Mesh error analysis and solution

ST5

Bowen ZHU

Date: Oct 2023

Contents

1 Error Analysis with Regular Mesh **2**

1.1 Error with Respect to Mesh Element Size: $\|u^* - u_h\|_2 \leq Ch^\alpha$ 2

1.2 Error with Respect to Wave Number: $\|u^* - u_h\|_2 \leq Ck^\beta$ 3

1.3 Combined Error Analysis: $\|u^* - u_h\|_2 \leq Ch^\alpha k^\beta$ 3

2 Error Analysis with Irregular Mesh **4**

2.1 Error Analysis: Element Size vs Shift Amplitude 5

2.2 Error Analysis: Wave Number vs Shift Amplitude 6

3 Solving the Helmholtz equation **7**

3.1 Question 1, solution domain 8

3.2 Question 2 9

3.2.1 preliminary 9

3.2.2 Setting the Neumann boundary condition 12

3.2.3 New plot function 13

3.2.4 Helmholtz function solution- plane wave incoming 14

3.3 question 3 & 4 17

3.3.1 Boundary Condition Homogenization 17

3.3.2 Existance surface 21

3.3.3 Quantification of Energy Dissipation 23

1 Error Analysis with Regular Mesh

1.1 Error with Respect to Mesh Element Size: $\|u^* - u_h\|_2 \leq Ch^\alpha$

Consider a square mesh of edge length 1 and triangular elements determined by *nelemx* elements horizontally. The mesh size is then defined as:

$$h = \frac{1}{nelemx}$$

Fixing the wave number at π, and taking the logarithm on both sides, we have:

$$\log \|u^* - u_h\|_2 \leq \log C + \alpha \log h$$

A log-log plot is used to determine the value of α. From the plot, we deduce that

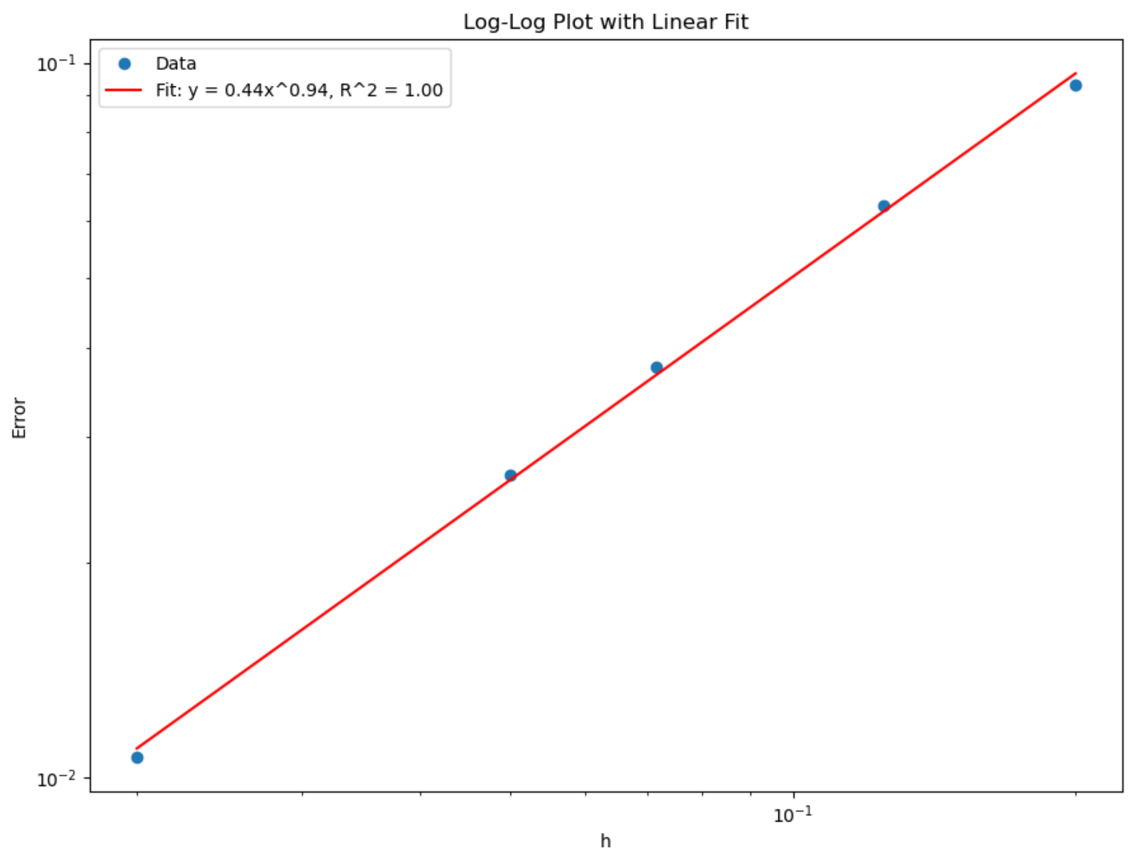


Figure 1: Log-log plot of error against element size.

$\alpha \approx 1$.

1.2 Error with Respect to Wave Number: $\|u^* - u_h\|_2 \le Ck^\beta$

For a mesh as described previously, and fixing the element size at $\frac{1}{30}$, we obtain:

$$\log\|u^* - u_h\|_2 \le \log C + \beta \log k$$

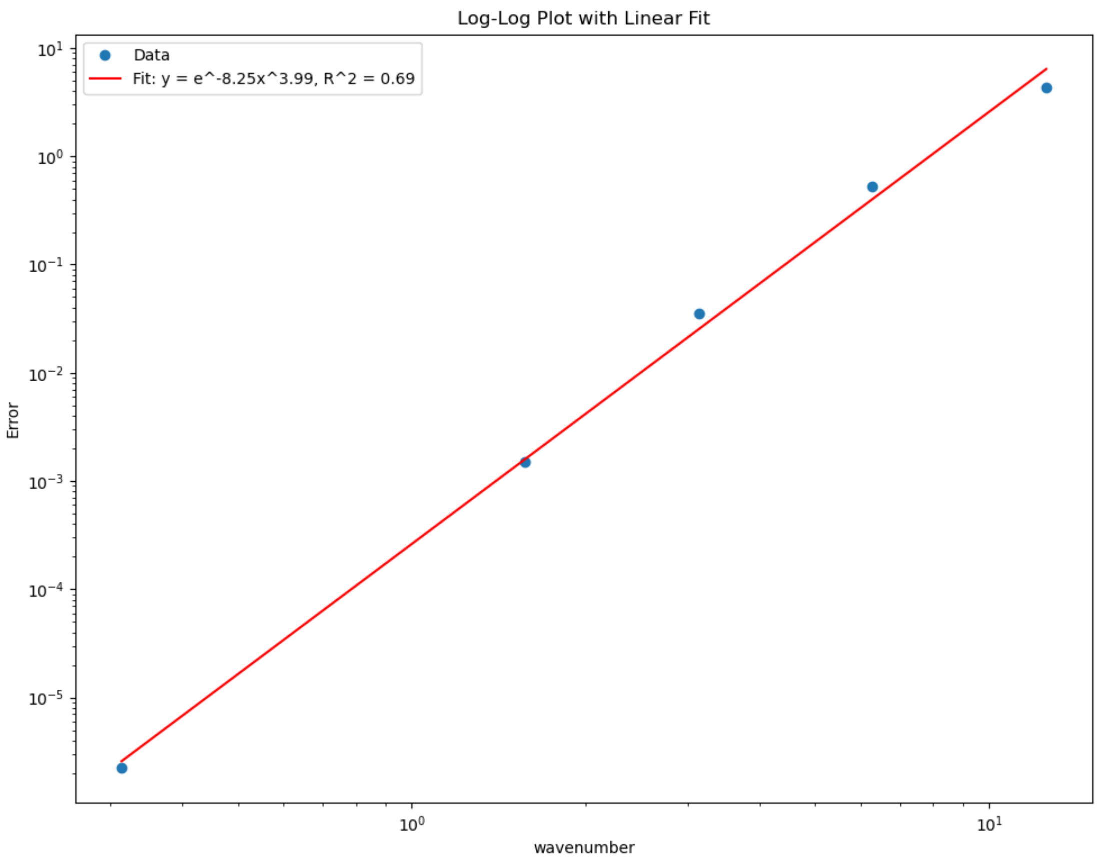


Figure 2: Log-log plot of error against wave number.

This yields $\beta \approx 4$.

1.3 Combined Error Analysis: $\|u^* - u_h\|_2 \le Ch^\alpha k^\beta$

The relationship in logarithmic form becomes:

$$\log\|u^* - u_h\|_2 \le \log C + \beta \log k + \alpha \log h$$

The relationship is predominantly linear, except when the element size or wave number becomes exceedingly large.

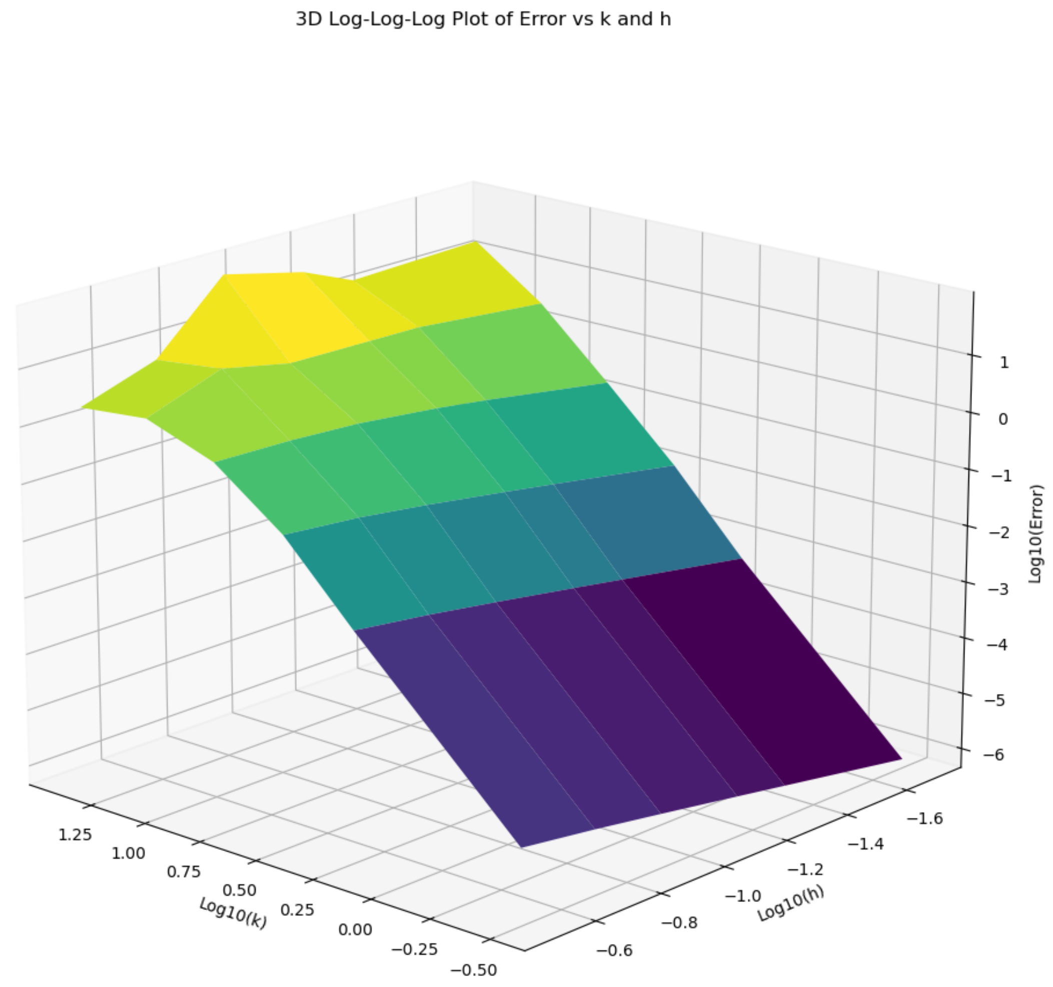


Figure 3: 3D log-log-log plot of error against wave number and element size.

Note: The exact thresholds for "large" element sizes or wave numbers should be experimentally determined to ensure robustness in applications.

2 Error Analysis with Irregular Mesh

In the context of an irregular mesh, we maintain consistency with the preceding section by defining the average element size. Given the mesh's irregularity due to shifting node positions, **both the total number of elements and the total area remain unchanged**. As such, the average element size h_{avg} is equivalent to the mesh size before shifting. For reproducibility, we set the random seed to 42 during node shifting.

The average element size is formulated as:

$$\text{Average element area} = \frac{\text{total area}}{\text{total number of elements}}$$
$$\text{Average element area} = \frac{h_{avg}^2}{2}$$

The Python function for node shifting is provided below:

```python
def random_shift_node(boundary_idx, node_coords, amp, seed=42):
    if seed is not None:
        np.random.seed(seed)

    mask = np.ones(node_coords.shape[0], dtype=bool)
    mask[boundary_idx] = False

    random_shifts = np.random.uniform(-amp, amp, size=node_coords
        .shape)
    node_coords[mask] += random_shifts[mask]

    return node_coords
```

2.1 Error Analysis: Element Size vs Shift Amplitude

The error relationship mirrors the previous section for small relative shift amplitudes (0.1 or 0.2). However, for shifts of 0.5 or larger, increased errors emerge for smaller h values, attributed to deteriorating element quality. This error does not significantly change for larger elements, even with increased shift amplitude.

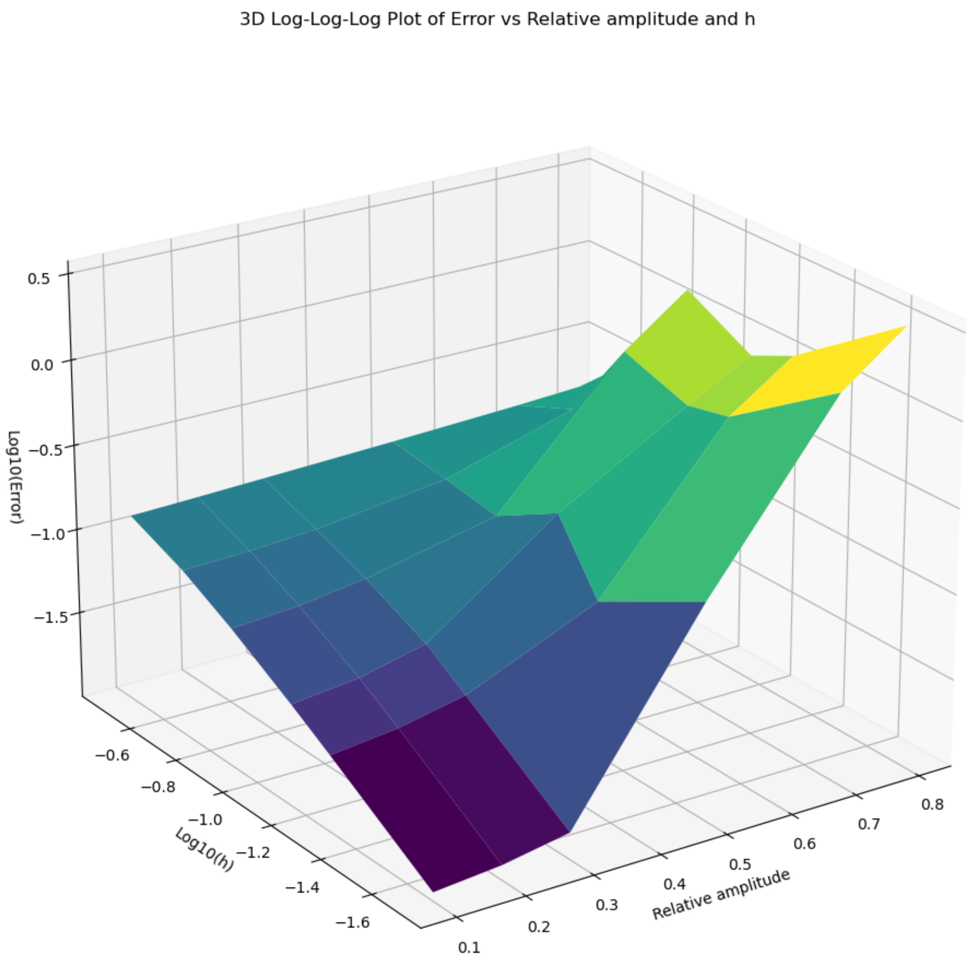


Figure 4: Error against element size and shift amplitude.

2.2 Error Analysis: Wave Number vs Shift Amplitude

Similar degradation in the error relationship with k is observed as the shifting amplitude rises.

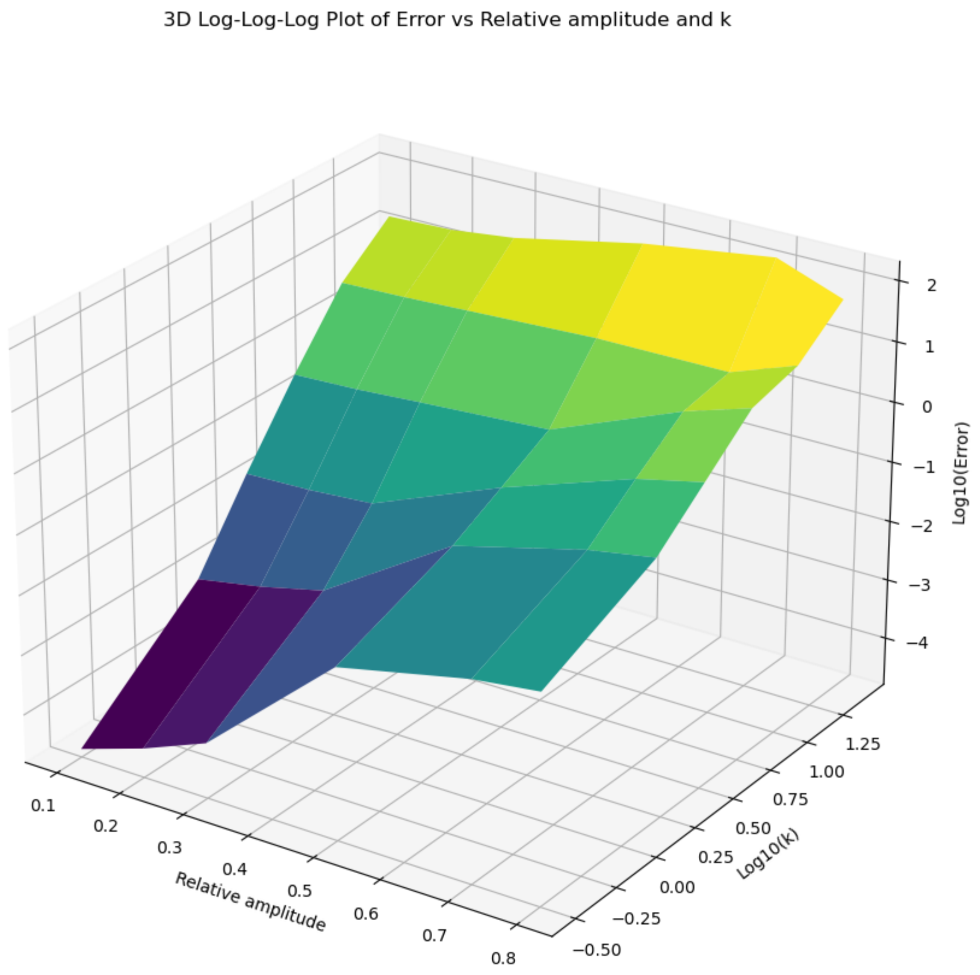


Figure 5: Error against wave number and shift amplitude.

3 Solving the Helmholtz equation

For the simplicity, we modify the boundary condition of the Helmholtz equation.

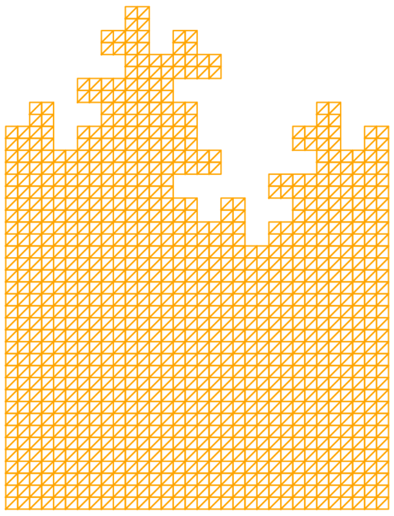
Consider a domain $\Omega = [0,1] \times [0,1]$, $\partial\Omega = \Gamma_d \cup \Gamma_n$, where Γ_d is the bottom boundary. The Helmholtz equation with following boundary condition is considered:

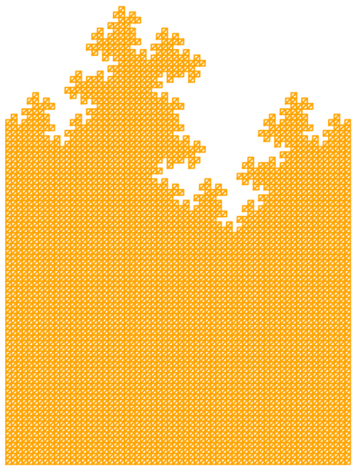
$$\begin{aligned} \Delta u + k^2 u = f, \text{in } \Omega \quad f = 0 \\ u = a, \text{on } \Gamma_d, \quad a \neq 0 \\ \frac{\partial u}{\partial n} = 0, \text{on } \Gamma_n \end{aligned}$$

This setup is to simulate the incoming sound into the room and encounter the 3 side reflective wall of different geometry.

3.1 Question 1, solution domain

We will consider the square koch curve up to degree of 3 in the following solution,


(a) koch curve degree 2


(b) koch curve degree 3

It should be noticed that the ploting function is changed for better plotting the solution with fractal boundary.

```python
def _plot_contourf(nelems, p_elem2nodes, elem2nodes, node_coords,
    node_data, **kwargs):
    """Plot node data parameter on the mesh."""
    x = node_coords[:, 0]
    y = node_coords[:, 1]
    z = node_data

    # create triangles for triangulation
    triangles = [elem2nodes[p_elem2nodes[i]:p_elem2nodes[i + 1]]
       for i in range(nelems)]

    # creates triangulation
    triang = matplotlib.tri.Triangulation(x, y, triangles)

    # Analyze triangulation to identify triangles that are inside
      holes
    analyzer = matplotlib.tri.TriAnalyzer(triang)
    mask = analyzer.get_flat_tri_mask(min_circle_ratio=0.01)
    triang.set_mask(mask)

    fig = matplotlib.pyplot.figure()
    axs = fig.add_subplot(projection='3d')
    axs.plot_trisurf(triang, z, cmap='viridis', edgecolor='none')
    matplotlib.pyplot.show()
```

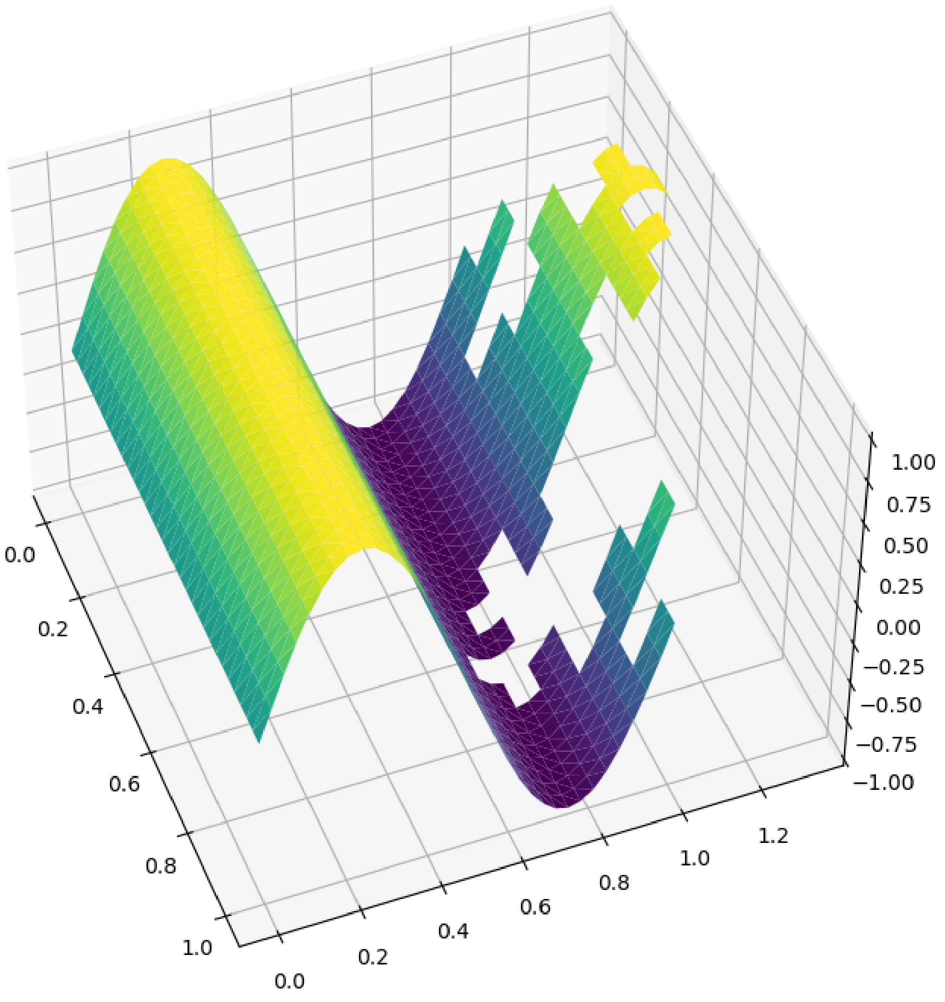


Figure 7: Plotting the demo solution

3.2 Question 2

3.2.1 preliminary

To modify the code and pose a neumann boundary condition, we must first understand how the code work to compute the solution.

The Helmholtz equation is given as:

$$\nabla^2 u + k^2 u = f \tag{1}$$

Applying the finite element method, we approximate u using shape functions N_i and unknown coefficients u_i:

$$u \approx \sum_{i=1}^{N} N_i u_i \tag{2}$$

With this approximation, the weak form of the Helmholtz equation, after integration by parts, becomes:

$$\int_{\Omega} \nabla N_i \cdot \nabla N_j \, d\Omega \, u_j - k^2 \int_{\Omega} N_i N_j \, d\Omega \, u_j = \int_{\Omega} N_i f \, d\Omega \tag{3}$$

Element-wise local stiffness matrix

$$K_{ij}^{(e)} = \int_{\Omega_e} \nabla N_i \cdot \nabla N_j \, d\Omega \tag{4}$$

Where $K^{(e)}$ is the local stiffness matrix for element e.

For a triangular element with nodes (x_1, y_1), (x_2, y_2), and (x_3, y_3), the linear shape functions N_i are defined as:

$$N_1 = \frac{1}{2A_e}(x_2y_3 - x_3y_2 + (y_2 - y_3)x + (x_3 - x_2)y) \tag{5}$$

$$N_2 = \frac{1}{2A_e}(x_3y_1 - x_1y_3 + (y_3 - y_1)x + (x_1 - x_3)y) \tag{6}$$

$$N_3 = \frac{1}{2A_e}(x_1y_2 - x_2y_1 + (y_1 - y_2)x + (x_2 - x_1)y) \tag{7}$$

Where A_e is the area of the triangle and is given by:

$$A_e = \frac{1}{2} \left| x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2) \right| \tag{8}$$

The gradients of these shape functions are constant within the triangle and are given by:

$$\nabla N_1 = \begin{bmatrix} \frac{y_2 - y_3}{2A_e} \\ \frac{x_3 - x_2}{2A_e} \end{bmatrix} \tag{9}$$

$$\nabla N_2 = \begin{bmatrix} \frac{y_3 - y_1}{2A_e} \\ \frac{x_1 - x_3}{2A_e} \end{bmatrix} \tag{10}$$

$$\nabla N_3 = \begin{bmatrix} \frac{y_1 - y_2}{2A_e} \\ \frac{x_2 - x_1}{2A_e} \end{bmatrix} \tag{11}$$

Considering the linear nature of the shape functions N_i, and since their gradients are constant over the triangular domain, this expression simplifies to:

$$K_{ij}^{(e)} = area(\Omega_e)\nabla N_i \cdot \nabla N_j \tag{12}$$

Element-wise local mass matrix The local mass matrix for a triangular element, using the provided shape functions, is:

$$M_{ij}^{(e)} = \int_{\Omega_e} N_i N_j \, d\Omega$$

Given the shape functions

Integrating over the triangle(with some calculation), we get:

$$M_{ii}^{(e)} = \frac{A_e}{6}$$

and

$$M_{ij}^{(e)} = \frac{A_e}{12} \quad \text{for} \quad i \neq j$$

Element-wise local RHS term Given a triangular finite element, the elemental force term for each node can be expressed as:

$$F_i^{(e)} = \int_{\Omega_e} N_i(x,y) f(x,y) \, d\Omega \tag{13}$$

To obtain a computable form, we approximate f over the element using its nodal values:

$$f(x,y) \approx f_1 N_1(x,y) + f_2 N_2(x,y) + f_3 N_3(x,y) \tag{14}$$

Substituting this approximation into the force term yields:

$$F_i^{(e)} = \int_{\Omega_e} N_i(x,y) \left(f_1 N_1(x,y) + f_2 N_2(x,y) + f_3 N_3(x,y)\right) d\Omega \tag{15}$$

This can be expressed in matrix-vector form for all the nodes of the triangular element:

$$\mathbf{F^{(e)}} = \mathbf{M_e f} \tag{16}$$

Assemble:Direct Stiffness Method Direct Stiffness Method

```
for i in range(0, 3):
    F[nodes[i]] = F[nodes[i]] + Fe[i]
    for j in range(0, 3):
        K[nodes[i], nodes[j]] = K[nodes[i], nodes[j]] + coef_k[e]
            * Ke[i, j]
        M[nodes[i], nodes[j]] = M[nodes[i], nodes[j]] + coef_m[e]
            * Me[i, j]
```

Dirichlet boundary condition Given the finite element system:

$$\begin{bmatrix} A_{ii} & A_{ip} \\ A_{pi} & A_{pp} \end{bmatrix} \begin{bmatrix} x_i \\ x_p \end{bmatrix} = \begin{bmatrix} b_i \\ b_p \end{bmatrix}$$

Where:

- x_i: Unknown values (internal nodes).
- x_p: Prescribed values (Dirichlet nodes).
- A_{ii}, A_{ip}, A_{pi}, A_{pp}: Submatrices of A.
- b_i, b_p: Subvectors of $\mathbf{b}$.

The Dirichlet boundary condition sets $x_p = b_p$. To impose these conditions, the system is transformed as:

$$\begin{bmatrix} A_{ii} & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} x_i \\ x_p \end{bmatrix} = \begin{bmatrix} b_i - A_{ip} b_p \\ b_p \end{bmatrix}$$

Where I is the identity matrix. This transformation effectively enforces $x_p = b_p$, which is the Dirichlet condition, and adjusts the internal forces b_i to account for the fixed boundary values.

3.2.2 Setting the Neumann boundary condition

Given the differential equation:

$$-\frac{d^2u}{dx^2} = f \quad \text{in} \quad \Omega \tag{17}$$

with boundary conditions:

$$u = g_D \qquad \text{on} \quad \partial\Omega_D \tag{18}$$

$$\frac{du}{dx} = g_N \qquad \text{on} \quad \partial\Omega_N \tag{19}$$

where $\partial\Omega_D$ and $\partial\Omega_N$ denote the Dirichlet and Neumann boundaries, respectively.

Multiplying both sides of the differential equation by a test function v and integrating over the domain Ω, we have:

$$\int_\Omega -v \frac{d^2u}{dx^2}\, dx = \int_\Omega vf\, dx \tag{20}$$

Applying integration by parts:

$$\int_\Omega \frac{dv}{dx}\frac{du}{dx}\, dx - \int_{\partial\Omega} v\frac{du}{dx}\, ds = \int_\Omega vf\, dx \tag{21}$$

If Neumann boundary conditions are specified, we modify the boundary term:

$$\int_\Omega \frac{dv}{dx}\frac{du}{dx}\, dx - \int_{\partial\Omega_N} vg_N\, ds = \int_\Omega vf\, dx \tag{22}$$

It's noteworthy that when the Neumann boundary condition is 0 (i.e., $g_N = 0$), the boundary integral term disappears, thus it's inherently accounted for in the variational formulation and does not require additional treatment.

3.2.3 New plot function

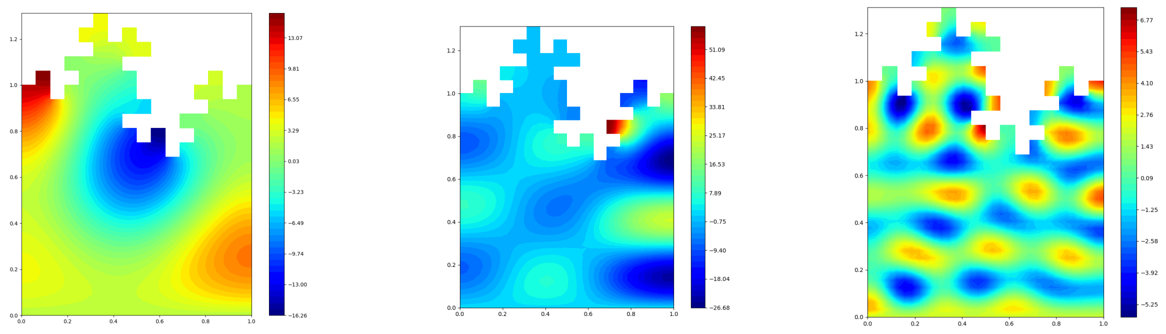
To better show case the wave form, a new 2d plot function is used instead of the 3d one.

```
def _plot_2d_contour(nelems, p_elem2nodes, elem2nodes, node_coords, node_data, **kwargs):
    """Plot node data parameter on the mesh as 2D contours."""
    x = node_coords[:, 0]
    y = node_coords[:, 1]
    z = node_data

    # create triangles for triangulation
    triangles = [elem2nodes[p_elem2nodes[i]:p_elem2nodes[i + 1]] for i in range(nelems)]

    # creates triangulation
    triang =matplotlib.tri.Triangulation(x, y, triangles)

    # Analyze triangulation to identify triangles that are inside holes
    analyzer = matplotlib.tri.TriAnalyzer(triang)
    mask = analyzer.get_flat_tri_mask(min_circle_ratio=0.01)
    triang.set_mask(mask)
```

```python

    fig, ax = matplotlib.pyplot.subplots()
    levels = numpy.linspace(min(z), max(z), 50)  # setting contour levels
    contour = ax.tricontourf(triang, z, levels=levels, cmap='jet', **kwargs)
    fig.colorbar(contour)
    ax.set_aspect('equal')
    matplotlib.pyplot.show()
```

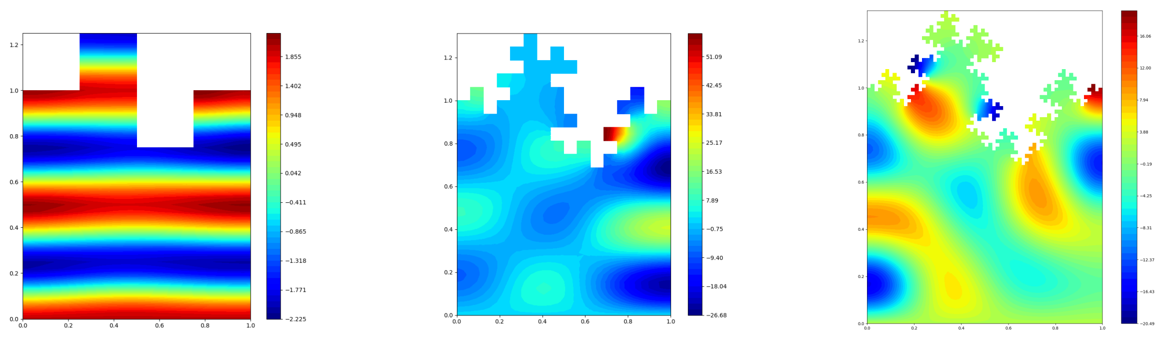
3.2.4 Helmholtz function solution- plane wave incoming

```python
def plane_wave(wavenumber, degree, a, plot_type="2d",alignment="right"):
    # Depending on the 'degree', select the appropriate mesh resolution
    if degree > 2:
        if alignment=="right":
            node_coords, p_elem2nodes, elem2nodes, boundary_idx = fractal_mesh_tri(64, 0, 16, degree)
        else:
            node_coords, p_elem2nodes, elem2nodes, boundary_idx = fractal_mesh_quad(64, 0, 16, degree)
            node_coords, p_elem2nodes, elem2nodes=quad2tri(node_coords, p_elem2nodes, elem2nodes)
    else:
        if alignment=="right":
            node_coords, p_elem2nodes, elem2nodes, boundary_idx = fractal_mesh_tri(32, 0, 8, degree)
        else:
            node_coords, p_elem2nodes, elem2nodes, boundary_idx = fractal_mesh_quad(32, 0, 8, degree)
            node_coords, p_elem2nodes, elem2nodes=quad2tri(node_coords, p_elem2nodes, elem2nodes)
    # Determine the number of nodes and elements from the mesh data
    nnodes = node_coords.shape[0]
    nelems = len(p_elem2nodes) - 1

    # Plot the mesh using the provided node and element data
    fig = matplotlib.pyplot.figure(1)
    ax = matplotlib.pyplot.subplot(1, 1, 1)
    ax.set_aspect('equal')
    ax.axis('off')
    solutions1._plot_mesh(p_elem2nodes, elem2nodes, node_coords,
        color='orange')
    matplotlib.pyplot.show()

    # Build the mapping from nodes to elements
    p_node2elems, node2elems = build_node2elems(p_elem2nodes,
        elem2nodes)

    # Identify the nodes lying on the southern boundary
    nodes_on_south = solutions1._set_square_nodes_boundary_south(
        node_coords)
    nodes_on_boundary = nodes_on_south

    # Set Dirichlet boundary conditions
    values_at_nodes_on_boundary = np.zeros((nnodes, 1), dtype=np.
        complex128)
    values_at_nodes_on_boundary[nodes_on_boundary] = a

    # Prepare finite element matrices and the right-hand side
    f_unassembled = np.zeros((nnodes, 1), dtype=np.complex128)
    coef_k = np.ones((nelems, 1), dtype=np.complex128)
    coef_m = np.ones((nelems, 1), dtype=np.complex128)
    K, M, F = solutions1._set_fem_assembly(p_elem2nodes,
        elem2nodes, node_coords, f_unassembled, coef_k, coef_m)

    # Formulate the system matrix A
    A = K - wavenumber**2 * M
    B = F



    # Apply the Dirichlet boundary conditions to the system
    A, B = solutions1._set_dirichlet_condition(nodes_on_boundary,
         values_at_nodes_on_boundary, A, B)
    print(np.linalg.matrix_rank(A))
    print(np.linalg.matrix_rank(np.hstack([A, B])))

    # Solve the system. If A is singular, use the pseudo-inverse
    try:
        sol = scipy.linalg.solve(A, B)
        print("not singular")
    except:
        print("singular")
        pseudo_inv_A = scipy.linalg.pinv(A)
        sol = pseudo_inv_A.dot(B)

    # Plot the solution (either in 3D or 2D) based on the provided 'plot_type'
    solreal = sol.reshape((sol.shape[0],))
    if plot_type == "3d":
        _ = solutions1._plot_contourf(nelems, p_elem2nodes, elem2nodes, node_coords, np.real(solreal))
    else:
        _ = solutions1._plot_2d_contour(nelems, p_elem2nodes, elem2nodes, node_coords, np.real(solreal))
```



(a) k=2 π (b) k=4 π (c) k=8 π

Figure 8: degree 2 fractal boundary with varying wavenumber



(a) Order=1 (b) Order=2 (c) Order=3

Figure 9: constant wavenumber k=4pi fractal boundary with varying degree(order)

3.3 question 3 & 4

3.3.1 Boundary Condition Homogenization

In order to homogenize the Helmholtz equation, we define:

$$u = v + a \tag{23}$$

Integrating this into the Helmholtz equation and recognizing a as a constant, we get:

$$\begin{aligned} \Delta v + k^2 v &= -k^2 a \quad \text{in } \Omega \\ v &= 0 \quad \text{on } \Gamma_d \\ \frac{\partial v}{\partial n} &= 0 \quad \text{on } \Gamma_n \end{aligned}$$

It's evident that the eigenvalues of this equation coincide with the original due to a constant shift by a. Thus, we've established that the focus should be on finding the eigenvalues and the eigenmodes of the revised equation.

Determining the Eigenvalues and Eigenmodes Consider a domain defined as:

$$\Omega = [0, 1] \times [0, 1] \tag{24}$$

with the boundary:

$$\partial\Omega = \Gamma_d \cup \Gamma_n \tag{25}$$

and Γ_d representing the bottom boundary. The Helmholtz equation, given these boundary conditions, is:

$$\begin{aligned} \Delta u + k^2 u = f &\quad \text{in } \Omega \\ u = 0 &\quad \text{on } \Gamma_d \\ \frac{\partial u}{\partial n} = 0 &\quad \text{on } \Gamma_n \end{aligned}$$

Aligning with the Sturm-Liouville system:

$$\Delta \phi = \lambda \phi \tag{26}$$

Upon discretization, we derive:

$$K\phi = \lambda M \phi \tag{27}$$

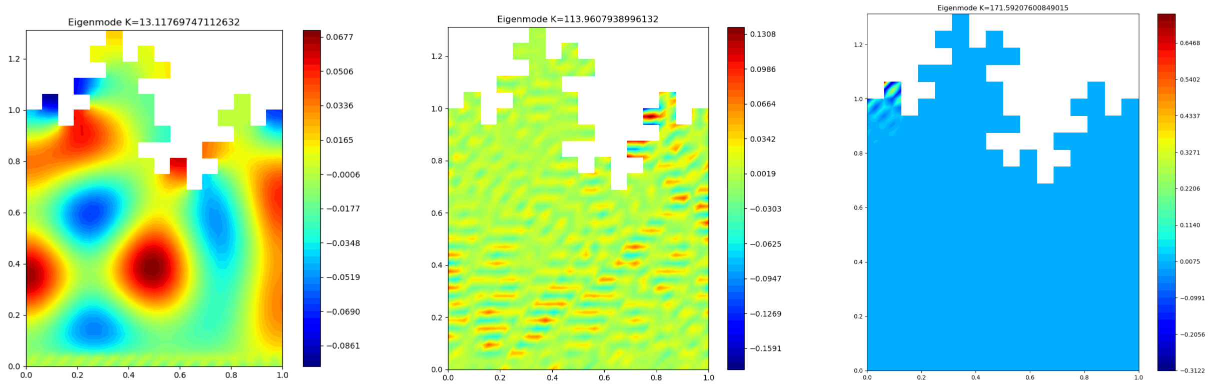
We employ a bespoke function to apply the 0 Dirichlet boundary condition, which yields the condensed system.

```python
def _set_dirichlet_condition_eig(K, M, dirichlet_nodes):
    """Apply Dirichlet boundary conditions to K and M matrices by
        removing rows and columns.

    Parameters:
        K (ndarray): Stiffness matrix
        M (ndarray): Mass matrix
        dirichlet_nodes (list): List of node indices with
            Dirichlet conditions
    """

    # Sort dirichlet_nodes in descending order to avoid index
        issues when deleting
    dirichlet_nodes = sorted(dirichlet_nodes, reverse=True)
    for node in dirichlet_nodes:
        # Delete the row and column in K
        K = numpy.delete(K, node, axis=0)
        K = numpy.delete(K, node, axis=1)

        # Delete the row and column in M
        M = numpy.delete(M, node, axis=0)
        M = numpy.delete(M, node, axis=1)
    return K, M
```

and we solve for the eigenvalue. We solve for the first 20 eigenvalue for efficiency.

```python
def find_eig(degree,alignment="right"):
    if degree > 2:
        if alignment=="right":
            node_coords, p_elem2nodes, elem2nodes, boundary_idx = fractal_mesh_tri(64, 0, 16, degree)
        else:
            node_coords, p_elem2nodes, elem2nodes, boundary_idx = fractal_mesh_quad(64, 0, 16, degree)
            node_coords, p_elem2nodes, elem2nodes=quad2tri(node_coords, p_elem2nodes, elem2nodes)
    else:
        if alignment=="right":
            node_coords, p_elem2nodes, elem2nodes, boundary_idx = fractal_mesh_tri(32, 0, 8, degree)
        else:
            node_coords, p_elem2nodes, elem2nodes, boundary_idx = fractal_mesh_quad(32, 0, 8, degree)
            node_coords, p_elem2nodes, elem2nodes=quad2tri(node_coords, p_elem2nodes, elem2nodes)

    # Determine the number of nodes and elements from the mesh data
    nnodes = node_coords.shape[0]
    nelems = len(p_elem2nodes) - 1
    # Build the mapping from nodes to elements
    p_node2elems, node2elems = build_node2elems(p_elem2nodes, elem2nodes)

    # Identify the nodes lying on the southern boundary
    nodes_on_south = solutions1._set_square_nodes_boundary_south(node_coords)
    nodes_on_boundary = nodes_on_south



    # Prepare finite element matrices and the right-hand side
    f_unassembled = np.zeros((nnodes, 1), dtype=np.complex128)
    coef_k = np.ones((nelems, 1), dtype=np.complex128)
    coef_m = np.ones((nelems, 1), dtype=np.complex128)
    K, M, F = solutions1._set_fem_assembly(p_elem2nodes,
        elem2nodes, node_coords, f_unassembled, coef_k, coef_m)
    K_reduced, M_reduced=solutions1._set_dirichlet_condition_eig(
        K, M, nodes_on_boundary)
    # Apply the Dirichlet boundary conditions to the system
    # eigenvalues, eigenvectors = eigsh(K, k=20, which='LM', M=M)
    if degree==3:
        eigenvalues, eigenvectors = eigsh(K_reduced, k=50, which=
            'LM', M=M_reduced)
    else:
        eigenvalues, eigenvectors = eig(K_reduced, M_reduced)

    sorted_indices = np.argsort(np.real(eigenvalues))
    sorted_eigenvalues = np.real(eigenvalues)[sorted_indices]
    sorted_eigenvectors = eigenvectors[:, sorted_indices]
    sorted_eigenvectors = np.insert(sorted_eigenvectors,
        nodes_on_boundary, 0, axis=0)
    return np.sqrt(sorted_eigenvalues ),sorted_eigenvectors,
        nelems, p_elem2nodes, elem2nodes, node_coords,boundary_idx
```

Following is the a few selected eigenmodes:



(a) degree 2 fractal, $k = 13$ (b) degree 2 fractal, $k = 113$ (c) degree 2 fractal, $k = 171$

Figure 10: Eigenmodes demonstration

Resonance phenomena Upon identifying the eigenvalue, one can insert its square root as k into the `plane_wave` function to deduce the helmholtz solution that is excited by such wave number.

Incorporating this eigenvalue into the Helmholtz equation with the inhomogeneous boundary condition unveils a peculiarity: the system becomes unsolvable. This is evident from the observation that

$$\text{Rank}(A) = n - 1 < n$$

where $A = K - k^2 M$, and simultaneously

$$\text{Rank}[A\ B] = n$$

This implies that solutions either don't exist or are infinite. Such a phenomenon is termed 'resonance'. The absence of solutions doesn't negate the possibility of such situations in reality. Instead, it underscores the constraints of the Helmholtz equation, which exclusively contemplates linear perturbations. As amplitude intensifies, perturbations surpass the linear limits, venturing into a nonlinear domain not captured by the Helmholtz equation. Nevertheless, resonance behaviors can be probed or approximated by making marginal adjustments to the eigenvalue, ensuring the equation's solvability.

3.3.2 Existance surface

In the discretized FEM solution we have the eigenmodes' value on every nodes of the mesh. In a zeroth-order approximation, each node is attributted a square around the nodes, and the approximated value in the square is the value of the nodes, this way it is very easy to caluclated the intergration.

say the nodal value is gathered in a vector ϕ_{vec}. then

$$\int |\phi|^2 dxdy \approx Area \times \phi_{vec}^T \phi_{vec} / number\ of\ nodes$$

In our case, the area is 1 due to the systry nature of fractal. and the eigenvector is aready normalized.

so

$$Area \times \phi_{vec}^T \phi_{vec} = 1$$

But we need

$$\int |\phi_{norm}|^2 dxdy = 1$$

Therefore we need to scale once again the eigenvectors

$$\phi_{vec}^{norm} = \phi_{vec} \times (number\ of\ nodes)^{0.5}$$

Therefore the existance surface gives:

$$\frac{1}{\int |\phi_{norm}|^4 dxdy} = \frac{number\ of\ node}{\Sigma |\phi_{vec}^{norm} i|^4}$$

```python
def plot_existance_surface(ks_list, eigenvectors_list, labels=None):
    fig, ax = plt.subplots()

    markers = ['o', 's', '^', '*', 'D']  # circle, square, triangle, star, diamond

    for i in range(len(ks_list)):
        ks = ks_list[i]
        eigenvectors = eigenvectors_list[i]

        power4 = np.real(eigenvectors)**4
        n_nodes = len(eigenvectors)
        existance_surface = 1 / (n_nodes * np.sum(power4,axis=0))

        ax.scatter(ks / np.pi, existance_surface, label=(labels[i] if labels else f"Set {i+1}"), s=80-10*i, marker=markers[i % len(markers)])

    ax.set_title("Existance Surface")
    ax.set_xlabel("k/pi")
    ax.set_ylabel("Existance Surface")

    if labels:
        ax.legend()

    plt.show()
```

Existence Surface Across Fractal Degrees The relationship between the existence surface and the boundary fractal order, as well as the wavenumber, is visualized in the figure below.

From the presented graph, it's evident that as k increases, the existence surface exhibits periodic fluctuations, typically trending downwards. Furthermore, as the degree of irregularity (or fractal level) heightens, the existence surface for the eigenmodes similarly diminishes.

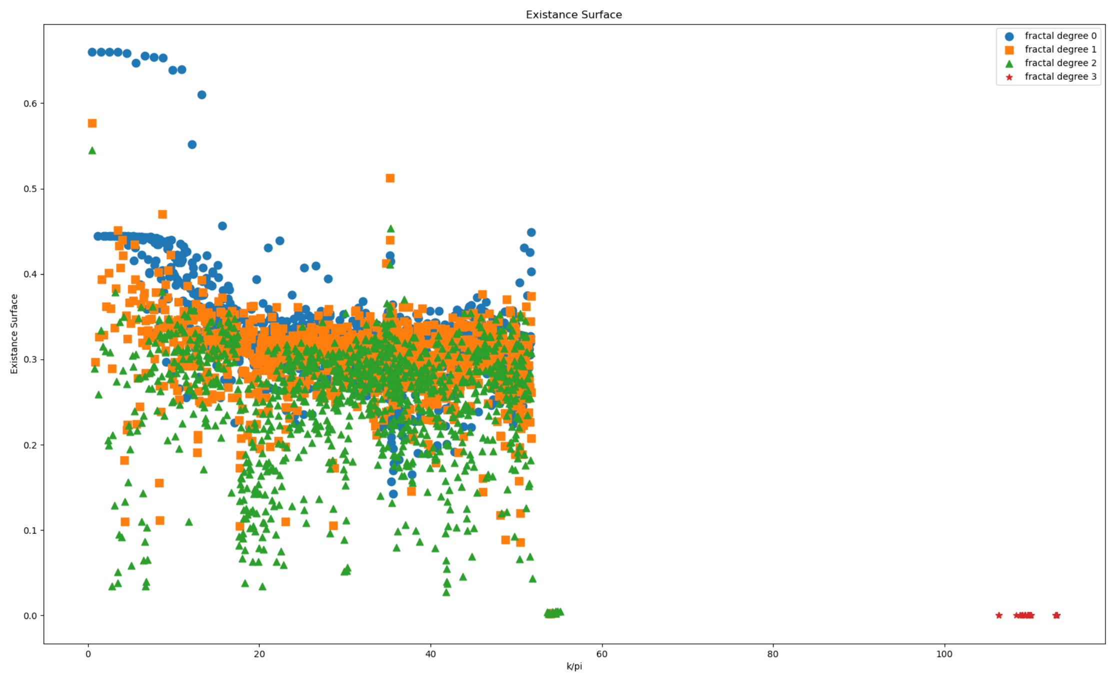


Figure 11: In this illustration, blue dots correspond to a boundary fractal degree of 0 (a flat line). Orange squares represent degree 1 fractals, green for degree 2, and red for degree 3. Due to computational constraints, only a subset of eigenmodes for degree 3 is calculated.

3.3.3 Quantification of Energy Dissipation

Energy dissipation, denoted as w_j, is expressed as:

$$w_j = \int_\Gamma |\phi_{norm}|^2 ds$$

Given that $\int |\phi_{norm}|^2 dxdy = 1$, where Γ signifies the boundary.

For numerical approximation, we employ the order 0 approximation. Let the element size be h and the number of nodes on the boundary be n_b. Since all elements share the same size, the boundary length, L_b, is given by $n_b \times h$. The normalized boundary nodal value is represented as ϕ^{norm}_{bvec}. Therefore, the approximation is:

$$\int_\Gamma |\phi_{norm}|^2 ds \approx h \times \phi^{normT}_{bvec} \phi^{norm}_{bvec}$$

```python
def plot_dissipated_energy(ks_list, eigenvectors_list,
    boundary_ls, h_ls, labels=None):
    fig, ax = plt.subplots()

    markers = ['o', 's', '^', '*', 'D']  # marker styles

    for i, (ks, eigenvectors, h, boundary_idx) in enumerate(zip(ks_list, eigenvectors_list, h_ls, boundary_ls)):
        eigenvectors_norm = eigenvectors * np.sqrt(len(eigenvectors))
        power2_b = np.real(eigenvectors_norm[boundary_idx, :])**2
        dissipated_energy = h * np.sum(power2_b, axis=0)
        ax.scatter(ks / np.pi, dissipated_energy, label=(labels[i] if labels else f"Set {i+1}"), s=80-10*i, marker=markers[i % len(markers)])

    ax.set_title("Dissipated Energy vs. Wavenumber")
    ax.set_xlabel("k/pi")
    ax.set_ylabel("Dissipated Energy")

    if labels:
        ax.legend()

    plt.show()
```

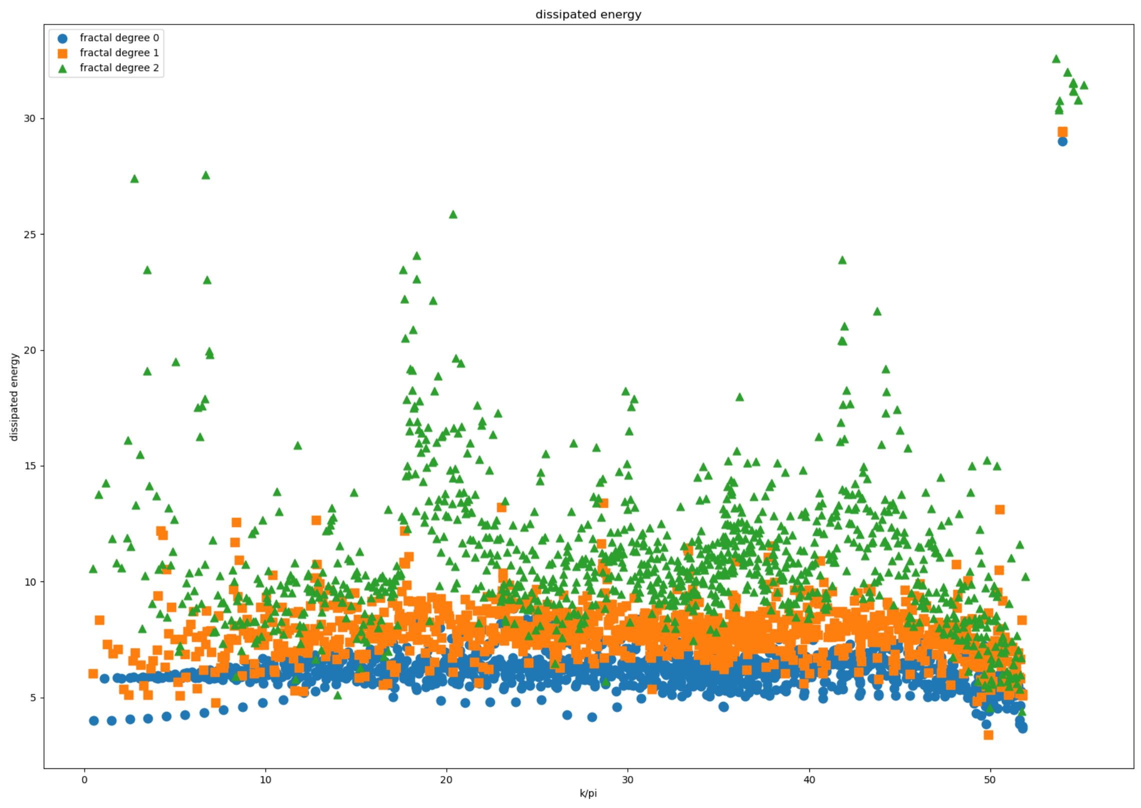


Figure 12: The blue dots signify a boundary fractal degree of 0 (a flat line), with orange squares and green triangles denoting degree 1 and degree 2 fractals, respectively.

Energy Dissipation Across Various Fractal Degrees Overall, the behavior of energy dissipation meets expectations. There's an observed increase in energy dissipation with the growth of fractal degree or irregularity. Notably, the energy dissipation for degree 3 fractals isn't displayed alongside the others due to its significantly higher values, often surpassing the largest values observed in degree 2 fractals.

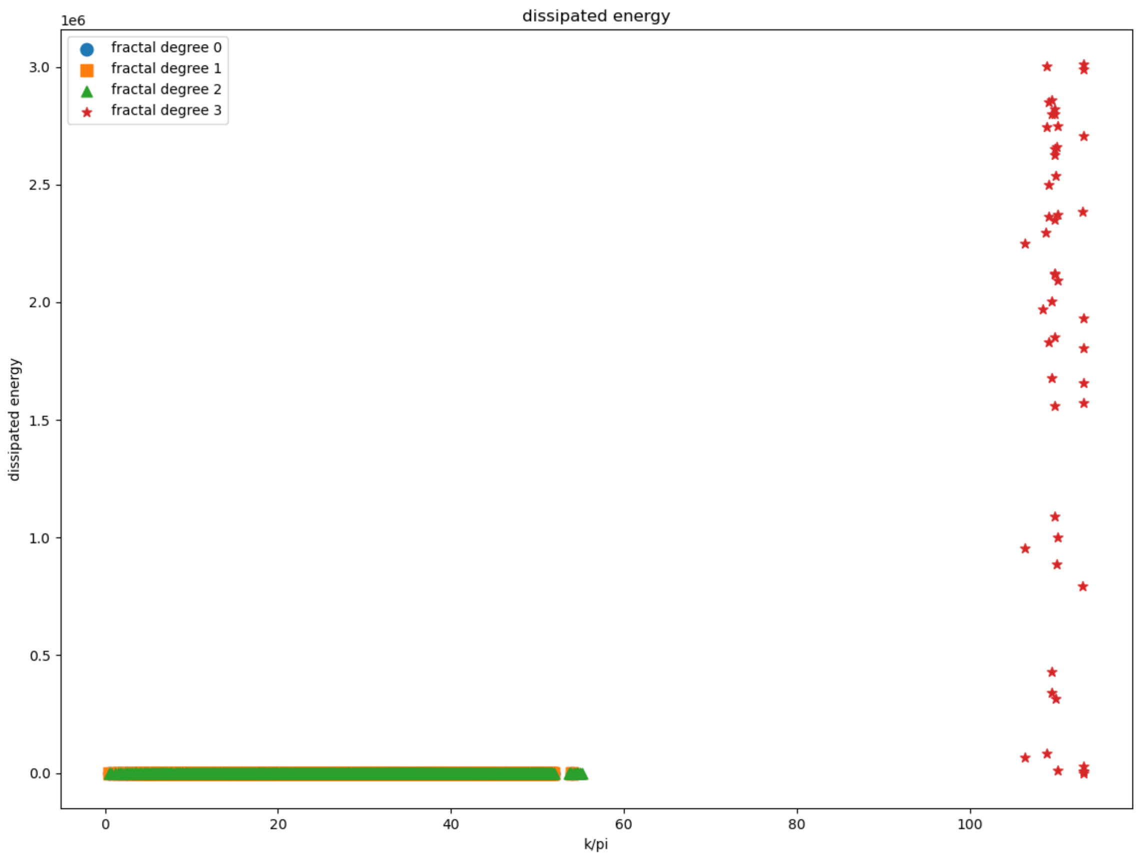


Figure 13: Representations include blue dots for degree 0, orange squares for degree 1, green triangles for degree 2, and red diamonds for degree 3. This highlights the considerable energy dissipation observed in degree 3 fractals.