Factoring Class Polynomials over the Genus Field

M. Martin
m.martin@ellipsa.net

July 27, 2006

Abstract

Aimed at computer scientists, this *how to* describes a method (with detailed algorithms) that allows to compute the factors of a class polynomial over the genus field.

Primality proving... Cryptography... As soon as we want to build an elliptic curve with a known order over a $\mathbb{Z}/p$ field using the so-called *complex multiplication*, we have to find a root of a class polynomial. Depending on the degree of this polynomial (and on the size of the modulus p), this operation might be very lengthy. More concretely, suppose we have to find a root of $H_{-12932920}(x)$ (the degree of this polynomial is 832). Suppose now we can compute a factor of degree 13 more quickly than we can compute the whole polynomial $H(x)$ itself... Of course, it would make the task easier. But how to do that?

Keywords: complex multiplication, genus field, class polynomial, factoring

Introduction

Let $h(-D)$ and $g(-D)$, denoted h and g in the sequel, be the class number and the genus number associated with a negative fundamental discriminant $-D$. The class number h is the number of primitive reduced forms (a, b, c) of discriminant $-D = b^2 - 4ac$. The genus number g is the number of genera associated with $-D$ (see, for instance, [3, pp. 221–230] or [4, pp. 53–63]).

The method described in this paper can be seen as an extension, that works when $h \geq g$, of the method of D. Bernardi [1] presented in [8, § 6.2.3].

Our goal is to build the factors of a class polynomial (when, of course, these factors exist, i.e., when $g > 1$ [2]) over a compositum of quadratic fields called the *genus field*. The genus field, denoted $K_{\mathcal{G}}$ in the sequel, is a field extension of $K = \mathbb{Q}(\sqrt{-D})$. More precisely, we want to obtain

$$H_{-D}(x) = \prod_{i=0}^{g-1} Q_i(x)$$

$$\text{with} \quad Q_i(x) = \frac{1}{g} \sum_{j=0}^{\frac{h}{g}-1} \left(\sum_{k=0}^{g-1} S_{i,k}\, B_k\, M_{k,j} \right) x^j + x^{\frac{h}{g}} \tag{1}$$

where S is a sign matrix (its coefficients are ± 1), B is a basis and M is a coefficient matrix (each of its column vectors consists of the coefficients of an integer of $K_{\mathcal{G}}$ multiplied by g).

Note that the matrix S and the basis B could be merged into a single matrix equal to $S * \mathrm{Diag}(B)$. They are not mainly for computational convenience: not only does the representation used minimize the memory needed to store the values, but B is not the same over $\mathbb{Z}/p$ as it is over $\mathbb{C}$ whereas S is the same for both cases.

Before going further, let us see the purpose of the sign matrix S with a small (and artificial) example.

Let $L = \mathbb{Q}(\sqrt{2}, \sqrt{3})$. L is a field extension of $\mathbb{Q}$ containing all the numbers of the form $u = a + b\sqrt{2} + c\sqrt{3} + d\sqrt{6}$ with $a, b, c, d \in \mathbb{Q}$. By definition, the Galois group $\mathrm{Gal}(L/\mathbb{Q})$ consists of all the automorphisms $\sigma : L \to L$ that fix $\mathbb{Q}$, i.e., such that $\sigma(q) = q$ for any $q \in \mathbb{Q}$. L being a Galois extension (because L is the splitting field of the separable polynomial $(x^2 - 2)(x^2 - 3) \in \mathbb{Q}[x]$ [3]), there are exactly

[1] The method of D. Bernardi only works when $h = g$, i.e., it only works with 56 discriminants called *Euler numbers* or *ideonal numbers* (counting the ones such that $g > 1$).

[2] If $g = 1$, there is a single factor, $H_{-D}(x)$ itself.

[3] A polynomial is *separable* if it has distinct roots.

$\dim_{\mathbb{Q}} L = 4$ such automorphisms:

$$\begin{array}{ll} \sigma_0, & \text{the identity map on } L, \\ \sigma_1, & \text{that takes } \sqrt{2} \text{ to } -\sqrt{2} \text{ and } \sqrt{3} \text{ to } \sqrt{3}, \\ \sigma_2, & \text{that takes } \sqrt{2} \text{ to } \sqrt{2} \text{ and } \sqrt{3} \text{ to } -\sqrt{3}, \\ \sigma_3, & \text{equal to } (\sigma_1 \circ \sigma_2). \end{array}$$

With these σ_k's, we can compute the conjugates of u. We get [4]

$$\begin{aligned} \sigma_0(u) &= a + b\sqrt{2} + c\sqrt{3} + d\sqrt{6}, \\ \sigma_1(u) &= a - b\sqrt{2} + c\sqrt{3} - d\sqrt{6}, \\ \sigma_2(u) &= a + b\sqrt{2} - c\sqrt{3} - d\sqrt{6}, \\ \sigma_3(u) &= a - b\sqrt{2} - c\sqrt{3} + d\sqrt{6}. \end{aligned}$$

In a matrix form, with $u = (a, b, c, d)$, i.e., with u expressed with respect to the basis $\widetilde{B} = (1, \sqrt{2}, \sqrt{3}, \sqrt{6})$, the previous equalities can be written

$$\begin{pmatrix} \sigma_0(u) \\ \sigma_1(u) \\ \sigma_2(u) \\ \sigma_3(u) \end{pmatrix} = \begin{pmatrix} + & + & + & + \\ + & - & + & - \\ + & + & - & - \\ + & - & - & + \end{pmatrix} * \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \sqrt{2} & 0 & 0 \\ 0 & 0 & \sqrt{3} & 0 \\ 0 & 0 & 0 & \sqrt{6} \end{pmatrix} * \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}.$$

Clearly, the line vectors of the sign matrix $\widetilde{S}$ consist of all the conjugates of the number $(1, 1, 1, 1)$ expressed with respect to the basis $\widetilde{B}$. Though the representation depends on the basis used, these lines represent the field automorphisms of $L/\mathbb{Q}$. They allow to compute any of the conjugates of any number $(a, b, c, d) \in L$, expressed with respect to $\widetilde{B}$, simply by doing dot products. For instance, with $v = (1, -2, 3, -4)$,

$$\sigma_2(v) = (+, +, -, -) \cdot (1, -2, 3, -4) = (1, -2, -3, 4).$$

In the equation (1), the $Q_i(x)$'s are conjugate polynomials, i.e., the g coefficients of degree j of the $Q_i(x)$'s are the g conjugates of an integer of $K_{\mathcal{G}}$ and the matrix S represents the field automorphisms of $K_{\mathcal{G}}/K$ exactly like $\widetilde{S}$ represents the ones of $L/\mathbb{Q}$. Being given $w = (a_0, \ldots, a_{g-1})$, a number of $K_{\mathcal{G}}$ expressed with respect to the basis B, the matrix S allows to get all of the conjugates of w.

In the sequel, the symbol $\oplus$ indicates the **xor** (bitwise exclusive **or**) operator. For instance, $5 \oplus 9 = 12$, i.e., $\overline{0101} \oplus \overline{1001} = \overline{1100}$ with a binary representation.

As numeric examples, all along the eight steps (see the contents above) we will use the data obtained with $-D = -2184$ for which we have $h = 24$ and $g = 8$.

[4]Instead of $\sigma(x)$, mathematicians often make use of σx. We use the former which is less ambiguous.

Step 1: Factoring the discriminant

A negative discriminant $-D$ is *fundamental* if D is a positive integer not divisible by the square of an odd prime and if $D \equiv 3 \pmod 4$ or $D \equiv 4, 8 \pmod{16}$.

First of all, we have to compute a small table F containing all the prime factors, possibly signed, of $-D$. For this operation -1 is regarded as a prime factor. Though this is not the way it is implemented, the factorization is simple: for all q's that are odd prime factors of D, we put $q^* = (-1)^{(q-1)/2}\, q$ in F, then we divide $-D$ by the product of all the q^*'s and the remaining even factor, if any, is divided by 4 and added to F.

For computational convenience, the first part of F, denoted $F-$ in the sequel, contains the negative factors in decreasing order (this part is never empty) ; the second part contains the positive factors in increasing order.

Algorithm 1.1 *(Factoring the discriminant)*

input

   D, *absolute value of a fundamental discriminant (small integer)*

outputs

   F, *array of factors (small integers)*

   g, *genus number associated with* $-D$ *(small integer)*

begin

```
  i ← 0
  j ← 0
  if (D mod 16) = 4 then
    F_0 ← −1
    i ← 1
    D ← D/4
  elseif (D mod 32) = 8 then
    F_0 ← −2
    i ← 1
    D ← D/8
  elseif (D mod 32) = 24 then
    T_0 ← 2  // T is a temporary table
    j ← 1
    D ← D/8
  endif
  while D > 1 do  // here D is a square-free product of odd primes
    p ← smallest prime factor of D
    D ← D/p
    if (p mod 4) = 3 then
      F_i ← −p
      i ← i + 1
```

else
$T_j \leftarrow p$
$j \leftarrow j + 1$
endif
endwhile
for k ***from*** 0 ***to*** $j - 1$ ***do*** $F_{i+k} \leftarrow T_k$ *// append T to F*
$g \leftarrow 2^{i+j-1}$
end

With $-D = -2184$, we have $-D = -8 * -3 * -7 * 13$, so we get $g = 8$ and $F = (-2, -3, -7, 13)$.

Using F, we can describe the genus field of $K = \mathbb{Q}(\sqrt{-2184}) = \mathbb{Q}(\sqrt{-546})$ as an extension of $\mathbb{Q}$, $K_{\mathcal{G}} = \mathbb{Q}(\sqrt{-2}, \sqrt{-3}, \sqrt{-7}, \sqrt{13})$, as well as an extension of K, $K_{\mathcal{G}} = K(\sqrt{-3}, \sqrt{-7}, \sqrt{13})$. In the latter case, there are several possible descriptions. In the sequel, we (implicitly) make use of $K_{\mathcal{G}} = K(\sqrt{6}, \sqrt{14}, \sqrt{13})$. The real extensions allow to build a basis B that has only real parts.

Step 2: The basis

In order to express the integers of $K_{\mathcal{G}}$ as tuples of coefficients, we need a basis. For our purposes, this basis consists of the square roots of the g possible positive products $\prod_{i=0}^{Size(F)-1} F_i^{e_i}$ where the e_i exponents are in $\{0,1\}$.

The possible values of the exponents suggest to use the binary representation of a small integer to code them. In the algorithm 2.1, we use the local variable x. Since we only want the positive products, we don't take in account the values of x such that $x \bmod 2^{Size(F-)}$ has an odd Hamming weight. With $-D = -2184$, we have $F = (-2, -3, -7, 13)$, so, for instance, the integer $x = \overline{1011}$ represents the product $F_3^1 * F_2^0 * F_1^1 * F_0^1 = 13 * 1 * -3 * -2 = 78$.

Algorithm 2.1 *(Computing the basis)*
inputs
    F, *factors of* $-D$ *(array of small integers)*
    g, *genus number of* $-D$ *(small integer)*
output
    A, *dot square of the basis (array[g] of small integers)*
begin
    $i \leftarrow 0$
    for j ***from*** 0 ***to*** $g + g - 1$ ***do***
        if *the bit parity of* (j *mod* $2^{Size(F-)}$) *is even* ***then***
            $A_i \leftarrow 1$
            $x \leftarrow j$
            $k \leftarrow 0$
            while $x > 0$ ***do***
                if *odd(x)* ***then*** $A_i \leftarrow A_i * F_k$ ***endif***
                $x \leftarrow x/2$
                $k \leftarrow k + 1$
            endwhile
            $i \leftarrow i + 1$
        endif
    endfor
end

With $F = (-2, -3, -7, 13)$, we get $A = (1, 6, 14, 21, 13, 78, 182, 273)$ and the basis is $B = (1, \sqrt{6}, \sqrt{14}, \sqrt{21}, \sqrt{13}, \sqrt{78}, \sqrt{182}, \sqrt{273})$.

It is not difficult to check that, with the composition law $\otimes$ defined as

$$A_i \otimes A_j = \frac{A_i * A_j}{\gcd(A_i, A_j)^2}\,, \tag{2}$$

the set of the A_k values is a group isomorphic to $(\mathbb{Z}/2\mathbb{Z})^t$ with $t = \log_2(g)$. Moreover, due to the one-to-one correspondence between the A_k's and the exponents used to build them (these exponents, with the $\oplus$ operation, are a subgroup of $(\mathbb{Z}/2\mathbb{Z})^{t+1}$ isomorphic to $(\mathbb{Z}/2\mathbb{Z})^t$), A is ordered such that

$$A_i \otimes A_j = A_{i \oplus j} \ . \tag{3}$$

Since $A_i * A_j = (A_i \otimes A_j) * \gcd(A_i, A_j)^2$ and since $D \equiv 0 \pmod{A_k}$, the equality (3) implies

$$\mathrm{Jacobi}(A_i * A_j, n) = \mathrm{Jacobi}(A_{i \oplus j}, n) \tag{4}$$

for any n such that $n > 1$ and $\gcd(n, 2D) = 1$.
These properties will be used at Step 5.

Note that we can build A using the field description $K_{\mathcal{G}} = K(\sqrt{6}, \sqrt{14}, \sqrt{13})$ as a seed. First, we set $A_1 = 6$, $A_2 = 14$ and $A_4 = 13$ (i.e., we set all the A_{2^i}'s), then, so that the equality (3) holds, we use it to compute the other A_k's. In the algorithm 2.1, we made use of the exponent trick because, from a computational point of view, this is more efficient (especially when working over $\mathbb{Z}/p$ fields as we will have to do at Step 8) but both methods are equivalent.

There are many ways to compute the bit parity of an integer (as it is required by the algorithm 2.1). For instance, if this integer is small enough, a lookup table of booleans might be a good choice. In the software *Primo* [5] [9, ellipsa.net], we make use of the following assembler routine

```
function ParityEven(N: Longword):Boolean; register;
asm
   mov   edx, eax  // edx := N
   shr   eax, 16
   xor   eax, edx
   xor   al, ah
   setpe al  // the result is returned in al
end;
```

[5] *Primo* was the first ECPP implementation that built class polynomials (and factored them over the genus field) *on the fly*.

Step 3: List of primitive reduced forms

A binary quadratic form is a polynomial $ax^2+bxy+cy^2 \in \mathbb{Z}[x, y]$ denoted (a, b, c) for short. Its discriminant is $-D = b^2 - 4ac$.
A form is *primitive* if $\gcd(a, b, c) = 1$. When its discriminant is negative, a form is *positive definite* if $a > 0$ and $c > 0$ and it is *reduced* if $|b| \leq a \leq c$ and if $b \geq 0$ whenever $a = c$ or $a = |b|$.
The set of the h positive definite, primitive and reduced binary quadratic forms of negative discriminant $-D$, with a law called *composition of forms* (see [2, pp. 247–249]), is a commutative group called the *class group* and denoted $\mathcal{C}(-D)$ in the sequel. The *principal form* is the identity element of $\mathcal{C}(-D)$. A form is *ambiguous* if it is its own inverse. The ambiguous forms are of the types $(a, 0, c)$, (a, a, c) or (a, b, a).

The algorithm 3.1 fills up a list L with $(h + g)/2$ positive definite, primitive and reduced forms (a, b, c) of discriminant $-D$. There are g forms of L that are ambiguous. Since we only store a form (a, b, c) and not its inverse $(a, -b, c)$, the remaining $(h - g)/2$ forms are half of all the non-ambiguous forms. The ordering of the forms in the list L has no importance.

Algorithm 3.1 *(Generating the forms)*
input
  D, *absolute value of a fundamental discriminant (small integer)*
outputs
  g, *genus number (small integer)*
  h, *class number (small integer)*
  L, *list of primitive reduced forms* (a, b, c) *(small integers)*
    $b^2 - 4ac = -D$ *for each* L_i
begin
  $bmax \leftarrow \left\lfloor \sqrt{D/3} \right\rfloor$
  $b \leftarrow D \bmod 2$
  $i \leftarrow 0$
  if $b = 0$ ***then***
    $q \leftarrow D/4$
    $a \leftarrow 1$
    $s \leftarrow 1$
    repeat
      if $(q \bmod a) = 0$ ***then***
        $L_i \leftarrow (a, 0, q/a)$
        $i \leftarrow i + 1$
      endif
      $s \leftarrow s + a$
      $a \leftarrow a + 1$

```
      s ← s + a  // s = a²
    until s > q
    b ← 2
  endif
  g ← i
  while b ≤ bmax do
    a ← b
    s ← a * a
    q ← (s + D) / 4
    repeat
      if (q mod a) = 0 then
        c ← q/a
        L_i ← (a, b, c)
        i ← i + 1
        if (a = b) or (a = c) then g ← g + 1 endif
      endif
      s ← s + a
      a ← a + 1
      s ← s + a  // s = a²
    until s > q
    b ← b + 2
  endwhile
  h ← i + i − g
end
```

With $-D = -2184$, we obtain $h = 24$, $g = 8$, and the list of the table 1.

In our example, the ambiguous forms are all located at the beginning of the list L. This is not always the case with other discriminants. The produced lists being sorted on b, ambiguous forms of the kind $a = b$ or $a = c$, if any, might be located anywhere.

To generate the primitive reduced forms one can also make use of the algorithm proposed in [6, § A.13.2] but note that, instead of fundamental discriminants $-D$, they make use of "reduced" discriminants $-d$ that are equal to either $-D$ or $-D/4$. The forms their algorithm produces are not always the same than the ones produced by the algorithm 3.1. With P1363 forms, the discriminant $-d$ is equal to $b^2 - ac$, not to $b^2 - 4ac$.
It is easy to get one discriminant knowing the other one:

- ***if*** $((d \bmod 4) = 1)$ or $((d \bmod 4) = 2)$ ***then*** $D \leftarrow d * 4$ ***else*** $D \leftarrow d$ ***endif***
- ***if*** $(D \bmod 4) = 0$ ***then*** $d \leftarrow D/4$ ***else*** $d \leftarrow D$ ***endif***

	(a, b, c)
L_0	$(1, 0, 546)$
L_1	$(2, 0, 273)$
L_2	$(3, 0, 182)$
L_3	$(6, 0, 91)$
L_4	$(7, 0, 78)$
L_5	$(13, 0, 42)$
L_6	$(14, 0, 39)$
L_7	$(21, 0, 26)$
L_8	$(5, 4, 110)$
L_9	$(10, 4, 55)$
L_{10}	$(11, 4, 50)$
L_{11}	$(22, 4, 25)$
L_{12}	$(15, 6, 37)$
L_{13}	$(17, 14, 35)$
L_{14}	$(19, 18, 33)$
L_{15}	$(23, 22, 29)$

Table 1: List of forms

Step 4: Weighting the genera

An integer n is *represented* by a form (a, b, c) if $ax^2 + bxy + cy^2 = n$ for some integers (x, y).
A *genus* is a set of forms. The *principal genus*, denoted $\mathcal{G}_0(-D)$ (or simply $\mathcal{G}_0$), is the genus containing the principal form. $\mathcal{G}_0$ is a subgroup of $\mathcal{C}(-D)$ constituted of all the "squares" $(a, b, c) \circ (a, b, c)$, i.e., constituted of all the forms composed with themselves. The other genera are cosets of $\mathcal{G}_0$.

Concretely, for each form (a, b, c) of the list L, we compute an integer n such that $n > 1$, $\gcd(n, 2D) = 1$ and n is represented by the form (with non-ambiguous forms, we compute n only for (a, b, c) since n is also represented by $(a, -b, c)$ [[6]]). Then we compute $\Phi(a, b, c) = (\mathrm{Jacobi}(F_i, n))$ (Φ is a tuple of $Size(F)$ signs identifying the genus of (a, b, c), see [4, pp. 53–54]) and we give it a weight w such that $0 \leq w < g$ and $w(\mathcal{G}_0) = 0$.

***Algorithm 4.1** (Weighting the genera)*
inputs
  F, *factors of* $-D$ *(array of small integers)*
  g, *genus number of* $-D$ *(small integer)*
  h, *class number of* $-D$ *(small integer)*
  L, *list of* $(h + g)/2$ *primitive reduced forms (3-tuples of small integers)*
output
  W, *weights (array[*$(h + g)/2$*] of small integers)*
begin
  for i ***from*** 0 ***to*** $(h + g)\ /\ 2 - 1$ ***do***
    $n \leftarrow$ *integer represented by* L_i
    $W_i \leftarrow 0$
    for j ***from*** Size$(F) - 1$ ***downto*** Size$(F-)$ ***do***
      $W_i \leftarrow W_i * 2$
      if Jacobi$(F_j, n) < 0$ ***then*** $W_i \leftarrow W_i + 1$ ***endif***
    endfor
    if Size$(F-) > 1$ ***then***
      u $\leftarrow$ *Jacobi*(F_0, n)
      for j ***from*** Size$(F-) - 1$ ***downto*** 1 ***do***
        $W_i \leftarrow W_i * 2$
        if Jacobi$(F_j, n) \neq u$ ***then*** $W_i \leftarrow W_i + 1$ ***endif***
      endfor
    endif
  endfor
end

With $F = (-2, -3, -7, 13)$, we obtain the values reported in the table 2.

[6] If n is represented by (a, b, c) with (x, y), it is represented by $(a, -b, c)$ with $(x, -y)$ or $(-x, y)$.

	(a,b,c)	(x,y)	n	J(-2,n)	J(-3,n)	J(-7,n)	J(13,n)	w
L_0	$(1,0,546)$	$(1,1)$	547	$+$	$+$	$+$	$+$	0
L_1	$(2,0,273)$	$(1,1)$	275	$+$	$-$	$+$	$-$	5
L_2	$(3,0,182)$	$(1,1)$	185	$+$	$-$	$-$	$+$	3
L_3	$(6,0,91)$	$(1,1)$	97	$+$	$+$	$-$	$-$	6
L_4	$(7,0,78)$	$(1,1)$	85	$-$	$+$	$+$	$-$	7
L_5	$(13,0,42)$	$(1,1)$	55	$-$	$+$	$-$	$+$	1
L_6	$(14,0,39)$	$(1,1)$	53	$-$	$-$	$+$	$+$	2
L_7	$(21,0,26)$	$(1,1)$	47	$-$	$-$	$-$	$-$	4
L_8	$(5,4,110)$	$(1,0)$	5	$-$	$-$	$-$	$-$	4
L_9	$(10,4,55)$	$(0,1)$	55	$-$	$+$	$-$	$+$	1
L_{10}	$(11,4,50)$	$(1,0)$	11	$+$	$-$	$+$	$-$	5
L_{11}	$(22,4,25)$	$(0,1)$	25	$+$	$+$	$+$	$+$	0
L_{12}	$(15,6,37)$	$(0,1)$	37	$-$	$+$	$+$	$-$	7
L_{13}	$(17,14,35)$	$(1,0)$	17	$+$	$-$	$-$	$+$	3
L_{14}	$(19,18,33)$	$(1,0)$	19	$+$	$+$	$-$	$-$	6
L_{15}	$(23,22,29)$	$(1,0)$	23	$-$	$-$	$+$	$+$	2

Table 2: List of weighted forms

In the table 2, we see that each genus (the forms having the same weight w are in the same genus) contains exactly one ambiguous form. This is not always the case with other discriminants. In fact, there is one ambiguous form in each genus if and only if h/g is odd (see [10, p. 44]).

J(6,n)	J(14,n)	J(13,n)	bit #0	bit #1	bit #2	weight
$+$	$+$	$+$	0	0	0	0
$-$	$+$	$-$	1	0	1	5
$-$	$-$	$+$	1	1	0	3
$+$	$-$	$-$	0	1	1	6
$-$	$-$	$-$	1	1	1	7
$-$	$+$	$+$	1	0	0	1
$+$	$-$	$+$	0	1	0	2
$+$	$+$	$-$	0	0	1	4

Table 3: Weights

The table 3 shows how the genera are weighted. In the table 2, the first Jacobi symbol column, which is always associated with a negative F_i, is combined (dot product) with all other columns associated with negative F_i's. If, for some discriminant, there is only one negative F_i, the first column is simply ignored. Doing so, the $\mathrm{J}(*,n)$'s of the table 3 header are always $\mathrm{J}(A_{2^0},n),\ldots,\mathrm{J}(A_{2^k},n),\ldots,\mathrm{J}(A_{\frac{g}{2}},n)$ with the A_i's computed at Step 2. Then we replace the value x of each cell by $(1-x)/2$ and we get the binary expressions of the weights.

Let $\mathcal{G}_w$ be the genus of weight w and let N_w be any n such that $\gcd(n, 2D) = 1$, $n > 1$ and n is represented by any form of $\mathcal{G}_w$ (for instance, using the table 2, N_3 could be equal to 185 or to 17). By construction, we have defined the weight w of a genus as being

$$w = \bigoplus_{i=0}^{log_2(g)-1} \left(\frac{1 - \mathrm{Jacobi}(A_{2^i}, N_w)}{2} * 2^i \right). \tag{5}$$

So, we have

$$\mathrm{Jacobi}(A_{2^i}, N_{2^j}) = \begin{cases} -1, & \text{if } i = j \\ +1, & \text{if } i \neq j \end{cases} \tag{6}$$

and

$$\mathrm{Jacobi}(A_{2^i}, N_m * N_n) = \mathrm{Jacobi}(A_{2^i}, N_{m \oplus n}). \tag{7}$$

To show (5) $\Rightarrow$ (6), it is sufficient to replace w by 2^j in (5). All the terms of the right hand side should be equal to 0 except the one for which $i = j$.

To show (5) $\Rightarrow$ (7), one can use the equality $\frac{(1-\alpha\beta)}{2} = \frac{(1-\alpha)}{2} \oplus \frac{(1-\beta)}{2}$ that always holds with $(\alpha, \beta) \in \{\pm 1\}^2$.

Now, since $\mathrm{Jacobi}(A_i * A_j, N_k) = \mathrm{Jacobi}(A_{i \oplus j}, N_k)$ (see Step 2) and since for any integer i such that $0 \leq i < 2^t$, $i = \bigoplus_{s=0}^{t-1} (\alpha_s * 2^s)$ with $\alpha_s \in \{0, 1\}$ ($\overline{\alpha_{t-1} \ldots \alpha_1 \alpha_0}$ is simply the binary representation of i), we have

$$\mathrm{Jacobi}(A_i, N_m * N_n) = \mathrm{Jacobi}(A_i, N_{m \oplus n}) \tag{8}$$

for any i in $0..g-1$ (and not only for i equal to a power of 2 less than g).
All these properties will be used at Step 5.

Note that, when indexing the genera as we do, with any form $f_i \in \mathcal{G}_i$ and any form $f_j \in \mathcal{G}_j$, the composed form $(f_i \circ f_j)$ is in $\mathcal{G}_{i \oplus j}$.

Also note that, in order to obtain the Φ tuples, by using the *Kronecker symbol* (see [2, pp. 28–30] for a description of this symbol) instead of the Jacobi symbol in the algorithm 4.1, we could avoid to compute the integers n represented by the forms since, for any i such that $0 \leq i < \mathrm{Size}(F)$,

$$\Phi_{F_i}(a, b, c) = \begin{cases} \mathrm{Kronecker}(F_i, a), & \text{if } \begin{cases} ((F_i = -1) \text{ and } (a \text{ is odd})) \\ \text{or} \\ (a \neq 0 \pmod{F_i})) \end{cases} \\ \mathrm{Kronecker}(F_i, c), & \text{if } \begin{cases} ((F_i = -1) \text{ and } (c \text{ is odd})) \\ \text{or} \\ (c \neq 0 \pmod{F_i})) \end{cases} \end{cases}.$$

The form (a, b, c) being primitive, if $F_i = -1$, a and c cannot be both even, and, if $F_i \neq -1$, a and c cannot be both divisible by F_i.

Step 5: The sign matrix

Let $U = (1, 1, \ldots, 1)$ be an integer of $K_{\mathcal{G}}$ expressed with respect to the basis B, then the line vector of the matrix S associated with the genus $\mathcal{G}_k$ is

$$\sigma_k(U) = (\text{Jacobi}(A_0, N_k),\ \text{Jacobi}(A_1, N_k),\ \ldots,\ \text{Jacobi}(A_{g-1}, N_k))\ [7]$$

(the σ_k's are the field automorphisms of $K_{\mathcal{G}}/K$) so, building the matrix S is straightforward.
We set the table 4 using our example $-D = -2184$. The most left column of the table contains the weights w. The four next columns reproduce the Φ tuples obtained at Step 4 for each genus $\mathcal{G}_w$. The sign matrix (the $8*8$ matrix on the right of the table) was obtained by means of dot products. For instance, the column A_5 = 78 = $-2 * -3 * 13$ is simply the product of the columns -2, -3 and 13.

w	-2	-3	-7	13	1	6	14	21	13	78	182	273
0	$+$	$+$	$+$	$+$	$+$	$+$	$+$	$+$	$+$	$+$	$+$	$+$
1	$-$	$+$	$-$	$+$	$+$	$-$	$+$	$-$	$+$	$-$	$+$	$-$
2	$-$	$-$	$+$	$+$	$+$	$+$	$-$	$-$	$+$	$+$	$-$	$-$
3	$+$	$-$	$-$	$+$	$+$	$-$	$-$	$+$	$+$	$-$	$-$	$+$
4	$-$	$-$	$-$	$-$	$+$	$+$	$+$	$+$	$-$	$-$	$-$	$-$
5	$+$	$-$	$+$	$-$	$+$	$-$	$+$	$-$	$-$	$+$	$-$	$+$
6	$+$	$+$	$-$	$-$	$+$	$+$	$-$	$-$	$-$	$-$	$+$	$+$
7	$-$	$+$	$+$	$-$	$+$	$-$	$-$	$+$	$-$	$+$	$+$	$-$

Table 4: Sign matrix

The sign matrix has an important property: $g * S^{-1} = {}^tS$ (the transposed of S).
1. Since the lines of S represent the conjugates of $U = (1, 1, \ldots, 1)$ in $K_{\mathcal{G}}$, their sum is equal to $\text{Trace}(U)_{K_{\mathcal{G}}/K} = (g, 0, \ldots, 0)$.
2. $\text{Jacobi}(A_i * A_j, N_w) = \text{Jacobi}(A_{i \oplus j}, N_w)$ (see Step 2).
So, if $c_{i,j}$ is a cell of ${}^tS * S$, we have

$$\begin{aligned} c_{i,j} = \sum_{k=0}^{g-1} {}^tS_{i,k} * S_{k,j} = \sum_{k=0}^{g-1} S_{k,i} * S_{k,j} &= \sum_{k=0}^{g-1} \text{Jacobi}(A_i * A_j, N_k) \\ &= \sum_{k=0}^{g-1} \text{Jacobi}(A_{i \oplus j}, N_k) = \begin{cases} g, & \text{if } i = j. \\ 0, & \text{if } i \neq j. \end{cases} \end{aligned}$$

Thus ${}^tS * S = g * I_g$.

The interesting consequence of the previous equality is that we have no matrix to inverse. The inverse we will use at Step 7 is given for free.

[7]This can be shown using the so-called *Artin symbol*.

At this point, we could build S using the Φ tuples as explained above but we can do a little better. With our orderings on the weights and on the A_k's, the sign matrix we get is particularly nice: it is symmetric, it depends only on g and it can be recursively built for any g starting with $S^{(1)} = (+)$.

In order to prove the previous claim, it is sufficient to show that, with $0 \le i < 2^k$ and $0 \le j < 2^k$, we always have $S_{i,\,j+2^k} = S_{i+2^k,\,j} = S_{i,\,j}$ and $S_{i+2^k,\,j+2^k} = -S_{i,\,j}$.

Identifying $S_{r,\,s}$ with $\text{Jacobi}(A_s, N_r)$ and using the equations (4) and (8), it comes

- $S_{r,\,s\oplus t} \;=\; \text{Jacobi}(A_{s\oplus t}, N_r) \;=\; \text{Jacobi}(A_s * A_t, N_r) \;=\; S_{r,\,s} * S_{r,\,t}$,
- $S_{r\oplus s,\,t} \;=\; \text{Jacobi}(A_t, N_{r\oplus s}) \;=\; \text{Jacobi}(A_t, N_r * N_s) \;=\; S_{r,\,t} * S_{s,\,t}$.

Since $(0 \le a < 2^b) \Rightarrow (a = \bigoplus_{n=0}^{b-1}(\alpha_n * 2^n)$ with $\alpha_n \in \{0,1\})$, using the equations (4), (6) and (8), we get

- $S_{i,\,2^k} \;=\; \text{Jacobi}(A_{2^k}, N_i) \;=\; \prod_{n=0}^{k-1} \text{Jacobi}(A_{2^k}, N_{\alpha_n * 2^n}) \;=\; 1$,
- $S_{2^k,\,j} \;=\; \text{Jacobi}(A_j, N_{2^k}) \;=\; \prod_{n=0}^{k-1} \text{Jacobi}(A_{\alpha_n * 2^n}, N_{2^k}) \;=\; 1$,
- $S_{2^k,\,2^k} \;=\; \text{Jacobi}(A_{2^k}, N_{2^k}) \;=\; -1$.

Now, since $(0 \le a < 2^b) \Rightarrow (a + 2^b = a \oplus 2^b)$, with the previous results, we finally get

- $S_{i,\,j+2^k} \;=\; S_{i,\,j} * S_{i,\,2^k} \;=\; S_{i,\,j}$,
- $S_{i+2^k,\,j} \;=\; S_{i,\,j} * S_{2^k,\,j} \;=\; S_{i,\,j}$,
- $S_{i+2^k,\,j+2^k} \;=\; S_{i,\,j} * S_{i,\,2^k} * S_{2^k,\,j} * S_{2^k,\,2^k} \;=\; -S_{i,\,j}$.

So, assuming the table A is built as explained at Step 2 and assuming the genera are weighted as indicated at Step 4, for any discriminant $-D$, the sign matrix can be built with

$$S^{(2m)} = \begin{pmatrix} S^{(m)} & S^{(m)} \\ S^{(m)} & -S^{(m)} \end{pmatrix} \quad \text{and} \quad S^{(1)} = (+).$$

Since we are interested in $S^{(8)}$ (for $-D = -2184$), here it is

$$S^{(8)} = \begin{pmatrix} S^{(4)} & S^{(4)} \\ S^{(4)} & -S^{(4)} \end{pmatrix} = \begin{pmatrix} S^{(2)} & S^{(2)} & S^{(2)} & S^{(2)} \\ S^{(2)} & -S^{(2)} & S^{(2)} & -S^{(2)} \\ S^{(2)} & S^{(2)} & -S^{(2)} & -S^{(2)} \\ S^{(2)} & -S^{(2)} & -S^{(2)} & S^{(2)} \end{pmatrix} =$$

$$
= \begin{pmatrix}
+ & + & + & + & + & + & + & + \\
+ & - & + & - & + & - & + & - \\
+ & + & - & - & + & + & - & - \\
+ & - & - & + & + & - & - & + \\
+ & + & + & + & - & - & - & - \\
+ & - & + & - & - & + & - & + \\
+ & + & - & - & - & - & + & + \\
+ & - & - & + & - & + & + & -
\end{pmatrix}
$$

Of course, the $8 * 8$ matrix is equal to the sign matrix of the table 4.

***Algorithm 5.1** (Computing the sign matrix)*

input
  g, genus number (small integer)
output
  S, sign matrix (array[g,g] of small integers (± 1))
begin
  $S_{0,0} \leftarrow 1$
  $k \leftarrow 1$
  while $k < g$ ***do***
    for i ***from*** 0 ***to*** $k - 1$ ***do***
      for j ***from*** 0 ***to*** $k - 1$ ***do***
        $S_{i+k,\, j} \leftarrow S_{i,\, j}$
        $S_{i,\, j+k} \leftarrow S_{i,\, j}$
        $S_{i+k,\, j+k} \leftarrow -S_{i,\, j}$
      endfor
    endfor
    $k \leftarrow k * 2$
  endwhile
end

Note that $S_{i,\,j} = (-1)^{W(i,j)}$ where $W(i,j)$ is the Hamming weight of $(i \wedge j)$ [8]. It is not difficult to show it since, with $0 \leq i < 2^k$ and $0 \leq j < 2^k$, we always have

- $W(i + 2^k, j) = W(i, j + 2^k) = W(i, j)$,
- $W(i + 2^k, j + 2^k) = W(i, j) + 1$.

[8]The symbol $\wedge$ indicates the bitwise **and** operator.

Step 6: Building the factors (floating point approximations)

There are no ways to know in advance the exact precision required in order to compute the floating point approximations of the roots of a class polynomial $H_{-D}(x)$ but, fortunately, there are rules to (over)estimate it. Concerning the invariants described in [6, § A.13.3], we use the rules proposed in [7, §4]. Since we are not computing the whole class polynomial but only factors, we can lower the precision and we use half the value found with their rules [9].

L_i	(a,b,c)	Associated root
L_0	$(1, 0, 546)$	$+33012526.575181343490717679407\dots$
...		
L_{11}	$(22, \pm 4, 25)$	$+0.078563664817619751654349053359\dots$ $\pm 0.53094269739398157019503618033\dots\ i$
...		

Table 5: Roots

We compute the roots [10] associated with all the forms of the list L. Then, we build g polynomials of degree h/g by regrouping the roots according to the genus of their associated forms. These polynomials, that always have real coefficients [11], are stored in a matrix $T[0..g-1,\ 0..h/g-1]$. We don't store the leading coefficients of the polynomials since they are always equal to 1. In the matrix T, the polynomial $T[k, \dots]$ is simply the polynomial built with the roots associated with the forms belonging to $\mathcal{G}_k$ (the genus of weight k). For instance, with $-D = -2184$, using the roots associated with the 3 forms of $\mathcal{G}_0$ (see Table 5), we get

$$T_0(x) = T_{0,0} + T_{0,1}\ x + T_{0,2}\ x^2 + x^3$$

where

$$T_{0,0} = -9509997.6729469079588896213936\dots$$
$$T_{0,1} = +5187170.4333430205234837861004\dots$$
$$T_{0,2} = -33012526.732308673125957182715\dots$$

[9] As a matter of fact, most of the time, we could lower it much more than that but it remains to find a rule depending on $-D$, on the invariant used, on g and on h/g, and that yields a correct precision for any discriminant.

[10] For our example, we made use of the P1363 class invariant.

[11] Assuming that, like here, we use a class invariant such that two conjugate roots are associated with two forms belonging to a same genus. With, for instance, the double η invariants of A. Enge and R. Schertz [5], this is not always the case.

Step 7: The coefficient matrix

When expressed with respect to a basis B as computed at Step 2, the integers of $K_{\mathcal{G}}$ have coefficients [12] that are not in $\mathbb{Z}$ but in $(\frac{1}{g})\mathbb{Z}$, so we can write them as $\frac{1}{g}\sum_{i=0}^{g-1} a_i B_i$ where the a_i's are in $\mathbb{Z}$. It is the reason why there is a $\frac{1}{g}$ factor in the equation (1). The column vectors of M are coefficients of integers of $K_{\mathcal{G}}$ multiplied by g so that they are integers and not fractions.

Identifying the coefficients (except the leading ones) of the factors $Q_i(x)$ with the matrix T computed at Step 6, let us rewrite the equation (1) in a matrix form. It comes

$$\frac{1}{g} * S * \mathrm{Diag}(B) * M = T$$
$$\frac{1}{g} * M = \mathrm{Diag}(B)^{-1} * S^{-1} * T$$
$$M = \mathrm{Diag}(B)^{-1} * {}^tS * T \quad (\text{using } g * S^{-1} = {}^tS)$$

and, since we are working with floating point approximations and not with exact values, we finally get

$$M_{i,j} = \mathrm{Round}\left(\frac{1}{B_i}\sum_{k=0}^{g-1} S_{k,i}\, T_{k,j}\right)$$

Algorithm 7.1 *(Computing the matrix M)*
inputs
  C, basis over $\mathbb{C}$ (inversed, i.e., $C_i = 1/\sqrt{A_i}$) (array[g] of big reals)
  g, genus number (small integer)
  h, class number (small integer)
  S, sign matrix (array[g,g] of small integers (± 1))
  T, matrix computed at Step 6 (array[g,h/g] of big reals)
output
  M, coefficient matrix (array[g,h/g] of big integers)
begin
  for i ***from*** 0 ***to*** $g-1$ ***do***
    for j ***from*** 0 ***to*** $h/g-1$ ***do***
      $x \leftarrow T_{0,\,j}$ *// we know that $S_{0,\,i} = 1$*
      for k ***from*** 1 ***to*** $g-1$ ***do*** $x \leftarrow x + S_{k,\,i} * T_{k,\,j}$ ***endfor***
      $M_{i,\,j} \leftarrow$ *Round($x * C_i$)*
    endfor
  endfor
end

[12]The $Q_i(x)$ polynomials having real coefficients, we only need the real integers of $K_{\mathcal{G}}$, so, since the basis B has no imaginary parts, the coefficients of these integers are in $K \cap \mathbb{R}$, i.e., they are real.

Of course, all the operations with the floating point numbers x and C_i's should be done using the precision found at Step 6.

With our example, $-D = -2184$, we get a matrix M equal to

$$\begin{pmatrix} -9509688 & 5187192 & -33012360 \\ -3882456 & 2117808 & -13477368 \\ -2541664 & 1386432 & -8823008 \\ -2075184 & 1131936 & -7203888 \\ -2637584 & 1438560 & -9155984 \\ -1076832 & 587328 & -3737952 \\ -704952 & 384496 & -2447064 \\ -575568 & 313920 & -1998000 \end{pmatrix}$$

Here, even if we did not multiply (implicitly) the coefficients by g, they would have been integers since they are all divisible by $g = 8$ but this is not always the case with other discriminants.

At this point, we have all we need in order to express the $Q_i(x)$'s of (1). For instance

$$\begin{aligned} Q_4(x) &= \frac{1}{8} \sum_{j=0}^{2} \left(\sum_{k=0}^{7} S_{4,k}\, B_k\, M_{k,j} \right) x^j + x^3 \\ &= Q_{4,0} + Q_{4,1}\, x + Q_{4,2}\, x^2 + x^3 \end{aligned}$$

where

$$\begin{aligned} Q_{4,0} = &- 1188711 - 485307\sqrt{6} - 317708\sqrt{14} - 259398\sqrt{21} \\ &+ 329698\sqrt{13} + 134604\sqrt{78} + 88119\sqrt{182} + 71946\sqrt{273} \\ Q_{4,1} = &+ 648399 + 264728\sqrt{6} + 173304\sqrt{14} + 141492\sqrt{21} \\ &- 179820\sqrt{13} - 73416\sqrt{78} - 48062\sqrt{182} - 39240\sqrt{273} \\ Q_{4,2} = &- 4126545 - 1684671\sqrt{6} - 1102876\sqrt{14} - 900486\sqrt{21} \\ &+ 1144498\sqrt{13} + 467244\sqrt{78} + 305883\sqrt{182} + 249750\sqrt{273} \end{aligned}$$

Step 8: Working over $\mathbb{Z}/p$ fields

The factorization (1) is also valid over any $\mathbb{Z}/p$ field assuming p is an odd prime such that $\gcd(p, D) = 1$ and $4p = x^2 + Dy^2$ for some $(x, y) \in \mathbb{Z}^2$.
In the sequel, to go on with our example $-D = -2184$, we will use the prime $p = \frac{1418446^2 + 2184 * 809283^2}{4} = 358099677116323$.

Computing the basis over $\mathbb{Z}/p$ is not as straightforward as computing it over $\mathbb{C}$. We have to count the number of negative factors, except F_0, used for each B_i and to apply the rule $\sqrt{-1} * \sqrt{-1} = -1$ in order to select the right root.

Algorithm 8.1 *(Computing the basis over $\mathbb{Z}/p$)*

```
input
   F, factors of −D (array of small integers)
   g, genus number of −D (small integer)
   p, odd prime such that 4p = x^2 + Dy^2 for some (x, y) ∈ Z^2 (big integer)
output
   B, basis over Z/p (array[g] of big integers)
begin
   for i from 0 to Size(F) − 1 do
      R_i ← Sqrt(F_i) mod p  // any of the 2 possible roots is ok
   endfor
   i ← 0
   for j from 0 to g + g − 1 do
      if the bit parity of (j mod 2^Size(F−)) is even then
         B_i ← 1
         x ← j
         k ← 0
         n ← 0
         while x > 0 do
            if odd(x) then
               B_i ← (B_i * R_k) mod p
               // count the negative factors but F_0
               if (k > 0) and (k < Size(F−)) then n ← n + 1 endif
            endif
            x ← x/2
            k ← k + 1
         endwhile
         if odd(n/2) then B_i ← p − B_i endif  // “negate” the root
         i ← i + 1
      endif
   endfor
end
```

With $-D = -2184$ and $p = 358099677116323$, we obtain the basis
$B_0 = 1$
$B_1 = 138579447850272$
$B_2 = 195858486873162$
$B_3 = 206988590703680$
$B_4 = 345798145618681$
$B_5 = 203082986192536$
$B_6 = 316722244248718$
$B_7 = 289064347795142$

At Step 7, we implicitly multiplied the coefficients of the matrix M by g. We can now cancel this operation, done in order to get integers and not fractions, by dividing the coefficients of the basis by g modulo p (we divide the coefficients of the basis and not the ones of the matrix M simply because this is generally faster).

Algorithm 8.2 *(Dividing the basis by g modulo p)*

```
inputs
   B, basis over Z/p (array[g] of big integers)
   g, genus number of −D (small integer)
   p, odd prime (big integer)
output
   B, basis over Z/p divided by g modulo p (array[g] of big integers)
begin
   k ← g
   while k > 1 do
      for i from 0 to g − 1 do
         if odd(B_i) then B_i ← B_i + p endif
         B_i ← B_i/2
      endfor
      k ← k/2
   endwhile
end
```

With $-D = -2184$ and $p = 358099677116323$, we obtain
$B_0 = 223812298197702$
$B_1 = 17322430981284$
$B_2 = 114007230138226$
$B_3 = 25873573837960$
$B_4 = 267037066400037$
$B_5 = 25385373274067$
$B_6 = 308165038368332$
$B_7 = 304707801311635$

At last!

Algorithm 8.3 *(Computing one factor of $H_{-D}(x)$ over $\mathbb{Z}/p$)*
inputs
B, basis over $\mathbb{Z}/p$ divided by g modulo p (array of big integers)
g, genus number of $-D$ (small integer)
h, class number of $-D$ (small integer)
i, index of the wished factor (small integer in $0..g-1$)
M, coefficient matrix (array[g,h/g] of big integers)
p, odd prime such that $4p = x^2 + Dy^2$ for some $(x, y) \in \mathbb{Z}^2$ (big integer)
S, sign matrix (array[g,g] of small integers (± 1))
output
Q, polynomial of degree h/g, factor of $H_{-D}(x)$ over $\mathbb{Z}/p$
begin
$Q_{h/g} \leftarrow 1$
for j ***from*** 0 ***to*** $h/g - 1$ ***do***
$Q_j \leftarrow B_0 * M_{0,j}$ *// we know that $S_{i,0} = 1$*
for k ***from*** 1 ***to*** $g-1$ ***do*** $Q_j \leftarrow Q_j + S_{i,k} * B_k * M_{k,j}$ ***endfor***
$Q_j \leftarrow Q_j \bmod p$
endfor
end

Called with i running through $0..7$, the algorithm 8.3 produces the following $Q_i(x)$ polynomials modulo p

$$
\begin{aligned}
Q_0(x) &= x^3 + 349411664140631\,x^2 + 236118815942277\,x + 121688504601529\\
Q_1(x) &= x^3 + 168320784679033\,x^2 + 99689071127264\,x + 274269421593516\\
Q_2(x) &= x^3 + 280369563518210\,x^2 + 206498150577522\,x + 114371282890567\\
Q_3(x) &= x^3 + 43259257063184\,x^2 + 213239562466426\,x + 19424900783462\\
Q_4(x) &= x^3 + 142239847045079\,x^2 + 70353977608422\,x + 104800940627475\\
Q_5(x) &= x^3 + 337091300899581\,x^2 + 128594135500790\,x + 251124699723139\\
Q_6(x) &= x^3 + 221712819911823\,x^2 + 91086453164646\,x + 217088197522536\\
Q_7(x) &= x^3 + 248093115311714\,x^2 + 28718870148814\,x + 329630751213380
\end{aligned}
$$

And each of these 8 polynomials is a factor, over $\mathbb{Z}/p$, of

$$
\begin{aligned}
H_{-2184}(x) = x^{24} - 33012360\,x^{23} - 5499066444\,x^{22} - 38191097592\,x^{21}\\
-860945475774\,x^{20} + 2860345968552\,x^{19} + 7390791596004\,x^{18}\\
+18071068156632\,x^{17} + 49152082910703\,x^{16} + 73526500711728\,x^{15}\\
+80616276081768\,x^{14} + 104922626382288\,x^{13} + 137712813694364\,x^{12}\\
+104922626382288\,x^{11} + 80616276081768\,x^{10} + 73526500711728\,x^{9}\\
+49152082910703\,x^{8} + 18071068156632\,x^{7} + 7390791596004\,x^{6}\\
+2860345968552\,x^{5} - 860945475774\,x^{4} - 38191097592\,x^{3}\\
-5499066444\,x^{2} - 33012360\,x + 1
\end{aligned}
$$

Of course, when only one factor of $H_{-D}(x)$ is needed (for instance, in the context of an ECPP implementation), by merging the algorithms 7.1 and 8.3, we compute and make use of the coefficients of the matrix M without storing them. Note that, when $g > h/g$, it is better to divide by g not the basis but the coefficients (up to the degree $h/g - 1$) of the returned polynomial.

Algorithm 8.4 *(Computing a factor of $H_{-D}(x)$ over $\mathbb{Z}/p$ without matrix M)*
inputs
  B, *basis over $\mathbb{Z}/p$ divided by g modulo p (array of big integers)*
  C, *basis over $\mathbb{C}$ (inversed, i.e., $C_i = 1/\sqrt{A_i}$) (array of big reals)*
  g, *genus number of $-D$ (small integer)*
  h, *class number of $-D$ (small integer)*
  p, *odd prime such that $4p = x^2 + Dy^2$ for some $(x, y) \in \mathbb{Z}^2$ (big integer)*
  S, *sign matrix (array[g,g] of small integers (± 1))*
  T, *matrix computed at Step 6 (array[g,h/g] of big reals)*
output
  Q, *polynomial of degree h/g, factor of $H_{-D}(x)$ over $\mathbb{Z}/p$*
begin
  $Q_{h/g} \leftarrow 1$
  for j ***from*** 0 ***to*** $h/g - 1$ ***do***
    $Q_j \leftarrow 0$
    for i ***from*** 0 ***to*** $g - 1$ ***do***
      $x \leftarrow T_{0,j}$ *// we know that $S_{0,i} = 1$*
      for k ***from*** 1 ***to*** $g - 1$ ***do*** $x \leftarrow x + S_{k,i} * T_{k,j}$ ***endfor***
      $Q_j \leftarrow Q_j + B_i *$ *Round*$(x * C_i)$
    endfor
    $Q_j \leftarrow Q_j$ mod p
  endfor
end

Conclusion

We have detailed a method that is not difficult to implement and that works well in practice. Our ECPP implementation, the software *Primo* [9], makes use of it since now five years.

The only other method (that works when $h \geq g$) we are aware of is the one proposed by A. Atkin and F. Morain in [1, § 7.3]. But, having never implemented it, the only thing we can say is that it seems a little more complicated than the one we have presented.

Acknowledgments

We are grateful to Michael Scott for his helpful comments on a preliminary version of this *how to*.

References

[1] A. O. L. ATKIN, F. MORAIN, **Elliptic curves and primality proving**, in *Math. Comp.* 61, 203, July 1993, pp. 29-68.
http://www.lix.polytechnique.fr/~morain/Articles/ecpp.ps.gz

[2] H. COHEN, **A Course in Computational Algebraic Number Theory**. Graduate Texts in Mathematics, 3rd ed., Springer 1996.

[3] H. COHN, **Advanced Number Theory**, Dover Publications Inc., New York, 1980.

[4] D. COX, **Primes of the form $\mathbf{x^2 + ny^2}$**, John Wiley & Sons, Inc., 1989.

[5] A. ENGE, R. SCHERTZ, **Constructing elliptic curves over finite fields using double eta-quotients**, in *Journal de la Théorie des Nombres de Bordeaux* #16, pp. 555-568, 2004.
http://almira.math.u-bordeaux.fr/jtnb/2004-3/pages555-568.pdf

[6] **IEEE P1363 / D8 (Draft version 8)**. *Standard Specifications for Public Key Cryptography*. Annex A (informative). Number-Theoretic Background.
http://www.informatik.tu-darmstadt.de/TI/Veroeffentlichung/Artikel/Kryptographie/PublicKeyAllgemein/

[7] E. KONSTANTINOU, Y. STAMATIOU, C. ZAROLIAGIS, **On the Use of Weber Polynomials in Elliptic Curve Cryptography** in *Public Key Infrastructure: First European PKI Workshop, Research and Applications* - EuroPKI 2004, LNCS 3093, Springer, pp. 335–349, 2004.
http://students.ceid.upatras.gr/~konstane/papers/europki2004-weber-poly.pdf

[8] F. MORAIN, **Implementation of the Atkin-Goldwasser-Kilian Primality Testing Algorithm**, RR #911, INRIA, October 1988.
http://www.lix.polytechnique.fr/Labo/Francois.Morain/Articles/INRIA-RR911.ps.gz

[9] Primo - ECPP implementation.
http://www.ellipsa.net/

[10] J. ROBERTSON, **Computing in Quadratic Orders**, 2006.
http://hometown.aol.com/jpr2718/