Eigenmodes of a Minkowski island shaped drum

Fredrik Knapskog

January 2020

Abstract: Eigenmodes and eigenfrequencies of a Minkowski island shaped drum were computed by discretization of Helmholtz equation. The eigenmodes were visualized, and from the correction term in the Weyl-Berry conjecture the Hausdorff dimension was determined to 1.512

1 Introduction

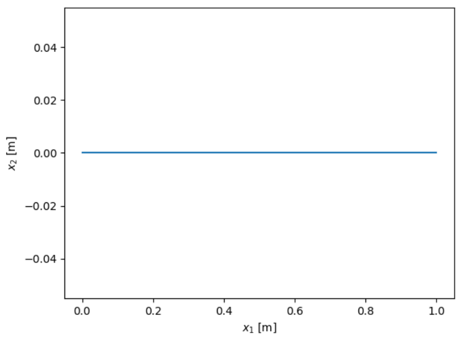
The shape of a drum cannot, by its sound, be determined completely for most drums. However some information can still be deduced. From the Weyl conjecture concerning the number of states N

$$N(\omega) = \frac{A}{4\pi}\omega^2 - \frac{L}{4\pi}\omega, \tag{1}$$

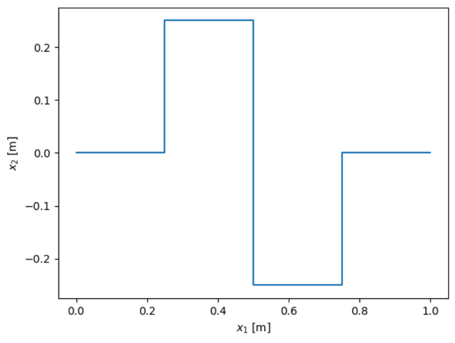
a continuous drum with a large surface, A, should have a lower tone, ω, than a drum of same shape, but smaller surface area. If two surface areas are equal, the drum with the shortest circumference, L, should have the lowest tone. How does this apply for fractal drums? In the 1990s Sapoval and his crew [1] experimented the modes of a fractal drum shaped as a Minkowski island as in figure 2. They observed modes localised to bounded regions of the drum, and were even capable of exciting them separately. For a continuous drum this would not be possible as striking any part would make the whole part vibrate. Sapoval showed corners facing inwards have large amplitude solutions for the wave motion equation. The large amplitude regions generates lots of large interfering amplitudes with dissipation resulting in damping. For the Minkowski island the narrow passages and strong damping prevent waves from travelling between two opposite piers. How does this affect the scaling of eigenmodes?

2 Theory

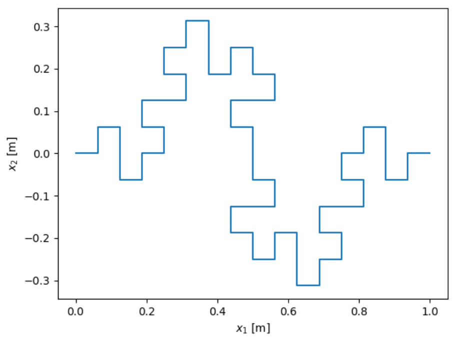
The Minkowski island is a figure composite of 4 Minkowski sausages put together. A Minkowski sausage is a recursive fractal where the next iteration is a chain of 8 links of the previous iteration. The length of the new links are a quarter of the previous chain. The new chain is then rotated 90° to the left after the first, fifth and sixth link. Then 90° to the right after the second, third and seventh link. The start sausage is a straight line, thus the first four iterations take form as in figure 1. The composite Minkowski island for the first two iterations are shown in figure 2 together with the discrete grids.



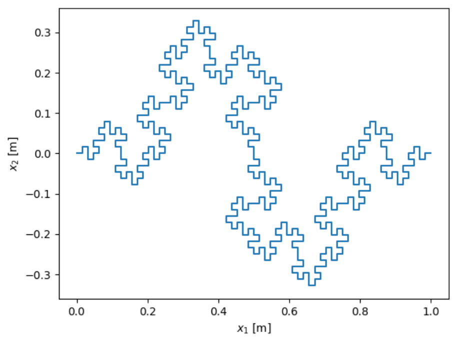
(a) Zeroth iteration Minkowski sausage.



(b) First iteration Minkowski sausage

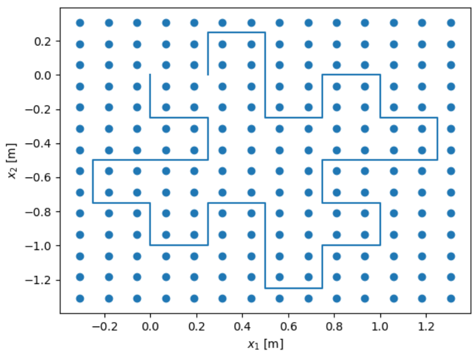


(c) Second iteration Minkowski sausage

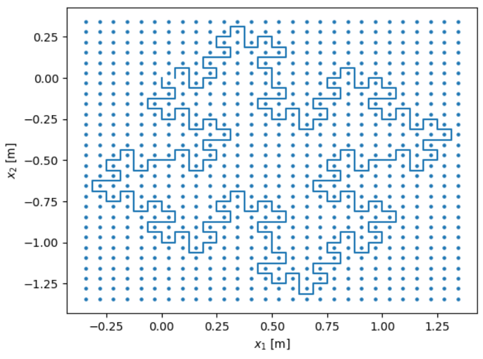


(d) Third iteration Minkowski sausage

Figure 1: The first four iterations of the Minkowski sausage. All starting at [0,0] and ending at [1,0].



(a) 1st iteration Minkowski island.



(b) 2nd iteration Minkowski island.

Figure 2: 1st and 2nd iteration Minkowski islands with grids. The resolutions are 2×2 points per square and 1×1 point per square respectively.

A deviation from equilibrium of the drum membrane, u, at position, $\vec{x}$, and time, t, is determined by the wave equation

$$\nabla^2 u(\vec{x}, t) = \frac{1}{v^2}\frac{\partial u}{\partial t}, \tag{2}$$

with the wave speed, v. Taking the Fourier transform of (2) yields the Helmholtz equation

$$-\nabla^2 U(\vec{x}, \omega) = \frac{\omega^2}{v^2} U(\vec{x}\omega), \text{ in } \Omega, \tag{3}$$

$$U(\vec{x}, \omega) = 0 \text{ on } \delta\Omega. \tag{4}$$

Ω is the domain and ω is the drum frequency. Thus the eigenmodes and eigenfrequencies of the drum are the eigenvectors and eigenvalues of the Laplacian matrix. From [1] and [2], the number of states for a fractal drum, also known as the Weyl-Berry conjecture,

$$N(\omega) = \frac{A}{4\pi}\omega^2 - C_d M \omega^d + \text{lower terms.} \tag{5}$$

C_d and M are constants, and d is the dimension of the perimeter. Equation (1) and (5) are not rigorous, but the integrated density of states will rather oscillate around the expressions from (1) and (5). The amplitude of said oscillations will asymptotically finite compared to the correction term

$$\Delta N(\omega) = \frac{A}{4\pi}\omega^2 - N(\omega), \tag{6}$$

which from (5) scales as ω^d.

3 Method

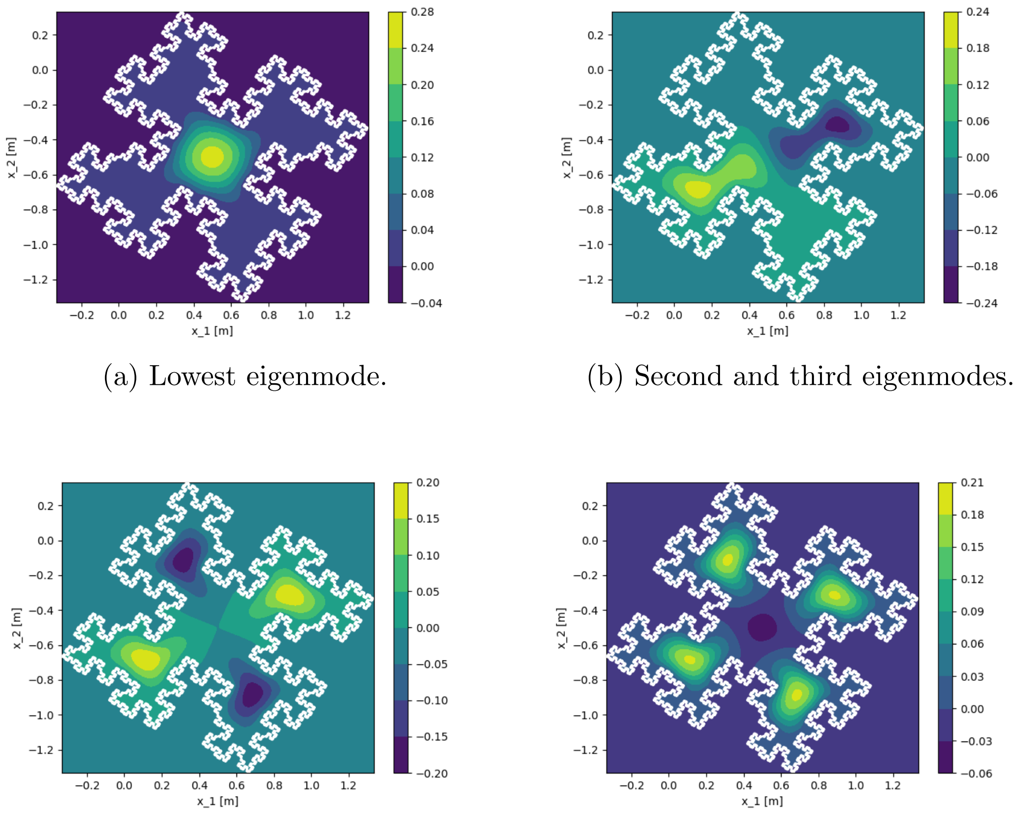
In (3), the matrix $U(\vec{x}, \omega) = U_{i,j}$, can be flattened to an array, V_j, where the elements not contained by the domain Ω are left out. The Laplacian matrix is thus a $N \times N$ matrix with N being the number of elements in V. Using a finite difference approximation of fourth order accuracy, or a so called 9 point stencil, the elements of the Laplacian are

$$h^2 \nabla^2_{i,j} = \begin{cases} -5 & \text{if } i = j \\ \frac{4}{3} & \text{if } V_j = U_{i\pm1,j} \text{ or } U_{i,j\pm1} \\ -\frac{1}{12} & \text{if } V_j = U_{i\pm2,j} \text{ or } U_{i,j\pm2} \\ 0 & \text{else.} \end{cases} \tag{7}$$

As a result, the Laplacian becomes a very large, sparse matrix. For Minkowski levels larger than 3, the matrix becomes too large to store for most computers. Instead of storing it as a matrix, a dictionary is used to store only the nonzero values. Furthermore, due to it's sparseness an iterative method is chosen for finding the eigenvalues. More precisely, the implicitly restarted Arnoldi method. For the dictionary, the python package, scipy.sparse.dok_matrix was used, and for the eigenvalues, scipy.sparse.linalg.eigs.

4 Result and discussion

For a fourth iteration Minkowski island shaped drum, with initial side length of 1 m and thus a surface area of 1 m^2 and grid distance $h = 0{,}004$ m, the contour plots in figure 3 matches Sapoval's figures [1] well.



(a) Lowest eigenmode.

(b) Second and third eigenmodes.

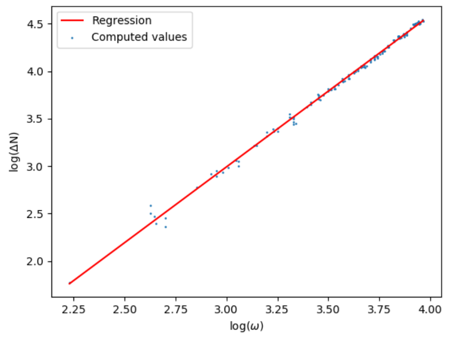
(c) Fourth eigenmode.

(d) Fifth eigenmode.

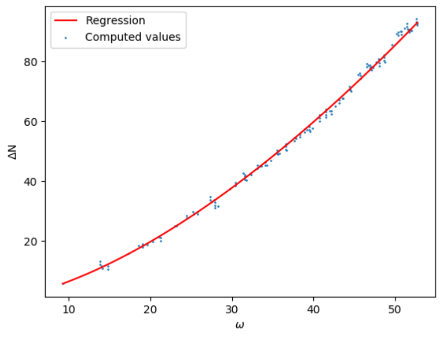
Figure 3: Contour plots of the first five eigenmodes of the third iteration Minkowski island. The white curve indicates the boundary of the drum.

The dimension, d, depends on how many eigenfrequencies are used for the regression. The regression in figure 4a and 4b, including only 130 eigenfrequencies, yields a dimension of 1,597, while in figure 4c and 4d , including 1000 eigenfrequencies, the dimension drops to 1,439. Figure 5 shows how the dimension varies for the number of eigenfrecuencies accounted for in the regression. Because of this, determining a fixed value for the dimension becomes difficult. Where should the list of eigenfrequencies be truncated? In figure 5 the theoretical Hausdorff dimension of $\log_4(8) = 1.5$ is contained within the upper and lower bound, so perhaps this method can give an interval where the Hausdorff dimension should be. The average of those bounds is 1,515 which is rather close. Another argument could be to use the lower bound as the dimension seems to converge towards it. However, more computational power is needed to test that.

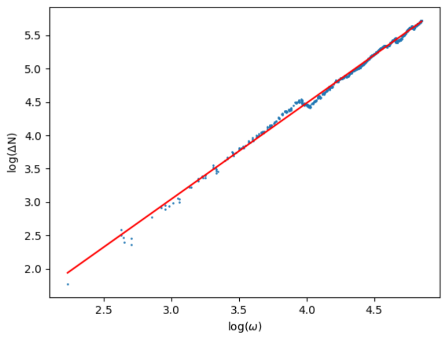
The numerical model assumes an infinitesimal thin and very elastic model. Also the boundary conditions says nothing about how the wave function approaches zero on the boundary. Thus the numerical model used may not be



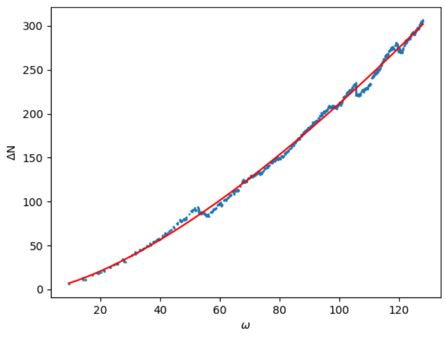
(a) $\log(\Delta N)$ for 130 ω.



(b) ΔN for 130 ω.



(c) $\log(\Delta N)$ for 1000 ω.



(d) ΔN for 1000 ω.

Figure 4: The correction term, ΔN from (6), for a fourth iteration Minkowski island shaped drump, plotted for 130 eigenfrequencies in the upper row and for 1000 in the lower row. The red lines in the left column are the linear regressions of the logarithm of the correction term, and in the right column the red lines are the eigenfrequencies to the power of the slopes found from the regressions.

completely consistent with the experiment. The grid resolution grows exponentially with the iteration level of the Minkowski island, so with the fourth order accuracy in space the finite difference error becomes quite small, but nevertheless the error still is there. What is more worth mentioning is how (1), (5) and (6) are not rigorous equations, but instead estimates where the true solution oscillates around the estimates.

The method used for determining whether a point is within the domain or not does not matter as much here as long as it is correct. The time that takes is nothing compared to finding the eigenvalues of the Laplacian matrix, so the scipy.sparse.linalg.eigs works as a bottle neck for computation time. The same goes for making the Minkowski islands. When it comes to the space for storing the Laplacian matrix, the dictionary for a 3rd iteration Laplacian takes up only 1.310.832 bytes. The regular matrix representation storing all zeros too take up 134.217.840 bytes, which is 100 times more. For the fourth iteration the dictionary takes up 20.971.632 bytes, which is 16 times more than for the third iteration. The matrix represantation here resulted in a memory error.

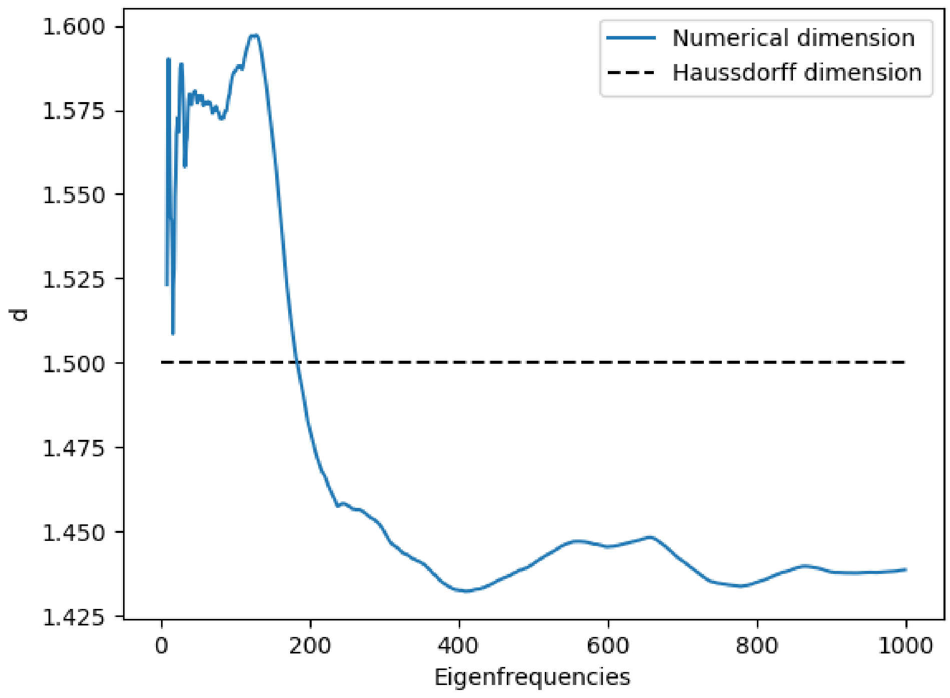


Figure 5: The dimension, d, of the fractal drum as a function of how many eigenfrequencies, ω, is accounted for in the regression. The dashed line is the Hausdorff dimension.

5 Conclusion

Even though the fractal shape of a drum cannot be completely determined by just the eigenfrequencies, a somewhat approximation can be made about the dimension of the drums perimeter. Or at least maybe the range interval where the dimension belongs can be determined.

References

[1] B. Sapoval, Th. Gobron, and A. Margolina. Vibrations of fractal drums. *Phys. Rev. Lett.*, 67:2974–2977, Nov 1991.

[2] Steven Homolya. Generalisation of the modified weyl–berry conjecture for drums with jagged boundaries. *Physics Letters A*, 318(4):380 – 387, 2003.

Appendices

```python
#!/usr/bin/env python
# coding: utf-8

# In[1]:


#Importing libraries
import numpy as np
from matplotlib import pyplot as plt
import copy
from scipy.spatial.transform import Rotation as R
from scipy.sparse.linalg import eigs
from scipy.sparse import dok_matrix
import sys
get_ipython().run_line_magic('matplotlib', 'notebook')


# In[341]:


#Function for rotating a set of points around the first point in said set
def rotation(coords, deg):
    x = len(coords[0])
    vectors = np.array([coords[0], coords[1], [0]*x]).T
    r = R.from_rotvec([0, 0, deg])
    a = r.apply(vectors)
    a = a[1:]
    a = a.T[0:-1]
    return a

#Recursive function for creating a Koch curve:
#Recursively inserts previous chain 8 times with a quarter length. Inserts one
#the recipy. The first point in each chain is cut out in order to avoid double
#Finally when the zeroth iteration is reached, just two points is returned.
def fractalGenerator(L, l):
    if l == 0:
        a = np.array([[0,L], [0,0]])
        return a

    segment = fractalGenerator(L/4, l-1)
    x = len(segment[0])-1
    corners = copy.copy(segment)
    corners = np.append(corners, [[corners[0][-1]]*x, [corners[1][-1]]*x] + rotat
    corners = np.append(corners, [[corners[0][-1]]*x, [corners[1][-1]]*x] + rotat
    corners = np.append(corners, [[corners[0][-1]]*x, [corners[1][-1]]*x] + rotat
    corners = np.append(corners, [[corners[0][-1]]*x, [corners[1][-1]]*x] + rotat
    corners = np.append(corners, [[corners[0][-1]]*x, [corners[1][-1]]*x] + rotat
    corners = np.append(corners, [[corners[0][-1]]*x, [corners[1][-1]]*x] + rotat
    corners = np.append(corners, [[corners[0][-1]]*x, [corners[1][-1]]*x] + rotat
    return corners

#Creating the Minkowski island with 4 Koch curves as sides. Each point in the r
def fractalCorners(L, l):
    segment = fractalGenerator(L, l)
    x = len(segment[0])-1
    corners = copy.copy(segment)
    corners = corners.T[1:].T
    corners = np.append(corners, [[corners[0][-1]]*x, [corners[1][-1]]*x] + rotat
    corners = np.append(corners, [[corners[0][-1]]*x, [corners[1][-1]]*x] + rotat
    corners = np.append(corners, [[corners[0][-1]]*x, [corners[1][-1]]*x] + rotat
    return np.round(corners, 12)

#Creates a grid corresponding to the Minkowski island. The commented out lines
#wanted on the boundary instead.
def makeGrid(L, l, N):
    corners = fractalCorners(L, l)
    x2, x1 = np.max(corners[0]), np.amin(corners[0])
    y2, y1 = np.max(corners[1]), np.amin(corners[1])
    Nx = N*(x2-x1)/((0.25)**l)+1
    Ny = N*(y2-y1)/((0.25)**l)+1
    dx, dy = (x2 - x1)/Nx, (y2 - y1)/Ny
    x = np.linspace(x1- 0.5*dx, x2 + 0.5*dx, int(Nx)+1)
    y = np.linspace(y1 - 0.5*dy, y2 + 0.5*dy, int(Ny)+1)
#     x = np.linspace(x1, x2, int(Nx))
#     y = np.linspace(y1, y2, int(Ny))
    X, Y = np.meshgrid(x, y)
    grid = np.array([X.flatten(),Y.flatten()])
    return grid, corners, x, y

#Sorts points after x and y coordinate, should have called testx for sortx, my
#This is done globally so it doesn't have to be done for each point going throu
def globalVariables(points):
    corners = points.T
    testx = copy.copy(corners)
    testy = copy.copy(corners.T)
    testy = np.flip(testx, axis = 1)
    testx = testx[np.lexsort(testx.T[::-1])].T
    testy = testy[np.lexsort(testy.T[::-1])].T
    return corners, testx, testy

#Function for getting the index for a point in an array with coordinates for th
def getIndex(arr, x, y):
    c1 = np.where(arr[0]==x)
    c2 = np.where(arr[1]==y)
    return np.intersect1d(c1, c2)

#This function determines whether a point is within the domain, outside the dom
#narrowed down to three cases. The points from earlier (not the grid points) re
#points fit on an equidistant grid (this is the grid I'll be talking about here
#Furthermore, these points come in order in the corner list. So first case is i
#the input point is not on a such grid line. That is, the input point does not
#either x or y axis. If that is the case, the 4 closest points from the corner
#can be used. If they clock-wise come in increasing order in the corner list, t
#by the domain, otherwise not.  To find these surrounding points, the lists sor
#neighbouring points by x-coordinates and the two by y-axis. If there is any of
#point must be outside. Second case is if the points lie on one grid line. Then
#corners on the rectangle surrounding the input point. Like the corners on a bi
#line between the center pockets. The same principles count as for the first ca
#directly after each other. Then the point is on the boundary. The last case is
#of two grid lines. If it lies on a corner point it is then on the boundary. If
#points like in a 5 point stencil. L = left, R = right, T = top, B = bottom

def determinePoint(x, y, testx, testy, corners):
    a, b = np.searchsorted(testx[0], x, side = 'left'), np.searchsorted(testx[0],
    c, d = np.searchsorted(testy[0], y, side = 'left'), np.searchsorted(testy[0],
    if a == b and c == d: #point lies off Koch grid lines
        a1 = np.searchsorted(testx[0], testx.T[a-1][0], side = 'left')
        if b >= len(testx[0]):
            return -1
        b1 = np.searchsorted(testx[0], testx.T[b][0], side = 'right')
        colL, colR = testx.T[a1:b], testx.T[b:b1]
        colR = np.flip(colR, axis = 1)
        colL = np.flip(colL, axis = 1)
        colL = colL[np.lexsort(colL.T[::-1])].T
        colR = colR[np.lexsort(colR.T[::-1])].T
        L = np.searchsorted(colL[0], y)
        R = np.searchsorted(colR[0], y)
        if L >= len(colL[0]) or R >= len(colR[0]) or L == 0 or R == 0: #is point
            return -1 #no
        LT = getIndex(corners.T, colL.T[L][1], colL.T[L][0])
        LB = getIndex(corners.T, colL.T[L-1][1], colL.T[L-1][0])
        RT = getIndex(corners.T, colR.T[R][1], colR.T[R][0])
        RB = getIndex(corners.T, colR.T[R-1][1], colR.T[R-1][0])
        if LT < RT < RB < LB or RT < RB < LB < LT or RB < LB < LT < RT or LB < LT
            return 1 #yes
        else: #has to be outside of drum
            return -1
    elif a == b: #does the point lie on a horisontal Koch points line?
        c1 = np.searchsorted(testy[0], testy.T[c][0], side = 'left')
        row = testy.T[c1:d]
        row = np.flip(row, axis = 1)
        row = row[np.lexsort(row.T[::-1])].T
        I = np.searchsorted(row[0], x)
        if I >= len(row[0]) or I == 0:
            return -1 #boundary condition
        diff = np.abs(getIndex(corners.T, row.T[I][0], row.T[I][1]) - getIndex(co
        if diff == 1 or diff == len(corners)-1:
            return 0 #point lies on the drum edge
        else:
            xR = row[0][I]
            xL = row[0][I-1]
            a1 = np.searchsorted(testx[0], xL, side = 'left')
            b1 = np.searchsorted(testx[0], xR, side = 'right')
            colL, colR = testx.T[a1:b], testx.T[b:b1]
            colR = np.flip(colR, axis = 1)
            colL = np.flip(colL, axis = 1)
            colL = colL[np.lexsort(colL.T[::-1])].T
            colR = colR[np.lexsort(colR.T[::-1])].T
            L = np.searchsorted(colL[0], y)
            R = np.searchsorted(colR[0], y)
            if L == 0 or L >= len(colL[0])-1 or R == 0 or R >= len(colR[0])-1: #
                return -1 #no
            LT = getIndex(corners.T, colL.T[L+1][1], colL.T[L+1][0])
            LB = getIndex(corners.T, colL.T[L-1][1], colL.T[L-1][0])
            RT = getIndex(corners.T, colR.T[R+1][1], colR.T[R+1][0])
            RB = getIndex(corners.T, colR.T[R-1][1], colR.T[R-1][0])
            if LT < RT < RB < LB or RT < RB < LB < LT or RB < LB < LT < RT or LB
                return 1 #point is inside drum
            else:
                return -1 #point lies outside
    elif c == d: #does the point lie on a vertical Koch points line?
        a1 = np.searchsorted(testx[0], testx.T[a][0], side = 'left')
        col = testx.T[a1:b]
        col = np.flip(col, axis = 1)
        col = col[np.lexsort(col.T[::-1])].T
        I = np.searchsorted(col[0], y)
        if I >= len(col[0]) or I == 0:
            return -1 #boundary condition
        diff = np.abs(getIndex(corners.T, col.T[I][1], col.T[I][0]) - getIndex(co
        if diff == 1 or diff == len(corners)-1:
            return 0 #point lies on the drum edge
        else:
            yT = col[0][I]
            yB = col[0][I-1]
            c1 = np.searchsorted(testy[0], yB, side = 'left')
            d1 = np.searchsorted(testy[0], yT, side = 'right')
            rowB, rowT = testy.T[c1:d], testy.T[d:d1]
            rowT = np.flip(rowT, axis = 1)
            rowB = np.flip(rowB, axis = 1)
            rowB = rowB[np.lexsort(rowB.T[::-1])].T
            rowT = rowT[np.lexsort(rowT.T[::-1])].T
            B = np.searchsorted(rowB[0], x)
            T = np.searchsorted(rowT[0], x)
            if B == 0 or B >= len(rowB[0])-1 or T == 0 or T >= len(rowT[0])-1: #
                return -1 #no
            else:
                LT = getIndex(corners.T, rowT.T[T-1][0], rowT.T[T-1][1])
                LB = getIndex(corners.T, rowB.T[B-1][0], rowB.T[B-1][1])
                RT = getIndex(corners.T, rowT.T[T+1][0], rowT.T[T+1][1])
                RB = getIndex(corners.T, rowB.T[B+1][0], rowB.T[B+1][1])
                if LT < RT < RB < LB or RT < RB < LB < LT or RB < LB < LT < RT or
                    return 1 #point is inside drum
                else:
                    return -1 #point is outside
    else: #point is on cross section between a horisontal line of Koch points a
        if len(getIndex(corners.T, x, y)):
            return 0 #point lies on top of a Koch point
        else:
            c1 = np.searchsorted(testy[0], testy.T[c][0], side = 'left')
            if d >= len(testy[0]):
                return -1 #boundary condition
            row = testy.T[c1:d]
            row = np.flip(row, axis = 1)
            row = row[np.lexsort(row.T[::-1])].T
            R = np.searchsorted(row[0], x)
            a1 = np.searchsorted(testx[0], testx.T[a][0], side = 'left')
            if b >= len(testx[0]):
                return -1 #boundary condition
            b1 = np.searchsorted(testx[0], testx.T[b][0], side = 'right')
            col = testx.T[a1:b]
            col = np.flip(col, axis = 1)
            col = col[np.lexsort(col.T[::-1])].T
            C = np.searchsorted(col[0], y)
            if C == 0 or C >= len(col[0]) or R == 0 or R >= len(row[0]):
                return -1 #boundary condition
            L = getIndex(corners.T, row.T[R-1][0], row.T[R-1][1])
            R = getIndex(corners.T, row.T[R][0], row.T[R][1])
            T = getIndex(corners.T, col.T[C][1], col.T[C][0])
            B = getIndex(corners.T, col.T[C-1][1], col.T[C-1][0])
            if T < R < B < L or R < B < L < T or B < L < T < R or L < T < R < B:
                return 1
            else:
                return -1 #point lies outside drum

#This function takes x points and y points for a grid, and the corners as input
#coordinates for the grid, and a state matrix for whether the grid points are i
def makeStates(points, xlist, ylist):
    corners, testx, testy = globalVariables(points)
    Nx, Ny = len(xlist), len(ylist)
    states = np.zeros([Nx, Ny])
    coords = np.zeros([Nx, Ny, 2])
    x0, y0 = xlist[0], ylist[0]
#     print(Nx)
    for i in range(Nx):
#         if i%10==0:
#             print(i)
        for j in range(Ny):
            states[i][j] = determinePoint(np.round(xlist[i],10), np.round(ylist[j
            coords[i][j] = [xlist[i], ylist[j]]
    return states, coords




#Flattening the grid matrix from above to an array, and excluding all points th
def Ulist(xlist, ylist, states):
    Nx, Ny = len(xlist), len(ylist)
    U = np.empty(shape=[0, 2], dtype = int)
    for i in range(Nx):
        for j in range(Ny):
            if states[i][j] > 0:
                U = np.append(U, [[int(i),int(j)]], axis = 0)
    return U

#Five point stencil Laplacian matrix
def laplace5(U, states):
    N = len(U)
    #Lap = dia_matrix((N, N), dtype=np.float64).toarray()
    Lap = dok_matrix((N, N), dtype=np.float64)
    for i in range(N):
        Lap[i,i] = -4
        if i > 0:
            Lap[i,i-1] = int(states[U[i][0]][U[i][1]-1]==1)
        if i < N-1:
            Lap[i,i+1] = int(states[U[i][0]][U[i][1]+1]==1)
        y0 = getIndex(U.T, U[i][0]-1, U[i][1])
        if len(y0):
            Lap[i,y0[0]] = int(states[U[i][0]-1][U[i][1]]==1)
        y1 = getIndex(U.T, U[i][0]+1, U[i][1])
        if len(y1):
            Lap[i,y1[0]] = int(states[U[i][0]+1][U[i][1]]==1)
    return Lap

#Nine point stencil Laplacian matrix
def laplace9(U, states):
    N = len(U)
    F = len(states)
    #Lap = np.zeros([N, N])
    #Lap = dia_matrix((N, N), dtype=np.float64).toarray()
    Lap = dok_matrix((N, N), dtype=np.float64)
    for i in range(N):
        Lap[i,i] = -5
        if i > 0:
            Lap[i,i-1] = int(states[U[i][0]][U[i][1]-1]==1)*4/3
        if i > 1:
            Lap[i,i-2] = int(states[U[i][0]][U[i][1]-2]==1)*-1/12
        if i < N-1:
            Lap[i,i+1] = int(states[U[i][0]][U[i][1]+1]==1)*4/3
        if i < N-2 and U[i][1]+2 < F:
            Lap[i,i+2] = int(states[U[i][0]][U[i][1]+2]==1)*-1/12
        y0 = getIndex(U.T, U[i][0]-1, U[i][1])
        if len(y0):
            Lap[i,y0[0]] = int(states[U[i][0]-1][U[i][1]]==1)*4/3
        y01 = getIndex(U.T, U[i][0]-2, U[i][1])
        if len(y01):
            Lap[i,y01[0]] = int(states[U[i][0]-2][U[i][1]]==1)*-1/12
        y1 = getIndex(U.T, U[i][0]+1, U[i][1])
        if len(y1):
            Lap[i,y1[0]] = int(states[U[i][0]+1][U[i][1]]==1)*4/3
        y11 = getIndex(U.T, U[i][0]+2, U[i][1])
        if len(y11):
            Lap[i,y11[0]] = int(states[U[i][0]+2][U[i][1]]==1)*-1/12
    return Lap



#This is a function for task 8, the boundary condition saying dU = 0 on d omega
# def BC(U, states, N, i):
#     if i > 0 and i < N:
#         y11 = getIndex(U.T, U[i][0]+1, U[i][1]-1)
#         y12 = getIndex(U.T, U[i][0]+1, U[i][1])
#         y13 = getIndex(U.T, U[i][0]+1, U[i][1]+1)
#         y21 = getIndex(U.T, U[i][0]-1, U[i][1]-1)
#         y22 = getIndex(U.T, U[i][0]-1, U[i][1])
#         y23 = getIndex(U.T, U[i][0]-1, U[i][1]+1)
#         if len(y11)*len(y12)*len(y13)*len(y21)*len(y22)*len(y23):
#             Neumann = int(states[U[i][0]][U[i][1]]==1)
#             Neumann *= int(states[U[i][0]][U[i][1]-1]==1)
#             Neumann *= int(states[U[i][0]][U[i][1]+1]==1)
#             Neumann *= int(states[U[i][0]-1][U[i][1]]==1)
#             Neumann *= int(states[U[i][0]-1][U[i][1]-1]==1)
#             Neumann *= int(states[U[i][0]-1][U[i][1]+1]==1)
#             Neumann *= int(states[U[i][0]+1][U[i][1]]==1)
#             Neumann *= int(states[U[i][0]+1][U[i][1]-1]==1)
#             Neumann *= int(states[U[i][0]+1][U[i][1]+1]==1)
#             return Neumann
#         else:
#             return 0
#     else:
#         return 0


#Here is the Biharmonic operator with BC
# def ClampedLaplace9(U, states):
#     N = len(U)
#     F = len(states)
#     #Lap = np.zeros([N,N])
#     Lap = dok_matrix((N, N), dtype=np.float64)
#     for i in range(N):
#         Lap[i,i] = -5*BC(U, states, N, i)
#         if i > 0:
#             Lap[i,i-1] = int(states[U[i][0]][U[i][1]-1]==1)*BC(U, states, N,
#         if i > 1:
#             Lap[i,i-2] = int(states[U[i][0]][U[i][1]-2]==1)*BC(U, states, N,
#         if i < N-1:
#             Lap[i,i+1] = int(states[U[i][0]][U[i][1]+1]==1)*BC(U, states, N,
#         if i < N-2 and U[i][1]+2 < F:
#             Lap[i,i+2] = int(states[U[i][0]][U[i][1]+2]==1)*BC(U, states, N,
#         y0 = getIndex(U.T, U[i][0]-1, U[i][1])
#         if len(y0):
#             Lap[i,y0[0]] = int(states[U[i][0]-1][U[i][1]]==1)*BC(U, states, N
#         y01 = getIndex(U.T, U[i][0]-2, U[i][1])
#         if len(y01):
#             Lap[i,y01[0]] = int(states[U[i][0]-2][U[i][1]]==1)*BC(U, states,
#         y1 = getIndex(U.T, U[i][0]+1, U[i][1])
#         if len(y1):
#             Lap[i,y1[0]] = int(states[U[i][0]+1][U[i][1]]==1)*BC(U, states, N
#         y11 = getIndex(U.T, U[i][0]+2, U[i][1])
#         if len(y11):
#             Lap[i,y11[0]] = int(states[U[i][0]+2][U[i][1]]==1)*BC(U, states,
#     return Lap.dot(Lap)

#This function deals with degeneracy. Returns eigenvalues without degenerates,
def aN(lambdas):
    omegas, N = np.array([]), np.array([])
    for i in range(len(lambdas)-1):
        ohm = np.round(np.real(lambdas[i]), 9)
        if ohm !=  np.round(np.real(lambdas[i+1]), 9):
            omegas = np.append(omegas, ohm)
            N = np.append(N, i+1)
    return omegas, N
```

```
#This function returns Delta N based on eigenfunctions, number of states and th
def deltaN(omegas, Nr,  A):
    X = len(omegas)
    deltaNs = np.zeros(X)
    for i in range(X):
        deltaNs[i] = A*omegas[i]**2/(4*np.pi) - Nr[i]
    return deltaNs

#Determines dimension with linear regression.
def regdim(kopi, A, delta):
    lambdas = copy.copy(np.abs(kopi))/delta**2
    omegas, Nas = aN(lambdas**0.5)
    #plt.plot(omegas, Nas)
    deltaNs = deltaN(omegas, Nas, A)
    z = np.log(deltaNs)
    rotohm = omegas
    logohm = np.log(omegas)
    dim = np.vstack([logohm, np.ones(len(logohm))]).T
    alpha, beta = np.linalg.lstsq(dim, z, rcond=None)[0] #Least squares linear r
    regx = np.log(np.linspace(omegas[0], omegas[-1], 1000))
    regy = alpha*regx + beta
    #print("d = ", alpha)
    dregx = np.linspace(omegas[0], omegas[-1], len(omegas))
    dregy = np.exp(beta)*dregx**(alpha)
    return regx, regy, rotohm, deltaNs, dregx, dregy, alpha



# In[171]:



def main(Koch, Res, S, n9, Nc):
    grid, P, xlist, ylist = makeGrid(S, Koch, Res)
    delta = xlist[1]-xlist[0]
    states, coords = makeStates(P, xlist, ylist)
    Nx = len(states)
    U = Ulist(xlist, ylist, states)
    N = len(U)
    A = S**2
    Lap9 = laplace9(U, states)
    values9, modes9 = eigs(Lap9, k = n9, which = "SM", v0 = [0.1]*N) #Finds n9 l
    order = np.argsort(-values9) #sorts them just in case
    lambdas9 = -values9[order]
    #lambdas9 = lambdas9[::-1]
    modes9 = modes9.T[order].T
    Nc = min(Nc, len(values9))
    cont = np.zeros([Nc, Nx, Nx])
    for i in range(Nc): #creates contour plots of eigenfunctions
```

```
        contour = np.zeros([Nx, Nx])
        for j in range(len(modes9)):
            contour[U[j][1]][U[j][0]] = modes9[j][i]
        cont[i] = contour*1/delta**0.5
    X, Y = np.meshgrid(xlist, ylist)
    return lambdas9, modes9, X, Y, cont, P


# In[252]:


valss9, mods9, X, Y, cont, P = main(4, 1, 1, 1000, 30)
regx, regy, rotohm, deltaNs, dregx, dregy, d = regdim(valss9, 1, X[0][0]-X[0][1])
```