Rig

A GAP4 Package

Version 0.6

by

Matas Graña

Departamento de matemática, FCEyN

Universidad de Buenos Aires

matiag@dm.uba.ar, http://mate.dm.uba.ar/~matiasg

and

Leandro Vendramin

Departamento de matemática, FCEyN

Universidad de Buenos Aires

lvendramin@dm.uba.ar, http://mate.dm.uba.ar/~lvendram

February 2011

Contents

1 Introduction

The GAP 4 package RiG provides a library of functions for computations related to racks. This package can also be used to make basic calculations of cohomology of finite racks.

In chapter 2 we give a short summary of parts of the theory of finite racks and Nichols algebras. This fixes the notations and definitions that we use. In chapter 3 we describe the functions that constitute the package.

To load the package, type:

```
gap> LoadPackage("rig");
--
RiG, A GAP package for racks, Version 0.6
http://mate.dm.uba.ar/~lvendram/rig
true
```

2 Background

In this chapter we summarize some of the theoretical concepts with which RiG operates.

3 List of functions

The following is a list of the functions available in the RiG package.

1 ▶ `AbelianRack(` *n* `)` ;
▶ `TrivialRack(` *n* `)` ;

creates an abelian rack of size n. For example:

```
gap> r := TrivialRack(3);;
gap> Display(r);
rec(
  isRack := true,
  matrix := [ [ 1, 2, 3 ], [ 1, 2, 3 ], [ 1, 2, 3 ] ],
  labels := [ 1 .. 3 ],
  size := 3,
  basis := "",
  comments := "",
  inn := "",
  aut := "",
  env := "" )
```

2 ▶ `AffineCyclicRack(` *n*, *x* `)`

creates an affine rack associated to the cyclic group of order n and the multiplication by x. For example, the dihedral rack of size 3 can be obtained as:

```
gap> r := AffineCyclicRack(3,2);;
gap> Display(r);
rec(
  isRack := true,
  matrix := [ [ 1, 3, 2 ], [ 3, 2, 1 ], [ 2, 1, 3 ] ],
  labels := [ 1 .. 3 ],
  size := 3,
  basis := "",
  comments := "",
  inn := "",
  aut := "",
  env := "" )
```

3 ▶ `AffineRack(` *field*, *field_element* `)`

creates an affine rack associated to *field* and the multiplication by the *field_element*. For example:

```
gap> r := AffineRack(GF(3), Z(3));;
gap> Display(r);
rec(
  isRack := true,
  matrix := [ [ 1, 3, 2 ], [ 3, 2, 1 ], [ 2, 1, 3 ] ],
  labels := [ 1 .. 3 ],
  size := 3,
  basis := "",
  comments := "",
  inn := "",
  aut := "",
  env := "" )
```

4 ► `AlexanderRack(` *n*, *s*, *t* `)`

returns the Alexander rack of size n associated to s and t. If $s = 1 - t$ then the result is the Alexander quandle.

5 ► `CyclicRack(` *n* `)`

returns the cyclic rack of order n. For example:

```
gap> r := CyclicRack(3);;
gap> Display(r);
rec(
  isRack := true,
  matrix := [ [ 2, 3, 1 ], [ 2, 3, 1 ], [ 2, 3, 1 ] ],
  labels := [ 1 .. 3 ],
  size := 3,
  basis := "",
  comments := "",
  inn := "",
  aut := "",
  env := "" )
```

6 ► `CoreRack(` *group* `)`

returns the core rack of the group *group*. For example:

```
gap> r := CoreRack(CyclicGroup(3));;
gap> Display(r);
rec(
  isRack := true,
  matrix := [ [ 1, 3, 2 ], [ 3, 2, 1 ], [ 2, 1, 3 ] ],
  labels := [ 1 .. 3 ],
  size := 3,
  basis := "",
  comments := "",
  inn := "",
  aut := "",
  env := "" )
```

7 ► `DihedralRack(` *n* `)`

creates a dihedral rack of size n. For example:

```
gap> r := DihedralRack(5);;
gap> Display(r.matrix);
[ [  1,  5,  4,  3,  2 ],
  [  3,  2,  1,  5,  4 ],
  [  5,  4,  3,  2,  1 ],
  [  2,  1,  5,  4,  3 ],
  [  4,  3,  2,  1,  5 ] ]
```

8 ► `DirectProductOfRack(` *rack1* `,` *rack2* `)`

returns the direct product of *rack1* and *rack2*. For example:

```
gap> r := DirectProductOfRacks(DihedralRack(3), TrivialRack(2));;
gap> Display(r.matrix);
[ [  1,  2,  5,  6,  3,  4 ],
  [  1,  2,  5,  6,  3,  4 ],
  [  5,  6,  3,  4,  1,  2 ],
  [  5,  6,  3,  4,  1,  2 ],
  [  3,  4,  1,  2,  5,  6 ],
  [  3,  4,  1,  2,  5,  6 ] ]
```

9 ► `HomogeneousRack(` *group*`,` *automorphism* `)`
► `TwistedHomogeneousRack(` *group*`,` *automorphism* `)`

creates a (twisted)homogeneous rack associated to the *automorphism* of the *group* given. For example:

```
gap> f := ConjugatorAutomorphism(SymmetricGroup(3), (1,2));;
gap> r := HomogeneousRack(SymmetricGroup(3), f);;
gap> Display(r.matrix);
[ [  1,  6,  3,  5,  4,  2 ],
  [  4,  2,  6,  1,  5,  3 ],
  [  1,  6,  3,  5,  4,  2 ],
  [  5,  3,  2,  4,  1,  6 ],
  [  4,  2,  6,  1,  5,  3 ],
  [  5,  3,  2,  4,  1,  6 ] ]
```

10 ► `RackByListOfPermutations(` *list* `)`

returns the rack given by the list of permutations given in *list*. For example:

```
gap> r := RackFromListOfPermutations([(2,3),(1,3),(1,2)]);;
gap> Display(r.matrix);
[ [  1,  3,  2 ],
  [  3,  2,  1 ],
  [  2,  1,  3 ] ]
```

11 ► `Rank(`*rack*`)`

return the rank of *rack*. If *rack* is a quandle, the result is 1.

12 ► `AutomorphismGroup(` *rack* `)`

returns the group of automorphism of *rack*. For example:

```
gap> AutomorphismGroup(TrivialRack(3));
Group([ (), (2,3), (1,2), (1,2,3), (1,3,2), (1,3) ])
```

13 ► `InnerGroup(` *rack* `)`

returns the inner group of the rack. For example

```
gap> InnerGroup(DihedralRack(3));
Group([ (2,3), (1,3), (1,2) ])
gap> InnerGroup(TrivialRack(5));
Group(())
```

14 ▶ `IsomorphismRack(` *r*, *s* `)`

computes an isomorphism between the racks *r* and *s* if they are isomorphic and returns `fail` otherwise.

```
gap> a := Rack(AlternatingGroup(4), (1,2,3));;
gap> b := Rack(AlternatingGroup(4), (1,3,2));;
gap> c := AbelianRack(4);
gap> IsomorphismRacks(a,b);
(3,4)
gap> IsomosphismRacks(a,c);
fail
```

15 ▶ `Rack(` *matrix* `)` F

creates a rack structure over the set $X = \{1, \cdots, n\}$ with the structure given by *matrix*. For example, to get the abelian rack of two elements:

```
gap> a := AbelianRack(2);;
gap> b := Rack([[1,2],[1,2]]);;
gap> Display(b.matrix);
[ [  1,  2 ],
  [  1,  2 ] ]
gap> a=b;
true
```

16 ▶ `Rack(` *group*, *group_element* `)` F

creates a rack structure from the conjugacy class in *group* of *group_element*. For example, the rack associated to the vertices of the tetrahedron is the rack of the conjugacy class of (1,2,3) in the alternating group in four letters:

```
gap> r := Rack(AlternatingGroup(4), (1,2,3));;
gap> Display(r.matrix);
[ [  1,  3,  4,  2 ],
  [  4,  2,  1,  3 ],
  [  2,  4,  3,  1 ],
  [  3,  1,  2,  4 ] ]
```

17 ▶ `Rack(` *set* `)`

creates a rack structure from the elements of the *set*. For example:

```
gap> set := Set([(1,2),(2,3),(1,3)]);;
gap> Rack(set) = Rack(SymmetricGroup(3), (1,2));
true
```

18 ▶ `BoundaryMap(` *rack*, *n* `)` F

computes the rack boundary map.

19 ▶ `RackHomology (` *rack*, *order* `)` F
▶ `RackCohomology (` *rack*, *order* `)` F

computes the abelian rack (co)homology.

```
gap> RackCohomology(DihedralRack(3),2);
[ 1, [  ] ]
gap> RackCohomology(TrivialRack(3),2);
[ 9, [  ] ]
gap> RackHomology(TrivialRack(2),2);
[ 4, [  ] ]
gap> RackHomology(DihedralRack(4),2);
[ 4, [ 2, 2 ] ]
```

20 ▶ `QuantumSymmetrizer(` *rack*, *q*, *n* `)`

computes the quantum symmetrizer of degree n.

21 ▶ `Dimension(` *nichols_datum*, *n* `)`

computes the dimension in degree n of the Nichols algebra *nichols_datum* In the following example we compute all dimensions of a known 12-dimensional Nichols algebra.

```
gap> r := DihedralRack(3);;
gap> q := [ [ -1, -1, -1 ], [ -1, -1, -1 ], [ -1, -1, -1 ] ];;
gap> n := NicholsDatum(r, q, Rationals);;
gap> for i in [0..5] do
Print("Degree ", i, ", dimension=", Dimension(n,i), "\n");
od;
Degree 0, dimension=1
Degree 1, dimension=3
Degree 2, dimension=4
Degree 3, dimension=3
Degree 4, dimension=1
Degree 5, dimension=0
```

22 ▶ `Relations4GAP(` *nichols_datum*, *n* `)`

returns the relation in degree n of the Nichols algebra *nichols_datum* The relations are returned in gbnp format. In the following example we calculate all relations in degree two of a 12-dimensional Nichols algebra.

```
gap> LoadPackage("gbnp");
gap> r := DihedralRack(3);;
gap> q := [ [ -1, -1, -1 ], [ -1, -1, -1 ], [ -1, -1, -1 ] ];;
gap> rels := Relations4GAP(r, q, 2);;
gap> PrintNPList(rels);
 a^2
 b^2
 ab + bc + ca
 ac + ba + cb
 c^2
```

23 ▶ `RackOrbit(` *rack*, *i* `)`

returns the orbit of the element i, given by the action of the inner group.

```
gap> r := DihedralRack(4);;
gap> RackOrbit(r, 1);
[1, 3]
```

24 ▶ `Nr_k (` *rack*, *n* `)`

returns the number of j such that the braided orbit of $(1, j)$ has n elements.

```
gap> Check := function(rack)
> local n,s,d;
> d := Size(rack);
> s := 0;
> for n in [3..d^2] do
>   s := s+(n-2)*Nr_k(rack,n)/(2*n);
> od;
> if s <= 1 then
>   return s;
> else
>   return fail;
> fi;
> end;
gap> a4 := AlternatingGroup(4);;
gap> s4 := SymmetricGroup(4);;
gap> s5 := SymmetricGroup(5);;
gap> Check(RackFromAConjugacyClass(a4, (1,2,3)));
1/2
gap> Check(RackFromAConjugacyClass(s4, (1,2)));
2/3
gap> Check(RackFromAConjugacyClass(s4, (1,2,3,4));
2/3
gap> Check(RackFromAConjugacyClass(s5, (1,2)));
1
gap> Check(AffineCyclicRack(5,2));
1
gap> Check(AffineCyclicRack(5,3));
1
gap> Check(AffineCyclicRack(7,5));
1
gap> Check(AffineCyclicRack(7,3));
1
gap> Check(DihedralRack(3));
1/3
```

25 ▶ `Nr_l` (*rack*, *n*)

returns the number of braided orbits with n elements, when *rack* is of group-type.

26 ▶ `Braiding` (*rack*)

returns the braiding given by *rack*.

```
gap> Display(Braiding(TrivialRack(2)));
[ [  1,  0,  0,  0 ],
  [  0,  0,  1,  0 ],
  [  0,  1,  0,  0 ],
  [  0,  0,  0,  1 ] ]
```

27 ▶ `SubracksUpToIso` (*rack*, *subr*, *n*)

returns all the subracks of *rack* containing *subr* of size less or equal than n (up to rack isomorphism).

```
gap> r := DihedralRack(8);;
gap> subracks := SubracksUpToIso(r,[1],8);;
gap> for s in subracks do
> Print(Size(s),"\n");
> od;
1
8
4
2
```

28 ▶ `IsConnected` (*rack*)
▶ `IsIndecomposable` (*rack*)

returns true if the rack is indecomposable (i.e. connected).

```
gap> r := DihedralRack(3);;
gap> IsIndecomposable(r);
true
gap> s := DihedralRack(4);;
gap> IsIndecomposable(s);
false
```

29 ▶ `IsHomogeneous` (*rack*)

returns true is the automorphism group of the rack acts transitively on *rack*.

30 ▶ `AreHomologous(` *rack*, *q1* , *q2*)
▶ `TorsionGenerators(` *rack*, *n*)
▶ `SecondCohomologyTorsionGenerators(` *rack*)
▶ `LocalExponent`
▶ `LocalExponents`
▶ `Degree`
▶ `RackAction`
▶ `InverseRackAction`
▶ `Hom`

31 ▶ `Permutations(` *rack*)

returns the list of permutations generating the *rack*.

```
gap> r := TetrahedronRack();;
gap> Permutations(r);
[ (2,3,4), (1,4,3), (1,2,4), (1,3,2) ]
```

32 ▶ `HurwitzOrbit(` *rack*, *vector*)

computes the Hurwitz orbit of the *vector*

```
gap> r := DihedralRack(3);
gap> HurwitzOrbit(r, [1,1,1]);
[ [ 1, 1, 1 ] ]
gap> HurwitzOrbit(r, [1,2,3]);
[ [ 1, 2, 3 ], [ 3, 1, 3 ], [ 1, 1, 2 ], [ 2, 3, 3 ], [ 3, 2, 1 ], [ 1, 3, 1 ], [ 3, 3, 2 ], [ 2,
1 ] ]
```

33 ▶ `HurwitzOrbits(` *rack*, *n*)

returns the list of all n-Hurwitz orbits associated to the rack *rack*.

```
gap> r := DihedralRack(3);;
gap> HurwitzOrbits(r, 3);
[ [ [ 1, 1, 1 ] ], [ [ 1, 1, 2 ], [ 1, 3, 1 ], [ 2, 1, 1 ], [ 1, 2, 3 ], [ 3, 2, 1 ], 3, 1, 3 ]
[ 3, 3, 2 ],
      [ 2, 3, 3 ] ], [ [ 1, 1, 3 ], [ 1, 2, 1 ], [ 3, 1, 1 ], [ 1, 3, 2 ], [ 2, 3, 1 ], 2, 1, 2
[ 2, 2, 3 ],
      [ 3, 2, 2 ] ], [ [ 1, 2, 2 ], [ 3, 1, 2 ], [ 2, 3, 2 ], [ 3, 3, 1 ], [ 2, 1, 3 ], 3, 2, 3
[ 2, 2, 1 ],
      [ 1, 3, 3 ] ], [ [ 2, 2, 2 ] ], [ [ 3, 3, 3 ] ] ]
```

34 ▶ `HurwitzOrbitsRepresentatives(` *rack*, *n* `)`

returns the list of representatives of all n-Hurwitz orbits of the rack *rack*.

```
gap> r := DihedralRack(3);;
gap> HurwitzOrbitsRepresentatives(r, 3);
[ [ 1, 1, 1 ], [ 1, 1, 2 ], [ 1, 1, 3 ], [ 1, 2, 2 ], [ 2, 2, 2 ], [ 3, 3, 3 ] ]
```

35 ▶ `HurwitzOrbitsRepresentativesWS(` *rack*, *n* `)`

returns the list of representatives of all n-Hurwitz orbits of the rack *rack*. The sizes of each orbit are included.

```
gap> r := DihedralRack(3);;
gap> SizesHurwitzOrbits(r, 3);
[ 1, 8 ]
gap> HurwitzOrbitsRepresentativesWS(r, 3);
[ [ [ 1, 1, 1 ], 1 ], [ [ 1, 1, 2 ], 8 ], [ [ 1, 1, 3 ], 8 ], [ [ 1, 2, 2 ], 8 ], [ [ 2 2, 2 ],
], [ [ 3, 3, 3 ], 1 ] ]
```

36 ▶ `NrHurwitzOrbits(` *rack*, *n*, *size* `)`

returns the number of n-Hurwitz orbits of a given *size*

```
gap> r := DihedralRack(3);;
gap> NrHurwitzOrbits(r, 3, 8);
3
```

37 ▶ `SizesHurwitzOrbits(` *rack*, *n* `)`

returns the sizes all n-Hurwitz orbits of *rack*.

```
gap> r := TetrahedronRack();;
gap> SizesHurwitzOrbits(r, 3);
[ 1, 8, 12 ]
```

38 ▶ `SmallIndecomposableQuandle(` *size*, *number* `)`

returns the indecomposable quandle.

Index

This index covers only this manual. A page number in *italics* refers to a whole section which is devoted to the indexed subject. Keywords are sorted with case and spaces ignored, e.g., "PermutationCharacter" comes before "permutation group".