MATH491

Summer Research Project

2006– 2007

A MATLAB Implementation of Elliptic Curve Cryptography

Hamish Silverwood

Department of Mathematics and Statistics
University of Canterbury

Math 491 Project: A MATLAB Implementation of Elliptic Curve Cryptography

Hamish G. M. Silverwood

Abstract

The ultimate purpose of this project has been the implementation in MATLAB of an Elliptic Curve Cryptography (ECC) system, primarily the Elliptic Curve Diffie-Hellman (ECDH) key exchange. We first introduce the fundamentals of Elliptic Curves, over both the real numbers and the integers modulo p where p is prime. Then the theoretical underpinnings of the ECDH system are covered, including a brief look at how this system is broken. Next we develop the individual elements that will be needed in the implementation of ECDH, such as functions for calculating modular square roots and the addition of points on an EC. We then bring these elements together to create a working ECDH program, and discuss the limitations of the MATLAB environment in which it was created. Finally we look at the real world application of ECC and its future in the realm of cryptography.

1 Introduction

The goal of this report is to first give a description of the mathematics behind Elliptic Curve Cryptography (ECC), in particular the Elliptic Curve Diffie-Hellman (ECDH) key exchange system, and secondly to describe and develop the algorithms and methods necessary for the implementation of the ECDH system in the MATLAB environment.

Section 2 introduces the fundamental mathematics of the Elliptic Curve, over both the real numbers and the integers modulo p, where p is prime. This includes the addition law denoted by $\boxplus$, and the construction of the abelian elliptic curve group. Next, in Section 3, we look at the ECDH key exchange system. The basis of this system is the Elliptic Curve Discrete Logarithm Problem (ECDLP), which is discussed in some detail. After outlining the steps necessary to perform an ECDH key exchange between two people, we give a brief overview of the methods available to solve the ECDLP and hence break the ECDH system.

From Section 4 onwards our focus changes away from the theoretical foundations and towards the practical application of ECC. We describe and develop the tools, methods and algorithms necessary for ECC, starting with the basic operations of modular exponentiation and the calculation of inverses over finite fields, and moving on to the more complicated tasks of finding modular square roots and the addition of points on the EC. In Section 5 we bring these individual pieces together to construct a functional ECDH system, and then discuss the limitations of this program and the MATLAB environment it was created in. Finally we discuss how ECC is implemented in the real world, and its future use and standardization.

2 Fundamentals of Elliptic Curves

Elliptic Curves are a type of algebraic curve with a general form described by the Diophantine equation

$$y^2 = ax^3 + bx^2 + cx + d\,. \tag{1}$$

They were utilised by Andrew Wiles in his proof of Fermat's Last Theorem, and they are gaining popularity in the realm of cryptography for their security and efficiency over current cryptographic methods. They form a large part of US National Security Agency's (NSA) Suite B of cryptographic algorithms that will, over the next decade, replace those currently in use, such as RSA and the Diffie-Hellman Key Exchange [NSA, a]. An illustration of the superiority of ECC over older methods can be seen by looking at the comparative key size needed to ensure a similar level of security: where RSA would need a key size of 1024 bits, methods based on ECs would only require one of 160 bits [NSA, b].

The elliptic curve can be defined over many fields, ranging from the complex numbers $\mathbb{C}$ and the rationals $\mathbb{Q}$ to the real numbers $\mathbb{R}$ and integers modulo p as covered in Sections 2.1 and 2.2 [Silverman, 2005, pg. 100-104]. For any field K we can in general find a group $(E(K), \boxplus)$, that consists of pairs of elements in K that are solutions to Equation 1, plus an abstract point denoted by ∞, which will be discussed further below. The $\boxplus$ operator that acts upon the elements of the group remains the same algebraicly for each field, though the procedure for calculating it may vary slightly.

2.1 Elliptic Curves over $\mathbb{R}$

An elliptic curve over the field $\mathbb{R}$, denoted as $E(\mathbb{R})$, is the set of real solutions to equation 1, with a, b, c, $d \in \mathbb{R}$ and $a \neq 0$, i.e. the set $\{(x, y) \in \mathbb{R}^2 | y^2 = ax^3 + bx^2 + cx + d\}$, with the addition of an abstract point ∞ [Martin, 2006, pg. 5]. We must also state the condition that to be an elliptic curve Equation 1 should have three distinct roots when $y = 0$, either real or complex. This guarantees that the graph of the curve is non-singular [Silverman, 2005, pg. 94], and hence a tangent line can be found at every point, the importance of which will become apparent later.

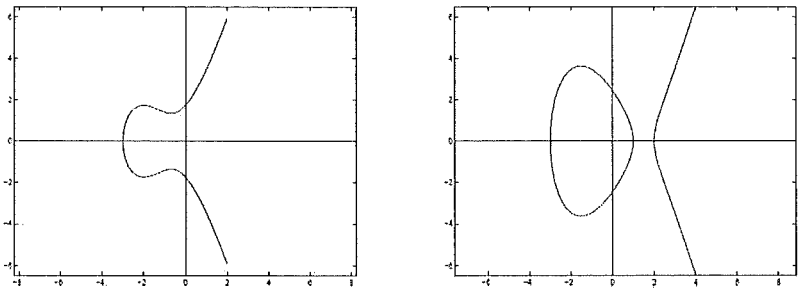

Figure 1: Elliptic Curve Graphs [Vercauteren, 2005]

The graph produced by plotting the elliptic curve on the x and y axes has one of the following forms: From this elliptic curve we can construct an Abelian group by introducing the binary operation $\boxplus$, called the addition law on E, which over $\mathbb{R}$ can be given by a simple geometric construction [Martin, 2006, pg. 8].

Definition 1. : $\boxplus$ *over* $\mathbb{R}$
Let P_1, $P_2 \in E(\mathbb{R})$, *with* $P_1 = (x_1, y_1)$ *and* $P_2 = (x_2, y_2)$.
Case One:

$$P_1 \boxplus P_2 = \begin{cases} P_1 : \textit{if } P_2 = \infty \\ P_2 : \textit{if } P_1 = \infty \end{cases}$$

Case Two:

$$P_1 \boxplus P_2 = \begin{cases} \infty : \textit{if } x_1 = x_2 \textit{ and } y_1 \neq y_2 \\ \infty : \textit{if } x_1 = x_2 \textit{ and } y_1 = y_2 = 0 \end{cases}$$

Case Three:
If Case One and Two are not met, we then have:

$$P_1 \boxplus P_2 = P_3 = (x_3, y_3)$$

where

$$x_3 = -x_1 - x_2 - \frac{b}{a} + \frac{m^2}{a} \tag{2}$$

$$y_3 = -y_1 + m(x_1 - x_3) \tag{3}$$

$$m = \begin{cases} \dfrac{y_2 - y_1}{x_2 - x_1} & : \textit{if } P_1 \neq P_2 \\ \dfrac{3a{x_1}^2 + 2bx_1 + c}{2y_1} & : \textit{if } P_1 = P_2 \end{cases} \tag{4}$$

To illustrate the geometrical intepretation of the addition law its best to first consider Case Three. Taking the two points P_1 and P_2, we connect them with a straight line. This line is gauranteed to intersect the elliptic curve at one other location - we denote this point as $P_3{}'$. We then reflect $P_3{}'$ along the x axis, and

this gives us our answer P_3.

The gradient of this connecting line is m, and is calculated using different formulas depending on whether the points are distinct or identical. If the points are distinct the gradient is easily calculated using the standard high school method that ammounts to 'rise over run'. If the points are identical then we must instead use the tangent line at the point, the gradient of which is found by implicit differentiation; i.e.

$$m = \frac{dy}{dx} = \frac{3a{x_1}^2 + 2bx_1 + c}{2y_1}$$

Cases One and Two are then variations upon this idea. The point ∞ can be thought of as lying an infinite distance above (or below) the x axis, so if P_2 is ∞ and P_1 is somewhere on the elliptic curve then the straight line connecting the two is considered vertical. ${P_3}'$ is then directly below (or above) P_1, and thus the reflection in the x-axis returns us to P_1. Case Two occurs when the two points have the same x value, resulting in P_3 being ∞.

2.2 Elliptic Curves over $\mathbb{Z}_p$

Despite the utility offered by elliptic curves over $\mathbb{R}$, it is only when one uses them over a finite field $\mathbb{F}$ that they can become the basis for practical cryptographic schemes. This is because we work with finite computers that are unable to accurately store and manipulate the elements of an infinite field such as (R), which has an infinite number of elements, some of which will be infinitely recuring decimals. We hence use finite fields, primarily $\mathbb{Z}_p$ (integers modulo p), where p is a prime.

The Elliptic Curve given in Equation 1 now has the limitation that the coefficients a, b, c, d are integers reduced modulo p, as are the variables x and y. We then define our Elliptic Curve as the set of integer solutions modulo p to equation 1 with the addition of the ∞ point, and denote it as $\bar{E}(\mathbb{Z}_p)$.

Although the geometrical interpretation of $\boxplus$ that we used in section 2.1 is no longer applicable, its algebraic definition can still be used, albeit with a slightly modified computational procedure.

Definition 2. : $\boxplus$ *over* $\bar{E}(\mathbb{Z}_p)$
Let P_1, $P_2 \in \bar{E}(\mathbb{Z}_p)$, *with* $P_1 = (x_1, y_1)$ *and* $P_2 = (x_2, y_2)$.
Case One:

$$P_1 \boxplus P_2 = \begin{cases} P_1 : \textit{if } P_2 = \infty \\ P_2 : \textit{if } P_1 = \infty \end{cases}$$

Case Two:

$$P_1 \boxplus P_2 = \begin{cases} \infty : \textit{if } x_1 = x_2 \textit{ and } y_1 \neq y_2 \\ \infty : \textit{if } x_1 = x_2 \textit{ and } y_1 = y_2 = 0 \end{cases}$$

Case Three:
If Case One and Two are not met, we then have:

$$P_1 \boxplus P_2 = P_3 = (x_3, y_3)$$

where

$$x_3 = -x_1 - x_2 - \bar{b}\bar{a}^{-1} + m^2\bar{a}^{-1} \tag{5}$$

$$y_3 = -y_1 + m(x_1 - x_3) \tag{6}$$

$$m = \begin{cases} (y_2 - y_1)(x_2 - x_1)^{-1} & : if\ P_1 \neq P_2 \\ (3\bar{a}{x_1}^2 + 2\bar{b}x_1 + \bar{c})\bar{2}^{-1}{y_1}^{-1} & : if\ P_1 = P_2 \end{cases} \tag{7}$$

As with the elliptic curves over $\mathbb{R}$, an abelian group can be formed by combining $\bar{E}(\mathbb{Z}_p)$ with the $\boxplus$ operator of Definition 2. Thus, as an abelian group, $(\bar{E}(\mathbb{Z}_p), \boxplus)$ has the following properties, stated without proof [Silverman, 2005, pg. 98]:

- Identity: $P \boxplus \infty = \infty \boxplus P = P$
- Inverse: $P \boxplus -P = \infty$
- Associativity: $(P \boxplus Q) \boxplus R = P \boxplus (Q \boxplus R)$
- Commutativity: $P \boxplus Q = Q \boxplus P$

for all $P, Q, R \in \bar{E}(\mathbb{Z}_p)$.

While the group $(\bar{E}(\mathbb{Z}_p), \boxplus)$ is not in general cyclic, a cyclic subgroup can be generated with the elements $\{\infty, \pm P, \pm 2P, \pm 3P, ...\}$, where P is any point on the EC.

3 The Elliptic Curve Diffie-Hellman Key Exchange

The Diffie-Hellman key exchange scheme is designed to allow two parties to exchange a key securely across insecure communication lines. This key, once transferred, would most likely be then used as the key for a symmetric encryption system. Diffie-Hellman relies on aspects of group theory, namely the Discrete Logarithm Problem (DLP), and now that we have a suitable group structure we can begin to apply our Elliptic Curves to this cryptographic scheme.

3.1 The Elliptic Curve Discrete Logarithm Problem

The generalised form of the Discrete Logarithm Problem (DLP) [Stinson, 2002, pg. 227] involves first taking a multiplicative group $(G, \cdot)$, an element $\alpha \in G$ with order n, and an element $\beta \in \langle\alpha\rangle$ (the cyclic subgroup generated by α). The problem is to try to find the unique integer a, $0 \leq a \leq n-1$, such that

$$\alpha^a = \beta. \tag{8}$$

Calculating β from α and a is computationally very easy; all it requires is the repeated application of the $\cdot$ operator. However, the reverse, calculating a from α and β, is (probably) difficult (see section 3.3 below) [Stinson, 2002, pg. 227]. If we fix α we can define the DLP in terms of the function $f(a) = \alpha^a = \beta$ (i.e. $f : \mathbb{Z} \longrightarrow G$). This function f is an example of a 'one way' function: given any x in the domain of such a function f, it is easy to compute $f(x)$, but given any y in the range of f it is computationally infeasible to calculate the inverse $x = f^{-1}(y)$ [Menezes et al., 1997, pg. 327].

The Elliptic Curve group $(\bar{E}(\mathbb{Z}_p), \boxplus)$ has the necessary properties to be used as the multiplicitive group in the Discrete Logarithm Problem. Firstly however, we must set our notation. We use additive notation for this group [Martin, 2006, pg. 12]:
If P_1 & $P_2 \in \bar{E}(\mathbb{Z}_p)$ then

- we write $P_1 \boxplus P_2$ instead of $P_1 P_2$

- for the inverse we write $-P_1$ rather than ${P_1}^{-1}$, and $-P_1$ is called the negative or additive inverse of P_1
- the identity for the group is ∞, as seen in Case One of the definition.
- if n is an integer, we write nP_1 instead of ${P_1}^n$

Also note that

- if $n \in \mathbb{N}$ then $(-nP_1)$ means $-(nP_1)$, or the additive inverse of nP_1.
- if $m \in \mathbb{N}$ then $m(nP_1) = (mn)P_1$

As a result of using the Elliptic Curve Group as the base group for the DLP, we get what is known as the Elliptic Curve Discrete Logarithm Problem (ECDLP). Given the elements $P, Q \in \bar{E}(\mathbb{Z}_p)$ such that

$$Q = nP \tag{9}$$

for some $n \in \mathbb{N}$, the problem is to find n knowing only P and Q. As with the DLP, calculating Q from n and P is a trivial matter, requiring only the repeated application of the $\boxplus$ operator. However calculating n knowing only P and Q is difficult, and it is this difficulty that gives the Diffie-Hellman scheme security.

3.2 The Elliptic Curve Diffie-Hellman Key Exchange system

Consider the hypothetical situation where Alice and Bob wish to talk over an insecure communication channel without their enemy Eve eavesdropping on their conversation. By utilising the following protocol, Alice and Bob each generate an identical key, with which they can then encrypt their conversation using a symmetric encryption system such as the Advanced Encryption Standard (AES), Twofish or Serpent.

- Alice and Bob first agree on a specific Elliptic Curve, a prime p and an element $P \in \bar{E}(\mathbb{Z}_p)$. This information is public, and does not need to be kept secret.
- Alice then chooses a secret value $a \in \mathbb{N}$, computes $A = aP$ and sends A to Bob.
- Bob also chooses a secret value $b \in \mathbb{N}$, then computes $B = bP$ and sends B to Alice.
- Alice then finds the Diffie-Hellman Key K by calculating $K = bA$, and Bob calculates $K = bA$.

Both Alice and Bob end up with the same key K because:

$$K_{Alice} = aB = a(bP) = (ab)P = b(aP) = bA = K_{Bob} \tag{10}$$

The security of the system comes from the fact that if Eve wishes to find the key K using only public information the only avenue currently open to her is to solve the ECDLP [Buchmann, 2004, pg. 189]. As shown in Equation 10 calculation of the key requires one or both of the secret numbers a and b. Since she does not have access to these, Eve would have to solve either $A = aP$, or $B = bP$ for a and b respectively, which are both forms of the ECDLP.

3.3 Breaking the Diffie-Hellman Key Exchange system

Although the focus of this report is the implementation of ECC systems rather than the cryptanalysis used to break them, it is nonetheless important to discuss briefly the potential methods of attack and their effectiveness.

Due to the dependence of ECDH on the ECDLP, the fundamental attack on this cryptographic system is to solve the ECDLP. Several methods currently exist to do this. For a general ECDLP the fastest known algorithms are the so called 'Collision Methods', such as Pollard's ρ Algorithm, which over the Elliptic Curve $\bar{E}(\mathbb{Z}_p)$ has a running time of $\mathcal{O}\left(\sqrt{p}\right)$ [Silverman, 2005, pg. 109]. (This means that for a large enough p the

maximum running time of the algorithm is $c\sqrt{p}$ where c is a positive constant [Cormen et al., 2001, pg. 44]). However, these methods are still fully exponential in the bit length l of the prime: if we write $p = 2^l$, then the running time becomes $\mathcal{O}\left(2^{l/2}\right)$, or $\mathcal{O}\left(2^{\mathcal{O}(l)}\right)$ [Hankerson et al., 2004, pg. 16]. Thus a prime number p with a suitably large bit length will ensure security from this attack: it is currently considered infeasible to solve the ECDLP for $p > 2^{160}$ [Silverman, 2005, pg. 109].

Regarding the security of the ECDLP a caveat must be added: it has not been proved that an efficient algorithm to solve the ECDLP does not exist [Hankerson et al., 2004, pg. 154]. In other words it is possible that at some time in the future an algorithm will be created that will be able to solve the ECDLP with a running time that is polynomial in l, and hence demolish ECDH and many other cryptographic systems. This issue has links to the famous P versus NP problem, a long standing open question in theoretical computer science and one of the Clay Mathematics Institute's 'Millenium Problems'. P and NP refer to two classes of problems: for those in the first, there exists a method or algorithm to find a solution in polynomial time, while those in the latter group only need to have a method of verifying a given solution in polynomial time [Odifreddi, 2004, pg. 177]. The question is whether or not a problem in the NP class is automatically also a member of the P class; i.e. if there exists a method of verifying a solution to the problem in polynomial time, does there also have to be a method of finding a solution in polynomial time? The ECDLP as stated in Section 3.1 is part of the NP class: we can easily (in time polynomial in l) verify a given solution n simply by calculating nP and checking if the result matches Q. If an algorithm was created that could find a solution in time polynomial in l then the ECDLP would also belong to the P class, proving that $P = NP$. On the other hand if it were shown that such an algorithm does not exist, then that would imply that $P \neq NP$ [Hankerson et al., 2004, pg. 154], but in either case the author would end up very wealthy.

4 Elements Necessary for Elliptic Curve Cryptography

Before we can create a functional ECDH program, we need to develop a number of MATLAB functions to perform the various operations required:

- Modular Exponentiation (`modexp.m`): This function, based on an algorithm from Garrett [2001, pg. 207-208], calculates $y = x^e \bmod m$ for large x and e.
- Multiplicitive Inverses over Finite Fields (`zmmlinv.m`): Here we use the Extended Euclidean Algorithm [Garrett, 2001, pg. 124-126] to calculate the inverse of an element a over the finite field $\mathbb{Z}_p$.
- Modular Square Roots (`sqrtmod.m`): This function uses the Shanks Algorithm to calculate the square root of a modulo p, where p is a prime - in other words it finds x such that $x^2 \equiv a \bmod p$. See Appendix A.1 for the full MATLAB code.
- Addition over an Elliptic Curve (`elcaddmod.m`): Based on Definition 2, this function finds the result of the addition of two points on an EC.
- Multiplication over an Elliptic Curve (`elcmultmod.m`): This function uses the repeated application of `elcaddmod.m` to calculate the result of the 'multiplication' of a single point on an EC.
- Elliptic Curve Populate (`ecpop.m`): This small function produces a list of all the points on a specified elliptic curve.
- Elliptic Curve Point Verification (`isecptmod.m`): This function verifies that a given point is on a specified elliptic curve, returning a true or false accordingly.

I have written MATLAB code to implement these functions, some of which is included in Appendix A.

4.1 Modular Exponentiation

One of the fundamental limitations of the MATLAB software package is its inability to deal with the large integers routinely found in number theory and cryptography. One area where this becomes apparent is modular exponentiaion, i.e. $y = x^e \bmod m$. For small x and e it is possible to calculate the result of the exponentiation first and then calculate the answer modulo m. However, for large x and e the result of x^e can easily become too large for MATLAB to store accurately (see Section 5.2). Even if the result of the modular exponentiation is below the necessary limits, the intermediate result of x^e can breach the bounds and hence introduce errors. To combat this problem the calculation of x^e must be performed in a series of steps with a modulo reduction done after each step.

The implementation set out below closely follows Garrett [2001, pg. 207-208], and is based on expressing e as a binary integer:

$$e = e_0 + e_1 2^1 + e_2 2^2 + ... + e_n 2^n$$

where $e_i \in (0, 1)$. We can then represent x^e as a composite of powers of two:

$$x^e = x^{e_0} \left(x^2\right)^{e_1} \left(x^4\right)^{e_2} \left(x^8\right)^{e_3} \left(x^{2^4}\right)^{e_4} ... \left(x^{2^n}\right)^{e_n} .$$

The powers of two can be easily calculated by sucessive squaring:

$$\begin{aligned} x^2 &= x.x \\ x^4 &= \left(x^2\right)^2 \\ x^8 &= \left(x^4\right)^2 \\ x^{2^4} &= \left(x^8\right)^2 \\ x^{2^5} &= \left(x^{2^4}\right)^2 ... \end{aligned}$$

The algorithm implementing this method of modulo arithmetic is very simple. We use three variables, X, E and Y, which are initialised to x, e and 1 respectively. Each step of the algorithm then consists of the following steps:

- If E is odd, then replace Y with $XY \bmod m$ and replace E with $E - 1$.
- If E is even, replace X with $X^2 \bmod m$ and replace E with $E/2$
- When E reaches zero the current value of Y is $x^e \bmod m$.

In MATLAB this is performed using the following code:

```
X=x;
E=e;
Y=1;
while E ~ = 0
     iseven(E)
         X = mod((X^ 2), m);
         E = E/2;
     elseif ~iseven(E)
         Y = mod((X*Y), m);
         E = E-1;
     end
end
```

4.2 Calculating Multiplicative Inverses over Finite Fields

When working with Elliptic Curves over the real numbers the calculation of inverses is simple: we just to performing the calculation $a^{-1} = \frac{1}{a}$. However, ECC is instead performed over the finite field made up of the integers modulo p. This complicates the process of calculating inverses somewhat; instead of calculating a simple fraction, calculating the inverse of a now means finding the element b of the finite field such that $a.b \equiv 1 \bmod p$.

Finding the inverse of an element a in $\mathbb{Z}_p$ is not as simple as it was for the real numbers, where calculating the fraction $\frac{1}{a}$ would be sufficient. It is possible to employ a 'brute force' checking method, where all elements of the finite field are checked until one element b satisfies the necessary condition that $a.b \equiv 1 \bmod p$. However this would be extremely laborious for large fields, so a more efficient method is required. One such method is the Extended Euclidean Algorithm, laid out in detail in Garrett [2001, pg. 124-126]. In the course of this project I implemented this algorithm in MATLAB.

4.3 The Shanks Algorithm and Modular Square Roots

To perform ECC, we need to be able to find points that lie on a particular elliptic curve. One approach would be to generate random x and y values, and check whether or not they satisfy the elliptic curve equation. However, a better method is to instead calculate random x values, then calculate the RHS of Equation 1, giving us a y^2 value from which we can calculate the necessary y value if one exists. Although vastly more efficient, this method requires the calculation of modular square roots. This is the purpose of the Shanks Algorithm, also known as the Shanks-Tonelli Algorithm.

Given an element a of the finite field, the square root of a is defined as the element x such that

$$x^2 \equiv a \bmod p$$

If a has such a square root, it is called a quadratic residue. Those elements which do not have a square root are called quadratic non-residues [Garrett, 2001, pg. 231]. In $\mathbb{Z}_p$ $\frac{p-1}{2}$ elements will be quadratic residues and and equal number will be quadratic non-residues [Lauritzen, 2003, pg. 37].

Before analysing the Shanks Algorithm, we must first lay out a few definitions and ideas:

- Let a be input to the square root function; i.e. we are attempting to find x such that $x^2 = a$.
- Let p be an odd prime number, with a and p relatively prime.
- The *order* of an element a of the finite field integers modulo p, writen $ord_p(a)$ is the smallest integer j such that

  $$a^j \equiv 1 \bmod p\,.$$

  [Garrett, 2001, pg. 137]
- If integer r divides integer q we write $r \mid q$ [Garrett, 2001, pg. 62].
- *Euler's Criterion* [Garrett, 2001, pg. 232]

  $$a^{\frac{p-1}{2}} = \begin{cases} 1 : \text{if } a \text{ has a square root} \\ -1 : \text{if } a \text{ does not have a square root} \end{cases}$$
- *Lagrange's Theorem for Groups*: If $a^q \equiv 1 \bmod p$ then $ord_p(a) \mid q$ [Garrett, 2001, pg. 269].
-

Lemma 1. *If* $y^2 \equiv 1 \bmod p$, *then* $y \equiv \pm 1 \bmod p$.

Proof.

$$\begin{aligned} y^2 &\equiv 1 \bmod p \\ y^2 - 1 &\equiv 0 \bmod p \\ \therefore p &\mid (y^2 - 1) \\ p &\mid (y+1)(y-1) \end{aligned}$$

if a prime divides a product, then the prime must also divide one of the factors, so

$$\begin{aligned} p &\mid (y \pm 1) \\ y \pm 1 &\equiv 0 \bmod p \\ \therefore y &\equiv \pm 1 \bmod p \end{aligned}$$

□

The general idea of the algorithm is to first calculate a likely candidate for the square root then check its validity. If it is not the correct answer we alter it by a 'fudge factor', and check it again. This fudge factor is updated with every cycle of the algorithm, and we are guaranteed to eventually find the correct square root. The structure of the Shanks Algorithm has been sourced from Brown [1999, pg 90-94], but much of the analysis performed below was skipped in his paper, such as lemmas 2 and 5.

We now analyse each step of the algorithm.

Step One
We first must obtain and check the input variables. The inputs are the integer a and the prime number p. A check is performed to determine if a and p are relatively prime.

Step Two
We then Check Euler's Criterion. $a^{\frac{p-1}{2}} \bmod p$ is calculated and if the result is equal to -1 then a has no quadratic residue, and the algorithm is forced to terminate. If, however, the result is equal to 1, a does indeed have a quadratic residue, and the algorithm can continue.

Step Three
We then have to find an element n of the finite field that is a quadratic non-residue; i.e. $n^{\frac{p-1}{2}} \equiv -1 \bmod p$. This is always possible as $\frac{p-1}{2}$ elements of the field will be quadratic non-residues. Several methods can be used to find n. A random search will eventually yield a quadratic non-residue as will a sequential search from the first element of the field. Each method has its relative advantages. A random search could potentially yield an answer on the first couple of tries, but could also theoretically search through all $\frac{p-1}{2}$ quadratic residues before finding a non-residue. The sequential search is however guaranteed to find a non-residue with far fewer than $\frac{p-1}{2}$ steps; if the extended Riemann hypothesis is correct, then the first quadratic non-residue will always be less than $\frac{3(\ln p)^2}{2}$ [Wedeniwski, 2001, pg. 122].

Step Four
Taking the prime number p, we factorise $p-1$ so that

$$p - 1 = s.2^e$$

where s is odd and e is greater than zero. This is performed by the following MATLAB code:

```
s = 0;
e = 0;
while rem(s,2) ~=1
     e = e + 1;
     s=(p-1)/(2^ e);
end
```

Step Five
We now initialise the variables x, b, g and r. Note that all variables for the rest of this section are calculated mod$p$.

$$\begin{aligned} x_0 &= a^{\frac{s+1}{2}} \\ b_0 &= a^s \\ g_0 &= n^s \\ r_0 &= e \end{aligned}$$

x_0 is our first guess at the square root - it will become apparent later why this is a good choice. b_0 will form the basis of our 'fudge factor' if it is needed further on in the algorithm.

Also note the following result:

$$\begin{aligned} x_0 &= a^{\frac{s+1}{2}} \\ {x_0}^2 &= a^{s+1} \\ &= a^s a \\ &= b_0 a \end{aligned}$$

Step Six
We now turn our attention to the variable b, and more specifically, its order.

$$\begin{aligned} {b_0}^{2^{r-1}} &= a^{s.2^{r-1}} \\ &= a^{\frac{s.2^r}{2}} \\ &= a^{\frac{p-1}{2}} \\ &= 1 \quad \text{by Euler's Criterion} \end{aligned}$$

by Euler's Criterion. Hence by employing Lagrange's Theorem we can say that $ord_p(b) \mid 2^{r-1}$. We can therefore set $ord_p(b) = 2^m$, where m is an integer such that $0 \leq m \leq r-1$. The aim of Step Six is to find this integer m. This is done by a simple sequential search, starting at $m = 0$ and calculating b^{2^m} for each value of m until we find one for which $b^{2^m} \equiv 1$.

Step Seven
Depending on the value of m, we enter one of two cases:

Case One: $m_0 = 0$
If $m_0 = 0$ then the algorithm has succedded and is terminated. For reasons that will be laid out below, x_0 is the square root of a.

Case Two: $m_0 \neq 0$
If, on the other hand, $m_0 \neq 0$, then we update the variables:

$$\begin{aligned} x_{i+1} &= x_i {g_i}^{2^{r_i - m_i - 1}} \\ b_{i+1} &= b_i {g_i}^{2^{r_i - m_i}} \\ g_{i+1} &= {g_i}^{2^{r_i - m_i}} \\ r_{i+1} &= m_i \end{aligned}$$

Once the variables have been updated, we return to Step Six of the Algorithm, and try to find the integer m_{i+1}. □

It is here that we find the main loop of the algorithm, and several questions must be answered. Will the loop terminate, or is it possible that the algorithm will continue indefinitely? If the loop does terminate, is the calculated answer, the variable x, the square root of a?

Before continuing, we shall state and prove two lemmas regarding the variables g and b that will be utilised further on.

Lemma 2. ${g_i}^{2^{r_i - 1}} \equiv -1$

Proof. We perform a proof by induction. For $i = 0$ we have:

$$\begin{aligned} g_0 &= n^s \\ {g_0}^{2^{r_0 - 1}} &= n^{s 2^{r_0 - 1}} \\ &= n^{s 2^{e-1}} \\ &= n^{\frac{s 2^e}{2}} \\ &= n^{\frac{p-1}{2}} \\ &\equiv -1 \end{aligned}$$

as n is a quadratic non-residue.

Assuming it holds for $i = n$, i.e. ${g_n}^{2^{r_n - 1}} \equiv -1$, we attempt to prove the $i = n + 1$ case:

$$\begin{aligned} g_{n+1} &= g_n^{2^{r_n - m_n}} \\ {g_{n+1}}^{2^{r_{n+1} - 1}} &= g_n^{2^{r_n - m_n + r_{n+1} - 1}} \\ \text{but} \quad r_{n+1} &= m_n \\ {g_{n+1}}^{2^{r_{n+1} - 1}} &= g_n^{2^{r_n - 1}} \\ {g_{n+1}}^{2^{r_{n+1} - 1}} &\equiv -1 \end{aligned}$$

□

A similar result is possible for the variable b:

Lemma 3. ${b_i}^{2^{m_i-1}} \equiv -1$

Proof.

$$\begin{aligned}
\text{Let} \quad & y = {b_i}^{2^{m_i-1}} \\
& y^2 = {b_i}^{2^{m_i}} \\
& y^2 \equiv 1 \\
& \therefore y \equiv \pm 1 \quad \text{by Lemma 1} \\
\text{but} \quad & y \not\equiv 1 \\
\text{as} \quad & ord_p(b_i) = 2^{m_i} \neq 2^{m_i-1} \\
& \therefore y \equiv -1 \\
& {b_i}^{2^{m_i-1}} \equiv -1
\end{aligned}$$

□

We now introduce a lemma that will answer the first question we asked regarding the termination of the algorithm.

Lemma 4. *If the order of b_i is 2^{m_i}, then the order of b_{i+1} is at most 2^{m_i-1}.*

Proof.

$$\begin{aligned}
b_{i+1} &= b_i {g_i}^{2^{r_i-m_i}} \\
{(b_{i+1})}^{2^{m_i-1}} &= \left(b_i {g_i}^{2^{r_i-m_i}}\right)^{2^{m_i-1}} \\
&= {b_i}^{2^{m_i-1}} {g_i}^{2^{r_i-m_i+m_i-1}} \\
&= {b_i}^{2^{m_i-1}} {g_i}^{2^{r_i-1}} \\
&\equiv -1. -1 \qquad \text{by Lemmas 2 \& 3} \\
\therefore &\equiv 1
\end{aligned}$$

Thus by Lagrange's Theorem $ord_p(b_{i+1}) \leq 2^{m-1}$, and our Lemma is proved. This guarantees that the order of the variable b will decrease through successive steps, and thus ensure that the algorithm will eventually terminate. □

The second question we asked about the algorithm was if it indeed terminated, was the final x value the correct square root of a? We perform an analysis of the general case, where i is any non-negative integer.

It was noted above that ${x_0}^2 \equiv b_0 a \bmod p$, but this result can be generalised.

Lemma 5. $x_i \equiv b_i a$

Proof. This is proved using induction. We already have the proof for i=0, so we assume that it holds for $i = n$:

$$x_n \equiv b_n a$$

We then test the $i = n + 1$ case:

$$\begin{aligned} x_{n+1} &= x_n {g_n}^{2^{r_n - m_n - 1}} \\ {x_{n+1}}^2 &= {x_n}^2 {g_n}^{2^{r_n - m_n}} \\ &\equiv b_n a {g_n}^{2^{r_n - m_n}} \\ &\equiv b_n {g_n}^{2^{r_n - m_n}} a \\ &\equiv b_{n+1} a \end{aligned}$$

□

So if on the i^{th} iteration of the algorithm we find that the integer $m_i = 0$, the loop will terminate. Is x_i the correct square root of a?

First note that ${b_i}^{2^{m_i}} \equiv b_i \equiv 1 \quad \text{as } 2^{m_i} = 1$. So from Lemma 5 we have

$$\begin{aligned} {x_i}^2 &\equiv b_i a \\ &\equiv 1.a \\ &\equiv a \end{aligned}$$

Thus the algorithm produces the correct square root of a.

4.4 Implementation of the Addition Law over $\bar{E}(\mathbb{Z}_p)$

The primary function used in Elliptic Curve encryption schemes is $\boxplus$, as described above in 2.2. Thus we require a MATLAB implementation of this, and also other functions in order to verify its efficacy. The functions fulfilling this need are `elcmultmod.m` and `elcaddmod.m`. The latter calculates $P_3 = P_1 \boxplus P_2$, while the former utilises the latter to calculate $Q_2 = nQ_1$, or $Q_2 = Q_1 \boxplus Q_1 \boxplus ... \boxplus Q_1$ n times.

4.4.1 Elliptic Curve Addition: `elcaddmod.m`

The inputs to this functiona are the x and y coordinates of two points, P_1 and P_2, the coefficients a, b, c and d that define the elliptic curve, and the prime that defines the finite field over which we are working. The ∞ point that was introduced in section 2.2 is represented by using the string `'I'` as the point's x and y coordinates. For example, the following command would add the point $(x, y) = (19, 21)$ to the ∞ point:

```
[x3, y3] = elcaddmod(19, 21, 'I', 'I', 12, 8, 9, 15, 29) .
```

The structure of the program is very similar to that of Definition 2; we test the x and y coordinates of the two points to determine which case they fit into, and calculate an answer accordingly. Case One simply involves checking if one of the points is ∞, and if so, setting the answer as equal to the other point.

Case Two deals with situations where the x coordinates are equal, and the y coordinates are either different or both equal to zero. If this arises, the output is set to the ∞ point.

Finally, if Cases One and Two are not satisfied, and error checking has not revealed any input errors, the program will enter Case Three. Here we first calculate m. When working with the Elliptic curve over $\mathbb{R}$ this quantity is the gradient of the line connecting the two points, and is calculated accordingly. The resultant formulas are easily transfered to the elliptic curve over the finite field $\mathbb{F}$, with each variable and parameter restricted to the elements of $\mathbb{F}$, and inverses being calculated using the function discussed in section 4.2

Once the correct m value has been calculated, it is then a simple matter to calculate the x and y coordinates of the resulting point, using Equations 5 and 6.

4.4.2 Elliptic Curve Multiplication: `elcmultmod.m`

One of the primary calculations used in the ECDH key exchange is what could be called multiplication in the additive notation we are using; i.e. $Q_2 = nQ_1$. This involves 'adding' Q_1 to itself n times. Althought this function is little more that repeated usage of `elcaddmod.m`, it is convenient to place it into a single function as it is used repeatedly in the implementation of the Key Exchange.

4.5 Verification of the Addition Law Functions

Before applying them to any cryptographic implementation it was necessary to test the Addition Law functions `elcaddmod.m` and `elcmultmod.m`. Several functions were writen to do this.

4.5.1 Elliptic Curve Populate: `ecpop.m`

The purpose of this function is to generate a list of all the points on a finite field Elliptic Curve. For example, the EC defined by $(a, b, c, d) = (4, 14, 9, 17)$ with $p = 19$ was populated with this function, yielding the following list of points:

x	y	y
0	13	6
1	14	5
2	3	16
4	18	1
5	0	0
9	14	5
10	15	4
11	3	16
12	3	16
15	14	5
17	2	17

This list and others like it allow us to confirm that the function `elcaddmod.m` is preserving the closure of the group - if two points from the list are inputted to the function the output should also be on the list. For testing purposes it was only possible to use ECs over small finite fields, with prime values ranging up to 19. Using this method it was determined that the implementations of the Addition Law were indeed correct.

4.5.2 Elliptic Curve Point Verify: `isecptmod.m`

This function, given a point, will determine whether the point lies on a specific elliptic curve, returning true and false (1 and -1) results accordingly. The true/false output of the function means that the function can be easily integrated into other functions as a form of error detection, as was done with `elcmultmod`.

Having, in effect, an integrated verification system, `elcmultmod` was easily debugged and verified.

5 Implementation of Elliptic Curve Cryptography

5.1 MATLAB implementation of Elliptic Curve Diffie-Hellman

In section 3 the theoretical underpinnings of ECDH were introduced, and in section 4 the necessary tools to implement this system were developed. We now put these two together to produce a working ECDH key exchange system. Note that the program has been written from the perspective of Alice, with the variable names corresponding to her part in the system. The full MATLAB program is listed in Appendix A.2

The first step in the process is gathering the required data. This consists of the coefficients a, b, c and d that determine the Elliptic Curve, the prime p that specifies the finite field, and a point P on this curve. All this information is public, and is to be agreed upon by Alice and Bob. At this stage the program checks that the inputted p is indeed prime and, using `isecptmod`, that the given point P lies on the Elliptic Curve. Alice and Bob then have to each choose a secret number, and this too is inputted into the program.

The first of the main calculations can now take place. By using `elcmultmod.m` the program calculates $A = aP$, and outputs it to the console. Alice can then send this number to Bob.

Once Alice receives B from Bob, she inputs it into the console. The program first checks for any transmission errors by verifying that the point lies on the elliptic curve, and then calculates the key $K = aB$. The key is then displayed on the console, and the program is complete.

This implementation of ECDH has has been successfully tested with primes ranging up to $p = 19999423$, or roughly 25 bits. While this is nothing near the level of security required in real world circumstances, it is limited by certain aspects of MATLAB which will be discussed further in Section 5.2.

To aid in the use of this program, a function has been created to locate a random point on the Elliptic Curve to be used as the puplic point P. This is performed by first generating a random $x \in \mathbb{Z}_p$, then calculating the corresponding y value using `sqrtmod.m`. If no such y value exists, another x is generated, and the process repeated until one can be found.

5.2 Limitations of MATLAB

As metioned earlier MATLAB is incapable of accurately dealing with the large integers that we encounter in cryptography. The problems this creates can sometimes be overcome, often through the use of alternative algorithms like that used for modular exponentiation above. However, this is not always possible; beyond a certain point even basic operations like squaring creates numbers that are too big for MATLAB to properly store.

The fundamental limitation is a result of the way MATLAB stores numbers. The double precision IEEE floating point form that MATLAB normally uses can accurately store integers up to 4,503,599,627,370,495 (52 bits). Any number that excedes this limit will have some of its least significant digits rounded out of existence, and hence loose accuracy. It is possible to store numbers as 64 bit unsigned integers, increasing the limit to 18,446,744,073,709,551,615, but MATLAB cannot perform arithmetic on numbers in such a form. Future versions of MATLAB that incorporate a proposed new IEEE standard will probably be able to accurately store integers with a bit length roughly double the current maximum, but even then it will still be unable to properly deal with the 160 bit numbers required to maintain security.

5.3 Elliptic Curve Cryptography in the Real World

Given the limitations of MATLAB discussed above, how is ECC implemented in the real world? The answer lies in the concept of arbitrary precision arithmetic, which allows calculations to be performed on integers of any length. There does not appear to be any packages to implement arbitrary precision arithmetic in MATLAB, but there are several such packages available for other languages like C/C++ and Java. Hence it is in these languages that real world ECC systems are created.

In 2005 the US National Security Agency (NSA) released Suite B, a group of five cryptographic algorithms that will over the next decade replace those currently in use by the US Government and Military. Three of the five algorithms are based on Elliptic Curves: ECDH, Elliptic Curve Menezes-Qu-Vanstone (ECMQV), another key exchange system, and Elliptic Curve Digital Signiture Algorithm (ECDSA) [NSA, a]. The reason for this is the efficiency that EC cryptography offers compared to more traditional non-EC based schemes. As mentioned in Section 3.3 the ECDH system is considered secure when a prime larger than 2^{160} is used, but in order to gain a similar level of security from the older RSA system it is necessary to use primes larger than 2^{1000} [Silverman, 2005, pg. 109]. Despite the extra complications involved in carrying out the $\boxplus$ operation compared to the modular multiplication used in RSA, the dramatic reduction in the size of the stored integers makes ECDH and other EC based systems faster and more efficient.

In Section 3.3 it was stated that for a general ECDLP the fastest known algorithms had running times of $\mathcal{O}\left(\sqrt{p}\right)$. However in certain special cases there are algorithms that have significantly faster running times, making ECDH insecure even for sufficiently large primes. In any practial implementation of ECDH and EC cryptography schemes in general it is necessary to avoid these special cases. To this end the US National Institute of Standards and Technology have published a list of suitable elliptic curves and their corresponding finite fields in Appendix Six of NIS [2000].

6 Conclusion

Elliptic Curves offer a rich variety of behaviour which we can utilize for the purposes of cryptography. As shown in Section 2 we can develop an abelian group structure that is then used as the basis for cryptographic schemes such as the ECDH key exchange. The security of these schemes relies on the difficulty of solving the ECDLP, as introduced in Section 3.1.

Over the course of this project I have created a limited but functional implementation of the ECDH key exchange system, using the MATLAB software package. This program, listed below in Appendix A.2, allows two parties, Alice and Bob, to generate identical keys with which to then symmetrically encrypt a conversation. If their enemy Eve wishes to eavesdrop on the conversation she must first break the ECDLP. For the relatively small numbers used here this could be done easily, but for real-world programs using very large numbers this would be computationally infeasible.

The limited nature of the implementation is due to the method by which MATLAB stores and works with large integers, and the apparent absence of any package that would allow MATLAB to deal with them differently. However, looking back at the work completed, we can see that many of the required algorithms and methods already developed and discussed are independent of the language they are eventually implemented in. Thus if we wanted to create a more effective program capable of handling the large numbers that would ensure security we merely have to 'translate' our current methods into a superior language such as C/C++, instead of starting again from scratch.

References

Ezra Brown. Square roots from 1; 24, 51, 10 to Dan Shanks. *The College Mathematics Journal*, 30(2):82–95, Mar 1999.

Johannes A. Buchmann. *Introduction to Cryptography.* Springer, 2nd edition, 2004.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 2nd edition, 2001.

Paul Garrett. *Making, Breaking Codes.* Prentice Hall, 2001.

Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography.* Springer, 1st edition, 2004.

Niels Lauritzen. *Concrete Abstract Algebra: From numbers to Gröbner Bases.* Cambridge University Press, 2003.

Ben Martin. Elliptic curve cryptography. Math 409 Notes, University of Canterbury, 2006.

Alfred J. Menezes, Paul C. van Oorschot, and Scot A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.

Federal Information Processing Standards Publication 186: Digital Signature Standard. NIST (National Institute of Standards and Technology), Jan 2000.

Fact Sheet: NSA Suite B Cryptography. NSA (National Security Agency), a. `http://www.nsa.gov/ia/industry/crypto_suite_b.cfm` (Accessed 7 February 2007).

Fact Sheet: The Case for Elliptic Curve Cryptography. NSA (National Security Agency), b. `http://www.nsa.gov/ia/industry/crypto_elliptic_curve.cfm` (Accessed 7 February 2007).

Piergiorgio Odifreddi. *The Mathematical Century: The 30 Greatest Problems of the Last 100 Years.* Princeton University Press, 2004.

Joseph H. Silverman. Elliptic curves and cryptography. In Paul Garrett and Daniel Lieman, editors, *Public-Key Cryptography.* American Mathematical Society, 2005.

Douglas R. Stinson. *Cryptography: Theory and Practice.* Chapman & Hall/CRC, 2002.

F. Vercauteren. Elliptic curve discrete logarithm problem. Lecture Slides, Katholieke Universiteit Leuven, 2005.

Sebastian Wedeniwski. *Primality Tests of Commutator Curves.* PhD thesis, Eberhard Karls University of Tübingen, 2001.

A MATLAB .m Files

A.1 sqrtmod.m

```
function X=sqrtmod(a, p)
%SQRTMOD square root modulo prime p
% function sqrtmod finds the square root of integer a modulo p if it exists
%
%   Date          Programmer              Description of Change
%   ~~~~          ~~~~~~~~~~              ~~~~~~~~~~~~~~~~~~~~~
%   15/12/06      H. Silverwood           Original Code
%   20/12/06      H. Silverwood           Modified to include fast modular
%                                           exponentiation
%
% Calling Sequence:
%   X=sqrtmod(a,p)
%
% Inputs:
%   a   =   integer
%   p   =   prime number, >2 & relatively prime to a
%
% Outputs:
%   X   =   square root of a
%       =   'E' if there is no square root
%

%Check conditions on p
%---------------------------------------%
%Check that prime is greater than 2
if p<=2
    error('p must be greater than 2: p>2')
end

%Check relative primality of a and p
if gcd(a,p)~=1
    X='E'; %a and p not relatively prime
    return
end
%---------------------------------------%

%Check Euler's Criterion
%---------------------------------------%
Eu=modexp(a, (p-1)/2, p);
if Eu==(p-1)
    X='E'; %Square root does not exist
    return
elseif Eu~=1
    error('Eu does not equal 1 or negative 1')
end
%---------------------------------------%

%Factorise Prime Number
```

```
%---------------------------------------%
s=0;
e=0;
while rem(s,2)~=1
    e=e+1;
    s=(p-1)/(2^e);
end
%---------------------------------------%

%Find suitable n value
%---------------------------------------%
n=1;
nres=0;
while nres~=(p-1)
    n=n+1;
    nres=modexp(n, (p-1)/2, p);
end
%---------------------------------------%

%Perform Variable Initialization
%---------------------------------------%
x=modexp(a, (s+1)/2, p);
b=modexp(a, s, p);
g=modexp(n, s, p);
r=e;
%---------------------------------------%

%Primary Calculations
%---------------------------------------%
m=1; %Initialise m
while m~=0
    %Reset m
    m=0;
    %Find m
    bres=b;
    while bres~=1
        bres=mod(bres^(2),p);
        m=m+1;
    end

    %Replace Variables
    xtemp=modexp(2, (r-m-1), p);
    x=mod(x*(modexp(g, xtemp, p)), p);
    temp=modexp(2, (r-m), p);
    gtemp=modexp(g, temp, p);
    b=mod(b*gtemp, p);
    g=gtemp;
    r=mod(m, p);
end

X=x;
```

```
return
%--------------------------------------%
```

A.2 DiffieHellman.m

```
%Script file: Diffie-Hellman Key Exchange
%
%Purpose: To implement the Elliptic Curve Diffie-Hellman Key Exchange system
%
%   Date          Programmer              Description of Change
%   ~~~~          ~~~~~~~~~~              ~~~~~~~~~~~~~~~~~~~~~
%   10/01/07      H. Silverwood           Original Code
%
% Inputs:
%   a,b,c,d =   Elliptic Curve parameters
%   p       =   prime modulus of EC
%   P       =   Coordinates of point on Elliptic Curve [Px Py]
%   a       =   natural number (secret)
%
% Outputs:
%   A       =   aP (sent to counterpart)
%   K       =   common key
%

%Display Header
disp('^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^')
disp('   Elliptic Curve Diffie-Hellman Key Exchange')
disp(' ')
disp('This program implements the Elliptic Curve ')
disp('Diffie-Hellman Key Exchange system with sub')
disp('64-bit numbers.')
disp(' ')
%Enter Public Information
disp('Enter public information:')
disp(' ')
ECp=input('  EC parameters [a b c d] = ');
disp(' ')
p=input('  Prime number p = ');
    %Check Primality
    if isprime(p)==0
       error('Input is not prime')
    end

disp(' ')
P=input('  Point P = [Px Py] = ');
Px=P(1);
Py=P(2);

    %Verify that point P is on the EC
    PECflag=isecptmod(Px, Py, ECp(1), ECp(2), ECp(3), ECp(4), p);
    if PECflag==0
        error('Point P does not lie on the specified Elliptic Curve')
```

```
    end

disp(' ')
disp('Enter Private Information:')
a=input('   Natural Number a = ');

%Perform A=aP calculations
[Ax, Ay]=elcmultmod(a, Px, Py, ECp(1), ECp(2), ECp(3), ECp(4), p);
A=[Ax, Ay];
disp('Send the following EC point to counterpart')
fprintf('   A = [%3d  %3d  ]\n\n', Ax, Ay)

%Enter Recieved data
disp('Enter EC point recieved from counterpart')
B=input('   Point B = [Bx By] = ');
Bx=B(1);
By=B(2);

    %Verify that point B is on the EC
    BECflag=isecptmod(Bx, By, ECp(1), ECp(2), ECp(3), ECp(4), p);
    while ~BECflag
        disp('Point B does not lie on the specified EC;')
        disp('Enter new Point B value:')
        B=input('   Point B = [Bx By] = ');
        Bx=B(1);
        By=B(2);
        BECflag=isecptmod(Bx, By, ECp(1), ECp(2), ECp(3), ECp(4), p);
    end

%Calculate Key K
[Kx, Ky]=elcmultmod(a, Bx, By, ECp(1), ECp(2), ECp(3), ECp(4), p);
K=[Kx, Ky];

%Output Key K
fprintf('The key is K = [%3d  %3d  ]\n\n', Kx, Ky)
disp('^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^')
```