Formalising Mathematics - Coursework 3

Banach-Steinhaus Theorem

CID: 01871147

March 31, 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 extensions 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 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 Lean, dealing with a supremum of a norm requires working with extended non-negative real numbers and this requires handling type coercions appropriately. 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 $\mathrm{int}(A_{k_0})$ 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

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

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

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

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

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

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

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

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

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

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 that's true 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. That is because we need to correctly handle coercions between three different types : `real` `nnreal` and `ennreal`. 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 is that it was directly referring to my code, so it allowed me to observe inefficient patterns in my code 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 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 a general 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. After I identified the correct lemma that needed to be used, I replaced the invocation of that `squeeze_simp` 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 just by looking at 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 the following. 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. Below you can see another example of that refactoring process, it shows the usage of a complex `exact` tactic at the end of the proof which closes the goal by performing many logical transitions in one step.

```
have hA_union : (⋃ n : ℕ, (A n)) = univ, -- Before the refactoring.
{ ext,
  split,
  { intro x, triv, },
  { intros hx,
    cases h x with k hk,
    obtain ⟨k', hk'⟩ := exists_nat_ge k,
    rw mem_Union,
    use k',
    rw hA,
    rw mem_Inter,
    intro i,
    specialize hk i,
    exact le_trans hk hk', },},
  have hA_union : (⋃ n : ℕ, (A n)) = univ, -- After the refactoring.
{ ext,                                 -- Proving equality by set extensionality.
  refine ⟨λ hx, mem_univ x, _⟩,        -- First deal with the trivial forward inclusion.
  cases h x with k hk,                 -- Find the bound which works for x,
  obtain ⟨k', hk'⟩ := exists_nat_ge k, -- Find a natural number greater than the bound k.
  -- Conclude that x is a member of the union because it belongs to Ak'
  exact λ hx, mem_Union.mpr ⟨k', mem_Inter.mpr (λ i, le_trans (hk i)  hk')⟩, },
```

Listing 4: Refactoring which combines many steps into one usage of `exact`.

As you can see above, the refactored code is much shorter, however it's arguably less readable as some steps arent that explicit. In the line 21 in the listing above one can see an application of the `exact` tactic which uses lambda functions and function composition to close the goal by executing multiple steps at once. Because of its impact on readability, after my refactorings I always inserted descriptive comments to document what is happening in the proof.

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 use the function composition too often as I felt like it makes the code too obscure.

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 got 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.

One possible danger that I identified is that once the code is refactored and condensed to the shortest form possible, in order to understand it, one has to know the API of all of the functions which are used in the complicated refactored expression. It is much harder to get a general understanding of what is happening in the proof. A following example can illustrate that:

```
have supr_le_K' : (⨆ i, ↑‖f i‖₊) ≤ ennreal.of_real(K'),
 { refine supr_le _,
   intro i,
   specialize hK' i,
   rw ennreal.of_real_eq_coe_nnreal,
   { rw coe_le_coe,
     exact hK', },
   { exact le_trans (norm_nonneg (f i)) hK', }, },
```

Listing 5: Proof that a supremum of a set of operator norms is bounded by K'.

The code above explicitly uses the lemma `supr_le` which tells us that a supremum over an indexed set is less than or equal to a bounding constant if each element of that set is less than or equal to the constant. After that we take an arbitrary index, use some hypothesis `hK'` on it and then deal with type coercions between real numbers and extended non-negative real numbers. After refactoring that argument, I managed to express it in a following way:

```
have supr_le_K' : (⨆ i, ↑‖f i‖₊) ≤ ennreal.of_real(K'),
   from supr_le (λ i, le_of_eq_of_le'
       (eq.symm (ennreal.of_real_eq_coe_nnreal (le_trans (norm_nonneg (f i)) (hK' i))))
       (coe_le_coe.mpr (hK' i))),
```

Listing 6: Proof from Listing 5 after refactoring.

Note that even though the argument above spans over three lines of code, it was done only to improve readability and it is in fact a single-line proof. Having contrasted the two listings above, I think that in terms of refactoring the code the programmer should strive towards a middle-ground between writing verbose and explicit code and using clever optimisations. My aim throughout this project was to write code of a similar level of complexity to the one in the mathlib library. Because of this I was trying to make it as short as possible at a cost of sacrificing readability for the inexperienced users.

Challenges

One of the biggest challenges that I faced when working on this final project was connected with the nature of the proof that I was formalising. The main idea of the proof is to be able to bound $\|f_i(x)\|$ by rewriting x in a clever way (see Listing 2) and then using the linearity of f_i and absolute homogeneity of the norm to split the expression into separate terms and bound each one of them individually.

The difficulty with that approach is that performing numerous simple manipulations of expressions in Lean can be tedious at times. That is because whenever we want to apply some simplification lemma, the state of the goal needs to precisely match the hypothesis of that lemma (see `mul_le_mul_left` explained later). On top of that, sometimes we deal with expressions containing operations which are incompatible with each other in Lean, however in an informal argument we don't really notice a difference between them. An example of that would be regular multiplication between two real numbers and scalar multiplication. In Lean those are two completely distinct concepts and we need to use mathilb lemmas to perform manipulations on them. An example of that can be seen in the listing below where I needed to prove an inequality using properties of the norm and linearity of a function f.

```
lemma linear_manipulation (f : X→SL[ring_hom.id ℝ] Y) (x₀ x : X) {β : ℝ} (hβ : β ≠ 0) :
‖ f (β • (x₀ + (β⁻¹ • x) - x₀)) ‖ ≤ |β| * (‖ f (x₀ + β⁻¹ • x) ‖ + ‖ f x₀ ‖) :=
begin
  rw [continuous_linear_map.map_smulₛₗ f  β,
      norm_smul,
      ring_hom.id_apply,
      real.norm_eq_abs,
      continuous_linear_map.map_sub,
      mul_le_mul_left (abs_pos.mpr hβ)],
  exact norm_sub_le (f(x₀ + (β⁻¹ • x))) (f x₀),
end
```

Listing 7: Using linearity and properties of the norm.

As you can see the proof almost exclusively consists of `rw` statements followed by one application of the triangle inequality for norms. Despite the simplicity of the claim that we are trying to prove, it requires a certain amount of care to transform the goal state into precisely the expression that we require.

Another difficulty that I had was also connected with those kinds of manipulations, in the main proof near the end of the argument I needed to perform cancellation of positive constants from both sides of an inequality (e.g. $2 * x \leq 2 * y \iff x \leq y$). However the issue I had was that in order to perform that simplification, I needed to get the cancelled term to be precisely the first term on the left of each side of the inequality. Moreover, everything else needed to be enclosed in parentheses so that the lemma **mul_le_mul_left** could be applied (i.e. the required form was $2 * (\_) \leq 2 * (\_)$). It turned out to be surprisingly difficult to perform that kind of rearrangement. In order to do it, I needed to learn how to use the `conv` mode.

The `conv` mode is a very powerful tool to rewrite multiply nested expressions. In that mode, the programmer has access to all rewrite lemmas as well as a new set of tactics which allow for skipping parts of simplified expressions. After the parts that aren't modified are skipped, one can target the sub-expressions which need to be modified by using e.g. commutativity or associativity of multiplication.

Using this mode I was perform fine-grained manipulations on the goal state to transform it into a shape which could be accepted into the lemma. It allowed for working on both sides of the inequality separately and thanks to the `find` tactic I was able to rewrite only the sub-expressions of a certain desired form.

The code snippet illustrating this manipulation can be seen below:

```
    conv
    { conv
      { to_rhs,                            -- Convert RHS:  2 * 2 * K' / r * ‖x‖ =
        rw [mul_assoc, div_eq_mul_inv],         --     2 * (2 * K') * r⁻¹ * ‖x‖ =
        rw [mul_comm, ← mul_assoc, mul_comm],   -- r⁻¹ * (‖x‖ * (2 * (2 * K'))) =
        rw [← mul_assoc, ← mul_assoc],          --     r⁻¹ * ‖x‖ * 2 * (2 * K') =
        find (r⁻¹ * _) { rw mul_comm, },        --     ‖x‖ * r⁻¹ * 2 * (2 * K') =
        find (‖x‖ * r⁻¹ * _) { rw mul_comm, },  --   2 * (‖x‖ * r⁻¹) * (2 * K') =
        rw ← div_eq_mul_inv, },                 --       2 * (‖x‖ / r) * (2 * K')
      conv
      { to_lhs, congr,
        rw mul_div_assoc, }, -- Obtains: 2 * (‖x‖ / r) * (‖(f i) ...) on the LHS.
      rw [mul_assoc, mul_assoc, inv_div], },
```

Listing 8: Rearranging terms using the conv mode.

It is important to note that this form was achieved after properly refactoring the code and making it as short as possible while maintaining readability by adding comments. The initial version didn't use the conv mode and was over 2 times longer. This was because it involved numerous targeted rewrites such as the one below:

```
rw div_eq_inv_mul (2 * 2 * K') r,
```

Listing 9: Targeted rewrite with specified arguments.

The problem with that approach was that it made the code much longer and it wasn't obvious that the step was meant to only alter the shape of the target expression. After incorporating the conv mode the code is much cleaner because there is a very distinct separation between the actual logical transitions of the proof and those simple conversions of the shape of the expressions as the one discussed above.

Language Extensions

With respect to the advanced language features, this project is much simpler than the previous ones. Because of the extensive use of function compositions and optimising the code as much as possible, there weren't many places where I could introduce new language notation. This was caused by the refactoring process which transformed the code into a state where it mainly referred to the library functions and composed them together to get to the desired goal in just one step.

As an example we can consider the proof of the second version of the Banach-Steinhaus theorem. The second version states the claim in terms of suprema and I decided to prove it as a corollary of the first version because dealing with the concepts such as the supremum requires us to work inside of `ennreal` numbers and thus it inevitably leads to complicated type coercions. The closing statement of that theorem illustrates how there is no space left for the new notation after the code has been properly refactored.

```
  exact lt_of_le_of_lt
    (supr_le (λ i, le_of_eq_of_le'
        (eq.symm (ennreal.of_real_eq_coe_nnreal (le_trans (norm_nonneg (f i)) (hK' i))))
        (coe_le_coe.mpr (hK' i))))
    (ennreal.coe_lt_top),
```

Listing 10: Closing statement of the corollary to Banach-Steinhaus theorem.

As you can see above, the code is complex enough and I would argue that new notation would only make it harder to understand. The main motivation that I had when introducing the new notation in my past projects was to make the code shorter and more readable. This time, since writing optimised professional-looking code was a priority, some the readability from the perspective of inexperienced users has already been sacrificed. If I were to add new notation, all it could do is confuse the experienced language users and make the code more obscure. It also wasn't possible for the new notation to make the code shorter because it was already as short as possible.

The only language extension that I introduced was one for open balls in metric spaces. Personally I prefer to use $B_r(x)$ to denote an open ball centered at x of radius r. In mathlib however, we need to write `metric.ball x r` which is readable, but I always struggled to remember which argument is the radius and which one is the centre. Because of this, I decided to introduce the following alternative notation:

```
reserve prefix `B_{`:65
reserve infix `}`:65
notation `B_{` r `}` x := metric.ball x r
-- Example usage
lemma point_in_ball {r : ℝ} {x₀ x : X} (hr : (0 : ℝ) < r) (hx : x ≠ 0) :
(x₀ + (r / ((2 : ℝ) * ‖x‖)) • x) ∈ B_{r}(x₀)
:= begin ... end
```

Listing 11: Alternative notation for balls in metric spaces.

As it can be seen above, because of the new notation we can now read the declaration of the lemma more as if it was written using actual mathematical notation.

Conclusions and Future Extensions

The final project was a great opportunity to get more hands-on experience writing professional Lean code. The feedback that I received after my second project was a very useful starting point to guide the work on this project. Initially I planned to implement the Banach-Steinhaus theorem and some of it's generalisations which aren't yet implemented in the mathilb library.

After implementing the first version of the code, I reached out to the members of the mathlib community to see how does the state of the library look like when it comes to implementing those generalisations. I got a lot useful insights and suggestions as to which parts of the library I could use to implement the extensions. However, I also learned that one of the professional contributors to the library is already in progress of implementing that particular generalisation which I was planning to work on. As a result I've decided to pivot and focus on optimising the regular version of the Banach-Steinhaus theorem together with its corollary stated in terms of finite suprema.

Ultimately I think it was a good decision as it turned out that producing professional-level code is more time consuming than I anticipated. This was because refactoring often involves searching through the library to find precisely the right version of each lemma so that they can be composed into a single chain of function applications without introducing redundant rewrites in between steps. For instance, in the past when I needed to prove something using the transitivity of $\leq$, I would always use the mathilb lemma `le_trans`. In this project however I've learned that there are many more versions of that lemma which are applicable to different scenarios (e.g. $a = b \leq c \implies a \leq c$ can be shown using the `le_of_eq_of_le` lemma). Those small optimisations are time consuming, but eventually they lead to the code in which the amount of redundancies has been minimised.

To conclude, this project gave me a very useful insight into developing professional Lean code. I was able to gain appreciation of the trade-off between the code readability and performance. I was also able to reflect on the impact of introducing new notation on the general readability of the code. The theorem that I've proven helped me revise some of the concepts from functional analysis leading up to my final exam in May. Interacting with the mathilb community was also a great opportunity to learn how to contribute to open-source libraries. In the future, I'm planning to continue using Lean as a way to revise functional analysis and find a theorem which I've learned about this year and isn't yet implemented in mathilb. That way I could practise writing professional-level lean code and contribute to the library.