ≥ 0, ∀ (i : ι) x ∈ metric.ball x₀ r, ∥f i x∥ ≤ K' :=
begin ... end
```

Listing 1: Lemma which is a generally applicable theorem.

The theorem above can be contrasted with the simple lemma below that cannot be solved using `simp`.

```
lemma scale_add_zero_rescale {x : X} (x₀ : X) {r : ℝ} (hx : x ≠ 0) (hr : r ≠ 0) :
x = ((2 * ∥x∥) / r) • (x₀ + ((r / (2 * ∥x∥)) • x) - x₀) :=
by conv begin
  to_rhs,
  find (x₀ + _ - x₀) { -- RHS =   (2 * ∥x∥) / r) • (x₀ + ((r / (2 * ∥x∥)) • x) - x₀) =
    rw [add_comm,      --         (2 * ∥x∥) / r) • (((r / (2 * ∥x∥)) • x) + x₀ - x₀) =
        add_sub_assoc, --       (2 * ∥x∥) / r) • (((r / (2 * ∥x∥)) • x) + (x₀ - x₀)) =
        sub_self,      --              (2 * ∥x∥) / r) • (((r / (2 * ∥x∥)) • x) + 0) =
        add_zero], },  --                  (2 * ∥x∥) / r) • (((r / (2 * ∥x∥)) • x)) =
  rw [smul_smul,       --                  (((2 * ∥x∥) / r) * (r / (2 * ∥x∥))) • x  =
      div_eq_mul_inv,  --               (((2 * ∥x∥) * r⁻¹) * (r * (2 * ∥x∥)⁻¹)) • x =
      div_eq_mul_inv,  --               (((2 * ∥x∥) * r⁻¹) * (r * (2 * ∥x∥)⁻¹)) • x =
      mul_assoc],      --               ((2 * ∥x∥) * (r⁻¹ * (r * (2 * ∥x∥)⁻¹))) • x =
  conv { congr, congr, skip,    --                                                  =
         rw [← mul_assoc,       --       ((2 * ∥x∥) * ((r⁻¹ * r) * (2 * ∥x∥)⁻¹)) • x =
             inv_mul_cancel hr, --               ((2 * ∥x∥) * (1 * (2 * ∥x∥)⁻¹)) • x =
             one_mul] },        --                    ((2 * ∥x∥) * (2 * ∥x∥)⁻¹) • x =
  conv { congr,                 --                                          1 • x =
         rw mul_inv_cancel (mul_ne_zero two_ne_zero (norm_ne_zero_iff.mpr hx)), },
  rw one_smul,                  --                                              = x
end
```

Listing 2: Lemma allowing to perform a specific manipulation.

The Listing 2 above also illustrates a very insightful learning experience. When working on this final project I've decided to start incorporating some more of the advanced Lean language features into my proofs. In that listing, I have exclusively used the `conv` mode which is a very powerful tool where we need to perform very fine-grained rewrites on a very complex expression.

By considering the addition of each new lemma more thoroughly, I was able to address the first issue on the list of suggestions. The suggestions 2. and 3. involved not using computationally-expensive `simp` tactic in the middle of a proof and removing redundant `dsimp` and `rw` lines which only affected the way the goal is displayed (effectively the ones relying on definitional equality). This improvement was achieved by using the `squeeze_simp` tactic which shows the theorems that it uses to perform simplification and then replacing the invocation of that tactic with a targeted rewrite. Furthermore, I was trying to exploit the definitional equality wherever possible to avoid using the redundant rewrites. The reason behind those optimisations is that when the code is compiled (i.e. the proof is effectively getting checked) it runs much faster if the proof is comprised of fewer statements and if there aren't many usages of the `simp` tactic, whose biggest drawback is that it tries to search through all of the over 10000 lemmas present in mathlib to perform simplification.

In order to further remove redundancies and optimise the code I used the following iterative approach. At the start I wrote the "first draft" of the proof in Lean, using as many steps as I needed to easily understand the proof from the code. A snippet of that first draft can be found below.

```
have hA_closed : ∀ n : ℕ, is_closed (A n),
  { intro n,
    apply is_closed_Inter,
    intro i,
    apply is_closed_le,
    { exact continuous.norm (h_cont i), },
    { exact continuous_const, }, },
```

This snippet proves all sets in a particular collections are closed. In its nature, the argument is fairly mechanical, it involves taking an arbitrary set, applying lemmas and hypotheses and concluding the proof. After refactoring it, I managed to make it much shorter by using lambda functions:

```
have hA_closed : ∀ n : ℕ, is_closed (A n), from
λ n, is_closed_Inter (λ i, is_closed_le (continuous.norm (h_cont i)) continuous_const),
```

Listing 3: Code after refactoring.

My approach to refactoring the code was as follows. After I managed to compile the proof and see that there were no errors, I went through the code line-by-line and was trying to make it as short as possible using function compositions and lambda functions instead of the `intro` tactic.

It is important to understand the trade-off when performing refactorings similar to the one described above. On one hand, the code becomes more performant because it uses less steps. However on the other hand, the code readability is impacted which might make it less accessible to people who want to learn Lean. When it comes to the improved performance, I haven't experienced substantial benefits in terms of the compilation times of my code after refactoring it. However, I suspect that once the codebase grows larger (for example the mathlib one) every single line of code matters when we need to compile it.

In my opinion, the readability aspect of the code requires thorough consideration. In my previous projects my goal was to write code which is visually as close as possible to the informal hand-written argument. That's why I often introduced new notation which seemed more convenient to me. Even though I used shortcuts such as `intros` or `rcases` to make the code shorter, those tactics are used to let multiple variables be arbitrary or extract witnesses from existentially quantified statements which is almost always done in one step in an informal argument. Thus the readability wasn't negatively affected. However I didn't push the function composition to its limits as I felt like it makes the code too obscure.

I remember when I first started looking at the professional code in the mathilb library, I often struggled to understand it because of its extensive use of advanced tactics modes and function composition. However, over the course of this project I noticed that if one focuses on incorporating those optimisations, they require less and less cognitive effort. It happened to a point where I'm now able to refactor even some of the more complicated statements into one-line proofs. I must admit that it is very satisfying to be able to get a particular function composition just right and see that all goals have been accomplished. The only drawback that I'm worried about is that from the perspective of code maintainability and code reviews, sometimes the refactored code might look incomprehensible.

Methodology

Challenges

Language Extensions

Conclusions

Future Work