FM-23 Project 3
Geometry of Curves (and not surfaces): a first attempt at their formalisation

CID: 01875107

March 31, 2023

My intention with this project was to formalise the beginning of the Geometry of Curves and Surfaces course at Imperial, currently taught by Davoud Cheraghi. This is the only course in 3rd year level mathematics that I have taken this year, besides this one and one dealing in Logic, so my options for this project were to learn some algebra or algebraic topology to formalise, or work on this. In my personal experience, because algebra and group theory are built from axioms, they lend themselves well to being formalised in Lean, as the level of rigour required doesn't get so difficult to manage as it does dealing with analysis-esque maths. For this exact reason, my second project became about connectedness in topology rather than Banach's fixed point theorem, but I seemed to have forgotten this when I began this project as I became very excited at the prospect of writing my own tactic. This project therefore comes in a few parts: some tactics to help proofs of differentiability and continuous differentiability; some initial definitions of and relating to curves; and some examples and some sublemmas intended to be used in the final theorem. This theorem is that the length of a curve is independent of the parametrisation.

Throughout this document, when discussing the form of goals, I will omit some parameters where this does not add additional information for the reader. For example, I will shorten 'differentiable $\mathbb{R}$ f' to 'differentiable f'. Examples of the use of the tactics and definitions are available in the submitted code.

1 Tactics

An aside on Monad programming for tactics

Up until this project, all of the code I'd written in Lean was actual proofs. I tended to avoid writing definitions as far as possible, because I was unsure of the different use cases of structures, definitions and classes, and what needs including to allow the inference system to be as effective as possible. It's significantly easier to trust that the Mathlib definitions are well-chosen and won't cause unnecessary suffering. I also often avoided using high powered tactics where I didn't feel like I understand precisely what they were doing. The discovery that tactics are a monad in Lean was very enlightening, because it became clear to me exactly how tactics modified the current state and tried many approaches by failing and backtracking. This was exciting as well, because suddenly the door felt more open to me to experiment with writing my own tactics- if I understand

how to program using monads then why not. Unfortunately, writing genuinely useful tactics for Mathlib was still a very significant learning curve from this point, mostly because there is a large, complex set of tactics used by other tactics and I was unfamiliar with their interactions. So I jumped at the opportunity to try my hand at writing some tactics when it became clear to me that I could write a differentiability tactic using the complexity behind the continuity tactic and modifying the 'front' to my own purposes.

Differentiability proofs

In Mathlib, there is a continuity tactic written by Reid Barton, which approximately works as follows:

- If it's possible to apply 'intros' to the goal, do so.
- If the goal matches a lemma tactic with the continuity attribute, then apply this lemma to the goal.
- If the goal can be decomposed into the composition of two functions, where neither of the two are a constant or identity function, then do so, and create two subgoals to prove the continuity of these functions, and recursively apply the tactic to the subgoals.
- Otherwise, fail.

This tactic is quite effective in reducing goals of 'continuity f' into smaller subgoals which are easily solvable. This algorithm is equally appropriate to solve goals of the form 'differentiable', as we already have in Mathlib 'differentiable.comp', which says the composition of two differentiable functions is differentiable (the chain rule), and many lemmas about the differentiability of standard functions.
From this point I created a 'differentiability' tactic, which follows almost exactly the same structure. This involves creating an attribute to tag lemmas with, replacing things in the algorithm like 'continuous_id' with 'differentiable_id', and creating the actual tactic with a docstring. All of this was done in the same style as the continuity tactic, in an attempt to follow Mathlib conventions.

Initial result

This version of the tactic did not do what I expected. Instead of decomposing goals of the form 'differentiable f', it replaced them with '$\forall x$, differentiable_at f x' goals. So, a win for writing my first tactic (I had to learn a lot about meta programming and using monads in order to even begin to modify the continuity tactic in a way which compiled and was available to use on 'differentiable' goals), but as far as I could tell, my tactic was applying some kind of unfolding or rewriting to the goals before trying to use the tagged lemmas or decompose the functions. The tactic uses 'apply_rules' and 'tidy', both of which are tactics in Mathlib, presumably written for the purpose of writing other tactics, so I dug around in the code for those, and could not find a clear reason why either of these would have the effect I was observing. Additionally, these are not tactics which it would be reasonable for me to modify for my purposes- I am competent enough programming with monads to write small tactics, but I am unfamiliar with meta code in the tactic library in Lean 3, and the complexity of some of the pre-written tactics is way beyond something I could understand and modify in the scope of this project. I attempted to investigate myself a little by playing around with reducibility attributes, and restructuring the algorithm slightly but was unsuccessful.

Adaptations from the measurability tactic

The solution to the above problems came from Rémy Degenne on Zulip. They wrote a 'measurability' tactic a year or two prior, and in doing so, ran into precisely the same problems as I encountered above. I included two additional lines of code in my differentiability tactic, and one additional meta function based on their recommendations.

- The continuity tactic could decompose the given function into subgoals well, but could not actually close the subgoals. I modified the differentiability tactic to be able to close the completely decomposed subgoals itself, as done by Rémy for measurability.
- The rewriting from 'differentiable f' to '$\forall x$, differentiable_at f x' was actually caused by the application of intros before the attempted decomposition of the function. My adapted version of Rémy's solution to this was to only apply intros if the goal is not of the form 'differentiable f', which requires an additional meta function.

This algorithm can be seen in the differentiability.lean file submitted, in the 'meta def differentiability_tactics' function. As in continuity and measurability, it's then passed to tactic.tidy, which is responsible for applying it.

Limitations

Given the lemmas I manually tagged with the differentiable attribute, my tactic worked well on functions constructed out of regular operations such as addition, multiplication, and composition, as well as with special functions like exponentials and trigonometric functions. A demonstration of the tactic is given in the file diff_tests.lean with many successful proofs. The only place I was unsatisfied with the results is the below:

```
example (f g h : ℝ → ℝ) (hf : differentiable ℝ f) (hg : differentiable ℝ g)
  (hh : differentiable ℝ h) : differentiable ℝ (λ x, - (h x) + ((f ∘ g) x) ^ 3)
  :=
begin
  differentiability,
  -- Goal: differentiable ℝ (λ (x : ℝ), (f ∘ g) x)
  apply differentiable.comp,
  -- Goals: differentiable ℝ (f ∘ g) and differentiable ℝ (λ (x : ℝ), x)
    { exact differentiable.comp hf hg, },
    { exact differentiable_id, },
end
```

The tactic quits here, because we have specified that if applying differentiable.comp results in one of the functions being the identity, it should not do this. This condition is necessary because it prevents the tactic looping forever, decomposing functions into themselves and the identity. However, in this specific case, where function composition is written with as $(f \circ g)x$ rather than as $f(gx)$, this decomposition is actually necessary to solve the goal. I discussed this with Kevin Buzzard, and was told that people elect not to write $f \circ g$ in Mathlib, instead using the $f(gx)$ notation, which my tactic works successfully for, so I have not attempted to solve this. The same issue I am sure must appear in the measurability and continuity tactics, so I am left to assume that it does not cause substantial problems in the actual use cases.
Note also that I have not literally read through the entire differentiability API to manually tag

every possible lemma which could be helpful with my new attribute, so there are potentially many other types of functions which could easily be handled by this tactic if I were to be comprehensive about the 'differentiability' tactic. This seemed to be a waste of time for the purposes of this project. The lemmas I found and tagged myself are also all visible in the 'differentiability.lean' file.

Continuous differentiability

Having succeeded with my foray into tactics, and knowing that most of the Geometry I intended tackling required continuous differentiability, I duplicated my differentiability tactic to a cont_differentiability tactic, which solves goals of the form 'cont_diff_on f' (NOT cont_diff f'). An application of this on an example created by Kevin with a manual proof is given in the 'cont_diff_tests.lean' file, with an interesting situation where it fails to apply the initial 'cont_diff_on.prod'. I never figured out why this was the case, but once you apply this initially, the tactic does well in the rest of the proof.

```
noncomputable def ϕ₁ : ℝ → ℝ × ℝ × ℝ :=
λ x, (real.sin x, x^4+37*x^2+1, abs x)

example : cont_diff_on ℝ ⊤ ϕ₁ (set.Icc 0 1) :=
begin
  apply cont_diff_on.prod,
  cont_differentiability,
    { exact cont_diff.cont_diff_on cont_diff_sin, },
    { exact cont_diff_on_id, },
    { exact cont_diff_on_id, },
    { apply cont_diff_on.congr cont_diff_on_id,
      intros x hx, exact abs_of_nonneg hx.1, },
end
```

The first example shows how far an earlier version of my continuously differentiable tactic could get, which I include in the report because of an interesting observation I found. There was some asymmetry in closing the goals: A goal of the form 'cont_diff_on id' could be solved by a proof of 'cont_diff_on id' and 'cont_diff_on (λ x, x)', but a goal of the form 'cont_diff_on (λ x, x)' could only be solved by a proof of 'cont_diff_on (λ x, x)', and *not* 'cont_diff_on id'. This is precisely why we have two versions of these lemmas in the library for many propositions, and required me to write this additional lemma to solve this. The second example shows how far the final version of my tactic could get.

```
example : cont_diff_on ℝ ⊤ ϕ₁ (set.Icc 0 1) :=
begin
  apply cont_diff_on.prod,
  cont_differentiability,
    { exact cont_diff.cont_diff_on cont_diff_sin, },
    { apply cont_diff_on.congr cont_diff_on_id,
      intros x hx, exact abs_of_nonneg hx.1, },
end
```

Manual proofs still have to be supplied in this tactic for trigonometric and exponential functions, because these lemmas in Mathlib have the form 'cont_diff sin' and not 'cont_diff_on sin'. It could be argued that these lemmas should be in Mathlib, or that the tactic should decompose proofs of 'cont_diff_on f' to 'cont_diff f' by applying 'cont_diff.cont_diff_on' if it fails to close a goal or even that it's reasonable to leave these to the person writing the proof to finish. I was going to have an experiment with modifying the tactic to apply 'cont_diff.cont_diff_on', but this project quickly became large once I started to actually try to prove some lemmas, and I did not have the time to try to make these modifications. Note that the measurability tactic solves goals of three different forms in one tactic, so it's possible that by enhancing my tactic to solve goals of the form 'cont_diff' as well as 'cont_diff_on', the same modifications as in the measurability tactic would resolve this. I did not have time to investigate this fully.

My testing of the continuous differentiability tactic was less involved, as I am reasonably confident in my understanding that it works in exactly the same way as my differentiability tactic. I have only tagged the lemmas with the cont_differentiability attribute if my tactic needed them in proofs I've used myself, but again a complete version of this tactic would potentially involve tagging equivalent lemmas of more than 100 tagged with the continuity attribute. See Appendix.

Further extensions

It seems a little odd to me, that exactly the same tactic algorithm is used to solve goals of the form 'continuous', 3 forms of measurability, 'differentiable' and 'cont_diff_on'. The same algorithm could easily be used for 'differentiable_at', 'differentiable_on', 'cont_diff', 'differentiable_within', etc etc. At some point, it becomes far too much code duplication to be writing a separate tactic for each one. The approach that Rémy Degenne took by solving several measurability shaped goals in one tactic is an improvement, but doing this for differentiability could involve manually finding and adding the differentiability attribute to potentially hundreds of lemmas of several different forms, and even then the code duplication involved in using the same tactic on continuity, differentiability and measurability begins to seem too much. If Lean3 weren't to be replaced by Lean4 in the near future, I would argue that it would be much better programming practice to write a 'decomposition' tactic, to handle all of these. However, it's true that my tactics are already quite slow, so it's possible this hasn't been done because it's too slow for practical use. Additionally, it seems that tactics are written quite differently in Lean4, so this may no longer be applicable after the port.

2 Maths!

With all this investigation into writing tactics behind me, I wanted to see if it would actually help me formalise some geometry in Lean. My goal here was to formalise the whole first chapter of the Geometry of Curves and Surfaces course, but this was much too ambitious. Here's what I did formalise.

Regular curves

The relevant definitions from Davoud's course notes go as follows:

A parametrised curve in $\mathbb{R}^n$ is a smooth map ϕ : [a, b] $\to \mathbb{R}^n$. The parametrised curve ϕ is called regular, if for all $t \in [a, b]$, we have $|\phi'(t)| = 0$. Recall that a map from $[a, b]$ to $\mathbb{R}^n$ is called smooth, if for each component of the map, the partial derivatives of all orders exist and are continuous.

My Lean definitions:

```
structure regular_curve (n : ℕ) (f : ℝ → euclidean_space ℝ (fin n)) (a b : ℝ) :=
( cont_diff : cont_diff_on ℝ ⊤ f (set.uIcc a b) )
( nezero_deriv : ∀ t : ℝ, t ∈ set.uIcc a b → norm (fderiv ℝ f t) ≠ 0 )

structure regular_curve2 (f : ℝ → ℝ × ℝ) (a b : ℝ) : Prop :=
( cont_diff : cont_diff_on ℝ ⊤ f (set.uIcc a b) )
( nezero_deriv : ∀ t : ℝ, t ∈ set.uIcc a b → norm (fderiv ℝ f t) ≠ 0 )

structure regular_curve3 (f : ℝ → ℝ × ℝ × ℝ) (a b : ℝ) : Prop :=
( cont_diff : cont_diff_on ℝ ⊤ f (set.uIcc a b) )
( nezero_deriv : ∀ t : ℝ, t ∈ set.uIcc a b → norm (fderiv ℝ f t) ≠ 0 )
```

There are several things of note here. The first, is that although (I think) the best way to prove things about regular_curves with better generality is to work with euclidean spaces of finite dimension, defining actual examples on these is nasty, because these are definitionally equal to a function from n to $\mathbb{R}$ (see geometry.lean), and defining functions by their coordinate maps gets messy quickly. For this reason (and because this course only deals in examples in 2 and 3 dimensions), I have included an equivalent definition of regular_curves in 2 and 3 dimensions, using the product spaces $\mathbb{R} \times \mathbb{R}$ and $\mathbb{R} \times \mathbb{R} \times \mathbb{R}$.
The second important thing to note is that the norm Mathlib finds automatically on the product space is different to that on the euclidean space. The norm Lean finds in both cases is the norm of a continuous map defined as the infimum of it's bounds, but this depends on the underlying norm of the image space (either a euclidean_space $\mathbb{R}$ (fin n) who's norm is the typical euclidean norm or a product space $\mathbb{R} \times \mathbb{R}$ or $\mathbb{R} \times \mathbb{R} \times \mathbb{R}$ which finds a norm using the pointwise supremum). Note that although the norms may differ for elements which are nonzero, they must coincide on their zeros, and therefore the two definitions are equivalent.

Examples

5 examples were given in the course notes.

```
-- The below line segment, circle and helix are all regular curves on
-- [0, 1], [0, 2π] and [0, 6π] respectively.
-- The reducible just means I do not have to unfold it myself before applying my tactics
-- Trig functions are noncomputable in Mathlib.
@[reducible] def ϕ₁ : ℝ → ℝ × ℝ := λ t, (2 * t - 1, 3 * t + 2)

@[reducible] noncomputable def ϕ₂ (r : ℝ) : ℝ → ℝ × ℝ
  := λ θ, (r * real.cos θ, r * real.sin θ)

@[reducible] noncomputable def ϕ₃ : ℝ → ℝ × ℝ × ℝ := λ θ, (real.cos θ, real.sin θ, θ)
```

My continuous differentiability tactic successfully tackles the cont_diff proposition of regular_curves, and the only parts of this proof which have to be manually supplied are of the form 'cont_diff.cont_diff_on real.cont_diff_cos' (or equivalent for sin). Observe here that the tactic applies 'cont_diff_on.prod' successfully on $\mathbb{R}^3$, and so this is unlikely to be the cause of the problem given in the discussion of the tactic earlier.

```
-- And some curves which are not regular
@[reducible] def ϕ₄ : ℝ → ℝ × ℝ := λ x, (x, |x|)

@[reducible] def ϕ₅ : ℝ → ℝ × ℝ := λ x, (0, x^2)
```

The first of these is not even differentiable at 0, and the second fails the nonneg_deriv condition at 0. A partial proof of the first and a complete proof of the second are available in the geometry.lean file.

Parametrisation, tangents and lengths

I include in this report the relevant definitions for the later proofs, i.e. parametrisation, tangent lines and the length of a regular curve, all using the definition of regular curves with a euclidean space rather than a direct product. Again, alternative definitions using $\mathbb{R} \times \mathbb{R}$ and $\mathbb{R} \times \mathbb{R} \times \mathbb{R}$ can be found in the geom2.lean file, for use with examples of regular_curve2's and regular_curve3's.

```
noncomputable def tangent_line {n : ℕ} {f : ℝ → euclidean_space ℝ (fin n)} {a b : ℝ}
  (h : regular_curve n f a b) (t₀ : ℝ) : ℝ → euclidean_space ℝ (fin n)
  := λ t, f t₀ + ((fderiv ℝ f t₀) t)

def is_parametrisation {n : ℕ} {a b c d : ℝ} {ψ : ℝ → euclidean_space ℝ (fin n)}
  {ϕ : ℝ → euclidean_space ℝ (fin n)} (h1 : regular_curve n ψ c d)
  (h2 : regular_curve n ϕ a b)
  := ∃ f : ℝ → ℝ, cont_diff_on ℝ ⊤ f (set.uIcc c d) ∧ f '' {c, d} = {a, b} ∧
    (∀ t : ℝ, t ∈ set.uIcc c d → norm (fderiv ℝ f t) ≠ 0) ∧
    (∀ t, t ∈ set.uIcc c d → ϕ ∘ f = ψ)

def parametrises {n : ℕ} {a b c d : ℝ} {ψ : ℝ → euclidean_space ℝ (fin n)}
  {ϕ : ℝ → euclidean_space ℝ (fin n)} (h1 : regular_curve n ψ c d)
  (h2 : regular_curve n ϕ a b) (f : ℝ → ℝ)
  := cont_diff_on ℝ ⊤ f (set.uIcc c d) ∧ f '' {c, d} = {a, b} ∧
    (∀ t : ℝ, t ∈ set.uIcc c d → norm (fderiv ℝ f t) ≠ 0) ∧
    (∀ t, t ∈ set.uIcc c d → ϕ ∘ f = ψ)

noncomputable def length {n : ℕ} {f : ℝ → euclidean_space ℝ (fin n)} {a b : ℝ}
  (h : regular_curve n f a b) : ℝ
  := ∫ t in a..b, norm (fderiv ℝ f t)
```

Set.uIoo

I spent a long time trying to prove some statements which felt blindingly obvious to me, and yet eluded me in Lean. I often managed to prove some variant of the statement I wanted, with

additional conditions, with led me look deeper into whether what I was doing was correct, and I found that I had made a potential error with my definitions. I initially defined regular curves using set.Icc as the underlying set for the curve to be continuously differentiable on. The cause of my problems was the following distinction:
Differentiability on a set in Lean, to my understanding, is equivalent to our usual definition of differentiability as a limit, when considering *only* sequences contained within the set. This allows us to have a sensible definition of differentiability at the end points of a closed set. The implication of this is that for some function f and $\forall a < b$, differentiable_on f (set.Icc a b) is a sensible statement, and differentiable_on f (set.Icc b a) is a statement about the empty set. (As set.Icc b a $= \emptyset \neq$ set.Icc a b).
Integrability in Lean is implemented using measure theory (not the Riemann integral given in early year undergraduate courses) and is also defined on a set. Integrals of a function on $\mathbb{R}$ are equivalent to the Riemann integral, where for some function f and $a < b$, $\int t$ in $a..b, f\ t = -\int t$ in $b..a, f\ t$, as we'd expect having studied analysis at undergraduate level.
The implication of this in defining the length function on a regular curve, is that we need the additional hypothesis $a \leq b$ for the interval on which a regular curve is continuously differentiable and has a nonzero derivative. After some consideration, it seemed much more sensible to use set.uIcc as the underlying set, rather than require an additional hypothesis that $a \leq b$, as is done in the measure theory implementations of integrals, so I modified my definitions this way, and then promptly discovered that there did not exist an equivalent set.uIoo in Mathlib. I was informed by someone at Xena that this was likely just because no one had ever had a legitimate use case for them, so I have just written a quick definition and the parts of the API that I actually needed and submitted those in a separate file along with the rest of this project. See unordered_interval.lean, which follows an almost identical structure to the file of the same name in Mathlib.

The length of a curve is invariant under parametrisation

This was the major theorem I wanted to prove, and it turned out to be completely unfeasible in the time I had left to work on this project. A large bulk of my time was spent learning about tactics, and debugging my own, so by the time I began to actually write some proofs, I quickly lost a lot of time to misjudged definitions and only managed to prove some of the sublemmas required for this proof, rather than to complete it. The maths statement and proof are the following (taken from the course notes):

> Let $\phi : [a, b] \to \mathbb{R}^n$ be a regular curve, and $f : [c, d] \to [a, b]$ be a smooth function with $f'(s) \neq 0$, $\forall s \in [c, d]$, and $f(\{c, d\}) = \{a, b\}$. Then $\psi = \phi \circ f : [c, d] \to \mathbb{R}^n$ is another parametrisation of $\phi([a, b])$. We claim that the length of $\phi([a, b])$ is the same as the length of $\psi([c, d])$.
> Since $f'(s) \neq 0$, $\forall s \in [c, d]$, and f' is continuous on $[c, d]$, either $f' > 0$ on $[c, d]$ or $f' < 0$ on $[c, d]$. Without loss of generality, let us assume that $f' > 0$ on $[c, d]$. Then, $f(c) = a$ and $f(d) = b$. By the chain rule, we find $|\psi'(s)| = |(\phi \circ f)'(s)| = |f'(s)||\phi'(f(s))|$. Therefore, by the change of variable formula, $\ell(\psi([c, d])) = \int_c^d |\psi'(s)|ds = \int_c^d |f'(s)||\phi'(f(s))|ds = \int_{f(c)}^{f(d)} |\phi'(t)|dt = \ell(\phi([a, b]))$.

I successfully proved the first sentence (see lemma *continuous_on.all_lt_or_all_gt_of_ne*), and intended to proceed by supplying more or less identical proofs for the cases $f' > 0$ and $f' < 0$ as this

is somewhat easier in Lean. Lemma *one* has the proofs of the equalities following this for both cases. However I completed all of these proofs before I discovered the problem with unordered intervals and modified my definitions, so these proofs in some cases have the assumptions that $a \leq b$ and/or $c \leq d$. I suspect it would still be possible to apply these lemmas to the proof of the theorem without modification by passing $a \sqcup b$ and $a \sqcap b$ as parameters instead of a and b.
An almost complete proof of the equality by the chain rule can be seen in the lemma *deriv_of_parametrisation*. The missing section is unrelated to derivatives and I felt it best to focus on making progress on the rest of the proof.
The place this proof really fell down from here was the application of the change of variables formula. This requires some assumptions on the continuity of the functions which I could not prove, and were potentially not even true. In order to apply the change of variables formula, I needed to be able to prove continuity of the derivative of the regular curve everywhere in one place, and at the end points of the closed intervals I was using elsewhere, and I could only prove it for open sets within this closed interval. I suspect the most Mathlib-y way to recover from this is to require that regular curves are continuously differentiable everywhere.

Also in the geom2.lean file are other smaller lemmas I needed or felt I needed as part of proofs, or I felt I needed to use multiple times and therefore should be in Mathlib.

Reflection

Despite many things feeling unfinished with this project, I am quite proud of it, as I feel it really is demonstrative of how much I've learnt taking this course. I feel not only confident in my abilities to prove quite difficult maths in Lean given enough time, but that I have a much better understanding now of why definitions and API's are set up in the way they are in Mathlib, from the perspectives of using the lemmas in tactics, as part of written proofs and to aid the inference system. The place to go from here to make me a better Lean coder would definitely be in my understanding in the details of some of the type theory, as I am often unsure in using universes and types across those. This is also the first time I have tried to do something myself which is not already in Mathlib- both with my new tactics and also my definitions and proofs. It definitely caused a lot of headaches, because being able to look inside Mathlib for guidance on the best way to form definitions is a huge help, but it also led me to a much deeper understanding of the choices made inside Mathlib's definitions. I find myself finally beginning to have thoughts along the lines of "oh this is the right way to do something", as opposed to "oh maybe this might work". I think that's a cool place to be :)

References

1. Zhou, Zhouhang (2020), Mathlib [Source code].
   https://github.com/leanprover-community/mathlib/blob/master/src/data/set/intervals/unordered_interval.lean

2. Barton, Reid (2020), Mathlib continuity tactic [Source code].
   https://github.com/leanprover-community/mathlib/blob/master/src/topology/tactic.lean

3. Degenne, Rémy (2021), Mathlib measurability tactic [Source code]. https://github.com/leanprover-community/mathlib/blob/master/src/measure_theory/tactic.lean

4. Cheraghi, Davoud (2023), *Geometry of curves and surfaces*, lecture notes, Imperial College London

5. Multiple authors, (March 2023), parametrised curves [Zulip thread]

6. Multiple authors (March 2023), differentiability tactic [Zulip thread]

7. leanprover community (2020), *Videos 1-6*, Metaprogramming in Lean tutorial. [Online video] Available at: https://www.youtube.com/watch?v=o6oUjcE6Nz4

Appendix: A full set of attributes used by the continuity tactic

It's likely that a complete differentiability (equiv cont_diff, cont_diff_on, differentiable_at, etc) tactic would require manually finding and tagging equivalent lemmas to the majority of the below, as well as potentially others depending on the design of the API.

- continuous_multilinear_map.coe_continuous,
- linear_isometry.continuous,
- semilinear_isometry_class.continuous,
- continuous_algebra_map,
- affine_map.homothety_continuous,
- affine_map.line_map_continuous,
- path.truncate_const_continuous_family,
- path.truncate_continuous_family,
- path.continuous_trans,
- continuous.path_trans,
- path.trans_continuous_family,
- path.continuous_uncurry_extend_of_ continuous_family,
- path.continuous_symm,
- path.symm_continuous_family,
- path.continuous_extend,
- continuous.path_eval,
- path.continuous,
- unit_interval.continuous_symm,
- continuous.Icc_extend',
- continuous_proj_Icc,
- continuous_linear_equiv.continuous,
- continuous_linear_equiv.continuous_inv_fun,
- continuous_linear_equiv.continuous_to_fun,
- continuous_linear_map.continuous,
- continuous.sqrt,
- real.continuous_sqrt,
- continuous.edist,
- ennreal.continuous_pow,
- continuous_nnnorm,
- continuous_nnnorm',
- continuous_norm,
- continuous_norm',
- continuous.zpow$_0$,
- continuous.div,
- continuous.inv$_0$,
- continuous.div_const,
- continuous_abs,
- continuous.star,
- continuous.dist,
- continuous_dist,
- continuous.max,
- continuous.min,
- continuous.sub,
- continuous.div',
- continuous.zsmul,

- continuous.zpow,
- continuous_zsmul,
- continuous_zpow,
- continuous.inv,
- continuous_finset_sum,
- continuous_finset_prod,
- continuous_multiset_sum,
- continuous_multiset_prod,
- continuous.nsmul,
- continuous_nsmul,
- continuous_map.continuous_set_coe,
- continuous_map_class.map_continuous,
- continuous.vadd,
- continuous.const_vadd,
- continuous.const_smul,
- uniform_equiv.continuous_symm,
- uniform_equiv.continuous,
- add_opposite.continuous_op,
- mul_opposite.continuous_op,
- add_opposite.continuous_unop,
- mul_opposite.continuous_unop,
- homeomorph.continuous_symm,
- homeomorph.continuous,
- continuous.connected_components_lift_continuous,
- connected_components.continuous_coe,
- continuous_ulift_up,
- continuous_ulift_down,
- continuous.sigma_map,
- continuous_sigma,
- continuous_sigma_mk,
- continuous_single,
- continuous_mul_single,
- continuous_update,
- continuous_apply_apply,
- continuous_apply,
- continuous_quot_lift,
- continuous_quot_mk,
- continuous.cod_restrict,
- continuous.subtype_mk,
- continuous_subtype_val,
- continuous.sum_map,
- continuous.sum_elim,
- continuous_inr,
- continuous_inl,
- continuous.prod.mk_left,
- continuous.prod.mk,
- continuous.prod_mk,
- continuous_top,
- continuous_bot,
- continuous_induced_dom,