Using Python for computing on Elliptic Curves
(very) preliminary draft

Javier Fernandez

June 24, 2003

1 What?

If you want to experiment with elliptic curves you fairly soon realize that doing computations with integer numbers, (or rational numbers, or modulo p) can be time consuming, not to mention a little bit boring. This is why it is very convenient to have a calculator at hand. A problem with calculators is that if you want (need?) to work with integer numbers of more than 10 or 12 digits, most probably, you don't get exact answers. Thus, we look for some kind of "calculator" that would allow us to perform exact computations in all situations as well as, if possible, automatize some repetitive actions. These two goals can be satisfactorily met by using the program called `python` [1].

Let's get started. Go to your computer, login and get a terminal with the shell prompt (oh yes, this is "mildly" Unix oriented... but `python` is available in most other operating systems and using it is (99%) the same). Type the word `python` and hit `Return` (or `Enter`) afterwards.

```
$ python
Python 2.2b1 (#2, Nov  5 2001, 15:20:50)
[GCC 2.95.3 20010315 (release)] on sunos5
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

So, `python` has started and is waiting for your instructions (`>>>` is `python`'s prompt). We will try first some simple arithmetic (remember to press `Return` at the end of each of your commands):

```
>>> 3+2
5
>>> 3*2
6
>>> 3**2
```

[1] Have you ever heard of Monty Python? That is where the name is coming from.

```
9
>>> 1235412515*983475982457895
1214998536930403943555925L
```

Notice how we compute powers using the `**` symbol.

Also, `python` knows how to operate with parentheses:

```
>>> 3*(4-2)
6
```

Divisions?

```
>>> 3/2
1
```

Please notice the previous result: `3/2` evaluates to the "integer division" instead of the usual `1.5` that you would expect. If you want to get `1.5` you have to tell `python` that you want to work with "floating point" numbers:

```
>>> 3.0/2
1.5
```

As you can imagine, there are also variables:

```
>>> a=127
>>> b="hello people"
>>> c=1.27
>>> 2*a+1
255
```

Different values can be assigned to variables. Above, `a` was assigned an integer number, `b` the string `hello people`, and `c` a floating point number. Finally, you can operate with variables as you normally would.

How about arithmetic modulo p? We can ask `python` to reduce a number modulo p using the operator `%` as follows:

```
>>> 5 % 3
2
>>> 5234524452**18 % 19
1L
```

The first line computed the remainder of 5 modulo 3. The second, the remainder of 5234524452^{18} modulo 19, which is 1 (but you don't need `python` for this one...).

Before we move on to something more interesting, there are two things that you have to know: how to stop the program and how to ask for help (other than screaming...). Quitting the program is very simple (on a Unix console) just press the keys `Control` and `D` at the same time, and you are done. To ask for help, start `python` again (see instructions above). Then, on `python`'s prompt type `help()`, and follow the instructions to seek help on a specific topic. Remember to type `quit` when you are done with the help and want to continue using `python`. For more information about `python` you should check `http://www.python.org`.

Now that we know how to do basic arithmetic with `python`, could we check if $(1342, 41543452354)$ is on the elliptic curve $y^2 = x^3+x+1725858433486651246286$?

```
>>> x=1342
>>> y=41543452354
>>> x**3+x+1725858433486651246286
1725858433489068141316L
>>> y**2
1725858433489068141316L
```

As you see, the point is on the curve (because both sides of the equation evaluate to the same thing). Another possibility is to ask `python` to compare both sides automatically as follows:

```
>>> y**2==x**3+x+1725858433486651246286
1
```

Notice how we ask if two things are equal using the symbol `==` (instead of `=`). The answer is the number `1`, meaning that they are equal. If things are different we get `0`:

```
>>> "banana"=="orange"
0
```

while

```
>>> "banana"!="orange"
1
```

here `!=` means "not equal".

2 Functions

We saw in the previous section that `python` can do fairly complicated computations for you so it seems to provide a good calculator for our purposes. How about performing repetitive tasks?

Suppose that you want to check if the point (a, b) is on the curve

$$y^2 = x^3 + x + 1725858433486651246286. \tag{1}$$

You can do this as we did above, but if you want to do this same operation several times it is useful to define a function that does the computation for you:

```
>>> def on_curve(x,y):
...     """See if (x,y) is on the curve y**2==x**3+x+1725858433486651246286"""
...     return y**2==x**3+x+1725858433486651246286
```

We have just defined a function. There are several things to notice here. First line: `def on_curve(x,y):` all function definitions start with `def`, then follows the name of the function (`on_curve`, in this case) and then the parameters of the function, `x` and `y`. Finally, the "`:`". If you forget the colon you will see all sorts of strange complaints from `python`.

The second line is informative, a way of saying what the function does. It is (very) convenient to use it, especially when you write several functions with clever names such as `f`. Notice that the message goes with *triple* quotes.

The third line is where things happen: the return value of the function is the comparison of the equation. So `on_curve(x,y)` will be 0 if (x, y) is not on the curve and 1 otherwise.

We have skipped one essential detail: look at the indentation of the function. It is nice because it makes the function easy to read. Furthermore, if you don't indent your code as we did, `python` will complain.

Now that we have defined our function we can use it:

```
>>> on_curve(1342,41543452354)
1
>>> on_curve(-1342,41543452354)
0
```

and we see that while $(1342, 41543452354)$ is on the curve, $(-1342, 41543452354)$ is not.

Usually, functions are more involved than our `on_curve` example. In almost all cases it is more convenient to write (and perfect) your functions using a text editor and then load them into `python`. Our next section describes how to do that, so that then we can tackle our first real application: computing the sum of points on an elliptic curve.

3 Better interface = Emacs

As we mentioned above it is possible (and desirable) to write your functions using a text editor. We will use our favorite editor: `emacs`.

The first step is tell `emacs` about `python`. Start by editing (with `emacs`, why not?) the `.emacs` file that should be in your home directory. If you don't have this file there, don't worry: just open a new one.

Add the following lines to your `.emacs` file:

```
;;python mode stuff
(setq auto-mode-alist
      (cons '("\\.py$" . python-mode) auto-mode-alist))
(setq interpreter-mode-alist
      (cons '("python" . python-mode)
            interpreter-mode-alist))
(autoload 'python-mode "python-mode" "Python editing mode." t)
```

Don't worry about the meaning of these lines, they tell `emacs` what to do when you want to use `python`.

Save your modified `.emacs` file and restart `emacs`.

Open a new file called `elliptic.py` (you can call it anyway you like, but the termination `.py` is useful to distinguish `python`'s files). You can open a file in `emacs` by using the `File` menu at the upper left corner, choosing `open file` and typing the name of the file at the bottom line on the `emacs` screen (remember to type `Enter`).

Now you got a clean new file for your (ab)use. Let's start by retyping the definition of the function `on_curve`. Two things to notice: `emacs` colors automatically your syntax (if it doesn't, ask someone to tell you how to do this). Also, indentation is kept automatically: you only have to make sure that you un-indent when you want, as we will see later.

Now modify the function to test if the point is on the curve $y^2 = x^3 + 3x$. Your file should look like:

```
def on_curve(x,y):
    """See if (x,y) is on the curve y**2==x**3+3x"""
    return y**2==x**3+3*x
```

Now save it (using the menus: `File` and then `Save`).

You could go back to the `python` that you were using before but, instead, we will use `emacs` to run `python`. On `emacs`' `Python` menu, choose `Start interpreter`. If you did everything we said your `emacs` window will split and on one part you still have your `elliptic.py` file and on the other you have `python`. Now we have to tell `python` to load our function `on_curve`. Actually, what `python` loads is the whole file; to read the file go to `emacsPython` menu and choose `Import/Reload file`.

Now we can check if the point $(0, 0)$ is `on_curve`:

```
>>> on_curve(0,0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'on_curve' is not defined
```

Well, obviously something is not right... The point is that now we should call the function `on_curve` as `elliptic.on_curve` instead (because it is part of the `elliptic` file —or, more properly, *module*). Try again

```
>>> elliptic.on_curve(0,0)
1
```

Exercise 3.1 *Modify the function* `on_curve` *to work with the curve* $y^2 = x^3+8x$ *and test with* `python` *if some points are on this curve or not.* Remember to reload your function into `python` after the modifications are made.

You can now explore the other options available in `emacs`' `Python` menu.

4 First application: sums on an elliptic curve

Our goal now is to produce the first real application: we want to write a function `sum` that will compute the sum of two points on an elliptic curve, using the curve's group structure.

Before we start, we have to decide how we want to "describe" the curve and arbitrary points. We can start by assuming that the curve is given in Weierstrass form

$$y^2 = x^3 + ax^2 + bx + c \tag{2}$$

so that the curve is determined by the tuple (a, b, c). Fortunately, `python` knows what a tuple is:

```
>>> curve = (0,3,0)
>>> curve[0]
0
>>> curve[1]
3
>>> curve[2]
0
```

The first line stores in `curve` the tuple defining the elliptic curve $y^2 = x^3 + 3z$. The next lines show how you can access the components of a tuple. *Notice that the coordinates are numbered beginning from* 0.

Points can be stored in the same way, using tuples with two entries.

```
>>> p1=(1,2)
```

Now the idea is simple: if $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, $p_3 = p_1 + p_2$ satisfies[2] $p_3 = (x_3, y_3)$ with

$$x_3 = \lambda^2 - a - x_1 - x_2 \quad \text{and} \quad y_3 = -(\lambda x_3 + \nu)$$

with

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1} \quad \text{and} \quad \nu = y_1 - \lambda x_1.$$

We write our (first?) version of the `sum` function. Still working on the `elliptic.py` file we write:

```
def sum1(cur,p1,p2):
    """compute p1+p2 on the elliptic curve cur"""
    lam = (p2[1]-p1[1])/(p2[0]-p1[0])
    nu = p1[1] - lam*p1[0]
    x3 = lam**2 - cur[0] - p1[0] - p2[0]
    y3 = -(lam * x3 + nu)
    return (x3,y3)
```

[2]These formulas are from [1, page 31].

Then we reload the `elliptic.py` file (we won't mention this step in the future, so make sure you remember to do it on your own). We pick `p2` on the curve and compute `p1+p2`:

```
>>> p2=(0,0)
>>> elliptic.sum1(curve,p1,(0,0))
(3, -6)
```

Excellent! Now we can play with our `sum1` function. According with [1, page 31], the points $P_1 = (-1, 4)$ and $P_2 = (2, 5)$ are on the curve $y^2 = x^3 + 17$. Let's compute their sum:

```
>>> curve17=(0,0,17)
>>> p1=(-1,4)
>>> p2=(2,5)
>>> elliptic.sum1(curve17,p1,p2)
(-1, -4)
```

So, we obtain $P_1 + P_2 = (-1, -4)$... but this is not the result obtained in [1]. After thinking for a while you may discover that we are wrong! What we missed is that when we compute `lam`, we did it as a quotient of two *integer* numbers, so `python` computed the *integer* division that is not what we wanted. How do we fix this problem? One possibility would be to convert our numbers into floating point numbers so that `python` will compute the appropriate quotient. But the problem with this approach is the precision loss involved in using floating point operations. Instead, we will "teach" `python` about rational numbers. For that we will use the file `Rat.py` that you can obtain from the same place you obtained this document.

We modify the `elliptic.py` file to:

```
from Rat import rat

def on_curve(x,y):
    """See if (x,y) is on the curve y**2==x**3+3x"""
    return y**2==x**3+3*x

def sum1(cur,p1,p2):
    """compute p1+p2 on the elliptic curve cur"""
    lam = (p2[1]-p1[1])/rat(p2[0]-p1[0])
    nu = p1[1] - lam*p1[0]
    x3 = lam**2 - cur[0] - p1[0] - p2[0]
    y3 = -(lam * x3 + nu)
    return (x3,y3)
```

There are two changes: the first line instructs `python` to `import` the function `rat` (that creates a rational number) from the file (module) `Rat.py`. Then, in the definition of `lam` we use `rat` to turn the denominator into a rational number. This change will force `python` to perform a *rational* quotient so that `lam` gets the proper value:

```
>>> elliptic.sum1(curve17,p1,p2)
(Rat(-8,9), Rat(-109,27))
```

That is, $P_1 + P_3 = (-\frac{8}{9}, -\frac{109}{27})$ in accordance with [1].

Next we want to use `sum1` to compute $2P_1 = P_1 + P_1$:

```
 >>> elliptic.sum1(curve17,p1,p1)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "elliptic.py", line 9, in sum1
    lam = (p2[1]-p1[1])/rat(p2[0]-p1[0])
  File "Rat.py", line 151, in __rdiv__
    return Rat(a) / b
  File "Rat.py", line 145, in __div__
    return rat(a.__num * b.__den, a.__den * b.__num)
  File "Rat.py", line 38, in rat
    return Rat(num, den)
  File "Rat.py", line 45, in __init__
    raise ZeroDivisionError, 'rat(x, 0)'
ZeroDivisionError: rat(x, 0)
```

Wow! That was something... wrong. If you look at the last output line you will see the key: we are dividing by 0! Why would that be? When we are trying to compute `lam` we divide by the difference of the x-coordinates of the two points... that is 0 if the points are the same! We have to use the appropriate doubling formula, again from [1]: for $P_1 + P_1$, we have to use

$$\lambda = \frac{3x_1^2 + 2ax_1 + b}{2y_1}.$$

Taking this into account, `sum1` becomes:

```
def sum2(cur,p1,p2):
    """compute p1+p2 on the elliptic curve cur"""
    if p1==p2:
        lam = (3*p1[0]**2 + cur[0]* 2 *p1[0] + cur[1])/rat(2*p1[1])
    else:
        lam = (p2[1]-p1[1])/rat(p2[0]-p1[0])
    nu = p1[1] - lam*p1[0]
    x3 = lam**2 - cur[0] - p1[0] - p2[0]
    y3 = -(lam * x3 + nu)
    return (x3,y3)
```

Here we use the `if ... else` construction: it does the obvious thing, that is, if `p1` equals `p2` it uses the first definition of `lam`, otherwise it uses the second definition. Notice the important role of indentation: it determines what is executed in each case (other languages enclose these "blocks" of code with "{}"; in `python` it is just indentation).

Now we are ready to compute the sum of two points, even if they are equal:

```
>>> elliptic.sum2(curve17,p1,p1)
(Rat(137,64), Rat(-2651,512))
```

Exercise 4.1 *Use* `sum2` *to compute sums and multiples of points on* $y^2 = x^3 + 17$.

Exercise 4.2 *Find* $-P_1$. *Use* `sum2` *to compute* $P_1 + (-P_1)$.

Did you solve Exercise 4.2? Go ahead: do it!

Now that you tried to compute $P_1 + (-P_1)$, can you see what is wrong with `sum2` in this case? The problem is that even if $P_1 \neq -P_1$, they have the same x-component, so in the computation of λ the denominator still vanishes. In order to fix it, we have to understand what we are doing: $P_1 + (-P_1)$ should be the null element (for the sum on the curve), but that point is the point at infinity (for the curve in Weierstrass form), which is not given by a pair of numbers. So we have to represent the null element in some other way. One possibility is as follows:

```
def sum3(cur,p1,p2):
    """compute p1+p2 on the elliptic curve cur"""
    if p1==p2:
        lam = (3*p1[0]**2 + cur[0]* 2 *p1[0] + cur[1])/rat(2*p1[1])
    elif p1[0]==p2[0]:
        return "null_element"
    else:
        lam = (p2[1]-p1[1])/rat(p2[0]-p1[0])
    nu = p1[1] - lam*p1[0]
    x3 = lam**2 - cur[0] - p1[0] - p2[0]
    y3 = -(lam * x3 + nu)
    return (x3,y3)
```

That is, when we detect a $P + (-P)$ situation we simply return the string `"null_element"`.

```
>>> elliptic.sum3(curve17,p1,(-1,-4))
'null_element'
```

Exercise 4.3 *Modify* `sum3` *so that it can accept* `"null_element"` *as input and produces the correct sum. Check that you get the correct values for* `sum4(curve17,"null_element",p1)`, `sum4(curve17,p2,"null_element")` *and* `sum4(curve17,"null_element","null_element")`. *This function, the final, most general, version will be called* `sum`.

Exercise 4.4 *Write the function* `inverse` *that takes a point on an elliptic curve and produces the inverse of that point. Make sure that it also works for the* `"null_element"`.

5 Doing it many times

In the previous section we wrote a function that can handle the sum of any two points on an elliptic curve. Now suppose that you want to compute $17P$, for some point P on the curve. Of course we can use `sum` 16 times, but it would be nicer to have a function that does this repetitive process for us. In this section we will write a multiplication function. The idea is very simple:

```
def multi_1(cur,p,n):
    """Multiply n*p on the elliptic curve cur"""
    r=p
    for k in range(0,n-1):
        r=sum(cur,r,p)
    return r
```

The only new thing to notice is the `for ... in` construction, which is fairly similar to all other "for" constructions in other languages. `range(a,b)` is a function that produces the list[3] `[a, a+1, ..., a+b-1]`, so that in `multi_1` `k` ranges over `[0,1,...,n-2]`. We see how it works:

```
>>> range(1,10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

All together, we can compute $17P_1 = 17(-1,4)$ on $y^2 = x^3 + 17$:

```
 >>> elliptic.multi_1(curve17,p1,17)
(Rat(-391130476455942743315162782082215718768514928473353578695602917918 34
1800926822890671568019663725682516582416818877707989287781995218932991 1379
8780220186533687629519287745512837441,166960666390877819461817450642689484
6005859888440189333310435748549992126652243920916097904973047744495924 3208
59827011567402366387665190293675457613968093011796279764087609593 6321),
Rat(43914240702704114997013930542556606002587503515772554243889126365706 14
4502686969805836122939351785735090903020624356960645452809418777880544 2176
7762493145977613194071088750902095884826462846501539509841118972149981 3126
27344857625848100707804941096885690090187001465796,215735320473450899839 69
3180733113573884649656548881941897354108440094494125860760575368819770 5147
4470366628804863626466531422667250740400899169631049108107305665537611 0023
2613587036224247953249135979588106614491273898954185484335652960455492 7834
070987468769562794948 81))
```

and you see why it is not very healthy to do this kind of computation by hand.

Exercise 5.1 `multi_1` *can only handle multiplication by strictly positive integers. Modify* `multi_1` *to handle multiplication by all sorts of integers.*

[3]lists are very similar to tuples in `python`, except that they have more functionality. For instance, a list can be modified, whereas a tuple can not.

Exercise 5.2 *The approach of* `multi_1` *is a bit simplistic. Suppose that you want to compute the point* $64P$ *using* `multi_1`*: how many sums do we need to compute? Think about a better way of computing this point, using only sums. Write a better multiplication function.*

Another way of having some action iterated, especially convenient when you don't know, a priori, how many times it must be repeated is by using the `while ...` construction. As an example we write a new version of the multiplication function using while:

```
def multi_2(cur,p,n):
    """Multiply n*p on the elliptic curve cur (uses while)"""
    r=p
    while n>1:
        r=sum(cur,r,p)
        n-=1
    return r
```

The `while CONDITION` construct iterates the body of the `while` as long as the condition holds (if the condition is a number, as long as the number is non-zero). CONDITION can be, basically any expression, although usually is a logical or numerical expression. Notice that, because of the indentation, both lines `r=sum(cur,r,p)` and `n-=1` are the body of the while and are iterated as long as `n>1`.

The line `n-=1` is equivalent to `n=n-1` and similar constructions exist for the other arithmetic operations.

There are still other ways of repeating an action. For example, suppose that you want to find the orbit `p=(2,3)` in the group of $y^2 = x^3 + 1$ (that is, the set of points on the curve that are obtained by multiple sums of `p`). We can do it using a `for ... in` construction:

```
>>> curve1=(0,0,1)
>>> p=(2,3)
>>> for k in range (1,10):
...     elliptic.multi_1(curve1,p,k)
...
(2, 3)
(Rat(0,1), Rat(1,1))
(Rat(-1,1), Rat(0,1))
(Rat(0,1), Rat(-1,1))
(Rat(2,1), Rat(-3,1))
'null_element'
(2, 3)
(Rat(0,1), Rat(1,1))
(Rat(-1,1), Rat(0,1))
```

Alternatively, we can use the `map(...)` construction

```
>>> map(lambda k: elliptic.multi_1(curve1,p,k),range(1,10))
[(2, 3), (Rat(0,1), Rat(1,1)), (Rat(-1,1), Rat(0,1)), (Rat(0,1),
Rat(-1,1)), (Rat(2,1), Rat(-3,1)), 'null_element', (2, 3), (Rat(0,1),
Rat(1,1)), (Rat(-1,1), Rat(0,1))]
```

The `map` construction has two parts (separated by a "`,`"), the first part defines an action to be performed on each element of the list that is given as the second part of the function. Notice how the action is defined, using the keyword `lambda`: `lambda k:` says that the variable of the action (function) is `k` and the action is defined by `elliptic.multi_1(curve1,p,k)`.

Notice that from the orbit we see that the point $(2,3)$ has order 6 in the group of $y^2 = x^3 + 1$.

Exercise 5.3 *Write a function called* `order` *that computes the order of a point on an elliptic curve.*

Exercise 5.4 *Write a function called* `countp` *that counts the number of points on an elliptic curve over* $\mathbb{F}_p$*, for p prime.*

References

[1] Joseph H. Silverman and John Tate, *Rational points on elliptic curves*, Undergraduate Texts in Mathematics, Springer-Verlag, New York, 1992.