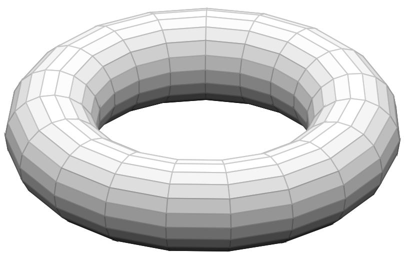
Introduction to Elliptic Curves

from a computational perspective



Siji Adisa
Jason Bohne
Besher Jabri
Amanda Morales Quintana
Christian Moscosa
Miles Shamo

Contents

1. Introduction

The purpose of this presentation is to give an account of work that was done in the Spring of 2020 by a group of five students at the University of Illinois at Chicago who had the goal of getting acquainted with the theory of elliptic curves by exploring their properties through a series of numerical experimentation. The five students are: Siji Adisa, Jason Bohne, Besher Jabri, Amanda Morales Quintana, Christian Moscosa, Miles Shamo, Adisa Siji. The project was sponsored by the Mathematical Computing Laboratory under the direction of Daniel Groves. The specific group of students was supervised by Evangelos Kobotis.

We started from scratch with our language of choice, Python, by considering computations that involve rational numbers. The object of the study was the group of rational points of an elliptic curve defined over $\mathbf{Q}$, so it was necessary to avoid decimals and to conduct all the computations with rational numbers. We knew that we would also need to look at the reductions of an elliptic curve over different finite fields and this is why we also considered finite field arithmetic.

This gave us the opportunity to discuss extensively the fields $\mathbf{F}_p$, where p is a prime number along with their extensions. Even though in our programming we got significant help by pre-existing Python packages we tried to understand the concepts from a completely elementary perspective and we oftened experimented with programs that were written from scratch. We examined topics such as establishing the simple arithmetic of the fields $\mathbf{F}_p$, studying the quadratic reciprocity law and explicitly finding irreducible polynomials over $\mathbf{F}_p$ with the goal of describing some of the finite extensions of these fields.

We then proceeded with the theory of elliptic curves. We worked with the simplified Weierstrass form and we introduced the discriminant as a means of ensuring that a given equation defines an elliptic curve. That enabled us to acquire an abundance of equations of elliptic curves and to start running simple experiments with them. One of the first experiments that we were interested in was to consider the multiples of a given rational point. A computational aspect of interest was to study the complexity of the resulting multiples. Indeed the multiples of points with integer coordinates may have non-integer coordinates and the size of the numerators and denominators that we encounter is a measure of the complexity of the operations that can be carried out on the basis of a given elliptic curve.

At the same time we were interested in the study of the torsion group of an elliptic curve. Mazur's theorem tells us that there are very few possibilities for this group and identifying curves with a given torsion group was one of our early goals. Once we had run some numerical experiments with all the basic aspects of elliptic curves, we decided to take different paths that would lead us to more advanced aspects of the theory. We tried to enlarge the field by considering finite extensions of $\mathbb{Q}$ and we also considered reductions of given elliptic curves over finite fields. In every step of the way the complexity of the resulting operations was one of our main goals and we did extensive testing on the time that it takes to carry out different tasks.

Around the end of our study we were experimenting with the Tate height of points of an elliptic curve and we did touch upon the theory of modular curves by looking at tesselations of the hyperbolic space. All in all, this study gave us the opportunity to consider some quite rich mathematical objects and to study their computational side. The present reports includes a number of the programs that were written and some of the observations that we made. We tried to give the proper credit to all the sources that we used in our work but it is fair to say that most of the programming and experimentation was done from scratch.

We would like to thank, the Math department of the University of Illinois at Chicago that provided significant assistance with the logistics of the project, the Mathematical Computing Laboratory at the University of Illinois at Chicago and its director Daniel Groves for providing the opportunity for this project to exist.

2. Elliptic curves

For our purposes an elliptic curve over $\mathbb{Q}$ is given by an equation of the form:

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

where a and b are rational numbers and it is assumed that the polynomial $x^3 + ax + b$ has not repeated roots. It turns out that a necessary and sufficient condition for the nonexistence of repeated roots is that:

$$4a^3 + 27b^2 \neq 0$$

One however defines the discriminant to be given by:

$$\Delta = -16(4a^3 + 27b^2)$$

and its nonvanishing ensures that the curve, given by the equation on the top of the page, is an elliptic curve. Given any field extension K of $\mathbb{Q}$, it turns out that the set of points whose coordinates satisfy the equation and belong to K form a group. To be more precise, one has to also consider the point at infinity. This is done by projectivizing the equation of the elliptic curve and thus obtain:

$$y^2 z = x^3 + axz^2 + bz^2$$

The solutions of this equation can be considered as points $[x : y : z]$ of the projective space. In fact every classical solution (x, y) of the initial equation (sometimes referred to as the *affine* equation) gives rise to the solution $[x : y : 1]$ of the projective equation equation. We recall that a point of the projective space is given by a triple of numbers with at least one of them being non zero. We identify two triples if one is a multiple of the other. In some sense the projective space can be identified with the set of lines in the three dimensional space. The interesting thing about elliptic curves is that a new point shows up that had not been considered in the affine equation. In particular the point $[0 : 1 : 0]$ is known as the point at infinity and can play the role of the zero element of the abelian group of points of an elliptic curve whose coordinates belong to a given field.

To become a little more descriptive suppose that the elliptic curve given by the equation $y^2 = x^3 + ax + b$ is denoted by E. if K is an extension of $\mathbb{Q}$, then we denote by $E(K)$ the set of points of the elliptic curve whose coordinates belong to K. Then there is a group law which is given by rational functions of x and y that establishes a group law in $E(K)$. The neutral element of the resulting group is the point at infinity. One of the main results in the theory of elliptic curves is the Mordell-Weil theorem, according to which if K is a finite extension of $\mathbb{Q}$, then the group $E(K)$ is finitely generated. This means that it is the direct sum between a free group (necessarily isomorphic to $\mathbb{Z}^r$, for some r) and of a finite group. The integer r is called the rank of the elliptic curve over K. It is conjectured that the rank of an elliptic curve over the field of rational numbers can become arbitrarily large but this conjecture remains unproven.

The program that follows provided the backbone for our study. It allowed us to perform all sorts of numerical experiments with a given elliptic curve. It was written by Miles Shamo and different variations of this program were used in a variety of computations. Different versions and improvements were provided for the duration of the project and it would be useful at some point to make this program available through an appropriate web interface.

The following classes allow us to store and access elliptic curves over the rational numbers, finite fields of degree p, and field extensions. While we reference a different package for each space, we note the structure between the packages is similar. Each package contains a *curve.py* and a *point.py* class which respectively store the elliptic curve and a singular point on the curve over the space. All being said, we will begin with elliptic curves over the rational numbers.

The package *Elliptic*04 consists of a *curve.py* class that stores an elliptic curve along with all the essential properties of them over the rational numbers. It is worth noting in our project we have restricted our analysis to curves written as $y^2 = x^3+ax+b$. The full class can be found below, however, the purpose of *curve.py* (and actually across all of our spaces) is to store the elliptic curve for later use.

```python

# curve.py - Miles Shamo
#
# This is a class designed to store an Elliptic03 curve.
# For the most part, we aren't doing calculations
# with the curves themselves, so this is mainly for use
# in other classes.
#
# These curves are written (at the moment) solely in the form
# y**2 == x**3 + a * x + b
#
# This does limit analysis some but it will be fine for the moment
#
# The most important function of an elliptic curve is
# being able to identify if a point is on it.

import Elliptic04
import fractions


class Curve:

    equationString = ""
    a = fractions.Fraction(0)
    b = fractions.Fraction(0)

    # __init__ is the constructor for python, making
    # sure to store the equation as a string.
    # at the moment, it also takes in two integers, a, b
    # but I would like to parse them from the equation in the future
    def __init__(self, eqS, a, b):
        try:
            self.equationString = eqS
        except (AttributeError, TypeError):
            raise AssertionError("Equations should be entered as strings")

        self.a = a
        self.b = b

    # operator overload to check if already created and untested points are on the curve
    def is_on(self, x, y=0):
        if isinstance(x, Elliptic04.Point): # This line allows for passing in point objects instead of coords
            y = x.yCoord
            x = x.xCoord

        # if y is infinite, it is on the curve
        if y == float('+inf') or y == float('-inf'):
            return True

        return eval(self.equationString)
```

The next class we will examine is *point.py* which stores a singular point on an elliptic curve over the rational numbers. We import our *curve.py* class from before so we can check whether the point does indeed belong to the elliptic curve or not. Other functionalities of the class include checking whether two points are equal, adding two points, and multiplying a point by an integer; all of which arise when performing operations over elliptic curves. In each specific function, however, we impose the additional check of whether the new point is still on the elliptic curve, utilizing our *curve.py* class from above. The full class can be found below:

```

# Point is a point on an eliptic curve
# Thus, it not only stores it's x and y as fractions
# but it stores the elliptic curve it was based on
# to ensure operations keep it on the curve
import fractions
import math # natural log
import Elliptic04 # luckily, this avoids self-reference


class Point:
    xCoord = fractions.Fraction(0)
    yCoord = fractions.Fraction(0)

    # Invalid curve, needs to be overwritten!
    Curve = Elliptic04.Curve("", 0, 0)

    # Python constructor to generate points
    def __init__(self, c, x, y):
        self.Curve = c

        # Ensures the point is on the curve
        assert self.Curve.is_on(x, y), "({}, {}) failed".format(x, y)

        # assigns
        self.xCoord = x
        self.yCoord = y

    # checking if two points are equal
    def __eq__(self, other):
        assert isinstance(other, Point)
        return self.xCoord == other.xCoord and self.yCoord == other.yCoord

    # printing a point
    def __str__(self):
        # if infinite, just infty
        if (self.yCoord == float('+inf')):
            return "Infinity\tOn the EC: " \
                   + self.Curve.equationString

        return "(" + str(self.xCoord) + ", " + str(self.yCoord) + ")\tOn the EC: " \
               + self.Curve.equationString

    # second program to print point without curve
    def printPointPlain(self):
        return str(self.xCoord) + ", " + str(self.yCoord)

    # Adding two points
    def __add__(self, other):
        if isinstance(other, Point):
            # tests to ensure both are on the same curve
            assert self.Curve.is_on(other) and other.Curve.is_on(self)

            # Now we can add (any return statement but the last is to catch an infinity)

            if self == other:
                #We have to make sure that we aren't supposed to have a vertical tangent.  If so, we get
```

```
    the same point out
                if self.yCoord == 0:
                    return Point(self.Curve, self.xCoord, float('+inf'))

                # if self.yCoord is 0, return 0 point
                if self.yCoord == float('+inf'):
                    return self

                # we use a special formulation of lambda when the points are equal to get a tangent instead of line
                L = fractions.Fraction((3 * (self.xCoord ** 2) + self.Curve.a)) / fractions.Fraction((2 * self.yCoord))

            else:
                # if either point is infinite return the reflection of the non-infinite point
                if self.yCoord == float('+inf') or other.yCoord == float('+inf'):
                    if self.yCoord == float('+inf'):
                        return Point(self.Curve, other.xCoord, -1 * other.yCoord)
                    else:
                        return Point(self.Curve, self.xCoord, -1 * self.yCoord)

                # Else we're in the nice case
                else:
                    # If lambda would be zero, return inf
                    if self.xCoord == other.xCoord:
                        return Point(self.Curve, self.xCoord, float('+inf'))

                    # In the normal case, Lamda is just the slope
                    L = fractions.Fraction(other.yCoord - self.yCoord) / fractions.Fraction(other.xCoord - self.xCoord)

            # Calculates the coordinate from lambda
            xNew = fractions.Fraction(L ** 2 - self.xCoord - other.xCoord)
            yNew = fractions.Fraction(-(L * xNew + (self.yCoord - L * self.xCoord))) # Negative to invert!

            return Point(self.Curve, xNew, yNew)

        else:  # if not two points, break
            print("We can only add two points")
            exit(-1)

    # taken from code provided by nullptr on stackoverflow
    # https://stackoverflow.com/questions/30226094/how-do-i-decompose-a-number-into-powers-of-2
    def __int_to_powers_of_two(self, x):
        powers = []
        i = 1
        while i <= x:
            if i & x:
                powers.append(i)
            i <<= 1
        return powers

    # Multiplies a point by an integer (adds it to itself int times)
    def __mul__(self, other):
        # copies self
        copy = self

        # converts int to list of powers of two
        powers = self.__int_to_powers_of_two(int(other))

        # pre-defines result as an impossible point
        result = Point(self.Curve, 0, float('-inf'))

        # loop to go through remaining powers
```

```python
        currPow = 1
        while (len(powers) > 0):

            # add self to self repeatedly
            while (currPow < powers[0]):
                currPow *= 2
                copy += copy

            # now we have a point at the first value, add it to the result
            if (result.yCoord == float('-inf')):
                result = copy
            else:
                result += copy

            # gets rid of this power
            powers.pop(0)


        # old multiplication code, for referrence
        # newP = self
        # for i in range(other - 1):
        #    newP = self + newP

        return result

    def __rmul__(self, other):
        return self * other  # associative

    def getHeight(self):
        return math.log(len(str(max(abs(self.xCoord.numerator), abs(self.xCoord.denominator)))))


```

Moving from rational numbers we created two additional packages $EllipticFinite04$ and $EllipticFiniteExtensions04$ which store elliptic curves and points over finite fields and extensions respectively. As before each package contains two classes *curve.py*, *point.py* however these classes differ based on their domain. For conciseness we will not include the full packages however we will highlight notable differences.

In $EllipticFinite04$ we modulate each point on our curve $\mod p$ to then compute operations. A majority of the script is identical to $Elliptic04$ however we include the excerpt of code responsible for modulating each point below.

```python
# Modulates the point to our finite field
            xMod = (L ** 2 - self.xCoord - other.xCoord) # self.Curve.prime
            yMod = (-(L * xMod + (self.yCoord - L * self.xCoord))) # self.Curve.prime

```

Elliptic Curves over Field Extensions

In $EllipticFiniteExtensions04$ we specify the defining polynomial for the domian, and we modulate each point on the elliptic curve accordingly. The excerpt responsible for this variation is below

```python
   # calculates the new point from lambda
        xMod = (L * L - self.xCoord - other.xCoord) # self.Curve.defPoly
        yMod = ((L * xMod + (self.yCoord - L * self.xCoord)) * -1) # self.Curve.defPoly

```

3. The Legendre symbol

Suppose that p and q are two distinct prime numbers. The Legendre symbol $\left(\frac{p}{q}\right)$ is defined to be 1 if p is a square in $\mathbb{F}_q$ and -1 if it is not. It can also be extended to all nonzero elements fo $\mathbb{F}_q$ with the same definition. The quadratic reciprocity law allows us to compute the Legendre symbol quite easily. In the case of elliptic curves over finite fields, the Legendre symbol can be used in order to identify the points of an elliptic curve defined over the given finite field. In fact suppose that we have an elliptic curve:

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

with a and b in $\mathbb{F}_q$. Then for a given $x \in \mathbb{F}_q$, we may ask the question whether the quantity $x^3 + ax + b$ is a square or not. In the former case x will be the first coordinate of a point of the given elliptic curve. In the second case it will not. So having such a tool allows to easily compute the number of points of an elliptic curve over a finite field. Studying elliptic curve over finite fields is not separated from the study of elliptic curves over fields of characteristic zero. There is a number of facts and conjecture that establish an intimate relation between the two. One of the most celebrated problems of contemporary mathematics, the Birch and Swinnerton-Dyer conjecture is intimately linked to such consideration.

The following program was written by Besher Jabri and was based on a program by Martin Thoma that was found online.

```python
import math
def isPrime(a):
  a=math.floor(a)
  return all(a % i for i in range(2, a))

def getFactors(n):
  factors = []
  p = 2
  while True:
      # while we can divide by smaller number, do so
      while n % p == 0 and n > 0:
          factors.append(p)
          n = n / p
      # p is not necessary prime, but n%p == 0 only for prime numbers
      p += 1
      if p > n / p:
          break
  if n > 1:
      factors.append(n)
  return factors

def legendreSymbol(a,p):
  if a >= p or a < 0:
      return legendreSymbol(a % p, p)
  elif (a == 0 or a == 1):
      return a
  elif a == 2:
      if p % 8 == 1 or p % 8 == 7:
          return 1
      else:
          return -1
  elif a == p - 1:
      if p % 4 == 1:
          return 1
      else:
          return -1
```

```
    elif not isPrime(a):
        factors = getFactors(a)
        product = 1
        for pi in factors:
            product *= legendreSymbol(pi, p)
        return product
    else:
        if ((p - 1) / 2) % 2 == 0 or ((a - 1) / 2) % 2 == 0:
            return legendreSymbol(p, a)
        else:
            return (-1) * legendreSymbol(p, a)
print(legendreSymbol(815,5311379928167670986895882065524686))
```

The Legendre Symbol code above was tested for time efficiency when applied to the context of finite fields in the research of the arithmetic of elliptic curves. The way this code was tested was to first find the next prime finite field (starting with $\mathbb{F}_2$). Once it located the next finite field, it tested $\left(\frac{p}{q}\right) \forall\, 0 \leq p \leq q \; p \in \mathbb{Z}$ where q represents that we are testing in $\mathbb{F}_q$. The time required to find the next finite field was extremely small. In testing just the ability to find the next prime number, it took milliseconds in between each prime number, and under 15 minutes to find all the prime numbers from 2 to 300,000. Therefore, finding the next prime number did not hinder the time efficiency of the function, and has negligible impact on the total time. It would be overwhelming to record every time stamp as there were tens of thousands of calculations being performed. Therefore, a time stamp was recorded every ten seconds, and the testing was run until the program exceeded 2 minutes. The format of the data is the total number of finite fields that have had every possible integer tested on them, followed by the current finite field being tested followed by the total runtime of the program. The following shows the recorded time stamps:

Total # of Fields	Current Finite Field	Time
2904	26431	14.39813756942749
3628	33871	23.11121964454651
4166	39619	30.043560028076172
4872	47251	41.31340551376343
5489	53887	51.62925100326538
5969	59051	61.48317742347717
6492	64969	72.60659003257751
6847	68899	80.78331184387207
7207	72901	90.4820454120636
7439	75527	97.08480072021484
7596	77317	100.40817093849182
8000	81799	110.9949049949646
8475	87281	124.30584836006165

In conclusion, the Legendre symbol package above was able to calculate all the quadratic residues in nearly 8,500 unique finite fields in less than two minutes.

4. General arithmetic in finite fields

In the previous section we studied the Legendre symbol. That provides for one of the non-trivial aspects of our study that we need for the study of elliptic curves over finite fields. Another such aspect is finding the reciprocals of non-zero elements. Practically speaking, given a prime number p and an integer b that is not divisible by p, one wants to find an integer b such that ab is congruent to 1 modulo p. In most cases we take a and b to be between 1 and $p-1$.

The following program was written by Jason Bohne. It computes the modular multiplicative inverse $\forall[1 : p-1]$ for all given primes p up to n. We separate the script into two functions $modInverse(a, prime)$ which returns the modular multlipicative inverse for a given $a \mod p$. Our second function $returnInverses(n, prime)$ is responsible for iterating through $\forall[1 : p-1]$ for a given prime and determining the multiplicative inverse pairs. Finally we save all the primes to text files and print the time which it took to calculate all of the inverses. Note the results below

```
import time
start=time.time()

def modInverse(a, prime):
    a = a % prime
    for x in range(1, prime):
        if ((a * x) % prime == 1):
            return x
    return -1
#Determines the Inverses for 1 up to p-1 for each Prime P

def returnInverses(n, prime):
    newlist=[]
    for i in range(1, n + 1):
        q=modInverse(i, prime)
        if i>=q:
            newlist.append(modInverse(i, prime))
            if i>q:
                newlist.append (i)
    newlist=' '.join(str(x) for x in newlist)
    return newlist
# Collects Inverse Pairs and stores in an array, removes double pairs

f=open('Primesupto1000.txt', 'r')
for line in f:
    list = ([elt.strip() for elt in line.split(',')])
list=map(int,list)
#Opens the File, splits commas and converts strings to Integers

for n in list:
    with open(str(n)+'.txt', 'w') as newfile:
        newfile.write(returnInverses(n-1,n))

#Iterate Through each prime in our inputted List and saves Inverses in a file

end=time.time()
total=round(end-start,2)
string=str("It took "+str(total)+ " seconds to calculate the inverses for all elements in the finite fields of "
                                  "all primes up to 1000 and export in the  associated files")
with open (str("Results012") + '.txt', 'w') as newfile:
    newfile.write (string)
#prints stats
```

Example of modular multiplicative inverses for $p = 29$

[1 5] [6 3] [10 8] [11 9] [13 2] [15 12] [17 16] [20 18] [21 4] [22 23] [24 7] [25 19] [26 14] [27 28]

Time for all modular multiplicative inverses up to n

n	Time
500	0.54 sec
1000	4.01 sec
2500	67.27 sec
5000	518.34 sec

5. Hasse's theorem

Hasse's theorem is a valuable result in the study of the arithmetic of elliptic curves. It provides us with useful bounds that are readily necessary when one defines the L - function of an elliptic curve. According to it if E is an elliptic curve over a field with q elements, then

$$||E(\mathbb{F}_q)| - (q+1)| \leq 2\sqrt{q}$$

Note that here q may be a prime number or a prime power. At this point one needs to recall that every finite field is an extension of a field of the form $\mathbb{F}_p$ where p is a prime number. For this reason, its cardinality is necessarily a power of p. Conversely, given a power of p, say $q = p^r$, then there exists a unique, up to isomorphism, field of cardinality q. In fact it coincides with the splitting field of the polynomial $x^q - x = 0$.

The following program was written by Jason Bohne. It applies Hasse's Theorem to a wide class of different Finite Fields. We first remark that we import the rational roots program as a package which allows us to solve elliptical curves on finite fields. Therefore in our function $accuracy(p)$ we compute the number of points on our elliptical curve on a given finite field of order p along with the bound by Hasse's Theorem. We then return each value along with the difference. We compute the values of such for all primes up to n and save to a text file in addition to the computation time.

```
import math
from RationalRoots01 import util
from RationalRoots01.util import *
import time
start=time.time()

#Made rational roots program into a package, imported it

eq=[1,3]
def accuracy(p):
    num=solver(p,eq)
    points=abs(num-p+1)
    bound=2*(math.sqrt(p))
    string=str((str(points),  str(bound),  str(points-bound)))
    return string
# Calculate both values for Hasse's Thrm and return


f=open('Primesupto1000.txt', 'r')
for line in f:
    list = ([elt.strip() for elt in line.split(',')])
list=map(int,list)
#Opens the File, splits commas and converts strings to Integers

for n in list:
    with open(str(n)+'.txt', 'w') as newfile:
        newfile.write(accuracy(n))

#prints the results over a series of different finite fields
end=time.time()
total=round(end-start,2)
string=str("It took "+str(total)+ " seconds to calculate Hasses's Theorem of the given elliptic curve
    over all finite fields"
```

```
                                                  " of primes up to 1000, and export points, bound, and
    difference in the  associated files")
with open (str("Results020") + '.txt', 'w') as newfile:
    newfile.write (string)
print(total)
```

Example for $p = 29$ $(Points:\ '8', Bound:\ '10.770329614269007', Difference:\ '-2.7703296142690075')$

Time for computations up to n

n	Time
500	1.01 sec
1000	6.12 sec
2500	133.35 sec
5000	684.87 sec

6. Tessellations of the hyperbolic space

The hyperbolic space can be defined as the upper half plane in the complex plane equipped with an appropriate Riemannian metric. If we consider its complex structure, then we can consider the transformations that act on it, i.e. the Möbius transformations and examine the spaces that we get as quotients of such actions. To this end we consider fundamental domains that have representatives of all the orbits of the actions. One can get some very interesting pictures when considering the subdivision of the hyperbolic space into the translates of the fundamental domain. The pictures become even more fascinating when we consider a conformally equivalent space, namely the unit disc. In this sections we will look at some *tessellations* of the unit disc modelled as the hyperbolic space.

The following scripts generate various tessellations commonly referred to as hyperbolic geometries. Our main examples are tilings of the Poincare disc which is a much studied two-dimensional hyperbolic geometry. As we aim for our tilings to represent the geometries as accurately as possible there are many packages required in the code. One such package is hyperbolic, which is a Python library we will use to generate the hyperbolic geometry. It is worth mentioning the dependency, drawSvg, which is the library we will use to visualize all of our shapes.

Nonetheless, the function *drawtiles* enforces the layout of the line segments which manifest as circular arcs in the disc. We also include sold color tilings of the Poincare disc which are a result of the functions $TileDecoratorFish$ and $TileLayoutFish$ that color the different subspaces of the space.

Example 1

```python
##Most of this code for outputting different tessellations was taken from https://github.com/cduck/hyperbolic/blob/master/examples/tiles.ipynb
##I have left comments when deemed necessary

import math
import numpy as np
import drawSvg as draw
from drawSvg import Drawing
from hyperbolic import euclid, util
from hyperbolic.poincare.shapes import *
from hyperbolic.poincare import Transform
from hyperbolic.poincare.util import radialEuclidToPoincare, radialPoincareToEuclid, \
                                     poincareToEuclidFactor, triangleSideForAngles
import hyperbolic.tiles as htiles
#imports essential packages such as DrawSvg to draw vector images and Hyperbolic to create the tiling of the Poincare Disk

def drawTiles(drawing, tiles):
    for tile in tiles:
        d.draw(tile, hwidth=0.075, fill='white') #can change width and fill of tiles
    for tile in tiles:
        d.draw(tile, drawVerts=True, hradius=0.25, hwidth=0.075,
                     fill='red', opacity=0.6) # change color of tile borders, radius, width etc.
        #Function draws the tiles over each level up to depth
p1 = 4
p2 = 6
q = 3
theta1, theta2 = math.pi*2/p1, math.pi*2/p2
phiSum = math.pi*2/q
# above values of p1, p2, q will set angles of each triangle in the tessellation by dividing pi/2 over each
# specified value


r1 = triangleSideForAngles(theta1/2, phiSum/2, theta2/2)
r2 = triangleSideForAngles(theta2/2, phiSum/2, theta1/2)
#solves for the last side of each triange given input parameters of such

```

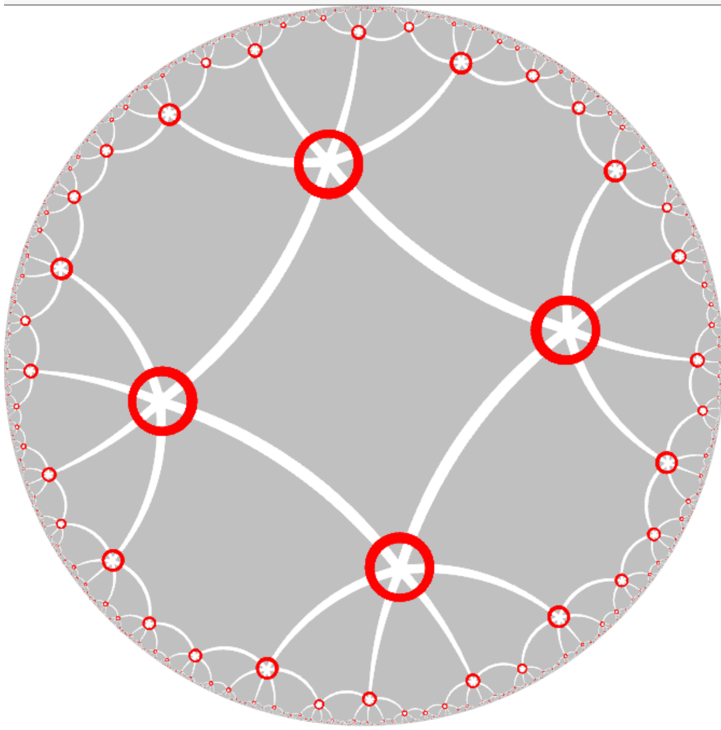


Figure 1: Tessellation Example 1 from above script

```
tGen1 = htiles.TileGen.makeRegular(p1, hr=r1, skip=1)
tGen2 = htiles.TileGen.makeRegular(p2, hr=r2, skip=1)
tLayout = htiles.TileLayout()
tLayout.addGenerator(tGen1, (1,)*p1)
tLayout.addGenerator(tGen2, (0,)*p2)
#Generates the iterations of tiles

startTile = tLayout.defaultStartTile(rotateDeg=10)
#draws tiles with 360/rotateDeg sides

tiles = tLayout.tilePlane(startTile, depth=4)
#depth specifies how many iterations tessellation will repeat

d = Drawing(2, 2, origin='center')
d.draw(euclid.shapes.Circle(0, 0, 1), fill='silver')
drawTiles(d, tiles)
#draws disk where tessllation will be contained within

d.setRenderSize(w=400)
d.saveSvg('Tess01.svg')
#will save in an image file called Tess01 in same project file
```

Example 2

```
##Most of this code for outputting different tessellations was taken from https://github.com/cduck/
    hyperbolic/blob/master/examples/tiles.ipynb
##I have left comments when deemed necessary

import math
import numpy as np
import drawSvg as draw
from drawSvg import Drawing
from hyperbolic import euclid, util
from hyperbolic.poincare.shapes import *
from hyperbolic.poincare import Transform
from hyperbolic.poincare.util import radialEuclidToPoincare, radialPoincareToEuclid, \
                                     poincareToEuclidFactor, triangleSideForAngles
import hyperbolic.tiles as htiles
#imports essential packages as in previous programs

class TileDecoratorFish (htiles.TileDecoratorPolygons):
    #Class that draws the points of each tile refrenced on the github here
    #https://github.com/cduck/hyperbolic/blob/master/hyperbolic/tiles/decorator.py
    def __init__(self, p1=4, p2=3, q=3):
        theta1, theta2 = math.pi * 2 / p1, math.pi * 2 / p2
        phiSum = math.pi * 2 / q
        r1 = triangleSideForAngles (theta1 / 2, phiSum / 2, theta2 / 2)
        r2 = triangleSideForAngles (theta2 / 2, phiSum / 2, theta1 / 2)
        tGen1 = htiles.TileGen.makeRegular (p1, hr=r1)
        tGen2 = htiles.TileGen.makeRegular (p2, hr=r2)

        t1 = tGen1.centeredTile ()
        t2 = tGen2.placedAgainstTile (t1, side=-1)
        t3 = tGen1.placedAgainstTile (t2, side=-1)
        pointBase = t3.vertices[-1]
        points = [Transform.rotation (deg=i * 360 / p1).applyToPoint (pointBase)
                  for i in range (p1)]
        vertices = t1.vertices

        edges = []
        for i, point in enumerate (points):
            v1 = vertices[i]
            v2 = vertices[(i + 1) % p1]
            edge = Hypercycle.fromPoints (*v1, *v2, *point, segment=True, excludeMid=True)
            edges.append (edge)
        edgePoly = Polygon (edges=edges, vertices=vertices)

        origin = Point (0, 0)
        corner = Point.fromHPolar (r1, theta=0)
        corner2 = Transform.rotation (rad=theta1).applyToPoint (corner)
        center = Point.fromHPolar (r2, theta=math.pi - phiSum / 2)
        center = Transform.translation (corner).applyToPoint (center)
        poly = Polygon.fromVertices ((origin, corner, center, corner2))
        desc = poly.makeRestorePoints ()
        descs = [Transform.rotation (deg=i * 360 / p1).applyToList (desc)
                 for i in range (p1)]

        super ().__init__ (edgePoly, polyDescs=descs)
        self.p1 = p1
        self.p2 = p2
        self.colors = ['#ffbf00', 'red','blue','purple','green','yellow','brown','pink']

    def toDrawables(self, elements, tile=None, layer=0, **kwargs):
        #class which assists in drawing more information on same github link
        if tile is None:
            trans = Transform.identity ()
            codes = range (self.p1)
```

```
        else:
            trans = tile.trans
            codes = [side.code[1] for side in tile.sides]
        polys = [Polygon.fromRestorePoints (trans.applyToList (desc))
                 for desc in self.polyDescs]
        ds = []
        if layer == 0:
            for i, poly in enumerate (polys[1:]):
                color = self.colors[codes[i]]
                d = poly.toDrawables (elements, fill=color, opacity=0.5, **kwargs)
                ds.extend (d)
        if layer == 1:
            dLast = polys[0].toDrawables (elements, hwidth=0.03, fill='white', **kwargs)
            ds.extend (dLast)
        if layer == 2:
            for i, poly in enumerate (polys[1:]):
                d = poly.toDrawables (elements, hwidth=0.01, fill='green', **kwargs)
                ds.extend (d)
        return ds

class TileLayoutFish (htiles.TileLayout):
    #specifies the layout of the tiles on each level of iteration
    #more info is refrenced in github here
    # https://github.com/cduck/hyperbolic/blob/1df6140233654ff672a903ca3b2db0c33998ce92/hyperbolic/tiles/TileLayout.py
    def calcGenIndex(self, code):
        ''' Override in subclass to control which type of tile to place '''
        if code == 0 or code == 1:
            return code
        index, color, cw = code
        return index

    def calcTileTouchSide(self, code, genIndex):
        ''' Override in subclass to control tile orientation '''
        return 0

    def calcSideCodes(self, code, genIndex, touchSide, defaultCodes):
        ''' Override in subclass to control tile side codes '''
        p = len (defaultCodes)
        if code == 0 or code == 1:
            c = (code + 1) % 2
            if p % 2 == 0:
                return ((c, 0, 1), (c, 1, 0)) * (p // 2)
            else:
                return ((c, 0, 1), (c, 1, 2), (c, 2, 0)) * (p // 3)
        index, color, otherColor = code
        c = (index + 1) % 2
        # 0=yellow, 1=green, 2=red, 3=blue
        if index == 1:
            newColors = {
                                (0, 1): (0, 2, 3),
                                (1, 0): (1, 3, 2),
                                (0, 2): (0, 3, 1),
                                (2, 0): (2, 1, 3),
                                (0, 3): (0, 1, 2),
                                (3, 0): (3, 2, 1),
                                (1, 2): (1, 0, 3),
                                (2, 1): (2, 3, 0),
                                (1, 3): (1, 2, 0),
                                (3, 1): (3, 0, 2),
                                (2, 3): (2, 0, 1),
                                (3, 2): (3, 1, 0),
                            }[(color, otherColor)] * 10
        elif index == 0:
            newColors = {
```

```
                            (0, 1): (0, 3, 0, 3),  # 0,1,2
                            (1, 2): (1, 3, 1, 3),
                            (2, 0): (2, 3, 2, 3),
                            (0, 2): (0, 1, 0, 1),  # 0,2,3
                            (2, 3): (2, 1, 2, 1),
                            (3, 0): (3, 1, 3, 1),
                            (0, 3): (0, 2, 0, 2),  # 0,3,1
                            (3, 1): (3, 2, 3, 2),
                            (1, 0): (1, 2, 1, 2),
                            (1, 3): (1, 0, 1, 0),  # 1,3,2
                            (3, 2): (3, 0, 3, 0),
                            (2, 1): (2, 0, 2, 0),
                        }[(color, otherColor)] * 10
        else:
            assert False
        newColors = newColors[:p]
        codes = [(c, newColor, newColors[(i + 1) % len (newColors)])
                 for i, newColor in enumerate (newColors)]
        return codes

p1 = 4
p2 = 6
q = 7
rotate = 30
depth = 7

theta1, theta2 = math.pi * 2 / p1, math.pi * 2 / p2
phiSum = math.pi * 2 / q
r1 = triangleSideForAngles (theta1 / 2, phiSum / 2, theta2 / 2)
r2 = triangleSideForAngles (theta2 / 2, phiSum / 2, theta1 / 2)

tGen1 = htiles.TileGen.makeRegular (p1, hr=r1, skip=1)
tGen2 = htiles.TileGen.makeRegular (p2, hr=r2, skip=1.15)

decorator1 = TileDecoratorFish (p1, p2, q)

tLayout = TileLayoutFish ()
tLayout.addGenerator (tGen1, ((0, 1) * 5)[:p1], decorator1)
tLayout.addGenerator (tGen2, ((0, 1, 2) * 2)[:p2], htiles.TileDecoratorNull ())
startTile = tLayout.startTile (code=0, rotateDeg=rotate)

tiles = tLayout.tilePlane (startTile, depth=depth)

d = Drawing (3, 3, origin='center')
d.draw (euclid.shapes.Circle (0, 0, 1), fill='silver')
for tile in tiles:
    d.draw (tile, layer=0)
for tile in tiles:
    d.draw (tile, layer=1)
for tile in tiles:
    d.draw (tile, layer=2)

d.setRenderSize (w=800)
d.saveSvg ('Tess02.svg')
```

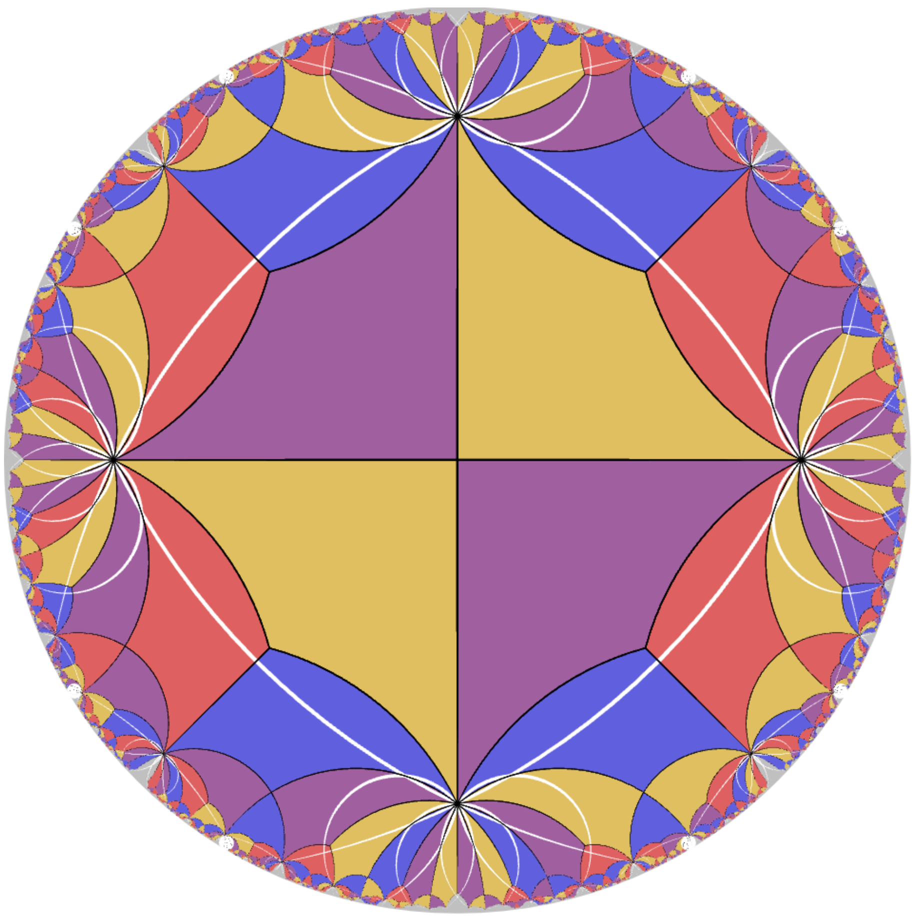


Figure 2: Tessellation Example 2 from above script

Tessellation Software:

7. The Tate height

The Tate height provides us with a quadratic form defined on the set of rational points that can be useful in many different respects. The Tate height can be obtained by a usual projective height through a limiting process that establishes very agreeable algebraic properties.

```python
import Elliptic03,numpy
#imports essential packages

curve = Elliptic03.Curve("y**2 == x**3-2*x+2", -2, 2) #sets elliptical curve
P = Elliptic03.Point(curve, 1, 1) #sets point of infinite order
n=25 #sets how far we want to calculte height of point
error=0.001 #sets error to determine how much the value stabalizes

def stabalizer(arr):
    for x in range(0,n-1):
        if abs(arr[x+1]-arr[x])<error:
            return (x+1)
# stabalizes function that inputs array and loops through to find the first time values are within a set error

array=[]
Q=P
for x in range(1,n):
    Q+=P
    largest=float(max(abs(Q.xCoord.denominator),abs(Q.xCoord.numerator)))
    value = numpy.log(largest)/((x+1)**2)
    array.append(value)
    #Loops through n times and calculates the height of the Logarithmic height of the elliptical curve, appends to an array
    #print(value)

print("It takes",stabalizer(array),"times before the difference in the logarithmic height of the point is within", error)
```

Python code to approximate the Tate height of an elliptic curve. Code written by Jason Bohne.

In order to test the Tate height, I decided to test it in two different ways. Both ways used the exact same elliptic curve $y^2 = x^3 - 2x + 2$ and calculated the Tate height on an infinite order point on that elliptic curve at varying heights. The first test was calculating the Tate height at a height of 1, then 2, then 3, and so on and so forth all in one execution of the program until the threshold of two minutes was reached. These were the results of the first test:

Height	Time
1	0.000086328125e-06
2	0.0004343986511230469
3	0.0007429122924804688
4	0.0011317729949951172
5	0.001260519027709961
6	0.0017027854919433594
7	0.0026140213012695312
8	0.004840850830078125
9	0.062137603759765625
10	0.02026200294494629
11	0.08976578712463379
12	0.14963364601135254
13	0.33268308639526367
14	0.4449632167816162
15	0.9001798629760742
16	1.6023006439208984
17	2.8649306297302246
18	4.6211159229278564
19	6.901501178741455
20	11.26978087425232
21	16.90306782722473
22	26.21683406829834
23	38.575201988220215
24	54.819432973861694
25	83.68168115615845
26	122.09917855262756

Therefore, the first test was able to evaluate the Tate height for an infinite order point on an elliptic curve for all the heights from 1 to 26 in under 2 minutes.
The second test I did was with the exact same elliptic curve, $y^2 = x^3 - 2x + 2$, and using an infinite order point on the elliptic curve, however I wanted to test the maximum height I can calculate the Tate height for in under two minutes. Therefore, I calculated the Tate height for a single height of the elliptic curve in each execution. This was the data of the second test:

Height	Time
25	0.014902353286743164
50	0.16010117530822754
75	0.5199761390686035
100	1.7837965488433838
125	5.163498401641846
150	10.797399520874023
175	20.3620867729187
200	37.30914258956909
225	59.70447611808777
250	102.89141893386841
255	104.91061878204346
260	128.77231621742249

The data of the second test shows that the program can calculate the Tate height for an infinite order point on an elliptic curve with a height ≤ 260 in under two minutes.

8. Elliptic curves over other fields

In this last section we will consider the possibility of studying the group of points of an elliptic curve over fields other than the field of rational numbers. We can do this in two different directions. We can study elliptic curves over a finite field. In this case we begin with an elliptic curve $y^2 = x^3 + ax + b$ with integers a and b and then reduce these modulo p in order to obtain an elliptic curve over $\mathbb{F}_p$. There is a finite amount of prime numbers for which we don't get an elliptic curve and we must be aware of that. The other direction is to begin with an elliptic curve over $\mathbb{Q}$ and then consider its rational points over a finite extension of $\mathbb{Q}$. In both cases we get very interesting questions of both programming and arithmetic. The programs in this section were written by Besher Jabri and Christian Moscosa.

```
from sympy.polys.domains import ZZ
from sympy.polys.galoistools import *

import copy # for deep copies


# a class that stores the defining polynomial of the field and the field it is over.
# this class is intended to be PRIVATE.  I won't import it by default and I won't expect
# the user to see it.  It only holds information for the main class,is extensions
class definingPolynomial:
    # 2 types of constructors
    def __init__(self):
        self.poly = ()
        self.field = -1

    def __init__(self, p, field):
        self.field = field

        assert isinstance(p, list), "ERROR: defining polynomial passed is not a list"

        assert gf_irreducible_p(p, self.field,ZZ), "ERROR: defining polynomial reducable"

        self.poly = tuple(gf_from_int_poly(p, self.field))

    # we want to be able to print this polynomial
    def __str__(self):
        str = ""
        for i in range(0, len(self.poly)):
            if (self.poly[i] != 0):
                if (i > 0):
                    str = str + " + "

                str = str + "{}".format(self.poly[i])
                # print powers of x if not the zeroith element
                if (i > 0):
                    str = str + " * (x**{})".format(i)

        str = str + " == 0"

        return str

    # a defining polynomial equals another if the tuples match and the fields match
    def __eq__(self, other):
        return self.poly == other.poly and self.field == other.field

    # divides another polynomial by this one and returns remainder
    def __rmod__(self, other):
        # asserts other is an instance

        # simply mods the other's polynomial
        newVal = copy.deepcopy(other)
        newVal.poly = gf_rem(other.poly, self.poly, self.field, ZZ)
```

```
        return newVal


# a class storing a standard polynomial over the field
# this is the main class that is used to handle arithmetic over these fields
class extensionVal:

    # either full initialization or field
    def __init__(self, p, d, field = None):
        assert isinstance(p, list), "Error: polynomial passed is not a list"
        # if we are defining from scratch
        if field != None:
            self.defPoly = definingPolynomial(d, field)
            self.poly = p
        # if we are defining off a given defining polynomial
        else:
            assert isinstance(d, definingPolynomial), "Error: cannot instantiate from invalid defining
    polynomial"
            self.defPoly = d
            self.poly = p

        # mods to fit curve
        self.poly = gf_rem(self.poly, self.defPoly.poly, self.defPoly.field, ZZ)

        # if the remainder was zero, we have an empty list, so add it back in
        if len(self.poly) == 0:
            self.poly = [0]

    # a way to print the value as powers of a
    def __str__(self):
        str = ""
        for i in range(0, len(self.poly)):
            # if (self.poly[i] != 0):
                if (i > 0):
                    str = str + " + "

                str = str + "{}".format(self.poly[i])

                if (i > 0):
                    str = str + " * (a**{})".format(i)

        # handles case of zero
        if (len(str) == 0):
            str = "0"

        return str

    # A quick way to tell if two extensions are over the same extended field
    def sameField(self, other):
        return self.defPoly == other.defPoly

    # all operations on the polynomial

    def __eq__(self, other):
        if isinstance(other, extensionVal):
            return self.defPoly == other.defPoly and self.poly == other.poly
        else:
            return False

    def __add__(self, other):
        if isinstance(other, extensionVal):
            assert self.defPoly == other.defPoly, "ERROR (add): defining polynomials do not match"
            return extensionVal(gf_add(self.poly, other.poly, self.defPoly.field, ZZ), self.defPoly)
        else:
            newPoly = list(self.poly)
```

```python
            newPoly[0] += other
            newPoly[0] %= self.defPoly.field
            return extensionVal(newPoly, self.defPoly)

    def __sub__(self, other):
        if isinstance(other, extensionVal):
            assert self.defPoly == other.defPoly, "ERROR (sub): defining polynomials do not match"
            return extensionVal(gf_sub(self.poly, other.poly, self.defPoly.field, ZZ), self.defPoly)
        else:
            newPoly = self.poly.copy()
            newPoly[0] -= other
            newPoly[0] %= self.defPoly.field
            return extensionVal(newPoly, self.defPoly)

    def __mul__(self, other):
        if isinstance(other, extensionVal):
            assert self.defPoly == other.defPoly, "ERROR (mul): defining polynomials do not match"
            return extensionVal(gf_mul(self.poly, other.poly, self.defPoly.field, ZZ), self.defPoly)
        else:
            return extensionVal([(e*other) % self.defPoly.field for e in self.poly], self.defPoly)

    def __truediv__(self, other):
        if isinstance(other, extensionVal):
            assert self.defPoly == other.defPoly, "ERROR (div): defining polynomials do not match"

            #if the other is zero, we don't divid, it just is zero

            # gf_gcdex...[1] is the inverse of other, so we simply create a new value which is this
    inverse
            # and return self * this value
            print(other.defPoly.poly, other.poly, other.defPoly.field, ZZ)
            inverse = extensionVal(gf_gcdex(other.defPoly.poly, other.poly, other.defPoly.field, ZZ)[1],
    other.defPoly)
            return self * inverse
        else:
            # division in a finite field is just multiplication by the inverse
            return extensionVal([(e * (-other)) % self.defPoly.field for e in self.poly], self.defPoly)

    # we need a pow for eval
    def __pow__(self, power, modulo=None):
        newVal = copy.deepcopy(self)

        for i in range(1, power): # from 1 to power-1 (as the first multiple is the deepcopy
            newVal = newVal * self

        return newVal
```

Python code to approximate the Tate height of an elliptic curve. Code written by Jason Bohne.

To test the efficiency of the programs above, it only took two simple tests. For both tests, the same elliptic curve, $y^2 = x^3 + 1$, was used. This way, we could test the efficiency over several different finite fields, and be able to compare without having to account for the equation of the elliptic curve as an independent or confounding variable. Using this equation for the elliptic curve, we start with a point on the elliptic curve. Then, we calculate how many multiples of that point (along with calculating its order) can be calculated prior to reaching the two minute threshold. These were the results:

$y^2 = x^3 + 1$ over $\mathbb{F}_7$ starting at

- Initial point: $(1, 4) \to 1,630,136$ multiples calculated
- Initial point: $(4, 3) \to 1,606,278$ multiples calculated

$y^2 = x^3 + 1$ over $\mathbb{F}_{997}$ starting at

- Initial point: $(2, 3) \rightarrow 1,595,753$ multiples calculated
- Initial point: $(0, 1) \rightarrow 1,587,307$ multiples calculated

Therefore, over a simple finite field, the script was able to calculate over 1.6 million multiples in 2 minutes. And using a much larger finite field, it was still able to calculate roughly the same amount of multiples in the same amount of time.

Elliptic curves offer an array of possibilities in terms of their arithmetic. While we have shown the arithmetic being performed over real numbers and finite fields, it is also interesting to perform arithmetic over strictly rationals, $\mathbb{Q}$, as the realm of rational numbers serves as an abelian group. Below is an extension for a class that will perform arithmetic over the rational numbers, and can be implemented similar to the code given above.

```
# this is a class that defines extension values of Q or finite fields
# based entirely on code provided by Besher

from sympy import *


# a polynomial to define a finite field extension in terms of it's root


class DefPoly:

    def __init__(self, array, Field=0):
        assert isinstance(array, list), "ERROR: must pass a polynomial as a list"

        # stores the field for converting other things
        self.Field = Field

        x = symbols('x')

        # if we havea field of zero, we want a finite field
        if (Field == 0):
            self.polynomial = Poly(array[::-1], x, domain=QQ)
        else:
            self.polynomial = Poly(array[::-1], x, domain=FF(Field))

        assert self.polynomial.is_irreducible


    def __eq__(self, other):
        return self.polynomial == other.polynomial

    def __rmod__(self, other):
        other.polynomial = rem(other.polynomial, self.polynomial, domain=self.polynomial.domain)
        return True # unneded


class ExtensionValue:

    def __init__(self, array, definingPolynomial):


        a = symbols('x') # named A here to distinguish this as the root instead of the equation
        self.defPoly = definingPolynomial

        # if we're constructing a new point
        if (isinstance(array, list)):
            # steal's domain from the defining polynomial
            self.polynomial = Poly(array[::-1], a, domain=self.defPoly.polynomial.domain)
```

```
        else: # this is a polynomial object
            self.polynomial = array

        # mods it (stores automatically)
        self % self.defPoly


    def __str__(self):
        a = symbols('x') # used to substitute a back in for x before printing
        return str(self.polynomial.subs(a, symbols('a')))

    def __eq__(self, other):
        if isinstance(other, ExtensionValue):
            return self.defPoly == other.defPoly and self.polynomial == other.polynomial
        else:
            return False

    # all functions

    def __add__(self, other):
        if isinstance(other, ExtensionValue):
            newPoly = self.polynomial + other.polynomial
        else:
            # mod other if needed
            if (self.defPoly.Field != 0):
                other %= self.defPoly.Field
            newPoly = self.polynomial.add_ground(other)
        return ExtensionValue(newPoly, self.defPoly)

    def __truediv__(self, other):
        if isinstance(other, ExtensionValue):

            # if on a finite field we invert
            if (self.defPoly.Field != 0):
                # tests for zero
                a = symbols('x')
                #gcdex[1] is the inverse of the polynomial
                newPoly = self.polynomial * invert(other.polynomial, other.defPoly.polynomial)
            else:
                newPoly = self.polynomial / other.polynomial
        else:
            # mod other if needed
            if (self.defPoly.Field != 0):
                other %= self.defPoly.Field
            newPoly = self.polynomial.exquo_ground(other)
        return ExtensionValue(newPoly, self.defPoly)

    def __mul__(self, other):
        if isinstance(other, ExtensionValue):
            newPoly = self.polynomial * other.polynomial
        else:
            # mod other if needed
            if (self.defPoly.Field != 0):
                other %= self.defPoly.Field
            newPoly = self.polynomial.mul_ground(other)
        return ExtensionValue(newPoly, self.defPoly)

    def __sub__(self, other):
        if isinstance(other, ExtensionValue):
            newPoly = self.polynomial - other.polynomial
        else:
            # mod other if needed
            if (self.defPoly.Field != 0):
                other %= self.defPoly.Field
            newPoly = self.polynomial.sub_ground(other)
```

```
        return ExtensionValue(newPoly, self.defPoly)

    def __pow__(self, power, modulo=None):
        newPoly = self.polynomial

        for i in range(1, power): # from 1 to power-1 (first multiple is above)
            newPoly *= self.polynomial

        return ExtensionValue(newPoly, self.defPoly)
```

Python extension class for arithemetic on an elliptic curve over the rational field $\mathbb{Q}$, as opposed to a finite field. Code written by Besher Jabri.