```
load "hu.m";

function IdealConjugate(I, K)
// Input: Z_K-Ideal I; Field K

// Output: Z_K-Ideal which is the complex conugate of I

    gens := [];
    for g in Generators(I) do
        Append(~gens, ComplexConjugate(K ! g));
    end for;

    return ideal<Integers(K)|gens>;

end function;


function ReduceByIsometry(Lattices)
// Input: List of lattices

// Output: Reduced list for which the elements are pairwise non-isometric

    LatticesReduced := [* *];
    Minima := [* *];
    NumShortest := AssociativeArray();
    SizeAuto := AssociativeArray();


    for i in [1..#Lattices] do
        L := Lattices[i];

        min_computed := false;
        minimum := 0;

        shortest_computed := false;
        shortest := 0;

        auto_computed := false;
        auto := 0;

        for j in [1..#LatticesReduced] do
            M := LatticesReduced[j];

            if not min_computed then
                min_computed := true;
                minimum := Min(L);
```

```
            end if;

            if not IsDefined(Minima, j) then
                Minima[j] := Min(M);
            end if;

            if minimum ne Minima[j] then
                continue;
            end if;


            if not shortest_computed then
                shortest_computed := true;
                shortest := #ShortestVectors(L);
            end if;

            if not IsDefined(NumShortest, j) then
                NumShortest[j] := #ShortestVectors(M);
            end if;

            if shortest ne NumShortest[j] then
                continue;
            end if;


            if not auto_computed then
                auto_computed := true;
                auto := #AutomorphismGroup(L);
            end if;

            if not IsDefined(SizeAuto, j) then
                SizeAuto[j] := #AutomorphismGroup(M);
            end if;

            if auto ne SizeAuto[j] then
                continue;
            end if;


            if IsIsometric(L, M) then
                continue i;
            end if;
        end for;

        Append(~LatticesReduced, Lattices[i]);

        NewIndex := #LatticesReduced;
```

```
        if min_computed then
            Minima[NewIndex] := minimum;
        end if;

        if shortest_computed then
            NumShortest[NewIndex] := shortest;
        end if;

        if auto_computed then
            SizeAuto[NewIndex] := auto;
        end if;

    end for;

    return LatticesReduced;
end function;


function ExtremalMinimum(l, n)
// Input: Square-free l in N; n in N

// Output: Minimum that a l-modular lattice of dimension n must have at least

    if l eq 1 then k := 24;
    elif l eq 2 then k := 16;
    elif l eq 3 then k := 12;
    elif l eq 5 then k := 8;
    elif l eq 6 then k := 8;
    elif l eq 7 then k := 6;
    elif l eq 11 then k := 4;
    elif l eq 14 then k := 4;
    elif l eq 15 then k := 4;
    elif l eq 23 then k := 2;
    end if;

    return 2 + 2*Floor(n/k);
end function;


HermiteBounds := [1, 1.1547, 1.2599, 1.1412, 1.5157, 1.6654, 1.8115, 2, 2.1327, 2.2637, 2.3934, 2.5218, 2.6494, 2.7759, 2.9015, 3.0264, 3.1507, 3.2744, 3.3975, 3.5201, 3.6423, 3.7641, 3.8855, 4.0067, 4.1275, 4.2481, 4.3685, 4.4887, 4.6087, 4.7286, 4.8484, 4.9681, 5.0877, 5.2072, 5.3267, 5.4462];
```

```
function GenSymbol(L)
// Input: Positive definite Numberfield Lattice L of square-free level

// Output: Genus symbol of L in the form [S_1, n, <2, n_2, epsilon_2, S_2, t_2>, <3, n_3, epsilon_3>, <5,...>, ...] for all primes dividing Det(L)
    Symbol := [* *];

    Rat := RationalsAsNumberField();
    Int := Integers(Rat);

    LNF := NumberFieldLatticeWithGram(Matrix(Rat, GramMatrix(L)));
    _, Grams2, Vals2 := JordanDecomposition(LNF,ideal<Int|2>);

    // Checks if all diagonal entries of the 1-component of the 2-adic jordan decomposition are even
    if Vals2[1] ne 0 or (Vals2[1] eq 0 and &and ([Valuation(Rationals() ! (Grams2[1][i][i]), 2) ge 1 : i in [1..NumberOfRows(Grams2[1])]])) then
        Append(~Symbol, 2);
    else
        Append(~Symbol, 1);
    end if;

    Append(~Symbol, Dimension(L));


    for p in PrimeDivisors(Integers() ! (Determinant(L))) do
        _, Gramsp, Valsp := JordanDecomposition(LNF, ideal<Int|p>);

        if Valsp[1] eq 0 then
            G := Matrix(Rationals(), 1/p * Gramsp[2]);
        else
            G := Matrix(Rationals(), 1/p * Gramsp[1]);
        end if;

        sym := <p, NumberOfRows(G)>;

        det := Determinant(G);
        det := Integers() ! (det * Denominator(det)^2);
```

```

        if p eq 2 then
            if IsDivisibleBy(det+1, 8) or IsDivisibleBy(det-1, 8) then
                Append(~sym, 1);
            else
                Append(~sym, -1);
            end if;

            if &and([Valuation(Rationals() ! (G[i][i]), 2) ge 1 : i in [1..sym[2]]]) then
                Append(~sym, 2);
            else
                Append(~sym, 1);
            end if;

            if sym[4] eq 2 then
                Append(~sym, 0);
            else
                Append(~sym, Integers() ! (Trace(G)*Denominator(Trace(G))^2) mod 8);
            end if;
        else
            Append(~sym, LegendreSymbol(det, p));
        end if;

        Append(~Symbol, sym);
    end for;

    return Symbol;

end function;


function ToZLattice(L)
// Input: Numberfield lattice L

// Output: L as Z-lattice
    B:= Matrix(ZBasis(L`Module));
    G:= B * L`Form * InternalConjugate(L, B);
    Form:= Matrix( Ncols(G), [ AbsoluteTrace(e) : e in Eltseq(G) ] );
    Form:=IntegralMatrix(Form);

    LZ := LatticeWithGram(LLLGram(Matrix(Integers(),Form)));

    return LZ;
```

```
end function;


function MiPoQuotient(sigma, L, p);
// Input : Automorphism sigma of L; Lattice L

// Output: Minimal polynomial of the operation of sigma on the partial dual quotient L^(#, p) / L

    sigma := Matrix(Rationals(), sigma);
    L := CoordinateLattice(L);
    LD := PartialDual(L, p : Rescale := false);
    _,phi := LD / L;
    MiPo := PolynomialRing(GF(p)) ! 1;

    B := [];

    for i in [1..Rank(LD)] do

        b := LD.i;
        if b in sub<LD|L,B> then
            continue;
        end if;
        Append(~B,b);

        dep := false;
        C := [Eltseq(phi(b))];
        while not dep do
            b := b*sigma;
            Append(~C, Eltseq(phi(b)));
            Mat := Matrix(GF(p),C);
            if Dimension(Kernel(Mat)) gt 0 then
                dep := true;
                coeff := Basis(Kernel(Mat))[1];
                coeff /:= coeff[#C];
                coeff := Eltseq(coeff);
                MiPo := LCM(MiPo, Polynomial(GF(p), coeff));
            else
                Append(~B, b);
            end if;
        end while;
    end for;

    return MiPo;

end function;
```

```
function IsModular(L, l)
// Input: Lattice L; l in N

// Output: true iff L is a l-modular lattice

    return IsIsometric(L, LatticeWithGram(l*GramMatrix(Dual(L:Rescale:=false))));

end function;

function IsStronglyModular(L,l)
// Input: Lattice L; l in N

// Output: true iff L is a strongly l-modular lattice

    return &and[IsIsometric(L, LatticeWithGram(m*GramMatrix(PartialDual(L, m : Rescale:=false)))) : m in [m : m in Divisors(l) | Gcd(m, Integers() ! (l/m)) eq 1]];

end function;




function PossibleCharPos(n, m, ListOfTypes)
// Input: n in N; m in N; List of types in format [<p_1, [a_1, a_2, ...]>, <p_2, [b_1, b_2, \dots]>, \dots] with prime divisors p_1, p_2,... of m and a_1, a_2,... denoting the dimensions of the fixed lattices of sigma^(m/p_1), b_1, b_2,... for the fixed lattices of sigma^(m/p2) and so on

// Output: List of all characteristic polynomials matching the given types. Format: [<[<p_1,a_i>, <p_2,b_j>,...], [<d_1,c_1>,...,<d_k,c_k>]>, ...] for the chosen a_i, b_j and the exponents c_l > 0 of the Phi_(d_l) for the divisors d_l

    Div := Divisors(m);
    Phi := [EulerPhi(d) : d in Div];
    t := #ListOfTypes;
    k := #Div;

    M := ZeroMatrix(Integers(), k, t+1);
    for i in [1..k] do
        for j in [1..t] do
            if IsDivisibleBy(Integers() ! (m/
```

```
ListOfTypes[j][1]), Div[i]) then
                M[i,j] := EulerPhi(Div[i]);
            end if;
        end for;
        M[i, t+1] := EulerPhi(Div[i]);
    end for;

    TypeChoice := CartesianProduct([[1..#pList[2]] : pList in ListOfTypes]);

    Results := [];

    for IndexList in TypeChoice do
        N := ZeroMatrix(Integers(), 1, t+1);
        for i in [1..t] do
            N[1][i] := ListOfTypes[i][2][IndexList[i]];
        end for;
        N[1][t+1] := n;

        C := CartesianProduct([[0..Floor(n/EulerPhi(d))] : d in Div]);

        for c in C do
            if &or[c[i] ne 0 : i in [1..k]] and Lcm([Div[i] : i in [1..k] | c[i] gt 0]) eq m then
                v := Matrix(Integers(), 1, k, [x : x in c]);
                if v*M eq N then
                    ChoiceList := [<ListOfTypes[i][1], IndexList[i]> : i in [1..t]];
                    ExpList := [<Div[i], c[i]> : i in [1..k] | c[i] gt 0];
                    Append(~Results, <ChoiceList, ExpList>);
                end if;
            end if;
        end for;
    end for;

    return Results;

end function;
```