Math ∩ Programming

Elliptic Curves as Python Objects

Posted on February 24, 2014 by j2kun

Last time (https://jeremykun.com/2014/02/16/elliptic-curves-as-algebraic-structures/) we saw a geometric version of the algorithm to add points on elliptic curves. We went quite deep into the formal setting for it (projective space $\mathbb{P}^2$), and we spent a lot of time talking about the right way to define the "zero" object in our elliptic curve so that our issues with vertical lines would disappear.

With that understanding in mind we now finally turn to code, and write classes for curves and points and implement the addition algorithm. As usual, all of the code (https://github.com/j2kun/elliptic-curves-rationals) we wrote in this post is available on this blog's Github page (https://github.com/j2kun).

Points and Curves

Every introductory programming student has probably written the following program in some language for a class representing a point.

```
1 class Point(object):
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
```

It's the simplest possible nontrivial class: an x and y value initialized by a constructor (and in Python all member variables are public).

We want this class to represent a point on an elliptic curve, and overload the addition and negation operators so that we can do stuff like this:

```
1 p1 = Point(3,7)
2 p2 = Point(4,4)
3 p3 = p1 + p2
```

But as we've spent quite a while discussing, the addition operators depend on the features of the elliptic curve they're on (we have to draw lines and intersect it with the curve). There are a few ways we could make this happen, but in order to make the code that uses these classes as simple as possible, we'll have each point contain a reference to the curve they come from. So we need a curve class.

It's pretty simple, actually, since the class is just a placeholder for the coefficients of the defining equation. We assume the equation is already in the Weierstrass normal form, but if it weren't one could perform a whole bunch of algebra to get it in that form (and you can see how convoluted the process is in this short report (https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=8&ved=0CHAQFjAH&url=http%3A%2F%2Ftrac.sagemath.org%attachment%2Fticket%2F3416%2Fcubic_to_weierstrass_documentation.pdf&ei=ruz_Uov-DOjhyQGCo4BQ&usg=AFQjCNHyMPTkzhy9KhgNr-MB0pI6pJXNTw&sig2=xT_VE49cQW7GvGnXlbn7Zw&bvm=bv.61535280,d.aWc&cad=rja) or page 115 (pdf p. 21) of this book (https://pendientedemigracion.ucm.es/BUCM/mat/doc8354.pdf)). To be safe, we'll add a few extra checks to make sure the curve is smooth.

```
class EllipticCurve(object):
   def __init__(self, a, b):
      # assume we're already in the Weierstrass form
      self.a = a
      self.b = b

      self.discriminant = -16 * (4 * a*a*a + 27 * b * b)
      if not self.isSmooth():
         raise Exception("The curve %s is not smooth!" % self)

   def isSmooth(self):
      return self.discriminant != 0

   def testPoint(self, x, y):
      return y*y == x*x*x + self.a * x + self.b

   def __str__(self):
      return 'y^2 = x^3 + %Gx + %G' % (self.a, self.b)

   def __eq__(self, other):
      return (self.a, self.b) == (other.a, other.b)
```

And here's some examples of creating curves

```
>>> EllipticCurve(a=17, b=1)
y^2 = x^3 + 17x + 1
>>> EllipticCurve(a=0, b=0)
Traceback (most recent call last):
  [...]
Exception: The curve y^2 = x^3 + 0x + 0 is not smooth!
```

So there we have it. Now when we construct a Point, we add the curve as the extra argument and a safety-check to make sure the point being constructed is on the given elliptic curve.

```
class Point(object):
   def __init__(self, curve, x, y):
      self.curve = curve # the curve containing this point
      self.x = x
      self.y = y

      if not curve.testPoint(x,y):
         raise Exception("The point %s is not on the given curve %s" % (self, curve))
```

Note that this last check will serve as a coarse unit test for all of our examples. If we mess up then more likely than not the "added" point won't be on the curve at all. More precise testing is required to be bullet-proof, of course, but we leave explicit tests to the reader as an excuse to get their hands wet with equations.

Some examples:

```
>>> c = EllipticCurve(a=1,b=2)
>>> Point(c, 1, 2)
(1, 2)
>>> Point(c, 1, 1)
Traceback (most recent call last):
  [...]
Exception: The point (1, 1) is not on the given curve y^2 = x^3 + 1x + 2
```

Before we go ahead and implement addition and the related functions, we need to be decide how we want to represent the ideal point $[0:1:0]$. We have two options. The first is to do *everything* in projective coordinates and define a whole system for doing projective algebra. Considering we only have one point to worry about, this seems like overkill (but could be fun). The second option, and the one we'll choose, is to have a special subclass of Point that represents the ideal point.

```
class Ideal(Point):
   def __init__(self, curve):
      self.curve = curve

   def __str__(self):
      return "Ideal"
```

Note the inheritance is denoted by the parenthetical (Point) in the first line. Each function we define on a Point will require a 1-2 line overriding function in this subclass, so we will only need a small amount of extra bookkeeping. For example, negation is quite easy.

```
class Point(object):
   ...
   def __neg__(self):
      return Point(self.curve, self.x, -self.y)

class Ideal(Point):
   ...
   def __neg__(self):
      return self
```

Note that Python allows one to override the prefix-minus operation by defining __neg__ on a custom object. There are similar functions for addition (__add__), subtraction, and pretty much every built-in python operation. And of course addition is where things get more interesting. For the ideal point it's trivial.

```
class Ideal(Point):
   ...
   def __add__(self, Q):
      return Q
```

Why does this make sense? Because (as we've said last time) the ideal point is the additive identity in the group structure of the curve. So by all of our analysis, $P + 0 = 0 + P = P$, and the code is satisfyingly short.

For distinct points we have to follow the algorithm we used last time. Remember that the trick was to form the line $L(x)$ passing through the two points being added, substitute that line for y in the elliptic curve, and then figure out the coefficient of x^2 in the resulting polynomial. Then, using the two existing points, we could solve for the third root of the polynomial using Vieta's formula (http://en.wikipedia.org/wiki/Vieta's_formulas).

In order to do that, we need to analytically solve for the coefficient of the x^2 term of the equation $L(x)^2 = x^3 + ax + b$. It's tedious, but straightforward. First, write

$$L(x) = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x - x_1) + y_1$$

The first step of expanding $L(x)^2$ gives us

$$L(x)^2 = y_1^2 + 2y_1\left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x - x_1) + \left[\left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x - x_1)\right]^2$$

And we notice that the only term containing an x^2 part is the last one. Expanding that gives us

$$\left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 (x^2 - 2xx_1 + x_1^2)$$

And again we can discard the parts that don't involve x^2. In other words, if we were to rewrite $L(x)^2 = x^3 + ax + b$ as $0 = x^3 - L(x)^2 + ax + b$, we'd expand all the terms and get something that looks like

$$0 = x^3 - \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 x^2 + C_1 x + C_2$$

where C_1, C_2 are some constants that we don't need. Now using Vieta's formula and calling x_3 the third root we seek, we know that

$$x_1 + x_2 + x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2$$

Which means that $x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_2 - x_1$. Once we have x_3, we can get y_3 from the equation of the line $y_3 = L(x_3)$.

Note that this only works if the two points we're trying to add are different! The other two cases were if the points were the same or lying on a vertical line. These gotchas will manifest themselves as conditional branches of our add function.

```
1  class Point(object):
2     ...
3     def __add__(self, Q):
4        if isinstance(Q, Ideal):
5           return self
6
7        x_1, y_1, x_2, y_2 = self.x, self.y, Q.x, Q.y
8
9        if (x_1, y_1) == (x_2, y_2):
10          # use the tangent method
11          ...
12       else:
13          if x_1 == x_2:
14             return Ideal(self.curve) # vertical line
15
16          # Using Vieta's formula for the sum of the roots
17          m = (y_2 - y_1) / (x_2 - x_1)
18          x_3 = m*m - x_2 - x_1
19          y_3 = m*(x_3 - x_1) + y_1
20
21          return Point(self.curve, x_3, -y_3)
```

First, we check if the two points are the same, in which case we use the tangent method (which we do next). Supposing the points are different, if their x values are the same then the line is vertical and the third point is the ideal point. Otherwise, we use the formula we defined above. Note the subtle and crucial minus sign at the end! The point (x_3, y_3) is the third point of intersection, but we still have to do the reflection to get the sum of the two points.

Now for the case when the points P, Q are actually the same. We'll call it $P = (x_1, y_1)$, and we're trying to find $2P = P + P$. As per our algorithm, we compute the tangent line $J(x)$ at P. In order to do this we need just a tiny bit of calculus. To find the slope of the tangent line we implicitly differentiate the equation $y^2 = x^3 + ax + b$ and get

$$\frac{dy}{dx} = \frac{3x^2 + a}{2y}$$

The only time we'd get a vertical line is when the denominator is zero (you can verify this by taking limits if you wish), and so $y = 0$ implies that $P + P = 0$ and we're done. The fact that this can ever happen for a nonzero P should be surprising to any reader unfamiliar with groups! But without delving into a deep conversation (https://jeremykun.com/2012/12/08/groups-a-primer/) about the different kinds of group structures out there, we'll have to settle for such nice surprises.

In the other case $y \neq 0$, we plug in our x, y values into the derivative and read off the slope m as $(3x_1^2 + a)/(2y_1)$. Then using the same point slope formula for a line, we get $J(x) = m(x - x_1) + y_1$, and we can use the same technique (and the same code!) from the first case to finish.

There is only one minor wrinkle we need to smooth out: can we be sure Vieta's formula works? In fact, the real problem is this: how do we know that x_1 is a *double* root of the resulting cubic? Well, this falls out again from that very abstract and powerful theorem of Bezout (http://en.wikipedia.org/wiki/B%C3%A9zout's_theorem). There is a lot of technical algebraic geometry (and a very interesting but complicated notion of *dimension*) hiding behind the curtain here. But for our purposes it says that our tangent line intersects the elliptic curve with multiplicity 2, and this gives us a double root of the corresponding cubic.

And so in the addition function all we need to do is change the slope we're using. This gives us a nice and short implementation

```
def __add__(self, Q):
      if isinstance(Q, Ideal):
         return self

      x_1, y_1, x_2, y_2 = self.x, self.y, Q.x, Q.y

      if (x_1, y_1) == (x_2, y_2):
         if y_1 == 0:
            return Ideal(self.curve)

         # slope of the tangent line
         m = (3 * x_1 * x_1 + self.curve.a) / (2 * y_1)
      else:
         if x_1 == x_2:
            return Ideal(self.curve)

         # slope of the secant line
         m = (y_2 - y_1) / (x_2 - x_1)

      x_3 = m*m - x_2 - x_1
      y_3 = m*(x_3 - x_1) + y_1

      return Point(self.curve, x_3, -y_3)
```

What's interesting is how little the data of the curve comes into the picture. Nothing depends on b, and only one of the two cases depends on a. This is one reason the Weierstrass normal form is so useful, and it may bite us in the butt later in the few cases we don't have it (for special number fields).

Here are some examples.

```
>>> C = EllipticCurve(a=-2,b=4)
>>> P = Point(C, 3, 5)
>>> Q = Point(C, -2, 0)
>>> P+Q
(0.0, -2.0)
>>> Q+P
(0.0, -2.0)
>>> Q+Q
Ideal
>>> P+P
(0.25, 1.875)
>>> P+P+P
Traceback (most recent call last):
  ...
Exception: The point (-1.958677685950413, 0.6348610067618328) is not on the given curve y^2 = x^3 + -2x +

>>> x = -1.958677685950413
>>> y = 0.6348610067618328
>>> y*y - x*x*x + 2*x - 4
-3.9968028886505635e-15
```

And so we crash headfirst into our first floating point arithmetic issue. We'll vanquish this monster more permanently later in this series (in fact, we'll just scrap it entirely and define our *own* number system!), but for now here's a quick fix:

```
>>> import fractions
>>> frac = fractions.Fraction
>>> C = EllipticCurve(a = frac(-2), b = frac(4))
>>> P = Point(C, frac(3), frac(5))
>>> P+P+P
(Fraction(-237, 121), Fraction(845, 1331))
```

Now that we have addition and negation, the rest of the class is just window dressing. For example, we want to be able to use the subtraction symbol, and so we need to implement __sub__

```
def __sub__(self, Q):
   return self + -Q
```

Note that because the Ideal point is a subclass of point, it inherits all of these special functions while it only needs to override __add__ and __neg__. Thank you, polymorphism! The last function we want is a *scaling* function, which efficiently adds a point to itself n times.

```
class Point(object):
   ...
   def __mul__(self, n):
      if not isinstance(n, int):
         raise Exception("Can't scale a point by something which isn't an int!")
      else:
            if n < 0:
                return -self * -n
            if n == 0:
                return Ideal(self.curve)
            else:
                Q = self
                R = self if n & 1 == 1 else Ideal(self.curve)

                i = 2
                while i <= n:
                    Q = Q + Q

                    if n & i == i:
                        R = Q + R

                    i = i << 1
   return R

   def __rmul__(self, n):
      return self * n

class Ideal(Point):
    ...
    def __mul__(self, n):
        if not isinstance(n, int):
            raise Exception("Can't scale a point by something which isn't an int!")
        else:
            return self
```

The scaling function allows us to quickly compute $nP = P + P + \cdots + P$ (n times). Indeed, the fact that we can do this more efficiently than performing n additions is what makes elliptic curve cryptography work. We'll take a deeper look at this in the next post, but for now let's just say what the algorithm is doing.

Given a number written in binary $n = b_k b_{k-1} \dots b_1 b_0$, we can write nP as

$$b_0 P + b_1 2P + b_2 4P + \cdots + b_k 2^k P$$

The advantage of this is that we can compute each of the $P, 2P, 4P, \dots, 2^k P$ iteratively using only k additions by multiplying by 2 (adding something to itself) k times. Since the number of bits in n is $k = \log(n)$, we're getting a huge improvement over n additions.

The algorithm is given above in code, but it's a simple bit-shifting trick. Just have i be some power of two, shifted by one at the end of every loop. Then start with Q_0 being P, and replace $Q_{j+1} = Q_j + Q_j$, and in typical programming fashion we drop the indices and overwrite the variable binding at each step (Q = Q+Q). Finally, we have a variable R to which Q_j is added when the j-th bit of n is a 1 (and ignored when it's 0). The rest is bookkeeping.

Note that __mul__ only allows us to write something like P * n, but the standard notation for scaling is n * P. This is what __rmul__ allows us to do.

We could add many other helper functions, such as ones to allow us to treat points as if they were lists, checking for equality of points, comparison functions to allow one to sort a list of points in lex order, or a function to transform points into more standard types like tuples and lists. We have done a few of these that you can see if you visit the code repository (https://github.com/j2kun/elliptic-curves-rationals), but we'll leave flushing out the class as an exercise to the reader.

Some examples:

```
>>> import fractions
>>> frac = fractions.Fraction
>>> C = EllipticCurve(a = frac(-2), b = frac(4))
>>> P = Point(C, frac(3), frac(5))
>>> Q = Point(C, frac(-2), frac(0))
>>> P-Q
(Fraction(0, 1), Fraction(-2, 1))
>>> P+P+P+P+P
(Fraction(2312883, 1142761), Fraction(-3507297955, 1221611509))
>>> 5*P
(Fraction(2312883, 1142761), Fraction(-3507297955, 1221611509))
>>> Q - 3*P
(Fraction(240, 1), Fraction(3718, 1))
>>> -20*P
(Fraction(872171688955240345797378940145384578112856996417727644408306502486841054959621893457430066791652
```

As one can see, the precision gets very large very quickly. One thing we'll do to avoid such large numbers (but hopefully not sacrifice security) is to work in finite fields, the simplest version of which is to compute modulo some prime.

So now we have a concrete understanding of the algorithm for adding points on elliptic curves, and a working Python program to do this for rational numbers or floating point numbers (if we want to deal with precision issues). Next time we'll continue this train of thought and upgrade our program (with very little work!) to work over other simple number fields. Then we'll delve into the cryptographic issues, and talk about how one might encode messages on a curve and use algebraic operations to encode their messages.

Until then!

This entry was posted in Algorithms, Cryptography, Group Theory, Number Theory and tagged elliptic curves, groups, python. Bookmark the permalink.

11 thoughts on "Elliptic Curves as Python Objects"

1. **Erik Taubeneck**

   February 25, 2014 at 12:30 am • Reply

   Great posts. Excited to see the rest of this series! I submitted a PR on GitHub to the underlying package with a few things I saw as I was reading.

2. **Fredrik Meyer (@FredrikMeyer)**

February 25, 2014 at 3:31 pm • Reply

Very good! Here is a similar tutorial for Macaulay2 (which perhaps is a slightly better software for doing this kind of algebra) http://www.math.uiuc.edu/Macaulay2/doc/Macaulay2-1.6/share/doc/Macaulay2/Macaulay2Doc/html/___Tutorial_co_sp__Divisors.html

- **j2kun**

  February 25, 2014 at 3:52 pm • Reply

  Warning to readers: this tutorial is intended for true algebraic geometers. Much jargon lies therein! 🙂

3. **infinity0**

   February 26, 2014 at 6:53 pm • Reply

   In practical crypto, it's important to do these algorithms in constant time, to avoid side-channel attacks. How might you implement __add__ in constant time, taking into account vertical and tangent lines?

   - **j2kun**

     February 26, 2014 at 7:34 pm • Reply

     You cannot. That is because we're working over the rational numbers in this step on the way to a more useful implementation. (I say that because I don't intend anything on this blog to be really efficient, but it will be secure).

     In the future of this series, I plan to adapt and extend this code to work for finite fields, where every operation will have take the same amount of time (constant time, since all points involved will have the same precision).

4. **Joe Hartman**

   March 1, 2014 at 4:32 pm • Reply

   Great series. Thank you for putting it together. I'm getting different answers for the first addition example(typo?):
   >>> C = EllipticCurve(a=-2,b=4)
   >>> P = Point(C, 3, 5)
   >>> Q = Point(C, -2, 0)
   >>> P+Q
   (0, -2)
   >>> Q+P
   (0, -2)

5. **Sam**

   July 3, 2014 at 11:04 am • Reply

   Hi Jeremy,

   Thanks for the interesting post. Just thought I'd point out that none of the snippets show the __repr__ definition, only the __str__ definition. If you run the examples in the interpreter after importing, all initialisations just print the memory address, rather than the str(self) that you would otherwise want.

   - **Sam**

     July 3, 2014 at 11:32 am • Reply

Also, perhaps worth noting that if someone is using python 2.x rather than python 3, then a / b automatically casts to int and so the gradients aren't what they should be. You can fix it by adding 'from __future__ import division' at the top of the module. This way you get floats rather than ints. Clearly once you move on to use fractions and such this won't matter, but I haven't got that far yet 🙂

- **j2kun**

  July 3, 2014 at 12:28 pm

  I believe someone did a github merge request to fix these issues. But yeah I tend to not care about some of these more minor details. Sorry if it caused you any frustration.

6. **Mark G**

   May 28, 2017 at 11:35 am • Reply

   If you want to get serious about elliptic curves in Python, you probably want to be using SageMath: http://doc.sagemath.org/html/en/reference/curves/sage/schemes/elliptic_curves/constructor.html

   It will find the rational points for you, tell you the group structure of the rational points, and make many much more sophisticated calculations as well. And you can work over more or less arbitrary rings to boot.

   - **j2kun**

     May 28, 2017 at 5:29 pm • Reply

     Yup, sage is cool.

     The point of this series is to provide a clear implementation of the basic ideas behind elliptic curve cryptography. It's pedagogical. Sage's implementation is clearly geared toward math research, and it (and its documentation) are illegible to non experts. Who in their right mind would learn modern (scheme-centric) algebraic geometry just to experiment with cryptography?