CHAPTER 5

Elliptic curves

5.1 Mathematics of elliptic curves over real numbers 104
5.2 Mathematics of elliptic curves over prime fields 108
5.3 Mathematics of elliptic curves over Galois Fields 109
5.4 Polynomial basis elliptic curve subroutines 114
5.5 Optimal normal basis elliptic curve subroutines 118
5.6 Multiplication over elliptic curves 120
5.7 Balanced integer conversion code 122
5.8 Following the balanced representation 125
5.9 References 126

The mathematics associated with elliptic curves is old. Formerly “pure” with no practical applications, the math is very deep and fascinating. We will barely skim the surface in this book of what is a very beautiful and fruitful line of research, even today.

The use of elliptic curves for cryptography is not the same as their use for factoring or for solving Fermat’s Theorem, as was announced a few years ago. We are going to use elliptic curves as an “algebra,” or higher-level abstraction over all the math discussed previously. Chill out—it’s not as scary as it sounds.

The magic of elliptic curves comes from the ability to take any two points on a specific curve, add them together, and get another point on the same curve. More importantly for cryptography is the difficulty of figuring out which two points were added together to get there. For the right choice of various parameters, that difficulty is exponential with key length. While the cryptanalyst must use very advanced mathematics to even begin attempting to crack a code, it does not take very many bits before the task is practically impossible. We will get into more details on that later.

5.1 Mathematics of elliptic curves over real numbers

First, let’s start with what mathematicians call the “Weierstrass” form of an elliptic curve equation [1, chapter 3; 2, 15]:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 . \tag{5.1}$$

The variables x and y cover a plane. In fact, x and y can be complex, real, integers, polynomial basis, optimal normal basis, or any other kind of field element. That’s part of what makes the math so deep.

Because we are only interested in a few special cases of equation (5.1), we’ll stick to some really simple aspects of the math. The interested reader should seek out [1] or [3] to get a better background.

Let’s start with something familiar: real numbers on the real plane. A simple form of equation (5.1), which will work for us, is:

$$y^2 = x^3 + a_4x + a_6 . \tag{5.2}$$

As an example, let’s plot the curve for $a_4 = -7$, $a_6 = 5$ for x and y in the set of real numbers. To do this we only need to find:

$$y = \sqrt{x^3 - 7x + 5} \tag{5.3}$$

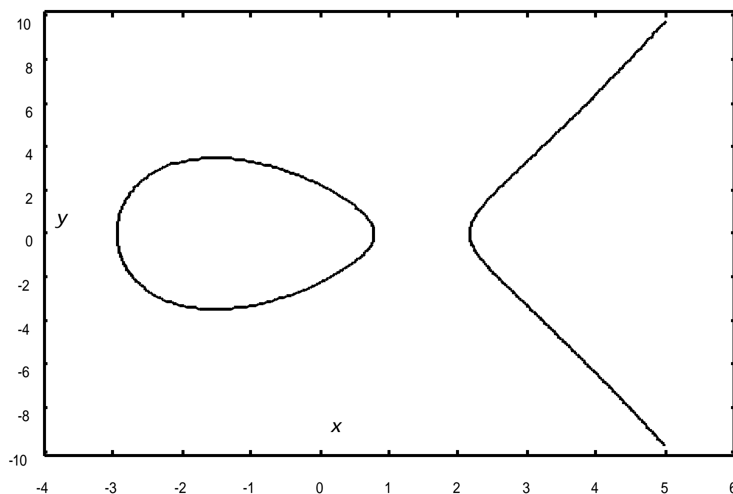

Figure 5.1 Plot of elliptic curve $y^2 = x^3 - 7x + 5$

and plot both negative and positive values of y for the same x. (See figure 5.1.)

This curve looks neat, but how do you create an algebra from something like that? The basic idea is to find a way to define "addition" of two points that lie on the curve such that the "sum" is another point on the curve. If we can do that, and invent an identity element, then we have an algebra—that is, a higher level of abstraction but following the same basic rules of math we're used to.

The identity element $\mathbf{O}_\infty$ is the point that, added to any point on the curve, gives the same point back:

$$P + \mathbf{O}_\infty = P. \tag{5.4}$$

It is also called "the point at infinity." Under normal conditions we'll never use this point in real code. The formulas to be presented later won't work if $\mathbf{O}_\infty$ is an input. However, we can still see if we are about to hit the identity element because:

$$-P = \mathbf{O}_\infty - (P). \tag{5.5}$$

To understand why mathematicians write down obvious equations like that, let's take a point $P = (x,y)$. The formula for finding $-P$, for the real valued equation (5.2), is:

$$-P = (x, -y). \tag{5.6}$$

Now, let's look at our previous example and see where the points P and $-P$ lie on our curve. (See figure 5.2.)

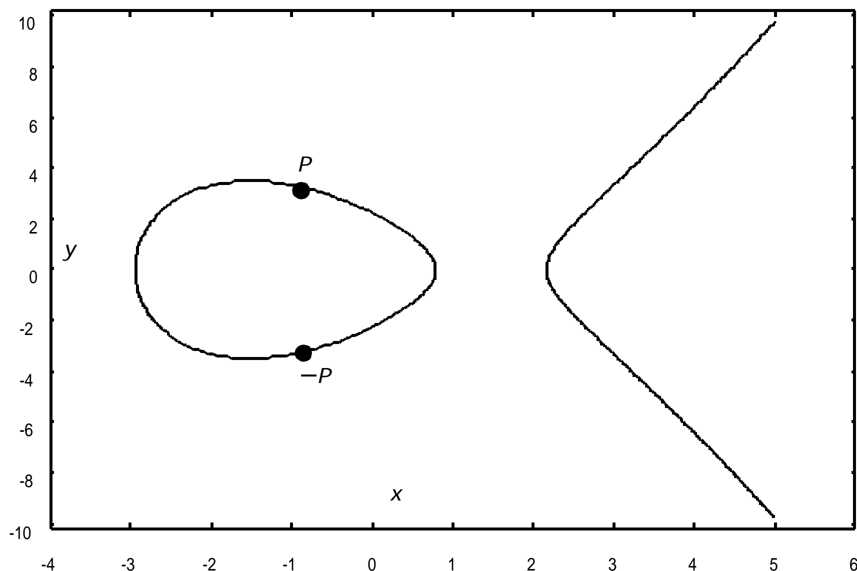

Figure 5.2 Arbitrary points P and $-P$ on example curve

We see that the two points lie on top of each other. Now, let's do another obvious thing; we'll add:

$$P + (-P) = \mathbf{O}_{\infty} \ . \tag{5.7}$$

If we draw a line between P and $-P$, the next point on the curve it hits is "the point at infinity." In fact, we can define the addition of any two points on an elliptic curve by drawing a line between the two points and finding the point at which the line intersects the curve.

For the math to work, the negative of the intersection point is defined as the "elliptic sum." (See figure 5.3.) Mathematically we write:

$$R = P + Q. \tag{5.8}$$

There is a direct geometric relationship between every point on the curve. It is a good homework problem to derive the relationship, and it has been done elsewhere [3]. The rules are as follows:

Given:

$$P = (x_1, y_1) \tag{5.9}$$

$$Q = (x_2, y_2) \tag{5.10}$$

then:

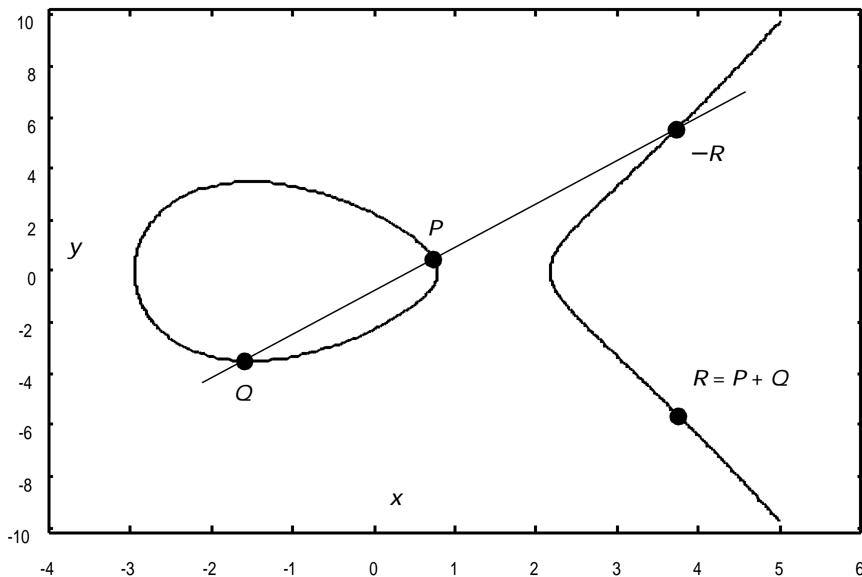

Figure 5.3 Addition of elliptic curve points over a real number curve

$$R = P + Q = (x_3, y_3) \tag{5.11}$$

where

$$x_3 = \theta^2 - x_1 - x_2 \tag{5.12}$$

$$y_3 = \theta(x_1 + x_3) - y_1 \tag{5.13}$$

$$\theta = \frac{y_2 - y_1}{x_2 - x_1} \quad \text{if} \quad P \neq Q \tag{5.14}$$

or

$$\theta = \frac{3x_1^2 + a_4}{2y_1} \quad \text{if} \quad P = Q. \tag{5.15}$$

Adding a point to itself is a special case; the line used is the tangent to the curve at P. This is shown in figure 5.4. There is a lot of algebra required to derive the above equations from the figures. You can cheat and use relationships between the roots of cubic equations, which are found in many reference books, and save most of the work (see [4, 17] and [5, chapter 6]).

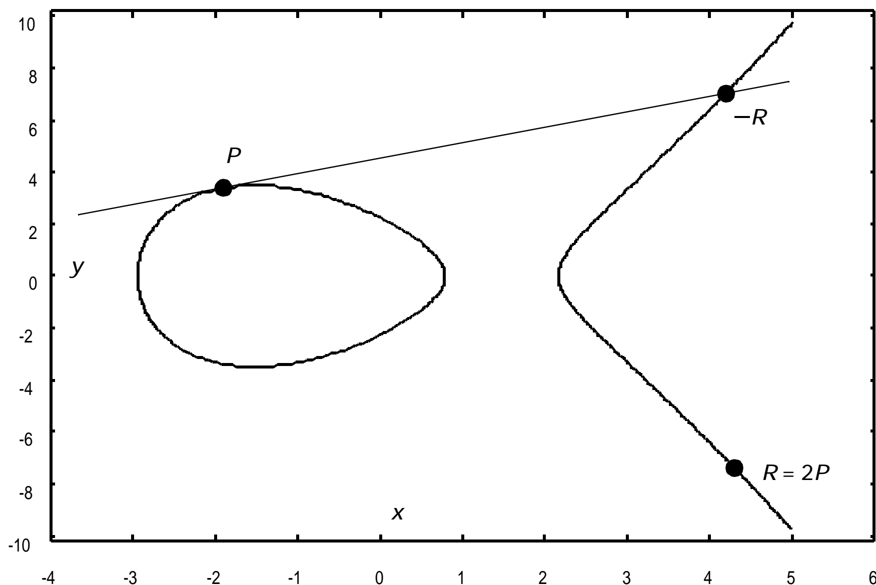

Figure 5.4 Addition of a point to itself, also called "doubling"

5.2 Mathematics of elliptic curves over prime fields

Section 5.1 is interesting, but what does it have to do with cryptography? It turns out that similar formulas work if we replace real numbers with finite fields. As a first attempt, let's look at finite fields generated using large primes (see [6] for more details).

The formulas stated previously don't change. But instead of using floating-point arithmetic we use a large number package and do all our calculations modulo a large prime. This method has been investigated by the academic community and is mentioned in the IEEE P1363 draft crypto standard. (See [6–8] for more details.)

The major advantage of choosing this for computing elliptic curves is that the number of points on the curve can be computed easily. We'll see why this is important later. For now, we'll define yet another new term: "the order of an elliptic curve." Since the math of a finite field only covers a finite set of points, the total number of points on an elliptic curve defined over a finite field is fixed. This number is sometimes called the "cardinality" of an elliptic curve.

The number of points on an elliptic curve over a finite field must satisfy Hasse's Theorem. Given a field, F_q, the order of the curve (N) will satisfy this equation [2, 5]:

$$|N-(q+1)| \le 2\sqrt{q}. \tag{5.16}$$

Another way to put it is:

$$q+1-2\sqrt{q} \le N \le q+1+2\sqrt{q}. \tag{5.17}$$

So the number of points on the curve is approximately the field size. For $q = 10^{40}$ this is a pretty big number, but there are other problems.

Both “order” and “cardinality” of a curve mean the total number of points that satisfy a specific equation. The order of a point is defined by the number of times we can add the point to itself until we get to the point at infinity. The order of any point on a curve will evenly divide the order of the curve. This amazing fact will be very useful later. (Math wizards will know that all Abelian groups have this property.) If the order of the curve is “smooth,” meaning lots of small factors, it is easier to crack.

Koyama et al. [8] give a description of how to pick elliptic curves such that finding their order is trivial. The problem they encounter is in using these elliptic curves: The computational cost is much higher than for modular exponentiation (e.g., RSA). Furthermore, the crypto schemes they present rely on using two fields and computing the math using a public key created by multiplying two large primes. The net end result is that this method is vulnerable to factoring just as RSA is, and it is six times slower too.

There may be schemes proposed in the future that solve these problems. For now, we'll consider it an academic curiosity and move on to polynomial and normal basis mathematics over finite fields.

5.3 Mathematics of elliptic curves over Galois Fields

Let's look at elliptic curves over F_{2^n}. That means our constants are either polynomial or normal basis numbers. It also means we cannot use the simplified version of equation (5.1), which we used for real numbers, for our elliptic curve equations.

The mathematicians tell us [2, 22] that we need to use either this version:

$$y^2 + xy = x^3 + a_2 x^2 + a_6 \tag{5.18}$$

or this version:

$$y^2 + y = x^3 + a_4 x + a_6 . \tag{5.19}$$

Now, the mathematicians can prove to you (if you care to listen) that the second form above, equation (5.19), is called a "supersingular" curve. These forms have the advantage that they can be computed quickly. However, being a special class of curves, they have some very special properties. These properties make supersingular curves unsuitable for cryptography.

The curves of equation (5.18) are called "nonsupersingular." To date, no method of attack is known to be less than fully exponential in time. Curves of this form are excellent for cryptographic applications. One must be careful in choosing the coefficients to get maximum benefit of security. A poor choice can create a curve that is easier for the cryptanalyst to attack.

For equation (5.18) to be valid, a_6 must never be 0. However, a_2 can be 0. The rules are the same as before: Take any two points on the curve; draw a line between them; and the negative of the third point, which intersects both the curve and the line, is the "sum" of the first two points.

Unfortunately, this is very hard to picture. For example, the rule to negate a point $P = (x,y)$ is:

$$-P = (x,\ y + x). \tag{5.20}$$

Because x and y are now variables in the field F_{2^n}, the + operation is just an exclusive-or of all the bits of x with all the bits of y.

The rules for adding two points over F_{2^n} can be stated as follows [2, 87; 9]:

Given:

$$P = (x_1,y_1) \tag{5.21}$$

$$Q = (x_2,y_2) \tag{5.22}$$

then:

$$R = P + Q = (x_3,y_3) \tag{5.23}$$

if $P \neq Q$:

$$\theta = \frac{y_2 - y_1}{x_2 - x_1} \tag{5.24}$$

$$x_3 = \theta^2 + \theta + x_1 + x_2 + a_2 \tag{5.25}$$

$$y_3 = \theta(x_1 + x_3) - y_1 \tag{5.26}$$

if $P = Q$:

$$\theta = x + \frac{y}{x} \quad (5.27)$$

$$x_3 = \theta^2 + \theta + a_2 \quad (5.28)$$

$$y_3 = x^2 + (\theta + 1)x_3 . \quad (5.29)$$

In equations (5.27) through (5.29), I left off subscript $_1$ to the x and y terms because there is only one input point.

Note that the initial calculation of θ requires an inversion operation over the field F_{2^n}. Schroeppel et al. [9] showed how this could be done in very few operations for the right choice of polynomial basis. This inversion calculation is the most time consuming for any basis and has prevented elliptic curve cryptography from really moving into the main stream. Over the past two years there have been some great improvements in speeding up this calculation. At the end of this book, I'll present one of the more amazing solutions to this problem I've ever seen. It should make elliptic curve crypto one of the faster public key systems available.

Before getting to the code, let's do a little recap. The elliptic curve math we've talked about so far is a higher-level algebra that adds any two points on a curve and gets a third one. To keep things straight, we'll talk about "adding points" to indicate addition over an elliptic curve and we'll say "adding field elements" when discussing addition using F_{2^n} (exclusive-or). This is probably the most confusing part of discussions about elliptic curve math. Once you get it straight, it will seem pretty simple.

The subroutine schematic for this chapter is shown in figure 5.5. The polynomial basis is parallel to the normal basis in the sense that either can be used to implement the elliptic curve math. Once you pick which basis to use, the choice of subroutines is fixed. Simple routines from chapters 3 and 4 are included in both versions, and this is seen as a dependency in the schematic.

After describing the definitions for points and curves, I'll go into code that is common to both polynomial and normal basis math, including some simple output routines. Then I'll get into sums of points over curves: first with polynomials and then with normal basis representations. Then I'll get into the code that does multiplication.

For code, the first thing we need to define is a point and the second is a curve. Our elliptic math header file, `eliptic.h` looks like this:

```
/******   eliptic.h   *****/
/*************************************************************
*                                                            *
*       These are structures used to create elliptic curve   *
*  points and parameters.  "form" is a just a fast way to check *
```

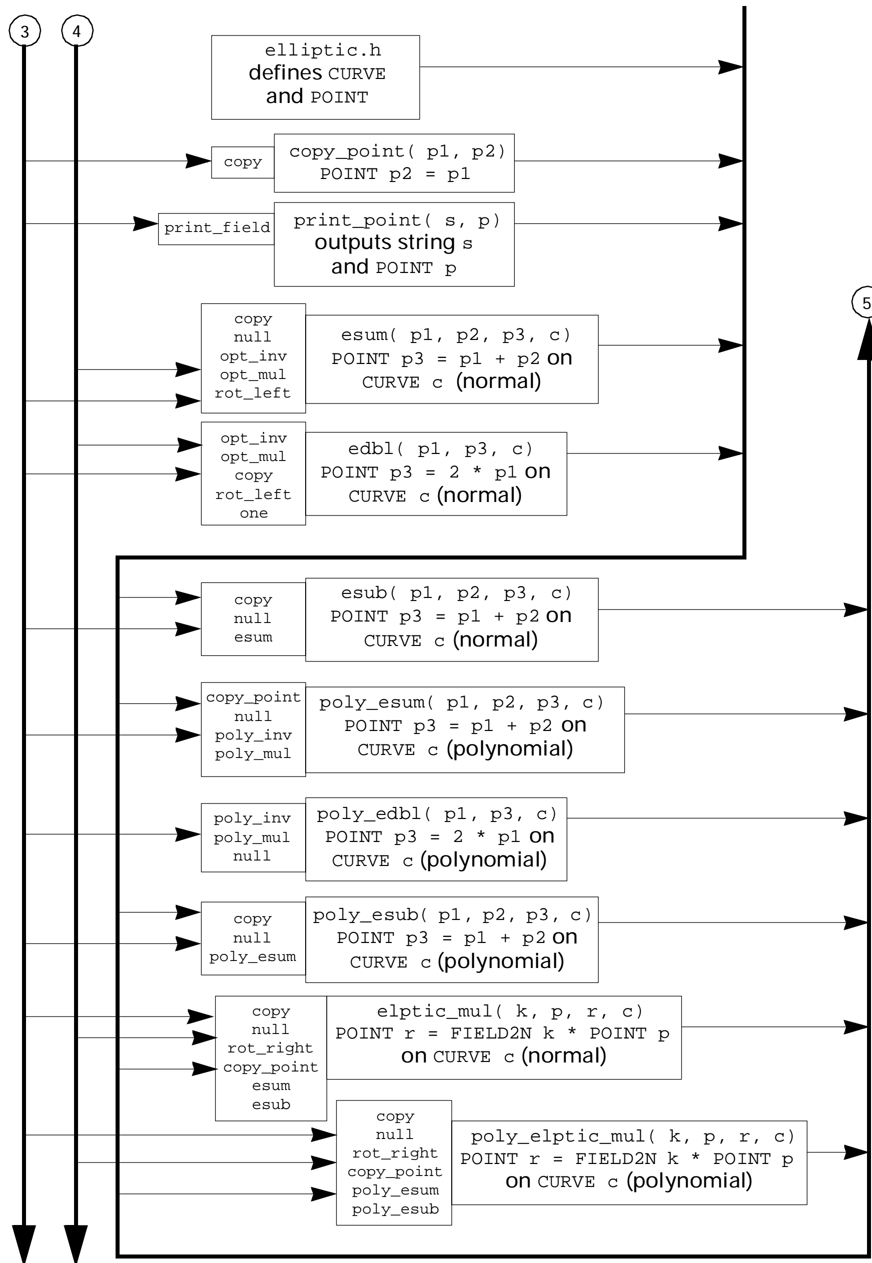

Figure 5.5 Subroutine schematic

```
*   if a2 == 0.                                                  *
*              form            equation                          *
*                                                                *
*               0              y^2 + xy = x^3 + a_6              *
```

```
*               1              y^2 + xy = x^3 + a_2*x^2 + a_6  *
*                                                               *
****************************************************************/

typedef struct
{
        INDEX   form;
        FIELD2N  a2;
        FIELD2N  a6;
} CURVE;

/*  coordinates for a point  */

typedef struct
{
        FIELD2N  x;
        FIELD2N  y;
} POINT;
```

A point is defined as an (x,y) pair over the field F_{2^n}. From here on we'll use the math of chapters 3 or 4 to operate on these values and to find new ones.

A curve is defined from equation (5.18). If you've looked at equations (5.21) through (5.29) for summing and doubling, you'll notice that a_6 in equation (5.18), does not enter into them anywhere. The choice of a_2 and a_6 does have cryptographic implications, so we'll save that for the next chapter. The value of the `INDEX` variable `form` is 0 for $a2 = 0$. I originally thought it would be useful for other possible curve formulas, but the forms listed above turn out to be the best possible for cryptographic use.

Let's implement equation (5.20) first, the negative of an elliptic curve point.

```
void eneg (p)
FIELD2N   *p;
{
          register      INDEX    i;

          SUMLOOP (i)   p-> y.e[i]  ^= p->x.e[i];
}
```

This routine steps through every element in the stored array and XOR's the x component with the y component. This negates the point in place. Because this is so simple, we don't really need to use `eneg`, but you'll see it in the subtraction routine as a single line of code.

One common routine we need for both polynomial and normal basis is very simple: It just copies a point from one storage array to another.

```
/*  need to move points around, not just values.  Optimize later.  */

void copy_point (p1, p2)
POINT *p1, *p2;
```

```
{
        copy (&p1->x, &p2->x);
        copy (&p1->y, &p2->y);
}
```

Here are three simple routines that I have found useful for debugging code. The only purpose is to see data printed out in a quick-and-dirty format. The first routine prints out a hex dump of a `FIELD2N` variable, the second prints out a `POINT`, and the third prints out the data of a `CURVE`. Feel free to change these to whatever your taste desires.

```
void print_field( string, field)
char *string;
FIELD2N *field;
{
        INDEX i;

        printf("%s : ", string);
        SUMLOOP(i) printf("%8x ", field->e[i]);
        printf("\n");
}

void print_point( string, point)
char *string;
POINT *point;
{
        printf("%s\n", string);
        print_field( "x", &point->x);
        print_field( "y", &point->y);
        printf("\n");
}
void print_curve( string, curv)
char *string;
CURVE *curv;
{
        printf("%s\n", string);
        printf("form = %d\n", curv->form);
        if (curv->form) print_field( "a2", &curv->a2);
        print_field( "a6", &curv->a6);
        printf("\n");
}
```

Each routine expects a string pointer and a structure pointer for arguments. All outputs from test code in this book are printed with these simple subroutines.

5.4 Polynomial basis elliptic curve subroutines

Next, let's implement equations (5.24) through (5.26) using the polynomial subroutines of chapter 3. To prevent the math package from going berserk I've defined $\boldsymbol{O}_\infty$ to be

(0,0). Since (0,0) cannot be on any nonsupersingular curve, it can be used as the point at infinity. On entrance to the summation routine we first check to see if either point is O_∞. If it is, the answer is the other point.

Here is the summation routine for polynomial math.

```
/************************************************************************
*                                                                      *
*   Implement elliptic curve point addition for polynomial basis form.     *
*  This follows R. Schroeppel, H. Orman, S. O'Mally, "Fast Key Exchange with*
*  Elliptic Curve Systems", CRYPTO '95, TR-95-03, Univ. of Arizona, Comp.   *
*  Science Dept.                                                       *
************************************************************************/

void poly_esum (p1, p2, p3, curv)
POINT   *p1, *p2, *p3;
CURVE   *curv;
{
    INDEX   i;
    FIELD2N  x1, y1, theta, onex, theta2;
    ELEMENT  check;

/*  check if p1 or p2 is point at infinity  */

        check = 0;
        SUMLOOP(i) check |= p1->x.e[i] | p1->y.e[i];
        if (!check)
        {
                copy_point( p2, p3);
                return;
        }
        check = 0;
        SUMLOOP(i) check |= p2->x.e[i] | p2->y.e[i];
        if (!check)
        {
                copy_point( p1, p3);
                return;
        }
```

The lines that compute $\theta = (y_1 + y_2)/(x_1 + x_2)$ also check to see if the sum $x_1 + x_2$ is 0. The error check only tells us if the input x field elements are identical. Since we are not doubling, this error check assumes we are adding P to $-P$. The result is the point at infinity. If we should be doubling, it is an error at a higher level.

```
/*  compute theta = (y_1 + y_2)/(x_1 + x_2)  */

    null(&x1);
    null(&y1);
    check = 0;
    SUMLOOP(i)
    {
            x1.e[i] = p1->x.e[i] ^ p2->x.e[i];
```

```
            y1.e[i] = p1->y.e[i] ^ p2->y.e[i];
            check |= x1.e[i];
    }
    if (!check)   /* return point at infinity  */
    {
       null(&p3->x);
       null(&p3->y);
       return;
    }
```

The error check does prevent us from attempting to invert 0. The θ value is computed first. The check for the value of `curv->form` is performed once, and the loop contains only the terms required to create the sum. The result is the value for x_3 in equation (5.25).

```
    poly_inv( &x1, &onex);
    poly_mul( &onex, &y1, &theta);  /*  compute y_1/x_1 = theta */
    poly_mul(&theta, &theta, &theta2);   /* then theta^2  */

/*  with theta and theta^2, compute x_3  */

    if (curv->form)
            SUMLOOP (i)
            p3->x.e[i] = theta.e[i] ^ theta2.e[i] ^ p1->x.e[i] ^ p2->x.e[i]
                 ^ curv->a2.e[i];
    else
            SUMLOOP (i)
            p3->x.e[i] = theta.e[i] ^ theta2.e[i] ^ p1->x.e[i]
                 ^ p2->x.e[i];

/*  next find y_3  */

    SUMLOOP (i) x1.e[i] = p1->x.e[i] ^ p3->x.e[i];
    poly_mul( &x1, &theta, &theta2);
    SUMLOOP (i) p3->y.e[i] = theta2.e[i] ^ p3->x.e[i] ^ p1->y.e[i];
}
```

The last three lines of code implement equation (5.26). All field elements are handled as members of the field modulo `poly_prime`, the irreducible polynomial. The (x,y) point value is returned in the designated location.

OK, now let's code up equations (5.27) through (5.29). This time there is one point of input, one curve input, and one point as output for $P_3 = 2P_1$.

```
/*  elliptic curve doubling routine for Schroeppel's algorithm over polymomial
    basis.  Enter with p1, p3 as source and destination as well as curv
    to operate on.  Returns p3 = 2*p1.
*/

void poly_edbl (p1, p3, curv)
POINT *p1, *p3;
CURVE *curv;
```

```
{
    FIELD2N  x1, y1, theta, theta2, t1;
    INDEX   i;
    ELEMENT  check;

    check = 0;
    SUMLOOP (i) check |= p1->x.e[i];
    if (!check)
    {
       null(&p3->x);
       null(&p3->y);
       return;
    }

/*  first compute theta = x + y/x  */

    poly_inv( &p1->x, &x1);
    poly_mul( &x1, &p1->y, &y1);
    SUMLOOP (i) theta.e[i] = p1->x.e[i] ^ y1.e[i];

/*  next compute x_3  */

    poly_mul( &theta, &theta, &theta2);
    if(curv->form)
       SUMLOOP (i) p3->x.e[i] = theta.e[i] ^ theta2.e[i] ^ curv->a2.e[i];
    else
       SUMLOOP (i) p3->x.e[i] = theta.e[i] ^ theta2.e[i];

/*  and lastly y_3  */

    theta.e[NUMWORD] ^= 1;           /*  theta + 1 */
    poly_mul( &theta, &p3->x, &t1);
    poly_mul( &p1->x, &p1->x, &x1);
    SUMLOOP (i) p3->y.e[i] = x1.e[i] ^ t1.e[i];
}
```

This routine is a straightforward copy of the equations. Adding 1 to the `theta` value is simple; we just flip the last bit. Again, we check to see if the input attempts to double a zero x value. The routine returns the point at infinity if so. That this is the correct value is obvious from equation (5.18). The point $(0, \sqrt{a_6})$ is on the curve, and, since $-P = (x,y + x)$, we have a point that is its own negative. Doubling this point is the same as adding the negative of the same point, which gives us the point at infinity.

Finally, we need to be able to subtract two points. Just as we can negate and add with integers, this routine negates the second point, as in equation (5.20), and adds it to the first point. Here's the code.

```
/*  subtract two points on a curve.  just negates p2 and does a sum.
    Returns p3 = p1 - p2 over curv.
*/

void poly_esub (p1, p2, p3, curv)
```

```
POINT   *p1, *p2, *p3;
CURVE   *curv;
{
    POINT   negp;
    INDEX   i;

    copy ( &p2->x, &negp.x);
    null (&negp.y);
    SUMLOOP(i) negp.y.e[i] = p2->x.e[i] ^ p2->y.e[i];
    poly_esum (p1, &negp, p3, curv);
}
```

Now let's take a quick look at the same routines transformed to work with normal basis representations.

5.5 *Optimal normal basis elliptic curve subroutines*

Equations (5.24) through (5.29) can also be implemented using optimal normal basis field elements. Instead of calling the polynomial subroutines, we call the normal basis field element addition and multiplication routines.

Here is the code for adding two points.

```
void esum (p1, p2, p3, curv)
POINT   *p1, *p2, *p3;
CURVE   *curv;
{
    INDEX   i;
    FIELD2N  x1, y1, theta, onex, theta2;

/*  compute theta = (y_1 + y_2)/(x_1 + x_2)  */

    null(&x1);
    null(&y1);
    SUMLOOP(i)
    {
            x1.e[i] = p1->x.e[i] ^ p2->x.e[i];
            y1.e[i] = p1->y.e[i] ^ p2->y.e[i];
    }
    opt_inv( &x1, &onex);
    opt_mul( &onex, &y1, &theta);
    copy( &theta, &theta2);
    rot_left(&theta2);

/*  with theta and theta^2, compute x_3  */

    if (curv->form)
            SUMLOOP (i)
            p3->x.e[i] = theta.e[i] ^ theta2.e[i] ^ p1->x.e[i] ^ p2->x.e[i]
```

```
                ^ curv->a2.e[i];
    else
            SUMLOOP (i)
            p3->x.e[i] = theta.e[i] ^ theta2.e[i] ^ p1->x.e[i] ^ p2->x.e[i];
/*  next find y_3  */

    SUMLOOP (i) x1.e[i] = p1->x.e[i] ^ p3->x.e[i];
    opt_mul( &x1, &theta, &theta2);
    SUMLOOP (i) p3->y.e[i] = theta2.e[i] ^ p3->x.e[i] ^ p1->y.e[i];
}
```

The only real difference here is the squaring of `theta`. This is a simple rotation. The `opt_inv` routine given in chapter 4 is a bit slow, but the call to it won't change if we replace it with something faster. Other than the replacement calls to `opt_*` routines instead of `poly_*` routines, the code is almost the same as previously described.

Here is the code to double a point using optimal normal basis mathematics.

```
/*  elliptic curve doubling routine for Schroeppel's algorithm over normal
    basis.  Enter with p1, p3 as source and destination as well as curv
    to operate on.  Returns p3 = 2*p1.
*/

void edbl (p1, p3, curv)
POINT *p1, *p3;
CURVE *curv;
{
    FIELD2N  x1, y1, theta, theta2, t1;
    INDEX   i;

/*  first compute theta = x + y/x  */

    opt_inv( &p1->x, &x1);
    opt_mul( &x1, &p1->y, &y1);
    SUMLOOP (i) theta.e[i] = p1->x.e[i] ^ y1.e[i];

/*  next compute x_3  */

    copy( &theta, &theta2);
    rot_left(&theta2);
    if(curv->form)
              SUMLOOP (i) p3->x.e[i] = theta.e[i] ^ theta2.e[i] ^ curv-
>a2.e[i];
    else
              SUMLOOP (i) p3->x.e[i] = theta.e[i] ^ theta2.e[i];

/*  and lastly y_3  */

    one( &y1);
    SUMLOOP (i) y1.e[i] ^= theta.e[i];
    opt_mul( &y1, &p3->x, &t1);
    copy( &p1->x, &x1);
    rot_left( &x1);
```

```
    SUMLOOP (i) p3->y.e[i] = x1.e[i] ^ t1.e[i];
}
```

The major differences here are, again, that the squaring operation is only a rotation and the way we add 1. The subroutine `one` is called to create the constant first and then it is added to `theta`. This takes a few more steps, but the squaring operation makes up for it.

Finally, we subtract two points. This is the same as the polynomial math routine, but we have to call the optimal normal basis algorithms to get the right answers.

```
void esub (p1, p2, p3, curv)
POINT   *p1, *p2, *p3;
CURVE   *curv;
{
    POINT   negp;
    INDEX   i;

    copy ( &p2->x, &negp.x);
    null (&negp.y);
    SUMLOOP(i) negp.y.e[i] = p2->x.e[i] ^ p2->y.e[i];
    esum (p1, &negp, p3, curv);
}
```

5.6 Multiplication over elliptic curves

The next idea we need to look at is multiplication over an elliptic curve. If not *the* most confusing term, certainly the idea of multiplying points is a touch weird. In fact, the idea refers to computing:

$$Q = kP \tag{5.30}$$

where Q and P are points on an elliptic curve and k is an integer. What this really means is that we add P to itself k times.

Some people have a hard time with the term “multiplication” in this context. After all, we are mixing integers and points, so they can’t really be multiplied. But this is where I think the mathematicians have named it correctly. When we multiply 3 times 5, we simply sum 5 plus 5 plus 5. That is multiplication by definition. When we add point P plus P plus P, we have added P three times, so why not call it 3 times P? Just as with multiplication of integers, there are more efficient ways to find the final answer than doing the straight sum.

Before getting into efficient methods of computing elliptic curve multiplication, let’s review again what we have in terms of mathematics.

1. An elliptic curve is defined over some field, F_q, where q is a large prime or 2^n.

2. The math used to compute addition of points on a particular elliptic curve is performed using a field.
3. The elliptic curve itself—that is, the points on it—forms a cyclic group (a field) also.

The last item is important, because the integer k in equation (5.30) need not be larger than the "order" of the point P. In fact, we can use modular math and reduce k modulo the order of P before doing anything else and save a great deal of processing time.

Unfortunately, finding the order of a point is not so easy. There has been a great deal of research performed in the late 1990s to reduce the effort needed to compute the order of a point. The order of any particular point will be one of the factors that compose the cardinality of the curve. Mathematicians use the symbol E to denote equation (5.18) (or any elliptic curve) and the combination $\#E$ to denote the order of an elliptic curve.

Factoring is an important field for cryptographers. The mathematics of factoring will not be described here (see [10] and references therein to get into factoring). Elliptic curves with specific properties are needed for factoring large numbers. In turn, factoring algorithms are needed for finding the properties of elliptic curves!

If we do not know the order of a point P we can still compute equation (5.30), but we may not be as efficient as we could be. Be warned that you may find the order of a point by accident. If N is the order of a point, then $NP = \boldsymbol{O}_\infty$ over an elliptic curve. A good crypto system will avoid this case if N is initially unknown and save N for future reference. For large field sizes the probability is exceptionally small that you'll find this by accident, so it's not really worth the effort.

So much for a few preliminaries. Let's see how to perform an elliptic multiply. This algorithm comes from [11]. The first assumption we need to make is that the number k, which is an integer, will fit into the same number of bits as our `FIELD2N`. This is a valid assumption based on Hasse's Theorem, equation (5.16).

Let's first look at the math. Suppose we want to compute $15P$. We can expand this as:

$$15P = P + 2(P + 2(P + 2P)). \tag{5.31}$$

What we have done here is a binary expansion of 15. Since $15 = 1111_2$, starting the chain with 0, the most significant bit is set so we add P. Then we double the result (2*) and add P, repeating until all bits are done. This is exactly the same algorithm used with the `mod_exp` routine of chapter 2. We just replace multiplication with addition over an elliptic curve. This expansion requires three doubling operations and three sums, a total of six operations instead of 15.

Now for Koblitz's, trick [11]: $15 = 16 - 1$. $15P = (2P)2 * 2 * 2 - P$. There are now only five operations instead of six. On top of that a doubling operation is slightly faster than a summing operation. Koblitz calls this a "balanced" expansion. The algorithm converts a string of set bits to a string of zero bits followed by –1. To make this clear, I'll show another example:

$$10045 * P = 10011100111101_2 * P. \tag{5.32}$$

The last bit in the chain of set bits is replaced with –1, all the other bits are replaced with 0's, and the leading 0 is set. So the balanced representation becomes:

$$1\ 0\ 1\ 0\ 0\ {-1}\ 0\ 1\ 0\ 0\ 0\ {-1}\ 0\ 1 \tag{5.33}$$

and the operations are:

$$(((((2P * 2 + P)2 * 2 * 2 - P)2 * 2 + P)2 * 2 * 2 * 2 - P)2 * 2 + P. \tag{5.34}$$

To follow this example, start with 0. Find the first bit (always set to +1) and add P. Going to the next bit we multiply by 2. That bit is clear, so just multiply by 2 again. The next bit is +1, so add P. Then multiply the whole thing by 2. That bit is clear, as is the next one, so we just multiply by 2 again. The next position is –1, so subtract P from the total this time. And so on down the chain.

Since 0, 1, and –1 all need to be represented, we need to expand the integer k from single bits to multiple bits. There are many ways to do this. For our purposes we'll use an array of characters. Clearly there is room for optimization to reduce RAM usage for embedded systems.

5.7 Balanced integer conversion code

A very recent paper [12] describes a simple way to create the balanced representation. It is called "nonadjacent form" in the paper, but it means the same thing. The method is called "Algorithm 2" and is shown in figure 5.6 in a slightly modified form. The input is n; the balanced array output is S.

Creating the balanced version of the integer comprises the major chunk of code in the elliptic multiply routine. It is also common to both polynomial and optimal normal basis versions. Let's look at this segment of code first.

The inputs to the elliptic multiply routine are the integer k, the point P, and the curve we are working with. The output is the new point $R = kP$. Internal to the routine we need the balanced array, a temporary working point, and various INDEX counters.

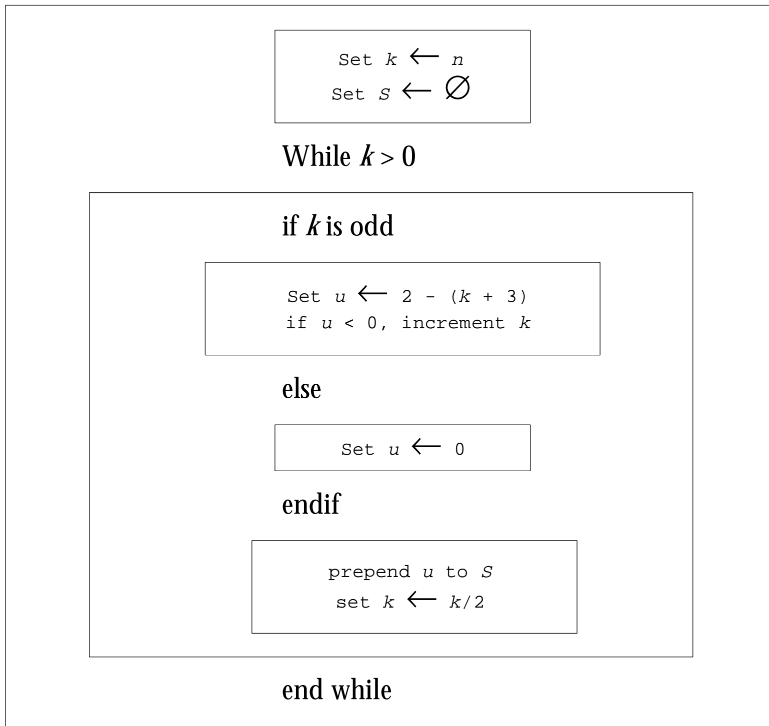

Figure 5.6 Algorithm 2 (slightly modified)

```
void  elptic_mul(k, p, r, curv)
FIELD2N         *k;
POINT           *p, *r;
CURVE           *curv;
{
        char            blncd[NUMBITS+1];
        INDEX           bit_count, i;
        ELEMENT         notzero;
        FIELD2N         number;
        POINT           temp;

/*  make sure input multiplier k is not zero.
        Return point at infinity if it is.
*/
        copy( k, &number);
        notzero = 0;
        SUMLOOP (i) notzero |= number.e[i];
        if (!notzero)
        {
                null (&r->x);
                null (&r->y);
                return;
        }
```

After saving a copy of the input, the first thing I do is check to see if an attempt is being made to multiply by 0. If so, the routine returns null for *x* and *y* values of the result point. Some people suggest that (0,0) be used to represent the point at infinity, so this may work out OK. It may be better to simply flag this as an error. Be careful to deal with these exceptions in a manner consistent with your application. I have used (0,0) with the polynomial package, but not the normal basis math. In normal basis this does not seem to be a problem (yet).

The next portion of code implements the above algorithm. I use the variable `bit_count` to keep track of which bit is being checked. The variable `u` is not required, because the compiler can do the `ELEMENT` to `char` conversion correctly for positive or negative 1. (Note for AIX users: This did not work, because the compiler assumed `unsigned char`, so check the output of your compiler compared with this code.)

```
/*  convert integer k (number) to balanced representation.
        Called non-adjacent form in "An Improved Algorithm for
        Arithmetic on a Family of Elliptic Curves", J. Solinas
        CRYPTO '97. This follows algorithm 2 in that paper.
*/
        bit_count = 0;
        while (notzero)
        {
        /*  if number odd, create 1 or -1 from last 2 bits  */

                if ( number.e[NUMWORD] & 1 )
                {
                        blncd[bit_count] = 2 - (number.e[NUMWORD] & 3);

        /*  if -1, then add 1 and propagate carry if needed  */

                        if ( blncd[bit_count] < 0 )
                        {
                                for (i=NUMWORD; i>=0; i--)
                                {
                                        number.e[i]++;
                                        if (number.e[i]) break;
                                }
                        }
                }
                else
                        blncd[bit_count] = 0;

        /*  divide number by 2, increment bit counter, and see if done  */

                number.e[NUMWORD] &= ~0 << 1;
                rot_right( &number);
                bit_count++;
                notzero = 0;
                SUMLOOP (i) notzero |= number.e[i];
        }
```

The only minor problem we have here with the algorithm is the increment step. This zeros out a sequence of ones. If an ELEMENT goes to 0, then we have to propagate a carry. Since we are only adding 1, the only time we actually have to propagate the carry is if we add 1 to an ELEMENT that has all bits set.

The last bit of number is cleared before we call the rot_right function. This performs the division with a predefined function, which is in both the polynomial and normal basis listings. The variable bit_count is bumped, and then I create the check to see if the algorithm is finished.

The last step is to unwind the balanced representation to perform the multiply. This is the same for both polynomial and normal basis, but we call different routines.

5.8 Following the balanced representation

The bit_count variable has to be decremented, because the last time through the above loop it was incremented once too many. This points us to the first entry in the balanced representation, which must be set. So we only need to copy the input point to the result. From there on, we only need to double the result and step through the blncd array to add, subtract, or do nothing as appropriate.

```
/*  now follow balanced representation and compute kP  */

        bit_count--;
        copy_point(p,r);       /* first bit always set */
        while (bit_count > 0)
        {
          edbl(r, &temp, curv);
          bit_count--;
          switch (blncd[bit_count])
          {
             case 1: esum (p, &temp, r, curv);
                                  break;
             case -1: esub (&temp, p, r, curv);
                                   break;
             case 0: copy_point (&temp, r);
           }
        }
}
```

Creating the balanced representation does not take too long. The doubling and summing loop takes the most time. Improving the efficiency of these routines, and especially inversion, will help improve overall throughput. Each time the loop ends, the result is stored in the destination POINT r. At the end of the loop, we can exit the routine.

The polynomial version of the elliptic curve multiplication routine ends as follows:

```
/*  now follow balanced representation and compute kP  */

        bit_count--;
        copy_point(p,r);        /* first bit always set */
        while (bit_count > 0)
        {
          poly_edbl(r, &temp, curv);
          bit_count--;
          switch (blncd[bit_count])
          {
             case 1: poly_esum (p, &temp, r, curv);
                                  break;
             case -1: poly_esub (&temp, p, r, curv);
                                   break;
             case 0: copy_point (&temp, r);
           }
        }
}
```

This is exactly the same as the optimal normal basis routines, but we replace the sum, double, and subtract routines with their polynomial equivalents.

As far as the math goes, we now have the core routines we need to implement elliptic curve crypto systems. The first choice is our basis: either polynomial or normal. The tradeoff between these is a tradeoff between speed (polynomial) or space (normal). Polynomial basis gives more speed but takes twice as much RAM, because multiplication doubles the number of bits stored before being reduced modulo the basis function. Normal basis uses minimal RAM but takes longer to compute, because the inversion routine of chapter 4 is slower. A combination of both can work the best, but this makes things a bit more complicated.

The second choice is our curve. There are important cryptographic implications here, and these need to be discussed in detail in the next chapter. Choosing points on the curve is somewhat arbitrary, but we don't want to choose points of particularly low order. So on with the show!

5.9 References

1. J. H. Silverman, *The Arithmentic of Elliptic Curves* (New York: Springer-Verlag, 1985).
2. A. J. Menezes, *Elliptic Curve Public Key Cryptosystems* (Boston: Kluwer Academic Publishers, 1993).
3. N. Koblitz, *Introduction to Elliptic Curves and Modular Forms* (New York: Springer-Verlag, 1993).
4. S. Abramowitz, *Handbook of Mathematical Fuctions*, 9th ed. (New York: Dover, 1972), 17.
5. N. Koblitz, *A Course in Number Theory and Cryptography* (New York: Springer-Verlag, 1987).

6 F. Morain, "Building Cyclic Elliptic Curves Modulo Large Primes," in *EUROCRYPT '91* (Berlin: Springer-Verlag, 1991), 328–336).

7 T. Beth and F. Schaefer, "NonSuper Singular Elliptic Curves for Public Key Cryptosystems," in *EUROCRYPT '91* (Berlin: Springer-Verlag, 1991), 316–327.

8 K. Koyama, U. M. Maurer, T. Okamoto, and S. A. Vanstone, "New Public Key Schemes Based on Elliptic Curves over the Ring Z_n," in *EUROCRYPT '91* (Berlin: Springer-Verlag, 1991), 252–266.

9 R. Schroeppel, H. Orman, and S. O'Mally, "Fast Key Exchange with Elliptic Curve Systems," TR-95-03 (Tucson, AZ: University of Arizona, Computer Sciences Department, 1995). (Also appears in *CRYPTO '95* [New York: Springer-Verlag, 1995].)

10 H. Riesel, *Prime Numbers and Computer Methods for Factorization*, 2d ed. (Boston: Birkhauser, 1987).

11 N. Koblitz, "CM—Curves with Good Cryptographic Properties" in *CRYPTO '91*(New York: Springer-Verlag, 1992), 279.

12 J. A. Solinas, "An Improved Algorithm for Arithmetic on a Family of Elliptic Curves," in *CRYPTO '97* (New York: Springer-Verlag, 1997).