TFY4235 - Fractal drums

Karl Kristian Ladegård Lockert[a]

[a] *Institutt for fysikk, Norges Teknisk-Naturvitenskapelige Universitet, N-7491 Trondheim, Norway.*

1. Introduction

The challenge we are facing is that of solving the second order differential equation

$$-\boldsymbol{\nabla}^2 U(\mathbf{x},\omega) = \left(\frac{\omega}{v}\right)^2 U(\mathbf{x},\omega). \tag{1}$$

This is a model of a drum strapped to a perimeter. The perimeter we are challenged to deal with is the Koch-fractal with $\pi/2$ rotational symmetry. The strategy is to implement a framework for setting up, classifying, and solving an eigenproblem with the best resolution computationally possible. The implementation will be written in Python (3.7), heavily reliant on a diverse range of numerical libraries that are both easy to use and (for the most part) implemented in either C/C++ or Fortran. In this way, almost all time-consuming code will be "optimized", for example by using Intel's Math Kernel Library. The implementation will at the beginning be somewhat derailed from the description in the assignment, with consequences (that I have dealt with) for the remainder of the tasks.

2. Fractal boundary

The implementation of a fractal, in general, is perhaps most intuitively realized by a recursive algorithm. This algorithm shoul take the desired recursion depth as input, and call on its self with the level below. Only at recursion level $l = 0$ should the main pattern be "drawn". My implementation of the fractal is somewhat different from the described method in the assignment, in the sense that i choose how many points I want to have between each corner of the fractal, before making the fractal at all! Thus, with consequences for the remainder of the tasks, the dimensions of a fractal specified with an initial configuration (l, p) = (recursion depth, number of points between each corner) is determined by the *integer* positions (x, y) of each boundary point. To be able to compare how the parameters affect the resulting eigenfrequencies, the grid is normalized to have sidelengths 1, implicitly defining the stepsize $h = \frac{1}{N}$ where N is the largest value of the positions on the boundary. In Figure 1, a few examples of the fractal is shown. With these definitions of the fractal, the mathematical expression for the maximum indices turns out to be

$$M(l,p) = \frac{p+1}{3}(5 \cdot 4^l - 2), \tag{2}$$

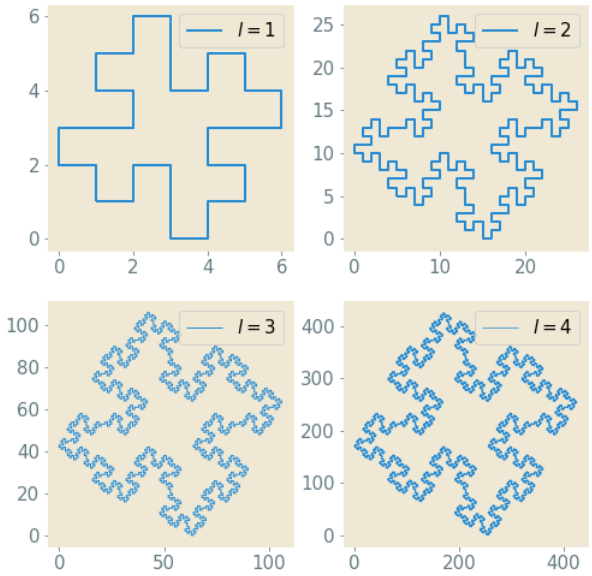


Figure 1: Fractal boundaries for $l = 1, \ldots, 4$ with $p = 0$. The x- and y-axis is just the index at each point. As we can see, a level four fractal requires more indices than a level three etc.

where again l is the recursion depth and p is the number of points between each corner. The mathematical expression is not important in detail, but it is more the fact that M is exponential in l and linear in p that is worth noticing as this have major consequences for the computation of the eigenvalues of the drum which we will discuss shortly. Evidently, the eigenvalues of an $N \times N$-matrix must be solved, where $N = M^2$. Considering that matrix operations are typically $\mathcal{O}(N^\omega)$ with $\omega \in [2, 3]$, the complexity will more or less scale in the range $\sim M^4 - M^6$, which by considering Equation (2) is highly affected by increasing the recursion level. This can, however, be reduced by utilizing the sparcity of the matrices in question.

The boundary is stored in a dictionary with its integer coordinates (indices) as the key, and a dummy boolean "True" as the value. The reason for using a dictionary, is that a lookup in a dictionary is $\mathcal{O}(1)$, whilst a search in a list is worst case $\mathcal{O}(n)$.

3. Classification of points

As we want to solve a differential equation inside the fractal, with boundary conditions $U = 0$ on and outside the boundary ($\partial\Omega$), we need to be able to classify if a given point is inside or outside the boundary. One of the advantages of classifying, apart from the necessity, is that we can remove a large number of points in the considereation, such that we can increase the resolution without running short on memory. The problem now is that we have a fractal, as in Figure 1, and need to find all points inside. For humans, classifying if a point is inside or outside a bounded region is incredibly simple, but for a computer a more challenging task.

I here propose two methods which are self-implemented, and two methods that are not.

3.1. Tracing

The first method is a method which takes as input any point in the ranges $(0, 0), (M, M)$[1] and looks for the shortest direction to an edge. There are many details to take into considerations while implementing this method, and a naive approach of flipping a bool at each crossing of a boundary point – in which the idea is based upon – is insufficient for a correct classsification. Unfortunately, I simply could not get this method to work, and after two entire days of debugging, a simpler solutions had to be utilized.

3.2. Floodfill

The second most "obvious" method is implementing a method that fills and encapsulated area recursively, i.e. such as the bucket tool in *MS Paint*. This method, called "floodfill", is also implemented, and is quite simple. An unsolved challenge with this method is that it quite quickly reaches a safety stop at maximum recursion depth. So this method has also been implemented unsuccsessfully.

3.3. Shapely

The third method uses the package "Shapely" [1] to construct a Polygon-object of the boundary points, and Point-objects from all possible points in the $M \times M$-lattice. This approach is quite intuitive, but is slow.

3.4. Matplotlib

The final method, and the method i have used thus far, is quite similar to the previous one. It uses Matplotlib[2] to create a *path*-object similar to the Polygon of Shapely. Matplotlib is written in C++, and this method is much quicker.

Method	# points	time [s]
Tracing	181476	15.5
Floodfill	181476	-
Shapely	2000	21.5
Matplotlib	181476	18.0

Table 1: Timings of the four different methods for classification of points.

3.5. Comparison

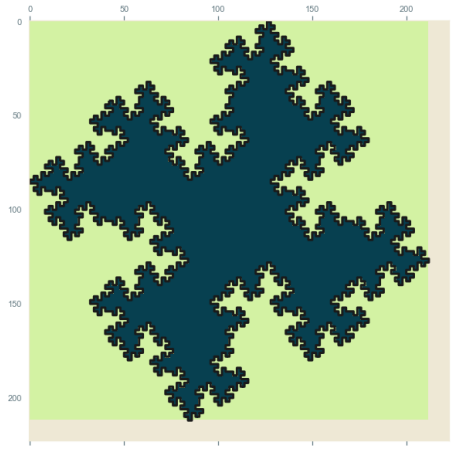
As a comparison of the timings, all four methods were tested on the configuration $(l, p) = (4, 0)$ and can be seen in Table 1. We see that our self implemented method is just as quick (even faster) than the impementation in Matplotlib. However, an unsatisfactory implementation forces the usage of the (slightly) slower method. To classify a grid with configuration (5,1) takes over 4.5 hours. Shapely is not used, since it only managed about 1% of the total points in about the same time as Matplotlib. As previously mentioned, the "floodfill"-method reaches maximum recursion depth. As a demonstration of the validity of the classifications, I have in Figure 2 plotted two cases to show that it does, indeed, work properly. The relevant implementations are found in *classification.py*.

4. Solving the eigenvalue problem

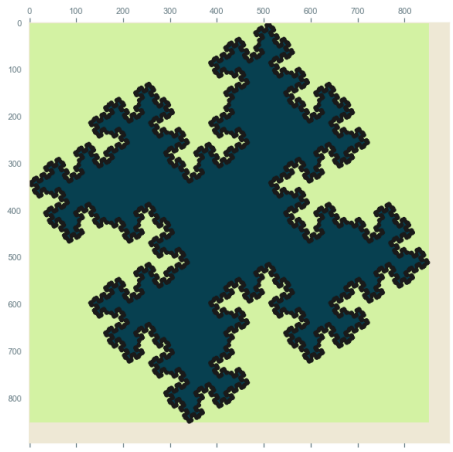
One important aspect to take into account is the importance of having $p > 0$. Consider for a brief moment that we set $p = 0$. Then, the outmost parts of the fractal are not even considered in computation. Physically we only expect these parts to be relevant at high energies, so that it might not be that relevant, but now we dont even take the fractal shape into account. Of this reason, the resulting calculations here are for $p > 0$. A visualization of this is seen in Figure 3. As a consequence of the implementation of the classification of points, some unwanted effects might come into account at the boundary for too low values of p.

The task of finding eigenvalues and eigenvectors are hard, and efficient computations rely heavily on the usage of optimized libraries. For this solution I have used Scipy[3] and its eigenvalue solver for sparse matrices. Scipy is installed with Intel's Math Kernel Library (MKL), and calls ARPACK in its backend. We are looking for the smallest eigenvalues λ of Equation (1), which is very slow compared to finding the largest values, according to the Scipy documentation. A suggestion is therefore to use the "Shift-Invert"-mode, also described in the documentation, but to use this i first have to find an approximation for the lowest energy. For example, the computation of the 100 smallest eigenvalues of the configuration (4, 1) took approximately

[1] "Euclidian product"; $(0, 0), (0, 1), \ldots, (1, 0), (1, 1), \ldots (M, M)$



(a) Classification of points with configuration (l,p) = (3,1)



(b) Classification of points with configuration (l,p) = (4,1)

Figure 2: The classification of points for two different configurations. The (far to small) numbers an the axes are the index (position) and are unimportant for the illustration.

2 hours 40 min[2]. Considering that this is in fact on average the computation of one eigenvalue and eigenvector of a 725904×725904-matrix in one and a half minute, where I used Equation (2), it does not seem terribly slow.

The results are indicative of a a successfull set up of the $M^2 \times M^2$-matrix. Since i am using arbitrary dimensions in the fractal set up, I am normalizing the eigenvalues such that the ground state energy maches that of ref. [4], where the ground state (or fundamental) frequency in $\Omega =$ $2.100\,\omega_0$. In this way, a comparison can be made by the relative energies (in units of ω_0 by dividing the normalizing the eigenfrequencies and divide by 2.100. The 10 smallest eigenvalues in this normalization scheme are presented in Table 2.

[2]Using the Shift-Invert-mode resulted in an error in ARPACK telling that the matrix is excactly singular

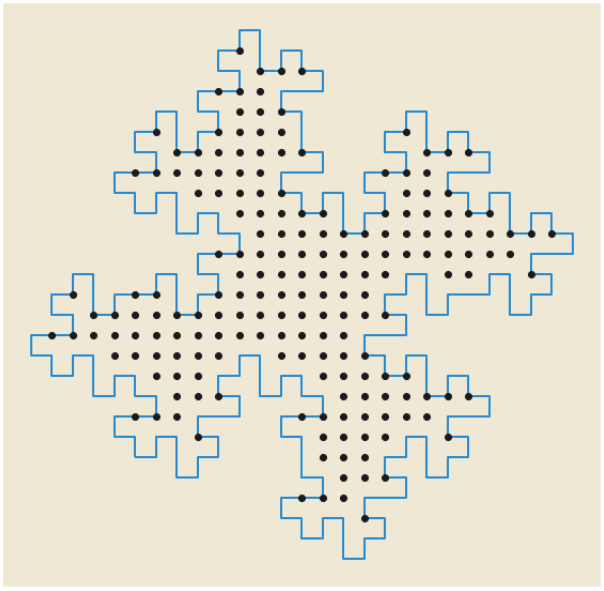


Figure 3: The effect of not having points in between the corners of the fractal. This is a level 2 fractal.

Eigenfrequencies in units ω_0

(lp)	(31)	(41)	(33)	(40)	(43)	Sapoval et.al
Ω_0	2.100	2.100	2.100	2.100	2.100	2.100
Ω_1	3.138	3.147	3.132	3.150	3.145	3.132
Ω_2	3.138	3.147	3.132	3.150	3.145	3.132
Ω_3	3.197	3.207	3.191	3.211	3.205	3.191
Ω_4	3.215	3.225	3.210	3.228	3.223	3.219
Ω_5	3.357	3.359	3.358	3.359		
Ω_6	3.357	3.359	3.358	3.359		
Ω_7	3.927	3.931	3.926	3.932		
Ω_8	4.203	4.209	4.200	4.212		
Ω_9	4.323	4.331	4.321	4.333		
Ω_{10}	4.323	4.331	4.325	4.333		

Table 2: The 10 smallest (normalized) eigenfrequencies of the fractal drum, paired with the results in ref. [4]

Notice that there are degenerate energies. This is a manifistation of the symmetry of the fractal, which has a 4-fold rotational symmetry. From a physical view, the energies and linearly independent solutions of a problem should reflect the symmetry group of the problem. In our case, the rotation group c_4 plus inversion symmetry is the most noteworthy. The ground state and the first couple of excited states are plotted in Figure 4 and the relative energies of the solutions can be read off in the corresponding

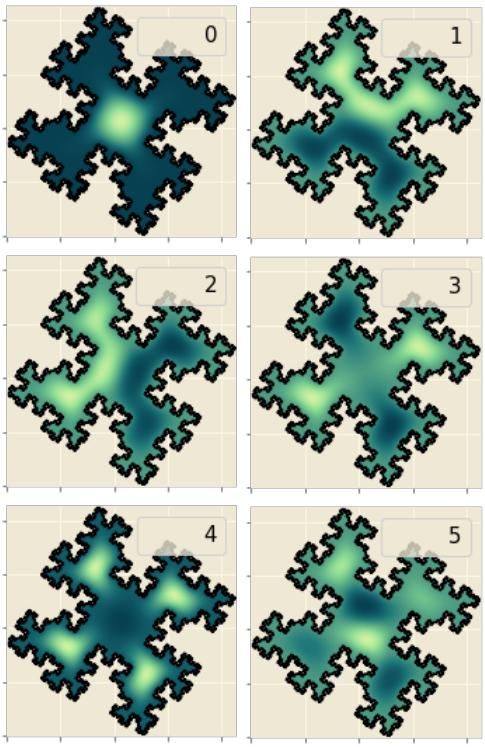


Figure 4: The ground state (0) and the five first excited states. Calculated for level $l = 4$ with $p = 1$ point between each corner. The colors on each subplot are not equivalent. Darker color corresponds to a lower value, and all plots have $U = 0$ on the boundary. The units along the axes are given by the index of the position in $[0, M]$

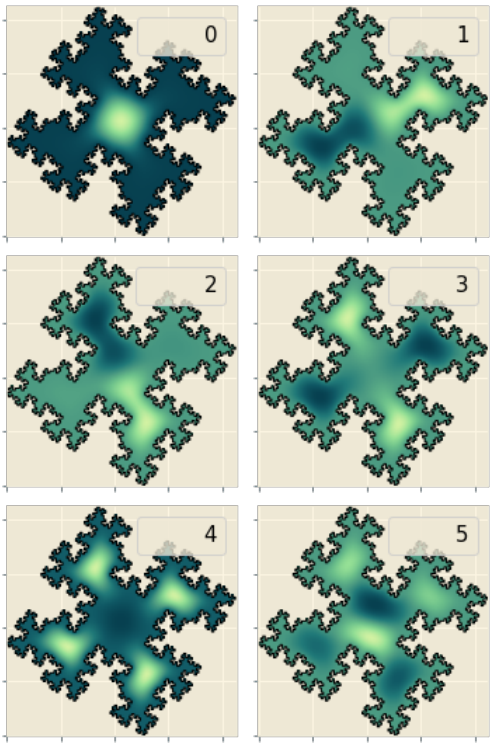


Figure 5: The first few exited states for $(l, p) = (3, 3)$

column in Table 2. The level of which I have been able to compute "efficiently" have been maximum $l = 4$, with one point between each corner. By comparing the results in Figure 4 and Figure 5, we see that the solutions (first and second excited states) are not the same. These states should have the same energy, and by superposing we can construct the states as linear combinations of states with equal energy. Mathematically this (as an example) may be written as

$$U^1_{(4,1)} = \frac{1}{2}\left(U^1_{(3,3)} - U^2_{(3,3)}\right),$$

and be an equally good solution for the eigenproblem. Having some results with the estimation of the ground state energy, I insert this estimation as a parameter in the eigensolver, and are now solving at shorter times. The constraints for the calculation of eigenvalues are strictly dependent on the size of the matrix, as previously mentioned. As an example, take the (4, 2)-configuration. This computation requires a sparse matrix with dimension $M^2 \times M^2$ with $M(4, 2) = 1278$, this corresponds to 2667616624656 bits using 64-bit (double precision) elements in the array. Considering the laptop I used, which turned itself off a multiple of times when the memory usage of the program was greater than 5GB, this calculation is impossible without using the sparse matrix format supported by scipy. For the (4, 2)-configuration, a dense matrix would require about 21TB of RAM. But the matrix used to solve the eigenequation is very sparse, and saves only 2777097 elements, or 22MB. The laptop I am using states that it has 7.89GB available RAM, but as aforementioned there are som serious problems with the capabilities of this machine. Given the same sparsity [3] as in the (4, 2)-case, the highest resolution the computer can handle is a (6, 2)-configuration, but even at (5, 1) memory-allocation errors start appearing. As previously discussed, the number of bits required scales as M^4. Thus, if we for example want to compute this equation on a level 8 fractal, we would require something like 1 Petabyte of memory – an impossible task without re-engineering the solution completely. To be able to solve these issues, one can for example use a dynamical grid, which has a much courser resolution in places where short-distance- physics are unimportant, far for the boundary. In this way, one really accelerates the solution and saves a lot of memory doing so.

[3] The sparsity should decrease for increasing l, there is a constant number of terms in each row, and the number of rows scales as M^2, not M^4. This is thus a rough estimate.

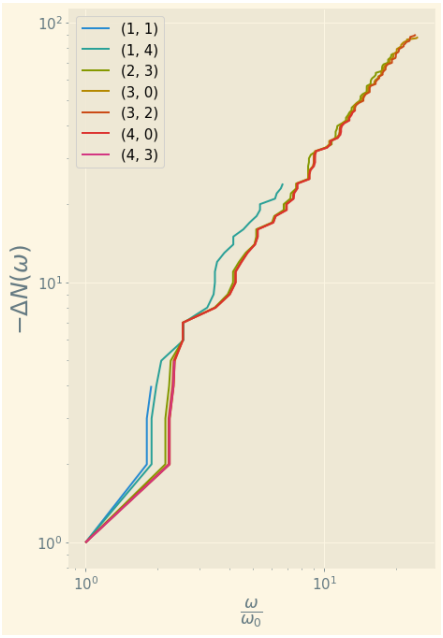


Figure 6: The calculated values for $\Delta N(\omega)$ for the relative eigenvalues. As we increase the number of frequencies and resolution, it suggests a straight line.

5. Scaling

We now turn to the scaling of $\Delta N(\omega)$, which are defined by the equations

$$A = 4\pi \lim_{\omega\to\infty} \frac{N(\omega)}{\omega^2} \tag{3}$$

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

Because of the way I have set up the system, this task have to be done in a different way than if i had defined the system with sidelenghts and kept myself to SI-units. This is because there is no concept of area in my system. Hovewer, the approach allows me to set $A = 1$, rewrite Equation (3) to be an equation for the relative eigenvalues $\lambda = \frac{\omega}{\omega_0}$, and use this to "renormalize" ω_0 for the different levels that I am calculating the scaling for. We are asked to find the scaling parameter d in

$$\Delta N(\omega) \sim \omega^d. \tag{5}$$

Taking the logarithm of Equation (5) and using linear regression on the "stable" parts will give a result for the scaling parameter d. This does, of course, rely on a good implementation of Equation (3), as many as possible eigenvalues, and a good resolution. In Figure 6, the value of $\Delta N(\omega)$ is calculated. A straight line in a log-log plot is a monomial, $y = ax^k$, and we are looking for the exponent. Using Numpy's [5] polyfit to one degree, we find that the scaling parameter varies between 1.5 and 1.6. Multiple calculations can be found in table Table 3. Note that we expect the calculations to be more consistent as we increase the number of eigenfrequencies, but that this requires much more computation for higher levels. Furthermore, the recursion level l does not seem to play a role, $d \neq d(l)$. What seems more surprising is that the number of points on each line segment seems to play a larger role in the fluctuating value of d. Also note that most of the calculations that had a larger amount of eigenvalues, and therefore also more data linearize, tends to be closer to 1.5 than the calculation that used fewer. This number was not something i controlled directly, but asking Scipy to give the 100 smallest eigenvalues also gives a couple that are most likely numerical artifacts, with eigenvalues of order 10^{-13} of the ground state.

(l, p)	d	# eigenvalues
$(1, 2)$	2.8589	11
$(1, 4)$	2.1226	24
$(2, 1)$	1.6704	49
$(2, 3)$	1.5255	76
$(2, 4)$	1.5075	80
$(3, 0)$	1.4915	88
$(3, 1)$	1.6351	45
$(3, 2)$	1.4676	90
$(3, 3)$	1.6487	43
$(4, 0)$	1.6261	46
$(4, 1)$	1.5034	75

Table 3: The scaling parameter found for different grid-configurations. The last column is the number of nonzero eigenvalues used in these calculations.

6. Higher order differential scheme

We now want to implement a higher order differential scheme. The scheme I am implementing is a fourth order finite difference scheme of the form (in one dimension)

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{x_i} = \frac{-u_{i+2} + 16u_{i+1} - 30u_i + 16u_{i-1} - u_{i-2}}{12h^4} \tag{6}$$

plus higher order terms in h. This will essentially just be adding four more diagonals in the matrix we are solving. A comparison between the second order and fourth order method is presented in Table 4. Note that the degeneracy in the first and second state seems to have vanished in the fourth-order case. This suggests an error in the implementation. The error was found at a much later time and is with regards to the positioning of the offset of y-terms in the matrix $\sim \frac{u_{i,j\pm 2}}{12h^4}$. This is a small quantity but large enough to give rise to false phenomena.

If we compare the 10 smallest eigenvalues of the fourth-and second order scheme, we see in Figure 8 that the eigenvalues coincide despite the implementation error. What I noticed here, is that it seems as if the eigenvalues of the fourth order method is more stable with respects to disturbances, because they seem to fluctuate a bit less than for the second order case.

$\omega[\omega_0]$	Second order	Fourth order
ω_0	1 (exact)	1 (exact)
ω_1	2.23	2.11
ω_2	2.23	2.24
ω_3	2.32	2.40
ω_4	2.34	2.47
ω_5	2.55	2.55

Table 4: The first few relative eigenvalues of the second and fourth order numerical scheme for the configuration (3,1).

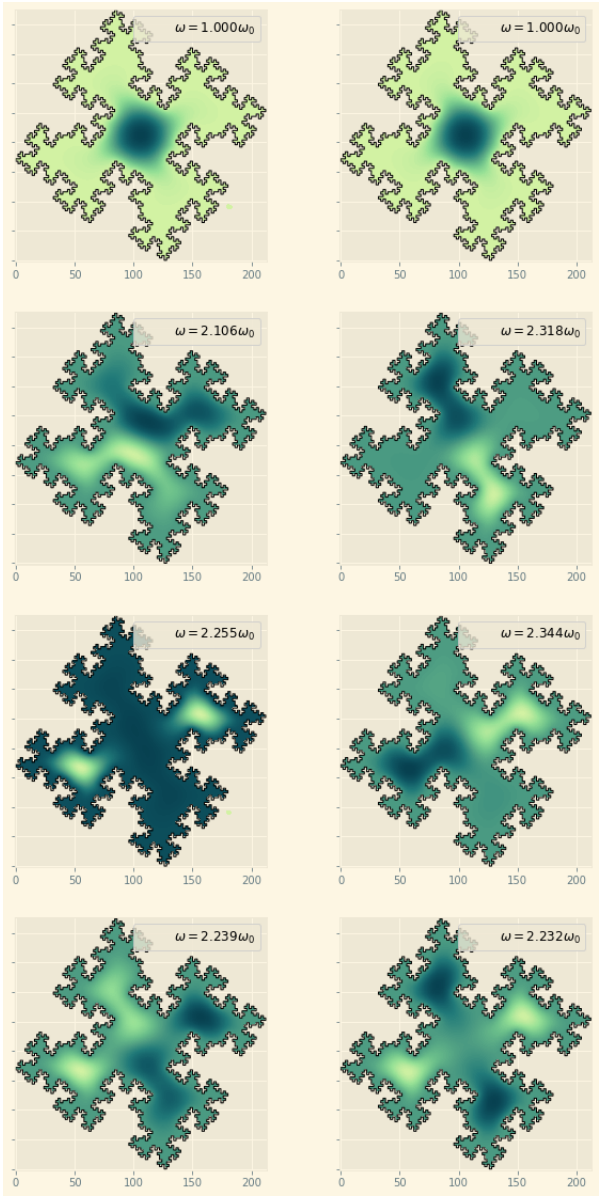


Figure 7: Some of the ground states of fourth (first column) and second (second column) order differention schemes.

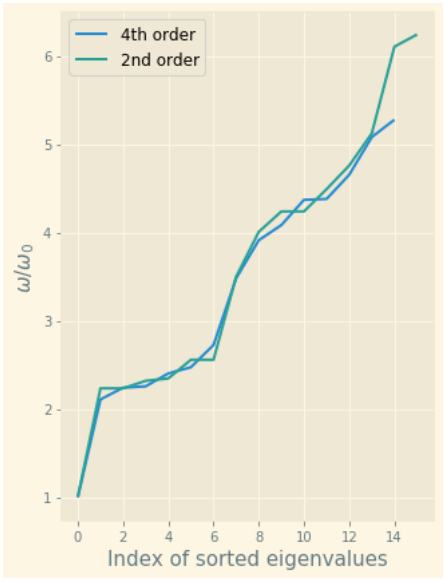


Figure 8: Plot of the 10 smallest eigenvalues of both fourth- and second order differention scheme.

7. Conclusion

In this assignment, we have been challenged to face the task of calculating eigenvalues and eigenvectors of a second order differential equation with complex boundary. Moreover, the task has for my part been an introduction to several new aspects of numerical solutions, and have made me learn a lot about recursion, sparse matrices, higher order finite difference methods, and how to implement complicated boundary conditions. The results match fairly well with that in ref [4] for the smallest couple of eigenvalues. The scaling parameter d is the part I am most unsure of, and I have not consulated an one else regarding the validity of $d \simeq 1.5$.

References

[1] Gillies Sean. Shapely 1.7.0. `https://github.com/Toblerity/Shapely`, 2020.

[2] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[3] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 2020.

[4] Bernard Sapoval, Th Gobron, and A Margolina. Vibrations of fractal drums. *Physical review letters*, 67(21):2974, 1991.

[5] Travis Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006–. [Online; accessed 23.02.2020].