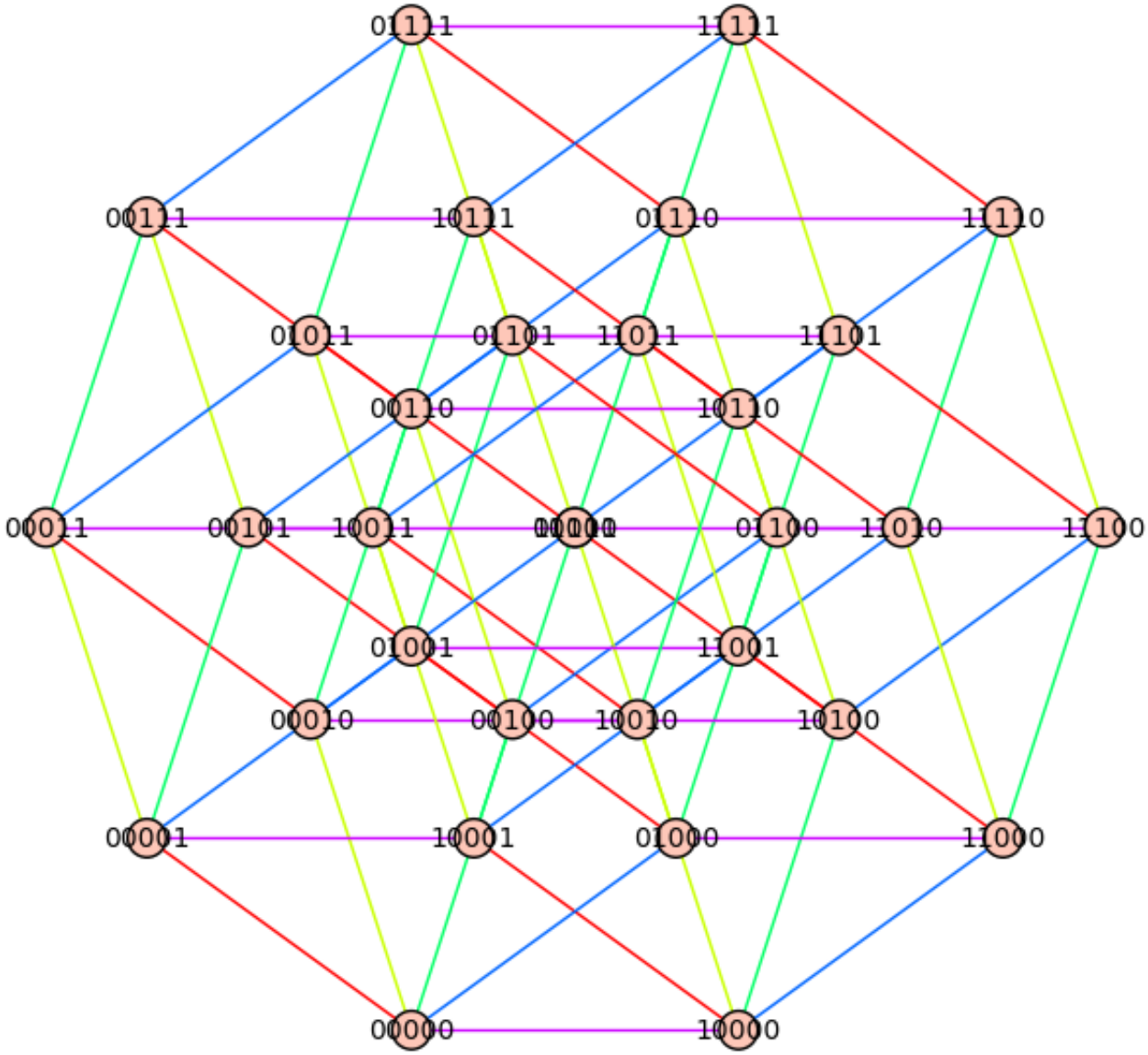
Copy of dessin

```
# insert your generator matrix except if k=0 leave as []
generator=[(1,1,1,1, 0)]
if generator!=[]:
    n=len(generator[0])
else:
    # if k=0 then insert n
    n=4
k=len(generator)
dash=[1, 2, 5, 10]
direction_black=[2, 3, 4, 5, 6, 10]
print [n,k],'code'
print generator

print 'Genus',1+2^(n-k-1)*(n/4-1)
print ''

# creates colored N-cube
from sage.plot.colors import rainbow
C = graphs.CubeGraph(n)
R = rainbow(n)
edge_colors = {}
for i in range(n):
   edge_colors[R[i]] = []
C.allow_multiple_edges(False)
for u,v,l in C.edges():
   for i in range(n):
       if u[i] != v[i]:
           C.set_edge_label(u,v,i+1)
C.plot(edge_colors=C._color_by_label()).show()
```

```
[5, 1] code
[(1, 1, 1, 1, 0)]
Genus 3
```




```
len(C.edges())
```

```
4
```

```
# creates dictionary to get vertex from equivalent tuple
dict={}
for i in C.vertices():
    word=[]
    for j in i:
        word.append(int(j))
    dict[tuple(word)]=i

V=GF(2)^n

if generator!=[]:
```

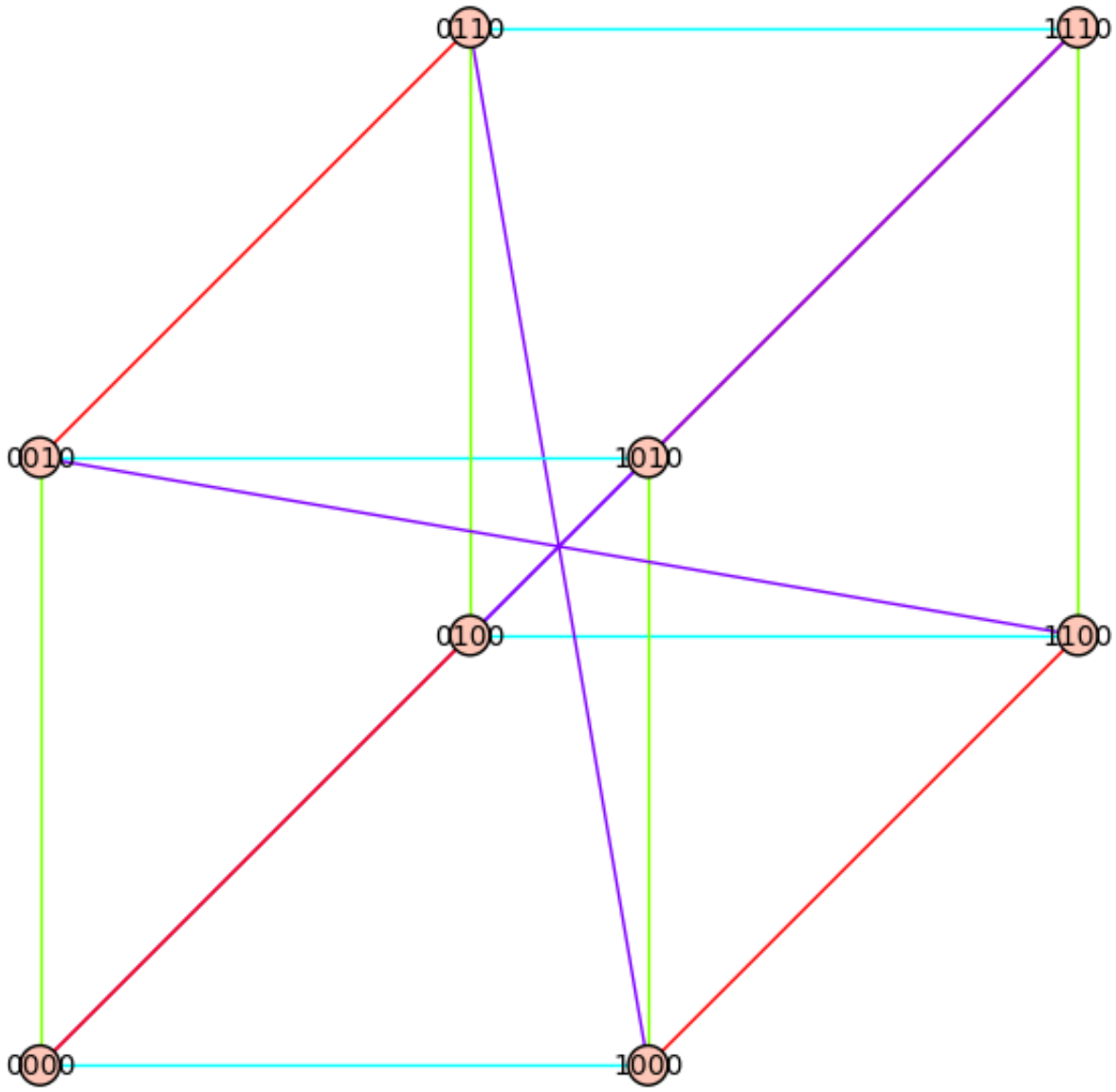
```
    # creates code from generator matrix
    MS = MatrixSpace(GF(2),k,n)
    gen = MS(generator)
    code= LinearCode(gen)

    # creates the equivalence class of vertices in N-cube
    eqclass=[]
    for i in V:
        if len(eqclass)==2^(n-k):
            break
        used='n'
        # check if class is done already
        for j in eqclass:
            if i in j:
                used ='y'
                break
        if used=='y':
            continue
        else:
            # add new class
            list=[]
            for l in code:
                list.append(i+l)
            eqclass.append(list)
    #print 'equivalence classes'
    #for i in eqclass:
        #print i
    #print ''

    # merge all vertices in an equivalence class
    for i in eqclass:
        for j in range(1,2^k):

C.merge_vertices([dict.get(tuple(i[0])),dict.get(tuple(i[j]))])
    print 'chromotoplology'
    for u,v,l in C.edges():
        edge_colors[R[l-1]].append((u,v,l))
    C.plot(edge_colors=C._color_by_label()).show()
    print ''
    # for dimension 0 codes
else:
    eqclass=[]
    for i in V:
        eqclass.append([i])
```

chromotoplology




```
#label edges 1 to d
#print 'edges'
count=0
for i in eqclass:
    if i[0].hamming_weight()%2==0:
        for u,v,l in C.edges():
            if u==dict.get(tuple(i[0])) or v==dict.get(tuple(i[0])):
                C.set_edge_label(u,v,l+count)
        count+=n
#print C.edges()
#print ''
```

```
# create list of pi(v,black)
sigb=[]
for i in eqclass:
    if i[0].hamming_weight()%2==0:
        pi=[]
        for j in range(1,n+1):
            for u,v,l in C.edges_incident([dict.get(tuple(i[0]))]):
                if l%n==j%n:
                    pi.append(l)
        sigb.append(tuple(pi))

# create list of pi(v,white)
sigw=[]
for i in eqclass:
    if i[0].hamming_weight()%2==1:
        pi=[]
        for j in range(n,0,-1):
            for u,v,l in C.edges_incident([dict.get(tuple(i[0]))]):
                if l%n==j%n:
                    pi.append(l+n*2^(n-1))
        sigw.append(tuple(pi))

# create list of tau
tau_list=[]
for i in range(1,len(C.edges())+1):
    tau_list.append((i,i+n*2^(n-1)))

# create sigma1 and sigma2 and tau
S=SymmetricGroup(n*2^(n))
sigma1=S(sigb)
sigma2=S(sigw)
tau=S(tau_list)
print 'sigma white'
print sigma1
print ''
print 'sigma black'
print sigma2
print ''
print 'tau'
print tau
print ''
print 'Pi Infinity'
print sigma2*sigma1*tau
print 'sigma 1'
print sigma1
```

```
print 'sigma 2'
print tau*sigma2*tau
print 'Sigma Infinty'
print tau*sigma2*tau*sigma1
```

```
sigma white
(1,2,3,4)(5,6,7,8)(9,10,11,12)(13,14,15,16)

sigma black
(33,48,43,38)(34,37,44,47)(35,46,41,40)(36,39,42,45)

tau
(1,33)(2,34)(3,35)(4,36)(5,37)(6,38)(7,39)(8,40)(9,41)(10,42)(11,43)
(12,44)(13,45)(14,46)(15,47)(16,48)

Pi Infinity
(1,34,5,38)(2,35,14,47)(3,36,7,40)(4,33,16,45)(6,39,10,43)(8,37,12,4
1)(9,42,13,46)(11,44,15,48)
sigma 1
(1,2,3,4)(5,6,7,8)(9,10,11,12)(13,14,15,16)
sigma 2
(1,16,11,6)(2,5,12,15)(3,14,9,8)(4,7,10,13)
Sigma Infinty
(1,13)(2,6)(3,15)(4,8)(5,9)(7,11)(10,14)(12,16)
```

```
Face=[]
true='y'
for i in range(1,len(C.edges())+1):
    temp=[]
    temp.append(i)
    temp.append(tau(sigma2(tau(i))))
    temp.append(sigma1(tau(sigma2(tau(i)))))
    temp.append(tau(sigma2(tau(sigma1(tau(sigma2(tau(i))))))))
    for j in Face:
        num=0
        for k in temp:
            if k in j:
                num+=1
        if num==4:
            true='n'
            break
    if true=='y':
        Face.append(temp)
    true='y'
print Face
```

```
[[1, 4, 3, 2]]
```

```
for i in eqclass:
    if i[0].hamming_weight()%2==0:
        print i
```

```
[(0, 0)]
[(1, 1)]
```

```
A1=[]
for i in eqclass:
    if i[0].hamming_weight()%2==0:
        B1=[]
        for j in range(1,len(C.edges())+1):
            for k in C.edges():
                if j==k[2]:
                    if dict.get(tuple(i[0])) in k[0] or
dict.get(tuple(i[0])) in k[1]:
                        B1.append(1)
                    else:
                        B1.append(0)
    if B1 not in A1:
        A1.append(B1)
for i in eqclass:
    if i[0].hamming_weight()%2!=0:
        B1=[]
        for j in range(1,len(C.edges())+1):
            for k in C.edges():
                if j==k[2]:
                    if dict.get(tuple(i[0])) in k[0] or
dict.get(tuple(i[0])) in k[1]:
                        B1.append(1)
                    else:
                        B1.append(0)
    if B1 not in A1:
        A1.append(B1)
print A1
```

```
[[1, 1, 0, 0], [0, 0, 1, 1], [1, 0, 0, 1], [0, 1, 1, 0]]
```

```
A2=[]
for i in Face:
    B2=[]
    for j in range(1,len(C.edges())+1):
        if j in i:
            B2.append(1)
        else:
            B2.append(0)
    A2.append(B2)
```

```
print A2
```

```
[[1, 1, 1, 1]]
```

```
M1=MatrixSpace(GF(2),len(C.vertices()),len(C.edges()))
H1=M1(A1)
H1
```

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

```
V1=H1.row_space()
```

```
perp=V1.basis_matrix().right_kernel()
```

```
perp
```

$$\mathrm{RowSpan}_{\mathbf{F}_2} \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{pmatrix}$$

```
M2=MatrixSpace(GF(2),len(C.edges())/2,len(C.edges()))
H2=M2(A2)
H2
```

```
Traceback (click to the left of this block for traceback)
...
TypeError: cannot construct an element of Full MatrixSpace of 2 by 4
dense matrices over Finite Field of size 2 from [[1, 1, 1, 1]]!
```

```
M3=MatrixSpace(GF(2),38,len(C.edges()))
HH1=H1.row_space().basis()
HH2=H2.row_space().basis()
HH=HH1+HH2
M3(HH),H1,H2
```

```
Traceback (click to the left of this block for traceback)
...
TypeError: cannot construct an element of Full MatrixSpace of 38 by
16 dense matrices over Finite Field of size 2 from [(1, 0, 0, 0, 0,
1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1), (0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0,
```

```
    1, 0, 0, 1, 0), (0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0),
    (0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1), (0, 0, 0, 0, 1, 1,
    1, 1, 0, 0, 0, 0, 0, 0, 0, 0), (0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1,
    0, 0, 0, 0), (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1), (1,
    0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1), (0, 1, 0, 1, 0, 0, 1,
    1, 0, 0, 0, 0, 0, 1, 1, 0), (0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0,
    0, 0, 0), (0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0), (0, 0,
    0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1), (0, 0, 0, 0, 0, 0, 0, 0,
    1, 1, 0, 0, 1, 1, 0, 0), (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0,
    1, 1)]!
```

```
F4.<omega>=GF(4)
```

```
Q1=[]
for i in H1:
    temp=[]
    for j in range(len(i)):
       if j in dash:
           temp.append(i[j]*omega)
       else:
           temp.append(i[j])
    Q1.append(temp)
Q2=[]
for i in H2:
    temp=[]
    for j in range(len(i)):
       if j in dash:
           temp.append(i[j]*omega^2)
       else:
           temp.append(i[j])
    Q2.append(temp)
```

```
R1=[]
used=[]
for i in Q1:
    temp=[]
    for j in range(len(i)):
       if j in direction:
           if i[j]!=0:
               if j not in used:
                   temp.append(omega^2)
                   used.append(j)
               else:
                   temp.append(i[j])
```

```
            else:
                temp.append(0)
        else:
            temp.append(i[j])
    R1.append(temp)
R2=[]
for i in Q2:
    temp=[]
    for j in range(len(i)):
        if j in direction:
            if i[j]!=0:
                if j not in used:
                    temp.append(i[j])
                    used.append(j)
                else:
                    temp.append(omega)
            else:
                temp.append(0)
        else:
            temp.append(i[j])
    R2.append(temp)
```

```
Traceback (click to the left of this block for traceback)
...
NameError: name 'direction' is not defined
```

```
print R1
print R2
```

```
[]
Traceback (click to the left of this block for traceback)
...
NameError: name 'R2' is not defined
```

```
def AdditiveCodeGenMat(H1,H2):
    B1=[]
    for i in H1:
        B1.append(i)
    B1.pop()
    B2=[]
    for i in H2:
        B2.append(i)
    B2.pop()
    W1=[omega*v for v in B1]
    W2=[omega^2*v for v in B2]
    W=W1+W2
    FMS=MatrixSpace(F4,len(W),len(H1.columns()))
    G=FMS(W)
```

```
    return G
```

```
def AdditiveCodeGenMatDash():
    Q1.pop()
    Q2.pop()
    Q=Q1+Q2
    FMS=MatrixSpace(F4,len(Q),len(H1.columns()))
    G=FMS(Q)
    return G
```

```
AA=AdditiveCodeGenMat(H1,H2)
```

```
AdditiveCodeGenMatDash()
```

$$\begin{pmatrix}
1 & \omega & \omega & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & \omega & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & \omega & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 0 & \omega & 0 & 0 & 0 & 0 & \omega & 0 & 0 & 0 & 0 & 1 \\
0 & \omega & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & \omega & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
1 & \omega+1 & 0 & 0 & 1 & \omega+1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \omega+1 & \omega+1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & \omega+1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \omega+1 & 1 & 0 & 0 & 1 & \omega+1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0
\end{pmatrix}$$

```
R1=[]
R2=[]
```

```
def AdditiveCodeGenMatDashOrient():

    count=0
    for i in Q1:
        temp=[]
        if count<len(Q1)/2:
            for j in range (len(i)):
                if j in direction_black:
                    temp.append(i[j]*omega)
                else:
                    temp.append(i[j])
```

```
            R1.append(temp)
            count+=1
        else:
            for j in range (len(i)):
                if j not in direction_black:
                    temp.append(i[j]*omega)
                else:
                    temp.append(i[j])
            R1.append(temp)

    for i in Q2:
        temp=[]
        for j in range(len(i)):
            if j in direction_black:
                temp.append(i[j]*omega^2)
            else:
                temp.append(i[j])
        R2.append(temp)
    R=R1+R2
    FMS=MatrixSpace(F4,len(R),len(H1.columns()))
    G=FMS(R)
    return G
```

```
Z=AdditiveCodeGenMatDashOrient()
Z
```

$$\begin{pmatrix}
1 & \omega & \omega+1 & \omega & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \omega & \omega+1 & \omega & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & \omega+1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
\omega & 0 & 0 & 0 & 0 & \omega & 0 & 0 & 0 & 0 & \omega & 0 & 0 & 0 & 0 & \omega \\
0 & \omega+1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \omega & 0 & 0 & \omega & 0 \\
0 & 0 & \omega & 0 & 0 & 0 & 0 & \omega & \omega & 0 & 0 & 0 & 0 & \omega & 0 & 0 \\
1 & 0 & 0 & \omega+1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
1 & \omega+1 & 0 & 0 & \omega+1 & \omega & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \omega+1 & \omega & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & \omega & \omega+1 & 0 & 0 & \omega+1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \omega+1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \omega & \omega+1 & 0 & 0 & 1 & \omega & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0
\end{pmatrix}$$

```
FMS1=MatrixSpace(F4,len(R1),len(H1.columns()))
FMS2=MatrixSpace(F4,len(R2),len(H1.columns()))
MM=FMS1(R1)
NN=FMS2(R2)
```

```
orth='y'
for i in MM:
    if orth=='n':
        break
    else:
        for j in NN:
            if i*j!=0:
                orth='n'
                print i, j
                break
if orth=='y':
    print 'orthogonal'
else:
    print 'not orthogonal'
```

```
(omega, 0, 0, 0, 0, omega, 0, 0, 0, 0, omega, 0, 0, 0, 0, omega) (1,
omega + 1, 0, 0, omega + 1, omega, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
not orthogonal
```

```
Z.row_space().basis()
```

$$[(1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0), (0, 1, 0, 0, 0, 0, 0, \omega + 1, 0, 0, 0, 0, 0, 0, 0, 0), (0, 0, 1$$

```
M3=MatrixSpace(F4,len(Q1),len(Q1[1]))
Z=M3(R1)
M4=MatrixSpace(F4,len(Q2),len(Q2[1]))
ZZ=M4(R2)
orth='y'
for i in Z:
    if orth=='y':
        for j in ZZ:
            if i*j!=0:
                orth='n'
                break
if orth=='y':
```

```
Traceback (click to the left of this block for traceback)
...
IndentationError: expected an indented block
```

```
omega*omega^2
```

```
1
```

```
M3=MatrixSpace(F4,len(Q1),len(Q1[1]))
M4=MatrixSpace(F4,len(Q2),len(Q1[2]))
Q11=M3(Q1)
Q22=M4(Q2)
orth='y'
```

```
for i in Q11.row_space().basis():
    for j in Q22.row_space().basis():
       if i*j!=0:
           orth='n'
           break
    if orth=='n':
        break
if orth=='y':
    print 'orthogonal'
```

```
orthogonal
```