This document explains the details of the MatLab code that generated the tables of gaps in Section 5 and also contains a lot of the code used to make other figures in the paper. The first section contains a list of all the functions and a brief description of what they do. The second section contains the actual code and more details of how everything works. Section 2.3 contains some code that I have not debugged as carefully. I have tried my best to make all of my code as user-friendly as possible, but please do not hesitate to email me with any questions or if you would like a copy of any of the code in a different format.

1. Description of the Functions

1.1. **General Functions.** The following are some functions not specific to calculating ratios in gaps. The algorithm is described as well for some of them.

(1) The function **drawSierpinski(list)** takes in a list of 2's and 3's and draws a picture of the corresponding gasket.
(2) The function **fromBinary(x)** takes in a list of 0's and 1's and converts the corresponding binary number to a base 10.
(3) The function **g(x,list)** approximates the function g_m of the paper. x is a number and 'list' specifies what b_{m+1}, $b_{m+2}\ldots$ are. This function does not have the renormalization factor from what happened at levels 1, $2, \ldots m$.
(4) The function **ginv(y,list)** computes the inverse to the function g.
(5) The function **R1(x)** is the spectral decimation function for SG_2. That is, if you input λ_{m+1}, it return λ_m.
(6) The function **R1inv(y)** is the 2-valued inverse to $R1$.
(7) The function **R2(x)** is the spectral decimation function for SG_3.
(8) The function **R2inv(y)** is the 4-valued inverse to $R2$. This was computed using Mathematica.
(9) The function **R11inv(y)** is the smallest inverse to $R1$.
(10) The function **R21inv(y)** is the smallest inverse to $R2$.
(11) The function **spectrum(x,values,start)** computes the (unnormalized) spectrum of the discrete Laplacian corresponding to the list x. If the variables 'values' and 'start' are specified, the spectrum is computed assuming we start at position 'start' in x with eigenvalues 'values' at that level.
(12) The function **toBinary(x,m)** converts a natural number x to a binary number with at least m digits. It returns a list of 0's and 1's.

(13) The function **Vertices(x)** takes in a list of 2's and 3's and returns the number of triangles, vertices, and cycles in the corresponding gasket.

1.2. **Functions for Computing Gaps.**

(1) The function **CGL(intervals, which, cutoff)** takes in two lists of intervals (in the cells array 'intervals') corresponding to eigenvalues of Δ_m, approximates the corresponding eigenvalues of Δ assuming only **R11inv** and **R21inv** are applied (which ones are applied is specified in 'which') and then computes all ratios of these eigenvalues. Ratios bigger than a certain value are discarded.
(2) The function **consolidateIntervals(x)** takes in a list of closed intervals and eliminates repeats and overlaps.
(3) The function **eigenGaps(list)** gives a covering eigenvalues 'coming' from a certain level n, where list specifies $b_n, b_{n-1}, \ldots$.
(4) The function **findGaps(b)** inputs a list of intervals and finds any gaps in the covering. It assumes the inputs are sorted (this could be changed).
(5) The function **intersectList(x,y)** inputs two lists of intervals and finds the intersection of the corresponding covers. The code for this is inefficient and a new version is included below.

Table 2 in section 5 of the paper was computed with the following MatLab command:

```
x=findGaps(CGL(eigenGaps([3 3 3 3]),[3 3 3 3 3 3],13))
```

Table 3 in section 5 of the paper was computed using the script that follows. This script is not as efficient as it could be, because the computations are not that intensive. I played around with the code and while I was not able to show there exists gaps in the ratios of eigenvalues from an arbitrary $HH(b)$, I do believe that stronger results than the ones given in our paper could be formulated (I had vague ideas of what sequences of 2's and 3's were preventing gaps from being found and vague ideas of how to focus the computer power on these particular sequences–please contact me if you would like to discuss further).

```
clear
x{1}=[2 3 2 3 2 3];
x{2}=[2 3 2 3 3 2];
x{3}=[2 3 3 2 2 3];
x{4}=[2 3 3 2 3 2];
x{5}=[3 2 3 2 3 2];
x{6}=[3 2 3 2 2 3];
x{7}=[3 2 2 3 2 3];
x{8}=[3 2 2 3 3 2];
```

```
y{1}=[2 3 2 3 2 3];
y{2}=[2 3 2 3 3 2];
y{3}=[2 3 3 2 2 3];
y{4}=[2 3 3 2 3 2];
y{5}=[3 2 3 2 3 2];
y{6}=[3 2 3 2 2 3];
y{7}=[3 2 2 3 2 3];
y{8}=[3 2 2 3 3 2];



for i=1:8
    temp=eigenGaps(x{i});
    for j=1:8
        disp([i j]);
        g{i,j}=findGaps(CGL(temp,y{j},13));
    end
end

%For each of 8 possible starts, find intersection over all 8 possible g functions.
for i=1:8
current=intersectList(g{i,1},g{i,2});
for j=3:8
disp([2 i j]);
current=intersectList(current,g{i,j});
end
intg{i}=current;
end

%Now intersect over all 8 possible starts
temp=intg{1};
for i=2:8
i
temp=intersectList(temp,intg{i});
end

temp
```

1.3. **Weyl Ratio.** At the time this document was written, it had been a long time since I had thought about the Weyl Ratio plots in the paper. The following functions are the MatLab code that was used to create them (I believe). I will try to add more details to this section. The four functions listed here and the function **spectrum** should be correct (at one point I looked over them very carefully), but I'm not as sure of them as I am of the ones used to compute the tables in Section 5 of the paper.

(1) The function **count(x,y)** inputs an $n \times 2$ matrix x consisting of ordered pairs of eigenvalues and their multiplicities and a list y of values and returns the number of eigenvalues (counting multiplicity) less than each element of y. It the input y is not specified, the first column of x is used.

(2) **fractalCounting(x,list,list2)**

(3) The function **plotWeylRatio(list,numDataPoints,needData)** inputs a list of 2's and 3's and the number of data points desired and graphs the corresponding Weyl Ratio function (as in Section 4 of the paper). I believe the variable 'needData' is no longer used.
(4) The function **pm(x)** returns the number of 2's and 3's respectively in a list x of 2's and 3's.
(5) The function **weyl(x,list,list2)** computes the Weyl Ratio at x for b equal to 'list' (see paper for notation). If an input 'list2' of 2's and 3's is given, it is used to compute **ginv**, otherwise it is assumed that **ginv** is the identity (once again, see paper for why we can assume this).

1.4. **Additional Code.** This code has been debugged, but not as thoroughly as the code above.

(1) This script orders the function **g(x,list)** for all 32 possible inputs of list that are of length 5. This script could easily be modified to deal with lists of length different from 5.
(2) This is a newer version of function **CGL**. At first glance it appears to be the same as "older" version, which makes me think the actual older version no longer exists.
(3) The function **newintersectList(x,y)** is a more efficient version of the function **intersectList(x,y)**.
(4) This script (or some variation of it) was used to produce figure 4.4 of the paper.

2. CODE

2.1. **General Functions.** Here is the matlab code of the functions described above.

2.1.1. *drawSierpinski.*

```
%Given a list of 2's and 3's, this function draws the corresponding gasket.
function z=drawSierpinski(list)
clf;

pts=[.5 sqrt(3)/2; 1 0;0 0;.5 sqrt(3)/2];

for i=1:length(list)

    if(list(i)==2)
        hpts1=pts(:,1)*.5;
        hpts2=pts(:,2)*.5;
        pts=[pts(1,:);[hpts1 hpts2]; [(hpts1+.5) hpts2];[(hpts1+.25) (hpts2+(sqrt(3)/4))]];
    elseif(list(i)==3)
        hpts1=pts(:,1)*(1/3);
        hpts2=pts(:,2)*(1/3);
        pts=[pts(1,:);[hpts1 hpts2];[(hpts1+1/3) hpts2];[(hpts1+2/3) hpts2];[(hpts1+.5)%continued on next line
```

```
         (hpts2+(sqrt(3)/6))];  [(hpts1+1/6) (hpts2+(sqrt(3)/6))];[(hpts1+1/3) (hpts2+(sqrt(3)/3))]];
    else
        error('Invalid input');
    end
end
```

2.1.2. *fromBinary.*

```
%Converts a list of 0's and 1's representing the binary expansion of a number to the number it represents.
function z=fromBinary(x)

n=length(x);
z=sum(x.*(2.^((n-1):-1:0)));
```

2.1.3. *g.*

```
%Approximates the function $g_m$ of the paper.
function z=g(x,list)

renormalize=1;
for i=1:length(list)
    if(list(i)==2)
        renormalize=renormalize*5;
        x=R11inv(x);
    else
        renormalize=renormalize*(90/7);
        x=R21inv(x);
    end
end

z=x*renormalize;
```

2.1.4. *ginv.*

```
%Computes the inverse to the function g.
function z=ginv(x,list)

renormalize=1;

for i=1:length(list)
    if(list(i)==2)
        renormalize=renormalize*5;
    else
        renormalize=renormalize*(90/7);
    end
end

x=x/renormalize;

for i=length(list):-1:1
    if(list(i)==2)
        x=R1(x);
    else
        x=R2(x);
    end
end

z=x;
```

2.1.5. *R1.*

```
%The spectral decimation equation for SG_2 (that is,
%\lambda_{m}=R1(\lambda_{m+1})
function z=R1(x)
z=(5+(-1).*x).*x;
```

2.1.6. *R1inv.*

```
%The inverse to R1 (2-valued)
function z=R1inv(y)
z=[(1/2).*(5+(-1).*sqrt(25+(-4).*y));(1/2).*(5+sqrt(25+(-4).*y))];
```

2.1.7. *R2.*

```
%The spectral decimation equation for SG_3 (that is,
%\lambda_{m}=R2(\lambda_{m+1})
function z=R2(x)
z=3.*((-5)+x).*((-4)+x).*((-3)+x).*x.*((-14)+3.*x).^(-1);
```

2.1.8. *R2inv.*

```
%The inverse to R2 (4-valued).  Computed in Mathematica.
function z=R2inv(y)
temp=[3+(-1/2).*sqrt((14/3)+(1/27).*2.^(1/3).*(441+180.*y).*(286+516.* ...\n  \
y+27.*y.^2+sqrt((-388800)+(-281088).*y+46500.*y.^2+(-4136).*y.^3+ ...\n  \
729.*y.^4)).^(-1/3)+(1/3).*2.^(-1/3).*(286+516.*y+27.*y.^2+sqrt(( ...\n  \
-388800)+(-281088).*y+46500.*y.^2+(-4136).*y.^3+729.*y.^4)).^(1/3) ...\n  \
)+(-1/2).*sqrt((28/3)+(-1/27).*2.^(1/3).*(441+180.*y).*(286+516.* ...\n  \
y+27.*y.^2+sqrt((-388800)+(-281088).*y+46500.*y.^2+(-4136).*y.^3+ ...\n  \
729.*y.^4)).^(-1/3)+(-1/3).*2.^(-1/3).*(286+516.*y+27.*y.^2+sqrt(( ...\n  \
-388800)+(-281088).*y+46500.*y.^2+(-4136).*y.^3+729.*y.^4)).^(1/3) ...\n  \
+(-1/4).*((-528)+(-8).*((-60)+(-1).*y)).*((14/3)+(1/27).*2.^(1/3) ...\n  \
.*(441+180.*y).*(286+516.*y+27.*y.^2+sqrt((-388800)+(-281088).*y+ ...\n  \
46500.*y.^2+(-4136).*y.^3+729.*y.^4)).^(-1/3)+(1/3).*2.^(-1/3).*( ...\n  \
286+516.*y+27.*y.^2+sqrt((-388800)+(-281088).*y+46500.*y.^2+( ...\n  \
-4136).*y.^3+729.*y.^4)).^(1/3)).^(-1/2));3+(-1/2).*sqrt((14/3)+( ...\n  \
1/27).*2.^(1/3).*(441+180.*y).*(286+516.*y+27.*y.^2+sqrt((-388800) ...\n  \
+(-281088).*y+46500.*y.^2+(-4136).*y.^3+729.*y.^4)).^(-1/3)+(1/3) ...\n  \
.*2.^(-1/3).*(286+516.*y+27.*y.^2+sqrt((-388800)+(-281088).*y+ ...\n  \
46500.*y.^2+(-4136).*y.^3+729.*y.^4)).^(1/3))+(1/2).*sqrt((28/3)+( ...\n  \
-1/27).*2.^(1/3).*(441+180.*y).*(286+516.*y+27.*y.^2+sqrt(( ...\n  \
-388800)+(-281088).*y+46500.*y.^2+(-4136).*y.^3+729.*y.^4)).^( ...\n  \
-1/3)+(-1/3).*2.^(-1/3).*(286+516.*y+27.*y.^2+sqrt((-388800)+( ...\n  \
-281088).*y+46500.*y.^2+(-4136).*y.^3+729.*y.^4)).^(1/3)+(-1/4).*( ...\n  \
(-528)+(-8).*((-60)+(-1).*y)).*((14/3)+(1/27).*2.^(1/3).*(441+ ...\n  \
180.*y).*(286+516.*y+27.*y.^2+sqrt((-388800)+(-281088).*y+46500.* ...\n  \
y.^2+(-4136).*y.^3+729.*y.^4)).^(-1/3)+(1/3).*2.^(-1/3).*(286+ ...\n  \
516.*y+27.*y.^2+sqrt((-388800)+(-281088).*y+46500.*y.^2+(-4136).* ...\n  \
y.^3+729.*y.^4)).^(1/3)).^(-1/2));3+(1/2).*sqrt((14/3)+(1/27).* ...\n  \
2.^(1/3).*(441+180.*y).*(286+516.*y+27.*y.^2+sqrt((-388800)+( ...\n  \
-281088).*y+46500.*y.^2+(-4136).*y.^3+729.*y.^4)).^(-1/3)+(1/3).* ...\n  \
2.^(-1/3).*(286+516.*y+27.*y.^2+sqrt((-388800)+(-281088).*y+ ...\n  \
46500.*y.^2+(-4136).*y.^3+729.*y.^4)).^(1/3))+(-1/2).*sqrt((28/3)+ ...\n  \
(-1/27).*2.^(1/3).*(441+180.*y).*(286+516.*y+27.*y.^2+sqrt(( ...\n  \
-388800)+(-281088).*y+46500.*y.^2+(-4136).*y.^3+729.*y.^4)).^( ...\n  \
-1/3)+(-1/3).*2.^(-1/3).*(286+516.*y+27.*y.^2+sqrt((-388800)+( ...\n  \
-281088).*y+46500.*y.^2+(-4136).*y.^3+729.*y.^4)).^(1/3)+(1/4).*(( ...\n  \
-528)+(-8).*((-60)+(-1).*y)).*((14/3)+(1/27).*2.^(1/3).*(441+180.* ...\n  \
y).*(286+516.*y+27.*y.^2+sqrt((-388800)+(-281088).*y+46500.*y.^2+( ...\n  \
-4136).*y.^3+729.*y.^4)).^(-1/3)+(1/3).*2.^(-1/3).*(286+516.*y+ ...\n  \
```

```
27.*y.^2+sqrt((-388800)+(-281088).*y+46500.*y.^2+(-4136).*y.^3+ ...\n  \
729.*y.^4)).^(1/3)).^(-1/2));3+(1/2).*sqrt((14/3)+(1/27).*2.^(1/3) ...\n  \
.*(441+180.*y).*(286+516.*y+27.*y.^2+sqrt((-388800)+(-281088).*y+ ...\n  \
46500.*y.^2+(-4136).*y.^3+729.*y.^4)).^(-1/3)+(1/3).*2.^(-1/3).*( ...\n  \
286+516.*y+27.*y.^2+sqrt((-388800)+(-281088).*y+46500.*y.^2+( ...\n  \
-4136).*y.^3+729.*y.^4)).^(1/3))+(1/2).*sqrt((28/3)+(-1/27).*2.^( ...\n  \
1/3).*(441+180.*y).*(286+516.*y+27.*y.^2+sqrt((-388800)+(-281088) ...\n  \
.*y+46500.*y.^2+(-4136).*y.^3+729.*y.^4)).^(-1/3)+(-1/3).*2.^( ...\n  \
-1/3).*(286+516.*y+27.*y.^2+sqrt((-388800)+(-281088).*y+46500.* ...\n  \
y.^2+(-4136).*y.^3+729.*y.^4)).^(1/3)+(1/4).*((-528)+(-8).*((-60)+ ...\n  \
(-1).*y)).*((14/3)+(1/27).*2.^(1/3).*(441+180.*y).*(286+516.*y+ ...\n  \
27.*y.^2+sqrt((-388800)+(-281088).*y+46500.*y.^2+(-4136).*y.^3+ ...\n  \
729.*y.^4)).^(-1/3)+(1/3).*2.^(-1/3).*(286+516.*y+27.*y.^2+sqrt(( ...\n  \
-388800)+(-281088).*y+46500.*y.^2+(-4136).*y.^3+729.*y.^4)).^(1/3) ...\n  \
).^(-1/2))];

z=real(temp);
```

2.1.9. *R11inv.*

```
%Gives the smallest inverse to R1
function z=R11inv(x)

z=.5*(5-sqrt(25-4*x));
```

2.1.10. *R21inv.*

```
%Gives the smallest inverse to R2
function z=R21inv(y)

z=real((3+(-1/2).*sqrt((14/3)+(1/27).*2.^(1/3).*(441+180.*y).*(286+516.* ...\n  \
y+27.*y.^2+sqrt((-388800)+(-281088).*y+46500.*y.^2+(-4136).*y.^3+ ...\n  \
729.*y.^4)).^(-1/3)+(1/3).*2.^(-1/3).*(286+516.*y+27.*y.^2+sqrt(( ...\n  \
-388800)+(-281088).*y+46500.*y.^2+(-4136).*y.^3+729.*y.^4)).^(1/3) ...\n  \
)+(-1/2).*sqrt((28/3)+(-1/27).*2.^(1/3).*(441+180.*y).*(286+516.* ...\n  \
y+27.*y.^2+sqrt((-388800)+(-281088).*y+46500.*y.^2+(-4136).*y.^3+ ...\n  \
729.*y.^4)).^(-1/3)+(-1/3).*2.^(-1/3).*(286+516.*y+27.*y.^2+sqrt(( ...\n  \
-388800)+(-281088).*y+46500.*y.^2+(-4136).*y.^3+729.*y.^4)).^(1/3) ...\n  \
+(-1/4).*((-528)+(-8).*((-60)+(-1).*y)).*((14/3)+(1/27).*2.^(1/3) ...\n  \
.*(441+180.*y).*(286+516.*y+27.*y.^2+sqrt((-388800)+(-281088).*y+ ...\n  \
46500.*y.^2+(-4136).*y.^3+729.*y.^4)).^(-1/3)+(1/3).*2.^(-1/3).*( ...\n  \
286+516.*y+27.*y.^2+sqrt((-388800)+(-281088).*y+46500.*y.^2+( ...\n  \
-4136).*y.^3+729.*y.^4)).^(1/3)).^(-1/2))));
```

2.1.11. *spectrum.*

```
%Given a list x of 2's and 3's, spectrum computes the spectrum of
%eigenvalues for this approximation
function z=spectrum(x,values,start)
n=length(x);

if(nargin==3)
    z=values;
elseif(x(1)==2)
    z=[2 1;5 2];
    start=2;
else
    z=[(3-sqrt(5)) 1; (3+sqrt(5)) 1; 3 2; 5 2; 6 1];
    start=2;
end
```

```
for i=start:length(x)

    if(x(i)==2)
        z=[NewNextLevel2(x(1:i-1));NextLevel2(z)];
    else
        z=[NewNextLevel3(x(1:i-1));NextLevel3(z)];
    end
end
z=sortrows(z);


%Helper functions%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Compute birth eigenvalues if next iteration is 2
function z=NewNextLevel2(x)
vert=Vertices(x);
z=[5 (vert(3)+2); 6 vert(2)-3];

%Compute birth eigenvalues if next iteration is 3
function z=NewNextLevel3(x)
vert=Vertices(x);
z=[3 (vert(3)+2); 5 (vert(3)+2); 6 (vert(1)+vert(2)-3)];

%Given a list of eigenvalues, performs spectral decimation using 2 rules
function z=NextLevel2(x)
n=size(x);
z=[];
access=1;


for i=1:n(1)
    if(x(i,1)==6)
        z(access,:)=[3 x(i,2)];
        access=access+1;
    else
        z((access:access+1),:)=[R1inv(x(i,1)) [x(i,2); x(i,2)]];
        access=access+2;
    end
end

%Same as NextLevel2 except using 3 rules
function z=NextLevel3(x)
n=size(x);
z=[];
access=1;
for i=1:n(1)

    if(x(i,1)==0)
        z((access:access+1),:)=[0 x(i,2); 4 x(i,2)];
        access=access+2;
    elseif(x(i,1)==6)
        z((access:access+1),:)=[(3+sqrt(2)) x(i,2); (3-sqrt(2)) x(i,2)];
        access=access+2;
    else
        z((access:access+3),:)=[R2inv(x(i,1)) [x(i,2); x(i,2); x(i,2); x(i,2)]];
        access=access+4;
```

```
    end
end
```

2.1.12. *toBinary.*

```
%Converts a number x to a binary number with at least m digits
function z=toBinary(x,m)

    z=[];
while (x>0)
    if (mod(x,2)==1)
        x=(x-1)/2;
        z=[1,z];
    else
        x=(x/2);
        z=[0,z];
    end
end

if(nargin==2)
    n=length(z);
    for i=(n+1):m
        z=[0, z];
    end
end
```

2.1.13. *Vertices.*

```
%Given a list x of 2's and 3's
%this function returns the number of
%triangles,vertices, and cycles for the
%corresponding gasket.
function z=Vertices(x)

triangles=1;
vert=3;
cycles=0;
n=length(x);

for i=1:n
    if(x(i)==2)
        vert=(vert+triangles*3);
        cycles=cycles+triangles;
        triangles=3*triangles;
    else
        vert=vert+7*triangles;
        cycles=cycles+3*triangles;
        triangles=triangles*6;
    end
end
z=[triangles, vert,cycles];
```

2.2. **Functions for Computing Gaps.**

2.2.1. *CGL.*

```
%Given two lists of intervals, this function returns all possible ratios of
%eigenvalues coming from these intervals (that is, eigenvalues obtained by
%repeatedly applying the smallest inverse to the something in one of the
%intervals).  The list 'which' determines whether or not we apply R21inv or
```

```
%R11inv.

function b=CGL(intervals,which,cutoff)

%int1 and int2 are respectively the coverings of the "small" and "large"
%eigenvalues
int1=intervals{1};
int2=intervals{2};


%This could possibly be made into a separate function.  We apply R21inv or
%R11inv to the intervals of eigenvalues.
for i=1:length(which)
    if(which(i)==2)
        int1=R11inv(int1);
        int2=R11inv(int2);
    else
        int1=R21inv(int1);
        int2=R21inv(int2);
    end
end



%Apparently at some point I determined initializing the variable ahead of
%time would make the program run faster.
n=length(int2);
m=length(int1);
access=1;
howLong=((n^2-n)/2 + m*n);
b=zeros(howLong,2);

%For each interval in int2 we compute all ratios
for i=n:-1:1
    b(access,:)=[1,int2(i,2)/int2(i,1)];
    access=access+1;

    %For each interval we compute with every other interval of int2
    for j=i-1:-1:1
        b(access,:)=[int2(i,1)/int2(j,2),int2(i,2)/int2(j,1)];
        access=access+1;
    end

    %Now we compute with every interval of int1
    for k=m:-1:1
        b(access,:)=[int2(i,1)/int1(k,2),int2(i,2)/int1(k,1)];
        access=access+1;

        %Stop the process if our gaps are getting too big
        if (b(access-1,1))>cutoff
            b(access-1,:)=[0 0];
            access=access-1;
            break;
        end
    end
end

%Get rid of any gaps that are too big
```

```
b=b(1:(access-1),:);
b=sortrows(b);
for i=1:length(b)
    b(i,2)=min(b(i,2),cutoff);
end
```

2.2.2. *consolidateIntervals.*

```
%Given a list of closed intervals, this function consolidates overlapping
%intervals and eliminates repeats
function z=consolidateIntervals(x)
    x=sortrows(x);
    n=size(x);
    i=2;
    current=x(1,:);
    temp=[];
    while i<=n(1)
        if(x(i,1))<=current(2)
            current=[current(1),max(current(2),x(i,2))];
        else
            temp=[temp; current]; %Consider making more efficient--don't think it matters much though
            current=x(i,:);
        end
        i=i+1;
    end

    z=[temp;current];
```

2.2.3. *eigenGaps.*

```
%Given a list of 2's and 3's, to get the spectral decimation
%process to converge to an eigenvalue, at some point we must
%only apply the "smallest" inverse.  As a result, for each eigenvalue there
%is a last stage n where something other than the "smallest inverse" is
%applied.  We say such an eigenvalue comes from level n.  To compute all
%the gaps in the spectrum of eigenvalues we only compute gaps of
%eigenvalues "coming from" consectutive levels.

%This function computes a covering for all eigenvalues coming from level n,
%where 'list' specifies what happened on levels n, n-1, n-2, . . .

%The description of how this algorithm work/what it is doing is admittedly
%poor.


function interval=eigenGaps(list)


%Initially, eigenvalues are in the range of [0,5] or [0,3+sqrt(5)]
if list(1)==2
    z2=[0 5];
else
    z2=[0 3+sqrt(5)];
end
z1=[];
n=length(list);

for i=2:(n-1)
    [z1,z2]=nextLevelEigenGaps([z1;z2],list(i));
```

```
end

%We want to ignore all eigenvalues that started to converge on an earlier
%stage.  That is, we do not apply the smallest inverse to eigenvalues the
%"small" eigenvalues.
[temp2,temp3]=nextLevelEigenGaps(z1,list(n));
[z1,temp]=nextLevelEigenGaps(z2,list(n));
z2=[temp3;temp;6 6]; %add 6 because it otherwise is not included.
interval={consolidateIntervals(z1),consolidateIntervals(z2)};


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%55
%Given a list of intervals that eigenvalues can be in at some level, this
%funtion computes the intervals they can be in on the next level (specified
%by which) keeping track of "small" and "large" eigenvalues (depending on
%whether or not the "smallest" inverse was applied)
function [smallInt,largeInt]=nextLevelEigenGaps(x,which)
smallInt=[];
largeInt=[];
n=size(x);

%If which is 2 use R1inv
if (which==2)
    largeInt=[3 3]; %we get eigenvalue 3 from eigenvalue 6 on previous level.
    for i=1:n(1)
        temp1=R1inv(x(i,1));
        temp2=R1inv(x(i,2));
        smallInt=[smallInt; temp1(1) temp2(1)];
        largeInt=[largeInt; temp2(2) temp1(2)];
    end

%Otherwise use R2inv
else
    largeInt=[3-sqrt(2) 3-sqrt(2); 3+sqrt(2) 3+sqrt(2)]; %These come from 6 of previous level.
    for i=1:n(1)
        temp1=R2inv(x(i,1));
        temp2=R2inv(x(i,2));
        smallInt=[smallInt; temp1(1) temp2(1)];
        largeInt=[largeInt; temp2(2) temp1(2); temp1(3) temp2(3); temp1(4) temp2(4)];
    end
end
```

2.2.4. *findGaps.*

```
%Given a n x 2 matrix where each row corresponds to an interval of
%eigenvalues, findGaps finds if there are any gaps in the spectrum of
%values

%Assumes b is sorted.
function z=findGaps(b)
n=size(b);
current=b(1,2);
z=[];
for i=2:n(1)
    if(current<b(i,1))
        z=[z; current b(i,1)];
        current=b(i,2);
    else
```

```
        current=max(b(i,2),current);
    end
end
```

2.2.5. *intersectList.*

```
%This function inputs a sorted n x 2 matrix x and a m x 2 matrix y.  Each row of
%x and y corresponds to an open interval.  The function returns a list of all
%possible intersections of intervals from x and y.

function z= intersectList(x,y)
    n=size(x);
    m=size(y);
    z=[];
    for i=1:n(1)
        for j=1:m(1)
            z=[z;intersect(x(i,:),y(j,:))];
        end
    end

%This function returns the intersection of two intervals x and y
function z=intersect(x,y)

z=[max(x(1),y(1)) min(x(2),y(2))];

if(z(1)>=z(2))
    z=[];
end
```

2.3. **Weyl Ratio.**

2.3.1. *count.*

```
%count(x) takes a (n x 2) entry x.  Each row of x corresponds to an eigenvalue and its multiplicity.
function z=count(x,y)

%Val will be a list of eigenvalues
val=[];
%numless(i) will be the number of eigenvalues less than val(i)
numless=[];

n=size(x);
current=0;

%Compute counting function with given list of eigenvalues
for i=1:n(1)
    current=current+x(i,2);
    val(i)=x(i,1);
    numless(i)=current;
    i=i+1;
end

%disp('there3');

%Assume as is the case with all of the gasket examples studies so far that
%there are no negative eigenvalues
val=[0 val];
numless=[0 numless];
```

```
%If there is only one input argument, just return val and numless.
%This will be suitable data for graphing the counting function
if(nargin==1)
z=[val; numless];
%If a second argument is specified, return the values of the counting
%function at the values of y
else
    y=sort(y);
    current=1;
    n=length(val);
    m=length(y);

    if(y(m)>val(n))
        disp('ERRRRRRRRRRRRRRRRRRRRROOOOOOOOOOOOOOOOOOOORRRRRRRRRRRRRRR:  Data is not valid');
    end

    for i=1:m
        for j=current:n
            if(y(i)<val(j))
                y2(i)=numless(j-1);
                break;
            elseif (j==n)
                y2(i)=numless(n);
            end

        end


    end
    z=[y;y2];
end
```

2.3.2. *fractalCounting.*

```
%x----> points to compute on
%list----->approximation to compute with
function z=fractalCounting(x,list,list2)

temp=pm(list);
disRenorm=(5^temp(1))*((90/7)^temp(2));

len=length(x);

%check this
if(x(len)>6*disRenorm)
    error('Unable to compute counting function with discrete data');
end

x=x/disRenorm;

%disp('there1');
if nargin==3

    z=count(spectrum(list),ginv(x,list2));
else

    z=count(spectrum(list),x);
end
```

2.3.3. *plotWeylRatio.*

```
function z= plotWeylRatio(list,numDataPoints,needData)

n=length(list);

if(list(1)==2)
    previous=5;
    current=5;
else
    previous=90/7;
    current=90/7;
end


for i=2:n
    if(list(i)==2)
        current=5*current;
    else
        current=(90/7)*current;
    end

    incr=((current-previous)/numDataPoints);
    x((numDataPoints*(i-2)+1):(numDataPoints*(i-1)))=(previous+incr):incr:current;
    previous=current;
end


y=weyl(x,list,[2 2 2 2 2]);

if (nargin==2)
    plot(log(x),log(y),'.','MarkerSize',.1)
    title(num2str(list))
else
    z=[x;y];
end
```

2.3.4. *pm.*

```
%Returns number of 2's and 3's in a list x of 2's and 3's
function z=pm(x)
    z(2)=sum(x-2);
    z(1)=length(x)-z(2);
```

2.3.5. *weyl.*

```
%Compute Weyl ratio.  If two arguments are given, it is computed assuming
%ginv is identity.  Otherwise it is computed using ginv and list2
%
%  x---->points to compute on.
%  list-----> approximation to compute with
%  list2---->approximations after current approximation

function z=weyl(x,list,list2)

if nargin==2
    temp=fractalCounting(x,list);
else

    temp=fractalCounting(x,list,list2);
```

```
end

z=temp(2,:)./(x.^alpha(x,list));
```

2.4. **Other Code.** The code here is code I haven't looked over as carefully, but still might be useful.

2.4.1. *gplottingscript.*

```
%Script to determine which of the "g" functions in the paper is largest and smallest.
%Of the 32  fifth level approximations, for all x>.1 g(x,[3,3,3,3,2]) is the smallest
%and g(x,[2,2,2,2,3]) is the largest.

%Of the 64 sixth level approximations for all x>.4 g(x,[3,3,3,3,3,2]) is the smallest
%and g(x,[2,2,2,2,2,3]) is the largest.


disp('max')

max=0; %variable to keep track of which of the "g" functions is greatest.
%This variable is only used when a change is made

newmax=0; %locally keeps track of which of the "g" functions is greatest.

%See which g function is greatest for each .1<i<6
for i=.1:.001:6
maxval=0;

%Cycle through all 32 g functions and see which one is largest for at i
for j=0:3
which=toBinary(j,2)+2;
temp=g(i,which);

%Update if new max has been found
if (temp>maxval)
maxval=temp;
newmax=j;
end
end

%If for this value of i the max g function is different than for the previous value of i,
%display this to screen
if (newmax ~= max)
[i,newmax]
max=newmax;
end
end

%Nearly identical to maximum algorithm
disp('min')
min=0;
newmin=0;
for i=.1:.001:6
minval=1000;

for j=0:3
which=toBinary(j,2)+2;
```

```
temp=g(i,which);
if (temp<minval)
minval=temp;
newmin=j;
end
end
if (newmin ~= min)
[i,newmin]
min=newmin;
end

end
```

2.4.2. *CGL.* This appears to be the same as above **CGL**

```
%Given two lists of intervals, this function returns all possible ratios of
%eigenvalues coming from these intervals (that is, eigenvalues obtained by
%repeatedly applying the smallest inverse to the something in one of the
%intervals).  The list 'which' determines whether or not we apply R21inv or
%R11inv.

function b=CGL(intervals,which,cutoff)

%int1 and int2 are respectively the coverings of the "small" and "large"
%eigenvalues
int1=intervals{1};
int2=intervals{2};


%This could possibly be made into a separate function.  We apply R21inv or
%R11inv to the intervals of eigenvalues.
for i=1:length(which)
    if(which(i)==2)
        int1=R11inv(int1);
        int2=R11inv(int2);
    else
        int1=R21inv(int1);
        int2=R21inv(int2);
    end
end



%Apparently at some point I determined initializing the variable ahead of
%time would make the program run faster.
n=length(int2);
m=length(int1);
access=1;
howLong=((n^2-n)/2 + m*n);
b=zeros(howLong,2);

%For each interval in int2 we compute all ratios
for i=n:-1:1
    b(access,:)=[1,int2(i,2)/int2(i,1)];
    access=access+1;

    %For each interval we compute with every other interval of int2
    for j=i-1:-1:1
```

```
        b(access,:)=[int2(i,1)/int2(j,2),int2(i,2)/int2(j,1)];
        access=access+1;
    end

    %Now we compute with every interval of int1
    for k=m:-1:1
        b(access,:)=[int2(i,1)/int1(k,2),int2(i,2)/int1(k,1)];
        access=access+1;

        %Stop the process if our gaps are getting too big
        if (b(access-1,1))>cutoff
            b(access-1,:)=[0 0];
            access=access-1;
            break;
        end
    end
end

%Get rid of any gaps that are too big
b=b(1:(access-1),:);
b=sortrows(b);
for i=1:length(b)
    b(i,2)=min(b(i,2),cutoff);
end
```

2.4.3. *newintersectlist.* The older version of this function **intersectlist** was simpler, but not nearly as efficient (I believe).

```
%This function inputs an n x 2 matrix x and a m x 2 matrix y.  Each row of
%x and y corresponds to an interval.  The function returns a list of all
%possible intersections of intervals from x and y.  The function assumes
% the intervals in x and y are disjoint and sorted.

function z= newintersectList(x,y)
    n=size(x);
    m=size(y);
    m=m(1);
    z=[];
    place=1; %Keep track of which intervals in y we have looked at

    %We will figure out the contribution of each interval in x
    for i=1:n(1)
    c=1;
    place=min(place,m); %Makes sure we get everything from the last entry of y
        %We look at intervals in y until there are no more left or until
        %the interval in y is to the right of the interval in x
        while (c & (place < (m+1)))
        temp=[max(x(i,1),y(place,1)), min(x(i,2),y(place,2))];
        %Compute the intersection of the interval from x and interval from y

        %If the intersection is empty see if the interval from y is to the
        %right or to the left to the one of x.  In the first case we move
        %on to the next interval in x, in the
        %latter case we keep on checking intervals from y
        if (temp(1)>temp(2))
            if (x(i,2)<y(place,1))
                place=max(1,place-1);
                c=0;
```

```
            else
                place=place+1;
            end

   %If the intersection is non-empty, we add it to z and check the next
   %interval of y
        else
            z=[z;temp];
            place=place+1;
        end

        end
    end
```

2.4.4. *gplottingscript.* This code (or some tweaking of it was used to generate Figure 4.4).

```
clear;
x1=0:.001:2;
x2=2:.001:4;
x3=4:.001:6;
x=0:.001:6;
fig=0;
colors={'k' 'r' 'm-' 'c-' 'r-' 'g-' 'b-' 'k--'};

for i=0:31

    if (mod(i,16)==0)
        fig=fig+1;
    end



    which=toBinary(i,5)+2;
    y1=g(x1,which);
    y2=g(x2,which);
    y3=g(x3,which);
    y=g(x,which);
    %subplot(2,2,subfig+1);
    figure(1)
    plot(x1,y1,colors{fig});
    hold on

    figure(2)
    plot(x2,y2,colors{fig});
    hold on

    figure(3)
    plot(x3,y3,colors{fig});
    hold on

    figure(4)
    plot(x,y,colors{fig});
    hold on


end
```