```
load "Utility.m";
load "IdealLattices.m";


function AutomorphismTypes(l, k, n, t)
// Input: Square-free l in N, k in N, n in N, t in N

// Output: List of all possible types of automorphisms of prime order for even lattices of level l with determinant l^k, dimension n and minimum greater or equal to t
  Results := [];

  lFactors := PrimeDivisors(l);

  for p in PrimesUpTo(n+1) do
    if p in lFactors then continue; end if;

    K<z> := CyclotomicField(p);
    Kpos := sub<K|z+1/z>;

    f := [];

    for q in lFactors do
      if p le 3 then
        Append(~f, 1);
      else
        Append(~f, InertiaDegree(Factorization(ideal<Integers(Kpos) | q>)[1][1]));
      end if;
    end for;

    for np in [i*(p-1) : i in [1..Floor(n/(p-1))]] do

      n1 := n - np;
      for s in [0..Min(n1, Integers() ! (np/(p-1)))] do
        if not IsDivisibleBy(s - Integers() ! (np / (p-1)), 2) then continue s; end if;
        if p eq 2 and not IsDivisibleBy(s, 2) then continue s; end if;

        if l eq 1 then
          if n1 gt 0 then
            Gamma1 := t/p^(s/n1);
            if Gamma1 gt HermiteBounds[n1] + 0.1 then continue; end if;
          end if;

          if np gt 0 then
            Gammap := t/p^(s/np);
            if Gammap gt HermiteBounds[np] + 0.1 then continue; end if;
          end if;
          type := <p, n1, np, s>;

          Append(~Results, type);
        else
```

```
     for kp in CartesianProduct([[2*f[i]*j : j in [0..Floor(Min(np,k)/(2*f[i]))]] : i in [1..#f]]) do

      k1 := [k - kp[i] : i in [1..#kp]];

      for i in [1..#kp] do
       if k1[i] gt Min(n1,k) then continue kp; end if;
       if not IsDivisibleBy(k1[i] - k, 2) then continue kp; end if;
       if not IsDivisibleBy(kp[i], 2) then continue kp; end if;
      end for;

      if n1 gt 0 then
       Gamma1 := p^s;
       for i in [1..#lFactors] do
        Gamma1 *:= lFactors[i]^k1[i];
       end for;
       Gamma1 := t / Gamma1^(1/n1);

       if Gamma1 gt HermiteBounds[n1] + 0.1 then continue; end if;
      end if;

      if np gt 0 then
       Gammap := p^s;
       for i in [1..#lFactors] do
        Gammap *:= lFactors[i]^kp[i];
       end for;
       Gammap := t / Gammap^(1/np);

       if Gammap gt HermiteBounds[np] + 0.1 then continue; end if;
      end if;

      if p eq 2 then
       if n1 gt 0 then
        Gamma1 := 1;
        for i in [1..#lFactors] do
         Gamma1 *:= lFactors[i]^k1[i];
        end for;
        Gamma1 := t/2 / Gamma1^(1/n1);

        if Gamma1 gt HermiteBounds[n1] + 0.1 then continue; end if;
       end if;

       if np gt 0 then
        Gammap := 1;
        for i in [1..#lFactors] do
         Gammap *:= lFactors[i]^kp[i];
        end for;
        Gammap := t/2 / Gammap^(1/np);

        if Gammap gt HermiteBounds[np] + 0.1 then continue; end if;
```

```
              end if;
            end if;

            type := <p, n1, np, s>;
            for i in [1..#lFactors] do
              Append(~type, lFactors[i]);
              Append(~type, k1[i]);
              Append(~type, kp[i]);
            end for;

            Append(~Results, type);
          end for;
        end if;
      end for;
    end for;
  end for;

  return Results;

end function;


function EnumerateGenusOfRepresentative(L)
// Input: Lattice L, t in N

// Output: List of all representatives of isometry-classes in the genus of L

  "Enumerate genus of representative";
  try return eval Read(Sprintf("GenusSymbols/Gen_%o", GenSymbol(L))); catch e; end try;

  if Dimension(L) le 6 then
    Gen := GenusRepresentatives(L);
    ZGen := [];
    for M in Gen do
      if Type(M) eq Lat then
        Append(~ZGen,LLL(M));
      else
        Append(~ZGen, LatticeWithGram(LLLGram(Matrix(Rationals(), GramMatrix(SimpleLattice(M))))));
      end if;
    end for;
    PrintFileMagma(Sprintf("GenusSymbols/Gen_%o",GenSymbol(L)), ZGen : Overwrite := true);
    return ZGen;
  end if;

  M := Mass(L);
  Gen := [L];
  Explored := [false];
  NumFound := [1];
  Minima := [Minimum(L)];
    NumShortest := [#ShortestVectors(L)];
    SizeAuto := [#AutomorphismGroup(L)];
  m := 1 / SizeAuto[1];
```

```
  p := 2;

  t0 := Realtime();

  while m lt M do
    //printf "So far %o classes found. Difference to actual mass is %o. \n", #Gen, M-m;
    if Realtime(t0) ge 120*60 then
      printf "2 hours have elapsed and not the whole genus was explored. Remaining difference to actual mass is %o. %o classes were found so far. The symbol is %o.\n", M-m, #Gen, GenSymbol(L);
      return Gen;
    end if;

    RareFound := [];
    MinCount := Infinity();

    if &and(Explored) then
      "All explored. Going to next prime.";
      Explored := [false : x in Explored];
      p := NextPrime(p);
      if p ge 5 and Dimension(L) ge 8 then
        printf "Prime too large, cannot continue constructing neighbours. Remaining difference to actual mass is %o. %o classes were found so far. The symbol is %o.\n", M-m, #Gen, GenSymbol(L);
        return Gen;
      end if;
      if p ge 3 and Dimension(L) ge 12 then
        printf "Prime too large, cannot continue constructing neighbours. Remaining difference to actual mass is %o. %o classes were found so far. The symbol is %o.\n", M-m, #Gen, GenSymbol(L);
        return Gen;
      end if;
    end if;

    for i in [1..#Gen] do
      if not Explored[i] then
        if NumFound[i] lt MinCount then
          RareFound := [i];
          MinCount := NumFound[i];
        elif NumFound[i] eq MinCount then
          Append(~RareFound, i);
        end if;
      end if;
    end for;

    i := RareFound[Random(1, #RareFound)];

    Neigh := [CoordinateLattice(N) : N in Neighbours(Gen[i], p)];
    Explored[i] := true;

```

```
    for N in Neigh do

          auto := #AutomorphismGroup(N);
          if auto lt 1/(M-m) then continue; end if;

          minimum := Minimum(N);
          shortest := #ShortestVectors(N);

          for j in [1..#Gen] do

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

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

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

              if IsIsometric(N, Gen[j]) then
               NumFound[j] +:= 1;
               continue N;
              end if;
          end for;

        Append(~Gen,N);
        Append(~Explored, false);
        Append(~NumFound, 1);
        Append(~Minima, minimum);
        Append(~NumShortest, shortest);
        Append(~SizeAuto, auto);
        m +:= 1/auto;
        if m eq M then
          break N;
        end if;
       end for;
    end while;

    PrintFileMagma(Sprintf("GenusSymbols/Gen_%o",GenSymbol(L)),
Gen : Overwrite := true);

    return Gen;

end function;


function EnumerateGenusDeterminant(det, n, even)
// Input: det in N; n in N; boolean even that indicates whether
only even lattices shall be enumerated

// Output: Representatives of all isometry-classes belonging to a
genus of integral lattices with determinant det, dimension n, and
square-free level
```

```
  if n eq 0 then
    return [LatticeWithGram(Matrix(Rationals(),0,0,[]))];
  end if;

  if n eq 1 then
    L := LatticeWithGram(Matrix(Rationals(), 1, 1, [det]));
    Symbol := GenSymbol(L);
    if even and not Symbol[1] eq 2 then return []; end if;
    if not IsSquarefree(Level(L)) then return []; end if;
    if even and IsDivisibleBy(Determinant(L), 2) then
      if not Symbol[3][4] eq 2 then return []; end if;
    end if;
    return [L];
  end if;

  if n eq 2 then
    Results := [];

    for m in [1..Floor(1.155*Sqrt(det))] do
      for a in [-m+1..m-1] do

        if not IsDivisibleBy(det + a^2, m) then continue; end if;
        b := Integers() ! ((det + a^2) / m);

        if b lt m then continue; end if;
        if even and not IsEven(b) then continue; end if;

        Mat := Matrix(Rationals(), 2, 2, [m,a,a,b]);
        if not IsPositiveDefinite(Mat) then continue; end if;

        L := LatticeWithGram(Mat);

        if not IsSquarefree(Level(L)) then continue; end if;

        Symbol := GenSymbol(L);
        if even and not Symbol[1] eq 2 then continue; end if;
        if even and IsDivisibleBy(Determinant(L), 2) then
          if not Symbol[3][4] eq 2 then continue; end if;
        end if;

        Append(~Results, L);
      end for;
    end for;

    return ReduceByIsometry(Results);
  end if;


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

  primes := PrimeBasis(det);
  exps := [Valuation(det, p) : p in primes];

```

```
  IdealList := [];
  if not 2 in primes then
    Append(~IdealList, <ideal<Int|2>, [[0,n]]>);
  end if;

  for i in [1..#primes] do
    p := primes[i];
    e := Abs(exps[i]);
    if n eq e then
      Append(~IdealList, <ideal<Int|p>, [[1,e]]>);
    elif e eq 0 then
      Append(~IdealList, <ideal<Int|p>, [[0,n]]>);
    else
      Append(~IdealList, <ideal<Int|p>, [[0,n-e],[1,e]]>);
    end if;
  end for;

  "Constructing representatives";
  try
    Rep := LatticesWithGivenElementaryDivisors(Rat, n, IdealList);
  catch e
    print "Error while trying to construct a representative. IdealList:";
    IdealList;
    return [];
  end try;

  Results := [];

  for L in Rep do

    LZ := ToZLattice(L);
    if IsSquarefree(Level(LZ)) then
      Symbol := GenSymbol(LZ);
      if even and not Symbol[1] eq 2 then continue L; end if;
      if even and IsDivisibleBy(det, 2) then
        if not Symbol[3][4] eq 2 then continue L; end if;
      end if;

      Gen := EnumerateGenusOfRepresentative(LZ);
      Results cat:= Gen;
    end if;
  end for;

  return Results;

end function;


function EnumerateGenusSymbol(Symbol)
// Input: Genus-symbol Symbol of positive definite lattices of square-free level; t in N

// Output: Representatives of all isometry-classes belonging to the genus
```

```
  try return eval Read(Sprintf("GenusSymbols/Gen_%o", Symbol)); catch e; end try;

  n := Symbol[2];

  if n eq 0 then
    return [LatticeWithGram(Matrix(Rationals(),0,0,[]))];
  end if;

  if n eq 1 then
    det := &*[Symbol[i][1]^Symbol[i][2] : i in [3..#Symbol]];
    L := LatticeWithGram(Matrix(Rationals(), 1, 1, [det]));
    if GenSymbol(L) eq Symbol then
      return [L];
    end if;
    return [];
  end if;

  if n eq 2 then
    det := &*[Symbol[i][1]^Symbol[i][2] : i in [3..#Symbol]];

    for m := 2 to Floor(1.155*Sqrt(det)) by 2 do
      for a in [-m+1..m-1] do

        if not IsDivisibleBy(det + a^2, m) then continue; end if;
        b := Integers() ! ((det + a^2) / m);

        if b lt m then continue; end if;
        if not IsEven(b) then continue; end if;

        Mat := Matrix(Rationals(), 2, 2, [m,a,a,b]);
        if not IsPositiveDefinite(Mat) then continue; end if;

        L := LatticeWithGram(Mat);

        if not IsSquarefree(Level(L)) then continue; end if;

        if Symbol eq GenSymbol(L) then
          return EnumerateGenusOfRepresentative(L);
        end if;
      end for;
    end for;

    return [];

  end if;

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

  IdealList := [];
  if Symbol[3][1] ne 2 then
    Append(~IdealList, <ideal<Int|2>, [[0,n]]>);
  end if;
```

```
  for i in [3..#Symbol] do
    p := Symbol[i][1];
    np := Symbol[i][2];

    if n eq np then
      Append(~IdealList, <ideal<Int|p>, [[1,np]]>);
    elif np eq 0 then
      Append(~IdealList, <ideal<Int|p>, [[0,n]]>);
    else
      Append(~IdealList, <ideal<Int|p>, [[0,n-np],[1,np]]>);
    end if;
  end for;

  "Constructing representatives";
  try
    Rep := LatticesWithGivenElementaryDivisors(Rat, n, IdealList);
  catch e
    print "Error while trying to construct a representative. IdealList:";
    IdealList;
    return [];
  end try;

  for L in Rep do
    LZ := ToZLattice(L);
    if GenSymbol(LZ) eq Symbol then
      Gen := EnumerateGenusOfRepresentative(LZ);
      return Gen;
    end if;
  end for;

  return [];

end function;


function SuperLatticesMagma(L, p, s, sigma)
// Input: Lattice L; Prime p; s in N; Automorphism sigma of L

// Output: All even sigma-invariant superlattices of L with index p^s using magmas method


  LD := PartialDual(L,p:Rescale:=false);

  G := MatrixGroup<NumberOfRows(sigma), Integers() | sigma >;
  den1 := Denominator(BasisMatrix(LD));
  den2 := Denominator(InnerProductMatrix(LD));

  A := LatticeWithBasis(G, Matrix(Integers(), den1*BasisMatrix(LD)), Matrix(Integers(), den2^2*InnerProductMatrix(LD)));

  SU := [];
  SU := Sublattices(A, p : Levels := s, Limit := 100000);
```

```

  if #SU eq 100000 then "List of sublattices is probably not complete."; end if;

  Results := [];

  for S in SU do

    M := 1/den1 * 1/den2 * S;

    if Determinant(M)*p^(2*s) eq Determinant(L) then
      Append(~Results, M);
    end if;
  end for;

  return [L : L in Results | IsEven(L)];
end function;


function SuperLattices(L1, Lp, p, sigma1, sigmap)
// Input: Lattice L1; Lattice Lp; Prime p; Automorphism sigma of L

// Output: All even superlattices of L1 + Lp invariant under diag (sigma1, sigmap) with index p^s using isometry-method

  M := OrthogonalSum(L1, Lp);

  L1Quot, phi1 := PartialDual(L1,p : Rescale:=false) / L1;
  LpQuot, phip := PartialDual(Lp,p : Rescale:=false) / Lp;

  m := #Generators(L1Quot);

  phi1Inv := Inverse(phi1);
  phipInv := Inverse(phip);

  G1 := ZeroMatrix(GF(p),m,m);
  Gp := ZeroMatrix(GF(p),m,m);
  for i in [1..m] do
    for j in [1..m] do
      G1[i,j] := GF(p) ! (p*InnerProduct(phi1Inv(L1Quot.i), phi1Inv(L1Quot.j)));
      Gp[i,j] := GF(p) ! (-p*InnerProduct(phipInv(LpQuot.i), phipInv(LpQuot.j)));
    end for;
  end for;

  V1 := KSpace(GF(p), m, G1);
  Vp := KSpace(GF(p), m, Gp);

  O1 := IsometryGroup(V1);

  sigma1Quot := ZeroMatrix(GF(p),m,m);
  for i in [1..m] do
    sigma1Quot[i] := Vector(GF(p), Eltseq(phi1(phi1Inv
```

```
(L1Quot.i)*Matrix(Rationals(),sigma1))));
  end for;

  sigmapQuot := ZeroMatrix(GF(p),m,m);
  for i in [1..m] do
    sigmapQuot[i] := Vector(GF(p), Eltseq(phip(phipInv(LpQuot.i)*Matrix(Rationals(),sigmap))));
  end for;

  CL1Quot := Centralizer(O1, O1 ! sigma1Quot);

  CL1 := Centralizer(AutomorphismGroup(L1), sigma1);

  CL1ProjGens := [];
  for g in Generators(CL1) do
    gProj := ZeroMatrix(GF(p),m,m);
    for i in [1..m] do
      gProj[i] := Vector(GF(p), Eltseq(phi1(phi1Inv(L1Quot.i)*Matrix(Rationals(), g))));
    end for;
    Append(~CL1ProjGens, gProj);
  end for;

  CL1Proj := MatrixGroup<m, GF(p) | CL1ProjGens>;

  _, psi := IsIsometric(V1,Vp);

  psi := MatrixOfIsomorphism(psi);
  _, u := IsConjugate(O1, O1 ! sigma1Quot, O1 !(psi*sigmapQuot*psi^(-1)));

  phi0 := u*psi;

  U, mapU := CL1Quot / CL1Proj;

  LphiList := [];
  for u in U do
    phi := Inverse(mapU)(u)*phi0;

    Gens := [];
    for i in [1..m] do
      x := phi1Inv(L1Quot.i);
      y := phipInv(LpQuot ! Eltseq(phi[i]));
      Append(~Gens, Eltseq(x) cat Eltseq(y));
    end for;

    Lphi := ext<M | Gens>;
    Append(~LphiList,LatticeWithGram(LLLGram(GramMatrix(Lphi))));
  end for;

  return [L : L in LphiList | IsEven(L)];
end function;
```

```
function SuperLatticesJuergens(L, p, s)
// Input: Lattice L; Prime p; s in N; t in N

// Output: All even superlattices of L with index p^s using juergens method

  if s eq 0 then
    return [L];
  end if;

  T,mapT:=PartialDual(L,p:Rescale:=false) / L;
  mapTinv := Inverse(mapT);

    m:=#Generators(T);
    G:=GramMatrix(L);
    G_F:=MatrixAlgebra(GF(p),m)!0;

    for i:=1 to m do
        for j:=1 to m do
            G_F[i,j]:=GF(p)!(p*InnerProduct(mapTinv(T.i),mapTinv(T.j)));
        end for;
    end for;

    V:=KSpace(GF(p),m,G_F);
    if not s le WittIndex(V) then
        return [];
    end if;

    M1:=MaximalTotallyIsotropicSubspace(V);
    M:=sub< M1 | Basis(M1)[1..s] >;

    O:=IsometryGroup(V);
    Aut:=AutomorphismGroup(L:Decomposition:=true);

    Gens:=[];
    for g in Generators(Aut) do
        g_F:=MatrixAlgebra(GF(p),m)!0;
        for i:=1 to m do
            g_F[i]:=V!Vector(Eltseq(mapT(mapTinv(T!Eltseq(V.i))*Matrix(Rationals(),g))));
        end for;
        Append(~Gens,g_F);
    end for;

    O_L:=sub< O | Gens>;
    mapS,S,Kernel:=OrbitAction(O_L,Orbit(O,M));
    Set:=[Inverse(mapS)(i[2]) : i in OrbitRepresentatives(S)];
    SuperLat := [CoordinateLattice(ext< L | [mapTinv(T!Eltseq(x)) : x in Basis(W)] >) : W in Set];

    return [L : L in SuperLat | IsEven(L)];

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

// Output: List of all extremal l-modular lattices that have a large automorphism sigma of order m, such that there is a prime divisor p of m with ggT(p,l) = 1 and mu_sigma / Phi_m | (x^(m/p) - 1)
  Results := [];

  min := ExtremalMinimum(l,n);
```

```
  AutoTypes := AutomorphismTypes(l, Integers() ! (n/2), n, min);

  for phim in [Integers() ! (n/2)+1 .. n] do

    n1 := n - phim;
    np := phim;

    for m in EulerPhiInverse(phim) do

      printf "m = %o\n", m;

      for p in PrimeDivisors(m) do
        //printf "Testing p = %o\n", p;
        if Gcd(p,l) ne 1 then continue; end if;
        d := Integers() ! (m/p);
        PossibleTypes := [type : type in AutoTypes | type[1] eq p and type[2] eq n1 and type[3] eq np and (type[4] eq 0 or EulerPhi(d) le type[4])];

        //printf "Have to check %o possible automorphism-types\n", #PossibleTypes;

        for type in PossibleTypes do
          s := type[4];

          detp := p^s;
          for i := 5 to #type by 3 do
            detp *:= type[i]^type[i+2];
          end for;

          // Enumerate ideal-lattices over K(zeta_m) with given determinant
            K<z> := CyclotomicField(m);
            Kpos := sub<K | z + z^(-1)>;

              A := ClassesModGalois(K);
              M, U, FundUnits := EmbeddingMatrix(K, Kpos);
              LpList := IdealLattices(detp, K, Kpos, A, M, U, FundUnits, false);

              LpList := [L : L in LpList | Minimum(L) ge min];
            LpList := ReduceByIsometry(LpList);

            if s eq 0 then
            Results cat:= [L : L in LpList | Minimum(L) ge min];
            continue m;
            end if;

            for Lp in LpList do
            sigmapList := [c[3] : c in ConjugacyClasses(AutomorphismGroup(Lp)) | MiPoQuotient(c[3], Lp, p) eq Polynomial(GF(p), CyclotomicPolynomial(d))];
              if #sigmapList eq 0 then
```

```
              continue Lp;
              end if;
            "Enumerate candidates for L_1";

            K<z> := CyclotomicField(p);
            Kpos := sub<K|z+1/z>;

              if p eq 2 then

              // In this case use the sublattice U of L_1 with U^{#,2} = U
              det1U := 1;
              for i := 5 to #type by 3 do
                det1U *:= type[i]^type[i+1];
              end for;

              UList := EnumerateGenusDeterminant(det1U, n1, false);

              L1List := &cat[SuperLatticesJuergens(LatticeWithGram(2*GramMatrix(U)), p, Integers() ! ((n1 - s)/2)) : U in UList | Dimension(U) eq 0 or Minimum(U) ge Ceiling(min/2)];
              L1List := [L : L in L1List | Dimension(L) eq 0 or (IsEven(L) and Minimum(L) ge min)];

            elif IsPrime(l) then
              // In this case the genus symbol of L_1 is known and allows for a more efficient enumeration
              k1 := type[6];
              kp := type[7];

              if p le 3 then
                f := 1;
              else
                f := InertiaDegree(Factorization(ideal<Integers(Kpos) | l>)[1][1]);
              end if;
              deltap := (-1)^(Integers() ! (kp/f + (p-1)/2 * (Binomial(Integers() ! (np / (p-1) + 1), 2) + Binomial(s, 2))));
              delta1 := deltap * (-1)^(Integers() ! (s*(p-1)/2));

              if l eq 2 then
                if IsDivisibleBy(np + s*(p-1), 8) then
                  epsilonp := deltap;
                else
                  epsilonp := -deltap;
                end if;

                if IsDivisibleBy(n, 8) then
                  epsilon := 1;
                else
                  epsilon := -1;
                end if;
```

```
          else
            epsilonp := (-1)^(Integers() ! (kp / f + (l-1)/2*Binomial(kp,2)));

            if IsDivisibleBy(n*(l+1), 16) then
              epsilon := 1;
            else
              epsilon := -1;
            end if;
          end if;

          epsilon1 := epsilonp*epsilon;

          Sym1 := [* 2, n1 *];
          if l eq 2 then
            Append(~Sym1, <2, k1, epsilon1, 2, 0>);
            Append(~Sym1, <p, s, delta1>);
          else
            if l lt p then
              Append(~Sym1, <l, k1, epsilon1>);
              Append(~Sym1, <p, s, delta1>);
            else
              Append(~Sym1, <p, s, delta1>);
              Append(~Sym1, <l, k1, epsilon1>);
            end if;
          end if;

          L1List := [L : L in EnumerateGenusSymbol(Sym1) | Dimension(L) eq 0 or (IsEven(L) and Minimum(L) ge min)];

        else

          det1 := p^s;
          for i := 5 to #type by 3 do
            det1 *:= type[i]^type[i+1];
          end for;

          L1List := [L : L in EnumerateGenusDeterminant(det1, n1, true) | Dimension(L) eq 0 or Minimum(L) ge min];

        end if;

        for L1 in L1List do
          sigma1List := [c[3] : c in ConjugacyClasses(AutomorphismGroup(L1)) | MiPoQuotient(c[3], L1, p) eq Polynomial(GF(p), CyclotomicPolynomial(d)) and Degree(MinimalPolynomial(c[3])) le EulerPhi(d)];
          if #sigma1List eq 0 then
            continue L1;
          end if;

          "Constructing superlattices";

          if <l,n> in [] then
            for sigma1 in sigma1List do
              for sigmap in sigmapList do
```

```
                    LList cat:= SuperLatticesMagma(CoordinateLattice(OrthogonalSum(L1,Lp)), p, s, DiagonalJoin(sigma1, sigmap));
                  end for;
                end for;
                elif <l,n> in [<7,18>,<7,20>,<1,24>,<2,24>,<5,24>] then
                LList := [];
                for sigma1 in sigma1List do
                  for sigmap in sigmapList do
                    LList cat:= SuperLattices(CoordinateLattice(L1), CoordinateLattice(Lp), p, sigma1, sigmap);
                  end for;
                end for;
                else
                LList := SuperLatticesJuergens(CoordinateLattice(OrthogonalSum(L1,Lp)),p,s);
                end if;

              Results cat:= [L : L in LList | Minimum(L) ge min];
            end for;
          end for;
        end for;
      end for;
    end for;
  end for;

  return ReduceByIsometry(Results);

end function;


procedure MainLoop()
  for n := 2 to 36 by 2 do
    for l in [1,2,3,5,6,7,11,14,15,23] do
      if l eq 1 and n in [2,4,6] then continue; end if;
      if l eq 2 and n eq 2 then continue; end if;
      if l eq 11 and n in [20,24,28,30,32,34,36] then continue; end if;
      if l eq 23 and n ge 6 then continue; end if;
      printf "dim = %o, l = %o\n", n, l;
      Results := ConstructLattices(l, n);
      ModList := [L : L in Results | IsModular(L, l)];
      StrongModList := [L : L in Results | IsStronglyModular(L,l)];
      PrintFileMagma(Sprintf("SubidealLattices/%o-Modular/%o-Dimensional", l, n), Results : Overwrite := true);
      PrintFileMagma(Sprintf("SubidealLattices/%o-Modular/%o-DimensionalModular", l, n), ModList : Overwrite := true);
      PrintFileMagma(Sprintf("SubidealLattices/%o-Modular/%o-DimensionalStronglyModular", l, n), StrongModList : Overwrite := true);

```

```
      if #Results gt 0 then
        printf "\n\n----------n = %o, l = %o: %o lattices found! %o of them are modular and %o are strongly modular----------\n\n", n, l, #Results, #ModList, #StrongModList;
      end if;
    end for;
  end for;
end procedure;
```