Computation of Class Groups – general toy algorithm (Math 582e)

William Stein

February 4, 2009

1 Applying LLL to computing class groups

Listing 1: Reduced Element of an Ideal

```
def ordered_embeddings(K, prec):
    """
    Return double precision real embeddings of K, followed by one
    of each conjugate pair of complex embeddings, finally followed
    by the other pairs in the same order.
    """
    phi = K.embeddings(ComplexField(prec))
    eps = 10^(-10)
    real    = [sigma for sigma in phi if abs(sigma.im_gens()[0].imag()) < eps]
    complex = [sigma for sigma in phi if abs(sigma.im_gens()[0].imag()) >= eps]
    r1, r2 = K.signature()
    if r1 != len(real):
        raise RuntimeError, "error computing real ordered embeddings"
    if 2*r2 != len(complex):
        raise RuntimeError, "error computing complex ordered embeddings"
    # Now divide the complex ones up into conjugate pairs.
    # Already they will be paired up next to each other, since the list is sorted.
    complex = [complex[2*i] for i in range(r2)] + [complex[2*i+1] for i in range(r2)]
    return real + complex

def LLL_reduce(I, v, prec):
    """
    Return short element of I along a direction v.

    INPUT:
        I -- ideal in a number field
        v -- list of real numbers

    EXAMPLES:
        sage: K.<a> = NumberField(x^4 - 2*x^2 + 3*x - 7)
        sage: I = K.ideal([19, a+9])
        sage: LLL_reduce(I, [1,1,1,1], prec=53)
        -2*a + 1
    """
    # Define quadratic form on I.
    K = I.number_field()
```

```
    n = K.degree()
    if len(v) != n:
        raise ValueError, "v must have length = degree(K)"
    sigma = ordered_embeddings(K, prec)  # double precision
    alpha = I.basis()
    R = RealField(prec)
    Q = matrix(R, n)

    r1, r2 = K.signature()
    for i in range(r1, r1+r2):
        if v[r2+i] != v[i]:
            raise ValueError, "pairs corr to conj. embeddings must be the same"

    # Exponentiate
    v = [R(x).exp() for x in v]

    # Compute quadratic form
    for i in range(len(alpha)):
        for j in range(len(alpha)):
            x = sum(v[k] * sigma[k](alpha[i]) *
                 (sigma[k](alpha[j])).conjugate() for k in range(len(v)))
            Q[i,j] = x.real()

    # LLL reduce and get change of basis matrix over ZZ
    L = pari(Q).qflllgram()._sage_()

    # Now take first column of L and take
    # that linear combination of the alpha[i]
    r = L.column(0)
    w = sum(alpha[i]*r[i] for i in range(n))
    return w
```

Listing 2: Class group using LLL

```
def primes_of_bounded_norm(K, B):
  (see previous handout)
def valuation_map(K, T):
  (see previous handout)

def class_group(f, B=None, k_bound=None, lll_rels=None, verbose=True, prec=200):
    """
    INPUT:
        f -- monic irreducible poly over ZZ
        B -- factor basis bound (default: min of Bach and Minkowski)
        k_bound -- search for smooth values of f(k) for |k| < k_bound
                   (default: 100)
        lll_rels -- number of extra LLL relations to use (default: number
                    of primes up to B).
        prec -- bits of precision in LLL ideal reduction
    OUTPUT: computation of group that covers Cl(K) using Stevenhagen Algorithm
    EXAMPLES:
        sage: z = class_group(x^4 - 2*x^2 + 3*x - 7, lll_rels=0)
```

```
    Pari gives class number =  1
    B  =  37
    #T =  14
    We find class number =  +Infinity
    Elementary invariants of quotient =  [0]
    sage: z = class_group(x^4 - 2*x^2 + 3*x - 7, lll_rels=5)
    Pari gives class number =  1
    B  =  37
    #T =  14
    We find class number =  1
"""
# normalize types and check preconditions
f = ZZ['x'](f)
if not f.is_monic(): raise ValueError, "f must be monic"

# Make the number field
K.<alpha> = NumberField(f)
if verbose: print "Pari gives class number = ", K.class_number()

# Compute bound if necessary
if B is None:
    B = floor(min(K.minkowski_bound(), 12*log(abs(K.discriminant()))^2))
if B <= 1:
    print "Obviously trivial class group"
    return [], None
if verbose: print "B  = ", B

# Compute the factor basis T using this B
T = primes_of_bounded_norm(K, B)
if verbose: print "#T = ", len(T)

# Compute the map F from the group of T-units
# to the free abelian group on T, which we view
# as #T copies of ZZ.
F = valuation_map(K, T)

# Compute the T-units coming from primes of ZZ with support in T
units_ZZ = []
primes_ZZ = prime_range(B+1)
for p in primes_ZZ:
    try:
        units_ZZ.append((K(p), F(p)))
    except ValueError: pass


# Compute smooth f(k)
if k_bound is None: k_bound = 100

primes_ZZ_set = set(primes_ZZ)
units_smooth = []
for k in [-k_bound..k_bound]:
```

```
        w = f(k)
        if set(w.prime_divisors()).issubset(primes_ZZ_set):
            try:
                units_smooth.append((alpha-k, F(alpha-k)))
            except ValueError: pass

    # Compute units coming from LLL reduction of random products of ideals
    if lll_rels is None:
        lll_rels = len(T)
    units_LLL = []
    r1, r2 = K.signature()
    import random
    for k in range(lll_rels):
        # 1. Random product I of primes in T
        I = prod(T[randint(0,len(T)-1)]^random.choice([-1,1]) for i in range(5))
        # 2. Random vector v
        v = [RDF.random_element(0.1,1) for i in range(r1+r2)]
        v += v[-r2:]  # copy the last r2 elements onto the end of v
        # 3. Short unit

        try:
            u = LLL_reduce(I, v, prec)
        except Exception, msg:
            if verbose: print "%s\nIssue reducing ideal due to precision."%msg
        else:
            try:
                units_LLL.append((u, F(u)))
            except ValueError: pass

    units = units_ZZ + units_smooth + units_LLL

    # Create relation matrix
    A = matrix([u[1] for u in units])

    # Elementary divisors gives the structure of the covering of the
    # class group
    e = A.elementary_divisors()[:A.ncols()]
    e = [i for i in e if i != 1]
    if verbose:
        h = prod(e)
        if h == 0:
            h = oo
        print "We find class number = ", h
        print "Elementary invariants of quotient = ", e
    return e, A
```

2 Other Applications

Listing 3: LLL (almost same as previous time)

```
def LLL(Q, verbose=True):
    n = Q.nrows()
```

```
    assert n >= 1 and Q.is_square(), "Q must be square and symmetric"
    e = Q.change_ring(RDF).eigenvalues()
    assert len(e) == n and min(e) > 0, "Q must be positive definite"
    # We start with the basis ZZ^n.
    V     = ZZ^n
    b     = list(V.basis())
    bstar = [V(0) for _ in range(n)]  # gram-schmidt basis
    mu    = matrix(Q.base_ring().fraction_field(), n, n)
    def dot(v,w): return (v*Q*w)
    def gram_schmidt(kmax=n-1):
        bstar[0] = b[0]
        for i in [1..kmax]:
            for j in [0..i-1]:
                mu[i,j] = dot(b[i], bstar[j]) / dot(bstar[j], bstar[j])
            bstar[i] = b[i] - sum(mu[i][j] * bstar[j] for j in [0..i-1])
    k = 1
    gram_schmidt()
    while k < n:
        # Step 1: If necessary, reduce b_k so that |mu(k,k-1)| <= 1/2
        # Now reduce b[k] by all other b[j] for j <= k-1
        for j in [k-1,k-2,..,0]:
            if abs(mu[k,j]) > 1/2:
                q = int(round(mu[k,j]))
                b[k] = b[k] - q * b[j]
                mu[k,j] = mu[k,j] - q
                for i in [0..j-1]:
                    mu[k,i] = mu[k,i] - q*mu[j,i]
        # Step 2: Check Lovasz condition
        if dot(bstar[k], bstar[k]) >= (3/4 - mu[k,k-1]^2
                    )*dot(bstar[k-1], bstar[k-1]):
            if verbose: print "k=%s; step 2 Lovasz condition satisfied"%k
            k += 1
        else:
            if verbose: print "k=%s; step 2 Lovasz failed, so swapping"%k
            b[k], b[k-1] = b[k-1], b[k]
            gram_schmidt()
            if k > 1: k -= 1
    # end while loop
    return b, mu
```

Listing 4: Using LLL to finding approximate linear relation

```
def q(x,alpha,N):
    return sum(z^2 for z in x) + N*sum(x[i]*alpha[i] for i in range(len(x)))^2

def dot(x,y, alpha,N):
    return (q(x+y, alpha,N) - q(x,alpha,N) - q(y,alpha,N))/2

def approx_relation(alpha, N):
    k = len(alpha)
    L = ZZ^k
    b = L.basis()
```

```
    R = Sequence(alpha).universe()
    Q = matrix(R,[[dot(b[i],b[j],alpha,N) for i in range(k)] for j in range(k)])
    return LLL(Q, verbose=False)[0][0]
```

Listing 5: Try out approx_relation

```
sage: R.<x> = QQ[]
sage: f = 2*x^2 + 3*x - 1
sage: f.roots(RDF)
[(-1.7807764064, 1), (0.280776406404, 1)]
sage: 0.280776406404^2
0.0788353903931442
sage: approx_relation([1,0.281,0.079], 100)
(0, 0, 1)
sage: approx_relation([1,0.281,0.079], 10000)
(1, -3, -2)
```

Listing 6: Factoring a polynomial over $\mathbf{Z}$

```
sage: R.<x> = QQ[]
sage: f = (x^3 + 2*x + 7)*(x^3 - 9*x - 5); f
x^6 - 7*x^4 + 2*x^3 - 18*x^2 - 73*x - 35
sage: f.factor()
(x^3 - 9*x - 5) * (x^3 + 2*x + 7)
sage: f.roots(RDF)
[(-2.66966384064, 1), (-1.56894640305, 1),
 (-0.576887523916, 1), (3.24655136456, 1)]
sage: alpha = f.roots(RDF)[0][0]
sage: approx_relation([1,alpha,alpha^2,alpha^3],10^6)
(-5, -9, 0, 1)
```