BRANDT MODULES

David R. Kohel

BRANDT MODULES

BRANDT MODULES

David R. Kohel

§1 Introduction

Brandt modules provide a representation in terms of quaternion ideals of certain cohomology subgroups associated to Shimura curves $X_0^D(N)$ which generalize the classical modular curves $X_0(N)$. The Brandt module datatype is that of a Hecke module – a free module of finite rank with the action of a ring of Hecke operators – which is equipped with a canonical basis (identified with left quaternion ideal classes) and an inner product which is adjoint with respect to the Hecke operators. The machinery of modular symbols, Brandt modules, and, in a future release, a module of singular elliptic curves, form the computational machinery underlying modular forms in MAGMA.

Brandt modules were implemented by David Kohel, motivated by the article of Mestre and Oesterlé [Mes86] on the method of graphs for supersingular elliptic curves, the article of Pizer [Piz80] on computing spaces of modular forms using quaternion arithmetic, and grew out of research in the author's thesis [Koh96] on endomorphism ring structure of elliptic curves over finite fields. The Brandt module machinery is described in the article [Koh01] and has been used, together with modular symbols, in the computation of component groups of quotients of the Jacobians $J_0(N)$ of classical modular curves [KS00].

§2 Brandt Module Creation

We first describe the various constructors for Brandt modules and their elements.

`BrandtModule(D)`

`BrandtModule(D,m)`

`ComputeGrams` BOOLELT *Default :* `true`

Given a product D of an odd number of primes, and a integer m which has valuation at most one at each prime divisor p of D, return a Brandt module of level (D, m) over the integers. If not specified, then the conductor m is taken to be 1.

The parameter `ComputeGrams` can be set to `false` in order to not compute the $h \times h$ array, where h is the left class number of level (D, m), of reduced Gram matrices of the quaternion ideal norm forms. Instead the basis of quaternion ideals is stored and the collection of degree p ideal homomorphisms is then computed in order to find the Hecke operator T_p for each prime p.

For very large levels, setting `ComputeGrams` to `false` is more space efficient. For moderate sized levels for which one wants to compute many Hecke operators, it is preferable to compute the Gram matrices and determine the Hecke operators using theta series.

BrandtModule(A)

BrandtModule(A,R)

ComputeGrams BoolElt *Default :* `true`

Given a definite order A in a quaternion algebra over $\mathbf{Q}$, returns the Brandt module on the left ideals classes for A, as a module over R. If not specified, the ring R is taken to be the integers. The parameter `ComputeGrams` is as previously described.

BaseExtend(M,R)

Forms the Brandt module with coefficient ring base extended to R.

Example E1

In the following example we create the Brandt module of level 101 over the field of 7 elements and decompose it into its invariant subspaces.

```
> A := QuaternionOrder(101);
> FF := FiniteField(7);
> M := BrandtModule(A,FF);
> Decomposition(M,13);
[
    Brandt module of level (101,1), dimension 1, and degree 9 over Finite
    field of size 7,
    Brandt module of level (101,1), dimension 1, and degree 9 over Finite
    field of size 7,
    Brandt module of level (101,1), dimension 1, and degree 9 over Finite
    field of size 7,
    Brandt module of level (101,1), dimension 6, and degree 9 over Finite
    field of size 7
]
```

We note that Brandt modules of non-prime discriminant can be useful for studying isogeny factors of modular curves, since it is possible to describe exactly the piece of cohomology of interest, without first computing a much larger space. In this example we see that the space of weight 2 cusp forms for $\Gamma_0(1491)$, where $1491 = 3 \cdot 7 \cdot 71$, is of dimension 189 (plus an Eisenstein space of dimension 7), while the newspace has dimension 71. The Jacobian of the Shimura curve $X_0^{1491}(1)$ is isogenous to the new factor of $J_0(1491)$, so that we can study the newspace directly via the Brandt module.

```
> DimensionCuspFormsGamma0(7*71*3,2);
189
> DimensionNewCuspFormsGamma0(7*71*3,2);
71
> BrandtModuleDimension(7*71*3,1);
72
> M := BrandtModule(3*7*71 : ComputeGrams := false);
```

```
> S := CuspidalSubspace(M);
> Dimension(S);
71
> [ Dimension(N) : N in Decomposition(S,13 : Sort := true) ];
[ 6, 6, 6, 6, 11, 12, 12, 12 ]
```

In this example by setting `ComputeGrams` equal to `false` we obtain the Brandt module much faster, but the decomposition is much more expensive. For most applications the default computation of Gram matrices is preferable.

---

§2.1 Creation of Elements

`M ! x`

Given a sequence or module element x compatible with M, forms the corresponding element in M.

`M . i`

For a Brandt module M and integer i, returns the i-th basis element.

§2.2 Operations on Elements

Brandt module elements support standard operations

`a * x`

The scalar multiplication of a Brandt module element x by an element a in the base ring.

`x * T`

Given a Brandt module element x and an element T of the algebra of Hecke operators of degree compatible with the parent of x or of its ambient module, returns the image of x under T.

`x + y`

Returns the sum of two Brandt module elements.

`x - y`

Returns the difference of two Brandt module elements.

`x eq y`

Returns `true` if x and y are equal elements of the same Brandt module.

`Eltseq(x)`

Returns the sequence of coefficients of the Brandt module element x.

`InnerProduct(x,y)`

Returns the inner product of x and y with respect to the canonical pairing on their common parent.

`Norm(x)`

Returns the inner product of x with itself.

§2.3 Categories and Parent

Brandt modules belong to the category `ModBrdt`, with elements of type `ModBrdtElt`, involved in the type checking of arguments in MAGMA programming. The `Parent` of an element is the space to which it belongs.

`Category(M)`

`Category(M)`

`Type(M)`

`Type(M)`

The category, `ModBrdt` or `ModBrdtElt`, of a Brandt module or Brandt module element.

`Parent(x)`

The parent module M of a Brandt module element x.

`x in M`

Returns `true` if M is the parent of x.

§2.4 Elementary Invariants

Here we describe the elementary invariants of the Brandt module, defined with respect to a definite quaternion order A in a quaternion algebra $\mathbf{H}$ over $\mathbf{Q}$. The *level* of M is defined to be the reduced discriminant of A, the *discriminant* is defined to be the discriminant of the algebra $\mathbf{H}$, and the *conductor* to be the index of A in any maximal order of $\mathbf{H}$ which contains it. We note that the discriminant of M is just the product of the ramified primes of $\mathbf{H}$, and the product of the conductor and discriminant of M is the reduced discriminant of A.

`Level(M)`

Returns the level of the Brandt module, which is the product of the discriminant and the conductor, and equal to the reduced discriminant of its defining quaternion order.

`Discriminant(M)`

Returns the discriminant of the quaternion algebra $\mathbf{H}$ with respect to which M is defined (equal to the product of the primes which ramify in $\mathbf{H}$).

`Conductor(M)`

Returns the conductor or index of the defining quaternion order of M in a maximal order of its quaternion algebra.

`BaseRing(M)`

The ring over which M is defined.

`Basis(M)`

Returns the basis of M.

§2.5 Associated Structures

The following give structures associated to Brandt modules. In particular we note the definition of the `AmbientModule`, which is the full module containing a given Brandt module whose basis corresponds to the left quaternion ideals. Elements of every submodule of the ambient module are displayed with respect to the basis of the ambient module.

`AmbientModule(M)`

The full module of level (D, m) containing a given module of this level.

`IsAmbient(M)`

Returns `true` if and only if M is its own ambient module.

`Dimension(M)`

`Rank(M)`

Returns the rank of M over its base ring.

`Degree(M)`

Returns the degree of the Brandt module M, defined to be the dimension of its ambient module.

`GramMatrix(M)`

The matrix $(\langle u_i, u_j \rangle)$ defined with respect to the basis $\{u_i\}$ of M.

`InnerProductMatrix(M)`

Returns the Gram matrix of the ambient module of M.

Example E2

The following example demonstrates the use of `AmbientModule` to get back to the original Brandt module.

```
> M := BrandtModule(3,17);
> S := CuspidalSubspace(M);
> M eq AmbientModule(M);
true
```

§2.6 Verbose Output

The verbose level for Brandt modules is set with the command `SetVerbose("Brandt",n)`. Since the construction of a Brandt module requires intensive quaternion algebra machinery for ideal enumeration, the `Quaternion` verbose flag is also relevant. In both cases, the value of n can be 0 (silent), 1 (verbose), or 2 (very verbose).

Example E3

In the following example we show the verbose output from the quaternion ideal enumeration in the creation of the Brandt module of level $(37, 1)$.

```
> SetVerbose("Quaternion",2);
> BrandtModule(37);
Ideal number 1, right order module
Full RSpace of degree 4 over Integer Ring
Inner Product Matrix:
[ 2  0  1  1]
[ 0  4 -1  2]
[ 1 -1 10  0]
[ 1  2  0 20]
Frontier at 2-depth 1 has 3 elements.
Number of ideals = 1
```

```
Ideal number 2, new right order module
Full RSpace of degree 4 over Integer Ring
Inner Product Matrix:
[ 2 -1  0  1]
[-1  8 -1 -4]
[ 0 -1 10 -2]
[ 1 -4 -2 12]
Ideal number 3, new right order module
Full RSpace of degree 4 over Integer Ring
Inner Product Matrix:
[ 2  1  0 -1]
[ 1  8  1  3]
[ 0  1 10 -2]
[-1  3 -2 12]
Frontier at 2-depth 2 has 4 elements.
Number of ideals = 3
Brandt module of level (37,1), dimension 3, and degree 3 over Integer Ring
```

---

§3 Subspaces and Decomposition

`EisensteinSubspace(M)`

Returns the Eisenstein subspace of the Brandt module M. When the level of M is square-free this will be the submodule generated by a vector of the form $(w/w_1, \ldots, w/w_n)$, if it exists in M, where w_i is the number of automorphisms of the i-th basis ideal and $w = \mathrm{LCM}(\{w_i\})$.

`CuspidalSubspace(M)`

Returns the cuspidal subspace, defined to be the orthogonal complement of the Eisenstein subspace of M. If the discriminant of M is coprime to the conductor, then the cuspidal subspace consists of the vectors in M of the form $(a_1, \ldots, a_n)$, where $\sum_i a_i = 0$.

`OrthogonalComplement(M)`

The Brandt module orthogonal to the given module M in the ambient module of M.

`M meet N`

Returns the intersection of the Brandt modules M and N.

Decomposition(M,B)

Sort BoolElt *Default :* `true`

Returns a decomposition of the Brandt module with respect to the Atkin–Lehner operators and Hecke operators up to the bound B. The parameter `Sort` can be set to `true` to return a sequence sorted under the operator `lt` as defined below.

SortDecomposition(D)

Sort the sequence D of spaces of Brandt modules with respect to the `lt` comparison operator.

Example E4

```
> M := BrandtModule(2*3*17);
> Decomp := Decomposition(M,11 : Sort := true);
> Decomp;
[
    Brandt module of level (102,1), dimension 1, and degree 4 over Integer Ring,
    Brandt module of level (102,1), dimension 1, and degree 4 over Integer Ring,
    Brandt module of level (102,1), dimension 1, and degree 4 over Integer Ring,
    Brandt module of level (102,1), dimension 1, and degree 4 over Integer Ring
]
> [ IsEisenstein(N) : N in Decomp ];
[ true, false, false, false ]
```

§3.1 Boolean Tests on Subspaces

IsEisenstein(M)

Returns `true` if and only if the space M is contained in the Eisenstein subspace of the ambient module.

IsCuspidal(M)

Returns `true` if and only if the space M is contained in the cuspidal subspace of the ambient module.

IsIndecomposable(M,B)

Returns `true` if an only if the Brandt module M does not decompose into complementary Hecke-invariant submodules under the Atkin-Lehner operators, nor under the Hecke operators T_n, for $n \leq B$.

M1 subset M2

Returns `true` if and only if $M1$ is contained in the module $M2$.

M1 lt M2

`Bound` RngIntElt *Default :* 101

Given two indecomposable subspaces, M_1 and M_2, returns `true` if and only if $M_1 < M_2$ under the following ordering:

(1) Order by dimension, with smaller dimension being less.

(2) An Eisenstein subspace is less than a cuspidal subspace of the same dimension.

(3) Order by Atkin–Lehner eigenvalues, starting with *smallest* prime dividing the level and with '+' being less than '–'.

(4) Order by $|\mathrm{Tr}(T_{p^i}(M_j))|$, p not dividing the level, and $1 \leq i \leq g$, where g is `Dimension`(M_1), with the positive one being smaller in the event of equality.

Condition (4) differs from the similar one for modular symbols, but permits the comparison of arbitrary Brandt modules. The algorithm returns `false` if all primes up to value of the parameter `Bound` fail to differentiate the arguments.

M1 gt M2

`Bound` RngIntElt *Default :* 101

Returns the complement of `lt`.

Example E5

```
> M := BrandtModule(7,7);
> E := EisensteinSubspace(M);
> Basis(E);
[
    (1 1 0 0),
    (0 0 1 1)
]
> S := CuspidalSubspace(M);
> Basis(S);
[
    ( 1 -1  0  0),
    ( 0  0  1 -1)
```

```
]
> PS<q> := LaurentSeriesRing(RationalField());
> qExpansionBasis(S,100);
[
    q + q^2 - q^4 - 3*q^8 - 3*q^9 + 4*q^11 - q^16 - 3*q^18 + 4*q^22 + 8*q^23
        - 5*q^25 + 2*q^29 + 5*q^32 + 3*q^36 - 6*q^37 - 12*q^43 - 4*q^44 +
        8*q^46 - 5*q^50 - 10*q^53 + 2*q^58 + 7*q^64 + 4*q^67 + 16*q^71 +
        9*q^72 - 6*q^74 + 8*q^79 + 9*q^81 - 12*q^86 - 12*q^88 - 8*q^92 -
        12*q^99 + 5*q^100 + O(q^101)
]
```

---

§4 Hecke Operators

A Brandt module M is equipped with a family of linear Hecke operators which act on it. These are returned as matrices, which act on the right with respect to the basis `Basis(M)`, but the Hecke operators of the ambient module may also be applied to elements of submodules. The system of Hecke operators are computed by default using the theta series which define the classical Brandt matrices. If a module is created with the `ComputeGram` parameter set to false, the Hecke operators are determined by means of enumeration of ideals in a p-neighbouring operation analogous to the method of graphs approach of Mestre and Oesterlé [Mes86].

`HeckeOperator(M, n)`

Compute a matrix representing the nth Hecke operator T_n with respect to `Basis(M)`.

`AtkinLehnerOperator(M, p)`

Computes the Atkin–Lehner operator on M, where p is a prime dividing the discriminant of M.

§5 q-Expansions

We can associate a theta series to any pair of elements of a Brandt module, which give embeddings (with respect to any fixed module element) of the Brandt module in a space of weight 2 modular forms.

`ThetaSeries(x, y, prec)`

Returns the theta series associated to the pair (x, y), as an element of a power series ring.

`qExpansionBasis(M, prec)`

A sequence of power series elements, to precision `prec`, spanning the image of the theta functions associated to pairs in M.

§6 Dimensions of Spaces

`BrandtModuleDimension(D,N)`

The dimension of the Brandt module of level (D, N), computed by means of standard formulas.

Example E6

In the following example we demonstrate the computation of the dimensions of the Brandt modules for the sequence of Eichler orders of index 3^i in a maximal order in the quaternion algebra of discriminant $2 \cdot 3 \cdot 7$.

```
> D := 2*5*7;
> for i in [0..10] do
>     BrandtModuleDimension(D,3^i);
> end for;
2
8
24
72
216
648
1944
5832
17496
52488
157464
```

§7 Bibliography

[**Bos00**] Wieb Bosma, editor. *ANTS IV*, volume 1838 of *LNCS*. Springer-Verlag, 2000.

[**Koh96**] D. Kohel. *Endomorphism rings of elliptic curves over finite fields*. PhD thesis, University of California, Berkeley, 1996.

[**Koh01**] D. Kohel. Hecke module structure of quaternions. In K. Miyake, editor, *Class Field Theory – its Centenary and Prospect*, 2001.

[**KS00**] D. Kohel and W. Stein. Component groups of quotients of $J_0(N)$. In Bosma [Bos00].

[**Mes86**] J.-F. Mestre. Sur la méthode des graphes, Exemples et applications. In *Proceedings of the international conference on class numbers and fundamental units of algebraic number fields*, pages 217–242. Nagoya University, 1986.

[**Piz80**] A. Pizer. An Algorithm for Computing Modular Forms on $\Gamma_0(N)$. *Journal of Algebra*, 64:340–390, 1980.

A1 ModBrdt Package

brandt_modules.m

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                    Brandt Modules of Quaternions                   //
//                            David Kohel                             //
//                                                                    //
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                         Creation functions                         //
//                                                                    //
////////////////////////////////////////////////////////////////////////


intrinsic BrandtModule(D::RngIntElt : ComputeGrams := true) -> ModBrdt
    {The Eichler-Brandt quaternion ideal divisor group.}
    prms := PrimeDivisors(D);
    require &*prms eq D and #prms mod 2 eq 1 :
        "Argument must be a product of an odd number of primes.";
    A := QuaternionOrder(D);
    return BrandtModule(A,Integers() : ComputeGrams := ComputeGrams);
end intrinsic;


intrinsic BrandtModule(D::RngIntElt,m::RngIntElt :
    ComputeGrams := true) -> ModBrdt
    {The Eichler-Brandt quaternion ideal divisor group.}
    prms := PrimeDivisors(D);
    require Max([ Valuation(D,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.";
        A := QuaternionOrder(D,m);
    return BrandtModule(A,Integers() : ComputeGrams := ComputeGrams);
end intrinsic;
function NormGramsOrder(left_ideals);
    auto_nums := [ Integers() | ];
    norm_grams := [ MatrixAlgebra(Integers(),4) | ];
    k := 1;
    h := #left_ideals;
```

```
    vprint Brandt : "Computing row entries...";
    for i in [1..h] do
        norm_grams[k] := ReducedGramMatrix(RightOrder(left_ideals[i]) );
        auto_nums[i] :=
            2*#ShortestVectors(LatticeWithGram(norm_grams[k]));
        RI := Conjugate(left_ideals[i]);
        k +:= 1;
        for j in [(i+1)..h] do
            J := RI*left_ideals[j];
            norm_grams[k] := Norm(J)^-1 * ReducedGramMatrix(J);
            k +:= 1;
        end for;
        if h ge 10 and i lt 10 then vprintf Brandt : " "; end if;
        if h ge 100 and i lt 100 then vprintf Brandt : " "; end if;
        if h ge 1000 and i lt 1000 then vprintf Brandt : " "; end if;
        vprintf Brandt : "%o ", i;
        if i mod 16 eq 0 then vprint Brandt : ""; end if;
    end for;
    if h mod 16 ne 0 then vprint Brandt, 2 : ""; end if;
    return norm_grams, auto_nums;
end function;


intrinsic BrandtModule(A::AlgQuatOrd : ComputeGrams := true) -> ModBrdt
    {The Eichler-Brandt quaternion ideal divisor group.}
    return BrandtModule(A,Integers() : ComputeGrams := ComputeGrams);
end intrinsic;


intrinsic BrandtModule(A::AlgQuatOrd,R::Rng : ComputeGrams := true)
    -> ModBrdt
    {The Eichler-Brandt quaternion ideal divisor group.}
    M := HackobjCreateRaw(ModBrdt);
    M`IsFull := true;
    M`RamifiedPrimes := RamifiedPrimes(QuaternionAlgebra(A));
    M`Conductor := Level(A);
    M`BaseRing := R;
    M`LeftIdeals := LeftIdealClasses(A);
    if not ComputeGrams then
        M`AutoNums := [ #Units(RightOrder(I)) : I in M`LeftIdeals ];
    else
        M`NormGrams, M`AutoNums := NormGramsOrder(M`LeftIdeals);
    end if;
```

```
    M‘HeckePrecision := 1;
    M‘ThetaSeries := [ PowerSeriesRing(R) | ];
    M‘HeckePrimes := [ Integers() | ];
    h := #M‘AutoNums;
    MatR := MatrixAlgebra(R,h);
    M‘HeckeOperators := [ MatR | ];
    M‘Module := RSpace(R,h,
        DiagonalMatrix(MatR,[ w div 2 : w in M‘AutoNums ]));
    return M;
end intrinsic;
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                          Coercions                                 //
//                                                                    //
////////////////////////////////////////////////////////////////////////
function init_bool_ModBrdtElt(M,v)
    val, v := IsCoercible(M‘Module,Eltseq(v));
    if not val then
        x := "Invalid coercion";
    else
        x := HackobjCreateRaw(ModBrdtElt);
        x‘Parent := M;
        x‘Element := v;
    end if;
    return val, x;
end function;


intrinsic HackobjCoerceModBrdt(M::ModBrdt,s::.) -> BoolElt, ModBrdtElt
    {}
    if Type(s) eq ModBrdtElt then
        N := Parent(s);
        if M eq N then
            return true, s;
        end if;
        A := AmbientModule(M);
        if A eq AmbientModule(N) then
            v := A‘Module!s‘Element;
            if v in M‘Module then
                return init_bool_ModBrdtElt(M,s);
            end if;
        end if;
    elif Type(s) eq ElementType(M‘Module) then
        if s in M‘Module then
```

```
            return init_bool_ModBrdtElt(M,s);
        end if;
    elif Type(s) eq SeqEnum and #s eq Degree(M) then
        R := Universe(s);
        if Type(R) eq RngInt or R eq BaseRing(M) then
            return init_bool_ModBrdtElt(M,s);
        end if;
    end if;
    return false, "Invalid coercion";
end intrinsic;
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                           Printing                                 //
//                                                                    //
////////////////////////////////////////////////////////////////////////


intrinsic HackobjPrintModBrdt(M::ModBrdt, level::MonStgElt)
    {}
    D := Discriminant(M);
    m := Conductor(M);
    printf "Brandt module of level (%o,%o), " *
        "dimension %o, and degree %o over %o",
        D, m, Dimension(M), Degree(M), BaseRing(M);
end intrinsic;


intrinsic HackobjPrintModBrdtElt(x::ModBrdtElt, level::MonStgElt)
    {}
    printf "%o", x`Element;
end intrinsic;


intrinsic '.'(M::ModBrdt,i::RngIntElt) -> ModBrdtElt
    {}
    require i in [1..Dimension(M)] :
        "Argument 2 should be in the range " *
        "[1.." * IntegerToString(Dimension(M)) * "]";
    return M!(M`Module).i;
end intrinsic;
////////////////////////////////////////////////////////////////////////
//                                                                    //
//            Membership, Parent, and Equality Relations              //
```

```
//                                                                          //
//////////////////////////////////////////////////////////////////////////////


intrinsic HackobjInModBrdt(x::., M::ModBrdt) -> BoolElt
    {Returns true if x is in X.}
    if Type(x) eq ModBrdtElt then
        if Parent(x) eq M then
            return true;
        elif AmbientModule(Parent(x)) eq AmbientModule(M) then
            return IsCoercible(M`Module,Eltseq(x));
        end if;
    end if;
    return false;
end intrinsic;


intrinsic HackobjParentModBrdtElt(x::ModBrdtElt) -> ModBrdt
    {}
    return x`Parent;
end intrinsic;


intrinsic 'eq' (X::ModBrdt,Y::ModBrdt) -> BoolElt
    {}
    if IsIdentical(X,Y) then
        return true;
    elif Degree(X) ne Degree(Y) or
        Dimension(X) ne Dimension(Y) or
        not IsIdentical(AmbientModule(X),AmbientModule(Y)) then
        return false;
    end if;
    dim := Dimension(X);
    M := AmbientModule(X)`Module;
    return sub< M | [ BX[i] : i in [1..dim] ] >
        eq sub< M | [ BY[i] : i in [1..dim] ] >
        where BX := BasisMatrix(X) where BY := BasisMatrix(Y);
end intrinsic;


intrinsic 'eq' (x::ModBrdtElt,y::ModBrdtElt) -> BoolElt
    {}
    return x`Parent eq y`Parent and x`Element eq y`Element;
end intrinsic;
```

access.m

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                     Attribute Access Functions                     //
//                            David Kohel                             //
//                                                                    //
////////////////////////////////////////////////////////////////////////


intrinsic AtkinLehnerPrimes(M::ModBrdt) -> SeqEnum
    {The sequence of primes for which the Atkin-Lehner operator
    can be computed.}
    if not assigned M`AtkinLehnerPrimes then
        N := Level(M);
        D := Discriminant(M);
        prms := PrimeDivisors(N);
        admit_prms :=
            [ p : p in prms | D mod p ne 0 or Valuation(N,p) le 2 ];
        M`AtkinLehnerPrimes := admit_prms;
    end if;
    return M`AtkinLehnerPrimes;
end intrinsic;
////////////////////////////////////////////////////////////////////////
//                        Ambient Module                              //
//       Only certain data will be cached on the ambient module,      //
//       and propagate to the submodules.                             //
////////////////////////////////////////////////////////////////////////


intrinsic AmbientModule(M::ModBrdt) -> ModBrdt
    {}
    if assigned M`AmbientModule then
        return M`AmbientModule;
    end if;
    return M;
end intrinsic;
////////////////////////////////////////////////////////////////////////
//                 Booleans for Ambient Determination                 //
////////////////////////////////////////////////////////////////////////


intrinsic IsFull(M::ModBrdt) -> BoolElt
    {}
```

```
    return M`IsFull;
end intrinsic;


intrinsic IsAmbient(M::ModBrdt) -> BoolElt
    {True if only M is equal to AmbientModule(M).}
    return M`IsFull;
end intrinsic;
////////////////////////////////////////////////////////////////////////
//                      Degree and Dimension                          //
////////////////////////////////////////////////////////////////////////


intrinsic Degree(M::ModBrdt) -> RngIntElt
    {}
    return Dimension(AmbientModule(M));
end intrinsic;


intrinsic Dimension(M::ModBrdt) -> RngIntElt
    {}
    return Dimension(M`Module);
end intrinsic;
////////////////////////////////////////////////////////////////////////
//                  Inner Product Module Structure                    //
////////////////////////////////////////////////////////////////////////


intrinsic Norm(x::ModBrdtElt) -> RngElt
    {}
    return Norm(x`Element);
end intrinsic;


intrinsic GramMatrix(M::ModBrdt) -> AlgMatElt
    {}
    return GramMatrix(M`Module);
end intrinsic;


intrinsic InnerProductMatrix(M::ModBrdt) -> AlgMatElt
    {}
    return InnerProductMatrix(M`Module);
```

```
end intrinsic;


intrinsic InnerProduct(x::ModBrdtElt,y::ModBrdtElt) -> RngElt
    {}
    return InnerProduct(x`Element,y`Element);
end intrinsic;
////////////////////////////////////////////////////////////////////////
//                  Inner Product Module Structure                    //
////////////////////////////////////////////////////////////////////////


intrinsic Conductor(M::ModBrdt) -> RngIntElt
    {}
    return M`Conductor;
end intrinsic;


intrinsic Discriminant(M::ModBrdt) -> RngIntElt
    {}
    return &*M`RamifiedPrimes;
end intrinsic;


intrinsic Level(M::ModBrdt) -> RngIntElt
    {}
    return Discriminant(M)*Conductor(M);
end intrinsic;
////////////////////////////////////////////////////////////////////////
//                       Base Ring and Basis                          //
////////////////////////////////////////////////////////////////////////


intrinsic BaseRing(M::ModBrdt) -> Rng
    {}
    return M`BaseRing;
end intrinsic;


intrinsic Basis(M::ModBrdt) -> SeqEnum
    {A basis for M.}
    return [ M!x : x in Basis(M`Module) ];
end intrinsic;
```

```
intrinsic BaseExtend(M::ModBrdt,R::Rng) -> ModBrdt
    {}
    N := HackobjCreateRaw(ModBrdt);
    N`IsFull := M`IsFull;
    N`BaseRing := R;
    N`Conductor := M`Conductor;
    N`RamifiedPrimes := M`RamifiedPrimes;
    N`Module := BaseExtend(M`Module,R);
    if M`IsFull then
        N`AutoNums := M`AutoNums;
        N`NormGrams := M`NormGrams;
        N`ThetaSeries := [ PowerSeriesRing(R) | f : f in M`ThetaSeries ];
        N`HeckePrimes := M`HeckePrimes;
        N`HeckeOperators := [
            MatrixAlgebra(R,Dimension(M))!T : T in M`HeckeOperators ];
    else
        N`AmbientModule := BaseExtend(AmbientModule(M),R);
    end if;
    return N;
end intrinsic;
////////////////////////////////////////////////////////////////////////
//                        Coordinate Access                           //
////////////////////////////////////////////////////////////////////////


intrinsic Eltseq(x::ModBrdtElt) -> SeqEnum
    {}
    return Eltseq(x`Element);
end intrinsic;
```

arithmetic.m

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                    Arithmetic operations, etc.                     //
//                                                                    //
////////////////////////////////////////////////////////////////////////
function init_ModBrdtElt(M,v)
    // _, v := IsCoercible(Representation(M),v);
    x := HackobjCreateRaw(ElementType(M));
    x`Parent := M;
    x`Element := v;
    return x;
end function;


intrinsic '*'(a::RngElt,x::ModBrdtElt) -> ModBrdtElt
    {}
    M := Parent(x);
    require Type(Parent(a)) eq RngInt or
        Parent(a) cmpeq BaseRing(M) : "Elements have different parents.";
    z := HackobjCreateRaw(Type(x));
    z`Parent := M;
    z`Element := a*x`Element;
    return z;
end intrinsic;


intrinsic '*'(x::ModBrdtElt,T::AlgMatElt) -> ModBrdtElt
    {}
    M := Parent(x);
    require Type(BaseRing(Parent(T))) eq RngInt or
        BaseRing(Parent(T)) eq BaseRing(M) :
        "Arguments have different coefficient rings.";
    // Assume that internal representation is via ambient module.
    if Degree(Parent(T)) eq Degree(M) then
        return init_ModBrdtElt(M,x`Element * T);
    elif Degree(Parent(T)) eq Dimension(M) then
        U := M`Module;
        B := BasisMatrix(M);
        y := ( Vector(Coordinates(U,x`Element)) * T ) * B;
        return init_ModBrdtElt(M,y);
    end if;
    require false : "Arguments have incompatible degrees.";
```

```
end intrinsic;


intrinsic '*'(x::ModBrdtElt,a::RngElt) -> ModBrdtElt
    {}
    M := Parent(x);
    require Type(Parent(a)) eq RngInt or
        Parent(a) cmpeq BaseRing(M) : "Elements have different parents.";
    z := HackobjCreateRaw(Type(x));
    z`Parent := M;
    z`Element := a*x`Element;
    return z;
end intrinsic;


intrinsic '+'(x::ModBrdtElt,y::ModBrdtElt) -> ModBrdtElt
    {}
    M := Parent(x);
    require Parent(y) eq M : "Elements have different parents.";
    return init_ModBrdtElt(M,x`Element + y`Element);
end intrinsic;


intrinsic '-'(x::ModBrdtElt,y::ModBrdtElt) -> ModBrdtElt
    {}
    M := Parent(x);
    require Parent(y) eq M : "Elements have different parents.";
    return init_ModBrdtElt(M,x`Element - y`Element);
end intrinsic;
```

attributes.m

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                          Verbose mode                              //
//                                                                    //
////////////////////////////////////////////////////////////////////////
declare verbose Brandt, 2;
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                      Attribute declarations                        //
//                                                                    //
////////////////////////////////////////////////////////////////////////
declare attributes ModBrdt:
    Module,          // inner product module of brandt module
    Degree,          // the degree r of the symmetric product Sym^r(I)
    BaseRing,        // the base ring
    LeftIdeals,
    IsIndecomposable,
    IsFull,          // either IsFull, or
    AmbientModule,   //    AmbientModule must be defined
    RamifiedPrimes,  // primes ramified of the quaternion algebra
    Conductor,       // index of the order in a maximal order
    AutoNums,        // the numbers of units
    NormGrams,       // quadratic norm lattices of ideals
    ThetaSeries,     // theta series for the quadratic modules
    HeckePrecision,  // precision of to which Hecke operators are known
    HeckePrimes,     // primes indices for known operators
    HeckeOperators,  // known hecke operators
    AtkinLehnerPrimes,  // the prime divisors of the level
    AtkinLehnerPermutations,  // The sequences of A-L coordinate perms
    CharacterPrimes,
    CharacterPermutations,
    FactorBases,
    FactorIsIndecomposable,
    DecompositionBound;
declare attributes ModBrdtElt:
    Parent,
    Element;
```

brandt_ideals.m

```
//////////////////////////////////////////////////////////////////////
//                                                                  //
//                  LOCAL QUATERNION IDEAL ARITHMETIC               //
//                         FOR BRANDT MODULES                       //
//                           by David Kohel                         //
//                                                                  //
//////////////////////////////////////////////////////////////////////
forward HeckeAdjacencyMatrix;
forward CompareOrders, CompareLeftIdeals, CompareGram;
//////////////////////////////////////////////////////////////////////
//                          Accessory Functions                     //
//////////////////////////////////////////////////////////////////////
function RandomElement(A,S)
    return A![ Random(S) : i in [1..4] ];
end function;
procedure ExtendAdjacencyHecke(M,prms)
    if not IsFull(M) then
        X := AmbientModule(M);
        ExtendAdjacencyHecke(X,prms);
    end if;
    for p in prms do
        if IsFull(M) and p notin M`HeckePrimes then
            h := Dimension(M);
            M`HeckeOperators cat:= [ HeckeAdjacencyMatrix(M,p) ];
            Append(~M`HeckePrimes,p);
        elif p notin M`HeckePrimes then
            g := Dimension(M);
            T := X`HeckeOperators[Index(X`HeckePrimes,p)];
            U := M`Module;
            B := BasisMatrix(M);
            Append(~M`HeckeOperators,
                Matrix([ Coordinates(U,U!(B[i]*T)) : i in [1..g] ]));
            Append(~M`HeckePrimes,p);
        end if;
    end for;
end procedure;
//////////////////////////////////////////////////////////////////////
//                                                                  //
//                          Isogeny Graphs                          //
//                                                                  //
//////////////////////////////////////////////////////////////////////
function HeckeAdjacencyMatrix(M,p)
```

```
    Ideals := M`LeftIdeals;
    Orders := [ RightOrder(I) : I in Ideals ];
    h := #Ideals;
    MZ := RSpace(Integers(),2);
    Frontier := [ MZ!P : P in P1Classes(p) ];
    FF := FiniteField(p);
    PF<X> := PolynomialRing(FF);
    tyme := Cputime();
    vprint Brandt : "Right order computation time:", Cputime(tyme);
    M := MatrixAlgebra(Integers(),h)!0;
    for i in [1..h] do
        I := Ideals[i];
        B := Orders[i];
        repeat
            x1 := RandomElement(B,[-p div 2..p div 2]);
            D1 := Trace(x1)^2 - 4*Norm(x1);
        until KroneckerSymbol(D1,p) eq -1;
        repeat
            x2 := RandomElement(B,[-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;
        for Q in Frontier do
            J := I*LeftIdeal(B,[x2*(B!Q[1] + Q[2]*x1), B!p]);
            C := RightOrder(J);
            j := 1;
            while j le h do
                if CompareOrders(C,Orders[j]) eq -1 then
                    j +:= 1;
                elif CompareLeftIdeals(Ideals[j],J) ne 0 then
                    j +:= 1;
                else
                    break;
                end if;
            end while;
            error if j eq h+1, "Brandt module ideal sequence incomplete.";
            M[i,j] +:= 1;
        end for;
        vprint Brandt, 2 : "Adjacency ideals computed for i =", i;
    end for;
    return M;
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;
```

decomposition.m

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                  Brandt Module Decompositions                      //
//                          David Kohel                               //
//                     Created September 2000                         //
//                                                                    //
////////////////////////////////////////////////////////////////////////
import "hecke_operators.m" : ExtendHecke;
import "brandt_ideals.m" : ExtendAdjacencyHecke;
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                          Decomposition                             //
//                                                                    //
////////////////////////////////////////////////////////////////////////
function Submodule(M,S)
    X := sub< M`Module | S >;
    if Dimension(X) eq Dimension(M`Module) then return M; end if;
    N := HackobjCreateRaw(ModBrdt);
    N`AmbientModule := AmbientModule(M);
    N`Module := X;
    N`IsFull := Dimension(N`Module) eq Degree(M);
    N`RamifiedPrimes := M`RamifiedPrimes;
    N`Conductor := M`Conductor;
    N`BaseRing := M`BaseRing;
    N`HeckePrecision := 0;
    N`HeckePrimes := [ Integers() | ];
    N`HeckeOperators := [ Mat | ]
        where Mat := MatrixAlgebra(N`BaseRing,Dimension(N`Module));
    return N;
end function;


intrinsic EisensteinSubspace(M::ModBrdt) -> ModBrdt
    {}
    // Should be modified for Brandt modules or non-Eichler orders
    // nonmaximal at primes dividing the discriminant.
    if IsFull(M) then
        D := Discriminant(M);
        m := Conductor(M);
        if GCD(D,m) eq 1 then
            r := LCM(M`AutoNums);
            eis := M`Module ! [ r div w : w in M`AutoNums ];
```

```
            E := Submodule(M,[eis]);
            E`IsIndecomposable := true;
        else
            h := Dimension(M);
            if assigned M`NormGrams then
                GenSeq := [
                    Genus(LatticeWithGram(A)) : A in M`NormGrams[1..h] ];
            else
                NormGrams := [
                    1/Norm(I)*GramMatrix(I) : I in M`LeftIdeals ];
                GenSeq := [
                    Genus(LatticeWithGram(A)) : A in NormGrams ];
            end if;
            GenSet := [ GenSeq[1] ];
            for H in GenSeq do
                if &and [ G ne H : G in GenSet ] then
                    Append(~GenSet,H);
                end if;
            end for;
            inds := [ [ i : i in [1..h] |
                G eq GenSeq[i] ] : G in GenSet ];
            B := [ M`Module | ];
            for I in inds do
                r := LCM([ M`AutoNums[i] : i in I ]);
                eis := M`Module ! [ (i in I select r else 0)
                    div M`AutoNums[i] : i in [1..h] ];
                Append(~B,eis);
            end for;
            E := Submodule(M,B);
            E`IsIndecomposable := false;
        end if;
        return E;
    end if;
    return M meet EisensteinSubspace(AmbientModule(M));
end intrinsic;


intrinsic IsEisenstein(M::ModBrdt) -> BoolElt
    {Returns true if and only if M is contained in the Eisenstein
    subspace of its abmient module.}
    return M subset EisensteinSubspace(AmbientModule(M));
end intrinsic;
```

```
intrinsic CuspidalSubspace(M::ModBrdt) -> ModBrdt
    {}
    // Should be modified for Brandt modules or non-Eichler orders
    // nonmaximal at primes dividing the discriminant.
    if IsFull(M) then
        D := Discriminant(M);
        m := Conductor(M);
        if GCD(D,m) eq 1 then
            g := Dimension(M);
            B := [ X | X.i-X.(i+1) : i in [1..g-1] ] where X := M‘Module;
            S := Submodule(M,B);
            if Dimension(S) eq 1 then
                S‘IsIndecomposable := true;
            end if;
        else
            h := Dimension(M);
            if assigned M‘NormGrams then
                GenSeq := [
                    Genus(LatticeWithGram(A)) : A in M‘NormGrams[1..h] ];
            else
                NormGrams := [
                    1/Norm(I)*GramMatrix(I) : I in M‘LeftIdeals ];
                GenSeq := [
                    Genus(LatticeWithGram(A)) : A in NormGrams ];
            end if;
            GenSet := [ GenSeq[1] ];
            for H in GenSeq do
                if &and [ G ne H : G in GenSet ] then
                    Append(~GenSet,H);
                end if;
            end for;
            inds := [
                [ i : i in [1..h] | G eq GenSeq[i] ] : G in GenSet ];
            B := [ M‘Module | ];
            for I in inds do
                B cat:= [ M‘Module |
                    Eltseq(M.I[i]-M.I[i+1]) : i in [1..#I-1] ];
            end for;
            S := Submodule(M,B);
            S‘IsIndecomposable := false;
        end if;
        return S;
```

```
    end if;
    return M meet CuspidalSubspace(AmbientModule(M));
end intrinsic;


intrinsic IsCuspidal(M::ModBrdt) -> BoolElt
    {Returns true if and only if M is contained in the cuspidal
    subspace of its abmient module.}
    return M subset CuspidalSubspace(AmbientModule(M));
end intrinsic;


intrinsic Decomposition(M::ModBrdt,B::RngIntElt
    : Proof := true, Sort := false ) -> SeqEnum
    {Decomposition of M with respect to the Hecke operators T_p
    for p up to the bound B.}
    require Characteristic(BaseRing(M)) notin {2,3} :
        "The characteristic of the base ring of " *
        "argument 1 must be different from 2 and 3.";
    if Dimension(M) eq 0 then return []; end if;
    if not assigned M`FactorBases then
        decomp := [ EisensteinSubspace(M), CuspidalSubspace(M) ];
        decomp := [ N : N in decomp | Dimension(N) ne 0 ];
        M`DecompositionBound := 1;
    else
        decomp := [ Submodule(M,B) : B in M`FactorBases ];
        for i in [1..#M`FactorBases] do
            decomp[i]`IsIndecomposable := M`FactorIsIndecomposable[i];
        end for;
    end if;
    done := &and [ assigned N`IsIndecomposable
        and N`IsIndecomposable : N in decomp ];
    // First decompose with respect to known Atkin-Lehner operators.
    // We don't currently treat Atkin-Lehner operators for primes not
    // dividing the discriminant, so we restrict to those remaining.
    /* Begin Atkin-Lehner decomposition. */
    if not done then
        prms := AtkinLehnerPrimes(M);
        vprint Brandt :
            "Decomposing with respect to Atkin-Lehner primes.";
        for p in prms do
            vprint Brandt : "  Prime =", p;
            tyme := Cputime();
```

```
        ndecomp := [ Parent(M) | ];
        for N in decomp do
            Q := BasisMatrix(N);
            W := AtkinLehnerOperator(N,p);
            PlusMinus := [ Kernel(W+1), Kernel(W-1) ];
            for V in PlusMinus do
                if Dimension(V) ne 0 then
                    S := [ M`Module | x*Q : x in Basis(V) ];
                    Append(~ndecomp,Submodule(N,S));
                end if;
            end for;
        end for;
        decomp := ndecomp;
        for i in [1..#decomp] do
            if Dimension(decomp[i]) eq 1 then
                decomp[i]`IsIndecomposable := true;
            end if;
        end for;
    end for;
    vprint Brandt, 2 : "    Dimensions:",
        [ Dimension(N) : N in decomp ];
    vprint Brandt, 2 : "    IsIndecomp:",
        [ assigned N`IsIndecomposable select 1 else 0 : N in decomp ];
    vprint Brandt, 2 : "    Decomposition time:", Cputime(tyme);
end if;
done := &and [ assigned N`IsIndecomposable
    and N`IsIndecomposable : N in decomp ];
/* End Atkin-Lehner decomposition. */
if not done and M`DecompositionBound lt B then
    vprint Brandt, 2 : "Begin Hecke decomposition.";
    vprint Brandt, 2 :
        "Hecke operators known up to bound", M`HeckePrecision;
    vprint Brandt : "Decomposing up to new bound", B;
    p := NextPrime(M`DecompositionBound);
    i := Floor(Log(2,M`DecompositionBound));
    while p le B do
        // Check precomputed Hecke operators...
        if p le B and p gt 2^i then
            i +:= 1;
            vprint Brandt, 2 :
                "  Recomputing Hecke operators up to bound",
                Min(B,2^i);
            if assigned AmbientModule(M)`NormGrams then
                tyme := Cputime();
```

```
                ExtendHecke(M,Min(B,2^i));
                vprint Brandt, 2 : "    Hecke time:", Cputime(tyme);
            else
                ExtendAdjacencyHecke(M,[p]);
            end if;
        end if;
        vprint Brandt : "  Prime =", p;
        M`DecompositionBound := p;
        // Compute individual operators...
        // T := HeckeOperator(AmbientModule(M),p);
        tyme := Cputime();
        decomp := [ N : N in decomp |
            assigned N`IsIndecomposable and N`IsIndecomposable ]
            cat &cat[
            Decomposition(N,HeckeOperator(N,p) : Proof := Proof)
            : N in decomp | not (assigned N`IsIndecomposable
            and N`IsIndecomposable) ];
        vprint Brandt, 2 : "    Dimensions:",
            [ Dimension(N) : N in decomp ];
        vprint Brandt, 2 : "    IsIndecomp:",
            [ assigned N`IsIndecomposable
            select 1 else 0 : N in decomp ];
        vprint Brandt, 2 : "    Decomposition time:", Cputime(tyme);
        if &and[ assigned N`IsIndecomposable and
            N`IsIndecomposable : N in decomp ] then
            M`DecompositionBound := Infinity();
            break;
        end if;
        p := NextPrime(p);
    end while;
end if;
if Sort and Characteristic(BaseRing(M)) eq 0 then
    decomp := SortDecomposition(decomp);
end if;
M`FactorBases := [ [ Eltseq(x) : x in Basis(N) ] : N in decomp ];
M`FactorIsIndecomposable := [
    assigned N`IsIndecomposable select N`IsIndecomposable else
    false : N in decomp ];
if not IsFull(M) then
    ; // Update decomposition of ambient module?
end if;
return decomp;
end intrinsic;
```

```
intrinsic Decomposition(M::ModBrdt,T::AlgMatElt :
    Proof := true, Sort := false) -> SeqEnum
    {Decomposition of M with respect to the matrix operator T.}
    require Dimension(M) eq Degree(Parent(T)) :
        "Arguments have incompatible degrees.";
    if Dimension(M) eq 0 then return []; end if;
    if assigned M`IsIndecomposable and M`IsIndecomposable then
        return [ M ];
    end if;
    if Dimension(M) eq 1 then
        M`IsIndecomposable := true;
        return [ M ];
    end if;
    decomp := [ Parent(M) | ];
    if Proof then
        chi := CharacteristicPolynomial(T);
    else
        chi := CharacteristicPolynomial(T :
            Al := "Modular", Proof := false);
    end if;
    fac := Factorization(chi);
    for f in fac do
        N := Kernel(Evaluate(f[1],T),M);
        if f[2] eq 1 then
            N`IsIndecomposable := true;
        end if;
        Append(~decomp,N);
    end for;
    if Sort then
        decomp := SortDecomposition(decomp);
    end if;
    return decomp;
end intrinsic;


intrinsic IsIndecomposable(S::ModBrdt,B::RngIntElt) -> BoolElt
    {True if and only if S is indecomposable under the action of
    the Hecke operators up to the bound B.}
    if assigned S`IsIndecomposable then
        return S`IsIndecomposable;
    end if;
    if Dimension(S) eq 1 then
```

```
        S`IsIndecomposable := true;
        return S`IsIndecomposable;
    end if;
    D := Discriminant(S);
    N := Level(S);
    for p in AtkinLehnerPrimes(S) do
        vprint Brandt, 2 :
            "Considering Atkin-Lehner decomposition at", p;
        W := AtkinLehnerOperator(S,p);
        f := CharacteristicPolynomial(W :
            Al := "Modular", Proof := false);
        charfac := Factorization(f);
        if #charfac gt 1 then
            S`IsIndecomposable := false;
            return S`IsIndecomposable;
        end if;
    end for;
    p := 2;
    while N mod p eq 0 do
        p := NextPrime(p);
    end while;
    while p le B do
        vprint Brandt, 2 : "Considering decomposition at", p;
        Tp := HeckeOperator(S,p);
        f := CharacteristicPolynomial(Tp :
            Al := "Modular", Proof := false);
        charfac := Factorization(f);
        if #charfac eq 1 and charfac[1][2] eq 1 then
            S`IsIndecomposable := true;
            return S`IsIndecomposable;
        elif #charfac gt 1 then
            S`IsIndecomposable := false;
            return S`IsIndecomposable;
        end if;
        p := NextPrime(p);
    end while;
    return false;
end intrinsic;
function MergeSort(C,D)
    i := 1; // index for C
    j := 1; // index for D
    while true do
        if i gt #C then
            break;
```

```
        elif j gt #D then
            D cat:= C[i..#C];
            break;
        elif C[i] lt D[j] then
            Insert(~D,j,C[i]);
            i +:= 1;
        else
            j +:= 1;
        end if;
    end while;
    return D;
end function;


intrinsic SortDecomposition(D::[ModBrdt]) -> SeqEnum
    {Sort the sequence of Brandt modules according to the 'lt' operator.}
    r := #D;
    vprint Brandt, 2 : "Sorting sequence of length", r;
    if r eq 2 then
        vprintf Brandt, 2 : "   Dimensions: %o, %o\n",
            Dimension(D[1]), Dimension(D[2]);
        if D[2] lt D[1] then
            vprint Brandt, 2 : "   Reversing";
            Reverse(~D);
        else
            vprint Brandt, 2 : "   Returning";
        end if;
    elif r gt 2 then
        s := r div 2;
        D := MergeSort(
            SortDecomposition(D[1..s]),
            SortDecomposition(D[s+1..r]) );
    end if;
    return D;
end intrinsic;


intrinsic OrthogonalComplement(M::ModBrdt) -> ModBrdt
    {}
    N := Kernel(InnerProductMatrix(M)*Transpose(BasisMatrix(M)));
    return Submodule(AmbientModule(M),Basis(N));
end intrinsic;
//////////////////////////////////////////////////////////////////////
```

```
//                                                                      //
//                          Linear Algebra                              //
//                                                                      //
//////////////////////////////////////////////////////////////////////////


intrinsic BasisMatrix(N::ModBrdt) -> ModMatRngElt
    {The matrix whose rows are the basis elements of N.}
    return Matrix(Dimension(N),Degree(N),
        &cat[ Eltseq(x) : x in Basis(N) ]);
end intrinsic;


intrinsic AtkinLehnerEigenvalue(M::ModBrdt,q::RngIntElt) -> RngElt
    {The trace of the qth Atkin-Lehner opertor on the
    indecomposable Brandt module S.}
    D := Discriminant(M);
    N := Level(M);
    val, p := IsPrimePower(q);
    require val : "Argument 2 must be a prime power.";
    require N mod q eq 0 and GCD(N div q,q) eq 1 :
        "Argument 2 must be an exact divisor of the level of argument 1.";
    W := AtkinLehnerOperator(M,q);
    if W eq 1 then return 1; elif W eq -1 then return -1; end if;
    require false :
        "Argument 1 is not indecomposable under Atkin-Lehner.";
end intrinsic;
function MatrixKernel(T,V)
    return Kernel(T) meet V;
end function;


intrinsic Kernel(T::AlgMatElt,M::ModBrdt) -> ModBrdt
    {}
    Mat := Parent(T);
    require BaseRing(M) eq BaseRing(Mat) :
        "Arguments have incompatible base rings.";
    A := AmbientModule(M);
    U := M`Module;
    if Dimension(M) eq Degree(Mat) then
        V := Kernel(T);
        t := Dimension(V);
        C := BasisMatrix(V)*BasisMatrix(U);
```

```
        return Submodule(M,[ C[i] : i in [1..t] ]);
    elif Degree(M) eq Degree(Mat) then
        S := Basis(Kernel(T) meet M`Module);
        return Submodule(M,S);
    end if;
    require false : "Arguments have incompatible degrees.";
end intrinsic;


intrinsic 'meet'(M::ModBrdt,N::ModBrdt) -> ModBrdt
    {}
    if M eq N then return M; end if;
    A := AmbientModule(M);
    require AmbientModule(N) eq A : "Modules have no covering module.";
    if IsFull(M) then
        return N;
    elif IsFull(N) then
        return M;
    end if;
    return Submodule(A,Basis(M`Module meet N`Module));
end intrinsic;


intrinsic 'subset'(M::ModBrdt,N::ModBrdt) -> BoolElt
    {}
    if M eq N then return true; end if;
    A := AmbientModule(M);
    require AmbientModule(N) eq A : "Modules have no covering module.";
    return M`Module subset N`Module;
end intrinsic;


intrinsic CharacteristicPolynomial(
    T::AlgMatElt,M::ModBrdt : Proof := true) -> RngUPolElt
    {The characteristic polynomial of T on M.}
    require Degree(Parent(T)) eq Degree(M) :
        "Arguments have incompatible dimensions.";
    require BaseRing(Parent(T)) cmpeq BaseRing(M) :
        "Arguments have incompatible base rings.";
    return CharacteristicPolynomial(T,M`Module : Proof := Proof);
end intrinsic;
//////////////////////////////////////////////////////////////////////
//                                                                  //
```

```
//            Ordering of Indecomposable Brandt Submodules            //
//                                                                    //
////////////////////////////////////////////////////////////////////////


intrinsic 'lt'(R::ModBrdt,S::ModBrdt : Bound := 101) -> BoolElt
    {Comparison operator extending that of Cremona.}
    /*
    (1) R < S if dim(R) < dim(S);
    (2) R < S if IsEisenstein(R) and IsCuspidal(S)
    (3) When the weight is two and the character is trivial:
        order by Wq eigenvalues, starting with *smallest* p|N and
        with "+" being less than "-";
    (4) Order by abs(trace(a_p^i)), p not dividing the level, and
        i = 1,...,g = dim(R) = dim(S), with the positive one being
        smaller in the the event of equality,
    */
    M := AmbientModule(R);
    require AmbientModule(S) eq M :
        "Arguments must be components of the same Brandt module.";
    /*
    require IsIndecomposable(R) and IsIndecomposable(S) :
       "Arguments must be indecomposable Brandt modules.";
    */
    if R eq S then return false; end if;
    g := Dimension(S);
    if Dimension(R) lt g then
        return true;
    elif Dimension(R) gt g then
        return false;
    end if;
    if IsEisenstein(R) and IsCuspidal(S) then
        return true;
    elif IsEisenstein(S) and IsCuspidal(R) then
        return false;
    end if;
    N := Level(S);
    D := Discriminant(S);
    atkin_lehner := AtkinLehnerPrimes(M);
    for p in atkin_lehner do
        q := p^Valuation(N,p);
        eR := AtkinLehnerEigenvalue(R,q);
        eS := AtkinLehnerEigenvalue(S,q);
        if eR lt eS then
```

```
            return true;
        elif eR gt eS then
            return false;
        end if;
    end for;
    p := 2;
    while p le Bound do
        Tp := HeckeOperator(M,p);
        fR := CharacteristicPolynomial(Tp,R : Proof := false);
        fS := CharacteristicPolynomial(Tp,S : Proof := false);
        if p notin atkin_lehner and fR ne fS then
            for i in [1..g] do
                aS := HeckeTrace(S,p^i : Proof := false);
                aR := HeckeTrace(R,p^i : Proof := false);
                if Abs(aR) lt Abs(aS) then
                    return true;
                elif Abs(aR) gt Abs(aS) then
                    return false;
                elif aR gt aS then
                    return true;
                elif aR lt aS then
                    return false;
                end if;
            end for;
        end if;
        p := NextPrime(p);
    end while;
    return false;
end intrinsic;


intrinsic 'gt'(N::ModBrdt,M::ModBrdt) -> BoolElt
    {}
    return M lt N;
end intrinsic;
```

dimensions.m

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                    Dimensions of Brandt Modules                    //
//                            David Kohel                             //
//                                                                    //
////////////////////////////////////////////////////////////////////////


intrinsic BrandtModuleDimension(D::RngIntElt) -> RngIntElt
    {The dimension of the Brandt module of level D.}
    return BrandtModuleDimension(D,1);
end intrinsic;


intrinsic BrandtModuleDimension(D::RngIntElt,N::RngIntElt) -> RngIntElt
    {The dimension of the Brandt module of level (D,N).}
    require GCD(D,N) eq 1 : "Arguments must be coprime.";
    Dprms := PrimeDivisors(D);
    Nprms := PrimeDivisors(N);
    require D eq &*Dprms and #Dprms mod 2 eq 1 :
        "Argument 1 must be the product of an odd number of primes.";
    return 1 + &+[ &*[ Integers() | 1 + Valuation(M,p) : p in Nprms ] *
        DimensionNewCuspFormsGamma0(D*(N div M),2) : M in Divisors(N) ];
end intrinsic;
```

dirichlet.m

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                    Dimensions of Brandt Modules                    //
//                            David Kohel                             //
//                                                                    //
////////////////////////////////////////////////////////////////////////


intrinsic DirichletCharacter(M::ModBrdt) -> GrpDrchElt
    {}
    return DirichletGroup(1,BaseRing(M))!1;
end intrinsic;
```

hecke_operators.m

```
//////////////////////////////////////////////////////////////////////////
//                                                                      //
//                  Hecke Operators for Brandt Modules                  //
//                            David Kohel                               //
//                                                                      //
//////////////////////////////////////////////////////////////////////////
import "brandt_ideals.m" : ExtendAdjacencyHecke;
forward ExtendHecke;
function LocalDualGram(L,q)
    Mat := MatrixAlgebra(ResidueClassRing(q), Rank(L));
    U := Kernel(Mat!GramMatrix(L));
    B := [ L!v : v in Basis(U) ] cat [ q*v : v in Basis(L) ];
    return MinkowskiGramReduction((1/q)*GramMatrix(sub< L | B >)
        : Canonical := true);
end function;
function brandt_index(i,j,h)
    if j lt i then
        return Binomial(h+1,2) - Binomial(h-j+2,2) + Abs(i-j) + 1;
    end if;
    return Binomial(h+1,2) - Binomial(h-i+2,2) + Abs(i-j) + 1;
end function;
function brandt_coordinates(k,h)
    i := 1;
    r := 1;
    while k gt (r+h-i) do
        i +:= 1;
        r := Binomial(h+1,2) - Binomial(h-i+2,2) + 1;
    end while;
    j := i + k - r;
    return [i,j];
end function;
function HeckeMatrix(A,S,W)
    n := Degree(A);
    M := Zero(A);
    k := 1;
    for i in [1..n] do
        M[i,i] := ExactQuotient(S[k],W[i]);
        k +:= 1;
        for j in [i+1..n] do
            M[i,j] := ExactQuotient(S[k],W[j]);
            M[j,i] := ExactQuotient(S[k],W[i]);
            k +:= 1;
```

```
        end for;
    end for;
    return M;
end function;
procedure ExtendHecke(M,prec)
    if prec le M`HeckePrecision then return; end if;
    if IsFull(M) then
        h := Dimension(M);
        PS := Universe(M`ThetaSeries);
        k := 1;
        for i in [1..h] do
            M`ThetaSeries[k] := PS![
                Coefficient(t,2*k) : k in [0..prec] ] where t is
                ThetaSeries(LatticeWithGram(M`NormGrams[k]),2*prec);
            k +:= 1;
            for j in [i+1..h] do
                M`ThetaSeries[k] := PS![
                    Coefficient(t,2*k) : k in [0..prec] ] where t is
                    ThetaSeries(LatticeWithGram(M`NormGrams[k]),2*prec);
                k +:= 1;
            end for;
        end for;
        M`HeckePrimes := [ t : t in [2..prec] | IsPrime(t) ];
        M`HeckeOperators := [
            HeckeMatrix(MatrixAlgebra(BaseRing(M),h),
            [ Coefficient(f,t) : f in M`ThetaSeries ], M`AutoNums)
            : t in M`HeckePrimes ];
        M`HeckePrecision := prec;
    else
        X := AmbientModule(M);
        ExtendHecke(X,prec);
        // The submodules get created and destroyed with frequency.
        // Only compute Hecke operators from the ambient module up
        // to the needed precision.
        g := Dimension(M);
        M`HeckePrimes := [ p : p in X`HeckePrimes | p le prec ];
        U := M`Module;
        B := BasisMatrix(M);
        M`HeckeOperators := [
            Matrix([ Coordinates(U,U!(B[i]*T)) : i in [1..g] ])
            : T in X`HeckeOperators[1..#M`HeckePrimes] ];
        M`HeckePrecision := prec;
    end if;
end procedure;
```

```
procedure AtkinLehnerSetup(M)
    h := Degree(M);
    if assigned M`NormGrams then
        OrderGrams := [ M`NormGrams[k] where
            k := brandt_index(i,i,h) : i in [1..h] ];
    else
        OrderGrams := [ ReducedGramMatrix(RightOrder(I)) :
            I in M`LeftIdeals ];
    end if;
    GramSet := SequenceToSet(OrderGrams);
    prms := AtkinLehnerPrimes(M);
    M`AtkinLehnerPermutations := [ [ i : i in [1..h] ] : p in prms ];
    for A in GramSet do
        L := LatticeWithGram(A);
        I := [ j : j in [1..h] | OrderGrams[j] eq A ];
        for t in [1..#prms] do
            p := prms[t];
            q := p^Valuation(Level(M),p);
            B := LocalDualGram(L,q);
            for i in I do
                if assigned M`NormGrams then
                    K := [ brandt_index(i,j,h) : j in I ];
                    Ip := [ k : k in K | M`NormGrams[k] eq B ];
                else
                    K := I;
                    Ip := [ j : j in I | IsIsometric(
                        LatticeWithGram( 1/(Norm(I)*Norm(J)) *
                        GramMatrix(Conjugate(J)*I)),LatticeWithGram(B))
                        where I := M`LeftIdeals[i]
                        where J := M`LeftIdeals[j] ];
                end if;
                assert #Ip eq 1;
                M`AtkinLehnerPermutations[t][i] := I[Index(K,Ip[1])];
            end for;
        end for;
    end for;
end procedure;


intrinsic AtkinLehnerOperator(M::ModBrdt,q::RngIntElt) -> AlgMatElt
    {}
    A := AmbientModule(M);
    N := Level(A);
```

```
    D := Discriminant(A);
    val, p, r := IsPrimePower(q);
    require val : "Argument 2 must be a prime power.";
    require p in AtkinLehnerPrimes(M) :
        "Atkin-Lehner operator does not exist " *
        "or is not computed for this prime.";
    // Convenient convention to ask for the Atkin-Lehner operator
    // by the prime instead of the prime power.
    if r eq 1 then
        r := Valuation(N,p);
        q := p^r;
    end if;
    require r eq Valuation(N,p) :
        "Argument 2 must be an exact divisor of the level of argument 1.";
    if not assigned A`AtkinLehnerPermutations then
        AtkinLehnerSetup(A);
    end if;
    // Create Atkin-Lehner matrix for A.
    h := Dimension(A);
    W := MatrixAlgebra(BaseRing(A),h)!0;
    k := Index(A`AtkinLehnerPrimes,p);
    pi := A`AtkinLehnerPermutations[k];
    eps := D mod p eq 0 select -1 else 1;
    for i in [1..h] do
        W[i,pi[i]] := eps;
        W[pi[i],i] := eps;
    end for;
    // Reconstruct Atkin-Lehner matrix for M.
    n := Dimension(M);
    U := M`Module;
    C := BasisMatrix(U)*W;
    return Matrix(n,&cat[ Coordinates(U,U!C[i]) : i in [1..n]]);
end intrinsic;


intrinsic CharacterOperator(M::ModBrdt,q::RngIntElt) -> AlgMatElt
    {}
    A := AmbientModule(M);
    N := Level(A);
    D := Discriminant(A);
    ZZ := Integers();
    val, p, r := IsPrimePower(q);
    require val : "Argument 2 must be a prime power.";
    if r eq 1 then
```

```
        r := Valuation(N,p);
        q := p^r;
    end if;
    require r eq 2 and Conductor(A) mod p eq 0 :
        "Level of argument 1 must be a square of a prime " *
        "and argument 2 must be a divisor of the conductor.";
    require assigned A`LeftIdeals : "LeftIdeals are not assigned.";
    if not assigned A`CharacterPrimes then
        A`CharacterPrimes :=
            [ p : p in A`RamifiedPrimes | Conductor(A) mod p eq 0 ];
        A`CharacterPermutations := [ [ZZ|] : p in A`CharacterPrimes ];
    end if;
    n := Dimension(A);
    i := Index(A`CharacterPrimes,p);
    if #A`CharacterPermutations[i] eq 0 then
        idls := A`LeftIdeals;
        R := LeftOrder(idls[1]);
        K := QuaternionAlgebra(R);
        O := MaximalOrder(K);
        u := K!ReducedBasis(O)[2];
        f := MinimalPolynomial(u);
        if ZZ!Discriminant(f) mod p eq 0 then
            u := K!ReducedBasis(O)[3];
            f := MinimalPolynomial(u);
        end if;
        assert KroneckerSymbol(ZZ!Discriminant(f),p) eq -1;
        F := ext< F | PolynomialRing(F)!f > where F := FiniteField(p);
        e := PrimitiveElement(F);
        u := c[1] + c[2]*u where c := ChangeUniverse(Eltseq(e),ZZ);
        jdls := [ lideal< R | [ p*u*x : x in B ] cat [ p^2*x : x in B ]
            where B := Basis(I) > : I in idls ];
        indices := [ [ j : j in [1..n] |
            IsLeftIsomorphic(idls[j],J) ][1] : J in jdls ];
        char_perms := A`CharacterPermutations;
        char_perms[i] := indices;
        A`CharacterPermutations := char_perms;
    end if;
    g := Sym(n)!(A`CharacterPermutations[i]);
    W := PermutationMatrix(BaseRing(M),g);
    m := Dimension(M);
    U := M`Module;
    C := BasisMatrix(U)*W;
    return Matrix(m,&cat[ Coordinates(U,U!C[i]) : i in [1..m] ]);
end intrinsic;
```

```
function HeckeRecursion(ap,p,r,k,e)
    if r eq 0 then
        return Parent(ap)!1;
    elif r eq 1 then
        return ap;
    else
        return ap * HeckeRecursion(ap,p,r-1,k,e) -
            e * p^(k-1) * HeckeRecursion(ap,p,r-2,k,e);
    end if;
end function;


intrinsic HeckeTrace(M::ModBrdt,q::RngIntElt : Proof := true) -> RngElt
    {The trace of the q-th Hecke operator on M, for prime power q.}
    val, p, r := IsPrimePower(q);
    require val : "Argument 2 must be a prime power.";
    g := Dimension(M);
    if q eq 1 then
        return BaseRing(M)!g;
    elif IsPrime(q : Proof := false) then
        return Trace(HeckeOperator(M,q));
    end if;
    Tp := HeckeOperator(M,p);
    fp := CharacteristicPolynomial(Tp :
        Al := "Modular", Proof := Proof);
    R<ap> := quo< Parent(fp) | fp >;
    // k := Weight(M);
    k := 2;
    // e := Evaluate(DirichletCharacter(M),p);
    e := 1;
    return Trace(HeckeRecursion(ap,p,r,k,e));
end intrinsic;


intrinsic HeckeOperator(M::ModBrdt,n::RngIntElt) -> AlgMatElt
    {}
    require n ge 1 : "Argument 2 must be positive.";
    if n eq 1 then
        return Identity(Universe(M`HeckeOperators));
    elif IsPrime(n : Proof := false) then
        if n in M`HeckePrimes then
            return M`HeckeOperators[Index(M`HeckePrimes,n)];
        end if;
```

```
    elif IsPrimePower(n) then
        // k := Weight(M);
        // e := Evaluate(DirichletCharacter(M),p);
        k := 2;
        e := 1;
        _, p, r := IsPrimePower(n);
        // Testing purposes: what happens at theses primes?
        if Level(M) mod p eq 0 then
            // By design, only computes up to Hecke operators
            // previous largest prime
            ExtendHecke(M,n);
            B := BasisMatrix(M);
            A := AmbientModule(M);
            T := HeckeMatrix(MatrixAlgebra(BaseRing(A),Degree(A)),
                [ Coefficient(f,n) : f in A`ThetaSeries ], A`AutoNums);
            U := M`Module;
            g := Dimension(M);
            return Matrix([ Coordinates(U,U!(B[i]*T)) : i in [1..g] ]);
        end if;
        return HeckeOperator(M,p) * HeckeOperator(M,p^(r-1))
            - e * p^(k-1) * HeckeOperator(M,p^(r-2));
    end if;
    fac := Factorization(n);
    m := fac[#fac][1];
    if m notin M`HeckePrimes then
        if assigned AmbientModule(M)`NormGrams then
            ExtendHecke(M,m);
        else
            ExtendAdjacencyHecke(M,[f[1] : f in fac]);
        end if;
    end if;
    return &*[ HeckeOperator(M,f[1]^f[2]) : f in fac ];
end intrinsic;
```

monodromy.m

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                         Creation functions                         //
//                                                                    //
////////////////////////////////////////////////////////////////////////


intrinsic MonodromyWeights(M::ModBrdt) -> SeqEnum
    {}
    return [ e div 2 : e in M`AutoNums ];
end intrinsic;
```

qexpansion.m

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                          David Kohel                               //
//               q-Expansion Bases for Brandt Modules                 //
//                                                                    //
////////////////////////////////////////////////////////////////////////
import "hecke_operators.m" : ExtendHecke;
forward EchelonSeries, ValuationOrder;


intrinsic ThetaSeries(
    u::ModBrdtElt, v::ModBrdtElt, prec::RngIntElt) -> RngSerElt
    {The value of the theta series pairing.}
    M := Parent(u);
    require M eq Parent(v) : "Arguments have different parents.";
    A := AmbientModule(M);
    if M ne A then
        return ThetaSeries(A!u, A!v, prec);
    end if;
    h := Dimension(M);
    if A`HeckePrecision lt prec then
        ExtendHecke(A,prec);
    end if;
    PS := Universe(A`ThetaSeries);
    f := PS!0;
    c1 := Eltseq(u); c2 := Eltseq(v);
    for i in [1..h] do
        k := Binomial(h+1,2) - Binomial(h-i+2,2) + 1;
        c := c1[i]*c2[i];
        if c ne 0 then f +:= c1[i]*c2[i]*A`ThetaSeries[k]; end if;
        for j in [i+1..h] do
            k := Binomial(h+1,2) - Binomial(h-i+2,2) + j-i+1;
            c := c1[i]*c2[j] + c1[j]*c2[i];
            if c ne 0 then f +:= c*A`ThetaSeries[k]; end if;
        end for;
    end for;
    return f + O(PS.1^(prec+1));
end intrinsic;


intrinsic qExpansionBasis(M::ModBrdt,prec::RngIntElt) -> SeqEnum
    {}
```

```
    PS := LaurentSeriesRing(RationalField());
    q := PS.1;
    B := Basis(M);
    F := EchelonSeries(&cat[ Parent([ PS | ]) |
        [ PS ! ThetaSeries(B[i], B[j], prec)
        : i in [1..j] ] : j in [1..#B] ]);
    return F;
end intrinsic;


intrinsic qExpansionBasis(B::[ModBrdtElt],prec::RngIntElt) -> SeqEnum
    {}
    PS := LaurentSeriesRing(RationalField());
    q := PS.1;
    F := EchelonSeries(&cat[ Parent([ PS | ]) |
        [ PS ! ThetaSeries(B[i], B[j], prec)
        : i in [1..j] ] : j in [1..#B] ]);
    return F;
end intrinsic;
function EchelonSeries(B)
    // Returns the echelonized sequence of power series spanning
    // the same space.
    error if not IsField(CoefficientRing(Universe(B))),
        "Series base ring must be a field";
    if #B eq 0 then return B; end if;
    for i in [1..#B] do
        Sort(~B,ValuationOrder);
        n1 := Valuation(B[i]);
        if n1 lt AbsolutePrecision(B[i]) then
            B[i] := B[i]/Coefficient(B[i],n1);
            for j in [1..#B] do
                if j lt i then
                    B[j] := B[j] - Coefficient(B[j],n1)*B[i];
                elif j gt i and (n1 eq Valuation(B[j])) then
                    B[j] := B[j] - Coefficient(B[j],n1)*B[i];
                    n2 := Valuation(B[j]);
                    if n2 lt AbsolutePrecision(B[i]) then
                        B[j] := B[j]/Coefficient(B[j],n2);
                    end if;
                end if;
            end for;
        end if;
    end for;
    while #B gt 0 and RelativePrecision(B[#B]) eq 0 do
```

```
        Remove(~B,#B);
        if #B eq 0 then break; end if;
    end while;
    return B;
end function;
function ValuationOrder(f,g)
    // Partial ordering of series based on valuation.
    n := Valuation(f);
    m := Valuation(g);
    if n gt m then return 1;
    elif n lt m then return -1;
    end if;
    return 0;
end function;
```