Project Number: CP-0100

ELLIPTIC CURVE GENERATION BY COMPLEX MULTIPLICATION IN JAVA

A Major Qualifying Project Report:

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Seth Hardy

Date: December 1, 2000

Approved:

Professor Christof Paar, Advisor

1. **cryptography**
2. **elliptic**
3. **java**

Abstract

As elliptic curves become more accepted and used in the field of cryptography, more curves will be needed for the growing number of elliptic curve cryptosystems. Traditional trial and error methods of curve generation can take a long time, and do not always produce a curve with desired characteristics. For an elliptic curve reference library, a curve generator was designed and written in the Java programming language, using the method of complex multiplication, desirable for its efficiency and flexibility.

Acknowledgments

I would like to thank my advisor, Professor Christof Paar, for all of his help in setting this project up, and for the assistance along the way. I would also like to thank, for their help, the people I worked for or with over the summer, at the Institute for Data Communication Systems of the University of Siegen, Germany: Professor Christoph Ruland, Oliver Weissmann, and Michael Schmidt.

Contents

1 Introduction **1**
1.1 Project Description 1

2 Mathematical Background **2**
2.1 Elliptic Curves 2
2.1.1 General Forms 2
2.1.2 Arithmetic on Elliptic Curves over $GF(p)$ 3
2.1.3 Other Properties of Elliptic Curves 4
2.1.4 Elliptic Curve Generation 5
2.2 Complex Multiplication 6
2.2.1 Selecting a Prime p for $GF(p)$ 6
2.2.2 Finding a Complex Multiplication Discriminant 6
2.2.3 Finding a Nearly Prime Order for the Curve 9
2.2.4 Calculating the Weber Polynomial 10
2.2.5 Finding Curve Coefficients 12
2.2.6 Finding a Generator for the Curve 14

3 Software Design **16**
3.1 Overview 16
3.2 Basic Requirements 16
3.3 Existing Java Foundations 17
3.4 Class Structure of the EC Generator 18
3.4.1 CMHelper 18
3.4.2 Complex 19
3.4.3 ECMath 19
3.4.4 GFpEC 19
3.4.5 GFpGenerator 19
3.4.6 PolyMath 20

3.4.7 ecgenerate 20

4 Implementation **21**
4.1 Implementation Overview 21
4.2 Difficuties in Implementation 21
4.3 Performance 24
4.4 Performance Analysis 24

5 Conclusions **25**
5.1 Summary 25
5.2 Future Work 25

A1Source Code **27**
A1.1 CMHelper.java 28
A1.2 Complex.java 39
A1.3 ECMath.java 43
A1.4 GFpEC.java 46
A1.5 GFpGenerator.java 49
A1.6 PolyMath.java 56
A1.7 ecgenerate.java 61

A2Program Documentation **64**

A3Program Timings **94**

List of Tables

4.1 Performance of the Elliptic Curve Generator, in seconds 24

A3.1 Performance of the Elliptic Curve Generator, in seconds 94

List of Figures

4.1 Organization of Generator Classes 22

Chapter 1

Introduction

1.1 Project Description

The purpose of this project is to implement elliptic curve generation using the method of complex multiplication in the Java programming language, for the ELIAS (Elliptic Curve Cryptography Standards Reference Implementation) reference library.

ELIAS is sponsored by the European Union, headed by the University of Siegen in Siegen, Germany. The goal of the ELIAS project is to create an elliptic curve reference library that is easy to understand, implement, and learn from. For these reasons, the library is being written in the Java programming language, using the object-oriented paradigm of programming. Documentation for the library should include well-commented code, and use of the Java documentation standard, so that others can easily learn from and reuse the written code. For more information on the ELIAS project, see [3].

Chapter 2

Mathematical Background

2.1 Elliptic Curves

2.1.1 General Forms

The general form for elliptic curve equations over a field K is:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{2.1}$$

where $a_i \in K$. This equation is known as the *affine Weierstrass equation*.

For cryptography, elliptic curves are used over finite fields of the form $GF(p^m)$, where p is a prime number. The two types of finite field most commonly used are $GF(p)$ and $GF(2^m)$. The general affine Weierstrass equation can be simplified for each case:

- For $GF(p)$:

  $$E : y^2 = x^3 + ax + b \tag{2.2}$$

  where $a, b \in GF(p)$ and $4a^3 + 27b^2 \neq 0 \bmod p$. This form is known as the *short Weierstrass form*.

- For $GF(2^m)$:

  $$E : y^2 + xy = x^3 + ax^2 + b \tag{2.3}$$

  where $a, b \in GF(2^m)$ and $b \neq 0$.

Points of the elliptic curve E are the points (x, y) that satisfy the equation for the curve, plus O, the point at infinity. The ELIAS project only uses elliptic curves over $GF(p)$, so those will be focused on here; however, more information on elliptic curves over $GF(2^m)$ and the more general case over $GF(p^m)$ can be found in a number of sources, including [6] for simple explanations and [1] for a more thorough theoretical background.

2.1.2 Arithmetic on Elliptic Curves over $GF(p)$

The addition operation $P + Q = R$, where $P = (x_1, y_1)$, $Q = (x_2, y_2)$, $R = (x_3, y_3)$ and $x_i, y_i \in GF(p)$, is defined as:

$$\lambda = \begin{cases} \frac{3x_1^2+a}{2y_1} & \text{if } x_1 = x_2,\ y \neq 0 \\ \frac{y_2-y_1}{x_2-x_1} & \text{otherwise} \end{cases} \tag{2.4}$$

$$x_3 = \lambda^2 - x_1 - x_2 \tag{2.5}$$

$$y_3 = \lambda(x_1 - x_3) - y_1 \tag{2.6}$$

The point at infinity O is similar to the number zero in the additive group of integers, acting as the additive identity:

$$P + O = P = O + P \tag{2.7}$$

The inverse of a point $P = (x, y)$ is equal to

$$P^{-1} = (x, -y) \tag{2.8}$$

From these definitions, the points on an elliptic curve form a group, with a binary operator (elliptic curve addition), identity element (the point at infinity), and existence of inverses. Groups over elliptic curves can be used in the place of other groups used in more traditional cryptography, such as the multiplicative group of the integers modulo p. The parts of an elliptic curve system correspond directly to those of a discrete logarithm system. Instead of using the integers modulo p, the set of points on the elliptic curve are used; instead of modular multiplication, elliptic curve addition is used. By extension, modular exponentiation is equivalent to multiplication of a point by an integer. For more information, see [6].

2.1.3 Other Properties of Elliptic Curves

To determine whether an elliptic curve is suitable for use in cryptography, a few of its other characteristics need to be determined.

The *order* of a curve (denoted $\sharp E$) is the number of points on the curve, including the point at infinity. The order of a point P is the smallest positive integer r such that $rP = O$. The order of a point is always a divisor of the order of the curve, due to Lagrange's Theorem (see [2], or any other book on abstract algebra). A point is called a *generator* if its order equals the order of the group.

For cryptography, one of the desired properties of a curve is that its order is *smooth*, that is, $\sharp E = kr$, where r is a large prime integer, and k a small integer. (Large and small are relative terms; the actual group of points being used is a subgroup of order r, so r should be as close to the desired bit length as possible.) A generator of order r needs to be calculated, so the order $\sharp E$ of a curve generally needs to be known.

The *Hasse Bound* for the order of a curve gives a limit on the possible values of $\sharp E$:

$$p - 2\sqrt{p} + 1 \leq \sharp E \leq p + 2\sqrt{p} + 1 \tag{2.9}$$

So, the order $\sharp E \approx p$. This is a very useful property to know when calculating the order of a curve, whether through a brute force method, or a more sophisticated technique (such as complex multiplication).

The MOV condition of an elliptic curve is whether the curve is resistant to the attack of Menezes, Okamoto, and Vanstone. (A more detailed explanation of this attack can be found in [1]; it was originally published in [5].) A curve is resistant to the MOV attack if finding elliptic curve discrete logarithms is as least as hard in $GF(q^l)$ as it is in $GF(q)$, i.e., the smallest value of l such that $q^l = 1 \bmod \sharp E$ is large. The exact value of l that the condition is tested against is based on the bit size of the curve. A table giving values used in implementation can be found in [6].

Curves should also be tested to see if they are anomalous; a curve is anomalous if $\sharp E = p$. Elliptic curves that fail the MOV condition, or are anomalous, are not appropriate for use in cryptography due to specialized attacks on these types of curves.

2.1.4 Elliptic Curve Generation

Finding a curve suitable for use in cryptography, with the abovementioned properties, is known as curve generation. There are three general methods of elliptic curve generation:

1. Trial and error

   Generate curves at random, computing the orders of each curve, until one with the desired properties is found.

2. Subfield Curves

   For $GF(2^m)$: If m is divisible by a (usually small) integer d, then a curve can be selected over $GF(2^d)$, and its order over $GF(2^m)$ can be calculated easily.

3. Complex Multiplication

   Pick a group order first, and then construct an elliptic curve for that order.

For the trial and error method, a curve is generated through random selection of curve coefficients and modulus. The order of the curve is computed through a point counting algorithm, such as the well known one by Schoof. The complexity of the standard version of Schoof's algorithm is $O(\log^8 p)$. In practice, improved versions of Schoof's algorithm are used, such as the Schoof-Elkies-Atkin algorithm. No matter the method, however, point counting is still a time consuming process.

When generating curves through the trial and error method, a bound on acceptable loss needs to be decided. Loss is defined as

$$\epsilon = 1 - \frac{\log_2 r}{\sharp E} \tag{2.10}$$

which equals the percentage of bits "lost" in the prime order subgroup from the entire curve order. If the loss is greater than the bound deemed as acceptable, then a new curve needs to be generated. The probability of a curve having loss less than or equal to ϵ is equal to ϵ.

2.2 Complex Multiplication

If a "good" curve order is selected first, and then the curve computed from that order, a point counting algorithm will not be needed. Additionally, the curve order can be a lot more finely tuned without the need for long searches for appropriate curves.

Generation of a curve in this fashion uses the method of complex multiplication. Generating a curve by complex multiplication involves the following steps:

1. Calculating the group order
   (a) Selecting a prime p for $GF(p)$
   (b) Finding a complex multiplication discriminant
   (c) Finding a nearly prime order for the curve
2. Calculating the curve parameters
   (a) Calculating the Hilbert/Weber polynomial (reduced class polynomial)
   (b) Finding curve coefficients
   (c) Finding a generator for the curve

2.2.1 Selecting a Prime p for $GF(p)$

Selecting a prime is simple: randomly generate a prime of the desired bit length.

Section A.15.6 of [6] gives an algorithm for generating random primes; also, Java's `BigInteger` class has a constructor that will generate primes of a desired size.

2.2.2 Finding a Complex Multiplication Discriminant

Finding a good group order can be done without knowing the curve, first. Manipulation of the Hasse Bound equation (2.9) gives the following:

$$4p \geq (p + 1 - \sharp E)^2 \tag{2.11}$$

This shows that $4p - (p + 1 - \sharp E)^2$ is positive. Now, let

$$DV^2 = 4p - (p + 1 - \sharp E)^2 \tag{2.12}$$

where D is a unique integer that is positive and squarefree (having no factors that are squares). Therefore, the factorization of Z is unique.

Also, let

$$W^2 = (p + 1 - \sharp E)^2 \tag{2.13}$$

which gives

$$4p = W^2 + DV^2 \tag{2.14}$$

Equation 2.14 is known as a *diophantine equation.* Solving the equation for W, and then rearranging the terms to solve for $\sharp E$, gives the following:

$$\sharp E = p + 1 \pm W \tag{2.15}$$

By knowing p and D (by picking any D that works), $\sharp E$ can be calculated, by first calculating W. D is called the *complex multiplication discriminant* for the curve.

There are specific congruence conditions for D to be a complex multiplication discriminant. Let

$$K = \lfloor \frac{(\sqrt{p} + 1)^2}{r_{min}} \rfloor \tag{2.16}$$

where r_{min} is the minimum order for the large subgroup of prime order. The congruence conditions for D (from [6], section A.14.2.1) are then:

$$D \equiv \begin{cases} 2, 3, 7 \pmod 8 & \text{if } p \equiv 3 \pmod 8 \\ 1, 3, 5, 7 \pmod 8 & \text{if } p \equiv 5 \pmod 8 \\ 3, 6, 7 \pmod 8 & \text{if } p \equiv 7 \pmod 8 \\ 3 \pmod 8 & \text{if } K = 1 \\ 0, 1, 2, 3, 4, 5, 6 & \text{if } K = 1, 2 \end{cases}$$

The algorithm in section A.14.2.2 of [6] gives the method for solving the diophantine equation, given the prime p and a squarefree positive integer value for D that satisfies the above congruence conditions. If the value for D makes it a valid complex multiplication discriminant, the algorithm will

return the value for W, which can then be used to find a possible order for the curve.

Algorithm A.14.2.2 from [6]

1. Find the square root modulo p of $-D$ or determine that none exist.
2. If the result of Step 1 indicates that no square roots exist, output "not a CM discriminant" and stop. Otherwise, the output of Step 1 is an integer B modulo p.
3. Let $A \leftarrow p$ and $C \leftarrow (B^2 + D)/p$.
4. Let $S \leftarrow \begin{pmatrix} A & B \\ B & C \end{pmatrix}$ and $U \leftarrow \begin{pmatrix} 1 \\ 0 \end{pmatrix}$
5. Until $|2B| \leq A \leq C$, repeat the following steps:
   (a) Let $\delta \leftarrow \lfloor \frac{B}{C} + \frac{1}{2} \rfloor$.
   (b) Let $T \leftarrow \begin{pmatrix} 0 & -1 \\ 1 & \delta \end{pmatrix}$
   (c) Replace U by $T^{-1}U$.
   (d) Replace S by $T^t S T$, where T^t denotes the transpose of T.
6. If $D = 11$ and $A = 3$, let $\delta \leftarrow 0$ and repeat 5b, 5c, 5d.
7. Let $U = \begin{pmatrix} X \\ Y \end{pmatrix}$.
8. If $D = 1$ or 3 then output $W \leftarrow 2X$ and $V \leftarrow 2Y$ and stop.
9. If $A = 1$ then output $W \leftarrow 2X$ and stop.
10. If $A = 4$ then output $W \leftarrow 4X + BY$ and stop.
11. Output "not a CM discriminant."

As can be seen from the algorithm, not all values of D will be complex multiplication discriminants, even if they do satisfy the congruence conditions.

2.2.3 Finding a Nearly Prime Order for the Curve

To find a curve order that takes the form $\sharp E = kr$ for a small integer k and large prime r, a trial and error method can be used on potential complex multiplication discriminants. This trial and error method is described in Section A.14.2.3 of [6]:

Algorithm A.14.2.3 from [6]

1. Choose a squarefree positive integer D, not already chosen, satisfying the congruence conditions.
2. Compute the Jacobi symbol $J = \left(\frac{-D}{p}\right)$. If $J = -1$ then go to Step 1.
3. List the odd primes l dividing D.
4. For each l, compute the Jacobi symbol $J = \left(\frac{p}{l}\right)$. If $J = -1$ for some l, then go to Step 1.
5. Test via algorithm A.14.2.3 whether D is a complex multiplication discriminant for p. If the result is "not a CM discriminant", go to Step 1. (Otherwise, the result is the integer W, along with V if $D = 1$ or $D = 3$.)
   - If $D = 1$ the orders are:
   $$p + 1 \pm W, p + 1 \pm V$$
   - If $D = 3$ the orders are:
   $$p + 1 \pm W, p + 1 \pm (W + 3V)/2, p + 1 \pm (W - 3V)/2$$
   - Otherwise the orders are
   $$p + 1 \pm W$$
6. Test each order for near-primality. If any order is nearly prime, output (D, k, r) and stop.
7. Go to Step 1.

This algorithm looks for good values for D, and when it finds one, attempts to solve the diophantine equation using that D. If there exists a solution, then there are multiple possible orders for a curve with that complex multiplication discriminant; the algorithm takes the first one that satisfies the condition of being nearly prime.

2.2.4 Calculating the Weber Polynomial

Calculation of the Weber polynomial, also known as the reduced class polynomial, is where the actual complex multiplication, for which the method is given its name, is done.

A reduced class polynomial for the complex multiplication discriminant D needs to be calculated, to later construct the elliptic curve with a known order.

The first step in calculating the reduced class polynomial is to find the class group and class number of D. Let

$$S = \begin{pmatrix} A & B \\ B & C \end{pmatrix} \tag{2.17}$$

where A, B, C are integers, and for all $A, B, C, \gcd(A, 2B, C) = 1, |2B| \leq A \leq C$, and if either $A = |2B|$ or $A = C$, then $B \geq 0$. A matrix of this form is known as a *reduced symmetric matrix.*

The determinant of the matrix is found in the usual way, and is equal to D:

$$D = AC - B^2 \tag{2.18}$$

The set of all reduced symmetric matrices of determinant D is known as the class group $H(D)$; the number of reduced symmetric matrices of determinant D is known as the class number $h(D)$.

The algorithm from Section A.13.2 of [6] gives a method for finding the class group and number of a squarefree determinant D:

Algorithm A.13.2 from [6]

1. Let s be the largest integer less than $\sqrt{D/3}$.
2. For B from 0 to s do:

(a) List the positive divisors $A_1, A_2, \cdots, A_r$ of $D + B^2$ that satisfy $2B \leq A \leq \sqrt{D + B^2}$.

(b) For i from 1 to r do:

i. Set $C \leftarrow (D + B^2)/A_i$

ii. If $\gcd(A_i, 2B, C) = 1$ then list $\begin{pmatrix} A_i & B \\ B & C \end{pmatrix}$

iii. If $\gcd(A_i, 2B, C) = 1$ and $0 < 2B < A_i < C$ then list $\begin{pmatrix} A_i & -B \\ -B & C \end{pmatrix}$

Once the class group has been found, the class invariant of each reduced symmetric matrix needs to be calculated. Section A.13.3 from [6] describes the following method for calculating the class invariant. (For more detailed information on why this works, see [1]).

$$\mathcal{C}(A, B, C) = (N\lambda^{-BL}2^{-I/6}(f_J(A, B, C))^K)^G \tag{2.19}$$

Where A, B, C are the values from the matrix, and the remainder of the values are calculated in the following ways:

$$G = \gcd(D, 3)$$

$$I = \begin{cases} 3 & \text{if } D \equiv 1,2,6,7 \pmod 8 \\ 0 & \text{if } D \equiv 3 \pmod 8 \text{ and } D \not\equiv 0 \pmod 3 \\ 2 & \text{if } D \equiv 3 \pmod 8 \text{ and } D \equiv 0 \pmod 3 \\ 6 & \text{if } D \equiv 5 \pmod 8 \end{cases}$$

$$J = \begin{cases} 0 & \text{for } AC \text{ odd} \\ 1 & \text{for } C \text{ even} \\ 2 & \text{for } A \text{ even} \end{cases}$$

$$K = \begin{cases} 2 & \text{if } D \equiv 1,2,6 \pmod 8 \\ 1 & \text{if } D \equiv 3,7 \pmod 8 \\ 4 & \text{if } D \equiv 5 \pmod 8 \end{cases}$$

$$L = \begin{cases} A - C + A^2C & \text{if } AC \text{ odd or } D \equiv 5 \pmod 8 \text{ and } C \text{ even} \\ A + 2C - AC^2 & \text{if } D \equiv 1,2,3,6,7 \pmod 8 \text{ and } C \text{ even} \\ A - C + 5AC^2 & \text{if } D \equiv 3 \pmod 8 \text{ and } A \text{ even} \\ A - C - AC^2 & \text{if } D \equiv 1,2,5,6,7 \pmod 8 \text{ and } A \text{ even} \end{cases}$$

$$M = \begin{cases} (-1)^{(A^2-1)/8} & \text{if } A \text{ odd} \\ (-1)^{(C^2-1)/8} & \text{if } A \text{ even} \end{cases}$$

$$
\begin{aligned}
N &= \begin{cases} 1 & \text{if } D \equiv 5 \pmod 8 \text{ or } D \equiv 3 \pmod 8 \text{ and } AC \text{ odd or } D \equiv 7 \pmod 8 \text{ and } AC \text{ even} \\ M, & \text{if } D \equiv 1,2,6 \pmod 8 \text{ or } D \equiv 7 \pmod 8 \text{ and } AC \text{ odd} \\ -M & \text{if } D \equiv 3 \pmod 8 \text{ and } AC \text{ even} \end{cases} \\
\lambda &= e^{\pi \imath K/24}
\end{aligned}
$$

Calculation of the f_J function is done as follows:

$$F(z) = 1 + \sum_{j=1}^{\infty} (-1)^j (z^{(3j^2-j)/2} + z^{(3j^2+j)/2}) \tag{2.20}$$

$$\theta = e^{\frac{-\sqrt{D}+B\imath}{A}\pi} \tag{2.21}$$

$$
\begin{aligned}
f_0(A,B,C) &= \theta^{-1/24} F(-\theta)/F(\theta^2) && (2.22) \\
f_1(A,B,C) &= \theta^{-1/24} F(\theta)/F(\theta^2) && (2.23) \\
f_2(A,B,C) &= \sqrt{2}\theta^{1/12} F(\theta^4)/F(\theta^2) && (2.24)
\end{aligned}
$$

Finally, polynomials created from the class invariants of each reduced symmetric matrix need to be multiplied by eachother. This step is the complex multiplication that gives the method its name; class invariants are complex numbers, and so the intermediate polynomials will have complex coefficients. The final result should be a polynomial of degree h, where h is the class number of D. This polynomial will have integer coefficients.

The reduced class polynomial is calculated through the following equation:

$$w_D(t) = \prod_{j=1}^{h} (t - \mathcal{C}(A_j, B_j, C_j)) \tag{2.25}$$

For more information on reduced class polynomials (both Weber and Hilbert polynomials, see [1].

2.2.5 Finding Curve Coefficients

The reduced class polynomial is used in the generation of an elliptic curve with a given order. Constructing the curve from an order is done in two steps: finding a curve with complex multiplication by the complex multiplication discriminant D, and then finding a related curve that also has the desired

order (along with a generator for the large prime subgroup). The algorithm for the first step can be found in [6], Section A.14.4.1, and for the second step in A.14.4.2.

Algorithm A.14.4.1 from [6]

1. Compute $w(t) \leftarrow w_D(t) \bmod p$.

2. Calculate W by solving the diophantine equation for some D (Algorithm A.14.2.2).

3. If W is even, then compute a linear factor $t - s$ of $w_D(t) \bmod p$. Let $V = (-1)^D 2^{4I/K} s^{24/(GK)} \bmod p$, where G, I, K are the values from calculating the reduced class polynomial.

4. If W is odd, then compute a cubic factor $g(t)$ of $w_D \bmod p$. Perform the following computations, in which the coefficients of the polynomials are integers modulo p:

$$\begin{aligned}
V(t) &= \begin{cases} -256t^8 \bmod g(t) & D \bmod 3 \equiv 0 \\ -t^{24} \bmod g(t) & \text{otherwise} \end{cases} \\
a_1(t) &= -3(V(t)+64)(V(t)+256) \bmod g(t) \\
b_1(t) &= 2(V(t)+64)^2(V(t)-512) \bmod g(t) \\
a_3(t) &= a_1(t)^3 \bmod g(t) \\
b_2(t) &= b_1(t)^2 \bmod g(t)
\end{aligned}$$

   Let σ be a nonzero coefficient from $a_3(t)$, and let τ be the corresponding coefficient from $b_2(t)$. Let

$$\begin{aligned}
a_0 &= \sigma\tau \bmod p \\
b_0 &= \sigma\tau^2 \bmod p
\end{aligned}$$

5. Output (a_0, b_0).

The output from the above algorithm will give the curve

$$y^2 = x^3 + a_0 x + b_0 \bmod p \tag{2.26}$$

The parameters a_0, b_0, along with the previously computed values p, k, r, will give the final curve with the following algorithm.

Algorithm A.14.4.2 from [6]

1. Select an integer ξ such that $0 < \xi < p$.
2. If $D = 1$ then set $a \leftarrow a_0 \xi \bmod p$, and $b \leftarrow 0$.
   If $D = 3$ then set $a \leftarrow 0$ and $b \leftarrow b_0 \xi \bmod p$.
   Otherwise, set $a \leftarrow a_0 \xi^2 \bmod p$ and $b \leftarrow b_0 \xi^3 \bmod p$.
3. Look for a point G of order r on the curve $y^2 = x^3 + ax + b \pmod p$. (See the next section for a description of this step.)
4. If the output is "wrong order", then output the message "wrong order" and stop.
5. Output the coefficients a, b and the point G.

This algorithm gives the final curve

$$E : y^2 = x^3 + ax + b(\bmod p) \tag{2.27}$$

2.2.6 Finding a Generator for the Curve

Algorithm A.11.3 from [6] is used for finding a point of order r on an elliptic curve of order $\sharp E = kr$, where k is a small integer and r a large prime.

Algorithm A.11.3 from [6]

1. Generate a random point P (not O) on E.
2. Set $G \leftarrow kP$.
3. If $G = O$ then go to Step 1.
4. Set $Q \leftarrow rG$.

5. If $Q = O$ then output "wrong order" and stop.
6. Output G.

Chapter 3

Software Design

3.1 Overview

The goal of the ELIAS project, as stated in the introduction to this report, is to create an elliptic curve reference library that is easy to understand, implement, and learn from. The aspect of the project that I worked on was the elliptic curve generator.

Elliptic curve generation is a "standalone" component of the reference library. The curve generator should be able to function on its own, as an independent program, used to generate curves that the rest of the library will then work with.

As with rest of the ELIAS project, all code written needs to be in the Java programming language.

3.2 Basic Requirements

The basic requirements of the elliptic curve generator are:

- **Functionality**. The generator should generate elliptic curves suitable for use in cryptography through the method of complex multiplication.

- **Simplicity**. The generator should be as simple as possible, so that it is as easy to understand as possible.

- **Documentation**. To help other people understand the code, for both learning how complex multiplication works, and how to reuse any al-

ready written code, there should be adequate documentation (in the form of external documents and comments) to explain what the written code is doing.

Although the program was intended to be a standalone generator, separate from the rest of the reference library, I decided to add the following requirements to the design criteria:

- **Reuseability**. Because the program is being written in Java, it will be a collection of classes. Java classes can be easily reused, as per the object-oriented paradigm; it could be seen as a waste to program a class that does the elliptic curve generation, which cannot be reused in another program by simply instantiating it.

- **Focus on the given task**. Although the generation process does need elliptic curve point arithmetic functions, for example, an entire elliptic curve point arithmetic library is not necessary. Only the required functions need to be written.

- **Allow later integration with the rest of the library**. For parts of the generator that are being written as a different part of the library (elliptic curve point arithmetic operations, for example), the generator should be written in a way that it would be simple to convert over to the "proper" way of doing things at a later point, when the rest of the reference library is written.

3.3 Existing Java Foundations

As is common in public-key cryptography, the elliptic curve generator program needs support for numbers larger than the standard long integer or double floating point types provided in most programming languages.

Java has two standard classes for this purpose, `java.math.BigInteger`, and `java.math.BigDecimal`. `BigInteger` provides support for immutable, arbitrary precision integers; `BigDecimal` provides similar support for immutable, arbitrary-precision floating point numbers.

`BigInteger` and `BigDecimal` remove the need for an external multiprecision number library, such as the GNU Multiprecision library (GMP) for C/C++. This reduces the difficulty of writing the generator, as it is not necessary to write a multiprecision number library specifically for this project.

`BigInteger` also has support for a few needed functions, such as primality testing and random prime number generation.

3.4 Class Structure of the EC Generator

The design criteria for the generator has a strong influence on the structure of the generator program. Instead of a single program, the generator should be implemented as a package of classes; this way, the generator will be easier to understand, and much simpler to reuse (whether as a whole, or specific parts). Because all of the necessary classes are implemented in a package, any other program can include the package with a single `import` statement, and call the generator with a single function call.

The structure of the generator package is broken down into seven classes, based on content:

- CMHelper
- Complex
- ECMath
- GFpEC
- GFpGenerator
- PolyMath
- ecgenerate

3.4.1 CMHelper

`CMHelper` is a library (of static functions) that the generator needs to properly generate elliptic curves by the method of complex multiplication. This is where most of the complex multiplication is done, including the setup steps (such as calculating the class groups, class invariants, and reduced class polynomials).

3.4.2 Complex

`Complex` provides Java with support for complex numbers. The basic Java API has no support for complex numbers at all. As the name of the method may suggest, complex multiplication requires a lot of arithmetic on complex numbers. To simplify these operations, the `Complex` class provides complex number support in the form of `Complex` objects with act as immutable complex numbers with all arithmetic built-in. `Complex` was designed to be completely reuseable in any situation where complex numbers are required.

3.4.3 ECMath

`ECMath` is a library (of static functions) that handles the few elliptic curve group operations needed for the generation of elliptic curves. These are used when looking for the generator of the subgroup of order r at the end of the generation process.

3.4.4 GFpEC

`GFpEC` is the basic "container" class for elliptic curves over $GF(p)$. All the class does is the absolute minimum, which is hold the necessary parameters for the curve. This can be changed easily later on if more functionality is desired, as the structure of how elliptic curves are going to be handled is beyond the scope of the generator.

3.4.5 GFpGenerator

`GFpGenerator` is the actual curve generator class. It is instantiated with parameters corresponding to the desired properties of the curve; the generator function can then be called as many times as necessary. Curves are returned as `GFpEC` objects, with no other output unless specifically requested (although output about the generation process is available in three different levels of description). This allows the generator to be reused in a wide range of applications, not just the standalone generator program for which it was written

The `GFpGenerator` class is where all the other classes (except for the front-end classes) come together and function as one unit. The other classes

were written as support for `GFpGenerator`; `GFpGenerator` is what appears to be doing (from an external standpoint) all of the work.

3.4.6 PolyMath

`PolyMath` is a library (of static functions) that handles all of the polynomial mathematics necessary for the generation of elliptic curves, used while calculating a complex multiplication discriminant's reduced class polynomial (Weber polynomial).

3.4.7 ecgenerate

The `ecgenerate` class acts as a simple front-end to the `GFpGenerator` class. As all of the generation routines can be called from a single function in the `GFpGenerator` class, and objects returned are of the `GFpEC` class, all the `ecgenerate` class needs to do is instantiate a `GFpGenerator` with given values, and output the values contained by the `GFpEC` class returned.

The `ecgenerate` class shows how easy it is to implement the actual generation routines in another program, even with no knowledge of how the underlying system works; only one line of code is needed to do the actual generation.

Chapter 4

Implementation

4.1 Implementation Overview

The implementation of the elliptic curve generator is the set of classes described in Section 3.4. The source code for these classes can be found in Appendix A1; documentation for the classes can be found in Appendix A2.

All of the classes but `ecgenerate` are in a separate package (called `mqp.shardy`); these classes can be used by placing them in the appropriate directory (usually `CLASSPATH/mqp/shardy`), and importing them in the usual method (`import mqp.shardy.*;`).

The classes are organized in the manner shown in Figure 4.1. An arrow from one class to another means that the class pointed to is required by the pointing class.

4.2 Difficuties in Implementation

For calculation of the Weber (reduced class) polynomial, very high precision for complex number arithmetic is needed. When the inner term polynomials are multiplied together, the imaginary terms of the complex numbers should equal zero, and the fractional parts of the complex numbers should also equal zero.

In the implementation, I used double precision floating point numbers to hold the values calculated; all of the intermediate calculations were also done at double precision. The `Complex` class uses double precision floating point numbers internally to store the real and imaginary terms of the complex

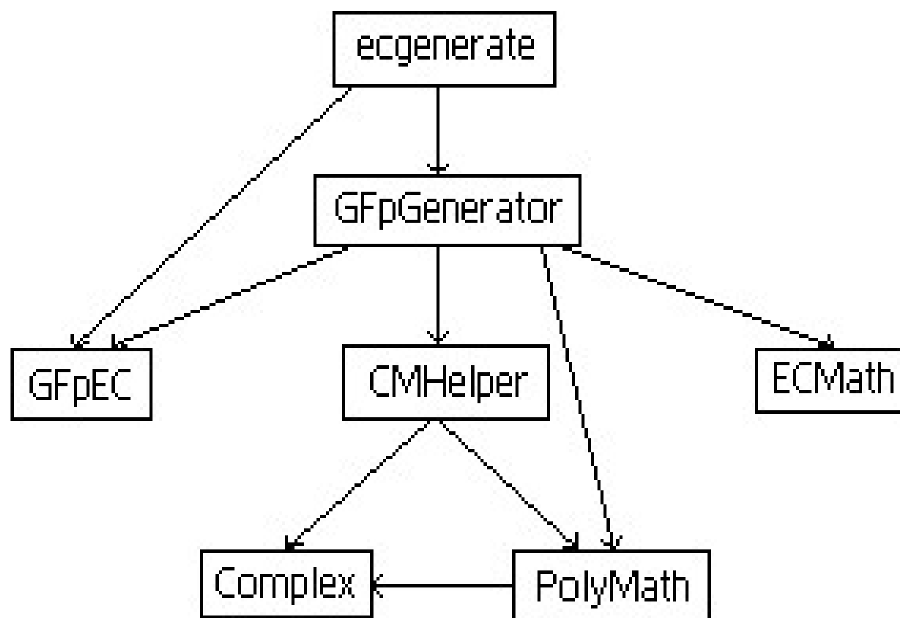

Figure 4.1: Organization of Generator Classes

number.

All parts of the implementation were independently tested for accuracy; the only part that did not consistently work properly was the calculation of the reduced class polynomials. Two types of errors would occur, during the calculations: overflow, when the terms of the polynomial were larger than can be stored in a double float; and rounding error, when inaccuracies due to precision propagated enough so that the end result would wrong to the wrong integer value (i.e., the error was greater than or equal to 0.5).

Calculation of the reduced class polynomial does not always fail, and often calculates the proper result. The major factor in whether it calculates the final result properly is the absolute value of the first class invariant. The first reduced symmetric matrix is always of the form

$$\begin{bmatrix} 1 & 0 \\ 0 & D \end{bmatrix}$$

where D is the complex multiplication discriminant. Due to the way the class invariant is calculated, will always result in a class invariant with no imaginary part. However, the real part can be very large, and with a large value, will cause rounding error or overflow to occur as multiplication by the individual polynomials progresses.

The errors in calculation of the reduced class polynomial do not throw off the curves generated by the generation program, however. It is easy to determine whether a reduced class polynomial is "good" or not; if the polynomial is not "good", that is, if it is not the correct reduced class polynomial for that complex multiplication discriminant, it will either not factor properly during the calculation of the curve coefficients, or will cause no generator to be founding for the discovered curve. This is because the curve generated, being incorrect, would not have complex multiplication by the given complex multiplication discriminant, and thus, would not have the same group order as calculated in the first part of the generation process. If either the factoring or finding the generator fails, then the reduced class polynomial is assumed to be incorrect, and the generation process is restarted.

Occasionally, there is no need for the reduced class polynomial to be calculated. [6] gives standard curve parameters for nine small values of D; if one of these is used, then the entire step involving the reduced class polynomial can be skipped.

4.3 Performance

All of the timings in this section were done on an Intel Pentium III processor running at 600 MHz, with 128 MB of RAM. Timings were obtained through use of the Unix `time` command. The operating system used was SuSE Linux (kernel 2.2.14). All times are in seconds. For a full listing of the timings of the generator trial runs, see Appendix A3.

Bit Size	Type	Minimum	Maximum	Average
40	CPU	5.3	23.9	8.0
	Real	13.1	31.9	15.9
160	CPU	5.7	292.1	131.8
	Real	13.5	303.0	141.0
320	CPU	66.4	966.2	538.2
	Real	74.7	990.0	551.6

Table 4.1: Performance of the Elliptic Curve Generator, in seconds

4.4 Performance Analysis

The large variation between some of the generation times is because of how the generation process will be restarted if the reduced class polynomial is not calculated properly, or if its calculation is not necessary. If the values of the curve parameters can be obtained by table lookup (as is the case for the nine values of D given in [6]), then the generation process will be much faster than if the reduced class polynomial needs to be calculated.

Chapter 5

Conclusions

5.1 Summary

Complex multiplication is a method of elliptic curve generation that works in the opposite manner of traditional curve generation: the curve order is generated first, and then the curve itself, so that the first curve generated will have the desired properties.

This software package implements elliptic curve generation using complex multiplication in the Java programming language. The package is intended for inclusion in a larger elliptic curve reference library, being created for the purpose of providing programmers and students a way to learn about and implement elliptic curves easily.

The elliptic curve generator program itself is only a front end to the generation package; designing the generator in this way allows for much greater reuseability. Anyone who wishes to generate elliptic curves in another program using this generator package simply needs to include the package, and call one function to do the actual generation. Everything else is transparent to the user.

5.2 Future Work

Changing the reduced class polynomial calculation method to use the `BigDecimal` class instead of double precision floating point numbers would prevent the rounding and overflowing errors that the current version of the generator has to deal with. However, the `BigDecimal` class is lacking a number of

basic mathematical functions that are needed for the manipulation of complex numbers, so poting it over would require considerably more work than changing variable types.

Creation of a `BigComplex` class that serves as an arbitrary precision equivalent to `Complex` would be the first step. `BigComplex` would be the "extension" of `Complex` in the same manner that `BigDecimal` is the extension to float, or `BigInteger` is the extension to `int`. The functions in `Complex` would need to be changed as to use greater precision equivalents; in some cases, functions in the `java.lang.Math` library would need to be rewritten for `BigDecimal` precision. Some examples of this are exponentiation and trigonometry functions.

Alternatively, if another multiprecision arithmetic library existed for Java, that could be used instead, but since one does not currently exist (that I have been able to find), "upgrading" `Complex` to `BigComplex`, and then implementing it in the reduced class polynomial calculation, would require adding a lot of missing functions to the `java.Math` libraries.

Appendix A1

Source Code

Contained here is the Java source for the following classes:

1. `CMHelper`
2. `Complex`
3. `ECMath`
4. `GFpEC`
5. `GFpGenerator`
6. `PolyMath`
7. `ecgenerate`

A1.1 CMHelper.java

```
package mqp.shardy;

import java.math.*;
import java.util.*;

/**
 * CMHelper provides all the ``helper'' functions for elliptic curve generation
 * using complex multiplication, including all operations involving reduced
 * symmetric matrices, and calculation of the reduced class polynomial (Weber
 * polynomial).
 *
 * @author  Seth Hardy
 * @version 1.0
 */

public class CMHelper {

  /**
   * The BigInteger constant 2.
   */

  public static final BigInteger BIGTWO   = new BigInteger("2");

  /**
   * The BigInteger constant 3.
   */

  public static final BigInteger BIGTHREE = new BigInteger("3");

  /**
   * The BigInteger constant 4.
   */

  public static final BigInteger BIGFOUR  = new BigInteger("4");

  /**
   * The BigInteger constant 5.
   */

  public static final BigInteger BIGFIVE  = new BigInteger("5");

  /**
   * The BigInteger constant 8.
   */

  public static final BigInteger BIGEIGHT = new BigInteger("8");

  private static final int CERTD = 30;   // Certainty for generated prime.
  private static final int MAXDIVS = 50; // Maximum number of prime divisors
  private static final double TOLERANCE = 1E-50; // Tolerance for calculations

  /**
   * Calculates the jacobi symbol (a/p) for the special case where p is prime.
   * <p>
   * Uses the method described in IEEE P1363 A.2.3.
   *
   * @param a       any integer
   * @param p       a prime
   * @return        the jacobi symbol (a/p)
   */

  public static int
  jacobip(BigInteger a, BigInteger p) {

    BigInteger ex;

    ex = p.subtract(BigInteger.ONE).divide(BIGTWO);
    a = a.modPow(ex,p);
    if ( a.add(BigInteger.ONE).mod(p).signum() == 0 )
      return -1;
    else return a.modPow(ex,p).intValue();
  }
```

```
/**
 * Calculates the square root of a BigInteger, as a BigDecimal. Uses Newton's
 * method for the calculation.
 *
 * @param x       number to take the square root of
 * @return        the square root of x
 */

public static BigDecimal
sqrt(BigInteger x) {

  final int ITERATIONS = 10000;

  BigDecimal n = new BigDecimal(x);
  BigDecimal k;
  BigDecimal next;

  n = n.setScale(100);
  k = new BigDecimal(BigInteger.ONE);
  for(int i=1;i<=ITERATIONS;i++) {
    next = n.divide(k,BigDecimal.ROUND_HALF_DOWN).add(k);
    k = next.divide(new BigDecimal("2"),BigDecimal.ROUND_HALF_DOWN);
  }
  return k;
}

/**
 * Calculates a specific term of the Lucas sequence for two given numbers
 * modulo an odd integer.
 * <p>
 * Uses the method described in IEEE P1363 A.2.4.
 *
 * @param n       odd integer for modulo reduction
 * @param P       first value to define Lucas sequence
 * @param Q       second value to define Lucas sequence
 * @param k       term of Lucas sequence
 * @return        term k of Lucas sequence modulo n
 */

public static BigInteger[]
lucas(BigInteger n, BigInteger P, BigInteger Q, BigInteger k) {

  BigInteger v0 = new BigInteger( BIGTWO.toByteArray() );
  BigInteger v1 = new BigInteger( P.toByteArray() );
  BigInteger q0 = new BigInteger( BigInteger.ONE.toByteArray() );
  BigInteger q1 = new BigInteger( BigInteger.ONE.toByteArray() );
  BigInteger f[]  = new BigInteger[2];
  int r = k.bitLength();

  for(int i=r;i>=0;i--) {
    q0 = q0.multiply(q1).mod(n);
    if( k.testBit(i) ) {
      q1 = q0.multiply(Q).mod(n);
      v0 = v0.multiply(v1).subtract( P.multiply(q0) ).mod(n);
      v1 = v1.multiply(v1).subtract( q1.multiply(BIGTWO) ).mod(n);
    }
    else {
      q1 = q0.multiply(BigInteger.ONE);
      v1 = v1.multiply(v0).subtract( P.multiply(q0) ).mod(n);
      v0 = v0.multiply(v0).subtract( q0.multiply(BIGTWO) ).mod(n);
    }
  }
  f[0] = v0;
  f[1] = q0;
  return(f);
}

/**
 * Performs modular exponentiation.
 *
 * @param a       base
 * @param b       exponent
 * @param p       modulus
 */

public static BigInteger
bigModPow(BigInteger a, BigInteger b, BigInteger p) {
```

```
  String s = b.toString(2);
  BigInteger c = a;

  for(int i=1;i<s.length();i++) {
    c = c.multiply(c);
    if( s.charAt(i) == '1' ) c = c.multiply(a);
    c = c.mod(p);
  }
  return c;
}

/**
 * Computes the square root of a number modulo p.
 * <p>
 * Uses the method described in IEEE P1363 A.2.5.
 *
 * @param g       number to take the modulo square root of
 * @param p       number modulo reduction is done by
 * @return        z such that z*z = g (mod p)
 */

public static BigInteger
sqrtmodp(BigInteger g, BigInteger p) {

  int pmod;
  BigInteger k;
  BigInteger w;
  BigInteger z;

  g = g.mod(p);
  if( p.mod(BIGFOUR).compareTo(BIGTHREE) == 0 ) {
    k = p.subtract(BIGTHREE).divide(BIGFOUR).add(BigInteger.ONE);
    z = g.modPow(k,p);
    w = z.modPow(BIGTWO,p);
    if( w.compareTo(g) == 0) return z;
      else return ( new BigInteger("-1") );
  }
  else {
    if( p.mod(BIGEIGHT).compareTo(BIGFIVE) == 0) {
      BigInteger y;
      BigInteger i;
      k = p.subtract(BIGFIVE).divide(BIGEIGHT);
  y = bigModPow(g.multiply(BIGTWO),k,p);
      i = y.pow(2).multiply(g).multiply(BIGTWO);
      i = i.subtract(BigInteger.ONE).mod(p);
      z = y.multiply(g).multiply(i).mod(p);
      w = z.modPow(BIGTWO,p);
      if( w.compareTo(g) == 0) return z;
        else return( new BigInteger("-1") );
    }
    else {
      BigInteger Q = g;
      BigInteger P = BigInteger.ZERO;
      BigInteger K;
      BigInteger[] vq = new BigInteger[2];
      K = p.add(BigInteger.ONE).divide(BIGTWO);
      do {
        P = P.add(BigInteger.ONE);
        vq = lucas(p,P,Q,K);
        z = vq[0].divide(BIGTWO);
        w = z.modPow(BIGTWO,p);
        if( w.compareTo(g) == 0) return z;
        else if( (vq[1].compareTo(BigInteger.ONE) == 1)
          && (vq[1].compareTo( p.subtract(BigInteger.ONE)) == -1) )
          return( new BigInteger("-1") );
       } while(P.compareTo(p) != 0);
    }
  }
  if( w.compareTo(g) == 0) return z;
  else return( new BigInteger("-1") );
}

/**
 * Calculates the class group H(D) for a squarefree determinant D.
 * <p>
 * Uses the method described in IEEE P1363 A.13.2.
```

```
 *
 * @param D       squarefree determinant
 * @return        set of all reduced symmetric matrices whose determinant = D
 */

public static Vector
classgroup(long D) {

  int count = 0;
  long s;
  long B;
  long db2;
  long C;
  long[] A = new long[MAXDIVS];
  long[] classg = new long[3];
  Vector group = new Vector();

  s = (long)Math.floor(Math.sqrt((double)D/3));
  for(B=0;B<=s;B++) {
    db2 = D + B*B;
    for(count=0;count<MAXDIVS;count++) {
      A[count] = 0;
    }
    count = 0;
    for(long x = 2*B;x<(Math.sqrt((double)db2));x++) {
      if( ( x != 0 ) && (db2 % x == 0) ) {
        A[count] = x;
        count++;
      }
    }
    for(int i=0;i<count;i++) {
      C = (long)(db2 / A[i]);

     if( gcd( gcd(A[i],2*B), C ) == 1 ) {

        classg[0] = A[i];
        classg[1] = B;
        classg[2] = C;
        group.addElement((long[])classg.clone());
        if( (2*B > 0) && (2*B < A[i]) && (A[i] < C)) {
          classg[1] = -B;
          group.addElement((long[])classg.clone());
        }
      }
    }
  }
  return(group);
}

/**
 * Calculates the class number h(D) for a squarefree determinant D.
 * <p>
 * Uses the method described in IEEE P1363 A.13.2.
 *
 * @param D       squarefree determinant
 * @return        count of reduced symmetric matrices whose determinant = D
 */

public static int
classnumber(long D) {

  int count = 0;
  int number = 0;
  long db2;
  long C;
  long s;
  long B;
  long[] A = new long[MAXDIVS];
  long[] classg = new long[3];

  s = (long)Math.floor(Math.sqrt((double)D/3));
  for(B=0;B<=s;B++) {
    db2 = D + B*B;
    for(count=0;count<MAXDIVS;count++) {
      A[count] = 0;
    }
    count = 0;
```

```
    for(long x = 2*B;x<(Math.sqrt((double)db2));x++) {
      if( ( x != 0 ) && (db2 % x == 0) ) {
        A[count] = x;
        count++;
      }
    }
    for(int i=0;i<count;i++) {
      C = (long)(db2 / A[i]);
     if( gcd( gcd(A[i],2*B), C ) == 1 ) {
        number++;
        if( (2*B > 0) && (2*B < A[i]) && (A[i] < C)) {
          number++;
        }
      }
    }
  }
  return(number);
}

/**
 * Calculates the class invariant for a reduced symmetric matrix. The form
 * for the reduced symmmetric matrix is M = [[A,B],[B,C]], where gcd(A,2B,C) = 1;
 * abs(2B) <= A <= C; and if either A = abs(2B) or A = C, then B >= 0.
 * <p>
 * Uses the method described in IEEE P1363 A.13.3.
 *
 * @param A       term A of the reduced symmetric matrix [[A,B],[B,C]]
 * @param B       term B of the reduced symmetric matrix [[A,B],[B,C]]
 * @param C       term C of the reduced symmetric matrix [[A,B],[B,C]]
 * @return        the class invariant of [A,B,C]
 */

public static Complex
classinv(long A, long B, long C) {

  int J;
  int M;
  int N;
  int d8;
  long D = A*C-B*B;
  long G = gcd(3,D);
  long I;
  long K;
  long L;
  Complex lambda;
  Complex ci;

  d8 = (int)(D % 8);

  if( (d8 == 1) || (d8 == 2) || (d8 == 6) ) {
    I = 3;
    K = 2;
  }
  else if(d8 == 7) {
    I = 3;
    K = 1;
  }
  else if(d8 == 3) {
    K = 1;
    if(D % 3 == 0) I = 2;
    else I = 0;
  }
  else {
    I = 6;
    K = 4;
  }
  if( (A*C) % 2 == 1 ) J = 0;
  else if( C % 2 == 0 ) J = 1;
  else J = 2;
  if( (J == 0) || ( (d8 == 5) && (J == 1) ) )
    L = A-C+A*A*C;
  else if( ((d8 == 1) || (d8 == 2) || (d8 == 3) || (d8 == 6) || (d8 == 7)) && (J == 1) )
    L = A+2*C-A*C*C;
  else if( (d8 == 3) && (J == 2) )
    L = A-C+5*A*C*C;
  else L = A-C-A*C*C;
  if(J == 2) M = (int)Math.pow((double)-1,(double)((C*C-1)/8));
```

```
  else M = (int)Math.pow((double)-1,(double)((A*A-1)/8));
  if( (d8 == 5) || ((d8 == 3) && (J == 0)) || ((d8 == 7) && (J != 0)) )
    N = 1;
  else if( (d8 == 1) || (d8 == 2) || (d8 == 6) ) N = M;
  else if( (d8 == 7) && (J == 0) ) N = M;
  else N = -M;
  lambda = new Complex(0,Math.PI*K/24);
  lambda = lambda.eRaisedTo();
  ci = lambda.pow(-B*L);
  ci = ci.inverse().multiply(N).multiply(Math.pow((double)2,(double)-I/6));
  ci = ci.multiply( fn(J,A,B,C).pow(K) ).pow(G);
  return ci;
}

/**
 * Calculates the Weber polynomial (reduced class polynomial) for a squarefree
 * determinant D.
 * <p>
 * Uses the method described in IEEE P1363 A.13.3.
 *
 * @param D       squarefree determinant
 * @return        reduced class polynomial for D
 */

public static BigInteger[]
rcpolynom(long D) {

  long[] cg = new long[3];
  Vector group = classgroup(D);
  Complex[] x = new Complex[PolyMath.MAXSIZE];
  Complex[] y = new Complex[PolyMath.MAXSIZE];
  Complex[] xx = new Complex[PolyMath.DOUBLESIZE];
  BigInteger[] z = new BigInteger[PolyMath.MAXSIZE];

  for(int i=0;i<PolyMath.MAXSIZE;i++) {
    x[i] = new Complex(0,0);
    y[i] = new Complex(0,0);
  }
  y[0] = new Complex(1,0);
  for(int i=0;i<group.size();i++) {
    cg = (long[])group.elementAt(i);
    x[0] = classinv(cg[0],cg[1],cg[2]).neg();
    x[1] = new Complex(1,0);
    xx = PolyMath.polyMult(x,y);
    for(int j=0;j<PolyMath.MAXSIZE;j++) {
      y[j] = xx[j];
      y[j] = xx[j];
    }
  }

  for(int i=0;i<PolyMath.MAXSIZE;i++)
    z[i] = new BigInteger(Long.toString(Math.round(y[i].getReal())));

  return z;
}

/**
 * Determines whether a number is a ``near prime'', and if so, returns the smooth
 * integer and prime that, when multiplied, give the original number.
 * <p>
 * Uses the method described in IEEE P1363 A.15.5.
 *
 * @param u       number to check for near-primeness
 * @param lmax    upper bound on trial division for smooth integer
 * @param rmin    lower bound for prime divisor
 * @param rmax    upper bound for prime divisor
 * @return        smooth integer and prime, or [0,0] if not near prime
 */

public static BigInteger[]
nearprime(BigInteger u, int lmax, BigInteger rmin, BigInteger rmax) {

  BigInteger ltemp = new BigInteger("1");
  BigInteger lmodr;
  BigInteger[] r = new BigInteger[2];

  r[0] = u;
```

```
  r[1] = BigInteger.ONE;
  for(int l = 2;l<=lmax;l++) {
    ltemp = ltemp.add(BigInteger.ONE);
    if( ltemp.isProbablePrime(CERTD) ) {
      lmodr = r[0].mod(ltemp);
      if( lmodr.compareTo(BigInteger.ZERO) == 0) {
        r[0] = r[0].divide(ltemp);
        r[1] = r[1].multiply(ltemp);
        if(r[0].compareTo(rmin) == -1) {
          r[0] = BigInteger.ZERO;
          r[1] = BigInteger.ZERO;
          return r;
        }
      }
    }
  }
  if( r[0].compareTo(rmax) == 1 ) {
    r[0] = BigInteger.ZERO;
    r[1] = BigInteger.ZERO;
    return r;
  }
  if( r[0].isProbablePrime(CERTD) ) return r;
  else {
    r[0] = BigInteger.ZERO;
    r[1] = BigInteger.ZERO;
    return r;
  }
}

/**
 * Extended Euclidean greatest common divisor algorithm.
 * <p>
 * Uses the method described in IEEE P1363 A.2.2.
 *
 * @param m       value 1 for gcd
 * @param h       value 2 for gcd
 * @return        the greatest common divisor of m and h
 */

public static long
gcd(long m, long h) {

  long r0, r1, r2;
  long q;

  if( (h == 1) || (m == 1) ) return 1;
  if( (h == 0) || (m == 0) ) return 1;
  else {
    r0 = m;
    r1 = h % m;
    while(r1 > 0) {
      q = (long)Math.floor((double)r0/r1);
      r2 = (r0 - q * r1) % m;
      r0 = r1;
      r1 = r2;
    }
    return r0;
  }
}

/**
 * Returns the next squarefree determinant that meets given congruence conditions.
 * <p>
 * Uses method described in IEEE P1363 A.14.2.1.
 *
 * @param k       congruence value based on p of GF(p) and minimum allowed prime
 *            for the order of the subgroup
 * @param pModEight   congruence value = p of GF(p) modulo 8
 * @param currD   current squarefree determinant
 * @return        next squarefree determinant that meets the congruence
 *            conditions
 */

public static long
getNextD(int k, int pModEight, long currD) {
  switch(pModEight) {
  case 1:
```

```
    if(k == 1) {
      do { currD++; }
      while ( !(squarefree(currD)) || (currD % 8 != 3) );
    }
    else if(k == 2 || k == 3) {
      do { currD++; }
      while ( !(squarefree(currD)) || (currD % 8 == 7) );
    }
    else {
      do { currD++; }
      while ( !(squarefree(currD)) );
    }
    break;
  case 3:
    if(k == 1) {
      do { currD++; }
      while ( !(squarefree(currD)) || (currD % 8 != 3) );
    }
    else if(k == 2 || k == 3) {
      do { currD++; }
      while ( !(squarefree(currD)) || ( (currD % 8 == 7)
              && (currD % 8 != 2) && (currD % 8 != 3) ) );
    }
    else {
      do { currD++; }
      while ( !(squarefree(currD)) || ( (currD % 8 != 2)
              && (currD % 8 != 3) && (currD % 8 != 7) ) );
     }
     break;
  case 5:
    if(k == 1) {
      do { currD++; }
      while ( !(squarefree(currD)) || (currD % 8 != 3) );
    }
    else if(k == 2 || k == 3) {
      do { currD++; }
      while ( !(squarefree(currD)) || ( (currD % 8 == 7)
              && (currD % 2 != 1) ) );
    }
    else {
       do { currD++; }
       while ( !(squarefree(currD)) || ( (currD % 2 != 1) ) );
    }
    break;
  case 7:
    if(k == 1) {
      do { currD++; }
      while ( !(squarefree(currD)) || (currD % 8 != 3) );
    }
    else if(k == 2 || k == 3) {
      do { currD++; }
      while ( !(squarefree(currD)) || ( (currD % 8 == 7)
              && (currD % 8 != 3) && (currD % 8 != 6) ) );
    }
    else {
      do { currD++; }
      while ( !(squarefree(currD)) || ( (currD % 8 != 3)
              && (currD % 8 != 6) && (currD % 8 != 7) ) );
    }
    break;
  }
  return currD;
}

/**
 * Tests whether a number is squarefree.
 *
 * @param d       number for squarefree test
 * @return        <code>true</code> if squarefree, <code>false</code> if not.
 */

public static boolean
squarefree(long d) {

  long current = 2;
  boolean sqfree = true;
```

```
  while( sqfree && (current*current <= d) ) {
    if(d % (current*current) == 0) sqfree = false;
    current++;
  }
  return sqfree;
}

/**
 * Determines the prime divisors of a number.
 *
 * @param d       number to determine prime divisors of
 * @return        array of prime divisors of d
 */

public static long[]
primedivs(long d) {

  int i;
  long current = 3;
  long[] divs = new long[MAXDIVS];

  for(i=0;i<MAXDIVS;i++) { divs[i] = 0; }
  i = 0;
  while( current <= d/2 ) {
    if( (d % current == 0) &&
        (new BigInteger(Long.toString(current)).isProbablePrime(CERTD))) {
      divs[i] = current;
      i++;
    }
    current++;
  }
  if( (new BigInteger(Long.toString(d)).isProbablePrime(CERTD)) && (d != 2) )
    divs[i] = d;
  return divs;
}

/**
 * Solves the diophantine equation 4p = W^2 + DV^2 for a given complex
 * multiplication discriminant.
 * <p>
 * Uses method described in IEEE P1363 A.14.2.2.
 *
 * @param p       p of GF(p)
 * @param D       complex multiplication discriminant
 * @return        [W,0] if solution found to diophantine equation; [W,1]
 *            if solution found and D = 1 or 3; [0,0] if solution
 *            to diophantine equation not found (D not a complex
 *            multiplication discriminant).
 */

public static BigInteger[]
getWV(BigInteger p, long D) {

  BigInteger bigdee = new BigInteger( Long.toString(D) );
  BigInteger B2;
  BigInteger delta;
  BigInteger[] S = new BigInteger[4];
  BigInteger[] St = new BigInteger[4];
  BigInteger[] U = new BigInteger[2];
  BigInteger[] Ut = new BigInteger[2];
  BigDecimal ddelta;

  S[1] = CMHelper.sqrtmodp( new BigInteger( Long.toString(-D) ).mod(p), p);
  S[0] = p;
  S[3] = S[1].multiply(S[1]).add(bigdee).divide(p);
  S[2] = S[1];

  if( S[1].signum() == -1 ) {
    U[0] = BigInteger.ZERO;
    U[1] = BigInteger.ZERO;
    return U;
  }

  U[0] = BigInteger.ONE;
  U[1] = BigInteger.ZERO;

  B2 = S[1].multiply(CMHelper.BIGTWO);
```

```
  B2 = B2.abs();

  while( (B2.compareTo(S[0]) == 1) || (S[0].compareTo(S[3]) == 1) ) {
    if( (D == 11) &&
        (S[0].compareTo(CMHelper.BIGTHREE) == 0) )
      delta = BigInteger.ZERO;
    else {
      ddelta = new BigDecimal(S[1]).divide(new BigDecimal(S[3]),BigDecimal.ROUND_HALF_UP);
      ddelta = ddelta.add(new BigDecimal("0.5"));
      if( ddelta.signum() == -1) delta = ddelta.toBigInteger().subtract(BigInteger.ONE);
      else delta = ddelta.toBigInteger();
    }

    Ut[0] = delta.multiply(U[0]).add(U[1]);
    Ut[1] = U[0].negate();

    U = (BigInteger[])Ut.clone();

    St[0] = new BigInteger( S[3].toByteArray() );
    St[1] = S[1].negate().add( delta.multiply(S[3]) );
    St[2] = St[1];
    St[3] = S[0].subtract( CMHelper.BIGTWO.multiply(delta).multiply(S[1]) );
    St[3] = St[3].add( delta.multiply(delta).multiply(S[3]) );

    S = (BigInteger[])St.clone();

    B2 = S[1].multiply(CMHelper.BIGTWO);
    B2 = B2.abs();

  }

  if( (D==1) || (D==3) ) {
    U[0] = U[0].multiply(CMHelper.BIGTWO);
    U[1] = U[1].multiply(CMHelper.BIGTWO);
    return U;
  }
  if( S[0].compareTo(BigInteger.ONE) == 0 ) {
    U[0] = U[0].multiply(CMHelper.BIGTWO);
    U[1] = BigInteger.ZERO;
    return U;
  }
  if( S[0].compareTo(CMHelper.BIGFOUR) == 0 ) {
    U[0] = U[0].multiply(CMHelper.BIGFOUR).add(S[1].multiply(U[1]));
    U[1] = BigInteger.ZERO;
    return U;
  }
  U[0] = BigInteger.ZERO;
  U[1] = BigInteger.ZERO;
  return U;
}

/**
 * Calculates F, the sum used in calculating the class invariant of a reduced
 * symmetric matrix.
 * <p>
 * Uses the method described in IEEE P1363 A.13.3.
 *
 * @param z       value of argument
 * @return        F(z) = 1 + sum from j to infinity of
 *            ((-1)^j)*(z^((3j^2-j)/2) + z^((3j^2+j)/2))
 */

public static Complex
F(Complex z) {
  Complex Fz = new Complex(1,0);
  Complex zj;
  for(int j=1;;j++) {
    zj = z.pow( (int)((3*j*j-j)/2) );
    if( (zj.real < TOLERANCE) && (zj.imag < TOLERANCE) ) break;
    else {
      Fz = Fz.add(zj);
      zj = z.pow( (int)((3*j*j+j)/2) );
      Fz = Fz.add(zj);
    }
  }
  return Fz;
}
```

```
  /**
   * Calculates theta, used in calculating the class invariant of a reduced
   * symmetric matrix.
   * <p>
   * Uses the method described in IEEE P1363 A.13.3.
   *
   * @param A       term A of the reduced symmetric matrix [[A,B],[B,C]]
   * @param B       term B of the reduced symmetric matrix [[A,B],[B,C]]
   * @param C       term C of the reduced symmetric matrix [[A,B],[B,C]]
   * @return        theta = e^(((-sqrt(D)*Bi)*Pi/A)
   */

  public static Complex
  theta(long A, long B, long C) {
    long D = A*C - B*B;
    Complex e = new Complex(-Math.sqrt(D)*Math.PI/A, B/A*Math.PI);
    return e.eRaisedTo();
  }

  /**
   * Calculates the f_n function, used in calculating the class invariant of
   * a reduced symmetric matrix.
   * <p>
   * Uses the method described in IEEE P1363 A.13.3.
   *
   * @param n       number of f-function to use, 0 <= f <= 2
   * @param A       term A of the reduced symmetric matrix [[A,B],[B,C]]
   * @param B       term B of the reduced symmetric matrix [[A,B],[B,C]]
   * @param C       term C of the reduced symmetric matrix [[A,B],[B,C]]
   * @return        f_n(A,B,C)
   */

  public static Complex
  fn(int n, long A, long B, long C) {
    Complex t = theta(A,B,C);
    Complex s;
    if(n == 2) s = ( t.root(12,0) ).multiply( new Complex( Math.sqrt(2), 0) );
    else s = ( t.root(24,0) ).inverse();
    switch(n) {
    case 0: s = s.multiply( F( t.neg() ) );
            s = s.divide( F( t.pow(2) ) );
            break;
    case 1: s = s.multiply( F(t) );
            s = s.divide( F( t.pow(2) ) );
            break;
    case 2: s = s.multiply( F( t.pow(4) ) );
            s = s.divide( F( t.pow(2) ) );
            break;
    default: break;
    }
    return s;
  }

}
```

A1.2 Complex.java

```
package mqp.shardy;

/**
 * Complex provides Java with support for complex numbers. Complex numbers are
 * implemented as immutable objects with double floating point precision for
 * both real and imaginary parts.
 *
 * @author  Seth Hardy
 * @version 1.0
 */

public class Complex {

  final static double TOLERANCE   = 1E-50;

  /**
   * Creates a new complex number, and initializes both real and imaginary
   * parts to zero.
   *
   */

  public Complex() {
    this.real = 0;
    this.imag = 0;
  }

  /**
   * Creates a new complex number with real part r and imaginary part i, each
   * from double floating point precision.
   *
   * @param r       the real part of the complex number
   * @param i       the imaginary part of the complex number
   */

  public Complex(double r, double i) {
    this.real = r;
    this.imag = i;
  }

  /**
   * Creates a new complex number with real part r and imaginary part i, each
   * from long integer precision. The long integers are cast into double
   * floats, so there is no loss of precision.
   *
   * @param r       the real part of the complex number
   * @param i       the imaginary part of the complex number
   */

  public Complex(long r, long i) {
    this.real = (double)r;
    this.imag = (double)i;
  }

  /**
   * Returns the real part of the complex number, to double floating point
   * precision.
   *
   * @return        the real part of this
   */

  public double
  getReal() {
    return this.real;
  }

  /**
   * Returns the imaginary part of the complex number, to double floating point
   * precision.
   *
   * @return        the imaginary part of this
   */

  public double
```

```
getImag() {
  return this.imag;
}

/**
 * Complex number addition: adds the complex number a to this.
 *
 * @param a       value a to be added to this
 * @return        this + a
 */

public Complex
add(Complex a) {
  return new Complex(this.real + a.real, this.imag + a.imag);
}

/**
 * Complex number subtraction: subtracts the complex number a from this.
 *
 * @param a       value a to be subtracted from this
 * @return        this - a
 */

public Complex
subtract(Complex a) {
  return new Complex(this.real - a.real, this.imag - a.imag);
}

/**
 * Complex number multiplication: multiplies this by a.
 *
 * @param a       value a to be multiplied by this
 * @return        this * a
 */

public Complex
multiply(Complex a) {
  return new Complex(this.real*a.real-this.imag*a.imag,
                     this.imag*a.real+this.real*a.imag);
}

/**
 * Complex number multiplication by a scalar: multiplies this by a.
 *
 * @param a       scalar value a to be multiplied by this
 * @return        this * a
 */

public Complex
multiply(double a) {
  return new Complex(this.real*a, this.imag*a);
}

/**
 * Complex number division: divides this by a.
 *
 * @param a       scalar value a to be divided by this
 * @return        this / a
 */

public Complex
divide(Complex a) {
  double r,i;
  r = (this.real*a.real+this.imag*a.imag) / (a.real*a.real+a.imag*a.imag);
  i = (this.imag*a.real-this.real*a.imag) / (a.real*a.real+a.imag*a.imag);
  return new Complex(r,i);
}

/**
 * Complex number division by a scalar: divides this by a.
 *
 * @param a       scalar value a to be divided by this
 * @return        this / a
 */

public Complex
divide(double a) {
```

```
  return new Complex(this.real/a, this.imag/a);
}

/**
 * Negates a complex number.
 *
 * @return        -this
 */

public Complex
neg() {
  return new Complex(-this.real, -this.imag);
}

/**
 * Squares a complex number.
 *
 * @return        this^2
 */


public Complex
square() {
  return new Complex(this.real*this.real-this.imag*this.imag,
                     2*this.real*this.imag);
}

/**
 * Complex number exponentiation: raises this to a scalar power.
 *
 * @param a       the exponent to raise this by
 * @return        this^a
 */

public Complex
pow(long a) {
  if(a == 0) return new Complex(1,0);
  String s = Integer.toBinaryString((int)Math.abs(a));
  Complex x = this;
  for(int i=1;i<s.length();i++) {
    x = x.square();
    if( s.charAt(i) == '1' ) x = x.multiply(this);
  }
  if(a < 0) return x.inverse();
  else return x;
}

/**
 * Complex number inversion.
 *
 * @return        1 / this
 */

public Complex
inverse() {
  double r = Math.sqrt(this.real*this.real+this.imag*this.imag);
  double theta = Math.atan2(this.imag,this.real);
  return new Complex(1/r*Math.cos(-theta), 1/r*Math.sin(-theta));
}

/**
 * Takes the n-th root of a complex number.
 *
 * @param n       the n-th root to return
 * @param j       of the n possible roots, return root j
 * @return        root number j that satisfies this^(1/n)
 */

public Complex
root(int n, int j) {
  double r = Math.sqrt(this.real*this.real+this.imag*this.imag);
  double theta = Math.atan(this.imag/this.real);
  double t = (theta + 2*Math.PI*j)/n;
  double rx = Math.pow(r,(double)1/n);
  return new Complex(rx*Math.cos(t),rx*Math.sin(t));
}
```

```
  /**
   * Compares two complex numbers. The number with the greater real part
   * is considered larger. In the case that the real parts are equal, the
   * part with the greater imaginary part is considered larger. If both
   * real parts and both imaginary parts are equal, then the two complex
   * numbers are considered equal.
   *
   * @param a       the complex number to compare this to
   * @return        -1 if this < a; 1 if this > a; 0 if equal
   */

  public int
  compareTo(Complex a) {
     if(this.real == a.real) {
       if(this.imag == a.imag) return 0;
       else if(this.imag > a.imag) return 1;
         else return -1;
     }
     else if(this.real < a.real) return -1;
       else return 1;
  }

  /**
   * Raises e to the this power. Uses Euler's identity: e^ix = cos x + i sin x.
   *
   * @return        e^this
   */

  public Complex
  eRaisedTo() {
    return new Complex(Math.cos(this.imag),Math.sin(this.imag)).multiply(Math.pow(Math.E,this.real));
  }

  /**
   * Converts a complex number to a string in the format (real,imag i).
   *
   * @return        string version of this
   */

  public String
  toString() {
    return "(" + this.real + "," + this.imag + "i)";
  }


  double real;  // the real part of the complex number
  double imag;  // the imaginary part of the complex number

}
```

A1.3 ECMath.java

```
package mqp.shardy;

import java.math.*;
import java.security.*;

/**
 * ECMath provides functions for elliptic curve arithmetic for curves over
 * GF(p). Points on the elliptic curves are represented as arrays of two
 * BigIntegers, as this library is not intended to provide an abstract data
 * type to represent points on a curve.
 *
 * @author  Seth Hardy
 * @version 1.0
 */

public class ECMath {

  /**
   * Adds two points on an elliptic curve over GF(p).
   * <p>
   * Uses the normal method for adding two points on an elliptic curve, which
   * can be found in IEEE P1363 section A.10.1.
   *
   * @param P   point on the curve over GF(p)
   * @param Q   point on the curve over GF(p)
   * @param a   curve parameter a such that y^2 = x^3 + ax + b
   * @param b   curve parameter b such that y^2 = x^3 + ax + b
   * @param p   prime p for GF(p)
   * @return    the point R = P + Q
   */

  public static BigInteger[]
  ecadd(BigInteger[] P, BigInteger[] Q, BigInteger a, BigInteger b,
        BigInteger p) {
    BigInteger[] R = new BigInteger[2];
    BigInteger[] infinity = new BigInteger[2];
    BigInteger lambda;
    BigInteger bottom;
    infinity[0] = BigInteger.ZERO;
    if( b.compareTo(BigInteger.ZERO) == 0)
      infinity[1] = BigInteger.ONE;
    else infinity[1] = BigInteger.ZERO;
    if( (P[0].compareTo(infinity[0]) == 0) &&
        (P[1].compareTo(infinity[1]) == 0)) {
      return Q;
    }
    else if( (Q[0].compareTo(infinity[0]) == 0) &&
             (Q[1].compareTo(infinity[1]) == 0)) {
      return P;
    }
    if( P[0].compareTo(Q[0]) != 0) {
      lambda = P[1].subtract(Q[1]);
      bottom = P[0].subtract(Q[0]).mod(p);
      lambda = lambda.multiply(bottom.modInverse(p));
      R[0] = lambda.multiply(lambda).subtract(P[0]).subtract(Q[0]).mod(p);
      R[1] = lambda.multiply(Q[0].subtract(R[0])).subtract(Q[1]).mod(p);
      return R;
    }
    else if(P[1].compareTo(Q[1]) != 0) {
      return infinity;
    }
    else if(Q[1].compareTo(BigInteger.ZERO) == 0) {
      return infinity;
    }
    else {
      lambda = Q[0].multiply(Q[0]).multiply(CMHelper.BIGTHREE).add(a);
      bottom = Q[1].multiply(CMHelper.BIGTWO);
      lambda = lambda.multiply( bottom.modInverse(p) );
      R[0] = lambda.multiply(lambda).subtract(P[0]).subtract(Q[0]).mod(p);
      R[1] = lambda.multiply(Q[0].subtract(R[0])).subtract(Q[1]).mod(p);
      return R;
    }
  }
```

```
/**
 * Calculates the inverse of a point on an elliptic curve over GF(p), by
 * the identity -(x,y) = (x,-y).
 *
 * @param P   point on the curve over GF(p)
 * @param p   prime p for GF(p)
 * @return    the point -P
 */

public static BigInteger[]
ecneg(BigInteger[] P, BigInteger p) {
  P[1] = P[1].negate();
  P[1] = P[1].mod(p);
  return P;
}

/**
 * Multiplies a point on an elliptic curve over GF(p) by a scalar value.
 * <p>
 * Uses the double and add method of scalar multiplication, which can be
 * found in IEEE P1363 section A.10.3.
 *
 * @param P   point on the curve over GF(p)
 * @param n   scalar value to multiply P by
 * @param a   curve parameter a such that y^2 = x^3 + ax + b
 * @param b   curve parameter b such that y^2 = x^3 + ax + b
 * @param p   prime p for GF(p)
 * @return    the point R = nP
 */

public static BigInteger[]
ecmult(BigInteger P[], BigInteger n, BigInteger a, BigInteger b, BigInteger p) {
  String s = n.toString(2);
  BigInteger[] infinity = new BigInteger[2];
  BigInteger[] x = new BigInteger[2];
  infinity[0] = BigInteger.ZERO;
  if( b.compareTo(BigInteger.ZERO) == 0)
    infinity[1] = BigInteger.ONE;
  else infinity[1] = BigInteger.ZERO;
  x[0] = P[0];
  x[1] = P[1];
  if( n.compareTo(BigInteger.ZERO) == 0) return infinity;
  for(int i=1;i<s.length();i++) {
    x = ecadd(x,x,a,b,p);
    if(s.charAt(i) == '1') x = ecadd(x,P,a,b,p);
  }
  return x;
}

/**
 * Generates a random point on an elliptic curve over GF(p).
 * <p>
 * Generates x values at random, and then if there exist points (x,y) and
 * (x,-y) on the curve, returns one of the two at random. This method can
 * be found in section A.11.1 of IEEE P1363.
 *
 * @param a   curve parameter a such that y^2 = x^3 + ax + b
 * @param b   curve parameter b such that y^2 = x^3 + ax + b
 * @param p   p for GF(p)
 * @return    a random point on the elliptic curve
 */

public static BigInteger[]
getRandPoint(BigInteger a, BigInteger b, BigInteger p) {
  SecureRandom rnd = new SecureRandom();
  BigInteger[] xy = new BigInteger[2];
  BigInteger alpha;
  do {
    xy[0] = new BigInteger(p.bitLength(),rnd);
    alpha = xy[0].pow(3);
    alpha = alpha.add(a.multiply(xy[0])).add(b).mod(p);
    if(alpha.compareTo(BigInteger.ZERO) == 0) {
  xy[1] = BigInteger.ZERO;
  return xy;
    }
    xy[1] = mqp.shardy.CMHelper.sqrtmodp(alpha,p);
```

```
    } while( xy[1].compareTo( new BigInteger("-1") ) == 0 );
    if(rnd.nextBoolean() == true) {
      xy[1] = xy[1].negate().mod(p);
    }
    return xy;
  }

  /**
   * Finds a generator of a subgroup on an elliptic curve over GF(p).
   * <p>
   * The algorithm can be found in IEEE P1363 section A.11.3.
   * @param r   order of the prime subgroup
   * @param k   cofactor such that k*r = #E
   * @param a   curve parameter such that y^2 = x^3 + ax + b
   * @param b   curve parameter such that y^2 = x^3 + ax + b
   * @param p   p for GF(p)
   * @return    a generator for the subgroup of order r, or the point at
   *        infinity if the wrong order is given.
   */

  public static BigInteger[]
  genG(BigInteger r, int k, BigInteger a, BigInteger b, BigInteger p) {
    BigInteger[] P = new BigInteger[2];
    BigInteger[] G = new BigInteger[2];
    BigInteger[] infinity = new BigInteger[2];
    infinity[0] = BigInteger.ZERO;
    if( b.compareTo(BigInteger.ZERO) == 0)
      infinity[1] = BigInteger.ONE;
    else infinity[1] = BigInteger.ZERO;
    do {
      P = getRandPoint(a,b,p);
      G = ecmult(P,new BigInteger( Integer.toString(k) ),a,b,p);
    } while( (G[0].compareTo(infinity[0]) == 0) &&
             (G[1].compareTo(infinity[1]) == 0) );
    P = ecmult(G,r,a,b,p);
    if( (P[0].compareTo(infinity[0]) == 0) &&
        (P[1].compareTo(infinity[1]) == 0) )
      return G;
    else return infinity;
  }

}
```

A1.4 GFpEC.java

```
package mqp.shardy;

import java.math.*;

/**
 * GFpEC is a simple class for the purpose of holding the parameters of an
 * elliptic curve over GF(p).
 *
 * @author  Seth Hardy
 * @version 1.0
 */

public class GFpEC {

  /**
   * Creates a new GFpEC object with the given parameters for the curve,
   * not using the complex multiplication discriminant D. This method should be
   * used when storing a curve not generated by the method of complex
   * multiplication.
   *
   * @param p       the prime number for GF(p)
   * @param r       the prime order r of the subgroup generated by G
   * @param k       the cofactor such that #E = k * r
   * @param x       the x-coordinate of the generator point G
   * @param y       the y-coordinate of the generator point G
   * @param a0      the curve parameter a such that y^2 = x^3 + ax + b
   * @param b0      the curve parameter b such that y^2 = x^3 + ax + b
   */

  public GFpEC(BigInteger p, BigInteger r, BigInteger k, BigInteger x,
        BigInteger y, BigInteger a0, BigInteger b0) {
    this.modulus  = p;
    this.ordG     = r;
    this.cofactor = k;
    this.ordE     = this.ordG.multiply(this.cofactor);
    this.Gx       = x;
    this.Gy       = y;
    this.a        = a0;
    this.b        = b0;
    this.D        = 0;
  }

  /**
   * Creates a new GFpEC object with the given parameters for the curve, also
   * storing the complex multiplication discriminant D. This method should be
   * used when storing a curve generated by the method of complex
   * multiplication.
   *
   * @param p       the prime number for GF(p)
   * @param r       the prime order r of the subgroup generated by G
   * @param k       the cofactor such that #E = k * r
   * @param x       the x-coordinate of the generator point G
   * @param y       the y-coordinate of the generator point G
   * @param a0      the curve parameter a such that y^2 = x^3 + ax + b
   * @param b0      the curve parameter b such that y^2 = x^3 + ax + b
   * @param dee     the complex multiplication discriminant D
   */

  public GFpEC(BigInteger p, BigInteger r, BigInteger k, BigInteger x,
        BigInteger y, BigInteger a0, BigInteger b0, long dee) {
    this.modulus  = p;
    this.ordG     = r;
    this.cofactor = k;
    this.ordE     = this.ordG.multiply(this.cofactor);
    this.Gx       = x;
    this.Gy       = y;
    this.a        = a0;
    this.b        = b0;
    this.D        = dee;
  }

  /**
   * Returns the modulus of the underlying field of the elliptic curve.
```

```
 *
 * @return        modulus p for GF(p)
 */

public BigInteger getP()  { return this.modulus; }

/**
 * Returns the prime order of the large subgroup of points on the elliptic
 * curve.
 *
 * @return        prime order r
 */

public BigInteger getR()  { return this.ordG; }

/**
 * Returns the cofactor of the order of the elliptic curve.
 *
 * @return        cofactor k
 */

public BigInteger getK()  { return this.cofactor; }

/**
 * Returns the order of the elliptic curve.
 *
 * @return        order #E
 */

public BigInteger getU()  { return this.ordE; }

/**
 * Returns the x-coordinate of the generator of the large subgroup of prime
 * order on the elliptic curve.
 *
 * @return        x-coordinate of G
 */

public BigInteger getGx() { return Gx; }

/**
 * Returns the y-coordinate of the generator of the large subgroup of prime
 * order on the elliptic curve.
 *
 * @return        y-coordinate of G
 */

public BigInteger getGy() { return this.Gy; }

/**
 * Returns the parameter a of the elliptic curve function y^2 = x^3 + ax + b.
 *
 * @return        a such that y^2 = x^3 + ax + b
 */

public BigInteger getA()  { return this.a; }

/**
 * Returns the parameter b of the elliptic curve function y^2 = x^3 + ax + b.
 *
 * @return        b such that y^2 = x^3 + ax + b
 */

public BigInteger getB()  { return this.b; }

/**
 * Returns the complex multiplication discriminant used for generation of
 * the elliptic curve.
 *
 * @return        CM discriminant D if CM used; else 0
 */

public long       getD()  { return this.D; }

/**
 * Returns whether complex multiplication was used for generation of this
 * elliptic curve, based on whether there is a value for the discriminant
```

```
   * or not.
   *
   * @return        <code>true</code> if D != 0; else <code>false</code>
   */

  public boolean    byCM()  { return (this.D != 0); }

  BigInteger modulus;  // The prime p for GF(p)
  BigInteger ordG;     // The prime order r of the subgroup generated by G
  BigInteger cofactor; // The cofactor such that #E = cofactor * r
  BigInteger ordE;     // Order of the curve = #E
  BigInteger Gx, Gy;   // x and y coordinates over GF(p) for the generator G
  BigInteger a, b;     // curve parameters such that y^2 = x^3 + ax + b
  long D;              // Complex multiplication discriminant D

}
```

A1.5 GFpGenerator.java

```
package mqp.shardy;

import java.math.*;
import java.security.*;

/**
 * GFpGenerator is the ``factory'' class for generating elliptic curves with
 * complex multiplication. GFpGenerator is instantiated with a curve's desired
 * properties (size, trial division limit for the cofactor, and certainty for the
 * generated prime, although the size is the most important of the parameters),
 * and provides a generator function that will generate a new curve with those
 * properties each time it is called.
 * <p>
 * Because of the black-box nature of curve generation, the generation process
 * appears very simple and easy from the outside. This allows the GFpGenerator
 * object to be easily implemented in any program where elliptic curve generation
 * is needed. The generated curves are returned as GFpEC objects, which can be
 * used as they are, or easily converted to another form for use in a different
 * implementation.
 *
 * @author  Seth Hardy
 * @version 1.0
 */


public class GFpGenerator {

  public static final int SIZED = 160;      // Default bit size.
  public static final int LMAXD = 8;        // Trial division limit for cofactor.
  private static final int CERTD = 30;       // Certainty for generated prime.
  private static final int MAXDIVS = 50;     // Maximum number of prime divisors
  private static final int RUNAWAYD = 10000; // Max D to test

  /**
   * Basic constructor: sets the size of the curve and the trial division limit
   * for the cofactor to the built-in limits. This option should be avoided, as
   * it is always best to know explicitly what you are generating...
   */

  public GFpGenerator() { this(SIZED,LMAXD); }

  /**
   * Sets the size of the curve to the given value, and uses the default trial
   * division limit for the cofactor.
   *
   * @param s       size of the curve (in bits) to be generated
   */


  public GFpGenerator(int s) { this(s,LMAXD); }

  /**
   * Sets the size of the curve to the given value, and uses the given trial
   * division limit for the cofactor.
   *
   * @param s       size of the curve (in bits) to be generated
   * @param l       trial division limit for the cofactor
   */


  public GFpGenerator(int s, int l) {
    BigInteger twoRadQ = BigInteger.ZERO.setBit(s-1);
    this.size = s;
    this.lmax = l;
    this.rmin = BigInteger.ZERO.setBit(size);
    this.rmin = this.rmin.subtract(twoRadQ).add(BigInteger.ONE);
    this.rmin = this.rmin.divide( new BigInteger( Integer.toString(l) ) );
    this.rmax = BigInteger.ZERO.setBit(size).add(twoRadQ);
    this.rmax = this.rmax.add(BigInteger.ONE);
  }

  /**
   * Sets the size of the curve to the given value, and the lower and upper
```

```
 * bounds on the prime subgroup order generated. The cofactor trial division
 * limit used is the default.
 *
 * @param s       size of the curve (in bits) to be generated
 * @param rm      lower bound on the prime subgroup order
 * @param rx      upper bound on the prime subgroup order
 */

public GFpGenerator(int s, BigInteger rm, BigInteger rx) {
  this.size = s;
  this.rmin = new BigInteger( rm.toByteArray() );
  this.rmax = new BigInteger( rx.toByteArray() );
  this.lmax = LMAXD;
}

/**
 * Sets the size of the curve to the given value, the lower and upper
 * bounds on the prime subgroup order, and the cofactor trial division
 * limit.
 *
 * @param s       size of the curve (in bits) to be generated
 * @param rm      lower bound on the prime subgroup order
 * @param rx      upper bound on the prime subgroup order
 * @param l       trial division limit for the cofactor
 */

public GFpGenerator(int s, BigInteger rm, BigInteger rx, int l) {
  this.size = s;
  this.rmin = new BigInteger( rm.toByteArray() );
  this.rmax = new BigInteger( rx.toByteArray() );
  this.lmax = l;
}

/**
 * Generates an elliptic curve with the desired properties.
 * <p>
 * This is where all the other classes come together and work!
 *
 * @param noise   level of noisiness of the generator; 0 = quiet,
 *            1 = basic information, 2 = verbose, 3 = debug
 * @exception NoSuchAlgorithmException    if the secure random object
 *            cannot find the standard algorithms used
 * @return        an elliptic curve generated by complex multiplication
 */

public GFpEC
generate(int noise)
throws NoSuchAlgorithmException {

  int k;
  int gencount;
  int i;
  int pModEight;
  int J;
  long D;
  long rG;
  long rI;
  long rK;
  long[] divs = new long[MAXDIVS];
  boolean reject;
  boolean finalcurve;
  BigInteger p;
  BigInteger r;
  BigInteger cofactor;
  BigInteger a;
  BigInteger b;
  BigInteger temp;
  BigInteger a0;
  BigInteger b0;
  BigInteger[] W      = new BigInteger[2];
  BigInteger[] np     = new BigInteger[2];
  BigInteger[] gen    = new BigInteger[2];
  BigInteger[] possR  = new BigInteger[6];
  BigInteger[] g      = new BigInteger[PolyMath.MAXSIZE];
  BigInteger[] V      = new BigInteger[PolyMath.MAXSIZE];
  BigInteger[] a1     = new BigInteger[PolyMath.MAXSIZE];
  BigInteger[] b1     = new BigInteger[PolyMath.MAXSIZE];
```

```
BigInteger[] pt1    = new BigInteger[PolyMath.MAXSIZE];
BigInteger[] pt2    = new BigInteger[PolyMath.MAXSIZE];
BigInteger[] bigrcp = new BigInteger[PolyMath.MAXSIZE];
BigDecimal ktemp;
SecureRandom prand;

D = 0;
i = 0;

if(noise > 0) {
  System.out.println("Starting generation process, noise level (" + noise + ")");
  if(noise >= 3) {
    System.out.println("Warning: this noise level is going to output a _lot_ of information.");
    System.out.println("You have been warned.");
  }
  System.out.println("Curve size (in bits): " + size);
  System.out.println("Cofactor trial division limit: " + lmax);
  System.out.println("Lower bound on prime subgroup order: " + rmin);
  System.out.println("Upper bound on prime subgroup order: " + rmax);
}

prand = new SecureRandom();

do {

do {

  finalcurve = true;
  D = 0;

  p = new BigInteger(this.size, CERTD, prand);
  if(noise > 0) System.out.println("Prime generated: " + p);

  pModEight = (int)(p.mod(CMHelper.BIGEIGHT)).longValue();
  if(noise >= 3) System.out.println("p % 8: " + pModEight);
  ktemp = CMHelper.sqrt(p);

  if(noise >= 3) System.out.println("sqrt(p): " + ktemp);

  ktemp.add( new BigDecimal(BigInteger.ONE) );
  ktemp = ktemp.multiply(ktemp);

  if(noise >= 3) System.out.println("( sqrt(p) + 1 )^2: " + ktemp);

  ktemp = ktemp.divide(new BigDecimal(this.rmin),BigDecimal.ROUND_HALF_DOWN);
  k = ktemp.intValue();

  if(noise >= 3) System.out.println("k value for D: " + k );

do {
  reject = false;
  D = CMHelper.getNextD(k,pModEight,D);
  if(D > RUNAWAYD) {
    finalcurve = false;
    if(noise >= 3) System.out.println("Runaway D! Restarting at 1.");
    D = 1;
    p = new BigInteger(this.size, CERTD, prand);
    if(noise > 0) System.out.println("Prime generated: " + p);
    pModEight = (int)(p.mod(CMHelper.BIGEIGHT)).longValue();
    if(noise >= 3) System.out.println("p % 8: " + pModEight);
    ktemp = CMHelper.sqrt(p);
    if(noise >= 3) System.out.println("sqrt(p): " + ktemp);
    ktemp.add( new BigDecimal(BigInteger.ONE) );
    ktemp = ktemp.multiply(ktemp);
    if(noise >= 3) System.out.println("( sqrt(p) + 1 )^2: " + ktemp);
    ktemp = ktemp.divide(new BigDecimal(this.rmin),BigDecimal.ROUND_HALF_DOWN);
    k = ktemp.intValue();
    if(noise >= 3) System.out.println("k value for D: " + k );
    D = CMHelper.getNextD(k,pModEight,D);
  }

  if(noise > 2)
    System.out.println("Trying D = " + D + " ");
  J = CMHelper.jacobip(new BigInteger(Long.toString(-D)),p);
  if( J == -1 ) reject = true;
  if(!reject) {
    divs = CMHelper.primedivs(D);
```

```
    i = 0;
    while( (divs[i] != 0) && (i < MAXDIVS) ) {
      J = CMHelper.jacobip(p,new BigInteger(Long.toString(divs[i])));
      if( J == -1 ) reject = true;
      i++;
    }
  }
  if(noise >= 3)
    if(reject) System.out.println("  Jacobi test failed.");
  if(!reject) {
    W = CMHelper.getWV(p,D);
    if( W[0].signum() == 0) reject = true;
    if(noise >= 3) {
      if( W[0].signum() == 0) {
        System.out.println("  W,V Combination not found: D = " + D + " not a CM discriminant.");
      }
      else {
        System.out.print("W: " + W[0]);
        if( W[1].signum() != 0) System.out.println(", V: " + W[1]);
        else System.out.println(", V not determined.");
      }
    }
  }
  for(i=0;i<6;i++) { possR[i] = BigInteger.ZERO; }
  if(!reject) {
    possR[0] = p.add(BigInteger.ONE).add(W[0]);
    possR[1] = p.add(BigInteger.ONE).subtract(W[0]);
    if(D == 1) {
      possR[2] = p.add(BigInteger.ONE).add(W[1]);
      possR[3] = p.add(BigInteger.ONE).subtract(W[1]);
    }
    if(D == 3) {
      temp = W[0].add(W[1].multiply(CMHelper.BIGTHREE)).divide(CMHelper.BIGTWO);
      possR[2] = p.add(BigInteger.ONE).add(temp);
      possR[3] = p.add(BigInteger.ONE).subtract(temp);
      temp = W[0].subtract(W[1].multiply(CMHelper.BIGTHREE)).divide(CMHelper.BIGTWO);
      possR[4] = p.add(BigInteger.ONE).add(temp);
      possR[5] = p.add(BigInteger.ONE).subtract(temp);
    }
  }
  if(!reject) {
    i=0;
    if(noise >= 2) System.out.println("Looking for good orders with D = " + D);         while( (i < 6) && (possR[i].signum() != 0) ) {
      if(noise >= 3) System.out.println("  Trying possible order " + i + "...");
      np = CMHelper.nearprime(possR[i],lmax,rmin,rmax);
      if( (np[0].signum() != 0) && (np[1].signum() != 0) ) break;
      i++;
    }
    if( ((D==1)&&(i==4)) || ((D==3)&&(i==6)) ) {
      reject = true;
      if(noise > 3) System.out.println("  None worked, moving on.");
    }
    else if(i==2) {
      reject = true;
      if(noise >= 3) System.out.println("  None worked, moving on.");
    }
  }
} while(reject);

r = new BigInteger( np[0].toByteArray() );
cofactor = new BigInteger( np[1].toByteArray() );

if(noise > 0) {
  System.out.println("CM Discriminant D: " + D);
  System.out.println("Subgroup order: " + r);
  System.out.println("Cofactor: " + cofactor);
}


if(D == 1) {
  a0 = new BigInteger("1");
  b0 = new BigInteger("0");
}
else if(D == 2) {
  a0 = new BigInteger("-30").mod(p);
  b0 = new BigInteger("56").mod(p);
}
```

```
else if(D == 3) {
  a0 = new BigInteger("0");
  b0 = new BigInteger("1");
}
else if(D == 7) {
  a0 = new BigInteger("-35").mod(p);
  b0 = new BigInteger("98").mod(p);
}
else if(D == 11) {
  a0 = new BigInteger("-264").mod(p);
  b0 = new BigInteger("1694").mod(p);
}
else if(D == 19) {
  a0 = new BigInteger("-152").mod(p);
  b0 = new BigInteger("722").mod(p);
}
else if(D == 43) {
  a0 = new BigInteger("-3440").mod(p);
  b0 = new BigInteger("77658").mod(p);
}
else if(D == 67) {
  a0 = new BigInteger("-29480").mod(p);
  b0 = new BigInteger("1948226").mod(p);
}
else if(D == 163) {
  a0 = new BigInteger("-8697680").mod(p);
  b0 = new BigInteger("9873093538").mod(p);
}
else {
bigrcp = CMHelper.rcpolynom(D);
if( (CMHelper.classnumber(D) > PolyMath.MAXSIZE) )
  finalcurve = false;

for(int qw=0;qw<PolyMath.MAXSIZE;qw++)
  bigrcp[qw] = bigrcp[qw].mod(p);

if( W[0].mod(CMHelper.BIGTWO).signum() == 0 ) {
  int d8 = (int)(D % 8);
  if( (d8 == 1) || (d8 == 2) || (d8 == 6) ) {
    rI = 3;
    rK = 2;
  }
  if(d8 == 7) {
    rI = 3;
    rK = 1;
  }
  if(d8 == 3) {
    rK = 1;
    if(D % 3 == 0) rI = 2;
    else rI = 0;
  }
  else {
    rI = 6;
    rK = 4;
  }
  rG = CMHelper.gcd(3,D);
  if(noise >= 3)
    System.out.println("Starting to look for a linear factor...");
  g = PolyMath.polyFactor(bigrcp,1,p);
  if(noise >= 3) {
    System.out.println("Factor[0] = " + g[0]);
    System.out.println("Factor[1] = " + g[1]);
  }
  temp = new BigInteger(Long.toString((long)Math.pow((double)-1,D)*(long)Math.pow(2,4*rI/rK)));;
  V[0] = g[0].negate().pow((int)(24/(rG*rK))).multiply(temp);
  if(noise >= 3) System.out.println("V = " + V[0]);
  a0 = V[0].add(new BigInteger("16"));
  a0 = a0.multiply(new BigInteger("64").add(V[0]));
  a0 = a0.multiply(p.subtract(CMHelper.BIGTHREE)).mod(p);
  b0 = V[0].add(new BigInteger("64"));
  b0 = b0.multiply(b0);
  b0 = b0.multiply(V[0].subtract(new BigInteger("8")));
  b0 = b0.multiply(p.subtract(CMHelper.BIGTWO)).mod(p);
}
else {
  if(noise >= 3) System.out.println("Starting to look for a cubic factor...");
  g = PolyMath.polyFactor(bigrcp,3,p);
```

```
    if(noise >= 3) {
      for(int o=0;o<4;o++) {
        System.out.println("Factor[" + o + "] = " + g[o]);
      }
    }
    if(PolyMath.degree(g) != 3) {
      if(noise >= 3) System.out.println("bad cubic factor!");
      finalcurve = false;
    }
    for(i=0;i<PolyMath.MAXSIZE;i++) {
      V[i] = BigInteger.ZERO;
    }
    if(D % 3 == 0) {
      V[8] = new BigInteger("-256");
      V = PolyMath.polyMod(V,g,p);
    }
    else {
      V[24] = new BigInteger("-1");
      V = PolyMath.polyMod(V,g,p);
    }
    pt1 = (BigInteger[])V.clone();
    pt1[0] = pt1[0].add(new BigInteger("64"));
    pt2 = (BigInteger[])V.clone();
    pt2[0] = pt2[0].add(new BigInteger("256"));
    a1 = PolyMath.polyMultMod(pt1,pt2,g,p);
    a1 = PolyMath.polySMult(a1,new BigInteger("-3"),p);
    a1 = PolyMath.polyMod(a1,g,p);
    pt2[0] = pt2[0].subtract(new BigInteger("758"));
    b1 = PolyMath.polyMultMod(pt1,pt1,g,p);
    b1 = PolyMath.polyMultMod(b1,pt2,g,p);
    b1 = PolyMath.polySMult(b1,CMHelper.BIGTWO,p);
    b1 = PolyMath.polyMod(b1,g,p);
    a1 = PolyMath.polyExpMod(a1,CMHelper.BIGTHREE,g,p);
    b1 = PolyMath.polyExpMod(b1,CMHelper.BIGTWO,g,p);
    i=0;
    do { i++; } while( (i < PolyMath.MAXSIZE - 1) && (a1[i].signum() == 0) );
    a0 = a1[i].multiply(b1[i]).mod(p);
    b0 = a1[i].multiply(b1[i]).multiply(b1[i]).mod(p);
  }

    if(noise >= 3) System.out.println("Keep curve? " + finalcurve);
  }
  } while(!finalcurve);

  if(noise >= 2) {
    System.out.println("a0 = " + a0);
    System.out.println("b0 = " + b0);
  }

  reject = true;
  finalcurve = true;
  gencount = 0;

  do {
    temp = new BigInteger( p.bitLength(), prand ).mod(p);
    if(D == 1) {
      a = a0.multiply(temp).mod(p);
      b = BigInteger.ZERO;
    }
    else if(D == 3) {
      a = BigInteger.ZERO;
      b = b0.multiply(temp).mod(p);
    }
    else {
      a = a0.multiply(temp).multiply(temp).mod(p);
      b = b0.multiply(temp).multiply(temp).multiply(temp).mod(p);
    }
    gen = ECMath.genG(r,cofactor.intValue(),a,b,p);
    if( gen[0].signum() != 0 ) {
      reject = false;
    }
    gencount++;
  } while( reject && (gencount < 100));

  if(gencount >= 100) {
    if(noise > 0) System.out.println("Couldn't find generator, restarting.");
    finalcurve = false;
```

```
    }

    } while(!finalcurve);

    if(noise > 0) {
      System.out.println("a: " + a);
      System.out.println("b: " + b);
      System.out.println("G: (" + gen[0] + "," + gen[1] + ")");
    }

    GFpEC curve = new GFpEC(p,r,cofactor,gen[0],gen[1],a,b,D);
    return curve;
  }


    private int size;           // size of the curve to be generated
    private BigInteger rmin, rmax;      // lower and upper bounds on ord(r)
    private int lmax;                   // trial division limit for cofactor

}
```

A1.6 PolyMath.java

```
package mqp.shardy;

import java.math.*;
import java.util.*;
import java.security.*;

/**
 * PolyMath provides all of the necessary functions for arithmetic on polynomials.
 * Does not create an abstract data type for holding the polynomials; polynomials
 * are represented as arrays of BigIntegers (or BigComplexes), as needed by the
 * elliptic curve generation process.
 *
 * @author  Seth Hardy
 * @version 1.0
 */

public class PolyMath {

  /**
   * Maximum polynomial size.
   */


  public static final int MAXSIZE    = 100;

  /**
   * Maximum double length polynomial size.
   */

  public static final int DOUBLESIZE = 2 * MAXSIZE - 1;

  /**
   * Determines the degree of a single length polynomial.
   *
   * @param x       single length polynomial
   * @return        the degree of x
   */

  public static int
  degree(BigInteger[] x) {
    int i;
    for(i=MAXSIZE-1;i>0;i--) {
     if( x[i].signum() != 0 ) break;
    }
    return i;
  }

  /**
   * Determines the degree of a double length polynomial.
   *
   * @param x       double length polynomial
   * @return        the degree of x
   */

  public static int
  ddegree(BigInteger[] x) {
    int i;
    for(i=DOUBLESIZE-1;i>0;i--) {
      if( x[i].signum() != 0 ) break;
    }
    return i;
  }

  /**
   * Multiplies a single length polynomial by a scalar value modulo a prime.
   *
   * @param x       single length polynomial
   * @param y       scalar to multiply by
   * @param p       prime to do modulo reduction by
   * @return        y*x, whose elements are mod p
   */

  public static BigInteger[]
```

```
polySMult(BigInteger[] x, BigInteger y, BigInteger p) {
  BigInteger t[] = new BigInteger[MAXSIZE];
  for(int i=0;i<MAXSIZE;i++) {
    t[i] = x[i].multiply(y).mod(p);
  }
  return t;
}

/**
 * Performs modulo reduction of one polynomial by another, modulo a prime.
 *
 * @param x       single length polynomial
 * @param y       single length polynomial
 * @param p       prime to do modulo reduction by
 * @return        x mod y, whose elements are mod p
 */

public static BigInteger[]
polyMod(BigInteger[] x, BigInteger[] y, BigInteger p) {
  int ly;
  int lr;
  BigInteger[] quot = new BigInteger[MAXSIZE];
  BigInteger[] rem  = (BigInteger[])x.clone();
  BigInteger l;
  for(int i=0;i<MAXSIZE;i++) {
    quot[i] = BigInteger.ZERO;
  }
  ly = degree(y);
  lr = degree(rem);
  while( ly <= lr ) {
    l = rem[lr].multiply( y[ly].modInverse(p) ).mod(p);
    for(int i=0;i<=ly;i++) {
      rem[lr-i] = rem[lr-i].subtract(y[ly-i].multiply(l)).mod(p);
    }
    lr = degree(rem);
  }
  for(int i=0;i<MAXSIZE;i++) {
    rem[i] = rem[i].mod(p);
  }
  return rem;
}

/**
 * Performs modulo reduction of one polynomial by another, modulo a prime.
 *
 * @param x       double length polynomial
 * @param y       single length polynomial
 * @param p       prime to do modulo reduction by
 * @return        x mod y, whose elements are mod p
 */

public static BigInteger[]
dpolyMod(BigInteger[] x, BigInteger[] y, BigInteger p) {
  int ly;
  int lr;
  BigInteger[] quot = new BigInteger[MAXSIZE];
  BigInteger[] rem  = (BigInteger[])x.clone();
  BigInteger l;
  BigInteger r[] = new BigInteger[MAXSIZE];
  for(int i=0;i<MAXSIZE;i++) {
    quot[i] = BigInteger.ZERO;
  }
  ly = degree(y);
  lr = ddegree(rem);
  while( ly <= lr ) {
    l = rem[lr].multiply( y[ly].modInverse(p) ).mod(p);
    for(int i=0;i<=ly;i++) {
      rem[lr-i] = rem[lr-i].subtract(y[ly-i].multiply(l)).mod(p);
    }
    lr = ddegree(rem);
  }
  for(int i=0;i<MAXSIZE;i++) {
    r[i] = rem[i].mod(p);
  }
  return r;
}
```

```
/**
 * Divides one polynomial by another, modulo a prime.
 *
 * @param x       single length polynomial
 * @param y       single length polynomial
 * @param p       prime to do modulo reduction by
 * @return        x / y, whose elements are mod p
 */

public static BigInteger[]
polyDiv(BigInteger[] x, BigInteger[] y, BigInteger p) {
  int lx;
  int ly;
  BigInteger[] quot = new BigInteger[MAXSIZE];
  BigInteger[] rem  = new BigInteger[MAXSIZE];
  rem = (BigInteger[])x.clone();
  for(int i=0;i<MAXSIZE;i++) {
    quot[i] = BigInteger.ZERO;
  }
  for(lx = MAXSIZE-1;lx>0;lx--) {
    if( x[lx].signum() != 0 ) break;
  }
  for(ly = MAXSIZE-1;ly>0;ly--) {
    if( y[ly].signum() != 0 ) break;
  }
  for(int i=0;i<=lx-ly;i++) {
    quot[i] = rem[i].multiply( y[0].modInverse(p) );
    for(int j=0;j<=ly-1;j++) {
      rem[j+i] = rem[j+i].subtract( quot[i].multiply( y[j] ) );
    }
  }
  for(int i=0;i<MAXSIZE;i++) {
    quot[i] = quot[i].mod(p);
  }
  return quot;
}

/**
 * Multiplies one polynomial by another, modulo a reduction polynomial.
 *
 * @param x       single length polynomial
 * @param y       single length polynomial
 * @param g       single length reduction polynomial
 * @param p       prime to do modulo reduction by
 * @return        x * y (mod g), whose elements are mod p
 */

public static BigInteger[]
polyMultMod(BigInteger[] x, BigInteger[] y, BigInteger[] g, BigInteger p) {
  BigInteger[] t = new BigInteger[DOUBLESIZE];
  for(int i=0;i<DOUBLESIZE;i++) {
    t[i] = BigInteger.ZERO;
  }
  for(int i=0;i<MAXSIZE;i++) {
    for(int j=0;j<MAXSIZE;j++) {
      t[i+j] = t[i+j].add(x[i].multiply(y[j])).mod(p);
    }
  }
  return dpolyMod(t,g,p);
}

/**
 * Exponentiation on a polynomial, modulo a reduction polynomial.
 * <p>
 * Uses method described in IEEE P1363 A.5.1.
 *
 * @param x       single length polynomial for base
 * @param y       exponent
 * @param g       single length reduction polynomial
 * @param p       prime to do modulo reduction by
 * @return        x^y (mod g), whose elements are mod p
 */

public static BigInteger[]
polyExpMod(BigInteger[] x, BigInteger y, BigInteger[] g, BigInteger p) {
  String s = y.toString(2);
  BigInteger[] t = (BigInteger[])x.clone();
```

```
  for(int i=1;i<s.length();i++) {
    t = polyMultMod(t,t,g,p);
    if( s.charAt(i) == '1' ) t = polyMultMod(t,x,g,p);
  }
  return t;
}

/**
 * Calculates the greatest common divisor of two polynomials modulo a prime.
 * <p>
 * Uses method described in IEEE P1363 A.5.2.
 *
 * @param x       single length polynomial
 * @param y       single length polynomial
 * @param p       prime to do modulo reduction by
 * @return        gcd(x,y), whose elements are mod p
 */

public static BigInteger[]
polyGCD(BigInteger[] x, BigInteger[] y, BigInteger p) {
  BigInteger[] a = (BigInteger[])x.clone();
  BigInteger[] b = (BigInteger[])y.clone();
  BigInteger[] c = new BigInteger[MAXSIZE];
  int i;
  while( degree(b) != 0 ) {
    c = polyMod(a,b,p);
    a = (BigInteger[])b.clone();
    b = (BigInteger[])c.clone();
  }
  i = MAXSIZE-1;
  while( a[i].signum() == 0 ) i--;
  return polySMult(a,a[i].modInverse(p),p);
}

/**
 * Calculates a given-degree factor of a polynomial.
 * <p>
 * Uses method described in IEEE P1363 A.5.3.
 *
 * @param x       single length polynomial
 * @param d       desired degree of factor
 * @param p       prime to do modulo reduction by
 * @result        a polynomial factor of x of degree d
 */

public static BigInteger[]
polyFactor(BigInteger[] x, int d, BigInteger p) {
  BigInteger[] g = new BigInteger[MAXSIZE];
  BigInteger[] u = new BigInteger[MAXSIZE];
  BigInteger[] c = new BigInteger[MAXSIZE];
  BigInteger[] h = new BigInteger[MAXSIZE];
  BigInteger e;
  g = (BigInteger[])x.clone();
  while( degree(g) > d ) {
    u = randMonPoly(2*d-1,p);
    e = p.pow(d).subtract(BigInteger.ONE).divide(CMHelper.BIGTWO).mod(p);
    c = polyExpMod(u,e,g,p);
    c[0] = c[0].subtract(BigInteger.ONE).mod(p);
    h = polyGCD(c,g,p);

    if( (degree(h) != 0) && ( degree(g) != degree(h) ) ) {
      if( 2*degree(h) > degree(g) ) {
        g = polyDiv(g,h,p);
      }
      else {
        g = (BigInteger[])h.clone();
      }
    }
  }
  return g;
}

/**
 * Generates a random monic polynomial of a given degree.
 *
 * @param degree      degree of random monic polynomial
 * @param p           prime to do modulo reduction by
```

```
   * @return            a random monic polynomial mod p with deg(degree)
   */

  public static BigInteger[]
  randMonPoly(int degree, BigInteger p) {
    BigInteger[] x = new BigInteger[MAXSIZE];
    SecureRandom rnd = new SecureRandom();
    for(int i=0;i<MAXSIZE;i++) {
      x[i] = BigInteger.ZERO;
    }
    if(degree>MAXSIZE-1) return x;
    x[degree] = BigInteger.ONE;
    for(int i=degree-1;i>=0;i--) {
      x[i] = new BigInteger( p.bitLength(), rnd ).mod(p);
    }
    return x;
  }

  /**
   * Multiplies two polynomials with elements of class Complex.
   *
   * @param a       polynomial with elements of class Complex
   * @param b       polynomial with elements of class Complex
   * @result        a * b
   */

  public static Complex[]
  polyMult(Complex[] a, Complex[] b) {
    Complex[] x = new Complex[DOUBLESIZE];
    for(int i=0;i<DOUBLESIZE;i++) {
      x[i] = new Complex(0,0);
    }
    for(int i=0;i<MAXSIZE;i++) {
      for(int j=0;j<MAXSIZE;j++) {
        x[i+j] = x[i+j].add(a[i].multiply(b[j]));
      }
    }
    return x;
  }

}
```

A1.7 ecgenerate.java

```
import mqp.shardy.*; import java.math.BigInteger; import
java.security.*;

/**
 * ecgenerate is a simple program that implements the GFpGenerator class.
 * It is a basic elliptic curve generator; it takes desired parameters for a
 * generated elliptic curve, generates a curve, and outputs its parameters to
 * the screen.
 *
 * @author  Seth Hardy
 * @version 1.0
 */

public class ecgenerate {

  public static void main(String[] args) {

    int x;
    int s = 0;
    int l = 0;
    int n = 0;
    BigInteger rmin = BigInteger.ZERO;
    BigInteger rmax = BigInteger.ZERO;
    String temp;
    GFpEC c;
    GFpGenerator f;

    if( args.length % 2 == 1 ) {
      help();
      return;
    }

    x = 0;

    while( x < args.length - 1) {
      if (args[x].compareTo("-s") == 0) {
        temp = args[x+1];
        try {
          s = Integer.parseInt(temp);
        }
        catch(NumberFormatException e) {
          help();
          return;
        }
        if(s <= 0) help();
      }
      else if (args[x].compareTo("-l") == 0) {
        temp = args[x+1];
        try {
          l = Integer.parseInt(temp);
        }
        catch(NumberFormatException e) {
          help();
          return;
        }
        if(l <= 0) help();
      }
      else if (args[x].compareTo("-rmin") == 0) {
        temp = args[x+1];
        try {
          rmin = new BigInteger(temp);
        }
        catch(NumberFormatException e) {
          help();
          return;
        }
      }
      else if (args[x].compareTo("-rmax") == 0) {
        temp = args[x+1];
        try {
          rmax = new BigInteger(temp);
        }
        catch(NumberFormatException e) {
```

```
        help();
        return;
      }
    }
    else if (args[x].compareTo("-n") == 0) {
      temp = args[x+1];
      try {
        n = Integer.parseInt(temp);
      }
      catch(NumberFormatException e) {
        help();
        return;
      }
      if(n < 0) help();
    }
    else {
      help();
      return;
    }
    x = x + 2;
  }

  if( (s != 0) && (l != 0) && (rmin.signum() != 0) && (rmax.signum() != 0))
    f = new GFpGenerator(s,rmin,rmax,l);
  else if( (s != 0) && (rmin.signum() != 0) && (rmax.signum() != 0))
    f = new GFpGenerator(s,rmin,rmax);
  else if( (s != 0) && (l != 0))
    f = new GFpGenerator(s,l);
  else if( s != 0 )
    f = new GFpGenerator(s);
  else f = new GFpGenerator();

  try {
    c = f.generate(n);
  }
  catch(java.security.NoSuchAlgorithmException e) {
    System.out.println("Default java.security SecureRandom algorithm not found.");
    return;
  }

  System.out.println("curve parameters:");
  System.out.println("  p  = " + c.getP());
  System.out.println("  r  = " + c.getR());
  System.out.println("  k  = " + c.getK());
  System.out.println("  #E = " + c.getU());
  System.out.println("  G  = (" + c.getGx() + ",");
  System.out.println("         " + c.getGy() + ")");
  System.out.println("  a  = " + c.getA());
  System.out.println("  b  = " + c.getB());
  System.out.println("  D  = " + c.getD());

  return;
}

static void
help() {
  System.out.println("ecgenerate v1.0 - a simple elliptic curve generator.");
  System.out.println("written by Seth Hardy, Worcester Polytechnic Institute.");
  System.out.println("usage:");
  System.out.println("  ecgenerate [ -s <curve size in bits> ]");
  System.out.println("             [ -t <cofactor trial division limit>]");
  System.out.println("             [ -rmin <min order> -rmax <max order> ]");
  System.out.println("             [ -noise <noise level> ]");
  System.out.println("guidelines:");
  System.out.println("  s and t should be positive.");
  System.out.println("  rmin and rmax should fall within the Hasse bound.");
  System.out.println("    don't use this option unless you know what you're doing!");
  System.out.println("  noise should be one of the values below.");
  System.out.println("defaults to a curve of 160 bits and trial division limit of 8.");
  System.out.println("noise levels: ");
  System.out.println("  0 = quiet (default).");
  System.out.println("  1 = basic information.");
  System.out.println("  2 = verbose information.");
  System.out.println("  3 = _very_ verbose information, for those who like to read.");
  System.out.println("");
  return;
}
```

```
}
```

Appendix A2

Program Documentation

Contained here is the documentation for the following classes:

1. `CMHelper`
2. `Complex`
3. `ECMath`
4. `GFpEC`
5. `GFpGenerator`
6. `PolyMath`

The documentation was written through the Java documentation standard, from documentation comment blocks in the code. The HTML documentation for the package `mqp.shardy` was created through the command `javadoc mqp/shardy/*.java`.

Appendix A3

Program Timings

Following is the full set of timings done on the elliptic curve generator.

Bit Size	Type						Average
40	CPU	6.6 5.4	10.3 23.9	5.5 5.5	5.4 5.4	5.6 6.6	8.0
40	Real	14.5 13.3	18.2 31.9	13.2 13.5	13.1 13.4	13.2 14.7	15.9
160	CPU	292.1 5.7	175.0 104.0	174.5 68.3	70.5 238.1	35.8 153.9	131.8
160	Real	303.0 13.5	184.5 113.2	184.3 76.8	79.2 248.5	43.9 163.1	141.0
320	CPU	714.4 661.6	259.6 610.7	66.4 966.3	415.5 276.3	617.4 793.7	538.2
320	Real	726.2 673.2	269.2 629.7	74.7 989.9	426.6 287.8	628.4 810.4	551.6

Table A3.1: Performance of the Elliptic Curve Generator, in seconds

Bibliography

[1] I. Blake, G. Seroussi, and N. Smart. Elliptic Curves in Cryptography. Cambridge University Press, 1999.

[2] J. Durbin. Modern Algebra: an Introduction. Wiley and Sons, 2000.

[3] ISIS Project Information Fiche. http://www.ispo.cec.be/isis/99elias.htm

[4] A. Menezes, P. van Oorschot, and S. Vanstone. Handbook of Applied Cryptography. CRC Press, 1996.

[5] A. Menezes, T. Okamoto, and S. Vanstone. Reducing elliptic curve logarithms to a finite field, IEEE Trans. Info. Theory, vol 39, pp. 1639-1646, 1993.

[6] IEEE P1363/D13 (Draft Version 13). Standard Specifications for Public Key Cryptography. November, 1999.