QUATERNION ALGEBRAS

David R. Kohel

QUATERNION ALGEBRAS

QUATERNION ALGEBRAS

David R. Kohel

§1 Introduction

A quaternion algebra A over a field K is a central simple algebra of dimension four over K. The matrix algebra $M_2(K)$, the *split* quaternion algebra, is a trivial case. Every quaternion algebra over K not isomorphic to $M_2(K)$ is a division algebra. Over some quadratic field extension L/K, the extension of scalars to L is the matrix algebra $M_2(L)$. The latter fact makes the inclusion of split quaternion algebras important for the purpose of having a well-defined class of algebras which are preserved under base extension. In particular, we note that a quaternion algebra can be formed over a finite field K, where every quaternion algebra is isomorphic to a matrix algebra. A quaternion algebra in Magma is a specialized type of dimension 4 associative algebra over a field, which has the type `AlgQuat`.

A principal interest in the study of quaternion algebras is the orders and ideals over an integral domain R with quotient field K. Specialized functions are available for quaternion algebras over the rational field $\mathbf{Q}$. Facilities for enumeration of all isomorphism classes of left or right ideals over an order are implemented, and this ties into the machinery for constructing the Brandt module – a free abelian group on these ideal classes, on which Brandt matrices act as the adjacency matrices of the graph of ideals under a certain p-neighbor relation on ideals. It is also possible to construct orders over a rational function field $k(x)$, for a field k. Orders and ideals in Magma have a common type `AlgQuatOrd` and are distinguished only by the presence of a unity. The main reference for material in this chapter is the book of Vignéras [**????**].

Acknowledgement. Central functionality for quaternion algebras, orders, and ideals as Magma associative algebras was provided by a port of these structures to the Magma kernel by Nicole Sutherland.

§2 Creation of Quaternion Algebras

A general constructor for a quaternion algebra over any field K creates a model in terms of two generators x and y and three relations $x^2 = a$, $y^2 = b$, and $xy = -yx$. The printing names i, j, and k are assigned to the generators x, y, and xy by default, unless the user assigns alternatives (see the function `AssignNames` below, or the example which follows).

Special constructors are provided for quaternion algebras over $\mathbf{Q}$, which return an algebra with a more general set of three defining quadratic relations (see below). In general the third generator need not be the product of the first two. This allows the creation of a quaternion algebra A such that the default generators $\{1, i, j, k\}$ form a basis for a maximal order.

`QuaternionAlgebra< K | a, b >`

This function creates the quaternion algebra A over the field K on generators i and j with relations $i^2 = a$, $j^2 = b$, and $ij = -ji$. A third generator is set to $k = ij$.

`AssignNames(~A,S)`

Given a string sequence S of length 3, assigns the strings to the generators of A, i.e. the basis elements not equal to 1. This function is called by the bracket operators `< >` at the time of creation of a quaternion algebra.

Example E1

A general quaternion algebra can be created as follows. Note that the brackets `< >` can be used to give any convenient names to the generators of the ring.

```
> A<x,y,z> := QuaternionAlgebra< RationalField() | -1, -7 >;
> x^2;
-1
> y^2;
-7
> x*y;
z
```

In the case of this constructor the algebra generators are of trace zero and are pairwise anticommuting. This contrasts with the following special constructors for quaternion algebras over the rationals.

`QuaternionAlgebra(N)`

The rational quaternion algebra with square-free discriminant N. The integer N must be a product of an odd number of primes, i.e. the algebra must be definite.

`QuaternionAlgebra(D1, D2, T)`

The rational quaternion algebra $\mathbf{Q}\langle i, j\rangle$, where $\mathbf{Z}[i]$ and $\mathbf{Z}[j]$ are quadratic suborders of discriminant D_1 and D_2, respectively, and $\mathbf{Z}[ij - ji]$ is a quadratic suborder of discriminant $D_3 = D_1D_2 - T^2$. The values D_1D_2 and T must have the same parity and D_1, D_2 and D_3 must each be the discriminant of some quadratic order, i.e. nonsquare integers congruent to 0 or 1 modulo 4.

Example E2

The above constructor is quite powerful for constructing quaternion algebras with given ramification. It can be verified that for any i and j, a commutator element such as $ij - ji$ has trace zero, so in the above constructor, the minimal polynomial of this element is $x^2 + n$, where

$n = (D_1D_2 - T^2)/4$. In the following example we construct such a ring, and demonstrate some of the relations satisfied in this algebra. Note that the minimal polynomial is an element of the commutative polynomial ring over the base field of the algebra.

```
> A<i,j,k> := QuaternionAlgebra(-7,-47,1);
> PQ<x> := PolynomialRing(RationalField());
> MinimalPolynomial(i);
x^2 - x + 2
> MinimalPolynomial(j);
x^2 - x + 12
> MinimalPolynomial(k);
x^2 - x + 24
> i*j;
k
```

From their minimal polynomials, we see that the algebra generators i, j, and k generate commutative subfields of discriminants -7, -47, and -95. The value $82 = (D_1D_2 - T^2)/4$, however, is a more important invariant of the ring. We give a preview of the later material by demonstrating the functionality for computing the determinant and ramified primes of an algebra over $\mathbf{Q}$.

```
> MinimalPolynomial(i*j-j*i);
x^2 + 82
> Discriminant(A);
41
> RamifiedPrimes(A);
[ 41 ]
```

The ramified primes of a quaternion algebra are those primes p for which A is locally (i.e. over $\mathbf{Q}_p$) isomorphic to a division algebra. The discriminant of a quaternion algebra is equal to the product of the ramified primes, and is well-defined up to squares. An elementary property of the ramifying primes is that they must be inert or ramified in every subfield. Moreover it must divide the norm of any commutator element $xy - yx$. We check these properties with the functions `Norm` and `KroneckerSymbol`.

```
> x := i;
> y := j;
> Norm(x*y-y*x);
82
> Factorization(82);
[ <2, 1>, <41, 1> ]
> x := i-k;
> y := i+j+k;
> Norm(x*y-y*x);
1640
> Factorization(1640);
[ <2, 3>, <5, 1>, <41, 1> ]
> KroneckerSymbol(-7,2);
1
> KroneckerSymbol(-47,2);
1
```

```
> KroneckerSymbol(-95,2);
1
> KroneckerSymbol(-7,41);
-1
> KroneckerSymbol(-47,41);
-1
> KroneckerSymbol(-95,41);
-1
```

The fact that the latter Kronecker symbols are -1, indicating that 41 is inert in the quadratic fields of discriminants -7, -47, and -95, proves that 41 is a ramified prime, and 2 is not.

The constructor `QuaternionAlgebra(D1,D2,T)` is the function underlying the constructor `QuaternionAlgebra(N)`, which returns a quaternion algebra over $\mathbf{Q}$ of discriminant N.

§3 Creation of Quaternion Orders

If R is a ring with field of fractions K, then an R-order S in an algebra A is an R-submodule whose rank over R is the same as the dimension of A/K. At present the rings R for which we can form quaternion orders are $\mathbf{Z}$ and $k[x]$ where k is a field. Special constructors are available for orders over $\mathbf{Z}$ in a quaternion algebra over $\mathbf{Q}$.

`MaximalOrder(A)`

A maximal order in A.

`QuaternionOrder(A,M)`

An order of index M in a maximal order of the rational quaternion algebra A. The second argument M can be at most of valuation 1 at any ramified prime of A.

`QuaternionOrder(N)`

`QuaternionOrder(N,M)`

An order of index M in a maximal order of the rational quaternion algebra A of discriminant N. The discriminant N must be a product of an odd number product of distinct primes, and the argument M can be at most of valuation 1 at any prime dividing N. When not provided, the integer M defaults to 1, i.e. the return value will be a maximal order.

`QuaternionOrder(D1, D2, T)`

The quaternion order $\mathbf{Z}\langle x, y\rangle$, where $\mathbf{Z}[x]$ and $\mathbf{Z}[y]$ are quadratic subrings of discriminant D_1 and D_2, respectively, and $\mathbf{Z}[xy - yx]$ is a quadratic subring of discriminant $D_1D_2 - T^2$.

Example E3

The above constructors permit the construction of Eichler orders over **Z**, if the discriminant N and the index M are coprime; or more generally of some order whose index in an Eichler order divides the discriminant.

```
> A := QuaternionOrder(103,2);
```

The following general constructors permit the construction of quaternion orders over other rings. At present a polynomial ring over a field is the only other permitted base ring of an order. These constructors can be of use for specifying a particular basis for the order, or for constructing more general orders over **Z** than those given by the standard constructors.

`QuaternionOrder(S)`

The order generated by the sequence S of elements in a quaternion algebra.

`QuaternionOrder(R,S)`

Given a ring R and a sequence S of elements of a quaternion algebra, returns the order with basis S. The sequence length must be 4 and the element 1 must be the first element.

Example E4

The basis constructors of orders permit a more general class of orders. First we construct an order over a polynomial ring.

```
> K<t> := FunctionField(FiniteField(7));
> A<i,j,k> := QuaternionAlgebra< K | t, t^2+t+1 >;
> i^2;
t
> j^2;
(t^2 + t + 1)
```

Next we demonstrate how to construct orders in quaternion algebras generated by a given sequence of elements. When provided with a sequence of elements of a quaternion algebra over **Q**, the sequence is reduced to form a basis. When provided with the ring over which these elements are to be interpretted, the sequence must be a basis with initial element 1, and the order with this basis is constructed.

```
> A<i,j,k> := QuaternionAlgebra< RationalField() | -1, -3 >;
> B := [ 1, 1/2 + 1/2*j, i, 1/2*i + 1/2*k ];
> O := QuaternionOrder(B);
> Basis(O);
[ 1, 1/2*i + 1/2*k, 1/2 - 1/2*j, -1/2*i + 1/2*k ]
> S := QuaternionOrder(Integers(),B);
> Basis(S);
```

```
[ 1, 1/2 + 1/2*j, i, 1/2*i + 1/2*k ]
```

---

§4 Elements of Quaternion Algebras

§4.1 Creation of Elements

`A ! 0`

`Zero(A)`

The zero element of the quaternion algebra A.

`A ! 1`

`One(A)`

The identity element of the quaternion algebra A.

`A . i`

`S . i`

Given a quaternion algebra A or order S and an integer $1 \leq i \leq 3$, returns the i-th generator as an algebra over the base ring. Note that the element 1 is always the first element of a basis, and is never returned as a generating element.

§4.2 Arithmetic of Elements

`x + y`

The sum of x and y.

`x - y`

The difference of x and y.

`x * y`

The product of x and y.

`x / y`

The quotient of x by the unit y in the quaternion algebra; if x and or y are elements of a quaternion order then the result is returned as an element of the algebra is created.

`x eq y`

True if the elements a and b of an algebra A are equal; otherwise false.

`x ne y`

True if and only if the elements x and y are not equal.

`x in A`

True if and only if x is in the algebra A.

`x notin A`

True if and only if x is not in the algebra A.

`Conjugate(x)`

The conjugate $\bar{x}$ of x, defined such that the reduced trace and reduced norm are $\bar{x} + x$ and $\bar{x}x$, respectively.

`ElementToSequence(x)`

`ElementToSequence(x)`

`Eltseq(x)`

`Eltseq(x)`

`Coordinates(x)`

`Coordinates(x)`

Given an element x of a quaternion algebra, order, or ideal, returns the sequence of coordinates of x in terms of the basis of its parent.

`Norm(x)`

The reduced norm $\mathrm{N}(x)$ of x, defined such that the characteristic polynomial for x is $x^2 - \mathrm{Tr}(x)x + \mathrm{N}(x) = 0$, where $\mathrm{Tr}(x)$ is the reduced trace.

`Trace(x)`

The reduced trace $\mathrm{Tr}(x)$ of x, defined such that the characteristic polynomial for x is $x^2 - \mathrm{Tr}(x)x + \mathrm{N}(x) = 0$, where $\mathrm{N}(x)$ is the reduced norm.

`CharacteristicPolynomial(x)`

The characteristic polynomial of degree 2 for x over the base ring of its parent.

`MinimalPolynomial(x)`

The minimal polynomial of degree 1 or 2 for x over the base ring of its parent.

Example E5

We demonstrate the relation between characteristic polynomial, and reduced trace and norm in the following example.

```
> A := QuaternionAlgebra< RationalField() | -17, -271 >;
> x := A![1,-2,3,0];
> Trace(x);
5
> Norm(x);
1640
> x^2 - Trace(x)*x + Norm(x);
0
```

Note that trace and norm of an element x of any algebra can be defined as the trace of norm of the linear operator of left or right multiplication by x. The reduced trace and norm in a quaternion algebra A are instead the corresponding trace or norm in any two dimensional matrix representation of A, generally over an extension of the base field. The former definition for a general algebra can be realised with the following lines.

```
> P<X> := PolynomialRing(RationalField());
> M := Matrix(4,4,&cat[ Eltseq(x*y) : y in Basis(A) ]);
> Trace(M);
10
> Factorization(CharacteristicPolynomial(M));
[
    <X^2 - 5*X + 1640, 2>
]
```

And it is easily verified that the general definition of trace is twice the reduced trace, and the general definition of norm is the square of the reduced norm.

§5 Attributes of Quaternion Algebras

`BaseField(A)`

`BaseRing(A)`

The base field of the quaternion algebra A.

`Basis(A)`

The basis of the algebra A.

`Discriminant(A)`

Given a quaternion algebra A returns the reduced discriminant as an element of the base ring, which is well-defined up to squares. If A is defined over the rationals, then the value is a square-free integer in $\mathbf{Q}$.

`RamifiedPrimes(A)`

Given a quaternion algebra A over $\mathbf{Q}$, returns the sequences of finite ramified primes, i.e. those primes dividing the discriminant. Note that the algebra is definite or indefinite, according to whether the sequence is of odd or even length.

Example E6

The sequence of ramified primes of a quaternion algebra A over $\mathbf{Q}$ determines the isomorphism class of the algebra.

```
> A := QuaternionAlgebra(-436,-503,22);
> RamifiedPrimes(A);
[ 17 ]
```

Provided the discriminant is of a size which can be factored, the ramified primes are determined efficiently using Hilbert symbols.

§6 Predicates on Algebras

`IsDefinite(A)`

Given a quaternion algebra over the rationals, returns `true` if and only if A is a definite quaternion algebra.

`IsIndefinite(A)`

Given a quaternion algebra over the rationals, returns `true` if and only if A is an indefinite quaternion algebra.

§7 Attributes of Orders and Ideals

`QuaternionAlgebra(S)`

The quaternion algebra in which S is an order.

`BaseRing(S)`

The base ring of the quaternion order or ideal S.

`Basis(S)`

The integral basis of the order or ideal S.

`EmbeddingMatrix(S)`

Given a quaternion order or ideal, returns the matrix defining the embedding in its quaternion algebra.

`Discriminant(S)`

Given an order S over $\mathbf{Z}$ in a quaternion algebra A over $\mathbf{Q}$, returns the reduced discriminant of S.

`Level(S)`

Given an order S over $\mathbf{Z}$ in a quaternion algebra A over $\mathbf{Q}$, returns the index in a maximal order of A containing it. Together with the reduced discriminant of the order, this serves to classify the local isomorphism class of an Eichler order. For an ideal, this returns the level of the left or right order.

Example E7

We note that the printing of order or order elements is in terms of the algebra which contains them, while the coordinates of the basis elements are with respect to the ideal or order elements.

```
> S := QuaternionOrder(11);
> i := S.1;
> P<x> := PolynomialRing(Integers());
> MinimalPolynomial(i);
x^2 + 1
> // Create an ideal generated by 2, 1 + i (of norm 2)
> I := LeftIdeal(S,[2,1+i]);
> I;
Quaternion Ideal of level (11, 1) with base ring Integer Ring
> Basis(I);
[ 1 + j + k, i + j + k, 2*j, 2*k ]
```

```
> [ Eltseq(x) : x in Basis(I) ];
[
[ 1, 0, 0, 0 ],
[ 0, 1, 0, 0 ],
[ 0, 0, 1, 0 ],
[ 0, 0, 0, 1 ]
]
```

To get the rational coordinates with respect to the quaternion algebra, it is necessary to explicitly coerce into the algebra. This data is equivalent to the function embedding matrix (which is only applied at the time of printing).

```
> A := QuaternionAlgebra(S);
> [ Eltseq(A!x) : x in Basis(I) ];
[
[ 1, 0, 1, 1 ],
[ 0, 1, 0, 1 ],
[ 0, 0, 2, 0 ],
[ 0, 0, 0, 2 ]
]
> EmbeddingMatrix(I);
[1 0 1 1]
[0 1 0 1]
[0 0 2 0]
[0 0 0 2]
```

---

§8 Ideal Theory of Orders

The ideal theory for orders over a maximal order R in quaternion algebras over a number field K is analogous to the ideal theory for order in quadratic extensions of K. For sufficiently nice classes of orders and ideals, every 1 or 2-sided ideal will be generated by two elements, which can be taken to lie in some quadratic extension of the center, i.e. is just the image of an ideal in a commutative quadratic order. On the other hand the left and right-module structures and the left and right orders, must be differentiated.

In terms of structures in MAGMA, ideals are only weakly differentiated from orders. Since the category of associative algebras does not require that an algebra have a unit, ideals are valid objects in the category of associative algebras. With this view, orders and ideals share a common type `AlgQuatOrd`, while printing is differentiated, depending whether the structure contains the unity element, thus forms an order.

The ideal theory of definite orders over $\mathbf{Z}$ is highly developed, allowing enumeration of all classes of ideals locally free over an order S, which is assumed to be an Eichler order or for which the index in an Eichler order is of valuation at most 1 at each ramified prime of the algebra. More general orders are supported but may not give complete results in the enumeration of all ideal classes.

§8.1 Creation and Access Functions

`LeftOrder(I)`

Given an ideal I, returns the left order of I, defined as the ring of all elements of the quaternion algebra of I mapping I to itself under left multiplication.

`RightOrder(I)`

Given an ideal I, returns the right order of I, defined as the ring of all elements of the quaternion algebra of I mapping I to itself under right multiplication.

`LeftIdeal(S,X)`

`lideal< S | X >`

The left ideal of S generated by the sequence S of elements in a quaternion algebra or order. The constructor `lideal< | >` permits a variety of arguments on the right, from sequences of elements to one or more elements.

`RightIdeal(S,X)`

`rideal< S | X >`

The right ideal of S generated by the sequence X of elements in a quaternion algebra or order. The constructor `rideal< | >` permits a variety of arguments on the right, from sequences of elements to one or more elements.

`ideal< S | X >`

The 2-sided ideal of S generated by the sequence X of elements in a quaternion algebra or order. The argument X can be a variety of data for elements, including a sequence of elements, a comma separated list of elements, or the coordinates of elements.

`PrimeIdeal(S,p)`

Given a quaternion order S over $\mathbf{Z}$, returns the unique 2-sided prime ideal P of S over the prime $p\mathbf{Z}$. If p is a ramified prime then P properly contains pS and need not be principal, and otherwise P is equal to pS.

`CommutatorIdeal(S)`

The two-sided ideal of S generated by elements of the form $xy - yx$.

Example E8

We demonstrate the construction of the 2-sided prime ideals and the relation with the commutator ideal.

```
> S := QuaternionOrder(2*5*11);
> P := PrimeIdeal(S,2);
> I := ideal< S | 2, [ x*y-y*x : x, y in Basis(S) ] >;
> P eq I;
true
> Q := PrimeIdeal(S,5);
> R := PrimeIdeal(S,11);
> P*Q*R eq CommutatorIdeal(S);
true
```

By way of explanation, we note that the ideal composition is well-defined, since each of these ideals is a 2-sided ideal for S, that is, a left ideal whose right order is also S. The collection of prime ideals over the ramified primes of a maximal order forms an elementary 2-abelian class group, and the commutator ideal is the product of these prime ideals.

§8.2 Enumeration of Ideal Classes

`LeftIdealClasses(S)`

A sequence of representatives for the left locally free ideal classes of S, where S is an order over $\mathbf{Z}$ in a definite quaternion algebra.

`RightIdealClasses(S)`

A sequence of representatives for the right locally free ideal classes of S, where S is an order over $\mathbf{Z}$ in a definite quaternion algebra.

`TwoSidedIdealClasses(S)`

Given an order over $\mathbf{Z}$ in a definite quaternion algebra, returns the sequence of 2-sided ideal class representatives.

Example E9

In the following example we construct a maximal order in the quaternion algebra ramified at 37, and enumerate the left ideal classes.

```
> S := QuaternionOrder(37);
> ideals := LeftIdealClasses(S);
> ideals;
[
    Quaternion Order of level (37, 1) with base ring Integer Ring,
    Quaternion Ideal of level (37, 1) with base ring Integer Ring,
    Quaternion Ideal of level (37, 1) with base ring Integer Ring
]
> [ Basis(I) : I in ideals ];
[
    [ 1, i, j, k ],
    [ 1 + j + k, i + k, 2*j, 2*k ],
    [ 2, i + j, 2*j, k ]
]
```

Note that these coordinates and the embedding matrix are all defined over the base field of the algebra – here the rationals. This is necessary, since even an integral basis of an order over **Z** can have arbitrary denominators in the algebra embedding, as is demonstrated by the example of the right orders of the computed ideal classes.

```
> _, I, J := Explode(ideals);
> R1 := RightOrder(I);
> Basis(R1);
[ 1, 1/2 - 1/2*j + 1/2*k, -1/2 - i + 1/2*j + 1/2*k, -i - j ]
> R2 := RightOrder(J);
> Basis(R2);
[ 1, 1/2*i - 1/2*j - 1/2*k, 1/2*i - 1/2*j + 1/2*k, i + j ]
> IsIsomorphic(R1,R2);
true
```

Note that the ideals I and J are nonisomorphic left ideals over S, yet also have isomorphic right orders. In the example following the next section we explore this phenomenon further.

---

§8.3 Operations on Ideals

`I * J`

`Composite(I,J)`

The composite of I and J, where the right order of I equals the left order of J.

`Conjugate(I)`

Given an ideal or order I, returns the conjugate ideal. In the base of an order this is tautologically the same object.

`I meet J`

Given ideals or orders I and J, returns the intersection $I \cap J$.

`Norm(I)`

Given an ideal I over **Z**, returns the norm of the ideal, defined as the positive generator of the image of the norm map in **Z**.

Example E10

Let's begin by creating the ideals of the previous example. Then we find an isomorphism between the

```
> A := QuaternionAlgebra(37);
> S := MaximalOrder(A);
> ideals := LeftIdealClasses(S);
> _, I, J := Explode(ideals);
> R := RightOrder(I);
> Q := RightOrder(J);
> IsIsomorphic(R,Q);
true
> // Get the x which conjugates R to Q:
> _, pi := Isomorphism(R,Q);
> Norm(pi);
37
> J := lideal< S | [ x*pi : x in Basis(J) ] >;
> RightOrder(J) eq R;
true
```

§9 Norm Spaces and Basis Reduction

`NormSpace(A)`

`NormSpace(S)`

`NormModule(S)`

Given an algebra A or an order or ideal S, returns the underlying space or module over its base ring, with inner product respect to the norm, followed by the map into the structure.

`GramMatrix(S)`

The Gram matrix with respect to the norm on the basis for S.

`ReducedBasis(S)`

Given an order or ideal S over $\mathbf{Z}$ in a definite quaternion algebra, returns a Minkowski-reduced basis for S.

`ReducedGramMatrix(S)`

The unique Minkowski-reduced Gram matrix of a reduced basis for the definite quaternion order or ideal S.

Example E11

The quaternion ideal machinery makes use of a Minkowski basis reduction algorithm which returns a uniquely normalized reduced Gram matrix for any definite quaternion ideal. This forms the core of the isomorphism testing for quaternion ideals.

```
> A := QuaternionOrder(19,2);
> ideals := LeftIdealClasses(A);
> #ideals;
5
> [ (1/Norm(I))*ReducedGramMatrix(I) : I in ideals ];
[
    [ 2  0  1  1]
    [ 0  2  1  1]
    [ 1  1 20  1]
    [ 1  1  1 20],

    [6 0 1 3]
    [0 6 3 1]
    [1 3 8 1]
    [3 1 1 8],

    [6 0 1 3]
    [0 6 3 1]
    [1 3 8 1]
    [3 1 1 8],

    [ 4  0  1 -1]
    [ 0  4  1  1]
    [ 1  1 10  0]
    [-1  1  0 10],

    [ 4  0  1 -1]
    [ 0  4  1  1]
    [ 1  1 10  0]
    [-1  1  0 10]
]
```

§10 Order and Ideal Isomorphisms

Using the inner product on a definite quaternion order, we can use basis reduction and lattice isomorphism techniques to compute isomorphisms between orders and ideals. Note that two ideals may be isomorphic as algebras (in the general sense, not implying existence of a unity), even if they share no left or right order. Since the isomorphism is given by conjugation, this implies, in particular, that they have the same norm. On the other hand, left and right isomorphisms are defined to be isomorphisms of modules, which in general scales the ideal norm by the norm some element which defines the isomorphism by right or left multiplication.

`IsIsomorphic(A,B)`

`IsIsomorphic(I,J)`

Given two definite quaternion algebras, orders, or ideals (over $\mathbf{Q}$, $\mathbf{Z}$, or $\mathbf{Z}$, respectively), returns `true` if and only if they are isomorphic as algebras.

`Isomorphism(A,B)`

`Isomorphism(I,J)`

Given two isomorphic definite quaternion algebras, orders, or ideals (over $\mathbf{Q}$, $\mathbf{Z}$, or $\mathbf{Z}$, respectively), returns an algebra isomorphism.

`IsLeftIsomorphic(I,J)`

Given two definite ideals over $\mathbf{Z}$ with the same left order S, returns `true` if and only if they are isomorphic as S-modules. The isomorphism and the transforming scalar – in the quaternion algebra – are returned as second and third values if true.

`LeftIsomorphism(I,J)`

Given two isomorphism left ideals over a definite order S, returns the S-module isomorphism between them, followed by the quaternion algebra element which defines the isomorphism by right multiplication.

`IsRightIsomorphic(I,J)`

Given two definite ideals over $\mathbf{Z}$ with the same right order S, returns `true` if and only if they are isomorphic as S-modules. The isomorphism and the transforming scalar – in the quaternion algebra – are returned as second and third values if true.

```
RightIsomorphism(I,J)
```

Given two isomorphism right ideals over a definite order S, returns the S-module isomorphism between them, followed by the quaternion algebra element which defines the isomorphism by left multiplication.

Example E12

We recall the construction of two non-isomorphic left ideals with the same left and right orders, then investigate their isomorphisms as right ideals.

```
> S := QuaternionOrder(37);
> ideals := LeftIdealClasses(S);
> _, I, J := Explode(ideals);
> R := RightOrder(I);
> _, pi := Isomorphism(R,RightOrder(J));
> J := lideal< S | [ x*pi : x in Basis(J) ] >;
> IsLeftIsomorphic(I,J);
true
> IsRightIsomorphic(I,J);
true Mapping from: AlgQuatOrd: I to AlgQuatOrd: J given by a rule [no inverse]
1 + i - 2*k
> h, x := RightIsomorphism(I,J);
> y := I![1,2,-1,3];
> h(y);
185 - 11*i - 27*j - 10*k
> x*y;
185 - 11*i - 27*j - 10*k
```

The existence of an isomorphism as a right ideal is due to the fact that the two 2-sided ideals of R do not have non-isomorphic counterparts in S.

```
> TwoSidedIdealClasses(R);
[
    Quaternion Order of level (37, 1) with base ring Integer Ring,
    Quaternion Ideal of level (37, 1) with base ring Integer Ring
]
> TwoSidedIdealClasses(S);
[
    Quaternion Order of level (37, 1) with base ring Integer Ring
]
```

Thus while `Conjugate(I)*J` is in the nonprincipal R-ideal class, the ideal `I*Conjugate(J)` represents the unique principal ideal class of S.

§11 Units and Unit Groups

`Units(S)`

Given a definite quaternion order S over $\mathbf{Z}$, returns a sequence of representatives of the units in S modulo the unit group of the base ring.

`UnitGroup(S)`

Given a definite quaternion order S over $\mathbf{Z}$, returns an abstract group isomorphic to S^* and the homomorphism into S.

Example E13

The following example illustrates the unit group computation for an order in a definite quaternion algebra over $\mathbf{Q}$.

```
> A := QuaternionAlgebra< RationalField() | -1, -1 >;
> S1 := MaximalOrder(A);
> S2 := QuaternionOrder(A,2);
> G1, h1 := UnitGroup(S1);
> #G1;
24
> [ h1(g) : g in G1 ];
[ 1, -1, i - k, -j + k, 1 - i - j, -j, 1 - k, -1 + i + j - k, -i, i,
1 - i - j + k, j, -1 + k, k, -i - j + k, 1 - j, -1 + i, 1 - i, -1 + j,
-k, i + j - k, j - k, -1 + i + j, -i + k ]
> G2, h2 := UnitGroup(S2);
> #G2;
8
> [ h2(g) : g in G2 ];
[ 1, -1, -i + k, i - k, -1 + i + j, 1 - i - j, -j + k, j - k ]
```

The unit groups of orders in indefinite quaternion algebras A are infinite arithmetic groups, which are twisted analogues of the groups $\mathrm{SL}_2(\mathbf{Z})$ and their families of subgroups. These are studied in relation to their actions on the upper half complex plane, via an embedding in $\mathrm{GL}_2(\mathbf{R})$ provided by some isomorphism $A \otimes \mathbf{R} \cong \mathrm{M}_2(\mathbf{R})$. Currently this perspective is not fully implemented, but is expected to be part of the generalization of the machinery for hyperbolic spaces and congruence subgroups.

§12 Bibliography

[**Vig80**] M.-F. Vignéras. *Arithmétique des Algèbres de Quaternions*, volume 800 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 1980.

A1 AlgQuat Package

attributes.m

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                        Attribute declarations                      //
//                                                                    //
////////////////////////////////////////////////////////////////////////
declare attributes AlgQuat:
   NormSpace;
declare attributes AlgQuatOrd:
   ReducedMatrix,
   ReducedGram,
   NormSpace,
   LeftIdealBases;
```

algebras.m

```
///////////////////////////////////////////////////////////////////////
//                                                                   //
//                        QUATERNION ALGEBRAS                        //
//                          by David Kohel                           //
//                                                                   //
///////////////////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////////////////
//                                                                   //
//                      Creation Functions                           //
//                                                                   //
///////////////////////////////////////////////////////////////////////
function Discriminants(N,M,B)
   // PRE: N is a product of ODD prime powers, and M is
   // relatively prime to N.
   // POST: Determines a list of discriminants in which all
   // prime factors of N are inert, and all prime factors of
   // M are split, up to a bound in N*M.
   B0 := Isqrt(4*N*M);
   error if GCD(N,M) ne 1,
      "Common prime in split and ramifying lists.";
   RamifyingFactors := Factorization(N);
   SplitFactors := Factorization(M);
   ReturnDiscs := [ -4*n+j : j in [1,0] , n in [Max(1,B0-B)..B0+B] ];
   for D in ReturnDiscs do
      for q in RamifyingFactors do
         if KroneckerSymbol(D,q[1]) ne -1 then
            Exclude(~ReturnDiscs,D);
         end if;
      end for;
      for p in SplitFactors do
         if KroneckerSymbol(D,p[1]) ne 1 then
            Exclude(~ReturnDiscs,D);
         end if;
      end for;
   end for;
   return ReturnDiscs;
end function;
function TraceValues(D1,D2,N,M)
   // PRE: All factors of N inert and all factors of M split
   // in D1 and D2.
   // POST: For an integer T set D = (D1*D2 - T^2).  Returns
   // the T for which D is positive, D mod 4*N*M is zero,
   // Q = D div 4*N*M is relatively prime to N, and all
   // factors of Q are either split in one of D1 or D2 or
   // inert in D1 or D2 and divide Q to an even power.
   // Stupid, but here goes...
   Tmax := Isqrt(D1*D2 div 4);
   T := -Tmax;
   prms := PrimeDivisors(N) cat PrimeDivisors(M);
   rts := [ Integers() | ];
   Tvals := [ (D1 mod 2)*(D2 mod 2) ];
   m := 2;
   if 2 in prms then
      Exclude(~prms,2);
      m := 8;
      D := ((D1 mod 4)*(D2 mod 4)) mod 8;
      if D in { 0, 1 } then
         Tvals := [ D, D + 4 ];
      elif D eq 4 then
         Tvals := [ 2, 6 ];
      end if;
   end if;
   for p in prms do
      t := PolynomialRing(FiniteField(p)).1;
      a := Roots(t^2 - D1*D2)[1][1];
      Append(~rts,Integers()!a);
   end for;
   for i in [1..#prms] do
      p := prms[i];
      a := rts[i];
      Tvals := &join[ { x : x in &join[ { y, p*m - y } :
         y in [ CRT([a0,a1],[p,m]) : a0 in [-a,a] ] ]
         | Abs(x) le Tmax } : a1 in Tvals ];
      m *:= p;
   end for;
   for T in Tvals do
      Q := (D1*D2 - T^2) div (4*N*M);
      if Q eq 1 then
         return [ T ];
      elif GCD(Q,N) ne 1 then
         Exclude(~Tvals,T);
      else
         OtherFactors := Factorization(Q);
         for p in OtherFactors do
            if KroneckerSymbol(D1,p[1]) ne 1 or
               KroneckerSymbol(D2,p[1]) ne 1 then
```

```
               Exclude(~Tvals,T);
            end if;
         end for;
      end if;
   end for;
return [ T : T in Tvals ];
end function;


intrinsic QuaternionAlgebra(N::RngIntElt) -> AlgQuat
   {Returns a rational quaternion algebra of discriminant N.}
   fact := Factorization(N);
   require (#fact mod 2) eq 1 :
      "This constructor exists only for definite quaternions.";
   require N gt 0 and &and [ p[2] eq 1 : p in fact ] :
      "Discriminant must be a square-free positive integer.";
   // First test for the easy cases.
   if N eq 2 then
      return QuaternionAlgebra(-3,-3,1);
   elif (N mod 4) eq 3 and
     &and [ KroneckerSymbol(-4,p[1]) eq -1 : p in fact ] then
     return QuaternionAlgebra(-4,-N,0);
   elif (N mod 8) eq 5 and
     &and [ KroneckerSymbol(-8,p[1]) eq -1 : p in fact ] then
     return QuaternionAlgebra(-8,-((N+1) div 2),2);
   end if;
   B := Ceiling(Log(N));
   for k in [1..8] do
      DiscList := Discriminants(N,1,2^k*B);
      DiscPairs := [ [D1,D2] : D2, D1 in DiscList | D1*D2 gt (4*N) ];
      for D in DiscPairs do
         TList := TraceValues(D[1],D[2],N,1);
         if #TList gt 0 then
            A := QuaternionAlgebra(D[1],D[2],TList[1]);
            if Discriminant(A) eq N then
               return A;
            end if;
         end if;
      end for;
   end for;
   return "Error, no order found (please report).";
end intrinsic;
function RandomElement(A,S)
   return A![ Random(S) : i in [1..4] ];
```

```
end function;
//////////////////////////////////////////////////////////////////////////
//                                                                      //
//                      Units and Unit Groups                           //
//                                                                      //
//////////////////////////////////////////////////////////////////////////


intrinsic Units(S::AlgQuatOrd) -> SeqEnum
   {The units in the quaternion order S over the integers.}
   K := QuaternionAlgebra(S);
   require Type(BaseRing(S)) eq RngInt :
      "Base ring of argument must be the integers.";
   require IsDefinite(K) : "Quaternion algebra must be definite.";
   if K!1 notin S then return [ S | ]; end if;
   X := ShortestVectors(LatticeWithGram(GramMatrix(S)));
   U := [ u * S!Eltseq(v) : u in [1,-1], v in X ];
   ords := [ #{ u^j : j in [0..5] } : u in U ];
   if #U le 6 then // cyclic group
       i := Index(ords,#U);
       U := [ (-1)^t*U[i]^j : t in [0,1], j in [0..(#U div 2) -1] ];
   elif #U eq 8 then
       C4 := {@ {U[i],-U[i]} : i in [1..8] | ords[i] eq 4 @};
       U := [S!1,S!-1] cat &cat[ [u,-u] where u := Rep(c) : c in C4 ];
   elif #U eq 12 then
       i := Index(ords,4); j := Index(ords,3);
       U := [ (-1)^t*U[i]^n*U[j]^m : t, n in [0..1], m in [0..2] ];
   elif #U eq 24 then
       i := Index(ords,3);
       C4 := {@ {U[i],-U[i]} : i in [1..24] | ords[i] eq 4 @};
       U4 := [S!1,S!-1] cat &cat[ [u,-u] where u := Rep(c) : c in C4 ];
       U := [ u*U[i]^j : u in U4, j in [0..2] ];
   end if;
   return U;
end intrinsic;
function RightRegularRepresentation(U,X)
    G := Sym(#U);
    gens := [ G![ Index(U,x*u) : x in U ] : u in U ];
    if #X eq 1 then
        H := sub< G | gens >;
    else
        subgens := [ G![ Index(U,x*u) : x in U ] : u in X ];
        H := sub< G | gens >;
        K := sub< G | subgens >;
        m := RegularRepresentation(H,K);
        gens := [ m(g) : g in H ];
        H := Codomain(m);
    end if;
    iso := [ <gens[i],U[i]> : i in [1..#U] ];
    return H, map< H -> U | iso >;
end function;


intrinsic UnitGroup(S::AlgQuatOrd) -> GrpPerm, Map
    {}
   A := QuaternionAlgebra(S);
   require Type(BaseRing(S)) eq RngInt :
      "Base ring of argument must be the integers.";
   require IsDefinite(A) : "Quaternion algebra must be definite.";
   require A!1 in S : "Argument must be a quaternion order.";
   U := Units(S);
   if #U ne 24 then
       X := [ U[1] ];
   else
       X := [ U[1], U[9], U[17] ];
   end if;
   return RightRegularRepresentation(Units(S),X);
end intrinsic;
```

orders.m

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                         Quaternion Orders                          //
//                            David Kohel                             //
//                                                                    //
////////////////////////////////////////////////////////////////////////
forward IntBasisClosure, PolyBasisClosure;
forward LLLAlgQuat, LLLSeq;
///////////////////////////////////////////////////////////////////////
//                    Auxiliary Random Functions                     //
///////////////////////////////////////////////////////////////////////
function RandomElement(A,S)
   return A![ Random(S) : i in [1..4] ];
end function;


intrinsic MaximalOrder(A::AlgQuat) -> AlgQuatOrd
    {A maximal quaternion order in A.}
    require Type(BaseField(A)) eq FldRat :
        "Argument must be a quaternion algebra over the rationals.";
    ZZ := Integers();
    V := NormSpace(A);
    N := ZZ!Discriminant(A);
    D := Isqrt(ZZ!Determinant(GramMatrix(V)));
    m := D div N;
    B := Basis(A);
    fact := [ p[1] : p in Factorization(m) ];
    while #fact ne 0 do
p := fact[1];
FF := FiniteField(p);
B1 := Basis(Kernel(
    Matrix(4,[FF|Trace(x*Conjugate(y)) : x, y in B])));
if p eq 2 then
    X := [ &+[ (ZZ!v[i])*B[i] : i in [1..4] ] : v in B1 ];
    for c in CartesianPower({0,1},#B1) do
if &or [ c[i] eq 1 : i in [1..#B1] ] then
    z := &+[ c[i]*X[i] : i in [1..#B1] ];
    if ZZ!Norm(z) mod 4 eq 0 then
break c;
    end if;
end if;
    end for;
```

```
elif #B1 eq 1 then
    z := &+[ (ZZ!B1[1][i])*B[i] : i in [1..4] ];
else
    // The inner product of elements in B1 with all other elements
    // in B1 is divisible by p.  Now we need to solve for a linear
    // combination z whose norm (i.e. inner product with itself) is
    // divisible by p^2, so that z/p is integral.
    X := [ &+[ (ZZ!v[i])*B[i] : i in [1..4] ] : v in B1 ];
    T := Matrix(#X,
[ FF | ZZ!(Trace(x*Conjugate(y))/p) : x, y in X ]);
    B2 := Basis(Kernel(T));
    if #B2 ne 0 then
z := &+[ (ZZ!B2[1][i])*X[i] : i in [1..#X] ];
    elif #B1 eq 2 then
assert #X eq 2;
P := PolynomialRing(GF(p));
f := T[1,1]*P.1^2 + 2*T[1,2]*P.1 + T[2,2];
z := (ZZ!r)*X[1] + X[2] where r := Roots(f)[1][1];
    else // T defines Gram matrix of a nonsingular conic
assert #B1 eq 3;
assert #X eq 3;
P2 := ProjectiveSpace(GF(p),2);
C := Conic(P2,&+[ T[i,j]*P2.i*P2.j : i, j in [1..3] ]);
s := Eltseq(RationalPoint(C));
z := &+[ (ZZ!s[i])*X[i] : i in [1..3] ];
    end if;
end if;
B := LLLAlgQuat(B cat [z/p]);
m div:= p;
if (m mod p) ne 0 then
    Exclude(~fact,p);
end if;
    end while;
    return QuaternionOrder([ u*x : x in B ]) where u is B[1]^-1;
end intrinsic;
//////////////////////////////////////////////////////////////////////////


intrinsic IMT(A::AlgQuatOrd,B::AlgQuatOrd) -> AlgQuatOrd
   {}
   // internal intersection function -- called by 'meet'
   K := QuaternionAlgebra(A);
   require QuaternionAlgebra(B) eq K :
      "Orders have different quotient algebras";
   R := BaseRing(A);
   require Type(R) in {RngInt,RngUPol} :
      "Base ring is of not of integral type";
   BA := [ K!x : x in Basis(A) ];
   BB := [ K!x : x in Basis(B) ];
   d := LCM([ LCM([ Denominator(a) : a in Eltseq(x) ])
      : x in BA cat BB ]);
   M := RSpace(R,4);
   N := sub< M | [ M!Eltseq(d*x) : x in BA ] > meet
        sub< M | [ M!Eltseq(d*x) : x in BB ] >;
   S := LLLAlgQuat( [ (K!Eltseq(x))/d : x in Basis(N) ] );
   return QuaternionOrder([ S[1]^-1*x : x in S ]);
end intrinsic;


intrinsic QuaternionOrder(S::[AlgQuatElt] : IsBasis := false)
    -> AlgQuatOrd
    {Quaternion order generated by S, preserving S if it constitutes
    a basis with initial element 1.}
    A := Universe(S);
    k := BaseField(A);
    if Type(k) eq FldRat then
if not IsBasis then
    test, S := IntBasisClosure(S);
    require #S eq 4 : "Argument does not generate an order.";
    require test : "Argument not closed under multiplication.";
    S := [ r*x : x in S ] where r := S[1]^-1;
end if;
R := Integers();
    elif Type(k) eq FldFunRat then
if not IsBasis then
    test, S := PolyBasisClosure(S);
    require #S eq 4 : "Argument does not generate an order.";
    require test : "Argument not closed under multiplication.";
end if;
R := PolynomialRing(BaseRing(k));
    else
require false :
    "Base ring must be specified for order creation.";
    end if;
    require #S eq 4 : "Argument does not define an order.";
    return QuaternionOrder(R,S);
end intrinsic;
```

```
intrinsic QuaternionOrder(A::AlgQuat,M::RngIntElt) -> AlgQuatOrd
    {Returns a quaternion order of level M in A.}
    require Type(BaseField(A)) eq FldRat :
"Argument must be quaternion algebra over the rationals.";
    M := Abs(M);
    N := Integers()!Discriminant(A);
    N1 := GCD(M,N);
    M1 := M div N1;
    O := MaximalOrder(A);
    if N1 ne 1 then
require Max([ Valuation(M,p) : p in RamifiedPrimes(A) ]) le 1 :
    "Argument 2 can be divisible by at most " *
    "the first power of a ramified prime.";
P := lideal< O | [ O | N1 ] cat [ x*y-y*x : x,y in Basis(O) ] >;
O := QuaternionOrder([ A!x : x in Basis(P) ]);
    end if;
    fact := Factorization(M1);
    for p in fact do
repeat
    x := RandomElement(O,[-p[1] div 2..p[1] div 2]);
    D := Trace(x)^2 - 4*Norm(x);
until KroneckerSymbol(D,p[1]) eq 1;
P := PolynomialRing(FiniteField(p[1]));
X := P.1;
a := Integers()!Roots(X^2 - Trace(x)*X + Norm(x))[1][1];
I := lideal< O | [ O | p[1]^p[2], (x-a)^p[2] ]>;
O := O meet RightOrder(I);
    end for;
    return O;
end intrinsic;


intrinsic QuaternionOrder(N::RngIntElt) -> AlgQuatOrd
    {Returns a maximal order in the rational quaternion algebra
    of discriminant N.}
    prms := PrimeDivisors(N);
    require #prms mod 2 eq 1 :
        "Argument 1 must be a product of an odd number of primes.";
    require Max([ Valuation(N,p) : p in prms ]) le 1 :
        "Argument 1 can have valuation at most 1 at each prime.";
    return MaximalOrder(QuaternionAlgebra(N));
end intrinsic;
```

```
intrinsic QuaternionOrder(N::RngIntElt,M::RngIntElt) -> AlgQuatOrd
    {Returns a quaternion order of level M in the rational quaternion
    algebra of discrinant N.}
    prms := PrimeDivisors(N);
    require #prms mod 2 eq 1 :
"Argument 1 must be a product of an odd number of primes.";
    require Max([ Valuation(N,p) : p in prms ]) le 1 :
"Argument 1 can have valuation at most 1 at each prime.";
    require Max([ Valuation(M,p) : p in prms ]) le 1 :
"Argument 2 can have valuation at most 1 " *
"at each prime divisor of argument 2.";
    return QuaternionOrder(QuaternionAlgebra(N),M);
end intrinsic;
////////////////////////////////////////////////////////////////////////
//                      Some Basis Reduction                          //
////////////////////////////////////////////////////////////////////////
function LLLSeq(B)
    // {An LLL reduced basis of the sublattice generated by B.}
    V := Universe(B);
    if Category(V) eq Lat then
return Basis(LLL(sub< V | B >));
    elif Category(V) in { ModTupFld, ModTupRng } then
N := LLL( Matrix(B) );
return [ N[i] : i in [1..Rank(N)] ];
    end if;
    error if false, "Invalid universe of argument";
end function;
function LLLAlgQuat(S)
    K := Universe(S);
    error if not Type(BaseRing(K)) eq FldRat,
       "Basis reduction valid only over the integers";
    if IsDefinite(K) then
        L := LatticeWithGram( MatrixAlgebra(BaseField(K),4) !
                 [ Trace(x*Conjugate(y)) : x, y in Basis(K) ] );
        T := &cat[ Eltseq(x) : x in S ];
        n := LCM([ Denominator(a) : a in T ]);
        M := LLL(sub< L | Matrix(4,[ Integers() | n*a : a in T ]) >);
        return [ (K!Eltseq(B[i]))/n : i in [1..4] ] where B := Basis(M);
    else
        V := RSpace(Integers(),4);
        T := &cat[ Eltseq(x) : x in S ];
        n := LCM([ Denominator(a) : a in T ]);
        M := Matrix(4,[ Integers() | n*a : a in T ]);
        U := sub< V | [ M[i] : i in [1..Nrows(M)] ] >;
        one := V![n,0,0,0];
        if one in U and GCD(Coordinates(U,one)) eq 1 then
            W, pi := quo< U | one >;
            B := [ one ] cat [ v@@pi : v in Basis(W) ];
        else
            B := Basis(U);
        end if;
        return [ (K!Eltseq(B[i]))/n : i in [1..Rank(U)] ];
    end if;
end function;
////////////////////////////////////////////////////////////////////////
function IntBasisClosure(S)
    // over the rationals!!!
    A := Universe(S);
    V := KSpace(RationalField(),4);
    if not &or [ x eq A!1 : x in S ] then
S := [ A!1 ] cat S;
    end if;
    S := LLLAlgQuat(S);
    // Problems if #S > 4 seen...
    if #S lt 4 then
S := LLLAlgQuat([ x*y : x, y in S ]);
if #S lt 4 then
    return false, S;
end if;
    end if;
    assert #S eq 4;
    M := Matrix([ Eltseq(x) : x in S ])^-1;
    k := 1;
    while k lt 4 do
T := [ (V!Eltseq(x*y))*M : x, y in S ];
bool := &and [ &and[ Denominator(a) eq 1 :
    a in Eltseq(v) ] : v in T ];
if bool then
    return true, S;
end if;
S := LLLAlgQuat([ A!Eltseq(v) : v in LLLSeq(T) ]);
k +:= 1;
    end while;
    return false, S;
end function;
function PolyBasisClosure(S)
```

```
    // over k(x)
    A := Universe(S);
    F := BaseField(A);
    k := BaseRing(F);
    error if not IsField(k),
"Base ring of function field must be a field.";
    M := RSpace(PolynomialRing(k),4);
    if not &or [ x eq A!1 : x in S ] then
S := [ A!1 ] cat S;
    end if;
    g := F!LCM(&cat[ [ Denominator(a) : a in Eltseq(x) ] : x in S ]);
    T := [ M!Eltseq(g*x) : x in S ];
    S := [ A![ f/g : f in Eltseq(v) ] : v in Basis(sub< M | T >) ];
    if #S gt 4 then
S := [ A![ f/g : f in Eltseq(v) ] : v in Basis(sub< M | T >) ];
    elif #S lt 4 then
S := [ x*y : x, y in S ];
g := F!LCM(&cat[
    [ Denominator(a) : a in Eltseq(x) ] : x in S ]);
T := [ M!Eltseq(g*x) : x in S ];
S := [ A![ f/g : f in Eltseq(v) ] : v in Basis(sub< M | T >) ];
if #S lt 4 then return false, S; end if;
    end if;
    V := KSpace(F,4);
    N := Matrix([ Eltseq(x) : x in S ])^-1;
    k := 1;
    while k le 3 do
X := [ (V!Eltseq(x*y))*N : x, y in S ];
bool := &and [ &and [ Denominator(a) eq 1 :
    a in Eltseq(v) ] : v in X ];
if bool then return true, S; end if;
S := [ x*y : x, y in S ];
g := F!LCM(&cat[
    [ Denominator(a) : a in Eltseq(x) ] : x in S ]);
T := [ M!Eltseq(g*x) : x in S ];
S := [ A![ f/g : f in Eltseq(v) ] : v in Basis(sub< M | T >) ];
k +:= 1;
    end while;
    return false, S;
end function;
```

ideals.m

```
//////////////////////////////////////////////////////////////////////////
//                                                                      //
//                    Ideal and Order Constructors                      //
//                            David Kohel                               //
//                                                                      //
//////////////////////////////////////////////////////////////////////////


intrinsic ILO(I::AlgQuatOrd) -> AlgQuatOrd
    {}
    // internal left order function - called from LeftOrder
    n := Norm(I);
    B := [QuaternionAlgebra(I)!x : x in Basis(I)];
    L := QuaternionOrder([x*Conjugate(y)/n : x, y in B]);
    return L;
end intrinsic;


intrinsic IRO(I::AlgQuatOrd) -> AlgQuatOrd
    {}
    // internal right order function - called from RightOrder
    n := Norm(I);
    B := [QuaternionAlgebra(I)!x : x in Basis(I)];
    R := QuaternionOrder([Conjugate(x)*y/n : x, y in B]);
    return R;
end intrinsic;


intrinsic LeftIdeal(A::AlgQuatOrd,S::SeqEnum) -> AlgQuatOrd
    {Left ideal of A with generator sequence S.}
    K := QuaternionAlgebra(A);
    U := Universe(S);
    if Type(U) eq RngInt or U cmpeq BaseRing(A) then
S := [ K!x : x in S ];
    elif Type(U) cmpeq AlgQuatOrd then
require &and [ IsCoercible(A,x) : x in S ] :
    "Incompatible ring and sequence universe.";
    else
require K cmpeq U : "Incompatible ring and sequence universe.";
require &and [ IsCoercible(A,x) : x in S ] :
    "Elements of argument 2 do not coerce into argument 1.";
    end if;
    return lideal< A | S >;
end intrinsic;


intrinsic RightIdeal(A::AlgQuatOrd,S::SeqEnum) -> AlgQuatOrd
    {Right ideal of A with generator sequence S.}
    K := QuaternionAlgebra(A);
    U := Universe(S);
    if Type(U) eq RngInt or U cmpeq BaseRing(A) then
S := [ A!x : x in S ];
    elif Type(U) cmpeq AlgQuatOrd then
require [ IsCoercible(A,x) : x in S ] :
    "Incompatible ring and sequence universe.";
    else
require K cmpeq U : "Incompatible ring and sequence universe.";
require &and [ IsCoercible(A,x) : x in S ] :
    "Elements of argument 2 do not coerce into argument 1.";
    end if;
    return rideal< A | S >;
end intrinsic;


intrinsic CommutatorIdeal(A::AlgQuatOrd) -> AlgQuatOrd
    {The ideal of A generated by elements of the form x*y - y*x.}
    return CommutatorIdeal(A,A);
end intrinsic;


intrinsic Different(A::AlgQuatOrd) -> AlgQuatOrd
    {The ideal of A generated by elements of the form x*y - y*x.}
    return CommutatorIdeal(A,A);
end intrinsic;


intrinsic PrimeIdeal(A::AlgQuatOrd,p::RngIntElt) -> AlgQuatOrd
    {The unique two-sided ideal over p.}
    require Type(BaseRing(A)) eq RngInt:
"Argument 1 must be an order over the integers";
    require IsPrime(p): "Argument 2 must be a prime.";
    if IsRamified(QuaternionAlgebra(A),p) then
return ideal< A | [ A ! p ] cat [ x*y - y*x : x,y  in Basis(A) ]>;
    else
return ideal< A | p >;
```

```
    end if;
end intrinsic;
```

ideal_classes.m

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                        QUATERNION IDEALS                           //
//                     Last Modified April 2000                       //
//                          by David Kohel                            //
//                                                                    //
////////////////////////////////////////////////////////////////////////
forward CompareOrders, CompareLeftIdeals, CompareGram;
forward IsogenousIdeals, SplitIsogenousIdeals;
////////////////////////////////////////////////////////////////////////
//                    Auxiliary Random Functions                      //
////////////////////////////////////////////////////////////////////////
function RandomElement(A,S)
    return A![ Random(S) : i in [1..4] ];
end function;
////////////////////////////////////////////////////////////////////////
//                         Two Sided Ideals                           //
////////////////////////////////////////////////////////////////////////


intrinsic TwoSidedIdealClasses(A::AlgQuatOrd) -> SeqEnum
    {A sequence of representatives for the 2-sided ideal classes of
    the order A.}
    D := ideal< A | [ x*y - y*x : x, y in Basis(A) ] >;
    pows := [ p[1]^p[2] : p in Factorization(Discriminant(A)) ];
    gens := [ ideal< A | [ A!q ] cat Basis(D) > : q in pows ];
    V := VectorSpace(GF(2),#pows);
    reprs := [ ];
    grams := { MatrixAlgebra(Integers(),4) | };
    for v in V do
I := A;
for i in [1..#pows] do
    if v[i] ne 0 then I *:= gens[i]; end if;
end for;
M := (1/Norm(I))*ReducedGramMatrix(I);
if M notin grams then
    Append(~reprs,I);
    Include(~grams,M);
end if;
    end for;
    return reprs;
end intrinsic;
```

```
//////////////////////////////////////////////////////////////////////////
//                                                                      //
//                          Isogeny Graphs                              //
//                                                                      //
//////////////////////////////////////////////////////////////////////////


intrinsic RightIdealClasses(A::AlgQuatOrd) -> SeqEnum
    {Representatives for the right ideals classes of A.}
    require Type(BaseRing(A)) eq RngInt:
"Ideals computed only for orders over the integers";
    require IsDefinite(QuaternionAlgebra(A)) :
"Ideals computed only for orders in definite algebras";
    return [ Conjugate(I) : I in LeftIdealClasses(A) ];
end intrinsic;
function CharacterVector(prms,p)
    return Vector([ GF(2) |
(1 - KroneckerSymbol(p,q)) div 2 : q in prms ]);
end function;


intrinsic LeftIdealClasses(A::AlgQuatOrd : IsogenyPrime := 1) -> SeqEnum
    {Representatives for the left ideals classes of A.}
    /*
    Consider additional parameters:
      IsogenyPrimeSequence, IdealSequence, GeneraSequences, etc.
    */
    require Type(BaseRing(A)) eq RngInt:
"Ideals computed only for orders over the integers";
    require IsDefinite(QuaternionAlgebra(A)) :
"Ideals computed only for orders in definite algebras";
    if assigned A`LeftIdealBases then
return [ lideal< A | [ A!M[i] : i in [1..4] ] >
    where M := UpperTriangularMatrix(B) : B in A`LeftIdealBases ];
    end if;
    K := QuaternionAlgebra(A);
    N := Discriminant(A);
    Q := RamifiedPrimes(K);
    CharacterPrimes := [ q : q in Q | q ne 2 and Valuation(N,q) gt 1 ];
    if IsogenyPrime ne 1 then
require IsPrime(IsogenyPrime) :
    "IsogenyPrime parameter must be prime.";
require N mod IsogenyPrime ne 0 :
    "IsogenyPrime parameter must be coprime to " *
    "the discriminant of the argument.";
p := IsogenyPrime;
CharacterPrimes := [ Integers() | ];
    else
p := 2;
while N mod p eq 0 do
    p := NextPrime(p);
end while;
    end if;
    if #CharacterPrimes eq 0 then
Idls := IsogenousIdeals(A,p);
    else
/*
        Currently the SplitIsogeny function computes the principal
        left ideal class (that of A) and one coset, that generated
        by odd p-power isogenies.  Therefore we have to construct
        enough primes to represent all classes, not just to generate
        all classes.
        Note that this incurs a significant overhead, not just in
        the number of primes, but that each time the principal class
        is recomputed.
        A much more efficient algorithm would be to compute only the
        principal class and then to construct the rest by multiplying
        on the left by the 2-sided ideals for A.
        This algorithm suffers two problems (1) (ALGORITHMIC) a
        verifiable algorithm for the 2-sided ideal classes, and
        (2) (THEORETICAL) some accounting for the fact that due to
        extra automorphisms the idelic generators for 2-sided ideals
        may have different kernels on different kernels under different
        left and right orders.  Provided that the primes act by
        characters which change the genus, then the action of the
        elementary 2-abelian group is free, and (2) is not an
        obstruction.
*/
r := #CharacterPrimes;
V := VectorSpace(GF(2),r);
CharacterClasses := [ v : v in VectorSpace(GF(2),r) | v ne 0 ];
IsogenyPrimes := [ Integers() | ];
while #CharacterClasses gt 0 do
    v := CharacterVector(CharacterPrimes,p);
    while N mod p eq 0 or v notin CharacterClasses do
p := NextPrime(p);
v := CharacterVector(CharacterPrimes,p);
```

```
    end while;
    Include(~IsogenyPrimes,p);
    Exclude(~CharacterClasses,v);
end while;
vprint Quaternion, 2 : "Building p-isogenies for p in", IsogenyPrimes;
IdealGenera := [ ];
for p in IsogenyPrimes do
    pGenera := SplitIsogenousIdeals(A,[A],p);
    if Index(IsogenyPrimes,p) eq 1 then
IdealGenera := pGenera;
    else
Append(~IdealGenera,pGenera[2]);
assert #IdealGenera[1] eq #pGenera[1];
    end if;
end for;
Idls := &cat IdealGenera;
    end if;
    Bases := [];
    for I in Idls do
M := HermiteForm(Matrix([ Eltseq(A!x) : x in Basis(I) ]));
Append(~Bases,Eltseq(M)[[1,2,3,4,6,7,8,11,12,16]]);
    end for;
    A`LeftIdealBases := Bases;
    return Idls;
end intrinsic;
function IsogenousIdeals(A,p)
    // Construct the sequence of left ideal classes for A
    // by means of p-isogenies.
    D := Discriminant(A);
    I := LeftIdeal(A,[A!1]);
    MZ := RSpace(Integers(),2);
    EmptyList := [ MZ | ];
    Orders := [ A ];
    Ideals := [ LeftIdeal(A,[1]) ];
    vprintf Quaternion, 2 : "Ideal number 1, right order module\n%o\n",
NormModule(Orders[1]);
    FF := FiniteField(p);
    PF<X> := PolynomialRing(FF);
    repeat
x1 := RandomElement(A,[-p div 2..p div 2]);
D1 := Trace(x1)^2 - 4*Norm(x1);
    until KroneckerSymbol(D1,p) eq -1;
    repeat
x2 := RandomElement(A,[-p div 2..p div 2]);
```

```
D2 := Trace(x2)^2 - 4*Norm(x2);
    until KroneckerSymbol(D2,p) eq 1;
    a2 := Integers()!Roots(X^2 - Trace(x2)*X + Norm(x2))[1,1];
    x2 := x2 - a2;
    T := 2*Trace(x1*Conjugate(x2)) - Trace(x1)*Trace(x2);
    D := (D1*D2 - T^2) div 4;
    r := 1;
    Frontier := [ [ MZ!P : P in P1Classes(p) ] ];
    while #Frontier[r] gt 0 do
if GetVerbose("Quaternion") ge 1 then
    printf "Frontier at %o-depth %o has %o elements.\n",
p, r, #Frontier[r];
    print "Number of ideals =", #Ideals;
end if;
for Q in Frontier[r] do
    I := LeftIdeal(A,[x2^r*(A!Q[1] + Q[2]*x1), A!p^r]);
    B := RightOrder(I);
    i := 1;
    while i le #Orders and CompareOrders(B,Orders[i]) eq -1 do
i +:= 1;
    end while;
    while i le #Orders and IsIsomorphic(B,Orders[i]) do
if CompareLeftIdeals(Ideals[i],I) eq 0 then
    Exclude(~Frontier[r],Q);
    i := #Orders + 1;
end if;
i +:= 1;
    end while;
    if i eq (#Orders + 1) then
if GetVerbose("Quaternion") ge 2 then
    printf "Ideal number %o, new right order module\n%o\n",
#Orders + 1, NormModule(B);
end if;
Append(~Orders,B);
Append(~Ideals,I);
    elif i le #Orders then
if GetVerbose("Quaternion") ge 2 then
    M := NormModule(B);
    printf "Ideal number %o, right order module\n%o\n",
#Orders + 1, NormModule(B);
end if;
Insert(~Orders,i,B);
Insert(~Ideals,i,I);
    end if;
end for;
Append(~Frontier,EmptyList);
for P in Frontier[r] do
    Q := P;
    if (P[2] mod p) ne 0 then // P[2] eq 1 by assumption.
for t in [0..(p-1)] do
    Q[1] := P[1] + t*p^r;
    Append(~Frontier[r+1],Q);
end for;
    else // P = Q equals <1,0 mod l>.
for t in [0..(p-1)] do
    Q[2] := P[2] + t*p^r;
    Append(~Frontier[r+1],Q);
end for;
    end if;
end for;
r +:= 1; // Increment and continue.
    end while;
    return Ideals;
end function;
function SplitIsogenousIdeals(A,S0,p)
    // Input a sequence S0 of left A-ideals, and a prime p such that
    // all p-isogenous left ideals are in a different genus from S0,
    // build the p^(2r+1)-isogenous ideals sequence S1 while extending
    // S0 with p^2r-isogenous ideals
    D := Discriminant(A);
    MZ := RSpace(Integers(),2);
    EmptyList := [ MZ | ];
    IdealSeq := [ S0, [ Parent(A) | ] ];
    OrderSeq := [ [ RightOrder(I) : I in S0 ], [ Parent(A) | ] ];
    /*
    Additionally for the input sequence of ideals we need a sequence
    indicating whether we have visited or "touched" the ideal class.
    We begin with (the class of) A, so mark this as touched; every
    other class is initially untouched.
    The secondary sequence is being built as we go, so every element
    is by definition touched at the time of creation.  We include
    the second sequence, but it will be the all true sequence.
    This could be omitted if we test the parity of t for each operation
    on the sequence(s).
    Here we assume that A is in the sequence of ideals, but we do
    not assume that it is the first element.
    */
    Touched := [ [ Norm(I) eq 1 : I in S0 ], [ Booleans() | ] ];
    if GetVerbose("Quaternion") ge 2 then
printf "Beginning with %o + %o ideals " *
    "in split isogeny routine.\n", #S0, 0;
    end if;
    FF := FiniteField(p);
    PF<X> := PolynomialRing(FF);
    repeat
x1 := RandomElement(A,[-p div 2..p div 2]);
D1 := Trace(x1)^2 - 4*Norm(x1);
    until KroneckerSymbol(D1,p) eq -1;
    repeat
x2 := RandomElement(A,[-p div 2..p div 2]);
D2 := Trace(x2)^2 - 4*Norm(x2);
    until KroneckerSymbol(D2,p) eq 1;
    a2 := Integers()!Roots(X^2 - Trace(x2)*X + Norm(x2))[1,1];
    x2 := x2 - a2;
    T := 2*Trace(x1*Conjugate(x2)) - Trace(x1)*Trace(x2);
    D := (D1*D2 - T^2) div 4;
    r := 1;
    Frontier := [ [ MZ!P : P in P1Classes(p) ] ];
    while #Frontier[r] gt 0 do
t := (r mod 2) eq 0 select 1 else 2;
if GetVerbose("Quaternion") ge 1 then
    Parity := t eq 1 select "Odd" else "Even";
    printf "Frontier at %o-depth %o has %o elements.\n",
p, r, #Frontier[r];
    h1 := #IdealSeq[1];
    h2 := #IdealSeq[2];
    printf "Number of ideals = %o + %o\n", h1, h2;
    printf "Number of untouched ideals = %o\n",
&+[ Integers() | 1 : i in [1..h1] | not Touched[1][i] ];
end if;
for Q in Frontier[r] do
    I := LeftIdeal(A,[x2^r*(A!Q[1] + Q[2]*x1), A!p^r]);
    B := RightOrder(I);
    i := 1;
    while i le #OrderSeq[t] and
CompareOrders(B,OrderSeq[t][i]) eq -1 do
    i +:= 1;
    end while;
    while i le #OrderSeq[t] and IsIsomorphic(B,OrderSeq[t][i]) do
if CompareLeftIdeals(IdealSeq[t][i],I) eq 0 then
    if not Touched[t][i] then
// Mark ideal as visited and continue.
```

```
Touched[t][i] := true;
    else
Exclude(~Frontier[r],Q);
    end if;
    i := #OrderSeq[t] + 1;
end if;
i +:= 1;
    end while;
    if i eq (#OrderSeq[t] + 1) then
if GetVerbose("Quaternion") ge 2 then
    printf "%o ideal number %o, " *
"new right order module\n%o\n",
Parity, #OrderSeq[t] + 1, NormModule(B);
end if;
Append(~OrderSeq[t],B);
Append(~IdealSeq[t],I);
Append(~Touched[t],true);
    elif i le #OrderSeq[t] then
if GetVerbose("Quaternion") ge 2 then
    M := NormModule(B);
    printf "%o ideal number %o, right order module\n%o\n",
Parity, #OrderSeq[t] + 1, NormModule(B);
end if;
Insert(~OrderSeq[t],i,B);
Insert(~IdealSeq[t],i,I);
Insert(~Touched[t],i,true);
    end if;
end for;
Append(~Frontier,EmptyList);
for P in Frontier[r] do
    Q := P;
    if (P[2] mod p) ne 0 then // P[2] eq 1 by assumption.
for t in [0..(p-1)] do
    Q[1] := P[1] + t*p^r;
    Append(~Frontier[r+1],Q);
end for;
    else // P = Q equals <1,0 mod l>.
for t in [0..(p-1)] do
    Q[2] := P[2] + t*p^r;
    Append(~Frontier[r+1],Q);
end for;
    end if;
end for;
r +:= 1; // Increment and continue.
```

```
    end while;
    return IdealSeq;
end function;
////////////////////////////////////////////////////////////////////////
//                         Comparison Functions                       //
////////////////////////////////////////////////////////////////////////
function CompareOrders(A,B)
    MA := ReducedGramMatrix(A);
    MB := ReducedGramMatrix(B);
    return CompareGram(MA,MB);
end function;
function CompareLeftIdeals(I,J)
    A := RightOrder(J);
    MA := Norm(I)*Norm(J)*ReducedGramMatrix(A);
    MB := ReducedGramMatrix(Conjugate(I)*J);
    return CompareGram(MA,MB);
end function;
function CompareGram(M1, M2)
    // Return 1 if M1 is less than M2, 0 if M1 and M2 are equal,
    // and -1 if M2 is less than M1.
    dim := Degree(Parent(M1));
    for i in [1..dim] do
if M1[i,i] lt M2[i,i] then
    return 1;
elif M1[i,i] gt M2[i,i] then
    return -1;
end if;
    end for;
    for j in [1..dim-1] do
for i in [1..dim-j] do
    if Abs(M1[i,i+j]) gt Abs(M2[i,i+j]) then
return 1;
    elif Abs(M1[i,i+j]) lt Abs(M2[i,i+j]) then
return -1;
    end if;
end for;
    end for;
    for j in [1..dim-1] do
for i in [1..dim-j] do
    if M1[i,i+j] gt M2[i,i+j] then
return 1;
    elif M1[i,i+j] lt M2[i,i+j] then
return -1;
    end if;
end for;
    end for;
    return 0;
end function;
```

arithmetic.m

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                     Arithmetic on Elements                         //
//                          David Kohel                               //
//                                                                    //
////////////////////////////////////////////////////////////////////////


intrinsic CharacteristicPolynomial(x::AlgQuatElt) -> RngUPolElt
   {The characteristic polynomial of x.}
   f := MinimalPolynomial(x);
   return f^(2 div Degree(f));
end intrinsic;


intrinsic CharacteristicPolynomial(x::AlgQuatOrdElt) -> RngUPolElt
   {The characteristic polynomial of x.}
   f := MinimalPolynomial(x);
   return f^(2 div Degree(f));
end intrinsic;


intrinsic Coordinates(x::AlgQuatElt) -> SeqEnum
   {The coordinates of x with respect to the basis of its parent.}
   return Eltseq(x);
end intrinsic;
```

norm_space.m

```
//////////////////////////////////////////////////////////////////////////
//                                                                      //
//                        Norm Spaces and Modules                       //
//                              David Kohel                             //
//                                                                      //
//////////////////////////////////////////////////////////////////////////


intrinsic NormSpace(A::AlgQuat) -> ModFld, Map
    {The inner product space of A with respect to the norm.}
   if not assigned A`NormSpace then
       F := BaseField(A);
       A`NormSpace := RSpace(F,4,MatrixAlgebra(F,4) !
   [ Norm(x+y) - Norm(x) - Norm(y) : x, y in Basis(A) ] );
   end if;
   V := A`NormSpace;
   return V, hom< V->A | x :-> V!Eltseq(x) >;
end intrinsic;


intrinsic NormSpace(B::[AlgQuatElt]) -> ModTupFld
    {The inner product subspace with respect to the norm generated
    by the sequence B.}
    V := NormSpace(Universe(B));
    return sub< V | [ V!Eltseq(x) : x in B ] >;
end intrinsic;


intrinsic NormSpace(O::AlgQuatOrd) -> ModTupRng, Map
    {The inner product module of O with respect to the norm.}
    if not assigned O`NormSpace then
R := BaseRing(O);
O`NormSpace := RSpace( R, 4,
    [ Norm(x+y) - Norm(x) - Norm(y) : x, y in Basis(O) ] );
    end if;
    M := O`NormSpace;
    return M, hom< O -> M | x :-> M!Eltseq(x) >;
end intrinsic;


intrinsic NormModule(O::AlgQuatOrd) -> ModTupRng
    {The inner product space of A with respect to the norm.}
    return NormSpace(O);
end intrinsic;
```

isomorphisms.m

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                    ISOMORPHISM OF QUATERNIONS                      //
//                -- ALGEBRAS, ORDERS, AND IDEALS --                  //
//                      Created January 2001                          //
//                         by David Kohel                             //
//                                                                    //
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
//                      Isomorphism testing                           //
////////////////////////////////////////////////////////////////////////


intrinsic IsIsomorphic(A::AlgQuatOrd,B::AlgQuatOrd) -> BoolElt
    {True if and only if A and B are isomorphic as algebras.}
    K := QuaternionAlgebra(A);
    require K cmpeq QuaternionAlgebra(B) :
"Arguments must be orders in the same algebra.";
    require Type(BaseRing(A)) eq RngInt :
"Arguments must be orders in a quaternion algebra over Q.";
    require IsDefinite(K) :
"Arguments must be orders in a definite quaternion algebra.";
    MA := ReducedGramMatrix(A);
    MB := ReducedGramMatrix(B);
    // This requires that the corresponding reduced Gram matrices
    // are reduced to a canonical representative!!!
    val := MA eq MB;
    // Returning the isomorphism gives a small overhead; this would
    // be minimized by putting this in the kernel and checking whether
    // a second argument is asked for.  Here we omit this step and
    // just return the boolean value.
    return val;
    if val then
h, t := Isomorphism(A,B);
return val, h, t;
    end if;
    return false, _, _;
end intrinsic;


intrinsic IsLeftIsomorphic(I::AlgQuatOrd,J::AlgQuatOrd) -> BoolElt
    {True if and only if I is isomorphic to J as an ideal over a
    common left order.}
    K := QuaternionAlgebra(I);
    require K cmpeq QuaternionAlgebra(J) :
"Arguments must be ideals in the same algebra.";
    require Type(BaseRing(I)) eq RngInt :
"Arguments must be ideals in a quaternion algebra over Q.";
    require IsDefinite(K) :
"Arguments must be ideals in a definite quaternion algebra.";
    require LeftOrder(I) eq LeftOrder(J) :
"Arguments must have the same left order.";
    if Norm(J)*ReducedGramMatrix(I) ne Norm(I)*ReducedGramMatrix(J) then
return false, _, _;
    end if;
    MA := ReducedGramMatrix(RightOrder(I));
    IJ := Conjugate(I)*J;
    MIJ := ReducedGramMatrix(IJ);
    val := MIJ eq Norm(I)*Norm(J)*MA;
    if val then
t := ReducedBasis(IJ)[1]/Norm(I);
Q := [ x*t : x in Basis(I) ];
h := hom< I -> J | x :-> &+[ x[i]*Q[i] : i in [1..4] ] >;
return true, h, t;
    end if;
    return false, _, _;
end intrinsic;


intrinsic IsRightIsomorphic(I::AlgQuatOrd,J::AlgQuatOrd) -> BoolElt
    {True if and only if I is isomorphic to J as an ideal over a
    common right order.}
    K := QuaternionAlgebra(I);
    require K cmpeq QuaternionAlgebra(J) :
"Arguments must be ideals in the same algebra.";
    require Type(BaseRing(I)) eq RngInt :
"Arguments must be ideals in a quaternion algebra over Q.";
    require IsDefinite(K) :
"Arguments must be ideals in a definite quaternion algebra.";
    require RightOrder(I) eq RightOrder(J) :
"Arguments must have the same right order.";
    if Norm(J)*ReducedGramMatrix(I) ne Norm(I)*ReducedGramMatrix(J) then
return false, _, _;
    end if;
    MA := ReducedGramMatrix(LeftOrder(I));
    IJ := I*Conjugate(J);
    MIJ := ReducedGramMatrix(IJ);
    val := MIJ eq Norm(I)*Norm(J)*MA;
    if val then
// I*Conjugate(J) = A*t, so find t.
t := ReducedBasis(IJ)[1]/Norm(I);
Q := [ t*x : x in Basis(I) ];
h := hom< I -> J | x :-> &+[ x[i]*Q[i] : i in [1..4] ] >;
return true, h, t;
    end if;
    return false, _, _;
end intrinsic;
//////////////////////////////////////////////////////////////////////////
//                            Isomorphisms                              //
//////////////////////////////////////////////////////////////////////////


intrinsic Isomorphism(A::AlgQuatOrd,B::AlgQuatOrd) -> Map
    {}
    K := QuaternionAlgebra(A);
    require Norm(A) eq 1 and Norm(B) eq 1 :
"Arguments must be quaternion orders.";
    require K eq QuaternionAlgebra(B) :
"Arguments must be orders in the same algebra.";
    L := LatticeWithGram(GramMatrix(A));
    M := LatticeWithGram(GramMatrix(B));
    val, T := IsIsomorphic(L,M);
    require val : "Arguments must be isomorphic.";
    Q := [ A!T[i] : i in [1..4] ];
    // Ensure that 1 :-> 1.
    if Q[1] ne 1 then
vprintf QuaternionIsomorphism :
    "Lattice isometry off by unit u = %o of argument 1.\n", Q[1];
vprint QuaternionIsomorphism :
    "MinimalPolynomial(u) =", MinimalPolynomial(Q[1]);
Q := [ u*x : x in Q ] where u := A ! Q[1]^-1;
T := Matrix(4,4,&cat[ Eltseq(x) : x in Q ]);
    end if;
    S := T^-1;
    P := [ B!S[i] : i in [1..4] ];
    U := BasisMatrix(A)^-1*S*BasisMatrix(B);
    h := hom< A -> B | x :-> &+[ x[i]*P[i] : i in [1..4] ],
    y :-> &+[ y[i]*Q[i] : i in [1..4] ] >;
    // Ensure that h is a homomorphism, not an anti-homomorphism.
    if h(A.1)*h(A.2) ne h(A.1*A.2) then
```

```
vprint QuaternionIsomorphism :
    "Isometry not an isomorphism, taking conjugates.";
Q := [ Conjugate(x) : x in Q ];
T := Matrix(4,4,&cat[ Eltseq(x) : x in Q ]);
S := T^-1;
P := [ B!S[i] : i in [1..4] ];
h := hom< A -> B | x :-> &+[ x[i]*P[i] : i in [1..4] ],
y :-> &+[ y[i]*Q[i] : i in [1..4] ] >;
U := BasisMatrix(A)^-1*S*BasisMatrix(B);
    end if;
    fac := Factorization(CharacteristicPolynomial(U));
    if GetVerbose("QuaternionIsomorphism") ge 1 then
printf "Factored charpoly: \n%o\n", fac;
    end if;
    // Characteristic polynomial is (X-1)^2*G(X).
    if #fac eq 1 then
// Characteristic polynomial of U is (X-1)^4, return identity.
x := K ! 1;
    elif Degree(fac[2][1]) eq 1 then
V := Kernel(U-1);
x := K!Basis(V)[2];
x -:= K!Trace(x)/2;
x *:= K!Denominator(Norm(x));
    else
chi := fac[2][1];
V1 := Kernel(U-1);
V2 := Kernel(Evaluate(chi,U));
if GetVerbose("QuaternionIsomorphism") ge 1 then
    printf "Kernel(U-1) =\n%o\n", V1;
    print "chi =", chi;
    print "Kernel(Evaluate(chi,U)) =", Kernel(Evaluate(chi,U));
end if;
// Let x be a non-central element of Kernel(U-1),
// y any element of Kernel(chi(U)), and z = y*U.
// Solve for c such that (x+c)^-1*y*(x + c) = z.
// ==> c*(y-z) = (x*z - y*x).
x := K!Basis(V1)[2];
y := K!Basis(V2)[1];
z := K!(Basis(V2)[1]*U);
x +:= (x*z-y*x)/(y-z);
x *:= K!Denominator(Norm(x));
    end if;
    if GetVerbose("QuaternionIsomorphism") ge 1 then
assert &and[ x^-1*y*x in B : y in Basis(A) ];
```

```
    end if;
    return h, x;
end intrinsic;


intrinsic LeftIsomorphism(I::AlgQuatOrd,J::AlgQuatOrd)
    -> Map, AlgQuatElt
    {}
    K := QuaternionAlgebra(I);
    require K cmpeq QuaternionAlgebra(J) :
"Arguments must be ideals in the same algebra.";
    require Type(BaseRing(I)) eq RngInt :
"Arguments must be ideals in a quaternion algebra over Q.";
    require IsDefinite(K) :
"Arguments must be ideals in a definite quaternion algebra.";
    require LeftOrder(I) eq LeftOrder(J) :
"Arguments must have the same left order.";
    val, h, t := IsLeftIsomorphic(I,J);
    require val : "Arguments must be isomorphic as left ideals.";
    return h, t;
end intrinsic;


intrinsic RightIsomorphism(I::AlgQuatOrd,J::AlgQuatOrd)
    -> Map, AlgQuatElt
    {}
    K := QuaternionAlgebra(I);
    require K cmpeq QuaternionAlgebra(J) :
"Arguments must be ideals in the same algebra.";
    require Type(BaseRing(I)) eq RngInt :
"Arguments must be ideals in a quaternion algebra over Q.";
    require IsDefinite(K) :
"Arguments must be ideals in a definite quaternion algebra.";
    require RightOrder(I) eq RightOrder(J) :
"Arguments must have the same right order.";
    val, h, t := IsRightIsomorphic(I,J);
    require val : "Arguments must be isomorphic as right ideals.";
    return h, t;
end intrinsic;
```

ramified_primes.m

```
//////////////////////////////////////////////////////////////////////////
//                                                                      //
//               Ramified Primes and Discriminant Sequences             //
//                              by David Kohel                          //
//                                                                      //
//////////////////////////////////////////////////////////////////////////


intrinsic RamifiedPrimes(a::RngIntElt,b::RngIntElt :
    MakeSquareFree := true) -> SeqEnum
    {}
    if MakeSquareFree then
        a := SquareFree(a); b := SquareFree(b);
    end if;
    vprintf Quaternion : "RamifiedPrimes(%o,%o):\n", a, b;
    if &or [ IsSquare(x) : x in {a,b,a+b} ] then
        return [ Integers() | ];
    end if;
    c := GCD(a,b);
    if c ne 1 then
        p := Factorization(c)[1][1];
        vprint Quaternion : "";
        prms := RamifiedPrimes(p,-(b div p))
            cat RamifiedPrimes(a div p,b);
        return Sort( [ p : p in { x : x in prms } |
            (#[ i : i in [1..#prms] | prms[i] eq p ] mod 2) eq 1 ] );
    end if;
    prms := [ Integers() | ];
    S1 := PrimeDivisors(a);
    vprint Quaternion : "  S1 =", S1;
    for p in S1 do
        vprint Quaternion : "  p =", p;
        if p eq 2 and (b mod 4) eq 3 then
            vprintf Quaternion :
                "    Case p = 2 and b = %o == 3 mod 4\n", b;
            if KroneckerSymbol(a+b,p) eq -1 then
                vprintf Quaternion :
                    "    (a+b,p) = (%o,%o) = -1", a+b, p;
                vprintf Quaternion : " appending p = %o\n", p;
                Append(~prms,p);
            end if;
        elif KroneckerSymbol(b,p) eq -1 then
```

```
            vprintf Quaternion :
                "    Case p ne 2 or b = %o == 1 mod 4\n", b;
            vprintf Quaternion : "      (b,p) = (%o,%o) = -1", b, p;
            vprintf Quaternion : " appending p = %o\n", p;
    Append(~prms,p);
end if;
    end for;
    S2 := PrimeDivisors(b);
    vprint Quaternion : "  S2 =", S2;
    for q in S2 do
vprint Quaternion : "  q =", q;
if q eq 2 and (a mod 4) eq 3 then
    vprintf Quaternion :
"    Case q = 2 and a = %o == 3 mod 4\n", a;
    if KroneckerSymbol(a+b,q) eq -1 then
vprintf Quaternion :
    "      (a+b,q) = (%o,%o) = -1", a+b, q;
vprintf Quaternion : " appending q = %o\n", q;
Append(~prms,q);
    end if;
elif KroneckerSymbol(a,q) eq -1 then
    vprintf Quaternion :
"    Case q ne 2 or a = %o == 1 mod 4\n", a;
    vprintf Quaternion : "      (a,q) = (%o,%o) = -1", a, q;
    vprintf Quaternion : " appending q = %o\n", q;
    Append(~prms,q);
end if;
    end for;
    if 2 notin prms and (a mod 4) eq 3 and (b mod 4) eq 3 then
vprint Quaternion : "  r = 2";
vprintf Quaternion : "    Case a = %o == 3 mod 4", a;
vprintf Quaternion : "       and b = %o == 3 mod 4;", b;
vprintf Quaternion : " appending p = 2\n";
Append(~prms,2);
    end if;
    vprint Quaternion : "  Returning ", prms;
    vprint Quaternion : "";
    return Sort(prms);
end intrinsic;
////////////////////////////////////////////////////////////////////////
//                    Discriminant Functionality                      //
////////////////////////////////////////////////////////////////////////
```

```
intrinsic DiscriminantSequence(O::AlgQuatOrd) -> SeqEnum
    {The sequence of 2x2 minors of the norm form on O with respect
    to a reduced basis for O.}
    require Type(BaseRing(O)) eq RngInt :
"Argument must be an order over the integers.";
    if IsDefinite(QuaternionAlgebra(O)) then
M := ReducedGramMatrix(O);
    else
M := GramMatrix(O);
    end if;
    S := [ [2,2], [3,3], [4,4], [2,3], [2,4], [3,4] ];
    return [ (MatrixMinor(M,[1,X[1]],[1,X[2]])) : X in S ];
    return [ Determinant(MatrixMinor(M,[1,X[1]],[1,X[2]])) : X in S ];
end intrinsic;


intrinsic DiscriminantSequence(B::[AlgQuatOrdElt]) -> SeqEnum
    {The sequence of 2x2 minors of the norm form on the sequence
    of elements of B.}
    M := Matrix(4,[ Norm(x+y)-Norm(x)-Norm(y) : x, y in B ]);
    S := [ [2,2], [3,3], [4,4], [2,3], [2,4], [3,4] ];
    return [ Determinant(MatrixMinor(M,[1,X[1]],[1,X[2]])) : X in S ];
end intrinsic;
//////////////////////////////////////////////////////////////////////////
//                Constructor from Discriminant Data                    //
//////////////////////////////////////////////////////////////////////////


intrinsic QuaternionOrderWithGram(M::AlgMatElt) -> AlgQuatOrd
    {A quaternion order in an algebra over Q with norm Gram matrix M.}
    require BaseRing(Parent(M)) cmpeq Integers() :
"Argument must be a matrix over the integers.";
    D1 := Determinant(MatrixMinor(M,[1,2],[1,2]));
    D2 := Determinant(MatrixMinor(M,[1,3],[1,3]));
    D3 := Determinant(MatrixMinor(M,[1,4],[1,4]));
    T1 := Determinant(MatrixMinor(M,[1,2],[1,3]));
    t1 := M[1,2]; t2 := M[1,3]; t3 := M[1,4];
    K := QuaternionAlgebra(-D1,-D2,-T1);
    B := Basis(K);
    B[2] +:= K!(t1-(D1 mod 2))/2;
    B[3] +:= K!(t2-(D2 mod 2))/2;
    M0 := Matrix(4,[Norm(x+y)-Norm(x)-Norm(y) : x, y in B ]);
    MQ := MatrixAlgebra(RationalField(),4)!M;
    w0 := Vector([ MQ[1,4], MQ[2,4], MQ[3,4] ]);
    v0, V0 := Solution(Submatrix(M0,1,1,4,3),w0);
    v1 := Basis(V0)[1];
    a := InnerProduct(v1*M0,v1);
    b := InnerProduct(v1*M0,v0);
    c := InnerProduct(v0*M0,v0);
    f0 := a*X^2 + 2*b*X + c - M[4,4]
where X := PolynomialRing(RationalField()).1;
    if f0 ne 0 then
v0 +:= Roots(f0)[1][1]*v1;
    end if;
    B[4] := &+[ v0[i]*B[i] : i in [1..4] ];
    return QuaternionOrder(B);
end intrinsic;
```

representations.m

```
freeze;


intrinsic IsSplittingField(A::AlgQuat,K::FldQuad) -> BoolElt
    {Returns true if and only if K is a splitting field for A.}
    require Type(BaseField(A)) eq FldRat :
        "Argument 1 must be defined over the rationals.";
    D := Discriminant(K);
    return &and[ KroneckerSymbol(D,p) ne 1 : p in RamifiedPrimes(A) ];
end intrinsic;
/*


intrinsic MatrixRepresentation(R::AlgQuatOrd,K::FldQuad) -> BoolElt
    {}
    A := QuaternionAlgebra(R);
    require IsSplittingField(A,K) :
"Argument 2 must be a splitting field for argument 1.";
    M := GramMatrix(R);
    P3<x,y,z> := PolynomialRing(K,3);
    return &+[ M[i,j]*P4.i*P4.j : i, j in [1..4] ]/2;
end intrinsic;
*/


intrinsic MatrixRepresentation(A::AlgQuat) -> BoolElt
    {Returns a 2-dimensional matrix representation over the splitting
    field generated by the first non-central basis element.}
    require Type(BaseField(A)) eq FldRat :
       "Argument must be defined over the rationals.";
    f := CharacteristicPolynomial(A.1);
    K := NumberField(f);
    MatK := MatrixAlgebra(K,2);
    i := A.1; j := A.2; k := A.3;
    B := [ 1, i, j, i*j ];
    t1 := Trace(i)/2;
    t2 := Trace(j)/2; n2 := t2^2-Norm(j);
    M0 := MatK!1;
    M1 := MatK![ K.1, 0, 0, t1-K.1 ];
    M2 := MatK![ t2, 1, n2, t2 ];
    M := Matrix(4,4,&cat[ Eltseq(x) : x in B ]);
    c := Eltseq(A.3);
    M3 := c[1]*M0 + c[2]*M1 + (c[3] + c[4]*M1)*M2;
    MatGens := [ M0, M1, M2, M3 ];
    return hom< A -> MatK | x :->
        &+[ c[i]*MatGens[i] : i in [1..4] ]
            where c := Eltseq(x) >;
end intrinsic;


intrinsic MatrixRepresentation(R::AlgQuatOrd) -> BoolElt
    {Returns a 2-dimensional matrix representation over the
    splitting field generated by the first non-central basis
    element of the quaternion algebra containing R.}
    A := QuaternionAlgebra(R);
    require Type(BaseField(A)) eq FldRat :
       "Argument must be defined over the integers.";
    f := MatrixRepresentation(A);
    MatK := Codomain(f);
    B := Basis(R);
    MatGens := [ f(A!x) : x in B ];
    return hom< R -> MatK | x :->
        &+[ c[i]*MatGens[i] : i in [1..4] ]
            where c := Eltseq(x) >;
end intrinsic;
```

verbose.m

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                         Verbose declarations                       //
//                                                                    //
////////////////////////////////////////////////////////////////////////
declare verbose QuaternionIsomorphism, 2;
```