Mathematical Analysis

Finally some real math in Lean! In this file we show how to define limits of sequences and continuity of functions in Lean. Of course it is just a toy version, far from the real Mathlib definitions. Nevertheless, that should be enough for you to get a taste of formalizing something that is not completely trivial.

Since we haven't touch division quite much yet, you may find it's difficult to deal with multiplication and division. `field_simp` tactic may help you a lot in such cases. It won't break things up as `simp` does. Anyway, don't worry too much about it for now.

```
import Mathlib

def TendsTo (a : ℕ → ℝ) (t : ℝ) : Prop :=
  ∀ ε > 0, ∃ n₀ : ℕ, ∀ n, n₀ ≤ n → |a n - t| < ε
```

[EXR] The limit of the constant sequence with value `c` is `c`.

```
theorem tendsTo_const (c : ℝ) : TendsTo (fun _ ↦ c) c := by
  unfold TendsTo
  intro ε hε
  use 1
  intro n hn
  simp [hε]
```

`-` commutes with `tendsTo`

```
theorem tendsTo_neg {a : ℕ → ℝ} {t : ℝ} (ha : TendsTo a t) : TendsTo (fun n ↦ -a n) (-t) := by
  unfold TendsTo
  intro ε hε
  specialize ha ε hε
  rcases ha with ⟨n₀, hn₀⟩
  use n₀
  intro n hn
  specialize hn₀ n hn
  simp
  -- what theorems should I use?
  rw [← abs_neg, add_comm]
  simp
  -- what theorems should I use?
  rw [← sub_eq_add_neg]
  exact hn₀
```

`+` commutes with `tendsTo`

```
theorem tendsTo_add {a b : ℕ → ℝ} {A : ℝ} {B : ℝ} (ha : TendsTo a A) (hb : TendsTo b B) :
    TendsTo (fun n ⇒ a n + b n) (A + B) := by
  intro ε hε
  specialize ha (ε / 2) (by linarith only [hε])
  specialize hb (ε / 2) (by linarith only [hε])
  rcases ha with ⟨n₀, ha⟩
  rcases hb with ⟨m₀, hb⟩
  use max n₀ m₀
  intro n hn
  -- what theorems should I use?
  rw [max_le_iff] at hn
  specialize ha n (by linarith only [hn.left])
  specialize hb n (by linarith only [hn.right])
  simp
  -- common tactic: eliminate abs to make use of `linarith`
  -- what theorems should I use?
  rw [abs_lt] at ha hb ⊢
  constructor
  · linarith only [ha, hb]
  · linarith only [ha, hb]
```

[EXR] - commutes with `tendsTo`

```
theorem tendsTo_sub {a b : ℕ → ℝ} {A B : ℝ} (ha : TendsTo a A) (hb : TendsTo b B) :
    TendsTo (fun n ⇒ a n - b n) (A - B) := by
  haveI := tendsTo_add ha (tendsTo_neg hb)
  -- [TODO] `congr` closes the goal directly here. Find out why.
  ring_nf at this
  exact this
```

≤ version of `TendsTo` is equivalent to the usual `TendsTo`.

```
def TendsTo_le (a : ℕ → ℝ) (t : ℝ) : Prop :=
  ∀ ε > 0, ∃ n₀ : ℕ, ∀ n, n₀ ≤ n → |a n - t| ≤ ε

-- [EXR]
theorem tendsTo_le_iff_TendsTo {a : ℕ → ℝ} {t : ℝ} : TendsTo_le a t ↔ TendsTo a t := by
  constructor
  · intro h ε hε
    rcases h (ε / 2) (by linarith [hε]) with ⟨n₀, hn₀⟩
    use n₀
    intro n hn; specialize hn₀ n hn
    linarith only [hn₀, hε]
  · intro h ε hε
    rcases h ε hε with ⟨n₀, hn₀⟩
    use n₀
    intro n hn; specialize hn₀ n hn
    linarith only [hn₀, hε]
```

a weaker version of `TendsTo` where we require `ε < l`. When `l > 0`, this is equivalent to `TendsTo`.

```
def TendsTo_εlt (a : ℕ → ℝ) (t : ℝ) (l : ℝ) : Prop :=
  ∀ ε > 0, ε < l → ∃ n₀ : ℕ, ∀ n, n₀ ≤ n → |a n - t| < ε

theorem tendsTo_εlt_iff_TendsTo {a : ℕ → ℝ} {t : ℝ} {l : ℝ} (l_gt_zero : l > 0) :
    TendsTo_εlt a t l ↔ TendsTo a t := by
  constructor
  · intro h ε hε
    specialize h (min ε (l / 2))
                 (by apply lt_min; all_goals linarith)
                 (by apply min_lt_of_right_lt; linarith only [l_gt_zero])
    rcases h with ⟨n₀, hn₀⟩; use n₀
    intro n hn; specialize hn₀ n hn
    rw [lt_min_iff] at hn₀
    exact hn₀.left
  · exact fun h ε hε _ ↦ h ε hε
```

∗ commutes with `tendsTo`. [TODO] I failed to finish the proof swiftly. You are welcome to optimize it!

```
theorem tendsTo_mul {a b : ℕ → ℝ} {A B : ℝ} (ha : TendsTo a A) (hb : TendsTo b B) :
    TendsTo (fun n ↦ a n * b n) (A * B) := by
  rw [← tendsTo_εlt_iff_TendsTo (show 1 > 0 by linarith)]
  intro ε hε hεlt1; simp
  specialize ha (ε / (3 * (|B| + 1))) (by
    apply div_pos hε
    linarith only [abs_nonneg B])
  rcases ha with ⟨n₁, ha⟩
  specialize hb (ε / (3 * (|A| + 1))) (by
    apply div_pos hε
    linarith only [abs_nonneg A])
  rcases hb with ⟨n₂, hb⟩
  use max n₁ n₂
  intro n hn
  rw [max_le_iff] at hn
  specialize ha n hn.left
  specialize hb n hn.right
  rw [show a n * b n - A * B = (a n - A) * (b n - B) + A * (b n - B) + B * (a n - A) by ring]
  repeat grw [abs_add]
  repeat grw [abs_mul]
  grw [ha, hb]
  -- sometimes you have no choice but add some manual steps
  have h1 : |A| * (ε / (3 * (|A| + 1))) < ε / 3 := by
    field_simp
    rw [div_lt_iff₀]
     · ring_nf
       linarith only [hε]
     · linarith only [abs_nonneg A]
  have h2 : |B| * (ε / (3 * (|B| + 1))) < ε / 3 := by
    field_simp
    rw [div_lt_iff₀]
     · ring_nf
```

```
      linarith only [hε]
    · linarith only [abs_nonneg B]
  have h3 : ε / (3 * (|B| + 1)) * (ε / (3 * (|A| + 1))) < ε / 3 := by
    field_simp
    rw [div_lt_iff₀]
    · repeat grw [← abs_nonneg]
      ring_nf
      calc
        _ = ε * ε := by ring
        _ ≤ 1 * ε := by grw [← hεlt1]
        _ = ε     := by ring
        _ < ε * 3 := by linarith only [hε]
    · repeat grw [← abs_nonneg]
      ring_nf
      linarith only
  linarith only [h1, h2, h3]
```

squeeze theorem for sequences

```
theorem tendsTo_sandwich {a b c : ℕ → ℝ} {L : ℝ} (ha : TendsTo a L) (hc : TendsTo c L)
    (hab : ∀ n, a n ≤ b n) (hbc : ∀ n, b n ≤ c n) : TendsTo b L := by
  unfold TendsTo
  intro ε hε
  specialize ha ε hε
  specialize hc ε hε
  rcases ha with ⟨n₀, hn₀⟩
  rcases hc with ⟨m₀, hm₀⟩
  use max n₀ m₀
  intro n hn
  rw [max_le_iff] at hn
  specialize hab n
  specialize hn₀ n (by linarith only [hn.left])
  specialize hm₀ n (by linarith only [hn.right])
  specialize hbc n
  rw [abs_lt] at hn₀ hm₀ ⊢
  constructor
  · linarith only [hn₀, hm₀, hbc, hab]
  · linarith only [hn₀, hm₀, hbc, hab]
```

constant sequence tends to zero iff condition

```
theorem tendsTo_zero_iff_lt_ε {x : ℝ} : TendsTo (fun _ ↦ x) 0 ↔ (∀ ε > 0, |x| < ε) := by
  constructor
  · intro h ε hε
    specialize h ε hε
    rcases h with ⟨n₀, hn₀⟩
    specialize hn₀ n₀ (by linarith only)
    simp at hn₀; exact hn₀
  · intro h
    intro ε hε
    specialize h ε hε
    use 0
    intro n hn
    simp; exact h
```

[EXR] zero sequence tends to x iff condition

```
theorem zero_tendsTo_iff_lt_ε {x : ℝ} : TendsTo (fun _ ↦ 0) x ↔ (∀ ε > 0, |x| < ε) := by
  constructor
  · intro h
    unfold TendsTo at h; simp at h
    intro ε hε
    specialize h ε hε
    rcases h with ⟨n₀, hn₀⟩
    specialize hn₀ n₀ (by linarith only)
    exact hn₀
  · intro h
    intro ε hε
    use 0
    intro n hn
    simp
    exact h ε hε
```

uniqueness of limits

```
theorem tendsTo_unique (a : ℕ → ℝ) (s t : ℝ) (hs : TendsTo a s) (ht : TendsTo a t) : s = t := by
  by_contra! hneq
  have hstp : 0 < |t - s| := by
    rw [abs_pos]
    contrapose! hneq
    apply_fun fun x ↦ x + s at hneq
    simp at hneq
    symm
    exact hneq
  have hst := tendsTo_sub hs ht
  simp at hst
  rw [zero_tendsTo_iff_lt_ε] at hst
  specialize hst |t - s| hstp
  rw [abs_sub_comm] at hst
  linarith only [hst]

def contAt (f : ℝ → ℝ) (x₀ : ℝ) : Prop :=
  ∀ ε > 0, ∃ δ > 0, ∀ x, |x - x₀| < δ → |f x - f x₀| < ε

def cont (f : ℝ → ℝ) : Prop := ∀ x₀ : ℝ, contAt f x₀
```

continuity of function composition

```
def contAt_comp {f g : ℝ → ℝ} {x₀ : ℝ} (hf : contAt f (g x₀)) (hg : contAt g x₀) :
    contAt (f ∘ g) x₀ := by
  intro ε hε
  rcases hf ε hε with ⟨δf, hδf, hf⟩
  rcases hg δf hδf with ⟨δg, hδg, hg⟩
  use δg, hδg
  intro x hx
  simp only [Function.comp_apply]
  specialize hg x hx
  specialize hf (g x) hg
  exact hf
```

[EXR] continuity of function composition

```
def cont_comp {f g : ℝ → ℝ} (hf : cont f) (hg : cont g) : cont (f ∘ g) := by
  intro x
  exact contAt_comp (hf (g x)) (hg x)
```

[EXR] continuity implies sequential continuity

```
def tendsTo_of_contAt {f : ℝ → ℝ} {x₀ : ℝ} (hf : contAt f x₀)
    {a : ℕ → ℝ} (ha : TendsTo a x₀) : TendsTo (f ∘ a) (f x₀) := by
  intro ε hε
  rcases hf ε hε with ⟨δ, hδ, hδf⟩
  specialize ha δ hδ
  rcases ha with ⟨n₀, hn₀⟩
  use n₀
  intro n hn
  specialize hn₀ n hn
  specialize hδf (a n) hn₀
  exact hδf
```

The uniform limit of a sequence of continuous functions is continuous.

```
def uconv (f : ℕ → ℝ → ℝ) (f₀ : ℝ → ℝ) : Prop :=
  ∀ ε > 0, ∃ N : ℕ, ∀ n ≥ N, ∀ x : ℝ, |f n x - f₀ x| < ε

theorem cont_of_cont_of_uconv
    (f : ℕ → ℝ → ℝ) (f_cont : ∀ n : ℕ, cont (f n))
    (f₀ : ℝ → ℝ) (h_uconv : uconv f f₀) : cont f₀ := by
  intro x₀ ε hε
  rcases h_uconv (ε / 3) (by linarith only [hε]) with ⟨N, hN⟩
  specialize hN N (by linarith)
  rcases f_cont N x₀ (ε / 3) (by linarith only [hε]) with ⟨δ, hδ, hδf⟩
  use δ, hδ
  intro x hx
  specialize hδf x hx
  have hNx := hN x
  have hNx₀ := hN x₀
  -- brute force `linarith` argument
  rw [abs_lt] at hNx hNx₀ hδf ⊢
  constructor
  all_goals linarith only [hNx, hNx₀, hδf]
```

The sequential definition of function continuity is equivalent to the epsilon-delta definition.

```
def contAt_seq (f : ℝ → ℝ) (x₀ : ℝ) : Prop :=
  ∀ a : ℕ → ℝ, TendsTo a x₀ → TendsTo (f ∘ a) (f x₀)
```

[TODO] I failed to solve it swiftly. You are welcome to optimize it!

```
theorem contAt_iff_seq (f : ℝ → ℝ) (x₀ : ℝ) :
    contAt f x₀ ↔ contAt_seq f x₀ := by
  constructor
  · intro hf a ha
    exact tendsTo_of_contAt hf ha
  · contrapose
    intro hnfcont hnfseq
    unfold contAt at hnfcont
    push_neg at hnfcont
    -- construct a sequence `a n` tending to `x₀`
    let a (n : ℕ) : ℝ := 1 / (n + 1)
    have a_gt_zero (n : ℕ) : a n > 0 := by simp [a]; linarith only
    have a_TendsTo_zero : TendsTo a 0 := by
      intro ε hε
      use Nat.ceil (1 / ε) -- ceiling function
      intro n hn
      rw [Nat.ceil_le] at hn
      simp
      rw [abs_of_pos (a_gt_zero n)]
      unfold a
      rw [div_lt_comm₀ (by linarith only) hε]
      linarith only [hn]
    -- construct a diverging sequence `f x` with `x` tending to `x₀`
    -- this requires us to extract `Type*` objects from an existence to form a function
    -- may meet universe issues if done naively
    -- we use `Classical.indefiniteDescription` here to extract such objects classically
    rcases hnfcont with ⟨ε, hε, hnf⟩
    let x_subtype (n : ℕ) := Classical.indefiniteDescription _ <| hnf (a n) (a_gt_zero n)
    let x (n : ℕ) : ℝ := (x_subtype n).val
    have x_lt_a (n : ℕ) : |x n - x₀| < a n := by
      unfold x
      exact (x_subtype n).property.left
    have fx_diverge (n : ℕ) : |f (x n) - f x₀| ≥ ε := by
      unfold x
      exact (x_subtype n).property.right

    have x_tendsTo_x₀ : TendsTo x x₀ := by
      suffices TendsTo (fun n ↦ x n - x₀) 0 by
        have h_add := tendsTo_add this (tendsTo_const x₀)
        simp at h_add; exact h_add
      refine tendsTo_sandwich (?_ : TendsTo (fun n ↦ -a n) 0) (?_ : TendsTo (fun n ↦ a n) 0) ?_ ?_
      · haveI := tendsTo_neg a_TendsTo_zero
        simp at this; exact this
      · exact a_TendsTo_zero
      all_goals
        intro n
        haveI := x_lt_a n
        rw [abs_lt] at this
        linarith only [this]
    -- but it is said that all such sequences converge
    haveI := hnfseq x x_tendsTo_x₀
    rcases this ε hε with ⟨n₀, hn₀⟩
    specialize hn₀ n₀ (by linarith only); simp at hn₀
    specialize fx_diverge n₀
    linarith only [hn₀, fx_diverge]
```