Elliptic Curves

Criptography

Carmen Azorín Martí

June 6th 2024

1 Introduction

This code written in Python is an implementation of elliptic curves.
The idea of the code is to draw the graph of an elliptic curve given its parameters a and b, so that the curve is defined by:

$$y^2 = x^3 + ax + b$$

In addition, two points P and Q belonging to the curve shall be generated and their sum and the opposite of P shall be plotted.

The elliptic curve has domain one or two intervals in the reals and one of them is not upper bounded. If any of the generated points of the curve is too large, you will see an unintuitive graph. So we have created this variable

```
MAX_INTERVAL_WIDTH = 3
```

The unbounded interval will have the size limited by the value of the variable.

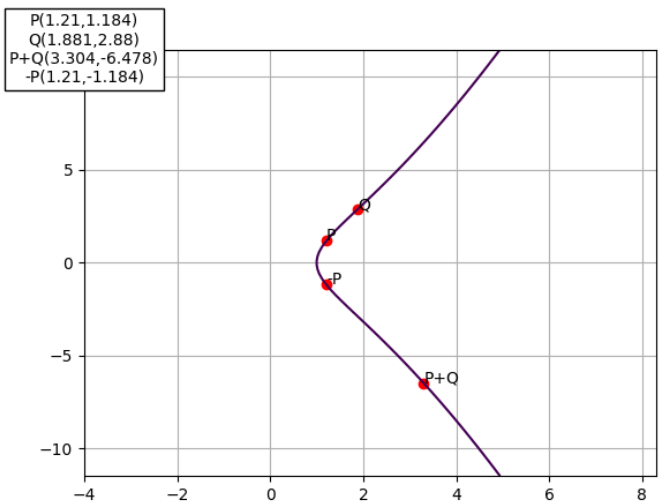
The *numpy* library will be used to perform number operations and generate random numbers. Also in some functions we will need arrays.
The *matplotlib* library is used to generate the curve graph and paint the points.

```
import numpy as np
import matplotlib.pyplot as plt
```

Each point is represented in this Python code by an array of 2 dimensions where position 0 represents x-coordinate and position 1 represents y-coordinate.
In this report we will follow the example of the elliptic curve given by the equation

$$y^2 = x^3 + 3x - 4$$

The resulting graph when executing the code is



2 Implemented functions

The first function implemented is called *realRootsEllipticCurve()* and receives two parameters a and b. And, as its name indicates, it returns the real roots of the curve, which can be either 1 or 3.
As the curve is symmetric above and below $y = 0$, we can take the real roots of the polynomial

$$y = x^3 + ax + b$$

and we will have the roots of the curve.
With the numpy library we can obtain the roots of a polynomial with the *.roots()* function and the coefficients as parameters.The roots array will store only the real roots, which are the ones we are interested in. Finally, we return the sorted array.
In our example, the roots of the polynomial $y = x^3 + 3x - 4$ are $r_1 = 1.0$ and, therefore, this function would return the array [1.0].

```
def realRootsEllipticCurve(a,b):
    coeff=[1,0,a,b]
    roots= [np.real(r) for r in np.roots(coeff) if np.isreal(r)]
    roots.sort()
    return roots
```

The next function is called *getPlotIntervals()* and will return the limits of the displayed curve. It receives the parameters of the curve, a and b, and the randomly generated points that we also want to show. In addition, *extraSpace* stores the margin that we will keep.

In the array *total_points_x* we will store first coordinate of all the points of the curve that we are interested in showing, those are the roots and the array passed as parameter.

We look for the first smallest coordinate, where we will start painting and the first larger coordinate, where we will finish showing the x-axis. Always respecting the margin.

As for the y-axis, we look for the image of the largest points in absolute value. We will start painting the y-axis in the positive part and finish in the negative part.

In our example, the x-coordinate of the root is the smallest. Therefore, the graph will start to be displayed from the value $1 - 5 = -4$ on the horizontal axis. The rightmost point is the $P + Q$, whose first coordinate is $x = 3.304$. So the graph will end at the value $3.304 + 5 = 8.304$ on the horizontal axis. The image of $x = 3.304$ is $y = \pm 6.478$, and these will be the upper and lower limits, adding the margin.

```
def getPlotIntervals(a, b, points, extraSpace=5):
    points_x = [x[0] for x in points]
    roots_x = realRootsEllipticCurve(a,b)
    total_points_x = points_x + roots_x
    MIN_X = np.min(total_points_x) - extraSpace
    MAX_X = np.max(total_points_x) + extraSpace

    images = [np.sqrt(p*p*p +a*p +b) for p in points_x] + [0]
    MIN_Y = -np.max(images)-extraSpace
    MAX_Y = np.max(images) +extraSpace
    return [[MIN_X,MAX_X],[MIN_Y,MAX_Y]]
```

This function is called *generateRandomPoint()* and takes a and b as parameters.

The way we are going to generate the points is: we generate a random number that is in the domain of the curve and look for its image. To do this, we must first find the domain of the curve.

The elliptic curve has 1 or 3 real roots. Let's call r_i each real root with position i in ascending order.

If it has 1 real root, then the domain will be $[r_1, \infty)$. The random number we generate has to be in $[r_1, r_1 + MAX\_INTERVAL\_WIDTH]$. The range of values that the random number can have is the length of the interval.

If it has 3 real roots, then the domain will be $[r_1, r_2] \cup [r_3, \infty)$. The random number we generate will have to be in $[r_1, r_2]$ or $[r_3, MAX\_INTERVAL\_WIDTH]$. The range of values that the random number can have is the sum of the sizes of the intervals and we store them in *total_length*.

This is how we calculate the number:
We generate a random number x between $[0, total\_length]$ and add r_1 to it. Then

$$x \in [r_1, r_1 + total\_length]$$

In case the total number of roots is 3, if $x <= r_2$, then the random number is in the first interval and is valid. Otherwise

$$x \in [r_2, r_2 + MAX\_INTERVAL\_WIDTH - r_3]$$

To this interval we can add the difference between r_3 and r_2 and it would be:

$$x \in [r_2 + r_3 - r_2, r_2 + r_3 - r_2 + MAX\_INTERVAL\_WIDTH - r_3] = [r_3, MAX\_INTERVAL\_WIDTH]$$

as we wanted.

As the roots are rounded, numbers may be generated very close to the domain but outside it. Therefore, we make an extra check to make sure that the generated point is on the curve.

Once we generate the x-coordinate, we calculate its image, which can be positive or negative (with the same probability) and return the point.

In our example, we generate two random points P and Q. When we call this function the array $roots = [1.0]$ and therefore $total\_length = MAX\_INTERVAL\_WIDTH - 1 = 2$.
In the loop a number x between 0 and 2 is generated. In the case of point P, it would be $x = 0.21$. We add $r_1 = 1$ and we would have $x = 1.21$, which is effectively in the domain.
When we leave the loop, we calculate its image

$$y = \pm\sqrt{1.21 \cdot 1.21 \cdot 1.21 + 3 \cdot 1.21 - 4} = \pm\sqrt{1.401} = \pm 1.184$$

and leave it positive with a probability of 0.5. Therefore, the point would remain $P(1.21, 1.184)$.

```
def generateRandomPoint(a,b):
    roots = realRootsEllipticCurve(a,b)
    if len(roots) == 1:
        total_length = MAX_INTERVAL_WIDTH-roots[0]
    if len(roots) == 3:
        total_length = roots[1]-roots[0]+MAX_INTERVAL_WIDTH

    while (True):
        x = np.random.random()*total_length
        x += roots[0]
        if len(roots) == 3 and x > roots[1]: x = roots[2]-roots[1]
        if x*x*x+a*x+b < 0:
            continue
        else:
            break

    if np.random.random() > 0.5: y = -np.sqrt(x*x*x +a*x +b)
    else: y = np.sqrt(x*x*x +a*x +b)
    return [x,y]
```

The following function sums two points $P(x_1, y_1)$ and $Q(x_2, y_2)$ passed as parameters and is called *sumPoints()*. We know how to add two elements of the group generated by the elliptic curve. First, we must calculate the line that passes through both points:

$$y = mx + c$$

. Where m is the slope of the line and is given by:

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

and the y-intercept c is given by:

$$c = y_1 - m \cdot x_1$$

The sum of the points will be the opposite of the intersection point between the line and the curve. That is, the opposite of the point whose first coordinate is $x = m^2 - x_1 - x_2$ and the second coordinate is $y = m \cdot x + c$.

In our example the points $P(1.21, 1.184)$ and $Q(1.881, 2.88)$ are added together. The line through both points is:

$$y = 2.5276 \cdot x - 1.8744$$

given by

$$m = \frac{2.88 - 1.184}{1.881 - 1.21} = \frac{1.696}{0.671} = 2.5376$$

and

$$c = 1.184 - 2.5376 \cdot 1.21 = -1.8744$$

The intersection point of the line and the curve is

$$I(2.5376^2 - 1.21 - 1.184, 2.5376 \cdot (2.5376^2 - 1.21 - 1.184) - 1.8744) = I(3.304, 6.478)$$

Finally, the sum of P and Q is the opposite of the intersection point

$$P + Q(3.304, -6.478)$$

```
def sumPoints(P,Q):
    m = (Q[1]-P[1])/(Q[0]-P[0])
    c = P[1]-m*P[0]
    x = m*m - P[0]-Q[0]
    y = m*x + c
    return [x,-y]
```

The following function is responsible for plotting a point and adding a label to it. For this, it uses the *.plot()* and *.text()* functions from the *matplotlib* library.

```
def representPoint(P,text):
    plt.plot(P[0],P[1],"ro")
    plt.text(P[0],P[1],text)
```

This method is responsible for plotting the elliptic curve using the parameters and the randomly generated points. For this, we need to know the limits of the horizontal and vertical axes, which we obtain by calling the *.getPlotIntervals()* function.
We add the box where the coordinates of all the points are written in the upper left part of the graph.
Now we generate a two-dimensional grid with these intervals. Finally, using the *.contour()* function, we plot the curve.

```
def plotEllipticCurve(a, b, arrayPoints,box_text):
    x = getPlotIntervals(a,b,arrayPoints)[0]
    y = getPlotIntervals(a,b,arrayPoints)[1]
    plt.text(x[0],y[1], box_text, ha='center', va='center', bbox=dict(boxstyle = "square",
                                                facecolor = "white"))
    y, x = np.ogrid[y[0]:y[1]:100j, x[0]:x[1]:100j]
    plt.contour(x.ravel(), y.ravel(), pow(y, 2) - pow(x, 3) - x * a - b, [0])
```

3 Implementation of main function

The following code initializes variables a and b to zero and uses a while loop to ensure that the discriminant $4a^3 + 27b^2$ is non-zero, which is a requirement for an elliptic curve to be non-singular.

```
a, b = 0, 0
while 4*a*a*a + 27*b*b == 0:
    a = float(input("Enter the value of a: "))
    b = float(input("Enter the value of b: "))
```

Two points P and Q are generated randomly on the elliptic curve using the parameters a and b. These points are then represented with labels 'P' and 'Q'. For each point that is generated, we want its coordinates to appear in the top left box of the graph. To do this, we add the text in the variables *box_text*

```
box_text = ""
P = generateRandomPoint(a, b)
representPoint(P,'P')
box_text += "P({},{})\n".format(np.round(P[0],3),np.round(P[1],3))

Q = generateRandomPoint(a, b)
```

A second while loop ensures that Q is not equal to P and that the x-coordinates of P and Q are not equal to avoid division by zero in the slope calculation.

```
while Q[0]-Q[1]== 0 or P[0]==Q[0] and P[1]==Q[1]:
    Q = generateRandomPoint(a, b)
representPoint(Q,'Q')
box_text += "Q({},{})\n".format(np.round(Q[0],3),np.round(Q[1],3))
```

The sum of the points P and Q is calculated using the function *sumPoints(P, Q)*, and this sum is represented with the label 'P+Q'. Additionally, the point opposite to P is calculated and represented with the label '-P'.

```
sum = sumPoints(P,Q)
representPoint(sum,'P+Q')
box_text += "P+Q({},{})\n".format(np.round(sum[0],3),np.round(sum[1],3))
oppositeP = [P[0],-P[1]]
representPoint(oppositeP,'-P')
box_text += "-P({},{})".format(np.round(P[0],3),np.round(-P[1],3))
```

Finally, the elliptic curve is plotted with the points P, Q, their sum, and the opposite of P. A grid is displayed and the plot is shown.

```
plotEllipticCurve(a, b, [P, Q, sum, oppositeP])

plt.grid()
plt.show()
```