HYPERELLIPTIC CURVE METHOD FOR FACTORING INTEGERS

WENHAN WANG

1. Thoery and Algorithm

The idea of the method using hyperelliptic curves to factor integers is similar to the elliptic curve factoring method. However, they are different in some aspect: the algebraic group attached to hyperelliptic curve is its Jacobian $J(C)$, which is the quotient of the group of divisor classes of zero degree Div^0 modulo the group of principal divisors. Therefore the elements of $J(C)$ cannot be properly represented as coordinates, which is the main difference of algorithm for group operations.

Usually we represent a divisor class of hyperelliptic curve over a field k by a pair of polynomials $u(x), v(x) \in k[x]$, which is called the Mumford representation, corresponding to so-called reduced divisor, where $u(x)$ and $v(x)$ should satisfy the following conditions: (1) $\deg u(x) \leq g$; (2) $\deg v(x) < \deg u(x)$; (3) $u(x)$ has no repeated roots; (4) $u \mid v^2 + vb - f$. The cantor's algorithm shows how to add two divisors and then reduce the sum so that $u(x)$ and $v(x)$ satisfy the above conditions.

If k is a field, the cantor's algorithm always give the sum of two divisors and reduce it with no problems. But if we consider hyperelliptic curves over the ring $\mathbb{Z}/N\mathbb{Z}$, which is not a field, as we can see later this result in some problems, Cantor's algorithm might fails and using proper algorithm this might give a non-trivial factor of N.

If we are working with $J(C)$, where the curve is $C : y^2+h(x)y = f(x)$ over $\mathbb{Z}/N\mathbb{Z}$, then for two reduced divisor (u_1, v_1) and (u_2, v_2), Cantor's algorithm computes $v_1 + v_2 + h$ in the second step. If the leading coefficient of v_1+v_2+h is not invertible, i.e., v_1.leading_coeffient().gcd$(N) \neq$

Date: 04-29-2009.

Key words and phrases. Hyperelliptic Curve, Integer factorization, Sage.

1 we may compute the value of v_1.leading_coeffient().gcd(N), which might not be invertible in the ring $\mathbb{Z}/N\mathbb{Z}$. However, the probability that the leading coefficient of $v_1 + v_2 + h$ is not so large. But we have another chance for factor N: in the reduction part of the Cantor's algorithm, as we iterates the algorithm, at some step the leading coefficient of v will be not invertible, and we may compute the above "gcd" and get a non-trivial factor.

If we set $N = pq$, where p, q are large odd primes, then we have $J(\mathbb{Z}/N\mathbb{Z}) \cong J(\mathbb{F}_p) \times J(\mathbb{F}_q)$. The above case happens if in the course of arithmetic the result is a divisor whose image is exactly 0 in the two components: in $J(\mathbb{F}_p)$ or in $J(\mathbb{F}_q)$. But there may be other conditions that make the above case happen, but I don't know yet.

There is a senior thesis[1] by a Japanese undergraduate student (written in English) with the similar idea and similar algorithm states that the factor of N is found by computing the gcd of the leading coefficient of $u(x)$ and N, but I ran the same example is SAGE, and got that the factor is produced by the gcd of N and the leading coefficient of $v_1 + v_2 + h$, not as the paper by Japanese. So in the algorithm, it is necessary to check also the gcd of leading coefficient of $v(x)$ and N.

In the papers[2] written by Lestra, et al., they pointed out that under some reasonable conjectures, the hyperelliptic curve method is less efficient than elliptic curves. The following theorem form there paper illustrates the fact:

Theorem 1.1. [2] *Under some reasonable hypothesis, the elliptic curve method takes optimal maximal choice of parameters takes at most:*

$$L_p[\frac{1}{2} + \sqrt{2}o(1)](\log n)^2.$$

The hyperelliptic curve method, under the same hypothesis, takes at most

$$L_p[\frac{1}{2} + \sqrt{2}o(1)](\log n)^2.$$

Although by this theorem we may conclude that the HECM is less fast as ECM, as Lenstra, and et al. pointed out, HECM works well for generating smooth numbers. So I think it is still worthwhile to implement the algorithm for factoring in SAGE.

2. Implement in SAGE

Several function is defined in sage for the implementation:

- `hecptgen(N)`: this function generates a point on a certain hyperelliptic curve over the ring $\mathbb{Z}/N\mathbb{Z}$. Such a point has small absolute x-coordinate value. This function generates the Mumford representation of a divisor corresponding to the points returned by the old version of `hecptgen(N)`. In this new version of William Stein, he combined `hecptgen` and `hecdivgen`, so that `hecptgen` can directly generate a divisor (class).

- `hecadd(u1,v1,u2,v2,N)`: this function adds two reduced divisors and return a 'raw' divisor, not reduced. The algorithm is the combination part of the Cantor's algorithm.

- `hecred(u,v,N)`: this function use the Cantor's reduction algorithm to reduce a divisor class (u, v). For factorization, this algorithm also computes the maximum of the gcd of the leading coefficient of u with N and the gcd of the leading coefficient of v. If the gcd is greater than 1, the function will output the non-trivial factor and break.

- `hecmul(u,v,m,N)`: this function computes m times (u, v), i.e., add (u, v) by itself and then reduce for m times.

- `factor_hecm(N)`: this function actually factor N by the hyperelliptic curve method using the idea in section 1.

I will then show some examples that HECM works in sage.

Example 2.1. In this example, I did exactly the same input in the Japanese paper and ran. However, the gcd of the leading coefficient of u and N is 1, and the output is truncated because the leading coefficient of v has non-trivial gcd with N, as shown in the result.

Here is the result using the code of William's version and exactly the same example in [1]:

```
    sage: def hecred(u,v,N):
...       while u.degree()>genus:
```

```
...           ur=(f-v*h-v^2)//u
...           url=ur.leading_coefficient()
...           g=gcd(url,N)
...           if g != 1:
...               print "Non-trivial factor=",g
...               raise RuntimeError, g
...               return u, v, g
...
...           vr=(-h-v)%ur
...           u=ur
...           v=vr
...           print "u=", u
...           print "v=", v
...           vl=v.leading_coefficient()
...           g=gcd(vl,N)
...           if g != 1:
...               print "Non-trivial factor=",g
...               return u, v, g
...       return u,v,1
sage: def hecadd(u1,v1,u2,v2,N):
sage: #   add (u1,v1) and (u2,v2)
...       d1,e1,e2=xgcd(u1,u2)
...       d,c1,s3=xgcd(d1,v1+v2+h)
...       s1=c1*e1
...       s2=c1*e2
...       ui=u1*u2//d^2
...       vi=((s1*u1*v2+s2*u2*v1+s3*(v1*v2+f))//d)%ui
...       return hecred(ui,vi,N)
sage: def hecmul(u,v,m,N):
sage: #   multiply (u,v) by m
...       sumu, sumv = u,v
...       for z in range(m-1):
...           sumu, sumv, g = hecadd(u,v,sumu,sumv,N)
...       return sumu,sumv
sage: N=77
sage: x=polygen(Zmod(N))
sage: f=x^5+3*x+40
sage: h=0
sage: genus=2
sage: u1=x+5%N
sage: v1=0*x+1%N
sage: u=hecmul(u1,v1,9,N)[0]
```

```
sage: v=hecmul(u1,v1,9,N)[1]
sage: u,v
u= x^2 + 18*x + 18
v= x + 19
u= x^2 + 45*x + 2
v= 72*x + 29
u= x^2 + 33*x + 5
v= 15*x + 76
u= x^2 + 39*x + 42
v= 62*x + 1
u= x^2 + 74*x + 39
v= 71*x + 58
u= x^2 + 50*x + 43
v= 49*x + 76
Non-trivial factor= 7
u= x^2 + 28*x + 48
v= 20*x + 75
u= x^2 + 18*x + 18
v= x + 19
u= x^2 + 45*x + 2
v= 72*x + 29
u= x^2 + 33*x + 5
v= 15*x + 76
u= x^2 + 39*x + 42
v= 62*x + 1
u= x^2 + 74*x + 39
v= 71*x + 58
u= x^2 + 50*x + 43
v= 49*x + 76
Non-trivial factor= 7
u= x^2 + 28*x + 48
v= 20*x + 75
(x^2 + 28*x + 48, 20*x + 75)
```

Now we can see before the statement `Non-trivial factor= 7`, the leading coefficient of v is 49, which has non-trivial gcd with 77, whereas the leading coefficient can always scaled to be 1. Therefore in [1], they did not consider all possible cases.

The William's version of source code and several output is printed below. The function `hecmul()` may be modified by calculating the addition in binary partitions.

```
    sage: def hecred(u,v,N):
...       while u.degree()>genus:
...           ur=(f-v*h-v^2)//u
...           url=ur.leading_coefficient()
...           g=gcd(url,N)
...           if g != 1:
...               print "Non-trivial factor=",g
...               raise RuntimeError, g
...               return u, v, g
...
...           vr=(-h-v)%ur
...           u=ur
...           v=vr
...           vl=v.leading_coefficient()
...           g=gcd(vl,N)
...           if g != 1:
...               print "Non-trivial factor=",g
...               return u, v, g
...       return u,v,1
sage: def hecadd(u1,v1,u2,v2,N):
sage: #   add (u1,v1) and (u2,v2)
...       d1,e1,e2=xgcd(u1,u2)
...       d,c1,s3=xgcd(d1,v1+v2+h)
...       s1=c1*e1
...       s2=c1*e2
...       ui=u1*u2//d^2
...       vi=((s1*u1*v2+s2*u2*v1+s3*(v1*v2+f))//d)%ui
...       return hecred(ui,vi,N)
sage: def hecmul(u,v,m,N):
sage: #   multiply (u,v) by m
...       sumu, sumv = u,v
...       for z in range(m-1):
...           sumu, sumv, g = hecadd(u,v,sumu,sumv,N)
...       return sumu,sumv
sage: def hecptgen(N):
...      x = polygen(Zmod(N))
...      R = x.parent()
...      for i in (0..(N-1)//2):
...          r = Mod(f(i),N)
...          if r.is_square():
```

```
...               return x-i, R(sqrt(r))
sage: f = None
sage: x = None
sage: def factor_hecm(N):
...       if is_prime_power(N):
...            print "N is prime"
...            return N
...       global f, x
...       x = polygen(Integers(N))
...       f = x^5+3*x+40
...       u, v = hecptgen(N)
...       j=1
...       gc=1
...       ur=u
...       vr=v
...       while gc==1:
...           try:
...               up=ur
...               print "up=", up
...               vp=vr
...               print "vp=", vp
...               upc, vpc = hecmul(up,vp,j,N)
...               print "upc=", upc
...               print "vpc=", vpc
...               gc=hecred(up,vp,N)[2]
...               print "gc=", gc
...               print "j=",j
...               print "======"
...               j+=1
...               print "j=",j
...               ur=upc
...               vr=vpc
...           except RuntimeError, msg:
...               return msg[0]
sage: a = factor_hecm(77)
up= x + 76
vp= 11
upc= x + 76
vpc= 11
gc= 1
j= 1
======
```

```
j= 2
up= x + 76
vp= 11
upc= x^2 + 75*x + 1
vpc= 53*x + 68
gc= 1
j= 2
======
j= 3
up= x^2 + 75*x + 1
vp= 53*x + 68
Non-trivial factor= 11
Non-trivial factor= 7
sage: a
7
```

3. Conclusion

The main purpose of this project is to implement HECM in SAGE. The result is that our code can factor a 2 decimal digits composite number somehow effectively. But for larger composite numbers, this algorithm is slow, because (1) the divisor multiplication step is not clever; and the HECM, like ECM, factors a number efficiently if the number of divisor classes on the Jacobian is smooth, otherwise it may take longer time. Another problem of HECM is that for some small numbers or numbers contains power of 2 or 3 could not be factored, like in ECM.

There is also some further work to do: (1) use binary algorithm to modify the function `hecmul()`; (2) if an input number is a prime or a prime power, break the loop somewhere and output the statement: it is prime or prime power, without check before factoring. (3) how to choose a proper curve such that the number of divisor classes in the Jacobian is smooth.

References

[1] Study on Elliptic Curve and Hyperelliptic Curve Methods for Integer Factorization. Takayuki Yato, 2000-02.

[2] A Hyperelliptic Smooth Text. H.W.Lenstra, J.Pila, Carl Pomerance. Phil. Trans. R. Soc. Lond. A Novenber 15, 1993.

E-mail address: hans.cryptologiste@gmail.com