FORMALISING THE BSD CONJECTURE

JAMIE BELL

Date: June 2021.

ABSTRACT. We aim to express the Birch–Swinnerton-Dyer conjecture in Lean theorem proving language, giving all definitions and the proofs of some intermediate results. We achieve this for elliptic curves over $\mathbb{Q}$, and for the algebraic rank of curves over number fields, but not the analytic rank of curves over number fields.

I would like to thank Kevin Buzzard for suggesting and supervising this project, especially for his frequent help fixing my code and pointing me in the right direction whenever I got stuck.

CONTENTS

1. INTRODUCTION

1.1. **The code.** The Lean code for this project is available in a repository at https://github.com/jamiebell2805/BSD-conjecture/. Specifically, this report refers to the commit 74614b2123d7397de60de8e771cefde1d8af66e0 on 24/06/2021.

1.2. **The mathematics.** The Birch–Swinnerton-Dyer conjecture is one of the Clay Institute's Millennium problems. It relates the rank of an elliptic curve to its L-function. The official statement [7] of the problem refers only to elliptic curves over $\mathbb{Q}$, but the conjecture is also believed to hold for elliptic curves over general number fields. This conjecture dates back to the work of Birch and Swinnerton-Dyer in the 1960s, when they spotted a potential link between the rank of an elliptic curve and its number of points mod p. This was soon adapted to the form we write about now, which links the rank to

the L-function of the elliptic curve. The conjecture has been verified in many cases, but not proven in general.

Formally, an elliptic curve is a non-singular genus 1 curve with a distinguished point [1]. There are many choices of coordinates, giving different models of the curve. We will work with a Weierstrass model of the form $y^2 = x^3 + ax + b$ where a and b are integers in the rational case and elements of the number field in the general case. A point at infinity is also included, which we may think of as being infinitely far up the y-axis. Any elliptic curve can be written in this form. Non-singularity is guaranteed by the non-vanishing of the discriminant, $\Delta = -16(4a^2 + 27b^3)$.

The usefulness of elliptic curves comes from the fact that their points form an abelian group, with identity the point at infinity. To add points not at infinity, take the line between the two, or the tangent if the points are the same. If this is vertical, the sum is the point at infinity. Otherwise, this will intersect the curve at a third point. Crucially, if the points we are adding have coordinates in a certain field, the third point will too. Now take a line from the point at infinity to this point, in our setting a vertical line. The other intersection of this line with the curve is the sum of the initial two points.

In this group law, the point at infinity is the identity, and for any other point (x, y) on the curve, $-(x, y) = (x, -y)$. It is simple to convince yourself that the identity, inverses and commutativity do hold for this operation. It is more challenging to show associativity directly as lots of algebra and special cases are required. This can be avoided by showing that the points are in bijection with elements of $\mathrm{Pic}^0(E)$, which is the group of degree 0 divisors of E quotiented by the group of principal divisors. The map $P \mapsto (P) - (0)$ is a homomorphism, which implies that the points form a group. In this project we will sidestep this issue by not attempting a proof.

The key theorem about this group is

Theorem 1.1 (Mordell-Weil)**.** *The points on the curve form a finitely-generated abelian group. This is therefore isomorphic to $\mathbb{Z}^r \times \Delta$, for some finite group Δ.*

We call r in this theorem the rank of the elliptic curve, or sometimes the algebraic rank. Computing this is often important, but can be rather difficult - there is no known algorithm to do so. In this project we do not prove that the group is finitely generated, but we do define the rank of a finitely-generated abelian group.

We can also study these elliptic curves over finite fields. For curves over $\mathbb{Q}$, with coefficients in $\mathbb{Z}$, we can reduce them modulo p. If p does not divide the discriminant of E, we get an elliptic curve over $\mathbb{F}_p$. For curves over a general

number field, we must use an appropriate model for the curve in order to reduce a and b modulo our prime, but if we can we again get an elliptic curve over the residue field. We call these primes not dividing Δ good primes.

For each good prime v of the number field, call the residue field k_v, and the reduced elliptic curve $\tilde{E}_v$. Let $q_v = \#k_v$. Then we define $a_v = q_v + 1 - \#\tilde{E}_v(k_v)$.

Definition 1.2 (Local factors)**.** *If v is a prime of good reduction for E, let $L_v(T) = 1 - a_v T + q_v T^2$.*

Definition 1.3 (L-function)**.** *The incomplete L-function of E is $\prod L_v(q_v^{-s})^{-1}$, where the product is over good primes v of our model E.*

The complete L-function has factors at the other places, but they are not needed to state BSD.

Hasse's theorem is that $|a_v| \leq 2\sqrt{q_v}$. This implies that the L function converges for $s > 3/2$. In fact, getting a weaker bound, such as $|a_v| \leq q_v$, also gives convergence on a smaller half-plane. It is proven for the rationals, and conjectured for number fields, that the L-function has an analytic continuation to $\mathbb{C}$. This allows us to define the analytic rank of E, as $\mathrm{ord}_{s=1} L(E, s)$. We can now state the Birch–Swinnerton-Dyer conjecture:

Conjecture 1.4 (BSD)**.** *The algebraic and analytic ranks of E are equal.*

1.3. **Lean theorem prover.** Lean is a theorem-proving language, initially developed by Leonardo de Moura at Microsoft research in 2013 [6]. The aim is to provide complete proofs of theorems, without the gaps that are present in normal written mathematics. There is a large library of mathematical results, called mathlib [8], starting from definitions and developing ideas completely rigorously. This now covers large amounts of algebra and analysis, as well as smaller amounts of topology, geometry and other areas.

Lean works using type theory. To a mathematician trying to use Lean, it is usually sufficient to think of a type as a set, and its elements are called terms. These terms are not themself types, unlike in set theory where sets only contain more sets. Any term has a single type. This means that the natural number 2 is different from the integer 2, the rational number 2 and the real number 2. This is consistent with the way we usually define these sets in mathematics. For instance, we define rationals as equivalence classes of pairs of integers. Taken literally, the integers are not elements of the rationals, but there is an embedding which means we can view them as such. The embedding maps between these will often need to be written in to the code, usually as $\uparrow 2$.

Another consequence is that types do not intersect. If we define types of integer multiples of 2 and integer multiples of 3, we cannot take an intersection of these to get the multiples of 6. Defining conventional sets is also possible if we wish to do so, but this is not how the underlying code works.

Using Lean can be done in a browser, but for projects it is easier to install a copy. When trying to prove a result in Lean, the Lean infoview window shows what variables we have, what type each of these terms has, what hypotheses we have, and what our goal is. This means it is always clear exactly what we have to show at each step in the proof. We can then use tactics to move towards our goal. Examples of tactics are `rw` (re-write), used to replace an expression with an equivalent one, and `intro`, used to change a $\forall x$ goal into a statement about a certain term x. More complex tactics also exist, such as simplifiers for expressions in rings and fields, the `change` tactic to convert our goal into one of a specified form, and the `tauto` tactic, which applies a truth table to make logical deductions.

1.4. **Previous work.** Elliptic curves have previously been formalised in other proof verification languages, including basic definitions and proofs of the group structure on the curve. It appears that elliptic curves haven't been formalised in Lean before, and there have been no attempts to define anything beyond the group law on the curve, such as the rank or the L-function.

The associative law was first verified in 1998 by Stefan Friedl using the Cocoa algebra system [2]. In 2007, Laurent Théry and Guillaume Hanrot [3] used the same methods to formalise elliptic curves in Coq, and prove the group axioms. In 2014, Evmorfia-Iro Bartzia and Pierre-Yves Strub [4] used a map to the Picard group to show non-computationally that the points form a group, also in Coq. Elliptic curves have also been formalised in Isabelle/HOL, by Thomas Hales and Rodrigo Raya [5]. They use curves in Edwards form rather than Weierstrass form, which has the advantage that addition formulas do not have several cases like Weierstrass addition does. This is useful in cryptographic applications, as well as in proofs. They were able to prove the group law algebraically. The Edwards form means that this requires little more than division of polynomials with remainder.

In this project, we will not give a proof of associativity. This is partly because it has already been done in other theorem provers, and also because the methods we use to prove the other axioms would be too slow to prove associativity. Also, unlike much of the previous work, we mostly restrict to curves over $\mathbb{Q}$ for simplicity. Instead, we aim to take these ideas further, and develop the ideas of ranks and L-functions of elliptic curves in order to state BSD. We aim to state every definition required to do this. The result over $\mathbb{Q}$ is a Lean file with `sorry`s, such that, if these were replaced by proofs, we

would have a complete proof of BSD. Obviously we do not expect that this will happen any time soon! Over number fields we do not get as far with the L-function, and it is in fact an open question whether it always has the required analytic continuation.

2. The algebraic rank

The aim of this section is to define the algebraic rank of an elliptic curve. We will do this over $\mathbb{Q}$ and over a general number field. I will refer to excerpts of the file for $\mathbb{Q}$, and explain where there are differences for the number field case.

We first define a structure called `elliptic_curve`. In order to do this, we need the discriminant.

```
def disc (a b : ℤ) : ℤ :=
-16*(4*a^3+27*b^2)

structure elliptic_curve :=
(a b : ℤ)
(disc_nonzero : disc a b ≠ 0)
```

What we have done here is define a function, the discriminant, from pairs of integers to the integers. An elliptic curve is then given by a pair of integers, together with a proof that the discriminant of a, b is non-zero. These can be recovered from an elliptic curve E. `E.1.1` or `E.a` gives us a, `E.1.2` is b, and `E.2` is the proof that the discriminant is non-zero. Note that this definition doesn't look much like a definition of an algebraic curve! We now move on to define the points of the curve.

```
def finite_points (E : elliptic_curve) :=
{P : ℚ × ℚ // let ⟨x, y⟩ := P in
  y^2  = x^3 + E.a*x + E.b}

def points (E : elliptic_curve) :=
with_zero E.finite_points
```

The idea here is that we make a subtype of $\mathbb{Q} \times \mathbb{Q}$ which contains the finite points of the curve. These finite points consist of a pair of rationals, together with a proof that they satisfy the right equation. We then define the points of the curve, which is this type with the addition of a zero. The way that

type theory works means that a non-zero term of type `points` is not a term of type `finite_points`, even though there is a bijection between them. After defining these, we need to make an API in order to prove theorems. This consists of simple lemmas, mostly with simple proofs. The lemmas tell us things such as 'if two points have the same coordinates, they are the same point' and 'all points are zero or finite'. We also define functions giving the coordinates of finite points.

The next step is to define the group law on the curve. We will do negation first.

```
lemma is_on_curve_neg {x y : ℚ}
(h : E.is_on_curve x y) : E.is_on_curve x (-y) :=

def neg_finite : finite_points E → finite_points E
| P :=
  let ⟨⟨x, y⟩, hP⟩ := P in
  ⟨(x, -y), E.is_on_curve_neg hP⟩

def neg : points E → points E
| 0 := 0
| (some P) := (some (neg_finite E P))
```

We first show that if (x, y) is on the curve, then so its negative, $(x, -y)$. The proof is omitted here but is simple. Next we define a function on the finite points of E, which sends P to its negative. To do this we need the proof that the negative is on the curve, using the lemma above. We then extend this to all points of E.

The next part of defining the group law is the special case of doubling a point.

```
def double : points E → points E
| 0 := 0
| (some P) :=
let ⟨⟨x, y⟩, h⟩ := P in
if h2 : y = 0 then 0 else
  let A : ℚ := E.a in
  let B : ℚ := E.b in
  let d := 2*y in
  let sd := (3*x^2+A) in
  let td := y*d-sd*x in
```

```
  let x₂dd := sd^2-2*x*d^2 in
  let y₂ddd := sd*x₂dd+td*d^2 in
  let y₂' := -y₂ddd/d^3 in
  let x₂ := x₂dd/d^2 in
  some ⟨⟨x₂, y₂'⟩, begin

  end⟩
```

This is done in cases depending on whether the point is zero. Then it checks whether $y = 0$. If it is, the tangent is vertical so the doubled point is zero. Finally we do the case where $y \neq 0$. We define the coordinates of the new point, (x_2, y_2'), and then need to prove that it is on the curve. The proof is omitted here - it would go between `begin` and `end`.

Next we need to define addition in general.

```
def add : points E → points E → points E
| 0 P := P
| P 0 := P
| (some P) (some Q) :=
let ⟨⟨x1, y1⟩, h1⟩ := P in
let ⟨⟨x2, y2⟩, h2⟩ := Q in
if hd : x1 = x2 then
(if y1 = y2 then double E (some P) else 0) else
  let A : ℚ := E.a in
  let B : ℚ := E.b in
  let d := (x1 - x2) in
  let sd := (y1 - y2) in
  let td := y1*d-sd*x1 in
  let x₃dd := sd^2-(x1+x2)*d*d in
  let y₃ddd := sd*x₃dd+td*d*d in
  let x₃ := x₃dd/d^2 in
  let y₃' := -y₃ddd/d^3 in
  some ⟨⟨x₃, y₃'⟩, begin

  end⟩
```

Again we have to break down into special cases when one of the points is 0, when they are equal, or when they have the same x-coordinate. Then we do the general case, and again need to prove this point is on the curve, which

has been omitted. Care was required at this point to do the proof in a way which did not require excessive computation time.

We are now in a place to prove that the points form an abelian group, or, in Lean terminology, an `add_comm_group`.

First we prove two lemmas, `add_zero` and `zero_add`, which state that the identity works as it should. We then prove `add_left_neg`, which states that $(-P) + P = 0$. We do not need to prove the reverse because we will show that addition is commutative.

```
theorem add_comm_finite {x1 x2 y1 y2 : ℚ}
  (h1 : E.is_on_curve x1 y1)
  (h2 : E.is_on_curve x2 y2) :
  E.points_mk h1 + E.points_mk h2 =
  E.points_mk h2 + E.points_mk h1 := begin

  end

theorem add_comm (P Q : points E) :
  P + Q = Q + P := begin

end
```

We first do it for addition of finite points. The function `E.points_mk` takes a proof that (x, y) is on the curve, and outputs the point (x, y). Again the proof is omitted. The proof involves splitting into cases depending on whether the points have the same coordinates. For the number field case, it was necessary to split the theorem into three to avoid a timeout. The general `add_comm` theorem is then just the cases involving zeros, as well as an application of `add_comm_finite`.

We omit the proof of associativity in this project. This has been shown in other formalisation languages. In Lean it is possible to state a theorem and assume it is true, by replacing a proof with the word `sorry`. This means we now have all the axioms of an `add_comm_group`, and can use all the general results and definitions about them from mathlib.

The next step is to show that this group is finitely generated, and to find its rank. The concept of finitely generated groups is in mathlib, but there is no function to give the rank, so we have had to define this ourselves. We do this by taking the quotient by the torsion subgroup to get a free abelian group with rank equal to that of our original group. We the define the rank as the smallest number of generators for this group.

```
theorem fg : add_group.fg (points E) := begin
  sorry,
end

def torsion_points (E : elliptic_curve) :
(set (points E)) :=
{P | ∃ (n : ℤ), (n • P = 0)∧(n ≠ 0)}

def torsion_free (E : elliptic_curve) :=
  (quotient_add_group.quotient (torsion_subgroup E))
```

Here we have stated the Mordell-Weil theorem that the group is finitely generated, and then defined the torsion points. We then have to show that the torsion points form a subgroup of the points (not shown here) before defining the quotient group, `torsion_free`.

One theorem not included in mathlib is that the quotient of a finitely generated group is finitely generated. We showed this result in the file fg_quotient.lean.

```
theorem fg_surj_image_fg: (function.surjective f)
   → (group.fg G) → (group.fg H):= begin

end

theorem quotient_surj:
function.surjective (quotient_group.mk' N) :=
quotient.surjective_quotient_mk'

theorem fg_quotient_of_fg: (group.fg G)
   → (group.fg (quotient_group.quotient N)) := begin

end
```

In this setting, G and H are groups, N a normal subgroup of G, and $f : G \to H$ a homomorphism. The first theorem is that if f is surjective and G is finitely generated, then H is also finitely generated. The key part of this proof is a result from mathlib that the image of the closure of a set is equal to the closure of its image. To apply this, we next show that the quotient map is surjective - the proof is just a general property of quotients. Finally we combine these to show that the quotient group G/N is finitely generated.

These results can be tagged in Lean to also produce equivalent results for additive groups, and so apply to our elliptic curve situation.

Now we apply this to our specific situation.

```
def generators (E : elliptic_curve) :=
  {S : set (torsion_free E) |
  (set.finite S) ∧ (add_subgroup.closure S = T)}

def sizes (E : elliptic_curve) : (set ℕ) :=
  {n : ℕ | ∃ (S : generators E),
  (fintype.card (fintype S)) = n}

theorem sizes_non_empty : ∃ (n : ℕ), (n ∈ sizes E)
  := begin

end
```

The quotient group is called `torsion_free`, and we have proved it is finitely generated. We now define a set, `generators`, of all the finite generating sets of `torsion_free`. Next we define a subset of $\mathbb{N}$ consisting of all the n for which there exists a generating set of size n. We prove that this is non-empty by applying the result that `torsion_free` is finitely generated. This result is needed to apply well-ordering to prove it has a smallest element. We can now make our final definition.

```
noncomputable def rank (E : elliptic_curve) : ℕ :=
  nat.find (sizes_non_empty E)
```

This gives the size of the smallest generating set for `torsion_free`, i.e. the rank of the group of points. This is a noncomputable definition, because Lean does not know how to compute this, but we can still make the definition as we know it exists.

3. The analytic rank

We will now define the analytic rank of a curve. In this project we have only done this for curves over $\mathbb{Q}$. Generalising this to number fields will take more work.

First we define the good primes of the curve, i.e. the ones that do not divide the discriminant. We use the discriminant of a particular model of the curve,

even though there may be primes which do not divide the discriminant of some other model, but it turns out this does not matter for stating BSD. This is in fact how the official Millennium Prize problem is stated.

```
def good_primes := {p : ℕ | nat.prime p ∧
  ¬ (↑p ∣ (disc E.a E.b))}

def p_points (E : elliptic_curve) (p : good_primes E)
 := {P : zmod p × zmod p | let ⟨x, y⟩ := P in
 y^2  = x^3 + E.a*x + E.b}

noncomputable def a_p (E : elliptic_curve)
  (p : good_primes E) : ℤ :=
  p - fintype.card (fintype (p_points E p))
```

Here we have defined the good primes, and the finite points of E over $\mathbb{F}_p$. Note that as we have not counted the infinite point, we can then define a_p as $p - \#\{\texttt{p\_points}\}$, with no $+1$ needed.

```
def half_plane := {z : ℂ // complex.re z > 3/2}

noncomputable def local_factor (E : elliptic_curve)
(s : ℂ) : good_primes E → ℂ
| p := 1 - (a_p E p) * p ^ (-s) + p ^ (1-2*s)

theorem hasse_bound (E :elliptic_curve)
(p : good_primes E) :
  (a_p E p)^2 ≤ 4 * p :=
```

The next thing we define is a type called `half_plane`, consisting of complex numbers with real part greater than $3/2$. We also define the local factors of the L-function. We do this for fixed $s \in \mathbb{C}$, so it is a function from the good primes to $\mathbb{C}$. This will later allow us to take the infinite product, because our definition of infinite product will work for elements of $\mathbb{C}$ rather than functions. We also state the Hasse bound, that $|a_p| \leq 2\sqrt{p}$, but do not prove this here. We have stated it in terms of the squares, so that Lean does not have to turn these integers into real numbers. This bound is necessary to show that the L-function converges on the half-plane, however we do not use it here as we do not prove convergence.

We can also use this to show why it doesn't matter that we are not using a complete L-function. There are finitely many primes which are not good for this model. In a full L-function, they would either be good, in which case we are missing a factor of $1-a_p p^{-s}+p^{1-2s}$, or bad, in which case we are missing a factor of 1, $1-p^{-s}$ or $1+p^{-s}$. What matters is the order of vanishing at $s=1$. This is not changed by multiplying by a function which is non-zero at 1. The factors in the bad case are clearly non-zero at $s=1$. For the good case, we must use the Hasse bound. For the factor to be zero, we must have $p+1=a_p \leq 2\sqrt{p}$. But this is a contradiction: $2\sqrt{x} < x+1$ for all $x > 1$. So our definition of the incomplete L-function will give the same analytic rank.

The next thing required is an infinite product. These are not in mathlib, but infinite sums are. This code is therefore a simple adaptation of the infinite sum code by Johannes Hölzl, in the file tsum.lean [9]. Our code for products is in the file tprod.lean.

```
def has_prod (f : β → α) (a : α) : Prop :=
tendsto (λs:finset β, ∏ b in s, f b) at_top (𝓝 a)

def prodable (f : β → α) : Prop := ∃a, has_prod f a

@[irreducible] def tprod {β} (f : β → α) :=
if h : prodable f then classical.some h else 1
```

Here we are working in the context of any type β, and a type α which is both a commutative monoid and a topological space - for example $\mathbb{C}$. There are a few interesting things to note about this definition. One is that it doesn't impose an order on the terms being multiplied - it only works if the product is absolutely convergent. The definition `has_prod` takes in a function $f : \beta \to \alpha$, and an element $a \in \alpha$, and takes a value of true or false, depending on whether the product of $f(b)$ over $b \in \beta$ converges to a. The terms `tendsto` and `at_top (𝓝 a)` are a definition of convergence in terms of filters. The function given with λ notation takes finite subsets s of β, and calculates the product of $f(b)$ over $b \in s$.

The next definition, `prodable`, is a proposition telling us whether a product converges to some $a \in \alpha$. Then `tprod` gives us this product, if it exists, otherwise gives 1. In our context, we now have:

```
theorem converges (E : elliptic_curve)
(s : half_plane) : prodable (local_factor E s) :=

```

```
noncomputable def L_function_product
(E : elliptic_curve) (s : half_plane) : ℂ :=
1/(tprod (local_factor E s))
```

Here we have stated (but do not prove) that the product of local factors converges in the `half_plane`, and then defined the L-function in this half-plane. Note that we can define this reciprocal without a proof that it is non-zero. In Lean this is because it is a noncomputable definition, but it happens to correspond to the idea that an infinite product cannot converge to zero but rather diverges.

Next we need to define the analytic continuation of the L-function. Proving that this exists for all curves is a consequence of the modularity theorem, and so is well beyond the mathematics that has been formalised so far. As usual, however, we can replace the proof with a `sorry` and keep working.

```
theorem analytic_continuation: ∃ f : ℂ → ℂ,
  (differentiable ℂ f)∧(∀ z : half_plane,
  f z = L_function_product E z)
  ∧ (∀ g : ℂ → ℂ, (differentiable ℂ g)∧
  (∀ z : half_plane, g z = L_function_product E z)
  → g = f) := begin
  sorry,
end

noncomputable def L_function : ℂ → ℂ :=
  classical.some (analytic_continuation E)
```

Here we have a theorem that there exists an $f : \mathbb{C} \to \mathbb{C}$, which is differentiable everywhere and agrees with our product on the `half_plane`. We also require that if g satisfies these properties, then $g = f$. This uniqueness of analytic continuation is a standard fact in complex analysis. We then define `L_function` to be this f.

We finally have to formalise the idea of an order of vanishing of an analytic function. It is a fact in complex analysis that a differentiable function is infinitely differentiable, and further that if it is not identically zero, then at each point there is some derivative which does not vanish. Here we include the function itself as the zeroth order derivative. The smallest order derivative which is non-zero is the order of vanishing. Fortunately mathlib has an iterated derivative function. We can then make definitions as follows:

```
noncomputable def L_derivative (E : elliptic_curve) :
  ℕ → ℂ → ℂ
|n := iterated_deriv n (L_function E)

theorem has_order_of_vanishing : ∃ n : ℕ,
  L_derivative E n 1 ≠ 0 :=
```

Here `L_derivative` is a function taking n, and giving a function $\mathbb{C} \to \mathbb{C}$ which is the n^{th} derivative of the L-function. The theorem, which we do not prove, is that there is some n for which this does not vanish at 1.

We are now in a place to define the analytic rank, and state BSD.

```
noncomputable def analytic_rank (E : elliptic_curve) :
  ℕ := nat.find (has_order_of_vanishing E)

theorem BSD : analytic_rank E = rank E := begin
  sorry,
end
```

The definition is the smallest n satisfying the required property, and then the BSD theorem says that the two types of rank are equal. With the current state of mathematics, this `sorry` is unavoidable.

4. Further work

To continue this work, there are a couple of natural directions to go in. One is filling in the `sorry`s in the rational case. These include the proof of associativity, which has been done in other provers, and the Mordell-Weil theorem, which would involve developing the theory of heights. On the analytic side, we would need to show that the L-function converges in some half-plane. This could be done for $\mathrm{Re}(s) > 3/2$ with the Hasse bound, but perhaps simpler would be to give a weaker bound like $|a_p| \leq p$, and get a smaller half-plane. It would then be very difficult to show that the L-function has an analytic continuation to $\mathbb{C}$ - this is a consequence of the modularity theorem. Showing the existence of an order of vanishing would probably be easier, and likely of more use in complex analysis generally. The final `sorry` is BSD itself.

Another thing to do would be to generalise to number fields. The main extra thing to do here is defining the reduction of the curve modulo prime ideals of

the number field. We would need to define the ring of integers and its primes, understand when a prime ideal divides the discriminant, and be able to use the size of the residue field in order to define the local factors. Showing the L-function has an analytic continuation is an open problem in this case. On the algebraic side, if we wanted to prove Mordell-Weil we would want to develop Galois cohomology, rather than heights as in the rational case. This would of course have applications beyond elliptic curves. A further generalisation could be to abelian varieties.

References

[1] J. H. Silverman, The Arithmetic of Elliptic Curves 2nd Edition, GTM 106, Springer 2009.
[2] Friedl, S.: An elementary proof of the group law for elliptic curves. Groups Complex. Cryptol. 9(2), 117–123 (2017)
[3] Théry, L., Hanrot, G.: Primality proving with elliptic curves. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 319–333. Springer, Heidelberg (2007).
[4] Bartzia, E.-I., Strub, P.-Y.: A formal library for elliptic curves in the Coq proof assistant. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 77–92. Springer, Cham (2014).
[5] Hales T., Raya R. (2020) Formal Proof of the Group Law for Edwards Elliptic Curves. In: Peltier N., Sofronie-Stokkermans V. (eds) Automated Reasoning. IJCAR 2020. Lecture Notes in Computer Science, vol 12167. Springer, Cham. https://doi.org/10.1007/978-3-030-51054-1_15
[6] Jeremy Avigad, Leonardo de Moura, and Soonho Kong, Theorem Proving in Lean, Release 3.23.0, 2021
[7] Andrew Wiles, The Birch and Swinnerton Dyer Conjecture, https://www.claymath.org/sites/default/files/birchswin.pdf
[8] https://github.com/leanprover-community/mathlib
[9] Johannes Hölzl, https://github.com/leanprover-community/mathlib/commit/4d69b0f4550eb6d86e6c61621edb3f67b426fa16