Circle

Adjoint groups of finite rings

Version 1.5.4

8 January 2016

Alexander Konovalov
Panagiotis Soules

Alexander Konovalov Email: alexk@mcs.st-andrews.ac.uk
Homepage: http://www.cs.st-andrews.ac.uk/~alexk/
Address: School of Computer Science
University of St Andrews
Jack Cole Building, North Haugh,
St Andrews, Fife, KY16 9SX, Scotland

Panagiotis Soules Email: psoules@math.uoa.gr
Address: Department of Mathematics
National and Capodistrian University of Athens
Panepistimioupolis, GR-15784, Athens, Greece

Abstract

The GAP4 package Circle extends the GAP functionality for computations in adjoint groups of associative rings. It provides functionality to construct circle objects that will respect the circle multiplication $r \cdot s = r + s + rs$, and to compute adjoint semigroups and adjoint groups of finite rings. Also it may serve as an example of extending the GAP system with new multiplicative objects.

Copyright

© 2006-2016 by Alexander Konovalov and Panagiotis Soules

Circle is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. For details, see the FSF's own site http://www.gnu.org/licenses/gpl.html.

If you obtained Circle, we would be grateful for a short notification sent to one of the authors.

If you publish a result which was partially obtained with the usage of Circle, please cite it in the following form:

A. Konovalov, P. Soules. *Circle — Adjoint groups of finite rings, Version 1.5.4;* 2016 (http://www.cs.st-andrews.ac.uk/~alexk/circle/).

Acknowledgements

We acknowledge very much Alexander Hulpke and James Mitchell for their helpful comments and advices, and the referee for testing the package and useful suggestions.

Contents

1 Introduction **4**
- 1.1 General aims 4
- 1.2 Installation and system requirements 5

2 Implementing circle objects **6**
- 2.1 First attempts 6
- 2.2 Defining circle objects 7
- 2.3 Installing operations for circle objects 10

3 Circle functions **13**
- 3.1 Circle objects 13
- 3.2 Operations with circle objects 15
- 3.3 Construction of the adjoint semigroup and adjoint group 17
- 3.4 Service functions 21

4 A sample computation with Circle **22**

References **24**

Index **25**

Chapter 1

Introduction

1.1 General aims

Let R be an associative ring, not necessarily with one. The set of all elements of R forms a monoid with the neutral element 0 from R under the operation $r \cdot s = r + s + rs$ defined for all r and s of R. This operation is called the *circle multiplication*, and it is also known as the *star multiplication*. The monoid of elements of R under the circle multiplication is called the adjoint semigroup of R and is denoted by R^{ad}. The group of all invertible elements of this monoid is called the adjoint group of R and is denoted by R^*.

These notions naturally lead to a number of questions about the connection between a ring and its adjoint group, for example, how the ring properties will determine properties of the adjoint group; which groups can appear as adjoint groups of rings; which rings can have adjoint groups with prescribed properties, etc.

For example, V. O. Gorlov in [Gor95] gives a full list of finite nilpotent algebras R, such that $R^2 \neq 0$ and the adjoint group of R is metacyclic (but not cyclic).

S. V. Popovich and Ya. P. Sysak in [PS97] characterize all quasiregular algebras such that all subgroups of their adjoint group are their subalgebras. In particular, they show that all algebras of such type are nilpotent with nilpotency index at most three.

Various connections between properties of a ring and its adjoint group were considered by O. D. Artemovych and Yu. B. Ishchuk in [AI97].

B. Amberg and L. S. Kazarin in [AK00] give the description of all nonisomorphic finite p-groups that can occur as the adjoint group of some nilpotent p-algebra of the dimension at most 5.

In [AS01] B. Amberg and Ya. P. Sysak give a survey of results on adjoint groups of radical rings, including such topics as subgroups of the adjoint group; nilpotent groups which are isomorphic to the adjoint group of some radical ring; adjoint groups of finite nilpotent \$p\$-algebras. The authors continued their investigations in further papers [AS02] and [AS04].

In [KS04] L. S. Kazarin and P. Soules study associative nilpotent algebras over a field of positive characteristic whose adjoint group has a small number of generators.

The main objective of the proposed GAP4 package Circle is to extend the GAP functionality for computations in adjoint groups of associative rings to make it possible to use the GAP system for the investigation of the above described questions.

Circle provides functionality to construct circle objects that will respect the circle multiplication $r \cdot s = r + s + rs$, create multiplicative structures, generated by such objects, and compute adjoint semigroups and adjoint groups of finite rings.

Also we hope that the package will be useful as an example of extending the GAP system with new multiplicative objects. Relevant details are explained in the next chapter of the manual.

1.2 Installation and system requirements

Circle does not use external binaries and, therefore, works without restrictions on the type of the operating system. This version of the package is designed for GAP4.5 and no compatibility with previous releases of GAP4 is guaranteed.

To use the Circle online help it is necessary to install the GAP4 package GAPDoc by Frank Lübeck and Max Neunhöffer, which is available from the GAP site or from http://www.math.rwth-aachen.de/~Frank.Luebeck/GAPDoc/.

Circle is distributed in standard formats (`tar.gz`, `tar.bz2`, `zip` and `-win.zip`) and can be obtained from http://www.cs.st-andrews.ac.uk/~alexk/circle/ or from the GAP homepage. To install the package, unpack its archive in the `pkg` subdirectory of your GAP installation.

Chapter 2

Implementing circle objects

In this chapter we explain how the GAP system may be extended with new objects using the circle multiplication as an example. We follow the guidelines given in the GAP Reference Manual (see (**Reference: Creating New Objects**) and subsequent chapters), to which we refer for more details.

2.1 First attempts

Of course, having two ring elements, you can straightforwardly compute their circle product defined as $r \cdot s = r + s + rs$. You can do this in a command line, and it is a trivial task to write a simplest function of two arguments that will do this:

Example

```
gap> CircleMultiplication := function(a,b)
>      return a+b+a*b;
>    end;
function( a, b ) ... end
gap> CircleMultiplication(2,3);
11
gap> CircleMultiplication( ZmodnZObj(2,8), ZmodnZObj(5,8) );
ZmodnZObj( 1, 8 )
```

However, there is no check whether both arguments belong to the same ring and whether they are ring elements at all, so it is easy to obtain some meaningless results:

Example

```
gap> CircleMultiplication( 3, ZmodnZObj(3,8) );
ZmodnZObj( 7, 8 )
gap> CircleMultiplication( [1], [2,3] );
[ 5, 5 ]
```

You can include some tests for arguments, and maybe the best way of doing this would be declaring a new operation for two ring elements, and installing the previous function as a method for this operation. This will check automatically if the arguments are ring elements from the common ring:

Example

```
gap> DeclareOperation( "BetterCircleMultiplication",
>      [IsRingElement,IsRingElement] );
gap> InstallMethod( BetterCircleMultiplication,
>      IsIdenticalObj,
>      [IsRingElement,IsRingElement],
>      CircleMultiplication );
gap> BetterCircleMultiplication(2,3);
11
gap> BetterCircleMultiplication( ZmodnZObj(2,8), ZmodnZObj(5,8) );
ZmodnZObj( 1, 8 )
```

Nevertheless, the functionality gained from such operation would be rather limited. You will not be able to compute circle product via the infix operator *, and, moreover, you will not be able to create higher level objects such as semigroups and groups with respect to the circle multiplication.

In order to "integrate" the circle multiplication into the GAP library properly, instead of defining *new* operations for existing objects, we should define *new* objects for which the infix operator * will perform the circle multiplication. This approach is explained in the next two sections.

2.2 Defining circle objects

Thus, we are going to implement *circle objects*, for which we can envisage the following functionality:

Example

```
gap> CircleObject( 2 ) * CircleObject( 3 );
CircleObject( 11 )
```

First we need to distinguish these new objects from other GAP objects. This is done via the *type* of the objects, that is mainly determined by their *category*, *representation* and *family*.

We start with declaring the category `IsCircleObject` as a subcategory of `IsAssociativeElement>` and `IsMultiplicativeElementWithInverse`. Thus, each circle object will "know" that it is `IsAssociativeElement` and `IsMultiplicativeElementWithInverse`, and this will make it possible to apply to circle objects such operations as `One` and `Inverse` (the latter is allowed to return `fail` for a given circle object), and construct semigroups generated by circle objects.

Example

```
gap> DeclareCategory( "IsMyCircleObject",
> IsAssociativeElement and IsMultiplicativeElementWithInverse );
```

Further we would like to create semigroups and groups generated by circle objects. Such structures will be *collections* of circle objects, so they will be in the category `CategoryCollections( IsCircleObject )`. This is why immediately after we declare the underlying category of circle objects, we need also to declare the category of their collections:

Example

```
gap> DeclareCategoryCollections( "IsMyCircleObject" );
```

On the next step we should think about the internal representation of circle objects. A natural way would be to store the underlying ring element in a list-like structure at its first position. We do not foresee any other data that we need to store internally in the circle object. This is quite common situation, so we may define first `IsPositionalObjectOneSlotRep` that is the list-like representation with only one position in the list, and then declare a synonym `IsDefaultCircleObject` that means that we are dealing with a circle object in one-slot representation:

Example

```
gap> DeclareRepresentation( "IsMyPositionalObjectOneSlotRep",
>     IsPositionalObjectRep, [ 1 ] );
gap> DeclareSynonym( "IsMyDefaultCircleObject",
>     IsMyCircleObject and IsMyPositionalObjectOneSlotRep );
```

Until now we are still unable to create circle objects, because we did not specify to which family they will belong. Naturally, having a ring, we want to have all circle objects for elements of this ring in the same family to be able to multiply them, and we expect circle objects for elements of different rings to be placed in different families. Thus, it would be nice to establish one-to-one correspondence between the family of ring elements and a family of circle elements for this ring. We can store the corresponding circle family as an attribute of the ring elements family. To do this first we declare an attribute `CircleFamily` for families:

Example

```
gap> DeclareAttribute( "MyCircleFamily", IsFamily );
```

Now we install the method that stores the corresponding circle family in this attribute:

Example

```
gap> InstallMethod( MyCircleFamily,
>     "for a family",
>     [ IsFamily ],
>     function( Fam )
>     local F;
>   # create the family of circle elements
>   F:= NewFamily( "MyCircleFamily(...)", IsMyCircleObject );
>   if HasCharacteristic( Fam ) then
>     SetCharacteristic( F, Characteristic( Fam ) );
>   fi;
>   # store the type of objects in the output
>   F!.MyCircleType:= NewType( F, IsMyDefaultCircleObject );
>   # Return the circle family
>   return F;
> end );
```

Similarly, we want one-to-one correspondence between circle elements and underlying ring elements. We declare an attribute `CircleObject` for a ring element, and then install the method to create new circle object from the ring element. This method takes the family of the ring element, finds corresponding circle family, extracts from it the type of circle objects and finally creates the new circle object of that type:

Example

```
gap> DeclareAttribute( "MyCircleObject", IsRingElement );
gap> InstallMethod( MyCircleObject,
>     "for a ring element",
>     [ IsRingElement ],
>     obj -> Objectify( MyCircleFamily( FamilyObj( obj ) )!.MyCircleType,
>                       [ Immutable( obj ) ] ) );
```

Only after entering all code above we are able to create some circle object. However, it is displayed just as `<object>`, though we can get the underlying ring element using the "!" operator:

Example

```
gap> a:=MyCircleObject(2);
<object>
gap> a![1];
2
```

We can check that the intended relation between families holds:

Example

```
gap> FamilyObj( MyCircleObject ( 2 ) ) = MyCircleFamily( FamilyObj( 2 ) );
true
```

We can not multiply circle objects yet. But before implementing this, first let us improve the output by installing the method for `PrintObj`:

Example

```
gap> InstallMethod( PrintObj,
>     "for object in ‘IsMyCircleObject’",
>     [ IsMyDefaultCircleObject ],
>     function( obj )
>     Print( "MyCircleObject( ", obj![1], " )" );
>     end );
```

This method will be used by `Print` function, and also by `View`, since we did not install special method for `ViewObj` for circle objects. As a result of this installation, the output became more meaningful:

Example

```
gap> a;
MyCircleObject( 2 )
```

We need to avoid the usage of "!" operator, which, in general, is not recommended to the user (for example, if GAP developers will change the internal representation of some object, all GAP functions that deal with it must be adjusted appropriately, while if the user's code had direct access to that representation via "!", an error may occur). To do this, we wrap getting the first component of a circle object in the following operation:

Example

```
gap> DeclareAttribute("UnderlyingRingElement", IsMyCircleObject );
gap> InstallMethod( UnderlyingRingElement,
>     "for a circle object",
>     [ IsMyCircleObject],
>     obj -> obj![1] );
gap> UnderlyingRingElement(a);
2
```

2.3 Installing operations for circle objects

Now we are finally able to install circle multiplication as a default method for the multiplication of circle objects, and perform the computation that we envisaged in the beginning:

Example

```
gap> InstallMethod( \*,
>     "for two objects in ‘IsMyCircleObject’",
>     IsIdenticalObj,
>     [ IsMyDefaultCircleObject, IsMyDefaultCircleObject ],
>     function( a, b )
>     return MyCircleObject( a![1] + b![1] + a![1]*b![1] );
>     end );
gap> MyCircleObject(2)*MyCircleObject(3);
MyCircleObject( 11 )
```

However, this functionality is not enough to form semigroups or groups generated by circle elements. We need to be able to check whether two circle objects are equal, and we need to define ordering for them (for example, to be able to form sets of circle elements). Since we already have both operations for underlying ring elements, this can be implemented in a straightforward way:

Example

```
gap> InstallMethod( \=,
>     "for two objects in ‘IsMyCircleObject’",
>     IsIdenticalObj,
>     [ IsMyDefaultCircleObject, IsMyDefaultCircleObject ],
>     function( a, b )
>     return a![1] = b![1];
>     end );
gap> InstallMethod( \<,
>     "for two objects in ‘IsMyCircleObject’",
>     IsIdenticalObj,
>     [ IsMyDefaultCircleObject, IsMyDefaultCircleObject ],
```

```
>     function( a, b )
>     return a![1] < b![1];
>     end );
```

Further, zero element of the ring plays a role of the neutral element for the circle multiplication, and we add this knowledge to our code in a form of a method for `OneOp` that returns circle object for the corresponding zero object:

Example

```
gap> InstallMethod( OneOp,
>     "for an object in ‘IsMyCircleObject’",
>     [ IsMyDefaultCircleObject ],
>     a -> MyCircleObject( Zero( a![1] ) ) );
gap> One(a);
MyCircleObject( 0 )
```

Now we are already able to create monoids generated by circle objects:

Example

```
gap> S:=Monoid(a);
<commutative monoid with 1 generator>
gap> One(S);
MyCircleObject( 0 )
gap> S:=Monoid( MyCircleObject( ZmodnZObj( 2,8) ) );
<commutative monoid with 1 generator>
gap> Size(S);
2
gap> AsList(S);
[ MyCircleObject( ZmodnZObj( 0, 8 ) ), MyCircleObject( ZmodnZObj( 2, 8 ) ) ]
```

Finally, to generate groups using circle objects, we need to add a method for the `InverseOp`. In our implementation we will assume that the underlying ring is a subring of the ring with one, thus, if the circle inverse for an element x exists, than it can be computed as $-x(1+x)^{-1}$:

Example

```
gap> InstallMethod( InverseOp,
>     "for an object in ‘IsMyCircleObject’",
>     [ IsMyDefaultCircleObject ],
>     function( a )
>     local x;
>     x := Inverse( One( a![1] ) + a![1] );
>     if x = fail then
>       return fail;
>     else
>       return MyCircleObject( -a![1] * x );
>     fi;
>     end );
gap> MyCircleObject(-2)^-1;
```

```
MyCircleObject( -2 )
gap> MyCircleObject(2)^-1;
MyCircleObject( -2/3 )
```

The last method already makes it possible to create groups generated by circle objects (the warning may be ignored):

Example

```
gap> Group( MyCircleObject(2) );
#I  default 'IsGeneratorsOfMagmaWithInverses' method returns 'true' for
[ MyCircleObject( 2 ) ]
<group with 1 generators>
gap> G:=Group( [MyCircleObject( ZmodnZObj( 2,8 ) )  ]);
#I  default 'IsGeneratorsOfMagmaWithInverses' method returns 'true' for
[ MyCircleObject( ZmodnZObj( 2, 8 ) ) ]
<group with 1 generators>
gap> Size(G);
2
gap> AsList(G);
[ MyCircleObject( ZmodnZObj( 0, 8 ) ), MyCircleObject( ZmodnZObj( 2, 8 ) ) ]
```

The GAP code used in this Chapter, is contained in the files `circle/lib/circle.gd` and `circle/lib/circle.gi` (without `My` in identifiers). For more examples of implementing new GAP objects and further details see (**Reference: Creating New Objects**) and subsequent chapters in the GAP Reference Manual.

Chapter 3

Circle functions

To use the Circle package first you need to load it as follows:

Example

```
gap> LoadPackage("circle");
------------------------------------------------------------------------------
Loading  Circle 1.4.0 (Adjoint groups of finite rings)
by Alexander Konovalov (http://www.cs.st-andrews.ac.uk/~alexk/) and
   Panagiotis Soules (psoules@math.uoa.gr).
------------------------------------------------------------------------------
true
gap>
```

Note that if you entered examples from the previous chapter, you need to restart GAP before loading the Circle package.

3.1 Circle objects

Because for elements of the ring R the ordinary multiplication is already denoted by $*$, for the implementation of the circle multiplication in the adjoint semigroup we need to wrap up ring elements as CircleObjects, for which $*$ is defined to be the circle multiplication.

3.1.1 CircleObject

▷ `CircleObject(x)` (attribute)

Let x be a ring element. Then `CircleObject(x)` returns the corresponding circle object. If x lies in the family `fam`, then `CircleObject(x)` lies in the family `CircleFamily` (3.1.5), corresponding to the family `fam`.

Example

```
gap> a := CircleObject( 2 );
CircleObject( 2 )
```

3.1.2 UnderlyingRingElement

▷ UnderlyingRingElement(x) (attribute)

Returns the corresponding ring element for the circle object x.

Example

```
gap> a := CircleObject( 2 );
CircleObject( 2 )
gap> UnderlyingRingElement( a );
2
```

3.1.3 IsCircleObject

▷ IsCircleObject(x) (Category)
▷ IsCircleObjectCollection(x) (Category)

An object x lies in the category IsCircleObject if and only if it lies in a family constructed by CircleFamily (3.1.5). Since circle objects can be multiplied via * with elements in their family, and we need operations One and Inverse to deal with groups they generate, circle objects are implemented in the category IsMultiplicativeElementWithInverse. A collection of circle objects (e.g. adjoint semigroup or adjoint group) will lie in the category IsCircleObjectCollection.

Example

```
gap> IsCircleObject( 2 ); IsCircleObject( CircleObject( 2 ) );
false
true
gap> IsMultiplicativeElementWithInverse( CircleObject( 2 ) );
true
gap> IsCircleObjectCollection( [ CircleObject(0), CircleObject(2) ] );
true
```

3.1.4 IsPositionalObjectOneSlotRep

▷ IsPositionalObjectOneSlotRep(x) (Representation)
▷ IsDefaultCircleObject(x) (Representation)

To store the corresponding circle object, we need only to store the underlying ring element. Since this is quite common situation, we defined the representation IsPositionalObjectOneSlotRep for a more general case. Then we defined IsDefaultCircleObject as a synonym of IsPositionalObjectOneSlotRep for objects in IsCircleObject (3.1.3).

Example

```
gap> IsPositionalObjectOneSlotRep( CircleObject( 2 ) );
true
gap> IsDefaultCircleObject( CircleObject( 2 ) );
true
```

3.1.5 CircleFamily

▷ `CircleFamily(`*fam*`)` (attribute)

`CircleFamily(fam)` is a family, elements of which are in one-to-one correspondence with elements of the family *fam*, but with the circle multiplication as an infix multiplication. That is, for x, y in *fam*, the product of their images in the `CircleFamily(fam)` will be the image of $x+y+xy$. The relation between these families is demonstrated by the following equality:

Example

```
gap> FamilyObj( CircleObject ( 2 ) ) = CircleFamily( FamilyObj( 2 ) );
true
```

3.2 Operations with circle objects

3.2.1 One

▷ `One(x)` (operation)

This operation returns the multiplicative neutral element for the circle object `x`. The result is the circle object corresponding to the additive neutral element of the appropriate ring.

Example

```
gap> One( CircleObject( 5 ) );
CircleObject( 0 )
gap> One( CircleObject( 5 ) ) = CircleObject( Zero( 5 ) );
true
gap> One( CircleObject( [ [ 1, 1 ],[ 0, 1 ] ] ) );
CircleObject( [ [ 0, 0 ], [ 0, 0 ] ] )
```

3.2.2 InverseOp

▷ `InverseOp(x)` (operation)

For a circle object `x`, returns the multiplicative inverse of `x` with respect to the circle multiplication; if such one does not exist then `fail` is returned.

In our implementation we assume that the underlying ring is a subring of the ring with one, thus, if the circle inverse for an element x exists, than it can be computed as $-x(1+x)^{-1}$.

Example

```
gap> CircleObject( -2 )^-1;
CircleObject( -2 )
gap> CircleObject( 2 )^-1;
CircleObject( -2/3 )
gap> CircleObject( -2 )*CircleObject( -2 )^-1;
CircleObject( 0 )
```

Example

```
gap> m := CircleObject( [ [ 1, 1 ], [ 0, 1 ] ] );
CircleObject( [ [ 1, 1 ], [ 0, 1 ] ] )
gap> m^-1;
CircleObject( [ [ -1/2, -1/4 ], [ 0, -1/2 ] ] )
gap> m * m^-1;
CircleObject( [ [ 0, 0 ], [ 0, 0 ] ] )
gap> CircleObject( [ [ 0, 1 ], [ 1, 0 ] ] )^-1;
fail
```

3.2.3 IsUnit

▷ IsUnit(*[R,]x*) (operation)

Let *x* be a circle object corresponding to an element of the ring *R*. Then the operation IsUnit returns true, if *x* is invertible in *R* with respect to the circle multiplication, and false otherwise.

Example

```
gap> IsUnit( Integers, CircleObject( -2 ) );
true
gap> IsUnit( Integers, CircleObject( 2 ) );
false
gap> IsUnit( Rationals, CircleObject( 2 ) );
true
gap> IsUnit( ZmodnZ(8), CircleObject( ZmodnZObj(2,8) ) );
true
gap> m := CircleObject( [ [ 1, 1 ],[ 0, 1 ] ] );;
gap> IsUnit( FullMatrixAlgebra( Rationals, 2 ), m );
true
```

If the first argument is omitted, the result will be returned with respect to the default ring of the circle object *x*.

Example

```
gap> IsUnit( CircleObject( -2 ) );
true
gap> IsUnit( CircleObject( 2 ) );
false
gap> IsUnit( CircleObject( ZmodnZObj(2,8) ) );
true
gap> IsUnit( CircleObject( [ [ 1, 1 ],[ 0, 1 ] ] ) );
false
```

3.2.4 IsCircleUnit

▷ IsCircleUnit(*[R,]x*) (operation)

Let x be an element of the ring R. Then `IsCircleUnit( R, x )` determines whether x is invertible in R with respect to the circle multilpication. This is equivalent to the condition that $1+x$ is a unit in R with respect to the ordinary multiplication.

Example

```
gap> IsCircleUnit( Integers, -2 );
true
gap> IsCircleUnit( Integers, 2 );
false
gap> IsCircleUnit( Rationals, 2 );
true
gap> IsCircleUnit( ZmodnZ(8), ZmodnZObj(2,8) );
true
gap> m := [ [ 1, 1 ],[ 0, 1 ] ];
[ [ 1, 1 ], [ 0, 1 ] ]
gap> IsCircleUnit( FullMatrixAlgebra(Rationals,2), m );
true
```

If the first argument is omitted, the result will be returned with respect to the default ring of x.

Example

```
gap> IsCircleUnit( -2 );
true
gap> IsCircleUnit( 2 );
false
gap> IsCircleUnit( ZmodnZObj(2,8) );
true
gap> IsCircleUnit( [ [ 1, 1 ],[ 0, 1 ] ] );
false
```

3.3 Construction of the adjoint semigroup and adjoint group

3.3.1 AdjointSemigroup

▷ `AdjointSemigroup(`R`)` (attribute)

If R is a finite ring then `AdjointSemigroup(`R`)` will return the monoid which is formed by all elements of R with respect to the circle multiplication.

The implementation is rather straightforward and was added to provide a link to the GAP functionality for semigroups. It assumes that the enumaration of all elements of the ring R is feasible.

Example

```
gap> R:=Ring( [ ZmodnZObj(2,8) ] );
<ring with 1 generators>
gap> S:=AdjointSemigroup(R);
<monoid with 4 generators>
```

3.3.2 AdjointGroup

▷ `AdjointGroup(R)` (attribute)

If R is a finite radical algebra then `AdjointGroup(R)` will return the adjoint group of R, given as a group generated by a set of circle objects.

To compute the adjoint group of a finite radical algebra, Circle uses the fact that all elements of a radical algebra form a group with respect to the circle multiplication. Thus, the adjoint group of R coincides with R elementwise, and we can randomly select an appropriate set of generators for the adjoint group.

The warning is displayed by `IsGeneratorsOfMagmaWithInverses` method defined in `gap4r4/lib/grp.gi` and may be ignored.

WARNINGS:

1. The set of generators of the returned group is not required to be a generating set of minimal possible order.

2. `AdjointGroup` is stored as an attribute of R, so for the same copy of R calling it again you will get the same result. But if you will create another copy of R in the future, the output may differ because of the random selection of generators. If you want to have the same generating set, next time you should construct a group immediately specifying circle objects that generate it.

3. In most cases, to investigate some properties of the adjoint group, it is necessary first to convert it to an isomorphic permutation group or to a PcGroup.

For example, we can create the following commutative 2-dimensional radical algebra of order 4 over the field of two elements, and show that its adjoint group is a cyclic group of order 4:

Example

```
gap> x:=[ [ 0, 1, 0 ],
>         [ 0, 0, 1 ],
>         [ 0, 0, 0 ] ];;
gap> R := Algebra( GF(2), [ One(GF(2))*x ] );
<algebra over GF(2), with 1 generators>
gap> RadicalOfAlgebra( R ) = R;
true
gap> Dimension(R);
2
gap> G := AdjointGroup( R );;
gap> Size( R ) = Size( G );
true
gap> StructureDescription( G );
"C4"
```

In the following example we construct a non-commutative 3-dimensional radical algebra of order 8 over the field of two elements, and demonstrate that its adjoint group is the dihedral group of order 8:

Example

```
gap> x:=[ [ 0, 1, 0 ],
>         [ 0, 0, 0 ],
>         [ 0, 0, 0 ] ];;
gap> y:=[ [ 0, 0, 0 ],
>         [ 0, 0, 1 ],
>         [ 0, 0, 0 ] ];;
gap> R := Algebra( GF(2), One(GF(2))*[x,y] );
<algebra over GF(2), with 2 generators>
gap> RadicalOfAlgebra(R) = R;
true
gap> Dimension(R);
3
gap> G := AdjointGroup( R );
<group of size 8 with 2 generators>
gap> StructureDescription( G );
"D8"
```

If the ring R is not a radical algebra, then Circle will use another approach. We will enumerate all elements of the ring R and select those that are units with respect to the circle multiplication. Then we will use a random approach similar to the case of the radical algebra, to find some generating set of the adjoint group. Again, all warnings 1-3 above refer also to this case.

Of course, enumeration of all elements of R should be feasible for this computation. In the following example we demonstrate how it works for rings, generated by residue classes:

Example

```
gap> R := Ring( [ ZmodnZObj(2,8) ] );
<ring with 1 generators>
gap> G := AdjointGroup( R );
<group of size 4 with 2 generators>
gap> StructureDescription( G );
"C2 x C2"
gap> R := Ring( [ ZmodnZObj(2,256) ] );
<ring with 1 generators>
gap> G := AdjointGroup( R );;
gap> StructureDescription( G );
"C64 x C2"
```

Due to the `AdjointSemigroup` (3.3.1), there is also another way to compute the adjoint group of a ring R by means of the computation of its adjoint semigroup $S(R)$ and taking the Green's H-class of the multiplicative neutral element of $S(R)$. Let us repeat the last example in this way:

Example

```
gap> R := Ring( [ ZmodnZObj(2,256) ] );
<ring with 1 generators>
gap> S := AdjointSemigroup( R );
<monoid with 128 generators>
gap> H := GreensHClassOfElement(S,One(S));
```

```
<Green's H-class: <object>>
gap> G:=AsGroup(H);
<group of size 128 with 2 generators>
gap> StructureDescription(G);
"C64 x C2"
```

However, the conversion of the Green's H-class to the group may take some time which may vary dependently on the particular ring in question, and will also display a lot of warnings about the default `IsGeneratorsOfMagmaWithInverses` method, so we did not implemented this as as standard method. In the following example the method based on Green's H-class is much slower than an application of earlier described random approach (20s vs 10ms):

Example

```
gap> R := Ring( [ ZmodnZObj(2,256) ] );
<ring with 1 generators>
gap> AdjointGroup(R);;
gap> R := Ring( [ ZmodnZObj(2,256) ] );
<ring with 1 generators>
gap> S:=AdjointSemigroup(R);
<monoid with 128 generators>
gap> AsGroup(GreensHClassOfElement(S,One(S)));
<group of size 128 with 2 generators>
```

Finally, note that if `R` has a unity 1, then the set $1+R^{ad}$, where R^{ad} is the adjoint semigroup of `R`, coincides with the multiplicative semigroup R^{mult} of R, and the map $r \mapsto (1+r)$ for r in R is an isomorphism from R^{ad} onto R^{mult}.

Similarly, the set $1+R^*$, where R^* is the adjoint group of `R`, coincides with the unit group of R, which we denote $U(R)$, and the map $r \mapsto (1+r)$ for r in R is an isomorphism from R^* onto $U(R)$.

We demonstrate this isomorphism using the following example.

Example

```
gap> LoadPackage( "laguna", false );
true
gap> FG := GroupRing( GF(2), DihedralGroup(8) );
<algebra-with-one over GF(2), with 3 generators>
gap> R := AugmentationIdeal( FG );;
gap> G := AdjointGroup( R );;
gap> IdGroup( G );
[ 128, 170 ]
gap> IdGroup( Units( FG ) );
#I  LAGUNA package: Computing the unit group ...
[ 128, 170 ]
```

Thus, dependently on the ring `R` in question, it might be possible that you can compute much faster its unit group using `Units(R)` than its adjoint group using `AdjointGroup(R)`. This is why in an attempt of computation of the adjoint group of the ring with one a warning message will be displayed:

Example

```
gap> Size( AdjointGroup( GroupRing( GF(2), DihedralGroup(8) ) ) );

WARNING: usage of AdjointGroup for associative ring <R> with one!!!
In this case the adjoint group is isomorphic to the unit group
Units(<R>), which possibly may be computed faster!!!

128
gap> Size( AdjointGroup( Integers mod 11 ) );

WARNING: usage of AdjointGroup for associative ring <R> with one!!!
In this case the adjoint group is isomorphic to the unit group
Units(<R>), which possibly may be computed faster!!!

10
```

If *R* is infinite, an error message will appear, telling that Circle does not provide methods to deal with infinite rings.

3.4 Service functions

3.4.1 InfoCircle

▷ `InfoCircle` (info class)

`InfoCircle` is a special Info class for Circle algorithms. It has 2 levels: 0 (default) and 1. To change info level to `k`, use command `SetInfoLevel(InfoCircle, k)`.

Example

```
gap> SetInfoLevel( InfoCircle, 1 );
gap> SetInfoLevel(InfoCircle,1);
gap> R := Ring( [ ZmodnZObj(2,8) ]);
<ring with 1 generators>
gap> G := AdjointGroup( R );
#I  Circle : <R> is not a radical algebra, computing circle units ...
#I  Circle : searching generators for adjoint group ...
<group of size 4 with 2 generators>
gap> SetInfoLevel( InfoCircle, 0 );
```

Chapter 4

A sample computation with Circle

Here we give an example to give the reader an idea what Circle is able to compute.

It was proved in [KS04] that if R is a finite nilpotent two-generated algebra over a field of characteristic $p > 3$ whose adjoint group has at most three generators, then the dimension of R is not greater than 9. Also, an example of the 6-dimensional such algebra with the 3-generated adjoint group was given there. We will construct the algebra from this example and investigate it using Circle. First we create two matrices that determine its generators:

Example

```
gap> x:=[ [ 0, 1, 0, 0, 0, 0, 0 ],
>         [ 0, 0, 0, 1, 0, 0, 0 ],
>         [ 0, 0, 0, 0, 1, 0, 0 ],
>         [ 0, 0, 0, 0, 0, 0, 1 ],
>         [ 0, 0, 0, 0, 0, 1, 0 ],
>         [ 0, 0, 0, 0, 0, 0, 0 ],
>         [ 0, 0, 0, 0, 0, 0, 0 ] ];;
gap> y:=[ [ 0, 0, 1, 0, 0, 0, 0 ],
>         [ 0, 0, 0, 0,-1, 0, 0 ],
>         [ 0, 0, 0, 1, 0, 1, 0 ],
>         [ 0, 0, 0, 0, 0, 1, 0 ],
>         [ 0, 0, 0, 0, 0, 0,-1 ],
>         [ 0, 0, 0, 0, 0, 0, 0 ],
>         [ 0, 0, 0, 0, 0, 0, 0 ] ];;
```

Now we construct this algebra in characteristic five and check its basic properties:

Example

```
gap> R := Algebra( GF(5), One(GF(5))*[x,y] );
<algebra over GF(5), with 2 generators>
gap> Dimension( R );
6
gap> Size( R );
15625
gap> RadicalOfAlgebra( R ) = R;
true
```

Then we compute the adjoint group of R:

Example

```
gap> G := AdjointGroup( R );;
gap> Size(G);
15625
```

Now we can find the generating set of minimal possible order for the group G, and check that G it is 3-generated. To do this, first we need to convert it to the isomorphic PcGroup:

Example

```
gap> f := IsomorphismPcGroup( G );;
gap> H := Image( f );
Group([ f1, f2, f3, f4, f5, f6 ])
gap> gens := MinimalGeneratingSet( H );;
gap> Length( gens );
3
```

One can also use UnderlyingRingElement(PreImage(f,x)) to find the preimage of x in G.

It appears that the adjoint group of the algebra from example will be 3-generated in characteristic 3 as well:

Example

```
gap> R := Algebra( GF(3), One(GF(3))*[x,y] );
<algebra over GF(3), with 2 generators>
gap> G := AdjointGroup( R );
<group of size 729 with 3 generators>
gap> H := Image( IsomorphismPcGroup( G ) );
Group([ f1, f2, f3, f4, f5, f6 ])
gap> Length( MinimalGeneratingSet( H ) );
3
```

But this is not the case in characteristic 2, where the adjoint group is 4-generated:

Example

```
gap> R := Algebra( GF(2), One(GF(2))*[x,y] );
<algebra over GF(2), with 2 generators>
gap> G := AdjointGroup( R );;
gap> Size(G);
64
gap> H := Image( IsomorphismPcGroup( G ) );
Group([ f1, f2, f3, f4, f5, f6 ])
gap> Length( MinimalGeneratingSet( H ) );
4
```

References

[AI97] O. D. Artemovych and Yu. B. Ishchuk. On semiperfect rings determined by adjoint groups. *Mat. Stud.*, 8(2):162–170, 237, 1997. 4

[AK00] B. Amberg and L. S. Kazarin. On the adjoint group of a finite nilpotent p-algebra. *J. Math. Sci. (New York)*, 102(3):3979–3997, 2000. 4

[AS01] B. Amberg and Ya. P. Sysak. Radical rings and their adjoint groups. In *Topics in infinite groups*, volume 8 of *Quad. Mat.*, pages 21–43. Dept. Math., Seconda Univ. Napoli, Caserta, 2001. 4

[AS02] B. Amberg and Ya. P. Sysak. Radical rings with soluble adjoint groups. *J. Algebra*, 247(2):692–702, 2002. 4

[AS04] B. Amberg and Ya. P. Sysak. Associative rings with metabelian adjoint group. *J. Algebra*, 277(2):456–473, 2004. 4

[Gor95] V. O. Gorlov. Finite nilpotent algebras with a metacyclic quasiregular group. *Ukraïn. Mat. Zh.*, 47(10):1426–1431, 1995. 4

[KS04] L. S. Kazarin and P. Soules. Finite nilpotent p-algebras whose adjoint group has three generators. *JP J. Algebra Number Theory Appl.*, 4(1):113–127, 2004. 4, 22

[PS97] S. V. Popovich and Ya. P. Sysak. Radical algebras whose subgroups of adjoint groups are subalgebras. *Ukraïn. Mat. Zh.*, 49(12):1646–1652, 1997. 4

Index

AdjointGroup, 18
AdjointSemigroup, 17

Circle package, 2
CircleFamily, 15
CircleObject, 13

InfoCircle, 21
InverseOp, 15
IsCircleObject, 14
IsCircleObjectCollection, 14
IsCircleUnit, 16
IsDefaultCircleObject, 14
IsPositionalObjectOneSlotRep, 14
IsUnit, 16

One, 15

UnderlyingRingElement, 14