```
//1.  The first set of functions deal with Euclidean Algorithm type
decompositions


function InversionMatrix(O);

return Matrix(O,2,2,[0,-1,1,0]);

end function;

function TranslationMatrix(O,a,b);

return Matrix(O,2,2,[1,-(a div b),0,1]);

end function;

//This is the psuedo-Euclidean Algorithm in as in Lingham to decompose into
Manin Symbols.
function CreateBList2(a,b,O,rev);
        List:=[Matrix(O,2,2,[1,0,0,1])];
        I:=InversionMatrix(O);

        while a ne 0 do
                T:=TranslationMatrix(O,b,a);
                A:=I*T;
                Append(~List, List[#List]*A^(-1) );
                v:=A*Matrix(O,2,1,[b,a]);
                b:=v[1,1];a:=v[2,1];

        end while;

                for i in [1..#List] do
                        List[i]:=SwapRows(List[i],1,2);
                        if rev then
                        List[i]:=SwapColumns(List[i],1,2);
                        end if;
                end for;


        return Reverse(Prune(Reverse(List)));
end function;

function ExpandModularSymbol(M,O);

        a:=M[1,1]; b:=M[1,2]; c:=M[2,1]; d:=M[2,2];
```

```
        return CreateBList2(b,d,0,true) cat CreateBList2(a,c,0,false);

end function;



function nonzero(x);
        if x ne 0 then return 1;
                else return 0;
        end if;
end function;

function gcdf(a,b);

        switch:=false;
        L:=[];  //quotient list
        q:= b div a;

        //Keep q not 0
        if q eq 0 then
                g:=[a,b];
                b:=g[1];
                a:=g[2];
                switch:=true;
        end if;

        r1:=a;
        r2:=b-q*a;

        if r2 eq 0 then return -q+1,1,a;//the gcd is already determined
        end if;

        Append(~L,q);

        while r2 ne 0 do

                q:=r1 div r2;

                t:=r2;
                r2:=r1-q*r2;
                r1:=t;
                if r2 ne 0 then Append(~L,q);
        end if;

        end while;
```

```
        /*
        Find the coefficients of a and b by plugging (1,0) or (0,1) into
the appropriate iterated expression.
        */
        t1:=0;
        t2:=1;

        w1:=1;
        w2:=0;

        for q in L do

                x:=t1;
                t1:=t2;
                t2:= x-t2*q;

                y:=w1;
                w1:=w2;
                w2:= y-w2*q;


        end for;

        n:=w2;
        m:=t2;


        if not switch then
                return m,n, r1;
        else return n,m, r1;
        end if;

end function;

//2. The following functions are for manipulating manin and/or analyzing
symbols

//The type 4 manin symbol is 0 in certain eigenspaces.  Assuming it is
nonzero, frustrates
//issues while applying Hecke operators
function ModificationCondition(k,l,w);

        return 2-k mod w ne 0 or 2-l mod w ne 0;

end function;
```

```
function InvertManin(m);
        return [m[3],m[4],m[1],m[2]];
end function;

function WhichSubspace(m);
        case [nonzero(m[1]),nonzero(m[2]),nonzero(m[3]),nonzero(m[4])]:
                when [1,1,1,1]:
                        return 1,m, 1;
                when [0,1,1,1]:
                        return 2,m, 1;
                when [1,1,0,1]:
                        return 2, InvertManin(m), -1;
                when [1,0,1,1]:
                        return 3, m, 1;
                when [1,1,1,0]:
                        return 3, InvertManin(m), -1;
                when [0,1,1,0]:
                        return 4, m, 1;
                when [1,0,0,1]:
                        return 4, InvertManin(m), -1;
                else:
                        return 0,0,0;
        end case;
end function;

function ImageOfManin(m);

        option,m,mult:=WhichSubspace(m);
        a:=m[1];b:=m[2];c:=m[3];d:=m[4];
        case option:
                when 1: return [0,0,a,b], [0,0,c,d],mult;
                when 2: return [c^(-1),0,0,b],[0,0,c,d],mult;
                when 3: return [0,d^(-1),a,0], [0,0,c,d],mult;
                when 4: return [c^(-1),0,0,b], [0,-b^(-1),c,0],mult;
        end case;

end function;

function LiftManin(m,I1,I2,s1,s2,p)
        a:=m[1];
        b:=m[2];
        c:=m[3];
        d:=m[4];
        u:=CRT(I1,I2,Inverse(s1)(a),Inverse(s2)(b));
        v:=CRT(I1,I2,Inverse(s1)(c),Inverse(s2)(d));
        while not IsUnit(GCD(u,v)) do
```

```
                v:=v+p;
        end while;
        return u,v;
end function;

function CreateTopOfManin(u,v,p,s1,s2);
        y,x, ll :=gcdf(u,v);
        return [s1(x),s2(x),s1(-y),s2(-y)];

end function;

//3. This section of functions is for computing the Hecke image of modular
symbols and
//for conversion between manin symbols and modular symbols

function ManinToModular(m,I1,I2,s1,s2,p,O);

        u,v:=LiftManin(m,I1,I2,s1,s2,p);
        y,x, ll :=gcdf(u,v);
        return Matrix(O,2,2,[x,y,u,v]);

end function;

function TPList(prime,p,O);
        A:=Matrix(O,2,2,[0,-1,p,0]);
        List:=[];
        t1:=Cputime();
        Q :=quo<O|O!prime>;
        Res := Elements(Q);
        for y in Res do
                if p mod prime eq 0 then
                        Append(~List,A*Matrix(O,2,2,[1,y,0,prime])*A);
                else Append(~List,Matrix(O,2,2,[1,y,0,prime]));
                end if;
        end for;
        if p mod prime ne 0 then
                n,m, ll:=gcdf(p,prime);
                Append(~List,Matrix(O,2,2,[m*prime,-n,p*prime,prime]));
        end if;
        print "TPList", t1-Cputime() ;
        return List;

end function;

//The Hecke action on the modular symbols (left action instead of right)
comes from this list
```

```
//multiplication
function LeftMultL(list,M)

        for i in[1..#list] do
                list[i]:=list[i]*M;
        end for;
        return list;
end function;

function ModularToManin(M,O,s1,s2);

        return [s1(M[2,1]),s2(M[2,1]),s1(M[2,2]),s2(M[2,2])];

end function;


function TPActionOnManin(m,I1,I2,s1,s2,p,prime,O,TP);

        manins:=[];
        result:=[];


        M:=ManinToModular(m,I1,I2,s1,s2,p,O);
        List:=LeftMultL(TP,M);
        for i in [1..#List] do
                //print "M:=", M, "List:=", ExpandModularSymbol(List[i],O);
                //print prime;
                //print ExpandModularSymbol(List[i],O);
                result cat:=ExpandModularSymbol(List[i],O);
        end for;
        for i in [1..#result] do
                Append(~manins,ModularToManin(result[i],O,s1,s2));
        end for;
        return manins;

end function;

//4. This section is for the boundary map

function ReduceCusp(cusp,k,l,w)
        change:=ModificationCondition(k,l,w);
        a:=cusp[1];b:=cusp[2];c:=cusp[3];d:=cusp[4];
        case [nonzero(a),nonzero(b),nonzero(c),nonzero(d)]:
                when [0,0,1,1]: return c^(2-k)*d^(2-l), 1;
                when [1,0,0,1]: if not change then return a^(k-2)*d^(2-l),
2;
```

```
                        else return 0,2;
                        end if;
                when [0,1,1,0]: if not change then return c^(2-k)*b^(l-2),
3;
                        else return 0,3;
                        end if;
        end case;

end function;

function VectorImageOfManin(m,F,k,l,w);

        v:=KSpace(F,3)!0;
        c1,c2,mult:=ImageOfManin(m);
        //print c1,c2,mult;
        coeff1, pos1:=ReduceCusp(c1,k,l,w);
        coeff2, pos2:=ReduceCusp(c2,k,l,w);
        //print coeff1, pos1;
        //print coeff2, pos2;
        v[pos1]:=mult*coeff1;
        v[pos2] -:=mult*coeff2;
        return v;
end function;

function BoundaryMatrix1(Rep,F,k,l,w);

        A:=KMatrixSpace(F,#Rep,3)!0;
        for n in [1..#Rep] do

                A[n]:=VectorImageOfManin(Rep[n],F,k,l,w);

        end for;
        return A;

end function;


//5. This section is for finding representative manin symbols.  The
technique is to use
//an orthogonal basis.  The projections of manin symbols to this basis are
in terms of
//character values.

function GroupSize(option,p,w);

        case option:
```

```
                when 1: return ((p-1)^2)/w;
                when 2: return (p-1)/w;
                when 3: return (p-1)/w;
                when 4: return 1;
        end case;

end function;

function ManinCoord(m, option,i,j,k,l);
        case option:
                when 1: if i eq -1 or j eq -1 then return 0; end if;
                        return m[1]^(1-i)*m[2]^(1-j)*m[3]^(i+1-k)*m[4]^(j
+1-l);
                when 2: if i ne -1 or j eq -1 then return 0; end if;
                        return m[2]^(1-j)*m[3]^(2-k)*m[4]^(j+1-l);
                when 3: if j ne -1 or i eq -1 then return 0; end if;
                        return m[1]^(1-i)*m[3]^(i+1-k)*m[4]^(2-l);
                when 4: if i ne -1 or j ne -1 then return 0; end if;
                        return m[3]^(2-k)*m[2]^(2-l);
        end case;
end function;

function NormalizeVector(v)

        length:=Degree(v);
        firstnonzero:=false;
        for i in [1..length] do
                if v[i] ne 0 and not firstnonzero then firstnonzero:=true;
                        divisor:=v[i];
                end if;
                if firstnonzero then
                        v[i]:=v[i]/divisor;
                end if;
        end for;
        return v;
end function;

function NormalizeMatrix(M);

        length:=#Rows(M);
        for i in [1..length] do
                M[i]:=NormalizeVector(M[i]);
        end for;
        return M;
end function;
```

```
function InvertVector(v);

        length:= Degree(v);
        for i in [1..length] do
                if v[i] ne 0 then
                        v[i]:=v[i]^(-1);
                end if;
        end for;
        return v;
end function;

function InvertMatrixVectors(M);

        length:=#Rows(M);
        for i in [1..length] do
                M[i]:=InvertVector(M[i]);
        end for;
        return M;

end function;

function ChgManinToCoord(m,F,N,k,l);
        option, m, mult:=WhichSubspace(m);
        v:=KSpace(F,#N)!0;
        for pos in [1..#N] do
                i:=N[pos][1];j:=N[pos][2];
                v[pos]:= ManinCoord(m,option,i,j,k,l);
        end for;
        return v;
end function;

function ReducedManin(m,Basis, NormalizedBasis, ITBasis, N,F,k,l,p,w)
        option, m, mult1:=WhichSubspace(m);
        if option eq 0 then return 0,0; end if;
        size:=GroupSize(option,p,w);
        v:=ChgManinToCoord(m,F,N,k,l);
        Rows:=Rows(NormalizedBasis);
        pos:= Position(Rows, NormalizeVector(v)) ;
        //Uses orthogonality of characters to obtain the coordinate in
terms of this basis.
        h:=ColumnSubmatrixRange(ITBasis,pos,pos);
        mult2:=(v*h)[1]/F!size;
        return pos, mult1*mult2;

end function;
```

```
//This function is just thrown down here because it uses something from
this section
function TPManinRow(m,I1,I2,s1,s2,p,prime,MB, NB, IB, N,F,k,l,w,O,TP);

        change:=ModificationCondition(k,l,w);
        v:=KSpace(F,#N)!0;
        manins:=TPActionOnManin(m,I1,I2,s1,s2,p,prime,O,TP);
        //print manins;
        for x in manins do
                if WhichSubspace(x) ne 0 and (not change or WhichSubspace
(x) ne 4) then
                pos, coeff:=ReducedManin(x,MB, NB, IB, N,F,k,l,p,w);
                if pos ne 0 then
                        v[pos]+:=coeff;
                end if;
                end if;
        end for;
        return v;

end function;


function ManinBasis(S,N,F,k,l)
        B:=KMatrixSpace(F,#S,#S)!0;
        for i in [1..#S] do
                B[i]:=ChgManinToCoord(S[i],F,N,k,l);
        end for;
        return B;
end function;

//6.  This section is for operators defined by list of matrices on manin
symbols

function ActOnManin(m,A,B,F);
        r1:=Matrix(F,1,2,[m[1],m[3]])*A;
        r2:=Matrix(F,1,2,[m[2],m[4]])*B;
        return [r1[1,1],r2[1,1],r1[1,2],r2[1,2]];
end function;

function ZeroInLift(L,m);
        for v in L do
                if v[m] ne 0 then return false;
                end if;
        end for;
        return true;
```

```
end function;

function Operator(F,Rep,N,k,l,p,w,Basis,NormalizedBasis, ITBasis,
I1,I2,s1,s2,list,splitlist,type,Lift1,Lift2, compare,O,fast,prime);
        smallerbasis:= compare ne #Rep;


        A:=KMatrixSpace(F,#Rep,#Rep)!0;
        if type eq 2 then TP:=TPList(prime,p,O);
        end if;
        t1:=Cputime();
        for i in [1..#Rep] do
                fast:=true;
                if  not fast or not ZeroInLift(Lift1,i) or not ZeroInLift
(Lift2,i) then
                        m:=Rep[i];

                        if type eq 2 then A[i]:=TPManinRow
(m,I1,I2,s1,s2,p,prime,Basis, NormalizedBasis, ITBasis, N,F,k,l,w,O,TP);
                                //if i mod 100 eq 0 then
                                //      print Round(100*i/#Rep), "percent
on operator construction";
                                //end if;
                        end if;
                        if type eq 1 then
                        for j in [1..#list] do

                                m2:=ActOnManin(m,splitlist[j][1],splitlist
[j][2],F);

                                if (type eq 1 ) and (not smallerbasis or
WhichSubspace(m2) ne 4)  then

                                        pos, mult:=ReducedManin(m2,Basis,
NormalizedBasis, ITBasis, N,F,k,l,p,w);
                                        if pos ne 0 then //This is the
condition for type 1 as in formula.
                                                A[i,pos]+:=mult;
                                        end if;
                                end if;
                        end for;
                        end if;

                end if;
        end for;
        print Cputime()-t1;
```

```
        return A;
end function;

function ProjectOperator(A,Lift1Matrix,Lift2Matrix,
proj1,proj2,F,bmap,N,Rep,MB);

        r1:=#Rows(Lift1Matrix); r2:=#Rows(Lift2Matrix);

        max:=Max([r1,r2]);

        B1:=KMatrixSpace(F,r1,r1)!0;
        B2:=KMatrixSpace(F,r2,r2)!0;


        if r1 ne 0 then
                A1:=Lift1Matrix*A;
        end if;



        if r2 ne 0 then
                A2:=Lift2Matrix*A;
        end if;



        for n in [1..max] do

                if n le r1 then
                        B1[n]:=proj1(A1[n]);
                end if;

                if n le r2 then
                        B2[n]:=proj2(A2[n]);
                end if;

        end for;


        return B1,B2;

end function;

function LiftBasis(B,f,F);
        r:=#B;
        g:=Inverse(f);
```

```
        deg:=Degree(Domain(f));
        A:=KMatrixSpace(F,r,deg)!0;
        for n in [1..r] do
                A[n]:=g(B[n]);
        end for;
        return A, Rows(A);

end function;

//7. This section is for lists of primes

function SplitList(list,s1,s2,F);
        newlist:=[];
        for M in list do
                Append(~newlist,[ChangeRing(M,F,s1),ChangeRing(M,F,s2)]);
        end for;
        return newlist;
end function;

function CDeltaListCreate(prime,Divs, O);

        list:=[];
        t1:=Cputime();
        for x in Divs do
                Q :=quo<O|x>;
                Res := Elements(Q);

                for y in Res do

                        M:=Matrix(O,2,2,[x,y,0,prime div x]);

                        Append(~list,M);

                        while M[1,2] ne 0 do

                                q := M[1,1] div M[1,2];
                                M := M*Matrix(O,2,2,[0,1,1,-q]);
                                        Append(~list,M);

                        end while;
                end for;
        end for;
        print "list at ", Cputime()-t1;
        return list;

end function;
```

```
function CDeltaLists(bound,O,s1,s2,pfirst,p);

        primes := PrimesUpTo(bound);

        if pfirst then
                Exclude(~primes,p);
                Insert(~primes,2,p);
        end if;

        primesabove:=[];
        DeltaLists:=[];
        Display:=[];

        for q in primes do
                Div:=Divisors(O!q);
                prime:=Div[2];
                Append(~primesabove,prime);
                Append(~DeltaLists, CDeltaListCreate(prime,[1,prime], O));
                Append(~Display, q);
                if #Div eq 4 and q ne p then
                        prime:=Div[3];
                        Append(~primesabove,prime);
                        Append(~DeltaLists, CDeltaListCreate(prime,
[1,prime], O));
                        Append(~Display,q);
                end if;
        end for;
        return DeltaLists, Display, primesabove;

end function;

function SplitLists(Lists,s1,s2,F);

        SLs:=[];
        for L in Lists do
                Append(~SLs,SplitList(L,s1,s2,F));
        end for;
        return SLs;

end function;

//8. These are initialization functions

function ModifyManinBasis(B,Rep,N,k,l,w);
```

```
        //basis is too big in this case.
        if ModificationCondition(k,l,w) then
                length:=Nrows(B);
                return Submatrix(B,1,1,length-1,length-1), Prune(Rep),
Prune(N);
        else return B, Rep, N;
        end if;

end function;


function CreateSpace(d,p,bound,pfirst);

        K := QuadraticField(d);
        O<epsilon> := MaximalOrder(K);
        F:=GF(p);
        U,f0:=UnitGroup(O);
        w:=#U;
        D:=Divisors(O!p);
        eta1:=D[2];
        eta2:=Conjugate(D[2]);
        I1:=ideal<O|eta1>;
        I2:=ideal<O|eta2>;
        I3:=ideal<O|O!p>;
        R1,s1:=ResidueClassField(O,I1);
        R2,s2:=ResidueClassField(O,I2);

        //This represents an initial basis for the Eigenspace of Manin
Symbols
        N := [ [i,j] : i,j in [0..p-2] | IsDivisibleBy(i-j,w) ];
        N cat:= [ [i,-1] : i in [0..p-2] | IsDivisibleBy(i-1,w) ];
        N cat:= [ [-1,j] : j in [0..p-2] | IsDivisibleBy(j-1,w) ];
        Include(~N,[-1,-1]);

        //This represents a list whose projections to the eigenspace forms
an initial basis.
        //This will be useful in computing the action of matrices
        f:=PrimitiveRoot(p);
        Rep:=[ [F!f^i,F!a,F!1,F!1] : i in [1..(p-1)/w], a in [0..p-1] ];
        Rep cat:=[ [F!0,F!f^i,F!1,F!1] : i in [1..(p-1)/w] ];
        Rep cat:=[ [F!0,F!1,F!1,F!0] ];

        SpecialList:=CDeltaListCreate(p,[1,eta1,eta2,p], O);
        SplitSpecial:=SplitList(SpecialList,s1,s2,F);
        DLists, Display, Primes:=CDeltaLists(bound,O,s1,s2,pfirst,p);
        SDLists:=SplitLists(DLists,s1,s2,F);
```

```
        return N, Rep, F, p, s1,s2, w, I1,I2, SpecialList, SplitSpecial,
DLists, SDLists, Display, Primes,O;

end function;

function Impose(k,l,N, Rep, F, p,w);

        MB:=ManinBasis(Rep,N,F,k,l);
        MB,Rep,N:= ModifyManinBasis(MB,Rep,N,k,l,w);
        NB:=NormalizeMatrix(MB);
        IB:=Transpose(InvertMatrixVectors(MB));

        return N, Rep, MB, NB, IB;

end function;

//9. These are some miscellaneous functions

function HeckeEigenVal(r,p,k,l,F,s1,s2);

        if r eq 1 or Norm(r) eq p then return F!1;
                else return F!Norm(r) + (F!s1(r))^(2-k)*(F!s2(r))^(2-l);
        end if;

        //return 1;

end function;

function EigenRel(A,ev)

        size:=NumberOfRows(A);
        for n in [1..size] do
                A[n,n]:=A[n,n]-ev;
        end for;
        return A;

end function;

function ChangeMatrixRows(A,f1,f2,f3,F);
        r:=Nrows(A);
        c:=Degree(Codomain(f3));

        B:=KMatrixSpace(F,r,c)!0;

        for n in [1..r] do
```

```
                B[n]:=f3(f2(f1(A[n])));
        end for;

        return B;

end function;

//10. The actual functions used for the program.
//Field=Q(sqrt(-d)); Split prime=p; bound=primes above primes up to this
for the Eisenstein Ideal;
//pfirst=true or false, do U_p operator first?; optionS=do all calculations
through modular symbols;
//so don't use Merel's formula.  This is true/false.

//Currently, Zero relations are not ever taken out.  But they easily could
be.
function ComputeSpaces(d,p,bound,pfirst,optionS);

        NonTrivial:=[];
        Ni, Repi, F, p, s1,s2, w, I1,I2, SpecialList, SplitSpecial, DLists,
SDLists, Display, Primes,O:=CreateSpace(d,p,bound,pfirst);

        for pair in [ [k,l] : k,l in [0..p-2] | IsDivisibleBy(k-l,w)] do
                        k := pair[1];
                        l := pair[2];
                        printf "Eigenspace: %o, %o\n", k, l;
                N, Rep, MB, NB, IB:=Impose(k,l,Ni, Repi, F, p,w);

                BM:=BoundaryMatrix1(Rep,F,k,l,w);

                V:=KSpace(F,#Rep);//The Space of Manin Symbols
                W:=KSpace(F,#Rep);//Will be the kernel of the boundary map
                X:=KSpace(F,3);//The space of cusps

                bmap1:=Hom(V,X)!BM;
                bmap2:=Hom(W,X)!BM;
                W:=Kernel(bmap2);
                X:=sub<X | Image(bmap1)>;

                V, proj1:=quo<V | [V!0]>;
                W, proj2:=quo<W | [W!0]>;
                X, proj3:=quo<X | [X!0]>;

                print "Initial Dimensions:", Dimension(V), Dimension(W),
Dimension(X);
```

```
		B1:=Basis(V);
		B2:=Basis(W);
		B3:=Basis(X);

		Lift1Matrix,Lift1:=LiftBasis(B1,proj1,F);
		Lift2Matrix,Lift2:=LiftBasis(B2,proj2,F);
		Lift3Matrix,Lift3:=LiftBasis(B3,proj3,F);

		if Dimension(V) eq 0 then return 0; end if;

		//Goes through a list of primes
		for i in [1..#Display] do
			if Display[i] eq p then fast:=false;
				else fast:=true;
			end if;

			//Create the Eisenstein relation
			list:=DLists[i];
			splitlist:=SDLists[i];
			if Display[i] ne p and not optionS then
				A:=Operator(F,Rep,N,k,l,p,w,MB,NB, IB,
I1,I2,s1,s2,list,splitlist,1,Lift1,Lift2, #Repi,O, fast,Primes[i]);
			else A:=Operator(F,Rep,N,k,l,p,w,MB,NB, IB,
I1,I2,s1,s2,list,splitlist,2,Lift1,Lift2, #Repi,O, fast,Primes[i]);
			end if;
			A1,A2:=ProjectOperator(A,Lift1Matrix,Lift2Matrix,
proj1,proj2,F,bmap1, N,Rep,MB);

			ev:=HeckeEigenVal(Primes[i],p,k,l,F,s1,s2);
			A1 := EigenRel(A1,ev);
			A2 := EigenRel(A2,ev);

			//quotients out by it
			V,f1:=quo<V | RowSpace(A1)>;
			W,f2:=quo<W | RowSpace(A2)>;
			X,f3:=quo<X | RowSpace(ChangeMatrixRows(A1,Inverse
(proj1),bmap1,proj3,F))>;

			proj1:=proj1*f1;
			proj2:=proj2*f2;
			proj3:=proj3*f3;

			B1:=Basis(V);
			B2:=Basis(W);
			B3:=Basis(X);
```

```
                        Lift1Matrix,Lift1:=LiftBasis(B1,proj1,F);
                        Lift2Matrix,Lift2:=LiftBasis(B2,proj2,F);

                        Lift3Matrix,Lift3:=LiftBasis(B3,proj3,F);


                        printf "Eisenstein Relation for %o: %o, %o, %o \n",
Display[i], Dimension(V), Dimension(W), Dimension(X);

                        if Dimension(V) eq 0 and Dimension(W) eq 0 and
Dimension(X) eq 0 then break;
                        end if;

                end for;

                if Dimension(V) ne 0 then
                        Append(~NonTrivial, [k,l]);
                end if;

        end for;

        return NonTrivial;


end function;

function ComputeSpace(d,p,k,l,bound,pfirst,optionS);

        NonTrivial:=[];
        Ni, Repi, F, p, s1,s2, w, I1,I2, SpecialList, SplitSpecial, DLists,
SDLists, Display, Primes, O:=CreateSpace(d,p,bound,pfirst);

                        printf "Eigenspace: %o, %o\n", k, l;
                N, Rep, MB, NB, IB:=Impose(k,l,Ni, Repi, F, p,w);


                BM:=BoundaryMatrix1(Rep,F,k,l,w);


                V:=KSpace(F,#Rep);
                W:=KSpace(F,#Rep);
                X:=KSpace(F,3);

                bmap1:=Hom(V,X)!BM;
                bmap2:=Hom(W,X)!BM;
```

```
                W:=Kernel(bmap2);
                X:=sub<X | Image(bmap1)>;

                V, proj1:=quo<V | [V!0]>;
                W, proj2:=quo<W | [W!0]>;
                X, proj3:=quo<X | [X!0]>;

                print "Initial Dimensions:", Dimension(V), Dimension(W),
Dimension(X);


                B1:=Basis(V);
                B2:=Basis(W);
                B3:=Basis(X);

                Lift1Matrix,Lift1:=LiftBasis(B1,proj1,F);
                Lift2Matrix,Lift2:=LiftBasis(B2,proj2,F);

                Lift3Matrix,Lift3:=LiftBasis(B3,proj3,F);

                if Dimension(V) eq 0 then return 0; end if;

                //Goes through a list of primes
                for i in [1..#Display] do
                        if Display[i] eq p then fast:=false;
                                else fast:=true;
                        end if;
                        //Create the Eisenstein relation
                        list:=DLists[i];
                        splitlist:=SDLists[i];
                        if Display[i] ne p and not optionS then
                                A:=Operator(F,Rep,N,k,l,p,w,MB,NB, IB,
I1,I2,s1,s2,list,splitlist,1,Lift1,Lift2, #Repi,O, fast,Primes[i]);
                        else A:=Operator(F,Rep,N,k,l,p,w,MB,NB, IB,
I1,I2,s1,s2,list,splitlist,2,Lift1,Lift2, #Repi,O, fast,Primes[i]);
                        end if;

                        //print Rank(A);
                        A1,A2:=ProjectOperator(A,Lift1Matrix,Lift2Matrix,
proj1,proj2,F,bmap1,N,Rep,MB);

                        ev:=HeckeEigenVal(Primes[i],p,k,l,F,s1,s2);
                        A1 := EigenRel(A1,ev);
                        A2 := EigenRel(A2,ev);

                        //quotients out by it
```

```
                        V,f1:=quo<V | RowSpace(A1)>;
                        W,f2:=quo<W | RowSpace(A2)>;
                        X,f3:=quo<X | RowSpace(ChangeMatrixRows(A1,Inverse
(proj1),bmap1,proj3,F))>;

                        proj1:=proj1*f1;
                        proj2:=proj2*f2;
                        proj3:=proj3*f3;

                        B1:=Basis(V);
                        B2:=Basis(W);
                        B3:=Basis(X);

                        Lift1Matrix,Lift1:=LiftBasis(B1,proj1,F);
                        Lift2Matrix,Lift2:=LiftBasis(B2,proj2,F);

                        Lift3Matrix,Lift3:=LiftBasis(B3,proj3,F);

                        printf "Eisenstein Relation for %o: %o, %o, %o \n",
Display[i], Dimension(V), Dimension(W), Dimension(X);

                        if Dimension(V) eq 0 and Dimension(W) eq 0 and
Dimension(X) eq 0 then break;
                        end if;
                end for;


        return V,W,X;


end function;
```