```
load "SubidealLattices.m";

function RestrictAutomorphismTypes(l,n)
// Input: l in N; n in N

// Output: Restricts the possible automorphism types for extremal modular lattices as much as possible

  min := ExtremalMinimum(l,n);

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

  RestrictedTypes := [];

  for type in Types do
    type;
    p := type[1];
    n1 := type[2];
    np := type[3];
    s := type[4];

    if p ne 2 and IsPrime(l) then

      k1 := type[6];
      kp := type[7];

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

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

      if n1 le 12 and n1 gt 0 then
        List := [L : L in EnumerateGenusSymbol(Sym1) | Minimum(L) ge min];
        if #List eq 0 then
          continue type;
        end if;
      end if;

      if np le 12 and np gt 0 then
        List := [L : L in EnumerateGenusSymbol(Symp) | Minimum(L) ge min];
        if #List eq 0 then
          continue type;
        end if;
      end if;

    else

      if n1 le 12 and n1 gt 0 then
        det1 := p^s;
        for i := 5 to #type by 3 do
          det1 *:= type[i]^type[i+1];
        end for;

        List := [L : L in EnumerateGenusDeterminant(det1, n1, true) | Minimum(L) ge min];
        if #List eq 0 then
          continue type;
        end if;
      end if;

```

```
      if np le 12 and np gt 0 then
        detp := p^s;
        for i := 5 to #type by 3 do
          detp *:= type[i]^type[i+2];
        end for;

        List := [L : L in EnumerateGenusDeterminant(detp, np, true) | Minimum(L) ge min];
        if #List eq 0 then
          continue type;
        end if;
      end if;

    end if;

    Append(~RestrictedTypes, type);
  end for;

  return RestrictedTypes;

end function;


function PossibleCharPos(l, n)
// Input: l in N; n in N

// Output: List of all characteristic polynomials of lattices possibly not found by the subideal-lattice algorithm. Format: [[<d_1,c_1>,...,<d_k,c_k>], ...] for the exponents c_i > 0 of the Phi_(d_l) for the divisors d_l

  Types := RestrictAutomorphismTypes(l,n);

  Results := [];

  for phim in [Integers() ! (n/2)+1..n] do
    for m in EulerPhiInverse(phim) do
      Div := Sort(Divisors(m));
      Phi := [EulerPhi(d) : d in Div];
      k := #Div;

      pList := [p : p in PrimeDivisors(m) | Gcd(p,l) eq 1];
      FixDimLists := [];
      for p in pList do
        FixDims := [];
        for type in Types do
          if type[1] eq p then
            FixDim := type[2];
            if not FixDim in FixDims then
              Append(~FixDims, FixDim);
            end if;
          end if;
        end for;
        if #FixDims eq 0 then
          continue m;
        end if;
```

```
      Append(~FixDimLists, FixDims);
    end for;

    t := #pList;

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

    if t gt 0 then
      TypeChoice := CartesianProduct([[1..#List]: List in FixDimLists]);

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

        MaxDim := [Floor(n/EulerPhi(d)) : d in Div];
        for i in [1..k] do
          for j in [1..t] do
            if IsDivisibleBy(Integers() ! (m / pList[j]), Div[i]) then
              MaxDim[i] := Minimum(MaxDim[i], Floor(N[1][j] / Phi[i]));
            else
              MaxDim[i] := Minimum(MaxDim[i], Floor((n-N[1][j]) / Phi[i]));
            end if;
          end for;
        end for;

        C := CartesianProduct([[0..MaxDim[i]] : i in [1..k]]);

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

```
        end for;
      else
        C := CartesianProduct([[0..Floor(n/EulerPhi(d))] : d in Div]);
        N := Matrix(Integers(), 1, 1, [n]);
        for c in C do
          v := Matrix(Integers(), 1, k, [x : x in c]);
          if v*M eq N then
            if Lcm([Div[i] : i in [1..k] | c[i] gt 0]) eq m then
              ExpList := [<Div[i], c[i]> : i in [1..k] | c[i] gt 0];
              Append(~Results, ExpList);
            end if;
          end if;
        end for;
      end if;
    end for;
  end for;

  return Results;

end function;
```