NFS with gory details

10 seconds ago by admin

NFS with gory details

Emmanuel Thomé[1], 2009

[1]CACAO, bât. A. INRIA Nancy Grand-Est, 615 rue du jardin botanique, 54602 Villers-lès-Nancy Cedex, France

This worksheet is a draft of a complete example run of (G)NFS, illustrating most difficulties which typically arise in this algorithm. It goes beyond the presentations typically found, which generally restrict to the «simplified» case where some complications of real-life GNFS are carefully avoided. This worksheet takes the opposite approach. We want to illustrate how NFS works when nearly *all* bad things occur:

- non monic defining polynomials (on both sides) ;
- maximal order of the number field not equal to the order $\mathbb{Z}[f_d\alpha]$;
- unit group of rank ≥ 1 ;
- a non-trivial 2-part of the class group.

Note yet that it's sometimes not possible to comprehensively pinpoint all the difficulties with a single example. The list of further topics at the end of this worksheet provides a list of directions which are not illustrated in this worksheet.

Setting up the number to factor and the polynomials

```
Z=Integers()
ZP.<x>=PolynomialRing(Z)
```

We'd like to do an attempt to factor something not completely ridiculous, so let's define a function to decide whether a candidate modulus is worth a try.

```
def number_is_ok_for_nfs(n):
   if not n.is_squarefree():
       return false
   f=n.factor()
```

```
   if len(f) != 2:
       return false
   if min([log(x[0])/log(n) for x in f]) > 0.4:
       return true
   return false
```

Now some code to pick a polynomial pair at random. Given a constrained leading coefficient for the algebraic polynomial, find a polynomial pair of appropriate skewness for a number having roughly the requested number of decimal digits.

```
def flog(x):
   return float(log(float(x)))

def fexp(x):
   return float(exp(float(x)))

def random_nfs_poly_pair(d,lc,nd):
   # d is the degree, nd the number of decimal digits of the integer
to factor.
   # lc is the leading coefficient.
   # returns f, g, and the skewness s
   logm=nd*flog(10)/(d+1)
   logs=(nd*flog(10)-(d+1)*abs(flog(lc)))*2/d/(d+1)
   if logs < 0:
       print "Argh, I prefer positive log-skewness (got %d). Increase
nd or decrease lc" % logs
   cf=[lc]
   s=int(fexp(logs))
   b=abs(lc)
   for i in range(1,d+1):
       b *= s
       cf.append(randint(-b,b))
   cf.reverse()
   b=int(fexp(logm)/sqrt(s))
   cg=[]
   cg.append(randint(-b,b))
   b *= s
   cg.append(randint(-b,b))
   cg.reverse()
   return ZP(cf),ZP(cg),s
```

Iterate the previous function until we get something reasonable.

```
def number_field_is_nasty_enough(f):
   t0=cputime()
   lc=f.leading_coefficient()
   d=f.degree()
```

```
   f_monic=lc^(d-1)*f(x/lc)
   K.<ah>=NumberField(f_monic)
   nr=sum([u[1] for u in f.real_root_intervals()])
   assert((d-nr) % 2 == 0)
   ns = (d-nr) / 2
   unit_rank = nr + ns - 1
   if unit_rank < 1:
       print "not good (small unit rank (%d)), try again [%.2f]" %
(U.rank(),cputime()-t0)
       return false
   # It's a pity, but we have no way of interrupting this computation
(which internally
   # does something which is much akin to nfs, by the way). We would
like to use for
   # instance alarm(), but that won't work because we're outside
python when crunching.
   h=K.class_number(proof=False)
   if h == 1:
       print "not good (principal, %s), try again [%.2f]" % (f,
cputime()-t0)
       return false
   if h % 2 != 0:
       print "not good (no 2 in h), try again [%.2f]" %
(cputime()-t0)
       return false
   return true
```

Polynomials with many roots mod small primes yield more relations, so we compute a rough measure of this fact. This measure can be much improved.

```
def poly_score(f):
   score=0
   for p in prime_range(200):
       __KP.<z>=PolynomialRing(GF(p))
       score += (1-gcd(__KP(f),z^p-z).degree())*flog(p)/(p-1)
   return score
```

```
def random_setup_we_like(d,lc,ndigits):
ntries=0
while true:
   f,g,s=random_nfs_poly_pair(d,lc,ndigits)
   ntries+=1
   if ntries % 200 == 0:
       print "%d..." % ntries
   n=f.resultant(g)
   if n < 0:
       n=-n
```

```
        g=-g
    ok = true
    ok = ok and number_is_ok_for_nfs(n)
    ok = ok and f.is_irreducible()
    ok = ok and number_field_is_nasty_enough(f)
    if ok:
        print "ok %s\n" % repr(f)
        break
return f,g,s,n
```

```
ndigits=9
lc=15
d=3
```

This is a means to create a new data set. Don't necessarily evaluate this cell, there's another one below with a pre-cooked setup.

```
# Evaluate either this one or the next one
best_score = 1000
best=()
try:
 while true:
   f,g,s,n=random_setup_we_like(d,lc,ndigits)
   score=poly_score(f)
   if (score < best_score):
       print "%s --> %.2f" % (f, score)
       if best_score == 1000:
           alarm(10)
       best_score = score
       best = (f,g,s,n)
except KeyboardInterrupt:
   print "ok, had enough"
f,g,s,n=best
m=g.roots(Rationals())[0][0]
f,g,s,n,factor(n),poly_score(f)
```

```
ok 15*x^3 + 47*x^2 + 60*x + 555

15*x^3 + 47*x^2 + 60*x + 555 --> 1.16
not good (principal, 15*x^3 - 19*x^2 - 188*x + 152), try again
[0.01]
not good (principal, 15*x^3 + 9*x^2 - 345*x + 922), try again [0.80]
ok 15*x^3 - 22*x^2 + 18*x + 128

15*x^3 - 22*x^2 + 18*x + 128 --> 1.08
not good (no 2 in h), try again [0.01]
not good (principal, 15*x^3 - 57*x^2 - 252*x + 977), try again
```

```
[0.10]
not good (principal, 15*x^3 + 43*x^2 - 191*x + 204), try again
[0.20]
not good (principal, 15*x^3 - 15*x^2 + 310*x - 1407), try again
[1.07]
200...
not good (no 2 in h), try again [1.60]
not good (principal, 15*x^3 + 74*x^2 - 363*x - 1111), try again
[0.04]
ok 15*x^3 - 52*x^2 + 235*x - 1151

not good (principal, 15*x^3 + 53*x^2 - 55*x + 1354), try again
[2.60]
ok, had enough
(15*x^3 - 22*x^2 + 18*x + 128, 31*x + 319, 5, 558032201, 6563 *
85027, 1.0787679153049718)
```

We have here some reference polynomial which is quite good ; it also helps reproducibility...

```
f,g,s,n=15*x^3 - 2*x^2 + 131*x - 366, -44*x + 127, 5, 30338281
m=g.roots(Rationals())[0][0]
f,g,s,n,factor(n),poly_score(f)
```

```
(15*x^3 - 2*x^2 + 131*x - 366, -44*x + 127, 5, 30338281, 2039 *
14879, -0.98084273158689239)
```

Definition of the number field

The number field itself, and associated objects

Now the number field. The root α of f is not an algebraic integer because f is not monic. In sage, we have to give a monic defining polynomial for the number field, thus we work with the one generated by $\hat{\alpha}$, denoted ah, which is defined as $\hat{\alpha} = f_d\alpha$. This one *is* an algebraic integer.

```
f_monic=lc^(f.degree()-1)*f(x/lc)
K.<ah>=NumberField(f_monic)
alpha=ah/lc
```

We are interested in the *maximal order* in K (a.k.a the ring of integers). Note that there is no reason for this to be equal to the order generated by $\hat{\alpha}$. The maximal order may be larger (the first easiest illustration of this being $\mathbb{Q}(\sqrt{-3})$, whose maximal order is $\mathbb{Z}[e^{2i\pi/3}]$, in which the order $\mathbb{Z}[\sqrt{-3}]$ has index 2).

```
OK=K.maximal_order()
```

```
OK_matrix=matrix([g_.vector() for g_ in OK.basis()])
OK.basis()
```

```
[1, 2/15*ah^2 + 1/15*ah, 1/2*ah^2]
```

```
EK=K.order(ah)
EK.index_in(OK)
```

```
30
```

Now because we've set it as obscure constraints above, this example has all the gory stuff happening:

```
U=K.unit_group(); U
```

```
Unit group with structure C2 x Z of Number Field in ah with defining
polynomial x^3 - 22*x^2 + 270*x + 28800
```

```
H=K.class_group(); H
```

```
Class group of order 2 with structure C2 of Number Field in ah with
defining polynomial x^3 - 22*x^2 + 270*x + 28800
```

Relation collection

We now consider the relation collection. The first step is to settle on a factor base. Therefore we have to build the list of rational primes (easy), as well as the list of algebraic primes. We are going to concentrate first and foremost on «easy» prime ideals: those which are unramified, and which yield a quotient of the ring of integers which is a prime field. Those are by far the most commonly encountered. But it's not the whole story, unfortunately...

```
B=200
rprimes=prime_range(B)
# Also need all prime ideals.
# Although refinements _are_ possible, we'll take it easy and flag
annoyances as ``to be
# handled later'', even though they can be handled systematically.
aprimes=[]
for p in rprimes:
   fp=PolynomialRing(GF(p),'z')(f)
   if fp.degree() == d:
       for u in fp.roots():
           if u[1] == 1:
               aprimes.append((p,Integers()(u[0])))
   # typical way to handle projective roots:
   # aprimes.append((p,p))
   # However one can have a double projective root which interacts
badly with a non-projective one...

def ipr_ideal(c):
   p,r=c
```

```
   if r < p:
       return OK.ideal([p,ah-lc*r])
   else:
       return OK.ideal([p,ah])

print "We need at least %d relations" % (len(rprimes)+len(aprimes))
```

```
We need at least 87 relations
```

We are going to consider candidate elements of the form $a - b\alpha$. The number of such elements to consider is chosen by us. Let's settle on an arbitrary value, say a coupe of thousands. The proper rectangle for looking for pairs is controlled by the skewness parameter s.

```
sieve_space=2^15
SA=int(sqrt(sieve_space/s/2))
print "Looking for pairs in the rectangle [0,%d] x [-%d*%s,%d*%d]\n"
% (SA,s,SA,s,SA)
```

```
Looking for pairs in the rectangle [0,57] x [-5*57,5*57]
```

Now, how do we detect that a pair (a,b) is satisfyingly smooth ? We simply consider the homogenized polynomials F, G obtained from f, g, and check their values.

```
def F(a,b):
   Z=Integers()
   za=Z(a)
   zb=Z(b)
   return Z(zb^f.degree()*f(za/zb))
def G(a,b):
   Z=Integers()
   za=Z(a)
   zb=Z(b)
   return Integers()(zb^g.degree()*g(za/zb))
def is_smooth(k,b):
   fc=k.factor()
   if len(fc) == 0: return true,fc
   t=max([u[0] for u in fc]) <= b
   return t, fc
```

Notice though that this does not exactly match the norm, as we would expect. Because α is not an algebraic integer, again. The polynomial $\prod_{\sigma \in \mathrm{Gal}(K/Q)} (X - \alpha^\sigma)$ is equal to $\frac{1}{f_d} f$, not simply f. Analogously, the norm of $(a - b\alpha)$, which is defined as $\prod_{\sigma \in \mathrm{Gal}(K/Q)} (a - b\alpha^\sigma)$, equals $\frac{1}{f_d} b^{\deg f} f(a/b)$. So the thing we're looking at is not *exactly* the norm. It's however reasonably close.

```
(3-5*alpha).norm(),F(3,5)
```

```
(3353/3, 16765)
```

Let's walk through our complete rectangle, seeking for smooth pairs. Of course, we

avoid trivial cases such as when the greatest common divisor of a and b is non-trivial. Such cases would yield spurious relations.

```
smooth_pairs=[]

for a in range(0,SA+1):
   for b in range(-SA*s,SA*s+1):
       if a == 0 or b == 0 or gcd(a,b) != 1:
           continue
       alg=F(a,b)
       rat=G(a,b)
       if rat == 0:
           continue
       ok,falg=is_smooth(alg,B)
       if ok:
           ok,frat=is_smooth(rat,B)
       if ok:
           smooth_pairs.append((Z(a),Z(b)))
print "Found %d smooth pairs" % len(smooth_pairs)
```

```
Found 128 smooth pairs
```

Making sense out of a relation

Now we're going to look more precisely into a particular smooth pair, and try to make some sense out of it.

```
a,b=smooth_pairs[randrange(len(smooth_pairs))]
a,b
```

```
(17, -122)
```

Let us be done with the rational side first, because it is so easy. We have factored $G(a,b)$, which is equal to $g_1 * (a - bm)$ (recall that m is the rational root of g). So we may safely write the complete factorization of the rational number $a - bm$, including the denominator m and a unit.

```
_,fr=is_smooth(G(a,b),B)
factor((a-b*m)*g[1]),fr
```

```
(-1 * 3 * 67 * 191, -1 * 3 * 67 * 191)
```

Now on to the algebraic side. We've been told that the only thing which really matters is the soundness of the factorization into *prime ideals*, so we're going to precisely aim at that. The thing we have computed, and asserted to be smooth, is the factorization of $F(a,b)$.

```
_,fa=is_smooth(F(a,b),B)
fa
```

```
-1 * 101 * 103 * 139 * 157
```

The ideal we want to work with (on the algebraic side) is:

```
Iab=OK.fractional_ideal(a-b*alpha)
Iab.norm(),(a-b*alpha).norm()
```

```
(227024669/15, -227024669/15)
```

Note that this ideal is not integral. It's however a property of the ring of integers that whether we speak of the norm of $(a - b\alpha)$ or the norm of the ideal $I_{a,b}$, it's the same thing. Since the factorization of the ideal $I_{a,b}$ carries over multiplicatively to the norm, a first task is to properly break up the factorization of the norm. Let us mention (again) that our integer value $F(a,b)$ does *not* correspond to the norm of $(a - b\alpha)$. We've seen that the leading coefficient comes into play and makes for the difference between the two. So this is at least one of the things to take into account if one wants to write down a sensible factorization for the ideal generated by $(a - b\alpha)$.

```
factor((a-b*alpha).norm())
```

```
-1 * 3^-1 * 5^-1 * 101 * 103 * 139 * 157
```

It is now time to introduce the handy ideal J, which is good at accounting for the defect of integrality in the ideals considered. It is defined as the inverse of the (fractional) ideal generated by 1 and α. The ideal J is an integral ideal, and a nice set of generators can be written.

```
JJ=OK.fractional_ideal([1,alpha])
J=OK.ideal([f.quo_rem(x^k)[0](alpha) for k in range(1,d+1)])
J*JJ
```

```
Fractional ideal (1)
```

It turns out that J, although not a prime ideal by itself, appears in just about every relation, because it is the denominator (or, more accurately, the denominator of the ideal factorization is a divisor of J). So multiplying by J is guaranteed to yield an integral ideal.

```
(Iab*J).is_integral(),factor((Iab*J).norm())
```

```
(True, 101 * 103 * 139 * 157)
```

We've therefore recovered exactly the thing computed by the expression $F(a,b)$.

```
def indices_for_normal_ideal_factorization(a,b):
   _,fa=is_smooth(F(a,b),B)
   Iab=OK.fractional_ideal(a-b*alpha)
   # We're now going to obtain the factorization into prime ideals.
It's very important
   # at this point that we know our ideal is integral. Because it
means that when we have
   # chopped off everything from the norm, we have indeed a unit
ideal (an ideal of norm 1
   # may differ from the unit ideal).
```

```
   # The easy part: unramified ideals of residue class degree 1 (the
ones which made it into our
   # algebraic factor base). It's quite simple to chase them.
   I_remaining=Iab*J
   i_indices=[]
   j=0
   for f in fa:
       p=f[0]
       e=f[1]
       while j < len(aprimes) and aprimes[j][0] < p:
           j += 1
       if (j == len(aprimes)):
           # exhausted the primes list
           break
       while e > 0 and j < len(aprimes) and aprimes[j][0] == p:
           r = aprimes[j][1]
           # Consider the ideal generated by p and alpha-r/lc -- or
in fact something better.
           # we consider the ideal generated by p and ah-r, which is
equal in most cases, and
           # which is always integral while the former isn't. This
ideal has norm p. If p divides the
           # leading coefficient lc, then the two ideals differ, and
their quotient is part of
           # the factorization of the ideal J.
           Ipr=ipr_ideal(aprimes[j])
           # Note that in the construction of aprimes, we've made
sure that r is a _simple_ root.
           assert Ipr.is_prime()
           if r < p:
               # on the projective line, we have the root (r:1). So
check whether a/b==r this way:
               check = a - r * b
           else:
               # Now it's (1:0). So we check this way:
               check = b
           if check % p == 0:
               # Note that this ideal may be principal, so if we try
to display this natively,
               # we might get a fairly huge generator, which we don't
really care about...
               # print "Found ideal generated by %s (mult. %d)" %
(repr(aprimes[j]), e)
               assert valuation(I_remaining,Ipr) == e
               I_remaining /= Ipr^e
               i_indices.append((j,e))
               e = 0
```

```
            j+=1
        # It might be that we don't go through the previous loop at
all. It means that no
        # well-behaving ideal correspond to this norm (most likely p
ramifies completely).
    return i_indices,I_remaining
```

```
i_indices,I_rem=indices_for_normal_ideal_factorization(a,b)
```

So, do we catch everything this way ? No, we don't (as expected). The ideal I_{rem} has non-trivial norm (sometimes, not always) thus is non trivial.

```
I_rem.norm().factor()
```

```
1
```

Part of the nasty stuff shows up at this point. Some ideals are simply not expressable in our easy form above. Fortunately these are rare. The following code calls for quite a bit of computer algebra baggage on the software side. We'll see later on that it's completely possible to forget about it using characters, so we'll be satisfied with employing a big machinery for our illustrative purpose.

```
special_ideals=[]
disc=f_monic.discriminant()
for p in primes(1,B):
    if valuation(disc,p) > 0:
        for u in factor(OK.ideal(p)):
                special_ideals.append(u[0])
#            if u[0].norm() > p or u[1] > 1:
#                special_ideals.append(u[0])
#            else:
#                # Another possibility. The ``root'' of the number
field polynomial may not
#                # uniquely identify _this_ ideal.
#                rf=u[0].residue_field()
#                FP_.<z>=PolynomialRing(rf)
#                fmp=FP_(f_monic);
#                if valuation(fmp,z-rf(ah)) > 1:
#                    special_ideals.append(u[0])

for i in special_ideals:
    print "Special ideal of norm %d, residue class degree %d,
ramification index %d" \
                %
(i.norm(),i.residue_class_degree(),i.ramification_index())
```

```
Special ideal of norm 2, residue class degree 1, ramification index
2
Special ideal of norm 2, residue class degree 1, ramification index
```

```
1
Special ideal of norm 3, residue class degree 1, ramification index
1
Special ideal of norm 3, residue class degree 1, ramification index
1
Special ideal of norm 3, residue class degree 1, ramification index
1
Special ideal of norm 25, residue class degree 2, ramification index
1
Special ideal of norm 5, residue class degree 1, ramification index
1
```

We now build a routine for killing the remaining part in the ideal factorization. Again, deciding on the valuation of an ideal at such a nasty prime ideal is not trivial (see Cohen's book for instance), but in the end it won't even be needed so we don't care too much.

```
def extra_indices_for_ideal_factorization(a,b,I):
   I_rem = I
   res=[]
   for i in range(len(special_ideals)):
       s=special_ideals[i]
       if I_rem.norm() % s.norm() == 0:
           v=valuation(I_rem,s)
           if v >= 1:
               res.append((i,v))
               I_rem /= s^v
   if not I_rem.is_trivial():
       print "Failed while trying to factor completely %s" %
repr((a,b))
       assert false
   return res

s_indices=extra_indices_for_ideal_factorization(a,b,I_rem)
```

This time, we can do the check:

```
Iab == 1/J * \
       prod([special_ideals[j[0]]^j[1] for j in s_indices]) * \
       prod([ipr_ideal(aprimes[j[0]])^j[1] for j in i_indices])
```

```
True
```

We've finally got it !

Building a matrix

Now we need to fit the pieces together and build a matrix so that an element of the nullspace can be obtained.

```
def matrix_row(a,b):
   row=[]
   i_indices, I_rem = indices_for_normal_ideal_factorization(a,b)
   s_indices=extra_indices_for_ideal_factorization(a,b,I_rem)
   fr=G(a,b).factor()
   # 0 : occurence of g_1 in the rational factorization (always)
   row.append(0)
   c = 1
   # 1 : rational unit
   if fr.unit() != 1: row.append(1)
   c = 2
   # 2..2+len(rprimes)-1 : rational primes
   for p in fr:
       if p[1] % 2 == 1:
           row.append(c+rprimes.index(p[0]))
   c += len(rprimes)
   # [c] : occurence of J (always)
   row.append(c)
   c+=1
   # [c..c+len(special_ideals)[ : special ideals
   for j in s_indices:
       if j[1] % 2 == 1:
           row.append(c + j[0])
   c += len(special_ideals)
   for j in i_indices:
       if j[1] % 2 == 1:
           row.append(c + j[0])
   return row

matrix_row(a,b)
```

```
[0, 2, 4, 10, 14, 48, 57, 58, 64, 66, 102]
```

```
ncols=len(rprimes)+2+len(aprimes)+1+len(special_ideals)
nrows=len(smooth_pairs)
mat=Matrix(GF(2),nrows,ncols)
for i in range(len(smooth_pairs)):
   a,b=smooth_pairs[i]
   row = matrix_row(a,b)
   for j in row:
       mat[i,j]=1
   if (i + 1) % 10 == 0:
       print "Done %d rows out of %d " % (i+1,nrows)
```

```
Done 10 rows out of 215
Done 20 rows out of 215
Done 30 rows out of 215
Done 40 rows out of 215
```

```
Done 50 rows out of 215
Done 60 rows out of 215
Done 70 rows out of 215
Done 80 rows out of 215
Done 90 rows out of 215
Done 100 rows out of 215
Done 110 rows out of 215
Done 120 rows out of 215
Done 130 rows out of 215
Done 140 rows out of 215
Done 150 rows out of 215
Done 160 rows out of 215
Done 170 rows out of 215
Done 180 rows out of 215
Done 190 rows out of 215
Done 200 rows out of 215
Done 210 rows out of 215
```

Solving the linear system

There isn't much to say here. We compute combinations which make all exponents in the ideal factorizations look even.
One trick though. By choosing elements $(a_i - b_i\alpha)$ as appearing either on the numberator or denominator of the final combination (instead of *not* having a fraction like that), we reduce the average valuation of ideals which appear in the factorization.
Note that element pairs which are built this way are already enough to provide a square on the rational side. On the algebraic side, the odds of getting a square are thin of course.

```
ker=mat.kernel()
def elements_from_kernel(kbasis):
   elems=[]
   ntr=0
   nta=0
   for vv in kbasis:
       elem_r=1
       elem_a=K(1)
       for i in range(nrows):
           if vv[i] > 0:
               a,b=smooth_pairs[i]
               e=(-1)^i
               elem_a *= (a - b * alpha)^e
               elem_r *= (a - b * m)^e
       elems.append((elem_r,elem_a))
```

```
        if elem_r.is_square(): ntr += 1
        if elem_a.is_square(): nta += 1
    print "We obtain a square with probabilities %.2f and %.2f,
respectively" \
          % (float(ntr/len(kbasis)),float(nta/len(kbasis)))
    return elems
elems=elements_from_kernel(ker.basis())
eta=2^len(H.elementary_divisors())
ur=U.rank()

print "2-part of the class number: %d ; unit rank: %d ; " % (eta,ur)
print "--> on the alg. side, we expect a probability 1/(%d*2^%d) =
%.2f" \
    % (eta,ur,float(1/(eta*2^ur)))
```

```
We obtain a square with probabilities 1.00 and 0.11, respectively
2-part of the class number: 4 ; unit rank: 1 ;
--> on the alg. side, we expect a probability 1/(4*2^1) = 0.12
```

We've given above an indication of what the theory says. The obstruction comes from the groups U/U^2 and H/H^2 (the latter being written additively sometimes, $H/2H$ means the same). These are isomorphic to $(\mathbb{Z}/2\mathbb{Z})^{(1+\text{rank}(U))}$ (the 1 part accounting for torsion units which always contain $\{\pm 1\}$), and to $(\mathbb{Z}/2\mathbb{Z})^{\eta}$, where η is exactly the number of parts of the decomposition of the finite abelian group H into cyclic parts. Note that η is *not* the 2-valuation of the class number !
Now above, there's apparently a glitch, because the torsion part of the unit groups does not seem to be taken into account. This is because it's in probably *already* covered by the rational character $\text{sign}(a - bm)$. At least it's what occurs in this case, because if the rational character is dropped by zeroing out the column of index one in the matrix, then the squareness probability drops on the algebraic side. It's also possible that this is a coincidence.

Back to our kernel. In case our matrix has a large kernel, we restrict our attention to the first few elements.

```
elems=[elems[i] for i in range(min(20,len(elems)))]
```

So, let's start with some of these. As mentioned, we already have a square on the rational side.

```
elem_r,elem_a=elems[1]
elem_r.sqrt(),elem_a.is_square()
```

```
(1309405387936579476 4/279212704488875885000553, False)
```

We don't expect to have a square on the algebraic side. However the ideal generated by the algebraic element *is* a square ideal, as can be seen by inspecting its valuations.

```
Ie=OK.fractional_ideal(elem_a)
```

```
fIe=Ie.factor()
[v[1] for v in fIe]
```

```
[-2, -2, -2, 2, -2, -2, -2, -2, 2, -2, 8, 4, -2, 2, 6, 4, 4, -2, 2,
4, -2, -2, -4, 4, -4, -2]
```

We're eager to consider the square root of this ideal. Unfortunately sage (pari) won't let us do that, because it insists on fetching a principal generator whenever it encounters a principal ideal. We don't care about this, so we have to resort to building the product by hand. The following is quite standard stuff for ideal arithmetic, see Cohen's book for instance.

```
A=matrix(ZZ,3,3,[0,0,1,0,1,0,1,0,0])
def matrix_gens(gg):
   return matrix(ZZ,matrix(sum([[(e*g).vector() for g in OK.gens()]
for e in gg],[]))*OK_matrix^-1)
def ideal_hnf_inner(MI):
   return A*matrix([ u for u in list((MI*A).hermite_form()*A) if not
u.is_zero()])
def ideal_hnf(I):
   ## note note note ; we expect the number of generators to be kept
reasonably
   ## small. We don't try to call gens_reduced, fearing a possible
penalty...
   MI=matrix_gens(I.gens())
   return ideal_hnf_inner(MI)
def ideal_gens_from_hnf(MI):
   return [sum([r[j]*OK.basis()[j] for j in range(d)]) for r in
MI.rows()]
def ideal_hnf_product(MI,MJ):
   gI=ideal_gens_from_hnf(MI)
   gJ=ideal_gens_from_hnf(MJ)
   gIJ=[gi*gj for gi in gI for gj in gJ]
   MIJ=matrix_gens(gIJ)
   return ideal_hnf_inner(MIJ)
def ideal_hnf_product_mixed(MI,J):
   gI=ideal_gens_from_hnf(MI)
   gJ=J.gens()
   gIJ=[gi*gj for gi in gI for gj in gJ]
   MIJ=matrix_gens(gIJ)
   return ideal_hnf_inner(MIJ)
def ideal_hnf_pow(I,e):
   gI=I.gens()
   MI=matrix_gens(gI)
   h=ideal_hnf_inner(MI)
   gh=ideal_gens_from_hnf(h)
   b=list(e.binary())
   b.reverse()
   assert b.pop() == '1'
```

```
   while len(b)>0:
       h=ideal_hnf_inner(matrix_gens([gi*gj for gi in gh for gj in
gh]))
       gh=ideal_gens_from_hnf(h)
       if b.pop() == '1':
           h=ideal_hnf_inner(matrix_gens([gi*gj for gi in gh for gj
in gI]))
           gh=ideal_gens_from_hnf(h)
   return h
def product_of_many_ideals_quick(list_of_ideals):
   h=identity_matrix(d)
   for t in list_of_ideals: h=ideal_hnf_product_mixed(h,t)
   return OK.ideal(ideal_gens_from_hnf(h))
# See cohen page 203 for this one
def ideal_inverse_hnf(M):
   T=matrix(ZZ,[[(OK.basis()[i]*OK.basis()[j]).trace() for j in
range(d)]for i in range(d)])
   Idiff=ideal_hnf(K.different())
   mul=abs(M.determinant()*K.discriminant())
   I2=matrix(ZZ,A*(T*M.transpose())^-1*mul*A).hermite_form()*A
   hprod=ideal_hnf_product(I2,Idiff)
   g=gcd([hprod[i,j] for i in range(hprod.nrows()) for j in
range(hprod.ncols())])
   hprod/=g
   mul/=g
   #I3=ideal_hnf_product_mixed(hprod,Iden)
   return hprod,mul
def fold_ideal_factorization(fac):
   hn=identity_matrix(d)
   hd=identity_matrix(d)
   for t in fac:
       e=t[1]
       I=t[0]
       h=ideal_hnf_pow(I,abs(e))
#       h=ideal_hnf_inner(matrix_gens((I^abs(e)).gens_reduced()))
       if e > 0:
           hn=ideal_hnf_product(hn,h)
       else:
           hd=ideal_hnf_product(hd,h)
   hD,D=ideal_inverse_hnf(hd)
   h=ideal_hnf_product(hn,hD)
   return h/D
```

Our routine compares quite well to what sage does for doing the same (or does not, for that matter, since it actually doesn't even terminate).

```
time hs=fold_ideal_factorization([(v[0],ZZ(v[1]/2)) for v in fIe])
```

```
hs.determinant() == sqrt(Ie.norm())
```

```
Time: CPU 1.56 s, Wall: 1.57 s
True
```

From the ideal in hermite normal form, we recover the ideal itself, and we can check that its factorization matches what we expect.

```
time Is=OK.fractional_ideal(ideal_gens_from_hnf(hs))
list(Is.factor()) == [(v[0],v[1]/2) for v in fIe]
```

```
Time: CPU 0.01 s, Wall: 0.01 s
True
```

Note that it is (in full generality) *not* a principal ideal. This is clearly an obstruction to obtaining an *element* which plays the role of a square root on the algebraic side.

```
Is.is_principal()
```

```
  ***   Warning: precision too low for generators, not given.
False
```

```
Is
```

```
Fractional ideal (390748466042751600,
1/41143527589257083540059436988l0*ah^2 +
16379577112972169690324727211303/41143527589257083540059436988l0*ah
-
964448882968285166056313114480777908711674927/6744840588402800580337\
612621)
```

However, since we have several element pairs to choose from, let's try to see what happens if we do combinations. We first compute a set of ideals corresponding to element pairs.

```
sqrt_ideals=[]
t0=cputime()
for e in elems:
   elem_r,elem_a=e
   Ie=OK.fractional_ideal(elem_a)
   fIe=Ie.factor()
   h=fold_ideal_factorization([(v[0],ZZ(v[1]/2)) for v in fIe])
   Is=OK.fractional_ideal(ideal_gens_from_hnf(h))
   sqrt_ideals.append(Is)
   print "%d, %.2f" % (len(sqrt_ideals),cputime()-t0)
   # we have Is^2 == OK.fractional_ideal(elem_a)
```

```
1, 1.79
2, 3.36
3, 5.23
4, 6.62
5, 8.17
6, 10.07
7, 12.00
8, 13.27
```

```
9, 14.76
10, 16.42
11, 18.42
12, 20.06
13, 21.48
14, 22.93
15, 24.34
16, 26.29
17, 27.82
18, 29.37
19, 31.13
20, 32.72
```

```
[I.is_principal() for I in sqrt_ideals]
```

```
  ***   Warning: precision too low for generators, not given.
[True, False, False, False, False, True, False, False, False, False,
True, False, False, False, False, False, True, False, False, False]
```

Some of them *are* principal, but in a few cases that's even too much luck because the reason is that the generator we have for the corresponding square ideal is already a square itself...

```
[(i,elems[i][1].is_square()) for i in range(len(sqrt_ideals)) if
sqrt_ideals[i].is_principal()]
```

```
[(0, True), (5, False), (10, True), (16, True)]
```

So we may look at the element of index five, because it seems that there's only one miracle happening in this case, not two (principality and squareness). Another interesting option is to look at quotients.

```
[(I/sqrt_ideals[1]).is_principal() for I in sqrt_ideals]
```

```
***   Warning: precision too low for generators, not given.
Traceback (click to the left for traceback)
...
sage.libs.pari.gen.PariError: impossible (44)
```

```
I0=sqrt_ideals[0]; e0=elems[0][1]
I1=sqrt_ideals[1]; e1=elems[1][1]
I2=sqrt_ideals[2]; e2=elems[2][1]
```

Here we have something interesting. We have overcome the principality obstruction, but the unit obstruction remains.

```
(I2/I1).is_principal(),(e2/e1).is_square()
```

```
***   Warning: precision too low for generators, not given.
Traceback (click to the left for traceback)
...
sage.libs.pari.gen.PariError: impossible (44)
```

```
(I2/I1)^2 == OK.fractional_ideal(e2/e1)
```

```
True
```

The big question is now «how far is e_2/e_1 from being a square ?». As we see, it's not a matter of torsion units here.

```
(e2/e1).is_square()
```

```
False
```

```
(-e2/e1).is_square()
```

```
False
```

The unit group has rank one, so there's an extra generator. Try it.

```
qu=(U.gens()[1]*e2/e1)
# Does not work because pari asks for 8 gigs of memory...
# qu.is_square()
```

Since sage cannot answer the question above, we code a quick squareness test. It's actually very similar to what we're going to do with characters soon !

```
def looks_like_a_square(v):
   for z in prime_range(10^4,10^4+100):
       Ip=OK.ideal(random_prime(1000)).factor()[0][0]
       if not Ip.residue_field()(v).is_square():
           return false
   return true
```

```
looks_like_a_square(qu)
```

```
True
```

Good. We have illustrated the fact that precisely *because of units*, we don't have a square.

Killing obstructions

As one knows, obstructions are easily killed by characters. Characters are just measures of squareness in arbitrary quotient rings. For computational ease, it is customary to select quotients of some special shape to do the computation, but from the theoretical point of view, just about *any* quotient of the ring of integers does the trick (any ? yes, any, even when the norm is a factor base prime. One just have to take care of valuations as well, so there's slightly more work).

```
ncharacters=10
pr=prime_range(10^4,10^4 * 2)
fields=[OK.ideal(pr[randrange(len(pr))]).factor()
[0][0].residue_field() for i in range(ncharacters)]

charmat=Matrix(GF(2),nrows,ncharacters)
for i in range(len(smooth_pairs)):
```

```
    a,b=smooth_pairs[i]
    v=a-b*alpha
    for j in range(ncharacters):
        if not fields[j](v).is_square():
            charmat[i,j]=1
    if (i + 1) % 10 == 0:
        print "Done %d rows out of %d " % (i+1,nrows)
```

```
Done 10 rows out of 215
Done 20 rows out of 215
Done 30 rows out of 215
Done 40 rows out of 215
Done 50 rows out of 215
Done 60 rows out of 215
Done 70 rows out of 215
Done 80 rows out of 215
Done 90 rows out of 215
Done 100 rows out of 215
Done 110 rows out of 215
Done 120 rows out of 215
Done 130 rows out of 215
Done 140 rows out of 215
Done 150 rows out of 215
Done 160 rows out of 215
Done 170 rows out of 215
Done 180 rows out of 215
Done 190 rows out of 215
Done 200 rows out of 215
Done 210 rows out of 215
```

We can now compute the characters corresponding to the kernel elements we have obtained so far.

```
mat2=matrix(ker.basis())*charmat
```

Note that we recover in this way that elements indexed 0,10,16 really look like squares, but that it's not true for the element of index 5.

```
for i in [0,10,16]: print mat2[i]
print mat2[5]
```

```
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
(0, 0, 1, 1, 1, 0, 0, 0, 1, 0)
```

We also note that e_2/e_1 not being a square is visible by inspecting characters:

```
mat2[2]-mat2[1]
```

```
(0, 0, 1, 1, 1, 0, 0, 0, 1, 0)
```

It's trivial to compute the second kernel -- i.e. How our kernel elements must be combined to cancel all characters.

```
ker2=mat2.kernel()
```

The first kernel element doesn't tell us much, since we've been lucky and element e_0 has all the required nice properties happening by chance. But other elements give interesting combinations.

```
ker2.basis()[1]
```

```
(0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
0)
```

So in replacement for our «old» kernel, we'd better use this one, which has all characters cancelled.

```
ker_combined=(matrix(ker2.basis())*matrix(ker.basis())).rows()
```

```
ker_combined[1]
```

```
(0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0,
1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0,
1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1,
1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0,
0, 1, 0, 1, 0, 0, 0, 0, 1)
```

```
elems_combined=elements_from_kernel(ker_combined)
```

```
We obtain a square with probabilities 1.00 and 1.00, respectively
```

It's now **much, much better** !
Pick now an element. Not all will yield non-trivial gcds, of course, but this one will.

```
elem_r,elem_a=elems_combined[8]
```

Make sure that the theory still looks right here. Notice that as far as sage is concerned, the number field is generated by the algebraic integer $\hat{\alpha} = f_d\alpha$, not by α. So when writing down an element as a polynomial, it is going to map to $\mathbb{Z}/n\mathbb{Z}$ by sending $\hat{\alpha}$ to $f_d m$.

```
Integers(n)(elem_r) == Integers(n)(elem_a.polynomial()(m*lc))
```

```
True
```

We have squares -- or more accurately, we suspect so because all our characters are zero. So let's compute square roots ! This computation is not always easy in general (to start with, computing the product is hard). Fortunately, with this toy example nothing gets too large, so it's doable and we don't have to work.

```
sr=elem_r.sqrt()
sa=elem_a.sqrt()
sa
```

```
133219155794987819608 81/124875584949375*ah^2 +
234050973224541927643 37/8325038996625*ah +
257430663282566433109 8/9098403275
```

And now, hope to get a non-trivial factor. Getting 1 or n here means bad luck.

```
gcd(n,(sa.polynomial()(m*lc)-sr) % n)
```

```
14879
```

```
n.factor()
```

```
2039 * 14879
```

That's it.

Further topics

The list of things which could be detailed as a follow-up to this text is endless. In particular, we have skipped the following interesting points:

- Polynomial selection and evaluation of relation yield
- Lattice sieving
- Projective roots
- Sieving with powers
- How to cope with most «special ideals» so that the list of ideals we don't handle gets smaller.
- How to cope with *all* ideals.
- How we can avoid having to compute the maximal order.
- Computing characters with any ideal.
- How the square root computation can be done the smart way.
- How the square root computation can be done the bovine way.
- In how far SNFS (the special Number Field Sieve) is likely to provide harder situations (large torsion unit group, non-generic Galois group, weird ramification patterns...
- And so on...

```

```