Norwegian University of
Science and Technology

Eigenmodes of a Fractal Drum

TFY4235 - Computational Physics

Author:
Henrik Friis

March, 2022

Abstract

In this report, a Minkowski fractal has been generated recursively and the eigenvalues and eigenmodes have been calculated. In order to calculate the modes, the fractal was discretized into many points, which were classified as inside or outside the interior of the fractal using ray tracing. A 5-point and 9-point stencil, approximating the laplacian in the Helmholtz equation, was represented by a matrix and the eigenvalues and modes were found by SuperLU decomposition from ARPACK in Scipy's `eigsh()`. The fractal dimension of the higher level fractals (4 and upwards) was found to be around 1.5.

1 Introduction

This report tries to explain the code written to compute the eigenvalues and eigenmodes of a fractal boundary, using the Helmholtz equation,

$$\begin{aligned} -\nabla^2 U(\mathbf{x},\omega) &= \frac{\omega^2}{v^2} U(\mathbf{x},\omega) \qquad &&\text{in } \Omega \\ U(\mathbf{x},\omega) &= 0 &&\text{on } \partial\Omega \end{aligned} \tag{1}$$

The code is written in Python (3.9.7), relying heavily on the numerical library NumPy [1] and just-in-time compilation using Numba [3].

2 Generation of a fractal boundary

To generate the boundary, I made two different recursive functions. The first function stores the desired number of points between each end of a line segment, hereafter called intermediate points. For the solution of the eigensystem, it is useful to have some intermediate points to capture details. The second function makes the boundary as if there were intermediate points on the line segments, but do not store them. This is more time and memory efficient, both for the generation of a fractal and especially for classification, since the number of vertices in the polygon is reduced, however both versions yield the same result.

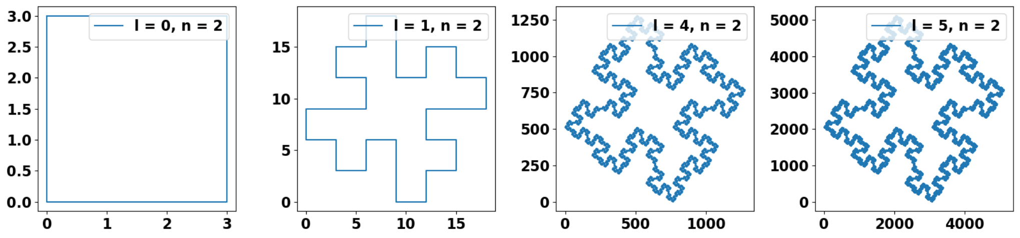

Figure 1: Some examples of fractal boundaries. l is the generation and n is the number of intermediate points.

3 Classification

Classifying whether a point is inside a closed curve is a common problem. The discrete version is often called a point-in-polygon problem. As the project progressed, classification of points as inside the curve and on the boundary or outside turned out to be the most time-consuming part of the code. Due to the runtime of methods, I have therefore tested 4 different types. All methods are thoroughly documented in "is_inside.py" and tests are provided in "test_is_inside.py". The methods that are not used in the implementation will be covered briefly. Since there is no practical difference between a point being on the boundary or outside, they will hereafter be called outside. There is. however, support for classifying points as on the boundary in the code, but it is not used.

3.1 Matplotlib

This method stores the boundary points as a path-object, using Matplotlib[2]. Points can then be passed into a path.contains_points() method and it will be classified as either inside or outside. This method is unstable for boundary points, but passing in the optional parameter radius, a small float, it will correctly classify the boundary. While yielding stable and correct results, the method is slow and thus discarded.

3.2 Shapely

This method is, at least on the surface, very similar to using a path-object from Matplotlib, but it is much slower.

3.3 Cauchy's Theorem

This method takes a totally different approach and uses Cauchy's theorem,

$$\oint_C f(z)dz = 0. \tag{2}$$

If the integral evaluates to 0, it means that the points is outside the curve and if it equals $2\pi i$, it is inside. It can be show that

$$\oint_C f(z)dz = \sum_{i=0}^{n-2}(\log(\frac{V_{i+1}-P}{V_i-P})) + \log(\frac{V_0-P}{V_{n-1}-P}), \tag{3}$$

which is implemented in code using the cmath module. While elegant and stable for boundary points, this method is unfortunately also very slow.

3.4 Ray tracing

This is a common method for solving point-in-polygon problems. The method works by extending a straight line from the point to infinity and count how many times the line crosses the boundary. If it is odd, it is inside and if it is even, it is outside. For an actual implementation, one must take into account edge cases, which are described in the documentation of the code. The operation is $\mathcal{O}(N)$ per classification. Considering that N is the number of nodes in the polygon and that the number of points in the grid can be huge, it is still very time consuming.

Apart from using just-in-time compilation and parallelization with numba, I have also added symmetry operations. The boundary has a four-fold rotational symmetry, that is taken advantage of. In short, only the first quadrant of the grid is classified, then it is rotated using numpy and put together to form the complete grid. Some points in the corners of the grids are also always outside and are excluded from the beginning. This reduces the runtime by a factor of 4, which helps considerably.

To further improve the method, one could reduce the length of the boundary by a factor 4 before applying symmetry operations. This would help considerably, since it runs in $\mathcal{O}(N)$. One must, however, first find a good way to split the boundary and this has not been implemented. Still, ray tracing is by far the fastest method and it is stable for boundary points.

3.5 Comparing the methods

The results from the speed comparison between the different methods are shown in figure 2. The first point would correspond to generation 0, the second to generation 1, etc. Some of the methods have fewer points than the others, due to very large runtime. It is easily seen that among the "other methods", matplotlib outperforms the rest by a good margin, however, it would probably take hours to classify a generation 5 fractal. Among the ray casting implementations, the parallel version utilizing symmetry outperforms the rest.

Figure 3 shows an example of the unstable matplotlib implementation, without passing in the radius keyword, while the rest shows successful classifications using the ray casting method.

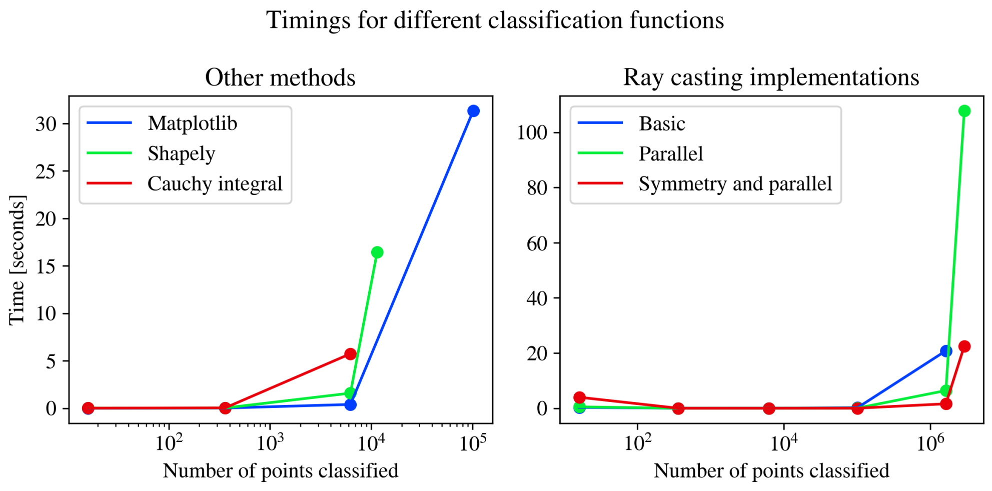

Figure 2: Some timings for the classification. Each point corresponds to a generation. Of the methods compared, ray tracing is the fastest.

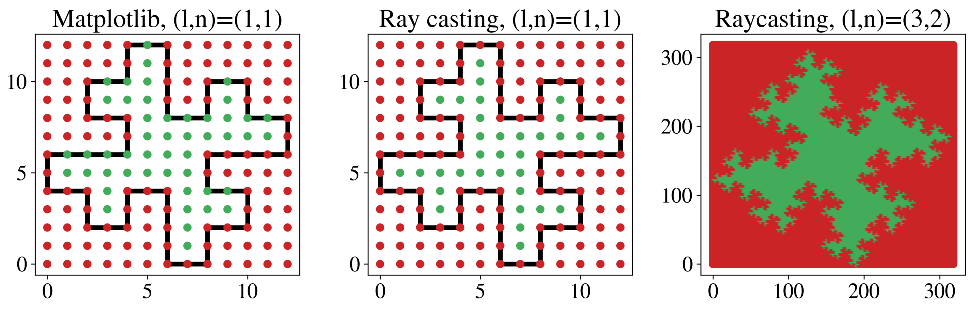

Figure 3: Classification of points using different methods. l is the generation and n is the number of points on the segments. When using Matplotlib, points can be classified as inside when sitting on the boundary.

4 A solution to the Helmholz equation

There are two main parts to solving the Helmholtz equation. Firstly, the laplacian needs to be approximated by a finite difference method. For the initial solution, a 5-point central difference approximation is used. It is then represented by a matrix. I have chosen to implement the matrix using sparse matrices from Scipy [5]. These only store non-zero data, and since these matrices consist mostly of 0's, it will dramatically reduce the memory consumption. The next part is to solve the eigenvalue problem, which is done using Scipy's `eigsh()` function.

4.1 Setting up the matrix

There are, to my knowledge, two main ways of constructing the matrix, either top-down or bottom-up. It turns out that bottom-up is the fastest and most correct way, even though both methods essentially gives the same result.

4.1.1 Top-down

Starting by top-down, it is constructed by initializing diagonals for a 5-point stencil with the length M^2, with M being the side length in the grid. Then one loops through every single point and, depending on whether the point is inside or not, makes changes to the matrix. If it is inside, it is left as is, and if it is outside, the elements of the row is set to 0, expect the diagonal element, which is unchanged to avoid an undetermined system. One is thus left with an $M^2 \times M^2$ matrix, where most of the elements are 0. I find this method intuitive, but it is a weakness that outside points are included in the matrix, even though they will have an insignificant impact on the end result. The code for this method is in the old_code folder, in the file solve_old.py.

4.1.2 Bottom-up

For this method the classified points are stored in a boolean matrix, where the point (0,0) in the grid corresponds to the (0,0) index of the matrix. If it is inside, it is represented by 1 and vice versa. In the function get_laplacian(), the indexes of all non-zero elements are stored in an array. Then the 5-points stencil is applied to each inside point and the corresponding values are added to the row. In this case the size of the matrix will be of the order $\frac{M^2}{3} \times \frac{M^2}{3}$, which will later speed up the solution to the eigensystem. The construction of this matrix is not trivial, since the distance between $x_{i,j}$ and $x_{i,j\pm1}$ is not constant, as is the case in the top-down approach. My solution to this problem is by list slicing and counting the number of times 1 (an inside point) occurs in the slices. This works fine, even though I suspect that there exist faster methods. The time used to make the matrix is of the same order of magnitude as the classification, so a faster method would not be of immediate help anyways.

4.1.3 Visualization

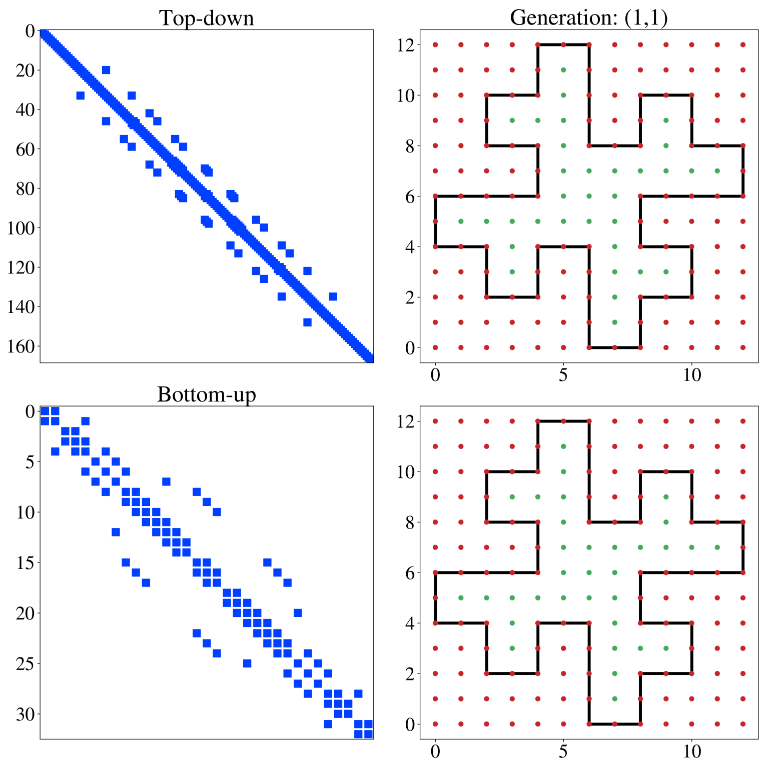

Figure 4: An attempt to visualize the construction of the matrix. Each point corresponds to a row in the top-down matrix, while only the green points corresponds to rows in the bottom up matrix.

Figure 4 shows a visualization of the different methods to construct a matrix representing the laplacian. Each point corresponds to a row in the top-down matrix, while only the green points corresponds to rows in the bottom up matrix. The grid maps to the matrix from top to bottom and left to right. One thus starts from the upper row in the grid and read left to right and downwards, like a book. Both matrices are symmetric, however the distance between $x_{i,j}$ and $x_{i,j\pm1}$ is constant for top-down, while it is not for bottom-up.

4.2 Solving the system

The code for solving the eigensystem is also found in the `solve.py` file, using the function `solve`. This function calls the Scipy-function `eigsh()`, which calculates the eigenvalues and eigenvectors of a hermitian matrix, using ARPACK. The Shift-Invert method is used to significantly speed up the calculation of the smallest eigenvalues. A table comparing the calculated eigenvalues of a generation 4 fractal with 3 intermediate points with values from literature is shown in table 1. ω_0' is calculated in accordance with the formula for ω_0 in [4], $\omega_0 = \sqrt{2}\pi c/L$, where L is the initial square side length and c is the wave velocity. L as used in my simulation can be calculated as $(n+1)\cdot 4^l$, where n is the number of intermediate points and l is the generation. As is seen, my results do not equal the results of Sapoval et. al., but they are of the same order of magnitude and follows the same trend.

Table 1: Computed and tabulated eigenvalues for a generation 4 fractal with 3 intermediate points.

$\mathbf{\Omega_i}$	**Computed**	**Sapoval et. al.**
Ω_0	$2.116\ \omega_0'$	$2.100\ \omega_0$
Ω_1	$3.169\ \omega_0'$	$3.132\ \omega_0$
Ω_2	$3.169\ \omega_0'$	$3.132\ \omega_0$
Ω_3	$3.223\ \omega_0'$	$3.191\ \omega_0$
Ω_4	$3.248\ \omega_0'$	$3.219\ \omega_0$
Ω_5	$3.384\ \omega_0'$	
Ω_6	$3.384\ \omega_0'$	
Ω_7	$3.961\ \omega_0'$	
Ω_8	$4.241\ \omega_0'$	
Ω_9	$4.363\ \omega_0'$	

The first 10 eigenmodes of the generation 4 fractal is plotted in figure 5. As can be seen in the figure, but also in table 1, there are degeneracies. Both b)/c) and f)/g) are degenerate. Both pairs are degenerate by a rotation of $\pi/2$, as is also stated in [4]. The degenerate pair b)/c) differ slightly in shape from the first plot of figure 5 in [4], but the figure could easily be reconstructed as a superposition of the states b)/c) and would also be a valid solution. One can also observe modes that are localized to the fractal corners. This would not be observed in normal drums, since the entire membrane would vibrate if it was struck. This is due to the strong damping in the narrow parts of the fractal, which in practice makes some parts of the drum inaccessible to the modes.

With my implementation, the highest level that I can practically calculate is a level 5 fractal. Using a sparse matrix to represent the laplacian, the memory needed to represent a level 5 fractal with 2 intermediate points is approximately 74 MB. If the matrix was dense, the necessary space needed to store it in memory would be around 600 TB. It is thus the code efficiency that stops my code from simulating a level 6 fractal, since I have 8 GB RAM on my computer. However, even when using sparse matrices, a normal personal computer (infact most computers at all) will not be able to store the laplacians for level 8 or 10 fractals in memory.

While the laplacian might be small enough to store at level 6, it must still be solvable for the method one calls. When I use Scipy's `eigsh()` it solves the system using SuperLU decomposition with ARPACK, and this implementation quickly uses more than 8 GB of memory at detailed level 5 fractals. In order to deal with this, one could perhaps decrease the resolution in areas of less physical importance, thus constructing a smaller laplacian which would reduce memory problems and time consumption.

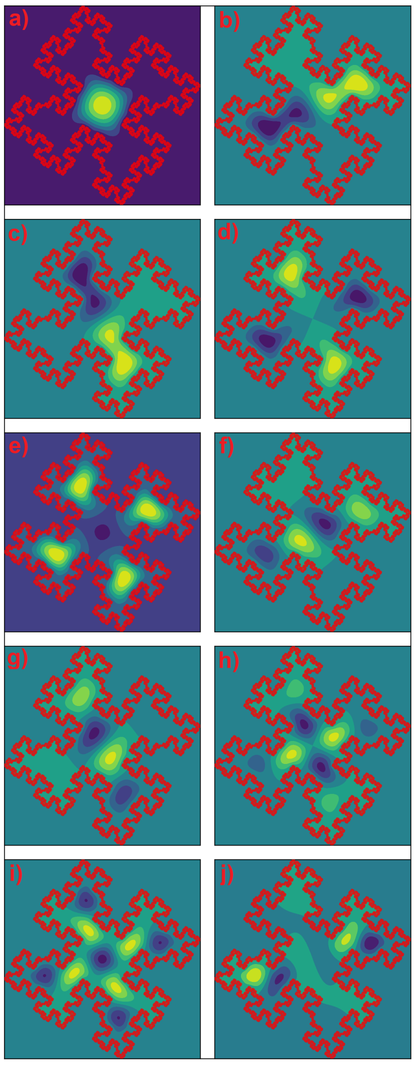

Figure 5: The contour plots for the first 10 eigenmodes of a generation 4 fractal with 3 intermediate points. Yellow has the highest value, while blue has the lowest.

4.3 Higher order approximation

The laplacian for the fractal can also be constructed using a higher order approximation, for instance a 9-point stencil. This 4-order finite difference method is also implemented in the file `solve.py` as the function `get_laplacian_order_4()` and is only a continuation of the previously described bottom-up method. There is one major difference in the implementation and that is checking whether an index is valid, which leads to some extra lines of code. This check was not necessary in the 4-point stencil-case, since all neighbouring points to points on the inside of the fractal were guaranteed to be in the matrix, while the second to closest neighbours are not. This can be seen in figure 3.

A comparison between the computed eigenvalues for a level 4 fractal with 3 intermediate points using a 2-order 4-order finite difference approximation is shown in table 2. There are only small differences between the values,

Table 2: Shows a comparison between the computed eigenvalues for a level 4 fractal with 3 intermediate points, using a 2- and 4-order finite difference approximation.

$\boldsymbol{\Omega_i}$	**2-order**	**4-order**
Ω_0	2.116 ω_0'	2.117 ω_0'
Ω_1	3.169 ω_0'	3.172 ω_0'
Ω_2	3.169 ω_0'	3.172 ω_0'
Ω_3	3.223 ω_0'	3.232 ω_0'
Ω_4	3.248 ω_0'	3.250 ω_0'
Ω_5	3.384 ω_0'	3.386 ω_0'
Ω_6	3.384 ω_0'	3.386 ω_0'
Ω_7	3.961 ω_0'	3.963 ω_0'
Ω_8	4.241 ω_0'	4.243 ω_0'
Ω_9	4.363 ω_0'	4.366 ω_0'

but the eigenvalues computed using a 4-order approximation is consistently a little higher than those using a 2-order approximation. The eigenmodes computed using the 4-order approximation are also very similar to the ones previously computed, as shown in figure 6. The most striking difference is that figure 6 is inverted compared to figure 5. This is probably not due to using a 4-order finite difference approximation, but more likely an artefact of the `eigsh()` function. However, b) and c) appear more detailed in figure 6.

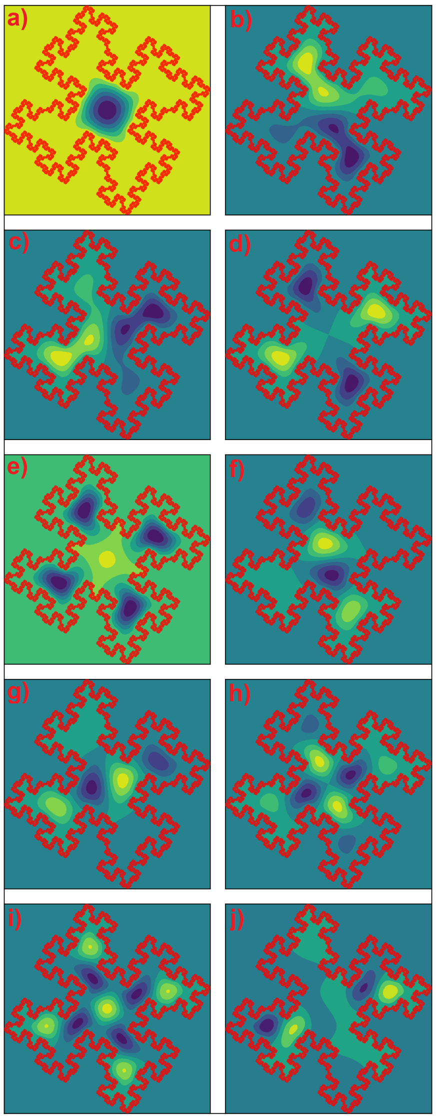

Figure 6: The contour plots for the first 10 eigenmodes of a generation 4 fractal with 3 intermediate points using a 4-order finite difference approximation. Yellow has the highest value, while blue has the lowest.

5 Scaling of $\Delta N(\omega)$

The code used to investigate the scaling of $\Delta N(\omega)$ and find an estimate for d is found in the file `scale.py`. In order to find an estimate for d, the eigenvalues are calculated using `main()` in the `solve.py` file, as earlier described. The eigenvalues are then loaded into the scaling file. The density of states below a certain eigenvalue is simply the number of eigenvalues smaller than the eigenvalue itself. The area was found from the initial sidelengths, utilizing that every time a transformation takes places, the same amount of area is added as is removed from the fractal. Then I took the logarithm of the data before fitting it to a linear function, since ω^d will be a linear function in a log-log plot.

Table 3: Shows the estimated dimension, d, of the fractal. l is the level and n are the number of intermediate points.

(l, n)	**d**	**Eigenvalues**
(2,1)	1.26	100
(2,15)	1.43	90
(3,4)	1.77	120
(3,15)	1.76	100
(4,5)	1.49	30
(5,1)	1.54	10

Table 3 shows that the fractal dimension is around 1.5, which is also stated in [4], but it is most prominent for high l and is probably improved by large n, as indicated by the precision of the (4,5) configuration. Another possible explanation might be that the increased number of eigenvalues used in the calculation adds to the precision. Due to memory issues, I was not able to calculate more eigenvalues for the (5,1) configuration. Also, it is important to keep in mind that low level fractals are not real fractals (nor are high level, but they are a much better approximation), so they might also not have a fractal dimension of 1.5, but probably rather something close to the topological dimension of 1. In figure 7a, the eigenvalues of a level 4 fractal is plotted against the correction term. The linear function that is fitted to the logarithmic data is $f(x) = 1.49 \cdot x + 9.54$. In figure 7b, one can see what can happen if one tries to use many eigenvalues to estimate $\Delta N(\omega)$. For the larger eigenvalues, $\Delta N(\omega)$ starts decaying and eventually becomes negative, making them useless. In order to estimate d, one should only use eigenvalues in the region where $\Delta N(\omega)$ is growing. This problem is most prominent for low levels, as one can calculate many more eigenvalues for a level 3 fractal before the same effect appears. For larger fractals, the time requirement would stop a normal computer from efficiently calculating enough eigenvalues to experience the problem.

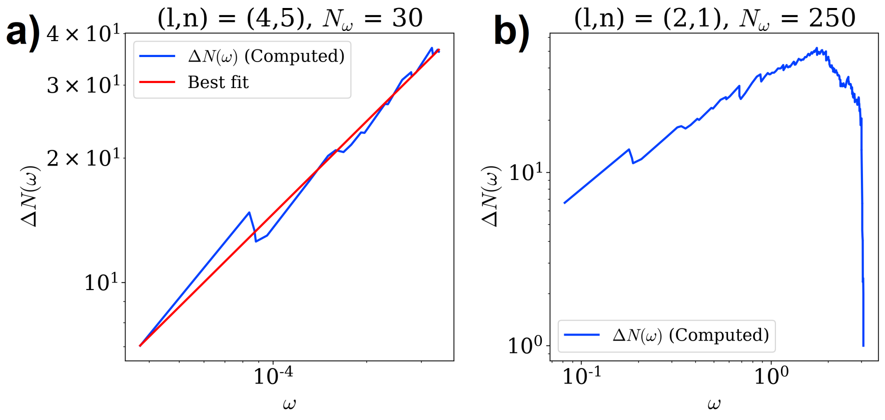

Figure 7: a) shows the eigenvalues of a level 4 fractal plotted against the correction term. The linear function that is fitted to the logarithmic data is $f(x) = 1.49 \cdot x + 9.54$. b) shows how the correction term, $\Delta N(\omega)$, oscillates and starts decaying for larger eigenvalues.

6 Conclusion

The eigenmodes of a fractal drum has been succesfully calculated and shown to be consistent with the literature. A main challenge in getting reasults from high level fractals is the rapidly increasing memory consumption when approximating and solving the eigensystem.

Bibliography

[1] Charles R. Harris et al. ‘Array programming with NumPy’. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[2] J. D. Hunter. ‘Matplotlib: A 2D graphics environment’. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[3] Siu Kwan Lam, Antoine Pitrou and Stanley Seibert. ‘Numba: A llvm-based python jit compiler’. In: (2015), pp. 1–6.

[4] B. Sapoval, Th. Gobron and A. Margolina. ‘Vibrations of fractal drums’. In: *Physical Review Letters* 67.21 (1991), pp. 2974–2977. DOI: 10.1103/physrevlett.67.2974.

[5] Pauli Virtanen et al. ‘SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python’. In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.