Formalising Mathematics - Coursework 2

Vitali's Theorem

CID: 01871147

March 7, 2023

Introduction

As the second part of the coursework I've decided to formalise the Vitali's theorem which is taught as a part of the second year measure theory course: MATH50006: Lebesgue Measure and Integration. I made this decision because I thoroughly enjoyed learning about measure theory in my second year. On top of that, formalising measure theoretic concepts was somewhat similar to the process of formalising theorems from Analysis I covered in the first part of my coursework. The similarity was in the structure of the proof; Vitali's theorem requires using $\varepsilon - \delta$ reasoning which I was familiar with after completing my first coursework. However, measure theory is also sufficiently different from analysis to make it very interesting to work with, provide an optimal level of challenge and expand my knowledge of the Lean programming language.

This report documents the process of formalising Vitali's 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.

After formalising two large proofs using this toolkit, I learned to appreciate that the mathlib library shouldn't be thought of as merely an extension of the basic capabilities of the language. Instead, mathlib provides all of the API's with which are needed when proving mathematical theorems. Without this library, proving complicated statements such as the Vitali's theorem would require us to first define all of the measure theory leading up to the statement of the theorem which would have taken thousands of lines of code.

Proof of the Vitali's 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 (Vitali's Theorem)**.** *Let X be a measurable space and μ be a measure on it. Assume $\mu(X) < \infty$. Let $f, f_n : X \to \mathbb{R}, n \in \mathbb{N}$ be functions all in $\mathcal{L}^1(\mu)$. Then the following are equivalent:*

(T1) $f_n \xrightarrow{\mu} f$ and $\mathcal{F} = \{f_n : n \in \mathbb{N}\}$ has uniformly absolutely continuous integrals

(T2) $\int |f_n - f| \, \mathrm{d}\mu \to 0$, as $n \to \infty$

It is also important to note that a family $\mathcal{F} \subset \mathcal{L}^1(\mu)$ of functions has uniformly absolutely continuous integrals if

$$\forall \varepsilon > 0, \exists \delta > 0, \text{ such that } \forall f \in \mathcal{F}, \forall A \in \mathcal{A}, \text{ we have } \mu(A) < \delta \implies \int_A |f| \, \mathrm{d}\mu < \varepsilon. \tag{1}$$

Where $\mathcal{A}$ is the sigma algebra on X.

Let us start by proving the backwards direction of the theorem i.e. that (T2) $\implies$ (T1). We assume that f_n converges to f in $\mathcal{L}^1(\mu)$. Let us first focus on showing that $f_n \xrightarrow{\mu} f$.

By definition, f_n converges to f in measure if:

$$\forall \varepsilon > 0, \ \mu\left(\{x \in X \mid \varepsilon \leq |f_n(x) - f(x)|\}\right) \to 0 \text{ as } n \to \infty.$$

Let $\varepsilon > 0$ be arbitrary, now by applying Markov's inequality we have:

$$\forall n \in \mathbb{N}, \ 0 \leq \mu\left(\{x \in X \mid \varepsilon \leq |f_n(x) - f(x)|\}\right) \leq \frac{1}{\varepsilon} \int |f_n - f| \, \mathrm{d}\mu.$$

Now by our assumption of convergence in $\mathcal{L}^1(\mu)$ we deduce that

$$\frac{1}{\varepsilon} \int |f_n - f| \, \mathrm{d}\mu \to 0 \text{ as } n \to \infty..$$

Hence, by applying the squeeze theorem we conclude that $f_n \xrightarrow{\mu} f$. Now let us show that the family $\mathcal{F} = \{f_n \mid n \in \mathbb{N}\}$ has uniformly absolutely continuous integrals (i.e. that (1) holds for $\mathcal{F}$).

Let $\varepsilon > 0$ be arbitrary, by convergence in $\mathcal{L}^1(\mu)$ we can find $n_0 = n_0(\varepsilon)$ satisfying :

$$\forall n \geq n_0, \text{ we have } \int |f_n - f| \, \mathrm{d}\mu < \frac{\varepsilon}{2}.$$

Let us consider the finite family $\mathcal{F}' = \{f, f_1, f_2, \ldots, f_{n_0}\}$, using the fact that any finite family has uniformly absolutely continous integrals, we can find a $\delta > 0$, such that:

$$\forall \ A \in \mathcal{A} \text{ with } \mu(A) < \delta, \text{ we have } \int_A |f| \, \mathrm{d}\mu < \frac{\varepsilon}{2} \text{ and } \max_{n \leq n_0} \int_A |f_n| \, \mathrm{d}\mu < \frac{\varepsilon}{2} \tag{2}$$

Now consider the following chain of inequalities for $n > n_0,\ A \in \mathcal{A},\ \mu(A) < \delta$:

$$\int_A |f_n|\,\mathrm{d}\mu \leq \int_A |f|\,\mathrm{d}\mu + \int_A |f_n - f|\,\mathrm{d}\mu \leq \int_A |f|\,\mathrm{d}\mu + \int |f_n - f|\,\mathrm{d}\mu. < \frac{\varepsilon}{2} + \frac{\varepsilon}{2} = \varepsilon$$

From the second part of (2) we can deduce that:

$$\max_{n \leq n_0} \int_A |f_n|\,\mathrm{d}\mu < \frac{\varepsilon}{2} < \varepsilon.$$

Hence, by combining the above results we get that:

$$\forall n \in \mathbb{N}, \int_A |f_n|\,\mathrm{d}\mu < \varepsilon.$$

Therefore we have shown that $\mathcal{F}$ has uniformly absolutely continuous integrals and consequently the backwards direction of the theorem holds.

Now let us show that (T1) $\implies$ (T2). Let us assume that f_n converges to f in measure and that the family $\mathcal{F}$ has uniformly absolutely continuous integrals. We'll prove that $\int |f_n - f|\,\mathrm{d}\mu \to 0$, as $n \to \infty$ using the fact that a sequence converges if every subsequence has a convergent subsequence. Let $(f_{n_k})_{k\in\mathbb{N}}$ be an arbitrary subsequence of $(f_n)_{n\in\mathbb{N}}$. Consequently, $(\int |f_{n_k} - f|\,\mathrm{d}\mu)_{k\in\mathbb{N}}$ is a subsequence of $(\int |f_n - f|\,\mathrm{d}\mu)_{n\in\mathbb{N}}$. We'll show that it has a subsequence:

$$\int |f_i - f|\,\mathrm{d}\mu \to 0 \text{ as } i \to \infty \text{ along } \Lambda.$$

Which will in turn imply that $\int |f_n - f|\,\mathrm{d}\mu \to 0$ as $n \to \infty$ because n_k was an arbitrary subsequence.

Let us observe that, since $f_n \xrightarrow{\mu} f$ and $(f_{n_k})_{k\in\mathbb{N}}$ is a subsequence of $(f_n)_{n\in\mathbb{N}}$, it implies that we also have:

$$f_{n_k} \xrightarrow{\mu} f.$$

Also given the fact that we have $\mu(X) < \infty$, we can apply the proposition (2.45) from the MATH50006: Lebesgue Measure and Integration course lecture notes, which tells us that there exists a subsequence $\Lambda \subset n_k$ such that:

$$f_i \to f\ \mu\text{-a.e. as } n \in \Lambda \to \infty \tag{3}$$

Furthermore, given that $\mathcal{F}$ has uniformly absolutely continuous integrals, we can pass to a subsequence and deduce that $\mathcal{F}'' = \{f_i \mid i \in \Lambda\}$ also has uniformly absolutely continuous integrals. That follows from the definition (1), if for and ε we get a δ such that:

$$\forall f \in \mathcal{F}, \forall A \in \mathcal{A}, \text{ we have } \mu(A) < \delta \implies \int_A |f|\,\mathrm{d}\mu < \varepsilon..$$

Then also, since $\mathcal{F}'' \subset \mathcal{F}$, we get that

$$\forall f \in \mathcal{F}'', \forall A \in \mathcal{A}, \text{ we have } \mu(A) < \delta \implies \int_A |f| \, \mathrm{d}\mu < \varepsilon \tag{4}$$

Now we use (3), (4) and the proposition that if a sequence of functions converges μ-a.e. and has uniformly absolutely continuous integrals, then it converges in $\mathcal{L}^1(\mu)$. Therefore we have shown:

$$\int |f_i - f| \, \mathrm{d}\mu \to 0 \text{ as } i \to \infty \text{ along } \Lambda.$$

Which concludes the proof as we have shown that an arbitrary subsequence of $(\int |f_n - f| \, \mathrm{d}\mu)_{n \in \mathbb{N}}$ has a subsequence convergent to 0. ■

The Process of Formalising

Methodology

In order to formalise Vitali's theorem, I followed a similar approach to what I did in my first project. This time however, I was able to avoid some of the mistakes that I made in my first project. The first modification was that I started proving the theorem by first breaking it down into separate lemmas which encapsulated the three 'branches' that one has to show in order to prove the theorem. In the proof presented above, we first needed to show that convergence in $\mathcal{L}^1(\mu)$ implies convergence in measure (first 'branch'), then we have shown that it implies that $\mathcal{F}$ has uniformly absolutely continuous integrals. Finally, we have shown the forward direction of the theorem which concluded the proof. The corresponding logical structure of the proof can be seen in the listing below.

```
theorem vitali_theorem {m : measurable_space X} {μ : measure X} [is_finite_measure μ]
(f : ℕ → X → ℝ) (g : X → ℝ) (hf : ∀ (n : ℕ), (f n) ∈_L1{μ}) (hg : g ∈_L1{μ}) :
f -→ g in_measure{μ} ∧ unif_integrable f 1 μ ⟺ f -→ g in_L1{μ} :=
begin
  split,
  { rintro ⟨h_tendsto_μ , h_unif_int⟩,
    exact tendsto_L1_of_unif_integr_of_tendsto_in_μ hf hg h_tendsto_μ  h_unif_int, },
  { intro h_tendsto_L1,
    split,
    { exact tendsto_in_measure_of_tendsto_L1 hf hg h_tendsto_L1 },
    { exact unif_integrable_of_tendsto_L1 hf hg h_tendsto_L1, }, },
end
```

Listing 1: Main statement of Vitali's theorem.

In my implementation, instead of starting to prove the theorem within one monolithic proof (which admittedly was my biggest mistake in my first project), I started by defining sub-lemmas for each of those three branches and working out the precise sets of hypotheses that they required to be shown.

Modularising the argument from the very beginning contributed to an improved experience of using the language. Because my proofs were shorter, they were compiled much faster which allowed me to get interactive feedback on the state of my proof a lot quicker and perform faster iterations if something wasn't quite working. As I prioritised introducing lemmas, I was trying to generalise some of the intermediate propositions that I was proving, which often allowed me to reuse one lemma multiple times. That modular approach also helped me make progress when I got stuck on something which I couldn't prove, in such case I would almost always introduce a lemma containing a generalised version of the problematic proposition, put a `sorry` tactic inside of it to make it compile, and then continue with the rest of the proof.

The second modification to my methodology was that I needed to start interacting with the API provided by the mathlib library more than I used to in my first project. This change was because in this project my goal was to prove a statement which involved advanced measure theory. I could no longer simply define my own definition of continuity as I did in my first project, this time it wasn't feasible to define my own notions of a measure, measurable functions or the three notions of convergence that I operated on.

A good example of that approach can be seen below, where in the process of proving that convergence in $\mathcal{L}^1(\mu)$ implies convergence in measure I needed to use Markov's inequality and relied on the implementation provided by the library.

```
theorem tendsto_in_measure_of_tendsto_L1 {m : measurable_space X} {μ : measure X}
{f : ℕ → X → ℝ} {g : X → ℝ} (hf : ∀ (n : ℕ), f n ∈_L1{μ}) (hg : g ∈_L1{μ})
(h : f -→ g in_L1{μ}) : f -→ g in_measure{μ} :=
begin
  -- Here we aim to apply the Markov's inequality to f_n - g for all n ∈ ℕ.
  have h2 : ∀ (ε : ℝ≥0∞), ε ≠ 0 → ∀ (n : ℕ), μ { x | ε ≤ ‖f n x - g x‖₊ } ≤
    ε⁻¹ * ∫⁻(f n) - g⁻dμ,
  { intros ε hε n,
    -- We use sub_strongly_measruable to show that f_n - g is ae_strongly_measurable.
    exact markov_ineq_L1 μ (sub_strongly_measurable (hf n) hg) ε hε, },
  simp at h2,
```

Listing 2: Using mathlib's Markov inequality.

Challenges

As the scope of this project included interacting with more advanced theory from second year university mathematics, formalising it has proven more difficult than the first year analysis that I was working on in my first project. The two interesting challenges that I have overcome were type coercions and proving statements involving finite objects.

The first one of these came up because I was dealing with measure theory, and since we want a measure to be a function which can return infinity, we need to work with extended non-negative real numbers. In usual mathematical papers it is not that difficult to handle it, we just declare those numbers to be defined as $\mathbb{R}^+ \cup \{+\infty\}$ and require a certain degree of caution when dealing with the 'infinity' element. However, in Lean the extended non-negative real numbers (`ennreal`) are a completely different type from the usual real numbers. Therefore, if we for instance require that $\mu(A) < \varepsilon$ for some $\varepsilon \in \mathbb{R}$ and $A \in \mathcal{A}$, that proposition doesn't even make sense in the world of Lean. That is because the return type is `ennreal` whereas ε is an ordinary real number. Because of this, in the code you need to convert types of variables using special functions which are called coercions. The difficult thing about this is that not always is it possible to coerce between types. For instance it doesn't make sense to convert a negative number into an `ennreal`. Because of this, often when converting between these types one has to supply additional hypotheses such as that an `ennreal` number is not equal to infinity if we want to convert it into a real number. An example of the usage of coercions can be seen in the listing below.

```
theorem ennreal_squeeze_zero {a b : ℕ → ℝ≥0∞} (ha : ∀ (n : ℕ), a n < ∞)
(hb : ∀ (n : ℕ), b n < ∞) (h0b : ∀ n, 0 ≤ b n) (hba : ∀ n, b n ≤ a n)
(ha_to_0 : tendsto a at_top (𝓝0)) : tendsto b at_top (𝓝0) :=
begin -- First we need to coerce the sequences to apply nnreal_squeeze_zero on them.
  lift a to ℕ → ℝ≥0, {exact lift_ennreal_to_nnreal ha },
  lift b to ℕ → ℝ≥0, {exact lift_ennreal_to_nnreal hb },
  rw [← ennreal.coe_zero, ennreal.tendsto_coe] at *, -- removes coercions from ℝ≥0 to ℝ≥0∞
  simp at hba,
  dsimp at h0b,
  have h0b : ∀ (n : ℕ), 0 ≤ b n,
  { intro n,-- Here we need to rewrite so that h0b matches hypothesis of nnreal_squeeze_zero
    exact ennreal.coe_le_coe.mp (h0b n), },
  exact nnreal_squeeze_zero h0b hba ha_to_0,
end
```

Listing 3: Coercions required to apply `squeeze_zero`

It is a proof of the squeeze theorem for `ennreal` numbers. In the main proof I needed to apply squeeze theorem to two sequences of ennreal numbers, in order to do this, I needed to cast those sequences into non-negative real numbers, then perform coercions on the limits to make the types match up and finally I was able to apply the library version of the squeeze theorem for non-negative real numbers. It illustrates how coercions make the proofs much more complicated as the programmer needs to ensure that each variable if of precisely the right type when it is in some expression involving terms of the specific type.

The second aspect of the proof that I found more difficult than expected was proving propositions for finite objects such as lists and sets. For instance, when proving that a singleton set $\{g\}$ containing a single measurable function $g \in \mathcal{L}^1(\mu)$ was more involved than I anticipated (see Listing 4).

```
/-- This lemma proves that if we take a single function in L1(μ) then this set
    has uniformly absolutely continuous integrals. -/
theorem unif_integrable_singleton {m : measurable_space X} {μ : measure X}
{g : X → ℝ} (hg :g ∈_L1{μ}) :
∀ {ε : ℝ} (hε : 0 < ε), ∃ (δ : ℝ) (hδ : 0 < δ), ∀ s, measurable_set s
  → μ s ≤ ennreal{δ} → (∫_{s}_|g|dμ) ≤ ennreal{ε} :=
begin
  -- We define a finite set {g}.
  let G : fin 1 → X → ℝ := λ i, g,
  have hG : ∀ (i : fin 1), (G i) ∈_L1{μ}, { intro i, exact hg, },
  have hG_unif_integrable : unif_integrable G 1 μ,
  -- Here we show that it is has uniformly abs. cont. integrals.
  { exact unif_integrable_fin μ (le_refl 1) ennreal.one_ne_top hG, },
  intros ε hε,
  specialize hG_unif_integrable hε,
  rcases hG_unif_integrable with ⟨δ, hδ, hG⟩,
  use δ,
  split,
  { exact hδ },
  { intros s hs,
    specialize hG 1 s hs,
    exact hG, },
end
```

Listing 4: Proof of a proposition for a singleton set.

As you can see above, in order to show that proposition I needed to define a new 'dummy' function `G`

which maps from `fin 1` which is a finite sequence of natural numbers up to and not including 1 (i.e. it is the set $\{0\}$.) Then I used the `unif_integrable_fin` theorem from the library which shows that any finite family of functions has uniformly absolutely continuous integrals.

Another interesting lemma which I have introduced aimed to show that if we have a list of two strongly measurable functions f and g then if we define a list $[f, g]$ then all items in that list are strongly measurable. When formulated using plain English, this propositions sound trivial, however in lean it required some careful case analysis based on whether the item is equal to the first or second item in the list. The reason why I introduced this lemma was that I needed to show that if $f-g$ is strongly measurable. Hence I needed to form a finite list $[f, -g]$ and then apply the proposition `ae_strongly_measurable_sum` which shows that a sum a finite list of strongly measurable functions is strongly measurable. However, after proving it I realised that it is actually a known fact in the library (called `ae_strongly_measurable.sub`) and I didn't need to do all of the list manipulations. Still, I decided to leave that proof in my implementation because I think it is an interesting exercise illustrating proofs with finite sets.

```
lemma list_elem_ae_strongly_measurable {m : measurable_space X} {μ : measure X}
{f g : X → ℝ} (hf : ae_strongly_measurable f μ) (hg : ae_strongly_measurable g μ)
{l : list (X → ℝ)} (hl : l = [f, g]) :
∀ (k : X → ℝ), k ∈ l → ae_strongly_measurable k μ :=
begin
  intros k k_in_l,
  by_cases k = f,
  { rw h, exact hf},
  { -- Here we show that if it is not equal to f then it must be equal to g
    -- as l has only two elements. As stated above it wasn't trivial.
    have hkg : k = g,
    { have h_cases : k = f ∨ k ≠ f ∧ k ∈ [g],
      { rw hl at k_in_l, exact list.eq_or_ne_mem_of_mem k_in_l, },
      cases h_cases,
      { exfalso, exact h h_cases, },
      { exact list.mem_singleton.mp h_cases.right }, },
    rw hkg,
    exact hg, },
end
```

Listing 5: Proof of a proposition for a finite list.

Language Extensions

One thing that I noticed when iterating with the API provided by mathlib is that even though it is very exhaustive and almost any imaginable theorem from measure theory is already available there, it sometimes isn't very easy to read or write the code because of the verbosity of the implementation that the library provides.

Consider the following example: let $\varepsilon \in \mathbb{R}$, $s \in \mathcal{A}$ and f be any integrable function, suppose I wanted to express the following inequality using mathlib:

$$\int_s |f| \, \mathrm{d}\mu \leq \varepsilon.$$

In my proofs I would have to write this:

```
snorm (s.indicator (f)) 1 μ ≤ ennreal.of_real (ε)
```

Listing 6: Limited expressiveness of the API.

Although it might not look too overwhelming, things can get out of control quite quickly if there are multiple integrals involved. Also another problem that I identified is that in the signature of the `snorm` function above, one has to write `1 μ` at the end which specifies that we want to use the $\mathcal{L}^1(\mu)$ norm. It might not be a very big issue, however I kept forgetting the order in which those two need to be specified. Moreover, the fact that these trailing arguments are a common occurrence among many function in the library didn't make it easier. One has to remember all of the function signatures or look them up constantly which slows down the progress. The listing below illustrates how the verbose syntax.

```
lemma extract_δ_uaci {m : measurable_space X} {μ : measure X} [is_finite_measure μ]
{ f : ℕ → X → ℝ } { g : X → ℝ }
(h_f_n_uaci : ∀ (ε : ℝ), ε > 0 →  ∃ (δ : ℝ) (hδ : 0 < δ), ∀ (n : ℕ) s, measurable_set s →
  μ s ≤ ennreal.of_real δ → snorm (s.indicator (f n)) 1 μ ≤ ennreal.of_real (ε / 3))
(h_g_uaci : ∀ (ε : ℝ), ε > 0 →  ∃ (δ : ℝ) (hδ : 0 < δ), ∀ s, measurable_set s →
  μ s ≤ ennreal.of_real δ → snorm (s.indicator (g)) 1 μ ≤ ennreal.of_real (ε / 3))
: ∀ (ε : ℝ), ε > 0 →  ∃ (δ : ℝ) (hδ : 0 < δ),
  ∀ s, measurable_set s → μ s ≤ ennreal.of_real δ →
  ∀ (n : ℕ), snorm (s.indicator (f n)) 1 μ ≤ ennreal.of_real (ε / 3) ∧
  snorm (s.indicator (g)) 1 μ ≤ ennreal.of_real (ε / 3) :=
begin ...
```

Listing 7: Verbosity of the mathlib functions.

In order to solve the problem of readability, I decided to add new notation which was specific to the file that I was working in. It is important to note that overriding notation with new definitions should be done with caution and in a consistent manner, so the professionals maintaining the mathlib library probably wouldn't opt for my custom definitions which are far from being perfect. I introduced the new notation only because it allowed me to explore some of the more advanced features of the language and make the code development process slightly more convenient. I wanted to indicate that I don't claim that the API in the mathlib library is flawed, my definitions were merely an exploration and shouldn't be considered an attempt to fix the library.

I began by introducing alias definitions and new notation for the basic measure theoretic concepts such as the notion of being in $\mathcal{L}^1(\mu)$ and the three types of convergence:

```
def tendsto_in_L1 {m : measurable_space X}
(μ : measure X) (f : ℕ → X → ℝ) (g : X → ℝ) : Prop :=
filter.tendsto (λ (n : ℕ), snorm (f n - g) 1 μ) filter.at_top (𝓝 0)

def tendsto_μ_ae {m : measurable_space X} (μ : measure X)
(f : ℕ → X → ℝ) (g : X → ℝ) : Prop :=
∀ᵐ (x : X) ∂μ, filter.tendsto (λ (n : ℕ), f n x) filter.at_top (𝓝 (g x))
-- Now three notions of convergence can be expressed using a consistent notation.
notation f `→` g `in_L1{` μ `}` := tendsto_in_L1 μ f g
notation f `→` g `in_measure{` μ `}` := tendsto_in_measure μ f at_top g
notation f `→` g `{` μ `}_a-e` := tendsto_μ_ae μ f g
-- Also the notion of being a member of L1 is somewhat hard to decipher.
notation f `∈_L1{` μ `}` :=  mem_ℒp f 1 μ
-- Notation for snorms
notation `∫|` f `|d` μ :=  (snorm (f) 1 μ)
notation `∫_{` s `}_|` f `|d` μ :=  snorm (s.indicator f) 1 μ
-- Notation for converting between types
notation `ennreal{` a `}` := ennreal.of_real(a)
notation `real{` a `}` := ennreal.to_real(a)
```

Listing 8: New notation definitions.

After that I replaced all occurrences of the previous notation with the custom definitions. The result of that refactoring, which can be seen in the listing below, can be contrasted with Listing 7. As we can see the expressions have become much shorter and arguably more readable assuming that one knows that there are some notations overrides in place.

```
lemma extract_δ_uaci {m : measurable_space X} {μ : measure X} [is_finite_measure μ]
{ f : ℕ → X → ℝ } { g : X → ℝ }
(h_f_n_uaci : ∀ (ε : ℝ), ε > 0 → ∃ (δ : ℝ) (hδ : 0 < δ), ∀ (n : ℕ) s, measurable_set s →
              μ s ≤ ennreal{δ} → ( ∫_{s}_|f n|dμ ) ≤ ennreal{ε / 3})
(h_g_uaci : ∀ (ε : ℝ), ε > 0 → ∃ (δ : ℝ) (hδ : 0 < δ), ∀ s, measurable_set s →
            μ s ≤ ennreal{δ} → ( ∫_{s}_|g|dμ ) ≤ ennreal{ε / 3})
: ∀ (ε : ℝ), ε > 0 → ∃ (δ : ℝ) (hδ : 0 < δ), ∀ s, measurable_set s → μ s ≤ ennreal{δ} →
  ∀ (n : ℕ), ( ∫_{s}_|f n|dμ ) ≤ ennreal{ε / 3} ∧ ( ∫_{s}_|g|dμ ) ≤ ennreal{ε / 3} :=
begin ...
```

Listing 9: Code from Listing 7 after refactoring.

The impact of the applied refactoring might not be apparent at the first glance, especially given how complex the definition above still is. However, I find it more readable and it arguably faster to type out because the raw number of characters required is slightly lower.

Conclusions

To conclude, the second part of my coursework was a great opportunity to learn more about proving theorems in Lean and also revise some concepts from my favourite second year module.

Having proved Vitali's theorem I have learned how to use the toolkit provided by mathlib in measure theoretic context. I also gained appreciation for the usefulness of the library and how indispensable it is when proving more advanced propositions. I realised how important it is to be able to search through the library efficiently and quickly identify if a given theorem in the library can be applied in the context of the proof that one is currently working on.

The project also allowed me to explore some of the more advanced features of the Lean programming language. Introducing the new notation was an interesting way to learn how prefix and infix operators are handled in lean and how overriding the precedence of operators can lead to unexpected results.