Author: Clay McGowen

Date: May 4, 2016

Boise State University

On Bachet's Equation

Senior Thesis

Math 401

Under the supervision of

Dr. Marion Scheepers

Boise State University

Abstract

Bachet's Equation is a famous Diophantine equation of the form $y^2 = x^3 + k$; we have been researching specifically $(y^2 = x^3 + k) \mod p$ with x, y, k in the set of integers less than p. This equation has a group operation and we are focused on the number of solution sets that have a prime size when we vary k, both including and excluding the identity element. Our goal has been to recognize, understand, and classify patterns within the experimental data by making use of the Online Encyclopedia of Integer Sequences.

Primary Subject **14H52**
Secondary Subject **11G05**
AMS 2010 Classifications

Contents

1 Introduction 3
- 1.1 Diophantine Equations . . . 3
- 1.2 Elliptic curves . . . 3
- 1.3 Focus of the Research . . . 4
- 1.4 The Algebraic Platform . . . 4
- 1.5 Structure of the Set of Solutions . . . 4

2 Experimental Data and Conjectures 5

3 Mathematical Analysis of the Conjectures 6
- 3.1 The First Conjecture . . . 6
- 3.2 The Second Conjecture . . . 7
- 3.3 The Infinitude of Primes of the Form $3x^2 + 3x + 1$ . . . 8
- 3.4 The Fourth Conjecture . . . 9
- 3.5 Study of the k's . . . 10

4 Acknowledgements 11

5 References 12

6 Useful Information 13
- 6.1 Definitions . . . 13
- 6.2 Additional Information . . . 13

A Appendix: Sage and IPython Code for Data 14
- A.1 All Data . . . 14
- A.2 p, $\mathcal{S}$, & $|E_k(p)|$ data . . . 24
- A.3 k Data Correlating to the Cuban Primes . . . 25

1 Introduction

1.1 Diophantine Equations

Diophantine Equations are multivariable polynomial functions and typically consider under the constraint that solutions be integers. They are named after the Greek mathematician Diophantus of Alexandria, who first investigated equations with these properties in the 3rd century. The study of these equations marks one of the first introductions of symbolism in algebra. Work with diophantine equations are notoriously difficult, for example, in 1970, Yuri Matiyasevich proved that it was impossible to develop any general algorithm to solve all diophantine equations.[5]

1.2 Elliptic curves

An elliptic curve is a plane algebraic curve[6.1] defined by an equation of the form:

$$y^2 = x^3 + ax + k \tag{1.2.1}$$

These curves take on distinctive shapes based on the values for a and k. As illustrated in Figure 1.

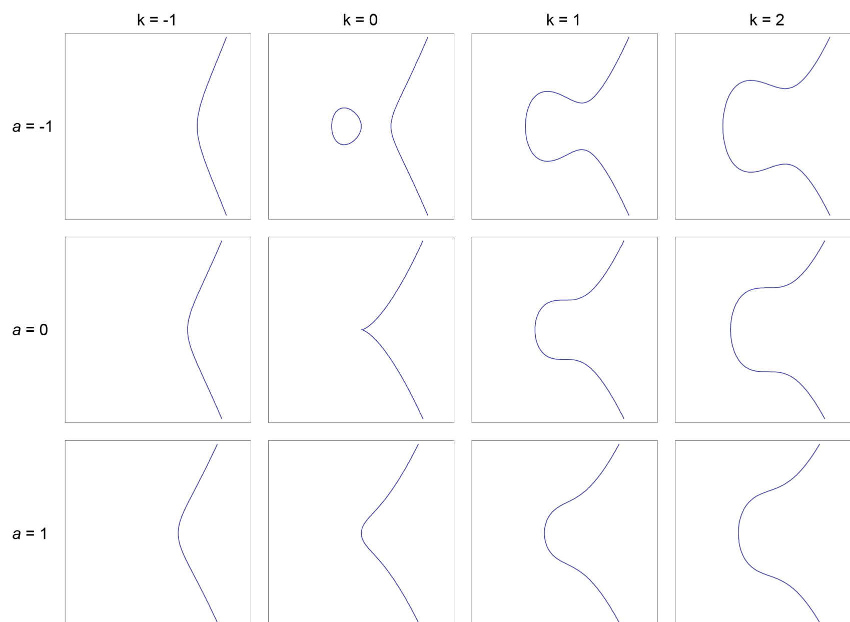

Figure 1: Examples of curves described by Equation (1.2.1) [8]

1.3 Focus of the Research

The primary focus of this paper is on Bachet's equation and the solutions thereto. Bachet's equation is the instance of the diophantine equation (1.2.1) where $a = 0$. Thus Bachet's equation is:

$$y^2 = x^3 + k \tag{1.3.1}$$

It is named after famous French mathematician *Claude Gaspard Bachet de Méziriac*, who examined this as a diophantine equation. Mordell investigated this diophantine equation and made a few important discoveries including:

- There exists more than one solution, as first proposed by Bachet and Fermat.
- There exists a means to calculate general solutions for certain values of k.

C. G. Bachet de Méziriac[7]

1.4 The Algebraic Platform

The focus of this thesis is finding solutions for Bachet's equation over the rings $(\mathbb{Z}_p, + \mod p, * \mod p)$ where p is some prime. Recall that:

$$\mathbb{Z}_p = \{0, 1, \ldots, p-2, p-1\}$$

1.5 Structure of the Set of Solutions

For a prime p and a $k \in \mathbb{Z}_p$, solutions in Bachet's equation are the elements of the set, $\mathcal{S}_k(p)$, of ordered pairs (x, y) where $x, y \in \mathbb{Z}_p$ and $y^2 = x^3 + k \mod p$. The set $S_k(p)$ can be supplemented with an extraneous element, here denoted by ID so that the set $E_k(p) = \mathcal{S}_k(p) \cup \{ID\}$ supports a group operation which will be denoted by $\oplus$. The group $(E_k(p), \oplus)$ is an example of an elliptic curve group. Our interest is in the order of this group.

Helmut Hasse[6] was a German mathematician working in algebraic number theory who proved the following restriction on the orders of these elliptic curve groups:

Theorem 1.1: The Hasse Interval Theorem

For primes p, the sizes of the elliptic curve groups over $\mathbb{Z}_p$ must be in the interval:

$$\Big((p+1) - 2\sqrt{p},\ (p+1) + 2\sqrt{p}\Big)$$

Note that the prime p is an element of this Hasse interval. In this thesis, we investigate the pairs (p, k) for which $|E_k(p)| = \#E_k(p) = p$

2 Experimental Data and Conjectures

Appendix A.1 contains code in sage that computes group orders for elliptic curve groups $E_k(p)$. Applying this code to the first 2043 primes, we found that the following list of primes, p, are the ones for which there exists some k such that $\#E_k(p) = p$:

$$\begin{aligned} p = \{&7, 19, 37, 61, 127, 271, 331, 397, 547, 631, 919, 1657, 1801, 1951,\\ &2269, 2437, 2791, 3169, 3571, 4219, 4447, 5167, 5419, 6211, 7057,\\ &7351, 8269, 9241, 10267, 11719, 12097, 13267, 13669, 16651\} \end{aligned} \tag{2.0.1}$$

To describe our data findings, define $\mathcal{K}_p$ to be $\mathcal{K}_p = \{k \ : \ \#E_k(p) = p\}$

Analysis of the members of $\mathcal{K}_p$ revealed that half of the primitive roots[6.3] of p appear in $\mathcal{K}_p$. This focused attention on two things: which of the primitive roots are in $\mathcal{K}_p$ and what other elements of $\mathbb{Z}_p$ are in $\mathcal{K}_p$

Conjecture 2.1

For each prime, p, for which $\mathcal{K}_p \neq \emptyset$ exactly half of the primitive roots of p are members of $\mathcal{K}_p$.

Conjecture 2.2

For a prime number p, if $\mathcal{K}_p$ is nonempty, then p is a prime of the form $3x^2 + 3x + 1$. (And conversely)

Further experimentation with primes suggest that there may be infinitely many cuban primes

Conjecture 2.3

There exist infinitely many primes p for which $\mathcal{K}_p \neq \emptyset$

Appendix A.3 contains code which computes for each cuban prime, p, the set of k for which $\#E_k(p) = p$.

Examining the data generated with this software, we observed a pattern which we formulated as the following conjecture.

Conjecture 2.4: On the number of k's

For a prime p such that $\mathcal{K}_p \neq \emptyset$,

$$\#\mathcal{K}_p = \frac{p-1}{6}$$

3 Mathematical Analysis of the Conjectures

We will make use of the following theorem, which appears as *Theorem 2.4* in [2]

Theorem 2.4. *Let $p > 3$ be an odd prime and let $k \not\equiv_p 0$. Let N_p denote $\#E(\mathbb{F}_p)$, where E is the elliptic curve $y^2 = x^3 + k$. Here, QR and CR denote quadratic residue and cubic residue, respectively.*

(a) *If $p \equiv_3 2$, then $N_p = p + 1$.*

(b) *If $p \equiv_3 1$, write $p = a^2 + 3b^2$,[1] where a, b are integers with b positive and $a \equiv_3 -1$. Then*

$$N_p = \begin{cases} p+1+2a & \text{if } k \text{ is a sixth power mod } p \\ p+1-2a & \text{if } k \text{ is a CR, but not a QR, mod } p \\ p+1-a\pm 3b & \text{if } k \text{ is a QR, but not a CR, mod } p \\ p+1+a\pm 3b & \text{if } k \text{ is neither a QR nor a CR mod } p. \end{cases}$$

3.1 The First Conjecture

Proposition 3.1: On k

If k is a primitive root of p and $p \mod 3 = 1$, then $N_p = p + 1 + a \pm 3b$.

Proof 3.1

In the following we use facts from 6.1

- If g is a generator of U_p then $\text{order}(g^2) = \frac{p-1}{\gcd(2,p-1)} = \frac{p-1}{2}$.
  Therefore, quadratic residues are not primitive roots.
- Further, if g is a generator of U_p then $\text{order}(g^3) = \frac{p-1}{\gcd(3,p-1)} = \frac{p-1}{3}$.
  Therefore, cubic residues are not primitive roots.
- Finally, if g is a generator of U_p then $\text{order}(g^6) = \frac{p-1}{\gcd(6,p-1)} = \frac{p-1}{6}$.
  Therefore, sixth power mod p's are not primitive roots.

Therefore $N_p = p + 1 + a \pm 3b$.

Now the challenge is to decide if we are in the addition or subtraction case. Through investigation, $p = u^2 + 3v^2$ where $u, v > 0$
If $u \mod 3 = 1$ put $a = -u$ and $b = v$, else put $a = u$ and $b = -v$. This enables us to always use the positive version for N_p leaving us $N_p = (p + 1) - a + 3b$

Observation 3.1: On the last case

Primitive roots are all in the last category of Case (b) of *Theorem 2.4*; however, data suggests that only half of these produce elliptic curve groups of size p.

3.2 The Second Conjecture

Proposition 3.2

If p is a prime such that for some $k \in \mathcal{K}_p$ we have $\#E_k(p) = p$ then p is of the form $3x^2 + 3x + 1$ for some integer x.

Proof 3.2

By Proposition 3.1, $p = N_p = p + 1 + a \pm 3b$. This means, $\mp 3b = a + 1$ or $a = \mp 3b - 1$. Thus

$$\begin{aligned} p &= a^2 + 3b^2 \\ &= (\mp 3b - 1)^2 + 3b^2 \\ &= 9b^2 \pm 6b + 1 + 3b^2 \\ &= 12b^2 \pm 6b + 1 \\ &= 3(2b)^2 \pm 3(2b) + 1 \\ &= 3(-2b)^2 + 3(-2b) + 1 \end{aligned}$$

which is of the form $3x^2 + 3x + 1$ where x is an integer.

Recall the definition of a cuban prime:

Definition 3.1: Cuban Primes

Let x be a positive integer.
Then *cuban primes* are primes that are of the form $(x + 1)^3 - x^3$.

Note that $p(x) = (x + 1)^3 - x^3 = 3x^2 + 3x + 1$ is a second degree polynomial; thus each cuban prime is a prime value of $p(x)$.

Remark 3.1

If in the above argument we can prove that x is positive, then we can conclude that p is a cuban prime. Our data suggests that in fact x should be positive in all cases.

A proof of the converse of Conjecture 2.2 would most likely aid in proving x is positive.

We turned to the Online Encyclopedia of Integer Sequences to see if the particular set of primes, given in Equation 2.0.1, was already recorded in a database. It turned out to be entry A002407, which is the list of primes which are the difference of two consecutive cubes, called the cuban primes.

3.3 The Infinitude of Primes of the Form $3x^2 + 3x + 1$

If Conjecture 2.3 is true, then Proposition 3.2 would imply that there are infinitely many primes of the form $3x^2 + 3x + 1$.

There is no known proof that some quadratic polynomial over the integers has infinitely many prime values; however, there is a classical conjecture that implies for certain polynomials over the integers, these polynomials have infinitely many prime values.

Conjecture 3.1: Bunyuakowski's

If $f(x)$ satisfies the following three conditions, then $f(n)$ is prime for infinitely many positive integers.

1. The leading coefficient of $f(x)$ is positive
2. The polynomial is irreducible over the integers
3. As n runs over the positive integers, the numbers $f(n)$ should be relatively prime (Thus, the coefficients of $f(x)$ should be relatively prime.)

Proposition 3.3

Bunyuakowski's conjecture implies that there are infinitely many primes of the form $3x^2 + 3x + 1$ over the integers.

Proof 3.3

1. The leading coefficient of $p(x)$ is 3 which is positive.
2. The discriminant of the quadratic polynomial is not an integer and thus this polynomial has no integer roots. Hence, $p(x)$ is irreducible over the integers.
3. The $\gcd(3, 3, 1) = 1$, thus the numbers $f(n)$ are relatively prime.

Bunyuakowski's conjecture, though a conjecture, is another source of confidence that would suggest that Conjecture 2.3 is true; however, we still have no proof thereof.

3.4 The Fourth Conjecture

Lemma 3.1

When p is of the form $3x^2 + 3x + 1$, then $(p - 1)/6$ is an integer.

Proof 3.4

$$p = 3x^2 + 3x + 1 \quad = 3(x^2 + x) + 1 \quad = 3x(x + 1) + 1$$

Noting that either x is even or $x + 1$ is even, $3x^2 + 3x + 1 \mod 6 = 1$.
Thus $(p - 1)/6$ is an integer.

At this point, we generated a table of: prime, number of k's the ratio of $p-1$ to 6, and the percent difference between the expected and gained result:

	p	$\mathcal{K}_p$	$\frac{(p-1)}{6}$	$\Delta\%$
1	7	1	1	0.000
2	19	3	3	0.000
3	37	6	6	0.000
4	61	10	10	0.000
5	127	21	21	0.000
6	271	45	45	0.000
7	331	55	55	0.000
8	397	66	66	0.000
9	547	91	91	0.000
10	631	105	105	0.000
11	919	153	153	0.000
12	1657	276	276	0.000
13	1801	300	300	0.000
14	1951	325	325	0.000
15	2269	378	378	0.000
16	2437	406	406	0.000
17	2791	465	465	0.000
18	3169	528	528	0.000
19	3571	595	595	0.000

Table 1: k Data

3.5 Study of the k's

Regarding Conjecture 2.4, half of the primitive roots of p appear in $\mathcal{K}_p$. However, there have been cases where there are more than primitive roots in $\mathcal{K}_p$, thus leaving some "mystery number," M.

So we developed this formula:

$$\#\mathcal{K}_p = \frac{\#\text{ Primitive Roots of } p}{2} + M \tag{3.5.1}$$

Using Conjecture 2.4, solving for M directly we obtain:

$$M = \frac{(p-1) - 3(\#\text{ Primitive Roots})}{6} \tag{3.5.2}$$

Recalling from Abstract Algebra,

Theorem 3.1: Number of Primitive Roots

The number of primitive roots of a prime p is $\varphi(p-1)$

Thus, from Corollary 3.1 and Formula 3.5.2,

Conjecture 3.2: Value of the mystery number

The value of the mystery number M should be

$$M = \frac{(p-1) - 3\varphi(p-1)}{6} \tag{3.5.3}$$

Thus

Conjecture 3.3: Number of k's

The the overall number of k's for which $|E_k(p)| = p$ can be calculated from

$$\#\mathcal{K}_p = \frac{\#\text{ Primitive Roots of } p}{2} + \frac{(p-1) - 3\varphi(p-1)}{6} = \frac{p-1}{6} \tag{3.5.4}$$

Aside from the conjectures formulated above, we also considered the question "Why do half the primitive roots occur among these k's? Additionally, which half thereof?"

From this, no clear pattern emerged from our data.

4 Acknowledgements

Throughout the project, I received extensive support in the way of coding from Charles Burnell. He used Sage and IPython to generate data and stored the data as a MySQL database.

Further, a special thank you to Digital Ocean for allowing us to use their cloud server free of charge.

The most adamant and continued support came from my thesis advisor, Dr. Marion Scheepers, who worked tirelessly in order to assist me in developing a unique body of work. Thank you.

5 References

[1] L. Babinkostova, K. M. Bombardier, M. M. Cole, T. A. Morrell, and C. B. Scott. Elliptic pairs and elliptic reciprocity. Boise State REU.

[2] L. Babinkostova, K. M. Bombardier, M. M. Cole, T. A. Morrell, and C. B. Scott. Elliptic Reciprocity. *ArXiv e-prints*, December 2012.

[3] Reinier Bröker and Peter Stevenhagen. *Elliptic curves with a given number of points.*

[4] Lawrence C. Washington. *Elliptic Curves : number theory and cryptography.* Chapman & Hall/CRC, 2nd edition, 2008.

[5] Eric W. Weisstein. Diophantine equation.

[6] Wikipedia. Helmut hasse. Electronic.

[7] Wikipedia. Claude gaspard bachet de méziriac, April 2016.

[8] Wikipedia. Elliptic curve, April 2016.

6 Useful Information

6.1 Definitions

Definition 6.1: Plane Algebraic Curve

A *Plane Algebraic Curve* is the set of points on the Euclidean plane whose coordinates are zeros of some polynomial in two variables.

Plane Algebraic Curve wiki article

Definition 6.2: Group

A group G is a finite or infinite set of elements together with a binary operation (called the group operation) that together satisfy the four fundamental properties of closure, associativity, the identity property, and the inverse property. The operation with respect to which a group is defined is often called the "group operation," and a set is said to be a group "under" this operation.

Rowland, Todd and Weisstein, Eric W. "Group." From MathWorld–A Wolfram Web Resource.

Definition 6.3: Primitive Roots

The group $(U_p, * \mod p)$ is cyclic when p is prime.
Traditionally, generators of these particular cyclic groups have been called Primitive Roots of p.

6.2 Additional Information

Note 6.1: Facts from group theory

If g generates the finite group (G, Δ), say $k = \text{order}(g)$ then

$$\text{order}(g^m) = \frac{\text{order}(g)}{\gcd(m, k)}$$

$U_p = \{a \in \mathbb{Z}_p \; : \; \gcd(a, p) = 1\} \implies \{1, 2, \ldots, p-1\}$ as p is prime and $|U_p| = p - 1$ which is even if $p > 2$.

A Appendix: Sage and IPython Code for Data

A.1 All Data

This is the process that was used to generate data

```
def nextBatch(cur,conn):
    P=Primes()
    batchsize=1000

# Gets the next check
# Calculates what the new NextToCheck is
# Updates NextToCheck

    cur.execute("SELECT * FROM NextToCheck;")
    nextCheck = list(cur.fetchone())
    nextCheck = [Integer(x) for x in nextCheck]


#Right here we need to make sure that the low isnt the highest
    newLow = nextCheck[3]+1
    curPrime = nextCheck[1]
    curLow = nextCheck[2]
    curHigh = nextCheck[3]
    #makes it so if it is a new prime ti makes a zero for it so its
    easier to prime the database.
    if nextCheck[2]==1:
        cur.execute("INSERT INTO PrimeNumPrime (Prime,SPrime,SUidPrime) VALUES
         ({0},0,0)".format(int(curPrime)))
        cur.connection.commit();
    #print(nextCheck)
    if curPrime<=newLow:
        newLow=1
        newPrime = P.next(curPrime)

        if newLow+batchsize>=newPrime:
            newHigh=newPrime-1
        else:
            newHigh=newLow+batchsize
        updateQuery='UPDATE NextToCheck SET Prime={0},FromConstant={1},
        ToConstant={2} WHERE 1;'.format(newPrime,newLow,newHigh)
        #updateQuery="""UPDATE NextToCheck
        SET Prime=%s,FromConstant=%s,ToConstant=%s WHERE 1"""
        #print"{0} {1} {2}\n{3}".format(newPrime,newLow,newHigh,updateQuery)
        cur.execute(updateQuery)
        #cur.execute(updateQuery,(newPrime,newLow,newHigh,))
```

```
        cur.connection.commit();
    else:
        if newLow+batchsize>=curPrime:
            newHigh=curPrime-1
        else:
            newHigh=newLow+batchsize
        #print"{0} {1}".format(newLow,newHigh)
        updateQuery='UPDATE NextToCheck SET Prime={0},FromConstant={1},
        ToConstant={2} WHERE 1;'.format(curPrime,newLow,newHigh)
        cur.execute(updateQuery)
        cur.connection.commit();
    rowToAdd = []
    SPrime = 0
    SUidPrime=0
    # curHigh+1 because it is needed to makeit reach to p-1
    for b in xrange(curLow,curHigh+1):
        A=EllipticCurve(GF(curPrime),[0,b])
        sizeA=A.cardinality()
        if is_prime(sizeA):
            SUidPrime+=1
        if is_prime(sizeA-1):
            SPrime+=1
        rowToAdd.append((curPrime,b,sizeA))
    #print "SPrime={0},SUidPrime={1}".format(SPrime,SUidPrime)
    primeUpdate = 'UPDATE PrimeNumPrime SET SPrime=SPrime+{0},SUidPrime=
    SUidPrime+{1} WHERE Prime={2};'.format(SPrime, SUidPrime,curPrime)
    #print (primeUpdate)
    cur.execute(primeUpdate)
    cur.connection.commit()
    query = "insert into PrimeConstSize (Prime,Constant,Size) values "
    print ",".join(str(row) for row in rowToAdd)
    query += ",".join(str(row) for row in rowToAdd)
    #print query
    cur.execute(query)
    cur.connection.commit()
```

Searches for the first 100 of each class if they exist otherwise as many as possible.

```
# -*- coding: utf-8 -*-
"""
Spyder Editor

"""

import sqlite3

pathway = "D:\School\EllipticCurves\database.sqlite"
conn = sqlite3.connect(pathway)
runningtotal = 0
```

```
#BASE DATA

resultPrimesClass = conn.execute("SELECT COUNT(*) FROM PrimeNumPrime")
totalNumData = resultPrimesClass.fetchone()[0]
print "Number of primes {0}".format(totalNumData)



#CLASS 1 DATA
print "\n\n Class1"
SPrimePart = "0"
SUidPrimePart = "0"
QueryWhere = " WHERE SPrime={0} AND SUidPrime={1}".format(SPrimePart,SUidPrimePart)

resultPrimesClass = conn.execute("SELECT COUNT(*) FROM PrimeNumPrime"+QueryWhere)
numResults = resultPrimesClass.fetchone()[0]
print "Number of primes {0}".format(numResults)
runningtotal+=numResults

resultPrimesClass = conn.execute("SELECT * FROM PrimeNumPrime"+QueryWhere)

row = resultPrimesClass.fetchone()
search = ""
i =0
while row != None and i<100:
    Prime = row[0]
    SPrime = row[1]
    SUidPrime = row[2]
    #print Prime
    search+=str(Prime)+","
    row = resultPrimesClass.fetchone()
    i+=1

print search

#CLASS 2 DATA
print "\n\n Class2"
SPrimePart = "(Prime-1)/6"
SUidPrimePart = "0"
QueryWhere = " WHERE SPrime={0} AND SUidPrime={1}".format(SPrimePart,SUidPrimePart)

resultPrimesClass = conn.execute("SELECT COUNT(*) FROM PrimeNumPrime"+QueryWhere)
numResults = resultPrimesClass.fetchone()[0]
print "Number of primes {0}".format(numResults)
runningtotal+=numResults

resultPrimesClass = conn.execute("SELECT * FROM PrimeNumPrime"+QueryWhere)

row = resultPrimesClass.fetchone()
search = ""
```

```
i =0
while row != None and i<100:
    Prime = row[0]
    SPrime = row[1]
    SUidPrime = row[2]
    #print Prime
    search+=str(Prime)+","
    row = resultPrimesClass.fetchone()
    i+=1

print search

#CLASS 3 DATA
print "\n\n Class3"
SPrimePart = "(Prime-1)"
SUidPrimePart = "0"
QueryWhere = " WHERE SPrime={0} AND SUidPrime={1}".format(SPrimePart,SUidPrimePart)

resultPrimesClass = conn.execute("SELECT COUNT(*) FROM PrimeNumPrime"+QueryWhere)
numResults = resultPrimesClass.fetchone()[0]
print "Number of primes {0}".format(numResults)
runningtotal+=numResults

resultPrimesClass = conn.execute("SELECT * FROM PrimeNumPrime"+QueryWhere)

row = resultPrimesClass.fetchone()
search = ""
i =0
while row != None and i<100:
    Prime = row[0]
    SPrime = row[1]
    SUidPrime = row[2]
    #print Prime
    search+=str(Prime)+","
    row = resultPrimesClass.fetchone()
    i+=1

print search

#CLASS4 DATA
print "\n\n Class4"
SPrimePart = "0"
SUidPrimePart = "(Prime-1)/3"
QueryWhere = " WHERE SPrime={0} AND SUidPrime={1}".format(SPrimePart,SUidPrimePart)

resultPrimesClass = conn.execute("SELECT COUNT(*) FROM PrimeNumPrime"+QueryWhere)
numResults = resultPrimesClass.fetchone()[0]
print "Number of primes {0}".format(numResults)
runningtotal+=numResults
```

```
resultPrimesClass = conn.execute("SELECT * FROM PrimeNumPrime"+QueryWhere)

row = resultPrimesClass.fetchone()
search = ""
i =0
while row != None and i<100:
    Prime = row[0]
    SPrime = row[1]
    SUidPrime = row[2]
    #print Prime
    search+=str(Prime)+","
    row = resultPrimesClass.fetchone()
    i+=1

print search

#CLASS5 DATA
print "\n\n Class5"
SPrimePart = "(Prime-1)/6"
SUidPrimePart = "(Prime-1)/3"
QueryWhere = " WHERE SPrime={0} AND SUidPrime={1}".format(SPrimePart,SUidPrimePart)

resultPrimesClass = conn.execute("SELECT COUNT(*) FROM PrimeNumPrime"+QueryWhere)
numResults = resultPrimesClass.fetchone()[0]
print "Number of primes {0}".format(numResults)
runningtotal+=numResults

resultPrimesClass = conn.execute("SELECT * FROM PrimeNumPrime"+QueryWhere)

row = resultPrimesClass.fetchone()
search = ""
i =0
while row != None and i<100:
    Prime = row[0]
    SPrime = row[1]
    SUidPrime = row[2]
    #print Prime
    search+=str(Prime)+","
    row = resultPrimesClass.fetchone()
    i+=1

print search

#CLASS5 DATA
print "\n\n Class6"
SPrimePart = "0"
SUidPrimePart = "(Prime-1)/6"
QueryWhere = " WHERE SPrime={0} AND SUidPrime={1}".format(SPrimePart,SUidPrimePart)

resultPrimesClass = conn.execute("SELECT COUNT(*) FROM PrimeNumPrime"+QueryWhere)
```

```
numResults = resultPrimesClass.fetchone()[0]
print "Number of primes {0}".format(numResults)
runningtotal+=numResults

resultPrimesClass = conn.execute("SELECT * FROM PrimeNumPrime"+QueryWhere)

row = resultPrimesClass.fetchone()
search = ""
i =0
while row != None and i<100:
    Prime = row[0]
    SPrime = row[1]
    SUidPrime = row[2]
    #print Prime
    search+=str(Prime)+","
    row = resultPrimesClass.fetchone()
    i+=1

print search

#CLASS7 DATA
print "\n\n Class7"
SPrimePart = "(Prime-1)/6"
SUidPrimePart = "(Prime-1)/6"
QueryWhere = " WHERE SPrime={0} AND SUidPrime={1}".format(SPrimePart,SUidPrimePart)

resultPrimesClass = conn.execute("SELECT COUNT(*) FROM PrimeNumPrime"+QueryWhere)
numResults = resultPrimesClass.fetchone()[0]
print "Number of primes {0}".format(numResults)
runningtotal+=numResults

resultPrimesClass = conn.execute("SELECT * FROM PrimeNumPrime"+QueryWhere)

row = resultPrimesClass.fetchone()
search = ""
i =0
while row != None and i<100:
    Prime = row[0]
    SPrime = row[1]
    SUidPrime = row[2]
    #print Prime
    search+=str(Prime)+","
    row = resultPrimesClass.fetchone()
    i+=1

print search

print totalNumData-runningtotal

#CLASS DATA SUidPrime =0
```

```
print "\n\n Class all SUidPrime=0"
SPrimePart = "SPrime=SPrime"
SUidPrimePart = "SUidPrime=0"
QueryWhere = " WHERE {0} AND {1}".format(SPrimePart,SUidPrimePart)

resultPrimesClass = conn.execute("SELECT COUNT(*) FROM PrimeNumPrime"+QueryWhere)
numResults = resultPrimesClass.fetchone()[0]
print "Number of primes {0}".format(numResults)

resultPrimesClass = conn.execute("SELECT * FROM PrimeNumPrime"+QueryWhere)

row = resultPrimesClass.fetchone()
search = ""
i =0
while row != None and i<100:
    Prime = row[0]
    SPrime = row[1]
    SUidPrime = row[2]
    #print Prime
    search+=str(Prime)+","
    row = resultPrimesClass.fetchone()
    i+=1

print search

#CLASS DATA SUidPrime =(Prime-1)/3
print "\n\n Class all SUidPrime=(Prime-1)/3"
SPrimePart = "SPrime=SPrime"
SUidPrimePart = "SUidPrime=(Prime-1)/3"
QueryWhere = " WHERE {0} AND {1}".format(SPrimePart,SUidPrimePart)

resultPrimesClass = conn.execute("SELECT COUNT(*) FROM PrimeNumPrime"+QueryWhere)
numResults = resultPrimesClass.fetchone()[0]
print "Number of primes {0}".format(numResults)

resultPrimesClass = conn.execute("SELECT * FROM PrimeNumPrime"+QueryWhere)

row = resultPrimesClass.fetchone()
search = ""
i =0
while row != None and i<100:
    Prime = row[0]
    SPrime = row[1]
    SUidPrime = row[2]
    #print Prime
    search+=str(Prime)+","
    row = resultPrimesClass.fetchone()
    i+=1

print search
```

```
#CLASS DATA SUidPrime =(Prime-1)/6
print "\n\n Class all SUidPrime=(Prime-1)/6"
SPrimePart = "SPrime=SPrime"
SUidPrimePart = "SUidPrime=(Prime-1)/6"
QueryWhere = " WHERE {0} AND {1}".format(SPrimePart,SUidPrimePart)

resultPrimesClass = conn.execute("SELECT COUNT(*) FROM PrimeNumPrime"+QueryWhere)
numResults = resultPrimesClass.fetchone()[0]
print "Number of primes {0}".format(numResults)

resultPrimesClass = conn.execute("SELECT * FROM PrimeNumPrime"+QueryWhere)

row = resultPrimesClass.fetchone()
search = ""
i =0
while row != None and i<100:
    Prime = row[0]
    SPrime = row[1]
    SUidPrime = row[2]
    #print Prime
    search+=str(Prime)+","
    row = resultPrimesClass.fetchone()
    i+=1

print search

#CLASS DATA SPrime =0
print "\n\n Class all SPrime=0"
SPrimePart = "SPrime=0"
SUidPrimePart = "SUidPrime=SUidPrime"
QueryWhere = " WHERE {0} AND {1}".format(SPrimePart,SUidPrimePart)

resultPrimesClass = conn.execute("SELECT COUNT(*) FROM PrimeNumPrime"+QueryWhere)
numResults = resultPrimesClass.fetchone()[0]
print "Number of primes {0}".format(numResults)

resultPrimesClass = conn.execute("SELECT * FROM PrimeNumPrime"+QueryWhere)

row = resultPrimesClass.fetchone()
search = ""
i =0
while row != None and i<100:
    Prime = row[0]
    SPrime = row[1]
    SUidPrime = row[2]
    #print Prime
    search+=str(Prime)+","
    row = resultPrimesClass.fetchone()
    i+=1
```

```
print search



#CLASS DATA SPrime =(Prime-1)/6
print "\n\n Class all SPrime=(Prime-1)/6"
SPrimePart = "SPrime=(Prime-1)/6"
SUidPrimePart = "SUidPrime=SUidPrime"
QueryWhere = " WHERE {0} AND {1}".format(SPrimePart,SUidPrimePart)

resultPrimesClass = conn.execute("SELECT COUNT(*) FROM PrimeNumPrime"+QueryWhere)
numResults = resultPrimesClass.fetchone()[0]
print "Number of primes {0}".format(numResults)

resultPrimesClass = conn.execute("SELECT * FROM PrimeNumPrime"+QueryWhere)

row = resultPrimesClass.fetchone()
search = ""
i =0
while row != None and i<100:
    Prime = row[0]
    SPrime = row[1]
    SUidPrime = row[2]
    #print Prime
    search+=str(Prime)+","
    row = resultPrimesClass.fetchone()
    i+=1

print search
```

This is the tree building program that finds the where each prime travels for each one

```
# -*- coding: utf-8 -*-
"""
Created on Mon Dec 14 17:18:48 2015

@author: Charles
"""

import pickle


#,PNPDict,PCSDict
def RecursionPrep(prime,PNPDict,PCSDict):

    #print len(({prime:[]},[],prime,PNPDict,PCSDict))
    return RecursionWork({prime:[]},[],prime,PNPDict,PCSDict)
```

```
def RecursionWork(dict,checked,prime,PNPDict,PCSDict):
    toCheck={}
    stillgoing = False
    for key in dict:
        if key not in checked:
            toCheck[key]=PCSDict[key]
            checked.append(key)
    Plist=[]
    for key in toCheck:
        for num in toCheck[key]:
            if num in PNPDict:
                Plist.append(num)
                if num not in dict.keys():
                    dict[num]=[]
                dict[key].append(num)

    for keys in dict:
        if keys not in checked:
            stillgoing=True
    if stillgoing:
        return RecursionWork(dict,checked,prime,PNPDict,PCSDict)
    else:
        return dict
PNPDict=pickle.load(open("PNPDict.p","rb"))
PCSDict=pickle.load(open("PCSDict.p","rb"))



This is the basis of the code we used to find the points for the weak elliptic cycle
\begin{verbatim}
foundupint=0
founddownint=0
foundup=False
founddown = False
for test in xrange(1,primecheck):
    A = EllipticCurve(GF(primecheck),[0,test]).cardinality()
    if A==primedown and not founddown:
        founddown=True
        founddownint=test
    if A==primeup and not foundup:
        foundup=True
        foundupint=test
    if foundup and founddown:
        break
founddownint
foundupint
```

A.2 p, $\mathcal{S}$, & $|E_k(p)|$ data

This is the process that was used to generate data for finding the following:

p	$\mathcal{S}$	$\lvert E_k(p)\rvert$

```
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 04 15:51:27 2016

@author: Charles
"""
import sqlite3

pathway = "D:\School\EllipticCurves\database.sqlite"
conn = sqlite3.connect(pathway)

results =[]
cubanPrimes = (7, 19, 37, 61, 127, 271, 331, 397, 547, 631, 919, 1657, 1801,
1951, 2269, 2437, 2791, 3169, 3571, 4219, 4447, 5167, 5419, 6211, 7057, 7351,
8269, 9241, 10267, 11719, 12097, 13267, 13669, 16651, 19441, 19927, 22447,
23497, 24571, 25117, 26227)
for cprime in cubanPrimes:
#    print "SELECT * FROM PrimeNumPrime WHERE Prime in {0}".format(cprime)
    resultCubanPrimes = conn.execute("SELECT * FROM PrimeNumPrime WHERE Prime
     in ({0})".format(cprime))
    results.append(resultCubanPrimes.fetchall())


for result in results:
    print result
```

A.3 k Data Correlating to the Cuban Primes

This is the process that was used to generate data for finding the constant k for which $|E_k(p)| = p$.

```
constant data finding program

# -*- coding: utf-8 -*-
"""
Created on Mon Apr 04 15:51:27 2016

@author: Charles
"""
import sqlite3

pathway = "D:\School\EllipticCurves\database.sqlite"
conn = sqlite3.connect(pathway)

results =[]
cubanPrimes = (7, 19, 37, 61, 127, 271, 331, 397, 547, 631, 919, 1657, 1801,
1951, 2269, 2437, 2791, 3169, 3571, 4219, 4447, 5167, 5419, 6211, 7057, 7351,
 8269, 9241, 10267, 11719, 12097, 13267, 13669, 16651)
for cprime in cubanPrimes:
#    print "SELECT * FROM PrimeNumPrime WHERE Prime in {0}".format(cprime)
    query = "SELECT * FROM PrimeConstSize WHERE Prime ={0} and Size = {0}"
    .format(cprime)
    print query
    resultCubanPrimes = conn.execute(query)
    results.append(resultCubanPrimes.fetchall())


for result in results:
    print "For Prime {0}".format(result[0][0])
    constants = []
    for subresult in result:
        constants.append(subresult[1])
    print constants
```