Formalising Mathematics Coursework 1

[redacted]

1 Sequences

In this project, I prove the Bolzano-Weierstrass Theorem (every bounded sequence has a convergent subsequence) and use it to show that Cauchy sequences are convergent, using my own implementations of real-valued natural-indexed sequences and convergence.

1.1 Inital Definitions

The first file of my project that Lean will read is `sequence.lean`, which contains the definition of sequences and various properties about them: sequences and convergence are defined in mathematically familiar ways:

```
/-- Sequence of real numbers indexed by natural numbers. -/
def seq := ℕ → ℝ

/-- Proposition that a sequence converges to a specified limit. -/
def tendsto (s : seq) (x : ℝ) : Prop := ∀ {ε}, 0 < ε → ∃ B : ℕ, ∀ {n}, B ≤ n → |s n - x| < ε
notation s ` ⟶ ` limit := tendsto s limit

/-- Proposition that there is a limit to which a sequence converges. -/
def seq.convergent (s : seq) : Prop := ∃ x, s ⟶ x
```

I used the bold curly brackets around ϵ and n in the `tendsto` definition to indicate they are weak implicit arguments, i.e. Lean should only try and fill them upon receiving the next explicit argument, rather than filling them with whatever it can find in the tactic state as soon there are no explicit arguments expected before them. Thanks to this behaviour, the arguments $\epsilon \in \mathbb{R}$ and $n \in \mathbb{N}$ are only determined when the hypotheses following them are given, namely $0 < \epsilon$ and $B \leq n$, which usefully include ϵ and n in them, ensuring Lean doesn't have to generate any metavariables.

I also introduced some notation for `tendsto`, as seen in the definition `convergent`.

1.2 Transformations

```
/-- Sequence formed by taking the absolute value of each term. -/
noncomputable def seq.abs (s : seq) : seq := λ k, |s k|

/-- Sequence formed by taking the additive inverse of each real term of the original sequence `s`. -/
def seq.neg (s : seq) : seq := λ n, - s n

/-- Sequence formed by ignoring the first `n` elements, such that `s.shift 0 = s n`. -/
def seq.shift (s : seq) (n : ℕ) : seq := λ m, s (m + n)
```

I have defined some useful ways to transform sequences: `abs`, `neg` and `shift`. The first of these is the simplest, taking the absolute value of each term in a sequence to produce a new one, and is primarily

used when the entire absolute-value sequence is needed, rather than the specific-value case where taking the absolute value of the term suffices (`|s n|` rather than `s.abs n`).

The `neg` transformation is also simple in its definition, sending each real term to its real additive inverse, but it has more theorems describing its behaviour. The main use of `neg` in this project is the fact that it preserves convergence (up to negating the limit) and monotonicity (up to swapping increasing and decreasing), allowing, for example, proofs about increasing sequences to be transformed into proofs about decreasing sequences. This all relies on a proof that negation is an involution (i.e. $-(-s) = s$), which follows from the same fact for negation on the real numbers.

In analysis, we often consider statements about sequences concerning all terms greater than a certain index, so to simplify the situation I have defined the `shift` transformation which removes a finite number of leading terms from a sequence, producing a new sequence beginning at the specified index. This transformation preserves some useful properties, such as convergence and, as we will later discuss, boundedness, and allows us to view parts of a sequence above a certain index as sequences themselves, which can be used as arguments in any sequence related theorems.

1.3 Properties

The Bolzano-Weierstrass Theorem requires definitions of bounded and monotone, each of which requires further definitions for their more specific versions.

```
/-- Proposition that a sequence is bounded above, i.e. all terms lie within `(-∞, M]`. -/
def seq.bounded_above (s : seq) : Prop := ∃ M, ∀ n, s n ≤ M

/-- Proposition that a sequence is bounded above, i.e. all terms lie within `[m, ∞)`. -/
def seq.bounded_below (s : seq) : Prop := ∃ m, ∀ n, m ≤ s n

/-- Proposition that a sequence is bounded above and below, i.e. all terms lie within `[-M, M]`. -/
def seq.bounded (s : seq) : Prop := s.abs.bounded_above
```

A sequence can be `bounded_above`, `bounded_below` or `bounded` - although the first two together are equivalent to the third, I chose to represent `bounded` by an upper bound on the absolute value and provide a theorem to show `bounded` $\iff$ `bounded_above` $\wedge$ `bounded_below`, as this provides one number $M \in \mathbb{R}$ such that M is an upper bound and $-M$ is a lower bound of a sequence, rather than separate upper and lower bounds that would arise from the definition `bounded` $\coloneqq$ `bounded_above` $\wedge$ `bounded_below`.

```
/-- Proposition that every term in a sequence is greater than or equal to any term before it. -/
def seq.increasing (s : seq) : Prop := ∀ {m n}, m ≤ n → s m ≤ s n
/-- Proposition that every term in a sequence is strictly greater than any term before it. -/
def seq.strictly_increasing (s : seq) : Prop := ∀ {m n}, m < n → s m < s n

/-- Proposition that every term in a sequence is less than or equal to any term before it. -/
def seq.decreasing (s : seq) : Prop := ∀ {m n}, m ≤ n → s n ≤ s m
/-- Proposition that every term in a sequence is strictly less than any term before it. -/
def seq.strictly_decreasing (s : seq) : Prop := ∀ {m n}, m < n → s n < s m

/-- Proposition that in a sequence, either every term is greater than or equal to any term before it,
  or every term is less than or equal to any term before it. -/
def seq.monotone (s : seq) : Prop := s.increasing ∨ s.decreasing
/-- Proposition that in a sequence, either every term is strictly greater than any term before it,
  or every term is strictly less than any term before it. -/
def seq.strictly_monotone (s : seq) : Prop := s.strictly_increasing ∨ s.strictly_decreasing
```

I defined `increasing`, `decreasing` and `monotone` (and their strict versions) as they are normally defined mathematically, paying special attention to use $m \leq n$ in the non-strict versions, so as to include the trivial $m = n$ case, as it saves us having to provide separate cases in later proofs. These definitions have various theorems to aid in their use, ranging from basic facts, like strict $\implies$ non-strict and negation swapping increasing

and decreasing, to important parts of the final proof, namely `convergent_of_bounded_above_increasing`, the proof that bounded above increasing sequences are convergent (and using negation theorems, we have the decreasing version: `convergent_of_bounded_below_decreasing`). This relies on a Mathlib theorem about the real numbers, `real.is_lub_Sup`, that shows that the suprenum of a set of real numbers is a least upper bound of that set, allowing me to prove that the sequence converges to the suprenum.

1.4 Finite Prefix

One fact needed for both the Peak Point Lemma and proving Cauchy sequences are convergent is that for any $n \in \mathbb{N}$, the first n terms of any sequence have maximal and minimal elements. This seems obvious to humans, but as with anything finite in Lean, it's a bit more complicated.

```
/-- A finite set made up of the values of the first `n` terms
  of the sequence `s` (from `s 0` to `s (n-1)`). -/
noncomputable def finite_prefix (s : seq) (n : ℕ) : finset ℝ :=
  (multiset.map s (multiset.range n)).to_finset
```

First, I map a `multiset` (i.e. unordered list) containing the natural numbers less than n to the values the sequence takes for them, and then I turn the resulting `multiset` into a `finset`, erasing any duplicate elements. This series of steps is needed as mapping indicies to their terms in a sequence is not necessarily injective, hence the result may contain duplicate elements, so I use `multiset`s. However, finite min and max are only defined on `finset`s, which is why the final step is required. The resulting `finset` can be used as an argument for various min and max functions, which I use to find the index of a maximal and a minimal term among the first n terms of any sequence (although I only ended up using the max functions, as the first theorem relatively simply implies the second, again using negation of sequences).

```
/-- For any `n`, the real numbers `|s m|` for `m < n` have a maximal element. -/
theorem finite_prefix_max (s : seq) {n : ℕ} (hn : n ≠ 0) :
  ∃ m, m < n ∧ ∀ k < n, s k ≤ s m :=
begin
  let fs := (finite_prefix s n),
  have hM := mem_finite_prefix.mp (fs.max'_mem (finite_prefix_nonempty s hn)),
  cases hM with m hm,
  use m, split, exact hm.left,
  intros k hk, rw [hm.right],
  exact fs.le_max' (s k)(mem_finite_prefix.mpr ⟨k, hk, rfl⟩)
end

/-- For any `n`, the real numbers `|s m|` for `m < n` have a minimal element. -/
theorem finite_prefix_min (s : seq) {n : ℕ} (hn : n ≠ 0) :
  ∃ m, m < n ∧ ∀ k < n, s m ≤ s k :=
begin
  cases finite_prefix_max (-s) hn with M hM,
  use M, split, exact hM.left,
  intros k hk,
  exact neg_le_neg_iff.mp (hM.right k hk)
end
```

2 Subsequences

In my project's next file, `subsequence.lean`, I define subindicies and various tools for constructing them, as well as using them to prove some of the project's main results, the Peak Point Lemma and the Bolzano-Weierstrass Theorem.

2.1 Subindex

```
/-- A subindex is a strictly increasing map from ℕ to itself. -/
structure subindex :=
  (ϕ : ℕ → ℕ)
  (mono : strict_mono ϕ)
```

```
/-- A sub-sequence, formed by composing a sequence and a subindex. -/
def seq.subseq (s : seq) (si : subindex) : seq := s ∘ si
```

A `subindex` represents a strictly increasing map $\mathbb{N} \to \mathbb{N}$, such that the composition of a subindex and a sequence produces the usual mathematical concept of a subsequence. This definition, however, is not very useful on its own, as it requires a fully defined map $\phi : \mathbb{N} \to \mathbb{N}$ with $\forall\, a < b$, $\phi(a) < \phi(b)$, whereas I wish to construct maps by a process that takes the value $\phi(n) \in \mathbb{N}$ for some $n \in \mathbb{N}$ and produces $\phi(n+1) \in \mathbb{N}$ such that $\phi(n) < \phi(n+1)$. To this end, I define a function `subindex_mk` that takes a map $\mathbb{N} \to \mathbb{N}$ and a hypothesis that consecutive terms are strictly increasing and returns a `subindex`, inductively using the hypothesis to show the strictly increasing condition.

```
/-- The strictly increasing property of subindex follows inductively from the
  function having strictly increasing consecutive terms. -/
def subindex_mk (f : ℕ → ℕ) (h : ∀ n, f n < f n.succ) : subindex :=
  ⟨f, ...⟩
```

```
/-- A function used for defining subindex structures, such that each
  index is `f` applied to the previous index, beginning with `f 0`. -/
def subindex_recursive (f : ℕ → ℕ) : ℕ → ℕ
| 0     := f 0
| (k+1) := f (subindex_recursive k)
```

Another useful tool is the function `subindex_recursive`, which given a map $f : \mathbb{N} \to \mathbb{N}$ constructs a map $\tilde{f}$ such that $\tilde{f}(0) = f(0)$, $\forall\, n \in \mathbb{N}$, $\tilde{f}(n+1) = f(\tilde{f}(n))$. Given a hypothesis $\forall\, n \in \mathbb{N}$, $n < f(n)$, we have $\tilde{f}(n) < f(\tilde{f}(n))$, which implies $\tilde{f}(n) < \tilde{f}(n+1)$, and so the result of `subindex_recursive` can be used with `subindex_mk` to construct the desired `subindex`.

2.2 Bolzano-Weierstrass

```
/-- Every sequence has a monotone sub-sequence. -/
theorem peak_point_lemma (s : seq) : ∃ si : subindex, (s.subseq si).monotone :=
begin
  by_cases h : ∀ N, ∃ m, m > N ∧ ∀ n, n > N →  s m ≤ s n,
  -- Increasing case.
  { ... },
  -- Decreasing case.
  { ... }
end
```

To prove the Bolzano-Weierstrass Theorem, I first proved `peak_point_lemma`: every sequence has a monotone subsequence. This is by far the longest proof in this project, split into two cases based on whether or not the sequence s satisfies: $\forall N \in \mathbb{N}, \exists m > N$ s.t. $\forall n > N, s(m) \leq s(n)$. It uses the method explained above, using `subindex_recursive` with `subindex_mk`, to construct an either increasing or decreasing subsequence, recursively using `classical.some` to choose ever greater numbers from the case hypotheses. As mentioned earlier, this proof also uses `finite_prefix_min`, in the second/decreasing case, to find a minimal term among the finite number of terms not bounded by the second case hypothesis.

```
/-- Every bounded sequence has a convergent subsequence. -/
theorem bolzano_weierstrass {s : seq} (hb : s.bounded) :
  ∃ si : subindex, (s.subseq si).convergent :=
begin
  cases peak_point_lemma s with si hm,
  cases hm with inc dec,
  { exact ⟨si, convergent_of_bounded_above_increasing (subseq_bounded_above
    (bounded_iff_above_below.mp hb).left si) inc⟩ },
  { exact ⟨si, convergent_of_bounded_below_decreasing (subseq_bounded_below
    (bounded_iff_above_below.mp hb).right si) dec⟩ }
end
```

Having proved `peak_point_lemma`, `bolzano_weierstrass` is relatively simple: by `peak_point_lemma`, any sequence is monotone, so increasing or decreasing, hence if it is bounded both above and below then it is either increasing and so satisfies `convergent_of_bounded_above_increasing`, or decreasing and so satisfies `convergent_of_bounded_below_decreasing`.

3 Cauchy Sequences

The last file of my project is `cauchy.lean`, where I define Cauchy sequences and prove that any sequence is convergent if and only if it is Cauchy.

```
/-- Proposition that a sequence is Cauchy, that is that its terms grow arbitrarily close together. -/
def seq.cauchy (s : seq) : Prop := ∀ {ε}, ε > 0 → ∃ B : ℕ, ∀ {m n}, B ≤ m → B ≤ n → |s m - s n| < ε
```

I define `cauchy` similarly to how I defined `tendsto`, including the bold curly brackets to signify weak implicit arguments. I require $B \leq m, n$ so that m or n can be equal to B, meaning B itself is a possible argument, as opposed to the more convoluted option of $B < m, n$ and using $B + 1$ as an argument.

I prove that any Cauchy sequence is bounded in `cauchy_is_bounded`, making use of the fact that boundedness (although not necessarily the bound itself) is preserved by arbitrary finite shifts, proved in `sequences.lean`, relying on the `finite_prefix` theorems.

```
/-- Proof that Cauchy and convergent are equivalent properties. -/
theorem convergent_iff_cauchy {s : seq} : s.convergent ↔ s.cauchy :=
begin
  split,
  { -- convergent → cauchy
    ... },
  { -- cauchy → convergent
    intro hc,
    -- We get the limit from the convergent subsequence `bolzano_weierstrass` gives us.
    cases bolzano_weierstrass (cauchy_is_bounded hc) with si hcon,
    ... }
end
```

The final proof of the project, that sequences are convergent if and only if they are Cauchy, is split into the two directions of implication: convergent $\implies$ Cauchy follows from manipulation of the inequalities given by the convergence; Cauchy $\implies$ convergent requires a limit to be found from somewhere, namely from the convergent subsequence that, by `bolzano_weierstrass` and `cauchy_is_bounded`, a Cauchy sequence must have. Inequalities from the convergencec of the subsequence and Cauchy-ness of the sequence are then used to show the original sequence converges to the same limit as the subsequence, completing the proof.