Reflection subgroups of complex reflection groups

Don Taylor

Version: 28 November 2019

The purpose of the MAGMA code in this file is to find all simple extensions of all reflection subgroups (up to conjugacy) of a complex reflection group W. The conjugacy classes of subgroups will be identified by a name derived from the Cohen-Coxeter name of the group.

The code presumes that the order of a reflection in W is either prime or 4. This restriction means that the code should not be used for the imprimitive Shephard and Todd groups $G(m, p, n)$ where m/p is neither a prime nor 4.

To use the code, first use the function setup to obtain the complex reflection group W and a set *refreps* of generators for the cyclic subgroups generated by reflections. Next use the function rankOne to find the reflection subgroups of rank 1, up to conjugacy. Finally, use simpleExtensions to recursively find all conjugacy classes of reflection subgroups and all of their simple extensions. For example:

```
W, refreps := setup(25); // the group of type L3
names, refgroup := rankOne(W,refreps);
extension := AssociativeArray(Parent(""));
simpleExtensions(~names,~extension,~refgroup,W,refreps);
```

In this example, *names* is the sequence of names of the conjugacy classes of reflection subgroups of the Shephard and Todd group G_{25} and *refgroup* is an associative array associating a name with a representative subgroup.

The associative array *extension* associates the name of a conjugacy class with the sequence of names of its simple extensions. For example:

```
> names;
[ L1, L2, L1L1, L3, L1L1L1 ]
> extension["L1L1"];
[ L3, L1L1L1 ]
```

For convenience, there is a function printTable which carries out these actions and prints a table of simple extensions. In the table, `P` indicates a parabolic subgroup and `N` indicates a non-parabolic subgroup.

```
> printTable(25);

P | L1 | [ L1L1, L2 ]
P | L1L1 | [ L3, L1L1L1 ]
P | L2 | [ L3 ]
P | L3 | []
N | L1L1L1 | [ L3 ]
```

1 Tools

This section contains a collection of functions which apply to all complex reflection groups.

Vectors

Given a set S of vectors, find a set of representatives for the one-dimensional subspaces that they span.

```
vector_reps := function(S)
    T := {};    // representatives
    U := {};    // subspaces
    for v in S do
        X := sub<Parent(v)|v>;
        if X notin U then Include(~T, v); Include(~U, X); end if;
    end for;
    return T;
end function;
```

Orbits

Given a group G acting on a set T, find representatives for the orbits of G. The action could either be a matrix action or action by conjugation on a union of conjugacy classes of G.

```
orbit_reps := function(G, T)
    reps := [];
    while #T gt 0 do
        t := Rep(T);
        Append(~reps, t);
        S := t^G;
        T := { x : x in T | x notin S };
    end while;
    return reps;
end function;
```

Components

If G is a reflection group, the line system $\mathcal{L}$ of G has the form $\mathcal{L} = S_1 \perp \cdots \perp S_n$, where the S_i are indecomposable and pairwise orthogonal. Furthermore, $G = G_1 \times \cdots \times G_n$, where G_i is generated by the reflections r_ℓ, where $\ell \in S_i$.

Given a sequence R of reflections, return a sequence of indecomposable components.

```
components := function(R)
    seq := [];
    X := R;
    while not IsEmpty(X) do
        r := X[1];
        Exclude(~X, r);
        C := [r];
        ndx := 0;
        while ndx lt #C do
            ndx +:= 1;
            a := C[ndx];
            extn := [ x : x in X | a*x ne x*a ];
            C cat:= extn;
            X := [ x : x in X | x notin extn ];
        end while;
        Append(~seq, C);
    end while;
    return seq;
end function;
```

Extending reflection subgroups

The general idea is to look at all the simple extensions of the rank k reflection subgroups H of W and check their orbits under the action of $N_W(H)$.

The function extendGrp returns the list of extensions of H.

```
extendGrp := function(W, refreps, H)
    N := Normaliser(W, H);
    X := [ r : r in refreps | r notin H ];
    orbreps := orbit_reps(N, X);
```

For each orbit of N on the reflections not in H we produce a simple extension by adjoining a representative of the orbit. The extension is added to the list only if it is not conjugate in W to an earlier extension.

```
    return [ G : r in orbreps |
                not exists{ E : E in Self() | IsConjugate(W, E, G) }
                        where G is sub< W | H, r > ];
end function;
```

We use the function extendGrp to find all extensions of a single subgroup H. This will be applied to a sequence of subgroups already constructed to produce a larger list. Some subgroups that we produce may be conjugate to groups already in the list. Therefore the following function checks for conjugacy and returns a reduced list.

```
reduceList := func< W, mainlst, extn |
    [ G : G in extn | not exists{ M : M in mainlst | IsConjugate(W, M, G) } ] >;
```

Parabolic closure

If H is a subgroup of W, its space of fixed points is written V^H. If U is a subset of V, its pointwise stabiliser in W is written $W(U)$.

If $H \subseteq W$ is parabolic, then $H = W(U)$ for some subspace U of V. Thus $H \subseteq W(V^H) \subseteq W(U) = H$ and so $H = W(V^H)$. Therefore, if H is a reflection subgroup of W, the subgroup $W(V^H)$ is the smallest parabolic subgroup containing H; it is the *parabolic closure* of H.

```
fix := func< G | &meet[Eigenspace(r, 1) : r in Generators(G)] >;

rank := func< G | Dimension(G) - Dimension(Fix(GModule(G))) >;

parabolicClosure := function(G, K)
    F := fix(K);
    R := &join[ Class(G, r) : r in Generators(G) ];
    T := {G| r : r in R | r notin K and F subset Eigenspace(r, 1)};
    L := K;
    while #T gt 0 do
        x := Rep(T);
        L := sub<G | L, x >;
        T := { t : t in T | t notin L };
    end while;
    return L;
end function;

isParabolic := func< G, P | P eq parabolicClosure(G, P) >;
```

Identification

Tools to identify irreducible unitary reflection groups given the group order and the number of cyclic subgroups generated by a reflection.

```
groupName := AssociativeArray(CartesianProduct(Integers(), Integers()));
groupName[<2, 1>] := "A1";    // ShephardTodd(1,1,2)
groupName[<3, 1>] := "L1";    // ShephardTodd(3,1,1)
groupName[<4, 1>] := "Z4";    // ShephardTodd(4,1,1)
groupName[<5, 1>] := "Z5";    // ShephardTodd(5,1,1)
groupName[<6, 3>] := "A2";    // ShephardTodd(1,1,3)
groupName[<8, 4>] := "B2";    // ShephardTodd(2,1,2)
```

```
groupName[<10,5>] := “D2(5)”;    // ShephardTodd(5,5,2)
groupName[<12,6>] := “D2(6)”;    // ShephardTodd(6,6,2)
groupName[<16,6>] := “B2(4)”;    // ShephardTodd(4,2,2)
groupName[<16,8>] := “G(8,8,2)”;    // ShephardTodd(8,8,2)
groupName[<18,5>] := “B2(3)”;    // ShephardTodd(3,1,2)
groupName[<20,10>] := “G(10,10,2)”;    // ShephardTodd(10,10,2)
groupName[<24,4>] := “L2”;    // ShephardTodd(4)
groupName[<24,6>] := “A3”;    // ShephardTodd(1,1,4)
groupName[<32,6>] := “G(4,1,2)”;    // ShephardTodd(4,1,2)
groupName[<32,10>] := “G(8,4,2)”;    // ShephardTodd(8,4,2)
groupName[<36,8>] := “G(6,2,2)”;    // ShephardTodd(6,2,2)
groupName[<48,9>] := “B3”;    // ShephardTodd(2,1,3)
groupName[<48,10>] := “G6”;    // ShephardTodd(6)
groupName[<48,12>] := “G12”;    // ShephardTodd(12)
groupName[<50,7>] := “G(5,1,2)”;    // ShephardTodd(5,1,2)
groupName[<54,9>] := “D3(3)”;    // ShephardTodd(3,3,3)
groupName[<64,10>] := “G(8,2,2)”;    // ShephardTodd(8,2,2)
groupName[<72,8>] := “G5”;    // ShephardTodd(5)
groupName[<96,6>] := “G8”;    // ShephardTodd(8)
groupName[<96,12>] := “D3(4)”;    // ShephardTodd(4,4,3)
groupName[<96,18>] := “G13”;    // ShephardTodd(13)
groupName[<100,12>] := “G(10,2,2)”;    // ShephardTodd(10,2,2)
groupName[<120,10>] := “A4”;    // ShephardTodd(1,1,5)
groupName[<120,15>] := “H3”;    // ShephardTodd(23)
groupName[<144,14>] := “G7”;    // ShephardTodd(7)
groupName[<144,20>] := “G14”;    // ShephardTodd(14)
groupName[<162,12>] := “B3(3)”;    // ShephardTodd(3,1,3)
groupName[<192,12>] := “D4”;    // ShephardTodd(2,2,4)
groupName[<192,15>] := “B3(4)”;    // ShephardTodd(4,2,3)
groupName[<192,18>] := “G9”;    // ShephardTodd(9)
groupName[<240,30>] := “G22”;    // ShephardTodd(22)
groupName[<288,14>] := “G10”;    // ShephardTodd(10)
groupName[<288,26>] := “G15”;    // ShephardTodd(15)
groupName[<336,21>] := “J3(4)”;    // ShephardTodd(24)
groupName[<360,20>] := “G20”;    // ShephardTodd(20)
groupName[<384,16>] := “B4”;    // ShephardTodd(2,1,4)
groupName[<576,26>] := “G11”;    // ShephardTodd(11)
groupName[<600,12>] := “G16”;    // ShephardTodd(16)
groupName[<648,12>] := “L3”;    // ShephardTodd(25)
groupName[<648,18>] := “D4(3)”;    // ShephardTodd(3,3,4)
groupName[<720,15>] := “A5”;    // ShephardTodd(1,1,6)
groupName[<720,50>] := “G21”;    // ShephardTodd(21)
groupName[<1152,24>] := “F4”;    // ShephardTodd(28)
groupName[<1200,42>] := “G17”;    // ShephardTodd(17)
groupName[<1296,21>] := “M3”;    // ShephardTodd(26)
groupName[<1536,24>] := “D4(4)”;    // ShephardTodd(4,4,4)
groupName[<1800,32>] := “G18”;    // ShephardTodd(18)
```

```
  groupName[<1920,20>] := "D5";   // ShephardTodd(2,2,5)
  groupName[<2160,45>] := "J3(5)";   // ShephardTodd(27)
  groupName[<3072,28>] := "B4(4)";   // ShephardTodd(4,2,4)
  groupName[<3600,62>] := "G19";   // ShephardTodd(19)
  groupName[<5040,21>] := "A6";   // ShephardTodd(1,1,7)
  groupName[<7680,40>] := "N4";   // ShephardTodd(29)
  groupName[<9720,30>] := "D5(3)";   // ShephardTodd(3,3,5)
  groupName[<14400,60>] := "H4";   // ShephardTodd(30)
  groupName[<23040,30>] := "D6";   // ShephardTodd(2,2,6)
  groupName[<40320,28>] := "A7";   // ShephardTodd(1,1,8)
  groupName[<46080,60>] := "O4";   // ShephardTodd(31)
  groupName[<51840,36>] := "E6";   // ShephardTodd(35)
  groupName[<51840,45>] := "K5";   // ShephardTodd(33)
  groupName[<155520,40>] := "L4";   // ShephardTodd(32)
  groupName[<174960,45>] := "D6(3)";   // ShephardTodd(3,3,6)
  groupName[<322560,42>] := "D7";   // ShephardTodd(2,2,7)
  groupName[<362880,36>] := "A8";   // ShephardTodd(1,1,9)
  groupName[<2903040,63>] := "E7";   // ShephardTodd(36)
  groupName[<5160960,56>] := "D8";   // ShephardTodd(2,2,8)
  groupName[<39191040,126>] := "K6";   // ShephardTodd(34)
  groupName[<696729600,120>] := "E8";   // ShephardTodd(37)

name := func< n, r |
  IsDefined(groupName, <n,r>) select groupName[<n,r>]
    else "<"*IntegerToString(n)*"|"*IntegerToString(r)*">" >;
```

2 Setting up

Begin by setting up the root system and reflection group for the Shephard and Todd group number n.

The function setup returns the complex reflection group W and *refreps*, where *refreps* is a set of representatives for the generators of the cyclic subgroups generated by reflections.

```
setup := function(n)
  roots, coroots, ρ, W, J := ComplexRootDatum(n);
```

The roots are constructed within a vector space with the standard inner product and therefore the universe needs to be changed so that the correct inner product will be used.

Added σ
25 Oct 2016

```
  F := BaseRing(J);
  if F eq Rationals() then
    V := VectorSpace(BaseRing(J), Nrows(J), J);
  else
    σ := map< F -> F | x :-> ComplexConjugate(x) >;
    V := UnitarySpace(J, σ);
  end if;
```

```
        ChangeUniverse(~roots, V);
        ChangeUniverse(~coroots, V);
        ρ := map< roots → coroots | a ↦ ρ(a) >;
        Φ := vector_reps(roots);
        R := {@ W ! PseudoReflection(a,ρ(a)) : a in Φ @};
        R join:= {@ r^2 : r in R | Order(r) eq 4 @};
        return W, R;
    end function;
```

3 Standard names

We keep track of the reflection subgroups via their names. For example, if H is the first reflection subgroup of W found to have type A_3 it will have name A3. If K is another subgroup of type A_3, not conjugate in W to H, it will have name A3-2, and so on.

Return the concatenation of the sorted list of names of the components of H.

```
    getTag := function(W, refreps, H)
        sform := [];
        R := { r : r in refreps | r in H };
        R := Setseq( R diff { r^2 : r in R | r^2 in R } );
        for C in components(R) do
            if Generators(W) subset C then
                sform := [name(#W,#C)]; break;
            else
                Append(~sform, name(Order(sub<W|C>),#C));
            end if;
        end for;
        return &cat Sort(sform);
    end function;
```

The parameter *refgroup* in the functions which follow is an associative array which associates the name of a conjugacy class of reflection subgroups with a representative subgroup.

The function sName returns the name of the group H.

```
    sName := function(W, refreps, refgroup, H)
        tag := getTag(W, refreps, H);
        error if tag notin Keys(refgroup), "sName: group not in list";
        base := tag;
        ndx := 1;
        while not IsConjugate(W, refgroup[tag], H) do
            ndx +:= 1;
            tag := base*"-"*IntegerToString(ndx);
        end while;
        return tag;
    end function;
```

The function newName creates a new name for H.

```
newName := function(W, refreps, refgroup, H)
    tag := getTag(W, refreps, H);
    base := tag;
    ndx := 1;
    while tag in Keys(refgroup) do
        ndx +:= 1;
        tag := base * "-" * IntegerToString(ndx);
    end while;
    return tag;
end function;
```

4 Reflection subgroups of rank 1

The function rankOne is used to begin the process of finding all reflection subgroups of W and their simple extensions.

```
rankOne := function(W, refreps)
    refgroup := AssociativeArray(Parent(""));
```

Collect the rank 1 reflection subgroups.

```
    rank1 := [ sub<W|r> : r in orbit_reps(W, refreps) ];
```

Assign names to the subgroups.

```
    names := [];
    for H in rank1 do
        tag := newName(W, refreps, refgroup, H);
        refgroup[tag] := H;
        Append(~names, tag);
    end for;
    return names, refgroup;
end function;
```

5 Main procedure

The list of simple extensions of a conjugacy class of reflection subgroups of a given type is held in the following associative array.

```
extension := AssociativeArray(Parent(""));
```

Added VERBOSE option 28 Nov 2019

The function simpleExtensions begins with the list of conjugacy classes of reflection subgroups in *subtypes* and recursively finds representatives (in *refgroup*) of all simple extensions (in *extension*). If VERBOSE is true, diagnostic information is printed.

```
simpleExtensions := procedure(~subtypes, ~extension, ~refgroup, W, refreps :
    VERBOSE := false)
  ndx := 0;
  subgroups := [refgroup[tag] : tag in subtypes];
  while ndx lt #subtypes do
    ndx +:= 1;
    tag := subtypes[ndx];
    if VERBOSE then print "Extending:", tag; end if;
    H := refgroup[tag];
    extn := extendGrp(W, refreps, H);
    extn2 := reduceList(W, subgroups, extn);
    if #extn2 gt 0 then
      for H in extn2 do
        Append(~subgroups, H);
        etag := newName(W, refreps, refgroup, H);
        refgroup[etag] := H;
        Append(~subtypes, etag);
      end for;
    end if;
    extension[tag] := [ sName(W, refreps, refgroup, H) : H in extn ];
    if VERBOSE then print "  ", extension[tag]; end if;
  end while;
end procedure;
```

The procedure testext checks to see if there is a non-parabolic subgroup with a parabolic simple extension of larger rank.

```
testext := procedure(names, refgroup, extension, W)
  for tag in names do
    H := refgroup[tag];
    if not isParabolic(W, H) then
      m := rank(H);
      for extn in extension[tag] do
        K := refgroup[extn];
        if rank(K) gt m then
          flag := isParabolic(W, K);
          print tag, extn, flag;
          if flag then print "~~~COUNTER EXAMPLE~~~~~~~~~"; end if;
        end if;
      end for;
    end if;
  end for;
end procedure;
```

The function extends checks whether the group with name B is a simple extension of the group with name A.

```
extends := function(A, B, refgroup, extension, W)
  flag, G := IsDefined(refgroup, B);
  if flag then
    flag, E := IsDefined(extension, A);
    if flag then
        return exists(X){ X : X in E | IsConjugate(W, X, G) };
    else
      return "Extensions not defined";
    end if;
  else
    return "Identity not known";
  end if;
end function;
```

6 Tables

In this section we define a procedure which computes the simple extensions for a given Shephard and Todd group and then prints the results as a table.

For finer control over the process, use the functions and procedures from the preceding sections.

```
getList := function(n : Verbose := false)
  error if n notin [4..37], "n must be in the range 4..37";
  W, refreps := setup(n);
  names, refgroup := rankOne(W, refreps);
  extension := AssociativeArray(Parent(""));
  simpleExtensions(~names, ~extension, ~refgroup, W, refreps :
    Verbose := Verbose);
  return names, refgroup, extension, W, refreps;
end function;

printTable := procedure(n : Verbose := false)
  names, refgroup, extension, W, _ := getList(n : Verbose := Verbose);
  for X in names do
    H := refgroup[X];
    para := isParabolic(W, H) select "P" else "N";
    print para, "|", X, "|", extension[X];
  end for;
end procedure;
```

Revision history

2016-10-25 Added complex conjugation σ to setup.

2019-11-28 Added a VERBOSE option to simpleExtensions, getList and printTable.