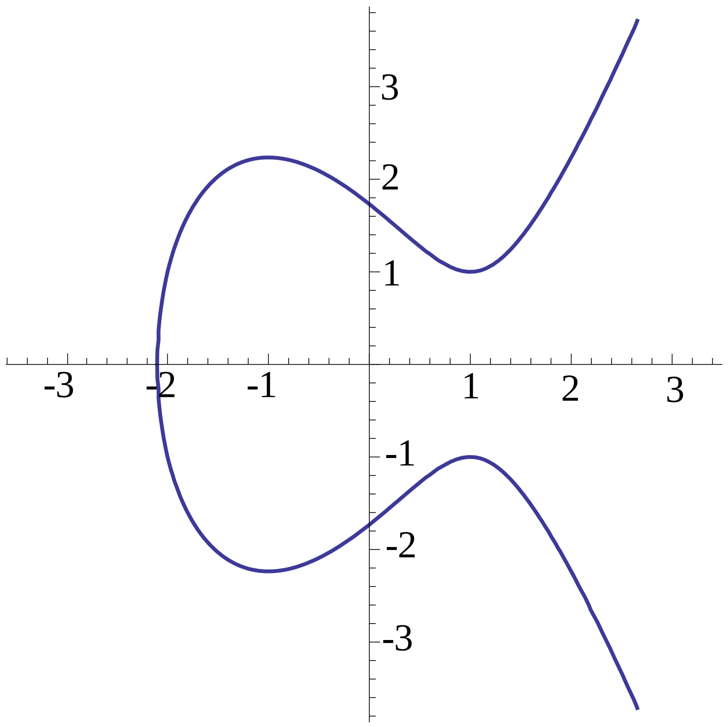
ImperialViolet

Elliptic curves and their implementation (04 Dec 2010)

So what's an elliptic curve? Well, for starters, it's *not* an ellipse. An elliptic curve is a set of points on a plane which satisfy an equation of the form $y^2 = x^3 + ax + b$.

As an example, here's the elliptic curve $y^2 = x^3 - 3x + 3$:



(Thanks to Wolfram Alpha for plotting the curve. Can't see the diagram? Get a better browser).

At this point I'm going to point out that I'm omitting details in order to keep things simple and I'm going to keep doing it without mentioning it again. For example, that elliptic curve equation is based on a transformation which is only valid if the underlying field characteristic is not 2 nor 3. If you want to full details then you should read up. Carrying on …

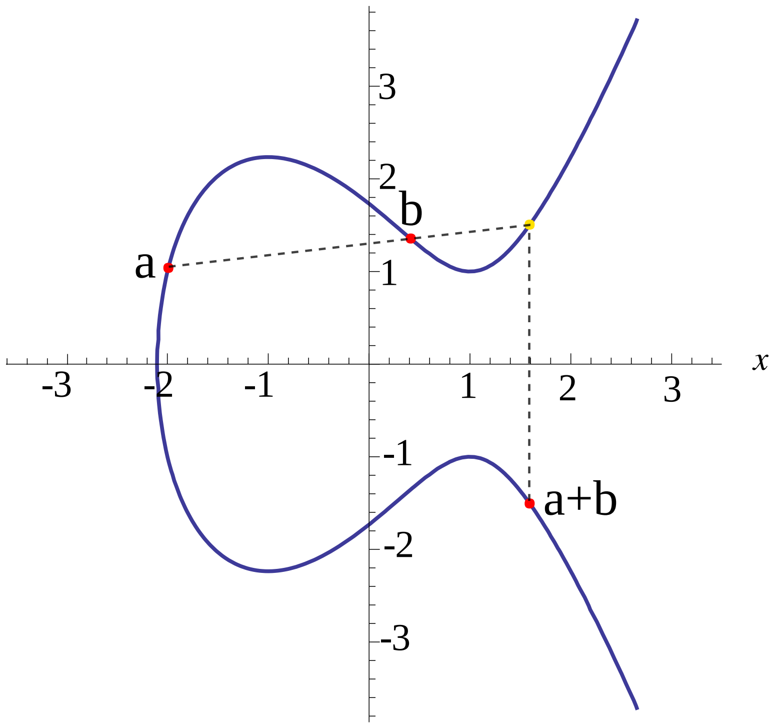
So the elliptic curve is the set of points which satisfy an equation like that. For the curve above you can see that (1, 1) and (1, -1) are both on the curve and a quick mental calculation should confirm that they fit into the equation.

But the points on an elliptic curve aren't just a set of points, with a small hack they have a structure to them which is sufficient to form a *group*. (A group is a mathematical term, not just a noun that I pulled from the air.)

Being a group requires four things: that you can add two elements to get a third element which is also in the group. That this is true for all elements: (a + b) + c = a + (b + c). That there's a zero element so that a + 0 = a. Finally that, for every element, there's a negative of that element (written -a), so that a + -a = 0.

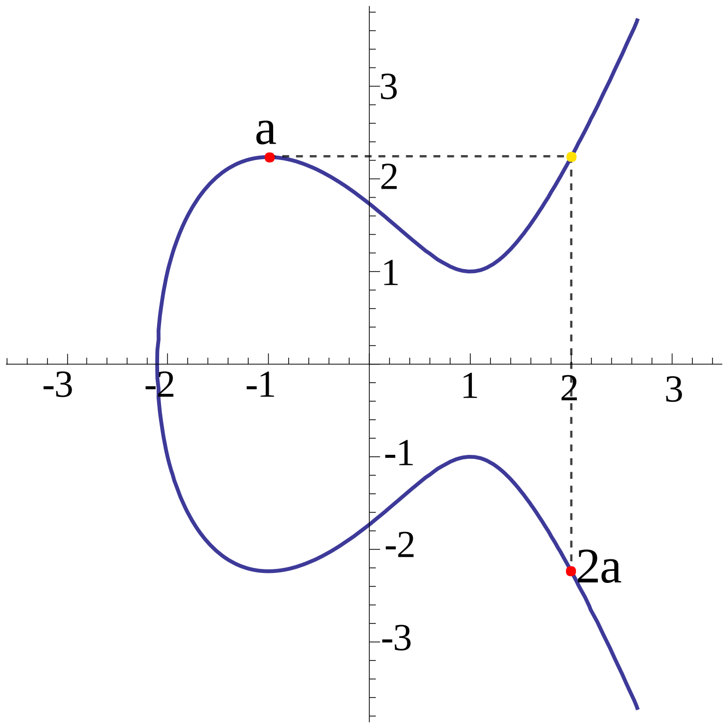
And, when you define addition correctly, the set of points on an elliptic curve has that structure. Also, the addition rule for elliptic curves has a very nice graphical definition, so I can give you a couple more diagrams.

The addition rule says that, to add *a* and *b*, you draw a line from *a* to *b*, intersect it with the curve and reflect in the *x*-axis:



There's a slight wrinkle in this: what if *a* and *b* are the same point? A line needs two, distinct points to be defined.

So there's a special rule for doubling a point: you take the tangent to the curve at that point, intersect it with the curve and then reflect in the *x*-axis:



So we defined addition, but the zero element is a bit of a hack. First, we define the negative of an element to be its reflection in the *x*-axis. We know that adding an element and its negative results in the zero element so think about what adding an element and its reflection means. We draw a line from the element to its reflection, which means that the line is vertical. Now we have to find the third point where that line intersects the curve. But, from looking at the y^2 term of the equation, it's clear that the curve only has two solutions for any given *x* value. So the line doesn't intersect the curve!

And this is where the zero element comes from. It's called the *point at infinity* and it's a magic element which is infinitely far away. This means that the vertical line ends up hitting it. It also means that it's directly above every point so, when you add the zero element to a point, you get its reflection which you then reflect back to the original point.

And, with that brief wart, we have a group! But what's the point? Well, given addition, you can define multiplication (it's just repeated addition). So, given a curve and a point on that curve (called the *base* *point* and written **G**), you can calculate some multiple of that point. But, given the resulting point, you can't feasibly work backwards to find out what the multiple was. It's a one-way function.

To see how this might be useful for cryptography our old friends, Alice and Bob, enter the scene. They are standing in a room with lots of other people and would like a private conservation, but there's not enough room. Thankfully, Alice and Bob both know a standard, public elliptic curve and base point. Each think up a random number and multiply the base point by that number.

We'll call Alice's random number *a* and Bob's, *b*. Each tell the other (and, therefore, everyone) *a***G** and *b***G**: the multiples of the base point. So Alice knows *a* and *b***G** and multiplies the latter by the former to get *ab***G**. Bob knows *b* and *a***G** and multiplies the latter by the former to get *ba***G**. *ab***G** = *ba***G** so both now have a shared, private key. The other people in the room know only *a***G** and *b***G** and they can't divide to get either *a* nor *b*, so they can't calculate the secret.

(That's an elliptic curve Diffie-Hellman key agreement.)

Implementing elliptic curve operations in software.

The diagrams for addition and doubling are pretty, but aren't code. Thankfully they translate pretty easily into equations which can be found at the top of the EFD page for Short Weierstrass curves (which is the specific subset of elliptic curve that we're dealing with).

Those equations are defined in terms of affine coordinates (i.e. (*x*,*y*) pairs) but serious implementations will invariably use a transformation. Transformations are applied at the start of a long calculation so that the steps of the calculation are easier. The transformation can be reversed at the end to get affine coordinates out.

A common example of a transformation is the use of polar coordinates. When working with points on the plane, working with polar coordinates can make some calculations vastly easier. Elliptic curve transformations are like that, although with different transformations.

The transformations that you'll commonly see are Jacobian coordinates and XZ (aka Montgomery's trick), although the latter sort of isn't invertible (more on that later).

The underlying field

When we defined the elliptic curve, above, we talked about a set of points but we never defined a point. Well, a point on the plane can be represented as an (*x*,*y*) pair, but what are *x* and *y*? If this were a maths class they would be elements of $\Re$, the set of real numbers. But real numbers on computers are either very slow or approximate. The rules of the group quickly breakdown if you are only approximate.

So, in cryptographic applications, we use an alternative, finite *field*. (Again, field is a mathematical term, like group.) The finite fields in this case are the numbers modulo a prime. (Although all finite fields of the same size are isomorphic so it doesn't really matter.)

At this point, we're going to speed up. I'm not going to explain finite fields, I'm going to jump into the tricks of implementation. If you made it this far, well done!

It's important to keep in mind that we have two different structures in play here. We have the elliptic curve group, where we are adding and doubling points on the curve, and we have the underlying field. In order to do operations on the group, we have to do a number of operations on elements of the underlying field. For example, here's the formula for group addition under a Jacobian transform. It's given a cost in terms of the underlying field: 11 multiplications plus five squarings.

Our job is to do these group and field operations quickly and in *constant time*. The latter is important! Look at the power of the timing attacks against AES and RSA!

Limb schedules

If the order of the field is between 2^{n-1} and 2^n then we call it an *n*-bit group. (The order can't be 2^n for $n > 0$ because the order must be prime and 2^n is even.)

For cryptographically useful field sizes, elements aren't going to fit in a single register. Therefore we have to represent field elements with a number of *limbs* (which we'll call *a*, *b*, *c*, ...). If we were working in a field of size $2^{127}-1$ then we might represent field elements with two limbs by $2^{64} \times a + b$. Think of it as a two digit number in base 2^{64}.

Since we're using a pair of 64-bit limbs and we're probably putting them in 64-bit registers this is a *uniform*, *saturated* (because the registers are full) limb schedule. As *a* is multiplied by 2^{64} we say that it's “at 2^{64}”.

However, we're going to hit some issues with this scheme. Think about squaring these field elements (multiplication is the same, but I would need more variables). Squaring means calculating $a^2 + 2ab + b^2$. That results in limbs at 2^0 (b^2), 2^{64} (the middle term) and 2^{128} (a^2).

But multiplying the 64-bit limbs gives a 128-bit result (x86-64 chips can do that, although you need some tricks to pull it off from C code). When it comes to the middle term, you can't multiply the 128-bit by 2 without overflowing. You would have to process the result into a form which you could manipulate. Also, if you look at the algorithms which we're implementing, there are small scalar factors and subtractions, both of which are much easier to do with some headroom.

So we could consider an *unsaturated* schedule. For example, for a 224-bit field, four 56-bit limbs works well. It's still uniform, but the results of multiplication are only 112-bit, so you can add several of those numbers together in a 128-bit limb without overflowing.

There are also *non-uniform* schedules. For a 255-bit field, a schedule of 25, 26, 25, 26, ... for ten limbs works well. It's best to keep the schedule as uniform as possible however, otherwise, when multiplying, you end up with values at odd positions and might not have enough headroom to shift them in-place to match up with a limb position.

Prime structures

When implementing curves you typically don't get to choose the prime. There are a few standard primes defined by NIST (in FIPS 186-3) and less common ones (like curve25519).

The structure of the prime matters because the most complex operation will be the reduction. When we did a squaring in the example above, we ended up with a term at 128 bits. We have to eliminate that otherwise, as we keep multiplying results, the results will just get bigger and bigger.

If you think of reducing 2^{127} modulo 2^{127}-1, I hope it's clear that the result is one. If you had a limb at 127 bits, then you could just take that value, add it in at 0 bits and drop the limb at 127 bits. A limb at 127 bits `reflects' off the term at 2^0 and ends getting added in at 0 bits.

Since we had a limb at 128 bits, its reflection off that term ends up at 1 bit. So we would have to left shift the limb by one before adding it in at 0. (And hope that your limb schedule gave you some headroom to do the left shift!). That's pretty much an ideal case and 2^{127}-1 is called a Mersenne prime. Sadly, there aren't any between 2^{127}-1 and 2^{521}-1, so you'll probably get something harder.

Take NIST's P256 curve. Here the prime is $2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$. It was designed so that, with 32-bit limbs, all the reflections end up on 32-bit boundaries. However, we have 64-bit systems these days and we really don't want to waste those big 64-bit full multipliers. However, the prime is nasty for other limb structures. That term at 224 bits is so high up that reflections off it usually end up above 256 and get reflected again. That's terrible if your limbs don't line up with the reflections. For example, if you try 52-bit, unsaturated limbs with that prime, then the reflections for the limb at 416 bits are at 224, 192, 160, 128, 96, 64, 32, and 0 bits! With all that shifting and adding (and more because you don't have enough headroom for all those shifts), you end up with a slow reduction.

I have typically ended up experimenting with different limb schedules and seeing which is the fastest. You can get a rough feeling for how well a given schedule will do but, with CPUs being so complex these days, it's never terribly clear which will be the fastest without trying them.

Inversion

The usual way to do inversion in a finite field is with Euclid's algorithm. However, this isn't constant time. Thankfully, we can use Fermat's little theorem ($x^p \cong x \pmod{p}$) and notice that it easily follows that $x^{p-2} \cong x^{-1} \pmod{p}$. So, just raise the element to power of p - 2. It's a long operation, but you don't do it very often.

(Credit to djb for that trick.)

Subtraction

Although you can use signed limbs, I consider the complexity to be too much to bother with. But, without signed limbs how do you do subtraction?

Well, everything is modulo *p*, so we can always add 0 mod p. If you have unsaturated limbs then the top bits of the limb overlap with the bottom bits of the next limb. So, you can add a large number to the first limb and subtract a small number from the next one. But what stops the next limb from underflowing? Well, you want to add the same large number there and subtract from the next etc.

This continues until you reach the top limb. Here, you can calculate the excess, reduce modulo p and correct the lower limbs to counteract it. In the end, you end up with a few, static values which you can quickly add to a field element without changing the element value, but so that you can now prove that the subtraction doesn't underflow any limbs.

(Credit to Emilia Kasper for that one.)

Remarks

I could write a lot more, but it's getting late. The reason for this write up is that I'm currently working on a fast P256 implementation for OpenSSL. It won't be as fast as I would like due to the issues with the P256 prime, given above. But, I hope to be done by the end of the week and hope to have some exciting news about it in the new year.