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

                    if np gt 0 then
                        Gammap := t/p^(s/np);
                        if Gammap gt HermiteBounds[np] + 0.1 then continue s; end if;
```

```
                    end if;
                    type := <p, n1, np, s>;

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

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

                        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 s; 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 s; 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 s; 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 s; 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 for;
            end if;
        end for;
    end for;

    return Results;

end function;


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

// Output: List of all representatives of isometry-classes in the genus of L with Minimum at least t
    if Dimension(L) le 8 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;
        return [M : M in ZGen | Minimum(M) ge t];
    end if;

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

    if Minimum(L) ge t then
        GenMin := [L];
    else
        GenMin := [];
    end if;

    if m lt M then
        "Entering Kneser neighboring-method";
    end if;
```

```
    while m lt M do
        printf "Difference to actual mass is %o\n", M-m;
        RareFound := [];
        MinCount := Infinity();

        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 := Neighbours(Gen[i], 2);
        Explored[i] := true;
        for N in Neigh do
            MinAuto := 1 / (M - m);

            // Efficient isometry test follows
            min_computed := false;
            minimum := 0;

            shortest_computed := false;
            shortest := 0;

            auto_computed := false;
            auto := 0;

            for j in [1..#Gen] do
                K := Gen[j];

                if not min_computed then
                    min_computed := true;
                    minimum := Min(N);
                end if;

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

                if minimum ne Minima[j] then
```

```
                continue;
            end if;


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

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

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


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

                 if auto lt MinAuto then continue N; end if;

            end if;

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

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


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

        Append(~Gen, N);
        Append(~Explored, false);
        Append(~NumFound,1);

```

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

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

            if not auto_computed then
            auto := #AutomorphismGroup(N);
            end if;
            SizeAuto[NewIndex] := auto;
        m +:= auto;

        if Minimum(N) lt t then
            Append(~GenMin, LLL(N));
        end if;

        if m eq M then
            break N;
        end if;
        end for;
    end while;

    return GenMin;

end function;


function EnumerateGenusDeterminant(det, n, t)
// Input: det in N, n in N, t

// Output: Representatives of all isometry-classes with minimum >= tbelonging to a genus with even lattices with determinant det, dimension n, and square-free level
    if n eq 2 then

        Results := [];

        for m:= t 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;

                Symbol := GenSymbol(L);
                if not Symbol[1] eq 2 then continue; end if;
                if 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 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]]>);
```

```
        else
            Append(~IdealList, <ideal<Int|p>, [[0,n-e],[1,e]]>);
        end if;
    end for;

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

    PossibleRep := [];
    for L in Rep do

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

            Append(~PossibleRep, LZ);
        end if;
    end for;

    return &cat([EnumerateGenusOfRepresentative(L, t) : L in PossibleRep]);

end function;


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

// Output: Representatives of all isometry-classes with minimum >= t belonging to the genus

    if Symbol[2] eq 2 then
        det := &*[Symbol[i][1]^Symbol[i][2] : i in
```

```
[3..#Symbol]];

        for m:= t 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, t);
                end if;
            end for;
        end for;

        return [];

    end if;

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

    n := Symbol[2];

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

    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
            return EnumerateGenusOfRepresentative(LZ, t);
        end if;
    end for;

    return [];

end function;


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

// Output: All integral superlattices of L with index p^s and minimum at least t
    T,_,mapT:=DualQuotient(L);
    mapTinv := Inverse(mapT);
    Tp:=pPrimaryComponent(T,p);

    m:=#Generators(Tp);
    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
```

```
(Tp.i),mapTinv(Tp.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(Tp!mapT(mapTinv(T!Tp!Eltseq(V.i))*MatrixAlgebra(Rationals(),Dimension(L))!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(Tp!Eltseq(x)) : x in Basis(W)] >) : W in Set];

    return SuperLat;

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 with n/2 < phi(m) < n, 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 [m : m in EulerPhiInverse(phim) | IsDivisibleBy(m,4)] 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 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);

                    for Lp in LpList do
                    //CAp := ConjugacyClasses(AutomorphismGroup(Lp));
                    //sigmaplist := [c[3] : c in CAp | MiPoQuotient(c[3], Lp, p) eq Polynomial(GF(p),CyclotomicPolynomial(d))] do

                    // Enumerate genus

                    if IsPrime(l) and p gt 2 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];

                        f := InertiaDegree(Factorization(ideal<Integers(Kpos) | l>)[1][1]);
                            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 := EnumerateGenusSymbol(Sym1, min);
                        else
                            det1 := p^s;
                            for i := 5 to #type by 3 do
                                det1 *:= type[i]^type[i+1];
                            end for;

                            L1List := EnumerateGenusDeterminant(det1, n1, min);
                        end if;

                        for L1 in L1List do

```

```
                        M := CoordinateLattice(OrthogonalSum(L1, Lp));
                        LList := SuperLattices(M, p, s);
                        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;


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;
        List := ConstructLattices(l, n);
        ModList := [L : L in List | IsModular(L, l)];
        StrongModList := [L : L in List | IsStronglyModular(L,l)];
        if #List gt 0 then
            printf "\nn = %o, l = %o: Found %o lattices! %o of them are modular and %o are strongly modular.\n\n", n, l, #List, #ModList, #StrongModList;
        end if;
        PrintFileMagma(Sprintf("SubidealLattices/%o-Modular/%o-Dimensional", l, n), List : 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);
    end for;
```

```
end for;
```