Simulation of Wave Propagation in a Time-modulated Metamaterial

Mauro Morini

May 8, 2024

Contents

1 Introduction

The goal of this short project was to simulate the propagation of a plane wave in two dimensions through a wave guide containing time modulated meta-materials functioning as resonators. These resonators slow down the speed of propagation of the wave by time dependant functions ρ and κ. The idea was to place resonators in the wave guide, such that they obstruct most of the waves path with the exception of a slit. When the wave hits the resonators we expected to see an interference pattern behind the resonators, as well as the interference of the scattered wave onto the incident wave in front of the resonators.

1.1 The Problem

We considered a rectangular, two-dimensional domain $\Omega = (0, L_x) \times (0, L_y)$ and a time interval of interest $(0, T)$. The boundary $\partial\Omega$ of Ω is subdivided into the four sides of the rectangle:

$$\Gamma_1 = \{(x, y) \in \Omega \,|\, x = 0\},\ \Gamma_2 = \{(x, y) \in \Omega \,|\, y = 0\},\ \Gamma_3 = \{(x, y) \in \Omega \,|\, y = L_y\},\ \Gamma_4 = \{(x, y) \in \Omega \,|\, x = L_x\}$$

We were interested in a solution u of the wave equation with time and space dependent wave speed $c(x, t) = \sqrt{\kappa(x, t)/\rho(x, t)}$. u should satisfy homogeneous Neumann boundary conditions on Γ_i for $i = 2, 3$ and should be described by a non homogeneous, time dependent Neumann boundary condition on Γ_1. For Γ_4 we choose a first-order absorbing boundary condition to imitate an ongoing wave guide and avoid unwanted reflection. This yields the following PDE:

$$\begin{cases} \dfrac{\partial}{\partial t}\Big(\dfrac{1}{\kappa(x,t)}\dfrac{\partial}{\partial t}u(x,t)\Big) - \nabla\cdot\Big(\dfrac{1}{\rho(x,t)}\nabla u(x,t)\Big) = 0, & \text{in} \quad \Omega \times (0,T) \\ u = u_0, \quad \dfrac{\partial u}{\partial t} = v_0, & \text{in} \quad \Omega \times \{0\} \\ \dfrac{\partial u}{\partial n} = g(x,t), & \text{on} \quad \Gamma_1 \times (0,T) \\ \dfrac{\partial u}{\partial n} = 0, & \text{on} \quad \Gamma_i \times (0,T) \qquad \text{for} \quad i = 2,3 \\ \dfrac{\partial u}{\partial t} + c\dfrac{\partial u}{\partial n} = 0, & \text{on} \quad \Gamma_4 \times (0,T) \end{cases} \tag{1}$$

1.2 Methodology

To simulate the plane wave we first had to create a triangulation of the given wave guide using Matlab's `PDE-Toolbox`. To then approximate the exact solution we used linear finite elements in space and the Leap-Frog method in time as well as some methods to reduce the computational cost.

2 Mesh Creation

We created two types of meshes:
Firstly an equidistant triangulation of the unit square $\Omega_2 := (0, 1) \times (0, 1)$ to test our code. For this we used our Matlab function `generateMesh2dUnitSquare` from *Numerical Methods for PDE*.
Then to triangulate our final domain $\Omega = (0, L_x) \times (0, L_y)$ we wrote a function `gen_mesh` which outputs a geometry matrix *dl* containing the necessary information of the mesh using Matlab's function `decsg` and a coordinate matrix containing the defining vertices of the domain and it's sub-domains (in our case the resonators).
We then wrote a script `generate_rectangular_Mesh_with_resonators` which creates said geometry matrix *dl* and introduces the data into a pde model object using the `generateMesh` command of Matlab. Finally we were able to extract the point, edge and connectivity matrices from the resulting pde model using the `meshToPet` command.
We chose the following variables for the domain Ω: $L_x = 20$ and $L_y = 10$ and the resonators we placed as rectangles with the following vertices:

R_1: $x_1 = (12, 0), x_2 = (12.25, 0), x_3 = (12.25, 5.5), x_4 = (12, 5.5)$,
R_2: $x_5 = (12, 6), x_6 = (12.25, 6), x_7 = (12.25, 10), x_8 = (12, 10)$.

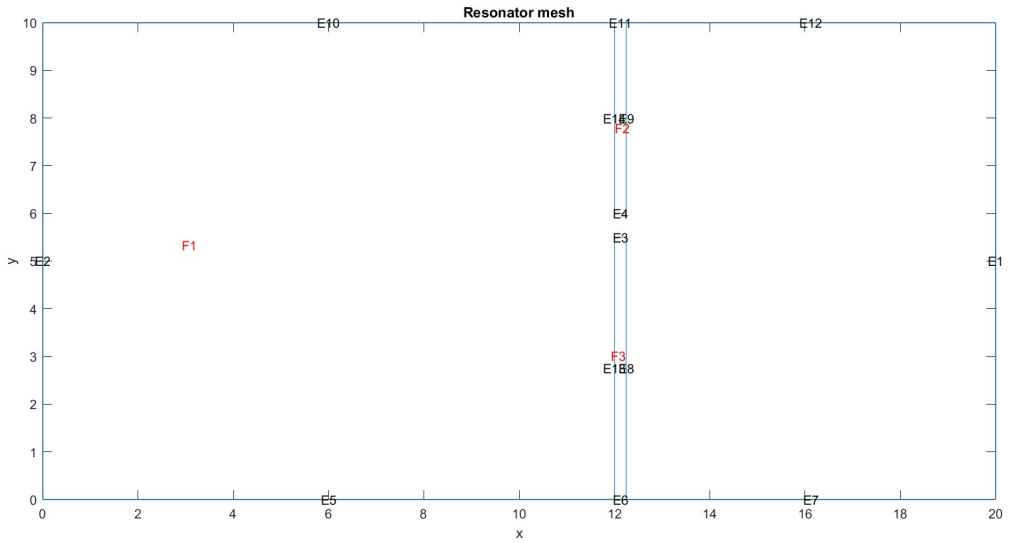

Figure 1: The domain Ω

In Figure (1) we can see the resonators denoted by $F2$ and $F3$ and the background medium by $F1$.

Since we expected the important part of our simulation to be happening inside and in the close vicinity of the resonators, we set the mesh size $h = 0.03$ there. On the other hand the in the background medium we expected simpler behaviour of the wave, which is why we chose a general maximal mesh size of $H = 0.5$ on all of Ω. The resulting triangulation can be see in Figure (2). For the final simulation we ended up decreasing the maximal mesh size $H = 0.1$ for smoother pictures.

3 Numerical Simulation

3.1 Time Independent Problem

First we needed to extend our already existing finite element code for the problem at hand and test our implementation. Firstly we were missing a non homogeneous Neumann boundary condition implementation, so we considered the following elliptical, time independent problem on the unit square $\Omega_2 = (0, 1) \times (0, 1)$:

$$\begin{cases} -\Delta u + u = f & \text{in} \quad \Omega_2 \\ \dfrac{\partial u}{\partial n} = g & \text{on} \quad \partial\Omega_2 \end{cases} \tag{2}$$

To solve this problem with finite elements, we first derived the weak formulation. For that we multiplied by a test function $v \in V := H^1(\Omega_2)$ and integrated over Ω_2:

$$\int_{\Omega_2} fv \,\mathrm{d}x = \int_{\Omega_2} -\Delta uv + uv \,\mathrm{d}x \overset{I.P.}{=} \int_{\Omega_2} \nabla u \cdot \nabla v + uv \,\mathrm{d}x - \int_{\partial\Omega_2} v \frac{\partial u}{\partial n} \,\mathrm{d}s = \int_{\Omega_2} \nabla u \cdot \nabla v + uv \,\mathrm{d}x - \int_{\partial\Omega_2} vg \,\mathrm{d}s$$

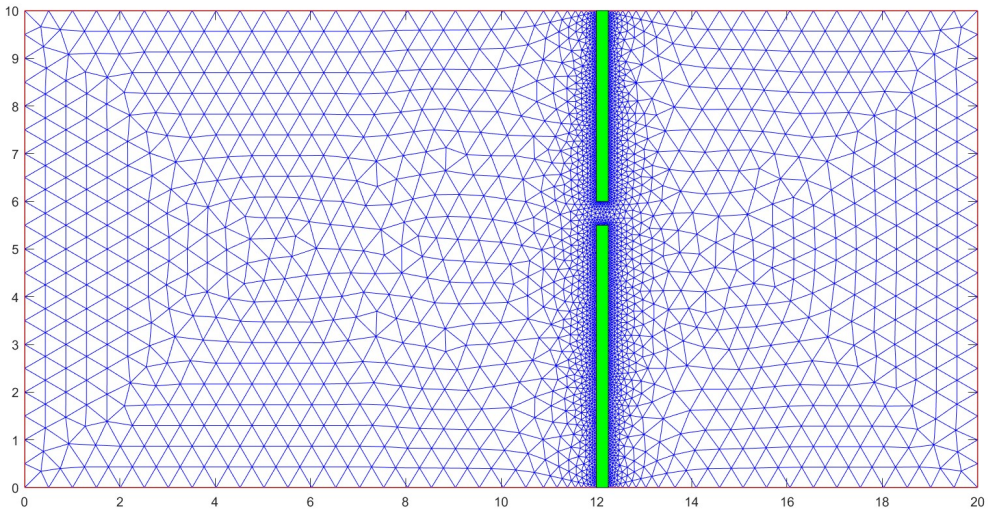

Figure 2: Triangulation of Ω with $H = 0.5$

This gave us the weak formulation: Find $u \in V$, such that

$$\begin{aligned} a(u,v) &= \ell(v) \qquad \forall v \in V \\ a(u,v) &:= \int_{\Omega_2} \nabla u \cdot \nabla v + uv \, \mathrm{d}x \\ \ell(v) &:= \int_{\Omega_2} fv \, \mathrm{d}x + \int_{\partial\Omega_2} vg \, ds \end{aligned} \tag{3}$$

Now after applying our standard Galerkin method to our finite element space $V_h = \{v \in C^0(\overline{\Omega}_2) | \, v|_K \in \mathbb{P}^1(K),\, K \in \mathcal{T}_h\}$ where $\mathcal{T}_h$ is a shape regular (in our case equidistant) triangulation of Ω_2, we got the following linear system:

$$(A + M)u = L + G$$

Here $A, M \in \mathbb{R}^{N \times N}$ are our standard stiffness, mass matrices respectively, to compute those in Matlab we used our old functions `stiffnessMatrix2D` and `massMatrix2D`. $L \in \mathbb{R}^N$ is our standard load vector computed by the function `loadVector2D` we implemented in *Numerical Methods for PDE's*. Finally $G \in \mathbb{R}^N$ is our new Neumann load vector containing the boundary conditions. To assemble this vector in Matlab one needs to iterate over the boundary edges of the mesh and compute one dimensional boundary integrals over each edge affected by the condition[1].

We chose the exact solution $u(x_1, x_2) = \sin(\pi x_1)\sin(\pi x_2)$. This leads to:

$$f = -\Delta u + u = (1 + \pi^2 + \pi^2)\sin(\pi x_1)\sin(\pi x_2)$$

To find g we recall that $\frac{\partial u}{\partial n} = \nabla u \cdot n$, so we get:

$$g(x_1, x_2) = \begin{cases} -\pi \cos(\pi x_1)\sin(\pi x_2), & x_1 = 0 \\ -\pi \cos(\pi x_2)\sin(\pi x_1), & x_2 = 0 \\ \pi \cos(\pi x_1)\sin(\pi x_2), & x_1 = 1 \\ \pi \cos(\pi x_2)\sin(\pi x_1), & x_2 = 1 \end{cases}$$

[1]For more information see: The Finite Element Method by Mats G. Larson and Frederik Bengzon; Chapter 4.6.2

We calculated the numerical solutions and the L^2-error for $h = 2^{-i}$, $i = 4, ..., 7$.
To compute the L^2-error we used the function `error2d` we implemented in the course last semester. As expected we observed the error to be decreasing like $\mathcal{O}(h^2)$.

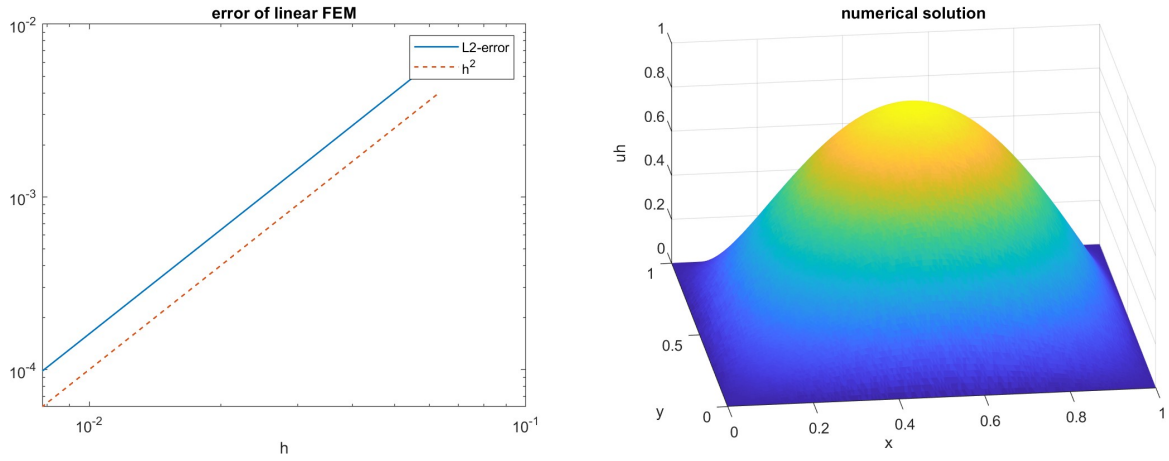

Figure 3: error and plot of $\sin(\pi x_1)\sin(\pi x_2)$

3.2 Variational Formulation of the Time Dependent Problem

After successfully testing our finite element code on a time independent problem, we returned to the time dependent problem (1). To derive the weak formulation we multiplied (1) by a test function $v \in V := H^1(\Omega \times (0,T))$, integrated over Ω and used integration by parts:

$$0 = \int_\Omega \frac{\partial}{\partial t}\Big(\frac{1}{\kappa}\frac{\partial u}{\partial t}\Big)v - \nabla\cdot\Big(\frac{1}{\rho}\nabla u\Big)v\,\mathrm{d}x \overset{I.P.}{=} \int_\Omega \frac{\partial}{\partial t}\Big(\frac{1}{\kappa}\frac{\partial u}{\partial t}\Big)v + \frac{1}{\rho}\nabla u\cdot\nabla v\,\mathrm{d}x - \int_{\partial\Omega}\frac{v}{\rho}\frac{\partial u}{\partial n}\mathrm{d}s$$

Then we inserted the boundary conditions into the boundary term:

$$\int_{\partial\Omega}\frac{v}{\rho}\frac{\partial u}{\partial n}\mathrm{d}s = \int_{\Gamma_1}\frac{v}{\rho}\underbrace{\frac{\partial u}{\partial n}}_{=\mathrm{g}}\mathrm{d}s + \underbrace{\int_{\Gamma_2\cup\Gamma_3}\frac{v}{\rho}\frac{\partial u}{\partial n}\mathrm{d}s}_{=0} + \int_{\Gamma_4}\frac{v}{\rho}\frac{\partial u}{\partial n}\mathrm{d}s = \int_{\Gamma_1}\frac{v}{\rho}g\,\mathrm{d}s - \int_{\Gamma_4}\frac{v}{\rho\,c}\frac{\partial u}{\partial t}\,\mathrm{d}s$$

So finally the weak formulation is: Find $u \in V$, such that:

$$\begin{aligned} a(u,v) &= \ell(v) \qquad \forall v \in V \\ a(u,v) &:= \int_\Omega \frac{\partial}{\partial t}\Big(\frac{1}{\kappa}\frac{\partial u}{\partial t}\Big)v + \frac{1}{\rho}\nabla u\cdot\nabla v\,\mathrm{d}x + \int_{\Gamma_4}\frac{v}{\rho\,c}\frac{\partial u}{\partial t}\,\mathrm{d}s \\ \ell(v) &:= \int_{\Gamma_1}\frac{v}{\rho}g\,\mathrm{d}s \end{aligned} \tag{4}$$

3.3 Galerkin Approximation

For simplicity we chose κ to be time-independent. Let V_h be our linear finite element space in 2D. Then for any fixed time $t \in (0,T)$ we sought a finite element solution of the form:

$$u_h(x,t) = \sum_{i=1}^{N} u_i(t)\varphi_i(x)$$

Where φ_i are our lagrangian basis functions and the coefficients u_i depend on the time t. After applying the standard Galerkin method we get the Galerkin approximation: For a fixed $t \in (0,T)$ find $u_h(t) \in V_h$ such that:

$$a(\sum_{j=1}^{N} u_j(t)\varphi_j, \varphi_i) = \ell(\varphi_i) \qquad i = 1, .., N$$

$$\Leftrightarrow \sum_{j=1}^{N} \frac{\partial^2}{\partial t^2} u_j(t) \int_\Omega \frac{1}{\kappa} \varphi_j \varphi_i \, \mathrm{d}x + \sum_{j=1}^{N} u_j(t) \int_\Omega \frac{1}{\rho} \nabla\varphi_j \cdot \nabla\varphi_i \, \mathrm{d}x + \sum_{j=1}^{N} \frac{\partial}{\partial t} u_j(t) \int_{\Gamma_4} \frac{1}{\rho\, c} \varphi_j \varphi_i \, \mathrm{d}s = \int_{\Gamma_1} \frac{1}{\rho} \varphi_i g \, \mathrm{d}s \qquad i = 1, .., N$$

By considering element wise contributions we derive the linear system:

$$M\ddot{u}(t) + A(t)u(t) + R(t)\dot{u}(t) = G(t) \tag{5}$$

M here is our normal mass matrix, but now it is dependent on κ. This dependency required not much adaptation in our code, since we are working with linear finite elements one can use the *midpoint quadrature rule* to approximate the integral of the product. Similarly A is our stiffness matrix which needed to be adapted to incorporate the scaling by $1/\rho$. G is the previously implemented Neumann load vector, which here takes g/ρ as an input. R is a newly implemented Neumann mass matrix. To assemble R one proceeds exactly as one would assembling a regular mass matrix, but in this case the integrals that are to be computed are not element wise, but rather boundary edge wise[2]. These edge wise integrals are line integrals in one dimension, hence the assembly of R is analogous to the 1D assembly of the mass matrix. The code for it can be found in `neumannMassMatrix2D`.

3.4 Time Discretization

To discretize in time we used Leap Frog for the second order derivative and a centred difference quotient for the first order derivative in time. Fix $\Delta t > 0$ and let $u^m = u(t_m) = u(m\Delta t)$ denote the finite element vector at a fixed time $t_m \in (0, T)$, then we got the following two step scheme in time from (5):

$$\begin{aligned} M\Big(\frac{u^{m+1} - 2u^m + u^{m-1}}{\Delta t^2}\Big) + A(t_m)u^m + R(t_m)\Big(\frac{u^{m+1} - u^{m-1}}{2\Delta t}\Big) &= G(t_m) \\ \Leftrightarrow \Big(\frac{1}{\Delta t^2} M + \frac{1}{2\Delta t} R(t_m)\Big)u^{m+1} = G(t_m) + \Big(\frac{2}{\Delta t^2} M - A(t_m)\Big)u^m - \Big(\frac{1}{\Delta t^2} M - \frac{1}{2\Delta t} R(t_m)\Big)u^{m-1} \end{aligned} \tag{6}$$

From the initial conditions $u^0 = u_0$ is given, so we needed to compute an approximation of u^1 using the second initial condition $\partial_t u^0 = v_0$. To do so we approximate $\partial_t u^0$ by a centred difference quotient implementing a *ghost point* u^{-1}:

$$v_0 = \frac{\partial u}{\partial t} \approx \frac{u^1 - u^{-1}}{2\Delta t} \qquad \Leftrightarrow \qquad u^-1 \approx u^1 - 2\Delta t v_0$$

Plugging this into our fully discrete scheme (6) for $m = 0$ gives us a formula for u^1. From there on we continue iteratively.

3.5 Time Discretization Test

Next we wanted to test our fully discrete scheme. To do so we considered the domain Ω_2 the unit square without any resonators, in particular we set $c \equiv \kappa \equiv \rho \equiv 1$ on all of Ω_2. We chose the exact solution to be $u(x_1, x_2, t) = \cos(\pi(x_1 - t))/\pi$, this lead to:

$$\begin{cases} u_0(x_1, x_2) = u(x_1, x_2, 0) = \dfrac{\cos(\pi x_1)}{\pi} \\ v_0(x_1, x_2) = \dfrac{\partial u}{\partial t}(x_1, x_2, 0) = \sin(\pi x_1) \\ g(x_1, x_2, t) = \nabla u \cdot (-1, 0)^T = \sin(\pi(x_1 - t)) \end{cases}$$

We set $T = 1$, $h = 2^{-i}$, for $i = 3, .., 7$ and $\Delta t = rh$ with $r = 0.5$, calculated the L^2-error for each mesh size h and plotted the numerical solution on the finest mesh at the times $t = 0.25, 0.5, 0.75, 1$ in Figure (4)

[2]For more information see: The Finite Element Method by Mats G. Larson and Frederik Bengzon; Chapter 4.6.2

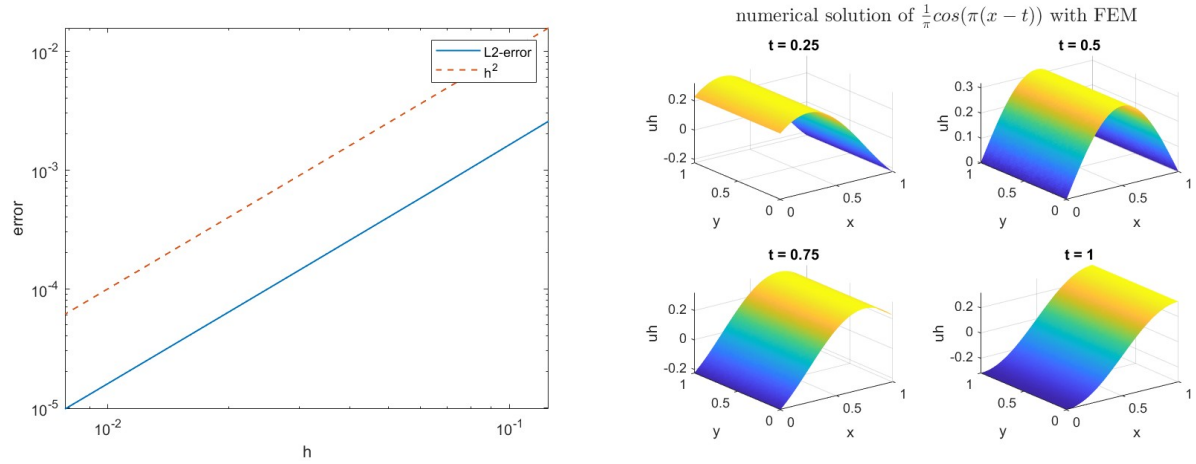

Figure 4: error and plot of $\cos(\pi(x_1 - t))/\pi$

3.6 Simulation with Resonators

After successfully testing our implementation we were then able to apply our code to the mesh we created in section 2. We define:

$$\kappa(x,t) = \begin{cases} 1, & x \notin R_i \\ 0.2, & x \in R_i \end{cases}, \qquad \rho(x,t) = \begin{cases} 1, & x \notin R_i \\ \rho_r(t), & x \in R_i \end{cases}$$

for $i = 1, 2$ with

$$\rho_r(t) = \frac{0.1}{1 + 0.2\cos(t\pi/2)}$$

For the non homogeneous Neumann boundary condition we used:

$$g(x_1, x_2, t) = \sin(\omega(x_1 - t)) \qquad \forall\, x_2 \in (0, L_y)$$

Where $\omega = 2\pi$. And we used homogeneous initial conditions $u_0 \equiv v_0 \equiv 0$. We calculated the numerical solution for different times in $(0, 30)$. As expected the wave propagates very regularly until it hits the resonators, because of the slit which allows a portion of the wave to propagate considerably faster than the rest we can observe the interference behind the resonator. On the other hand the resonators slow down the majority of the wave, which leads to the creation of a scattering wave in the opposite direction of the incident wave. These interference patterns can be observed in Figure (5)

3.7 Runtime Improvements

In the time dependent simulation we did not only have to solve a very big system of equations every time step but also assemble the time dependent stiffness matrix for each time. This turned out to be impossibly costly so we did two modifications which reduced the computational load drastically:

1. Isolating Time Dependence

Firstly we noted that although R is theoretically time dependent, in reality it is the matrix describing the absorbing boundary condition at the boundary Γ_4 and therefore only influenced by values at Γ_4, i.e.

$$\begin{aligned} \kappa(x,t) = \rho(x,t) = c(x,t) = 1 \qquad & \forall x \in \Gamma_4, t \in (0,T) \\ \Rightarrow R(t_m) = R(0) = R \qquad & \forall m \geq 0. \end{aligned}$$

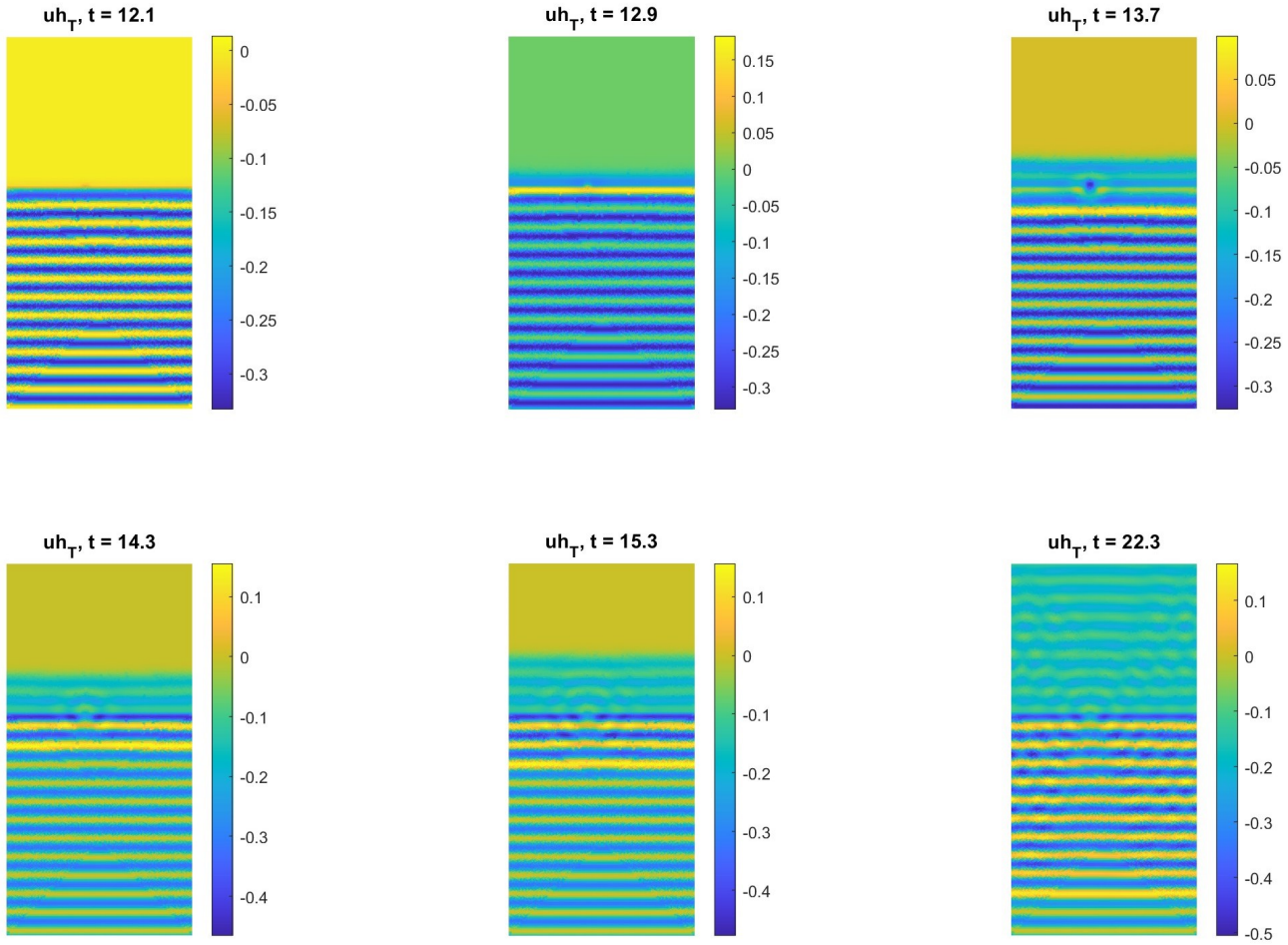

Figure 5: Interference pattern on resonator mesh

Secondly we noted that both κ and ρ are piecewise constant in space. So we got:

$$\begin{aligned}\int_\Omega \frac{1}{\rho}\nabla\varphi_j \cdot \nabla\varphi_i \,\mathrm{d}x &= \int_{R_1\cup R_2} \frac{1}{\rho}\nabla\varphi_j \cdot \nabla\varphi_i \,\mathrm{d}x + \int_{\Omega\setminus R_1\cup R_2} \frac{1}{\rho}\nabla\varphi_j \cdot \nabla\varphi_i \,\mathrm{d}x \\ &= \frac{1}{\rho}\int_{R_1\cup R_2} \nabla\varphi_j \cdot \nabla\varphi_i \,\mathrm{d}x + \int_{\Omega\setminus R_1\cup R_2} \nabla\varphi_j \cdot \nabla\varphi_i \,\mathrm{d}x\end{aligned}$$

This meant that we could separately assemble a background material stiffness matrix $A_{\text{backg}} \in \mathbb{R}^{N\times N}$ and a resonator stiffness matrix $A_{\text{res}} \in \mathbb{R}^{N\times N}$ which only consider the elements outside or inside the resonators respectively. Finally it holds that:

$$A(t) = A_{\text{backg}} + \frac{1}{\rho(x,t)}A_{\text{res}} \qquad \forall t \in (0,T)$$

where $x \in R_1 \cup R_2$ arbitrary.
Therefore we only had to assemble the stiffness matrix once at the beginning and only had to multiply the factor $1/\rho$ in each time step

2. Mass Lumping:

To simplify the system (6) (which we need to solve in each time step) we used the technique of *Mass Lumping*. Since both M and R are forms of mass matrices we could *lump* the element wise contributions onto the diagonal entries of the respective matrix, to do so we created diagonal matrices $M_{\text{lump}}, R_{\text{lump}}$ where each diagonal entry corresponds to the sum of the row elements in M, R. So we defined:

$$M_{\text{lump}} := \operatorname{diag}\Big(\sum_{j=1}^{N} M_{1,j}, .., \sum_{j=1}^{N} M_{N,j}\Big), \qquad R_{\text{lump}} := \operatorname{diag}\Big(\sum_{j=1}^{N} R_{1,j}, .., \sum_{j=1}^{N} R_{N,j}\Big)$$

and exchanged $M_{\text{lump}}, R_{\text{lump}} \leftrightarrow M, R$.
This makes inverting the matrix $(M_{\text{lump}}/\Delta t^2 + R_{\text{lump}}/(2\Delta t))$, and hence solving the system a lot cheaper.

4 References

1. Prof. Dr. Marcus J. Grote. *Numerical Methods for Partial Differential Equations.* Fall Semester, 2023
2. Prof. Dr. Marcus J. Grote. *Numerical Methods for Wave Propagation.* Spring Semester, 2024
3. Larson, G., Mats; Bengzon, Fredrik. *The Finite Element Method: Theory, Implementation and Applications.* Springer.

5 Matlab Code

5.1 Scripts

5.1.1 generate_rectangular_Mesh_with_resonators

```
% author: Mauro Morini
% last modified 14.04.24
clc;clear;close all;

R1coord = [12, 0; 12.25, 0; 12.25, 5.5; 12, 5.5];
R2coord = [12, 6; 12.25, 6; 12.25, 10; 12, 10];
RTotCoord = [R1coord; NaN, NaN; R2coord];
pgon = polyshape(RTotCoord(:,1),RTotCoord(:,2));
dl = gen_mesh(1);
model = createpde;
geometryFromEdges(model, dl);
generateMesh(model, GeometricOrder='linear', ...
    Hface={[2,3],0.03}, Hmax=0.5, Hgrad=1.3);

% get coordinate, edge and connectivity matrix
[p,e,t] = meshToPet(model.Mesh);
% p = p';
% t = t';
% t = t(:, 1:3);
% e = e';
% e = e(:, 1:2);

figure(2)
pdeplot(model)
hold on
% h = plot(pgon);
% h.FaceColor = 'green';
fill(R1coord(:,1),R1coord(:,2),'g')
fill(R2coord(:,1),R2coord(:,2),'g')
axis equal;axis tight;
hold off
```

5.1.2 NMWP_Project_E2

```
% author: Mauro Morini
% last modified 14.04.24
clc;clear;close all;

% Initializations
H = 2.^(-(3:6))/2;
numIter = length(H);
L2err = zeros(1, numIter);

% functions
syms x y
u = sin(pi*x)*sin(pi*y);
f = -laplacian(u) + u;
u = matlabFunction(u);
f = matlabFunction(f);
```

```
% g = @(x,y) (x == 0 || y == 0).*(-pi).*sin(x*pi).*sin(y*pi) + (x == 1 || y == 1).*pi.*sin
g = @(x,y)  (x == 1).*(pi).*cos(x*pi).*sin(y*pi) + ...
            (y == 1).*(pi).*sin(x*pi).*cos(y*pi) + ...
            (x == 0).*(-pi).*cos(x*pi).*sin(y*pi) + ...
            (y == 0).*(-pi).*sin(x*pi).*cos(y*pi);

for i = 1:numIter
    [p, t, e] = generateMesh2dUnitSquare(H(i));
    uh = solve_elliptic_BVP_2d_FEM_Neumann(p,t,e,f,g);
    u_exact = u(p(:,1), p(:, 2));
    L2err(i) = error2d(p, t, u, uh);
end

%% plot
figure()
plog = loglog(H,L2err,H,H.^2,'--');
legend("L2-error", "h^2")
title("error of linear FEM")
xlabel("h")
plog(1).LineWidth = 1;
plog(2).LineWidth = 1;

% figure()
% trisurf(t,p(:,1),p(:,2),u_exact, 'EdgeColor', 'none')
figure()
trisurf(t,p(:,1),p(:,2),uh, 'EdgeColor', 'none')
xlabel("x")
ylabel("y")
zlabel("uh")
title("numerical solution")

% pdeplot(p', t')
```

5.1.3 NMWP_Project_E3

```
% author: Mauro Morini
% last modified: 24.04.24
clc;clear;close all;

% Initializations
H = 2.^(-(2:6))/2;
r = 0.5;
T = [0.25, 0.5, 0.75, 1];
L2err = zeros(1, length(H));

% functions
u0 = @(x,y) 1/pi*cos(pi*x);
u1 = @(x,y) sin(pi*x);
g = @(x,y,t) sin(pi*(x - t));
uexact = @(x,y,t) 1/pi*cos(pi*(x-t));

for l = 1:length(H)
    h = H(l);
    dt = r*h;
    % mesh
```

```
    [p, t, e] = generateMesh2dUnitSquare(h);

    % calculate numerical solution
    [uh_T2, tvec] = waveEqLF2D(u0, u1, g, T, dt, p, e, t);

    % exact solution at tvec(end) ~= T(end)
    Uexact = uexact(p(:,1),p(:,2),tvec(end));

    % error
    L2err(l) = error2d(p,t,@(x,y) uexact(x,y,tvec(end)),uh_T2(:,end));
end

%% plots

% plot error
figure(1)
plog = loglog(H, L2err, H, H.^2, '--');
legend("L2-error", "h^2")
plog(1).LineWidth = 1;
plog(2).LineWidth = 1;
xlabel("h")
ylabel("error")


% plot exact and numerical solution in finest mesh
figure(2);
tld = tiledlayout('flow');
title(tld, "numerical solution of " + ...
    "$\frac{1}{\pi} cos(\pi(x - t))$ with FEM", 'Interpreter', 'latex')
for i = 1:length(T)
    nexttile
    trisurf(t,p(:,1),p(:,2),uh_T2(:,i), 'EdgeColor', 'none')
    %view(2)
    title("t = " + T(i))
    xlabel("x")
    ylabel("y")
    zlabel("uh")
end

figure(3);
tld = tiledlayout('flow');
title(tld, "exact solution of " + ...
    "$\frac{1}{\pi} cos(\pi(x - t))$ with FEM", 'Interpreter', 'latex')
for i = 1:length(T)
    nexttile
    trisurf(t,p(:,1),p(:,2),uexact(p(:,1),p(:,2),T(i)), 'EdgeColor', 'none')
    %view(2)
    title("t = " + T(i))
    xlabel("x")
    ylabel("y")
    zlabel("uh")
end
```

5.1.4 NMWP_Project_E4

```
% author: Mauro Morini
% last modified: 03.05.24
clc;clear;close all;

% Initialization
H = 0.08;
h = 0.01;
r = 0.4;
dt = h^2*r;
dt = 0.005;
dt = 0.001;
omega = 2*pi;
T = 0.5:0.2:30;

% create mesh
R1coord = [12, 0; 12.25, 0; 12.25, 5.5; 12, 5.5];
R2coord = [12, 6; 12.25, 6; 12.25, 10; 12, 10];
RTotCoord = [R1coord; NaN, NaN; R2coord];
dl = gen_mesh(0);
model = createpde;
geometryFromEdges(model, dl);
generateMesh(model, GeometricOrder='linear', Hface={[2,3],h}, ...
    Hmax=H, Hgrad=1.3);
[p,e,t] = meshToPet(model.Mesh);
p = p';
t = t';
t = t(:, 1:3);
e = e';
e = e(:, 1:2);

% functions
kappa = @(x,y) 1*~inpolygon(x,y,RTotCoord(:, 1), RTotCoord(:, 2)) + ...
        0.2*inpolygon(x,y,RTotCoord(:, 1), RTotCoord(:, 2));
roh_r = @(t) 0.1/(1+0.2*cos(pi/2*t));
roh = @(x,y,t) 1*~inpolygon(x,y,RTotCoord(:, 1), RTotCoord(:, 2)) + ...
        roh_r(t)*inpolygon(x,y,RTotCoord(:, 1), RTotCoord(:, 2));
g = @(x,y,t) sin(omega*(x - t));
u0 = @(x,y) x*0;
u1 = u0;

%%
% uh_T = zeros(size(p, 1), length(T));
% for i = 1:length(T)
%     uh_T(:, i) = waveEqLF2D(u0, u1, g, T(i), dt, p, e, t, kappa, roh);
%     disp("Computation for t = " + T(i) + " completed")
% end
uh_T = waveEqLF2D(u0, u1, g, T, dt, p, e, t, kappa, roh,RTotCoord);

%% plot
% for i = length(T):-1:1
%     figure;
%     tld = tiledlayout('flow');
%     nexttile
%     trisurf(t,p(:,1),p(:,2),uh_T(:,i), 'EdgeColor', 'none')
```

```
%       title("uh_T, t = " + T(i))
%       view(2)
% end
figure(1);
tld = tiledlayout('flow');
Tloc = [59, 63, 67, 70, 75, 110];
Tloc = 90;
for i = Tloc
    nexttile
    trisurf(t,p(:,1),p(:,2),uh_T(:,i), 'EdgeColor', 'none')
    xlabel("x")
    ylabel("y")
    zlabel("uh")
    title("uh_T, t = " + T(i))
    colorbar;
    axis equal; axis off; axis tight ; %colormap ('jet');
    %view(2)
    %view(270,90)
end


% save("NMWP_Project_E4_workspace_backup_v1.06.mat")
% save("uh_T_E4_v1.06.mat","uh_T")
% save("NMWP_Project_resonator_mesh_data_v1.06.mat")
```

5.2 Functions

5.2.1 stiffnessMatrix2D

```
% author: Mauro Morini
% last modified 24.04.24
function A = stiffnessMatrix2D(p, t, c)
% calculates global stiffness matrix for linear FEM in 2D int_omega
% a(x)*phi_i*phi_j dx
% Inputs :
% p : nPx2 coordinate matrix with points in rows
% t : nEx3 connectivity matrix with elements in rows
% c : if exists function handle of bilinear form
% Outputs :
% A : global stiffness matrix nPxnP

if ~exist('c','var')
    c = @(x,y) 1;
end

nP = size(p, 1);
nE = size(t, 1);
A = sparse(nP, nP);

% iterate over elements
for k = 1:nE
    % Element and points
    K = t(k, :);
    p0 = p(K(1), :).';
    p1 = p(K(2), :).';
```

```
    p2 = p(K(3), :).';

    cent = mean([p0, p1, p2], 2);        % center point

    % calculate J_K elementwise jacobian of the transformation
    Jk = [p1-p0, p2-p0];

    K_area = abs(det(Jk))/2;
    DN = [-1, -1; 1, 0; 0, 1];

    % elementwise local stiffness matrix
    A_loc = c(cent(1),cent(2))*K_area*DN*inv(Jk'*Jk)*DN';

    % Assembling
    A(t(k, :), t(k, :)) = A(t(k, :), t(k, :)) + A_loc;
end
end
```

5.2.2 massMatrix2D

```
% author: Mauro Morini
% last modified: 24.04.24
function M = massMatrix2D(p, t, c)
% calculates global mass matrix for linear
% FE in 2D M(i,j) = int_Omega c(x)*phi_i*phi_j
% Inputs :
% p : nPx2 coordinate matrix with points in rows
% t : nEx3 connectivity matrix with elements in rows
% c : if exists function handle of bilinear form
%
% Outputs :
% M : global stiffness matrix nPxnP

if ~exist('c','var')
    c = @(x,y) 1;
end

nP = size(p, 1);
nE = size(t, 1);
M = sparse(nP, nP);

% iterate over elements
for k = 1:nE
    % Element and points
    K = t(k, :);
    p0 = p(K(1), :).';
    p1 = p(K(2), :).';
    p2 = p(K(3), :).';

    cent = mean([p0, p1, p2], 2);        % center point

    % calculate J_K elementwise jacobian of the transformation
    Jk = [p1-p0, p2-p0];
    K_area = abs(det(Jk))/2;
```

```
    % elementwise local stiffness matrix
    Mloc = [2, 1, 1; 1, 2, 1; 1, 1, 2];
    Mloc = c(cent(1),cent(2))*K_area*(1/12)*Mloc;

    % Assembling
    M(t(k, :), t(k, :)) = M(t(k, :), t(k, :)) + Mloc;
end
end
```

5.2.3 loadVector2D

```
% author: Mauro Morini
% last modified 09.11.23
function L = loadVector2D(p, t, f)
% calculates global load vector for linear FEM in 2D
% Inputs :
% p : nPx2 coordinate matrix with points in rows
% t : nEx3 connectivity matrix with elements in rows
% f : function handle
% Outputs :
% L : global stiffness matrix nPx1

nP = size(p, 1);
nE = size(t, 1);
L = zeros(nP, 1);

% iterate over elements
for k = 1:nE
    % Element and points
    K = t(k, :);
    p0 = p(K(1), :).';
    p1 = p(K(2), :).';
    p2 = p(K(3), :).';

    % calculate J_K elementwise jacobian of the transformation
    Jk = [p1-p0, p2-p0];
    KArea = abs(det(Jk))/2;

    % approximate f by value in the center of the element K
    pCent = mean([p0, p1, p2], 2);
    fk = f(pCent(1), pCent(2));
    % fk = mean([f(p0(1), p0(2)), f(p1(1), p1(2)), f(p2(1), p2(2))], 2);

    % Assemble
    L(t(k,:)) = L(t(k,:)) + fk*KArea/3;
end

end
```

5.2.4 neumannMassMatrix2D

```
% author: Mauro Morini (adapted from "The Finite Element Theory" by
% Larson/Bengzon chapter 4.6)
% last modified 23.04.24
function R = neumannMassMatrix2D(p, e, c)
```

```
% assembles mass matrix for neumann b.c. on the edges int_gamma
% c*phi_i*phi_j ds
%
% Inputs:
% p : (nP,2) point matrix with x values in the first and y values in the
%       second column
% e : (nE,2) edge matrix containing two connecting edge point indices in
%       each row, of the boundary edges of the domain
% c : function handle from weak formulation default is 1
%
% Outputs:
% R :  (nP,nP) Neumann mass matrix with R_ij = int_{e_ij} c phi_i phi_j ds

if ~exist('c','var')
    c = @(x,y) 1;
end

% Initializations
nP = size(p, 1);
nE = size(e, 1);
R = sparse(nP,nP);

for i = 1:nE
    E = e(i,:);
    x1 = p(E(1),:);                                  % edge point 1
    x2 = p(E(2),:);                                  % edge point 2
    e_h = norm(x1 - x2, 2);                          % edge length
    cent = mean([x1', x2'], 2);

    % local element contributions, same as for mass matrix but here only
    % 2x2 because the integral is over e
    Rloc = e_h/6*[2, 1; 1 2]*c(cent(1), cent(2));
    R(E, E) = R(E, E) + Rloc;
end
end
```

5.2.5 neumannLoadVector2D

```
% author: Mauro Morini (adapted from "The Finite Element Theory" by
% Larson/Bengzon chapter 4.6)
% last modified 16.04.24
function L = neumannLoadVector2D(p, e, f)
% assembles load vector for neumann b.c. : du/dn = f on boundary Gamma given by
% the edge matrix e for 2d linear FE. In the weak formulation, the above
% b.c. has the following impact on the LHS: int_Gamma v*f ds for a
% testfunction v.
%
% Inputs:
% p : (nP,2) point matrix with x values in the first and y values in the
%       second column
% e : (nE,2) edge matrix containing two connecting edge point indices in
%       each row, of the boundary edges of the domain
% f : function handle of neumann b.c.
%
% Outputs:
```

```
% L : (nP,1) neumann load vector containing the pointwise impact of the
%       b.c. in each entry (zero for internal points)

% Initializations
nP = size(p, 1);
nE = size(e, 1);
L = zeros(nP,1);

for i = 1:nE
    E = e(i,:);
    x1 = p(E(1),:);                                  % edge point 1
    x2 = p(E(2),:);                                  % edge point 2
    e_h = norm(x1 - x2, 2);                          % edge length
    cent = mean([x1', x2'], 2);

    % local element contributions
    Lloc = e_h/2*f(cent(1),cent(2))*[1;1];
    L(E) = L(E) + Lloc;
end
end
```

5.2.6 waveEqLF2D

```
% author: Mauro Morini
% last modified: 26.04.24
function [uh_T, tvec] = waveEqLF2D(u0, u1, g, T, dt, p, e, t, kappa, roh, RTotCoord)
% solves the wave equation in 2D with Leap-Frog time discretization and
% mass lumping and FEM spacial discretization and returns u(x, T) at end
% time T with 4 boundary sections on which  2 have homogeneous neumann
% b.c. one has non-homogeneous neumann condition and one has an absorbing
% b.c. of first order (du/dt = -cdu/dn) on a rectangular mesh
% PDE : (1/kappa d^2/dt^2 - grad_x 1/roh grad_x)u(x,t) = 0
%
% Inputs :
% u0 : function handle for dirichlet initial condition u(x,0) = u0
% u1 : function handle for neumann initial condition du/dt (x,0) = u1
% g : function handle for neumann-b.c.
% T : scalar final time T, can be given as a row vector with times which should
%       be recorded (has to be sorted and actual recorded time t_actual
%       will be different from requested time by an error of dt)
% dt : scalar time step size
% p : nPx2 coordinate matrix with points in rows
% t : nEx3 connectivity matrix with elements in rows
% e : (nE,2) edge matrix containing two connecting edge point indices in
%       each row, of the boundary edges of the domain
% kappa : function handle kappa(x,y) for time dependant part
% roh : function handle roh(x,y,t) for grad part
%       RTotCoord: (nRp, 2) matrix with coordinates of the vertices
%       which contain the resonators, each resonator
%       is seperated by a [NaN,NaN] row (assumes resonators to be polygonal)
%
% Outputs :
% uh_T : (nP,1) numerical solution at time T, if T is a vector then uh_T
%           has size (nP, length(T))
```

```
% Initializations
hasResonators = false;
RA_timedep = true;
TisVec = false;
Lx = max(p(:, 1));                  % x length of wave guide

% case if T is a vector
if size(T, 2) > 1
    TisVec = true;
    TOld = T;
    iterT = 1;
    T = T(end);
    uh_T = zeros(size(p, 1), length(TOld));

    if TOld(1) < 2*dt
        error("smallest time in vector T is too small: Tmin = " + Tsorted(1))
    end
end

tvec = 0:dt:T;
N = length(tvec) - 1;

if ~exist('kappa','var')
    kappa = @(x,y) ones(size(x));
end
if ~exist('roh','var')
    roh = @(x,y,t) ones(size(x));
    RA_timedep = false;
end
c = @(x,y,t) sqrt(kappa(x,y)./roh(x,y,t));

if exist('RTotCoord','var')
    hasResonators = true;
end

% find boundary sections
pe1x = p(e(:, 1), 1);
pe1y = p(e(:, 1), 2);
pe2x = p(e(:, 2), 1);
pe2y = p(e(:, 2), 2);

% gamma1 x = 0
gamma1Idx = logical((pe1x == 0).*(pe2x == 0));
gamma1 = e(gamma1Idx, :);

% gamma4 x = Lx
gamma2Idx = logical((pe1x == Lx).*(pe2x == Lx));
gamma2 = e(gamma2Idx, :);

% find elements t(i,:) which are in the resonator
if hasResonators
    pt1 = p(t(:,1),:);
    pt2 = p(t(:,2),:);
    pt3 = p(t(:,3),:);
```

```
    resIdx = logical( ...
        inpolygon(pt1(:,1),pt1(:,2),RTotCoord(:, 1), RTotCoord(:, 2)).*...
        inpolygon(pt2(:,1),pt2(:,2),RTotCoord(:, 1), RTotCoord(:, 2)).*...
        inpolygon(pt3(:,1),pt3(:,2),RTotCoord(:, 1), RTotCoord(:, 2)));
    xRes = RTotCoord(1,1);
    yRes = RTotCoord(1,2);
else
    resIdx = false(size(t,1),1);
    xRes = 0;
    yRes = 0;
end
% Assemble stiffness matrix inside resonator and outside
Ares = stiffnessMatrix2D(p, t(resIdx,:));
Abackg = stiffnessMatrix2D(p, t(~resIdx,:));
A = Ares/roh(xRes,yRes,0) + Abackg;

%% test
% Astand = stiffnessMatrix2D(p, t, @(x,y) 1/roh(x,y,0));
% norm(full(Astand - A),'fro')

% Assemble and lump remaining matrices
M = massMatrix2D(p, t, @(x,y) 1/kappa(x,y));
R = neumannMassMatrix2D(p, gamma2,@(x,y) ((c(x,y,0).*roh(x,y,0)).^(-1)));
G = neumannLoadVector2D(p, gamma1, @(x,y) g(x,y,0)./roh(x,y,0));
MLump = diag(sum(M, 2));
RLump = diag(sum(R, 2));
M = MLump;
R = RLump;

% initial conditions
uhPrev = u0(p(:, 1), p(:, 2));
uhNow = (2/dt^2*M)\(G - (A - 2/dt^2*M)*uhPrev + (1/dt^2*M - 1/(2*dt)*R)*2*dt*u1(p(:, 1), p

for i = 1:N

    % construct system
    if RA_timedep
        % not used since roh is just 1 on the boundary
        % R = neumannMassMatrix2D(p, gamma2,@(x,y)((c(x,y,dt*i).*roh(x,y,dt*i)).^(-1)));
        % A = stiffnessMatrix2D(p, t, @(x,y) 1/roh(x,y,dt*i));
        A = Ares/roh(xRes,yRes,dt*i) + Abackg;
    end
    G = neumannLoadVector2D(p, gamma1, @(x,y) g(x,y,dt*i)./roh(x,y,dt*i));
    LHS = (1/dt^2*M + 1/(2*dt)*R);
    RHS = G - (A - 2/dt^2*M)*uhNow - (1/dt^2*M - 1/(2*dt)*R)*uhPrev;

    % solve system
    uhNext = LHS\RHS;
    uhPrev = uhNow;

    if TisVec && abs(dt*i - TOld(iterT)) < (dt - eps)
        uh_T(:, iterT) = uhNow;
        iterT = iterT + 1;
        disp("Computation for t = " + dt*i + " completed")
```

```
    end
    uhNow = uhNext;
end

if ~TisVec
    uh_T = uhNext;
end

if nargout > 1
    tvec = 0:dt:T;
end

% last step with smaller stepsize to arrive at T !!!!!NOT GOOD!!!!!
% if N*dt < T
%     dt2 = T - N*dt;
%
%     % construct system
%     if RA_timedep
%         % R = neumannMassMatrix2D(p, gamma2,@(x,y) ((c(x,y,T).*roh(x,y,T)).^(-1)));
%         A = stiffnessMatrix2D(p, t, @(x,y) 1/roh(x,y,T));
%     end
%     G = neumannLoadVector2D(p, gamma1, @(x,y) g(x,y,T)./roh(x,y,T));
%     LHS = (1/dt2^2*M + 1/(2*dt2)*R);
%     RHS = G - (A - 2/dt2^2*M)*uhNow - (1/dt2^2*M - 1/(2*dt2)*R)*uhPrev;
%
%     % solve system
%     uhNext = LHS\RHS;
%     if ~TisVec
%         uh_T = uhNext;
%     else
%         uh_T(:,iterT) = uhNext;
%         if iterT ~= length(TOld)
%             error("iterT is not the end time, something went wrong: iter: " + iter + " ..
%         end
%     end
%
% end
end
```

5.2.7 solve_elliptic_BVP_2d_FEM_Neumann

```
% author: Mauro Morini
% last modified 14.04.24
function uh = solve_elliptic_BVP_2d_FEM_Neumann(p, t, e, f, g)
% solves the problem -laplace(u) + u = f on the unit square in 2D with
% neumann b.c du/dn = g using linear FEM
%
% Inputs :
% p : nPx2 coordinate matrix with points in rows
% t : nEx3 connectivity matrix with elements in rows
% f : function handle RHS
% g : function handle, g:omega->R, for boundary condition
% e : connectivity matrix for the free edges (edges on the boundary)
%
% Outputs :
```

```
% uh : nPx1 coefficient vector of the FEM solution

% Initializations
nP = size(p, 1);

% assemble matrices
A = stiffnessMatrix2D(p,t);
M = massMatrix2D(p,t);
L = loadVector2D(p,t,f);
G = neumannLoadVector2D(p,e,g);

%% find interior points of triangularization
% idxExt = reshape(e, [], 1);
% idxExt = ismember(1:nP, idxExt);
% idxInt =~ idxExt;
%
%% incorporate b.c.
% G(idxInt) = 0;

% solve system
uh = (A + M)\(L + G);
end
```

5.2.8 error2d

```
% author: Mauro Morini
% last modified: 18.11.23
function L2err = error2d(p, t, u, uh)
% calculates the L^2 error of a linear FE soltion in 2D
%
% Inputs :
% p : nPx2 coordinate matrix with points in rows
% t : nEx3 connectivity matrix with elements in rows
% u : function handle @(x,y) of exact solution u
% uh : nPx1 column vector of FE solution
%
% Output :
% L2err : scalar value of L^2 error (||u-uh||_{L^2(omega)})

% Initalizations
[nE, r] = size(t);
nP = size(p, 1);
L2err = 0;
KhatArea = 1/2;

% weights of QF
w = (1/3)*[1, 1, 1];

% nodes of QF 2xr
xi = [0, 0; 1, 0; 0, 1]';

% shape functions
N = {@(xi) 1 - [1,1]*xi, @(xi) [1,0]*xi, @(xi) [0,1]*xi};

% iterate over elements
```

```
for i = 1:nE

    % Element and points
    K = t(i, :);
    p0 = p(K(1), :).';
    p1 = p(K(2), :).';
    p2 = p(K(3), :).';

    % calculate J_K elementwise jacobian of the transformation
    Jk = [p1-p0, p2-p0];

    % element map
    Fk = @(xi) p0 + Jk*xi;

    KArea = abs(det(Jk))/2;

    % calculate Quadrature
    Qval = zeros(1, r);
    for j = 1:r
        uhVal = 0;
        for q = 1:r
            uhVal = uhVal + uh(t(i, q))*N{q}(xi(:, j));
        end
        x = Fk(xi(:, j));
        Qval(j) = abs(u(x(1), x(2)) - uhVal)^2;
    end
    L2err = L2err + (KhatArea*KArea)*Qval*w';
end
L2err = sqrt(L2err);
end
```

5.2.9 gen_mesh

```
% author: Mauro Morini
% last modified 14.04.24
function geom = gen_mesh(flag)
%
%   Inputs :
%   flag : either 0 or 1 deciding if figure should be plotted or not
%
%   Outputs :
%   geom : decomposed geometriy matrix for reference see decsg
%   documentation

% meshdefining points x-values, y-values respectively
x = [-1,1,1,0,0,-1];
y = [-1,-1,1,1,0,0];
% components for decsg (can easily be acquired using the modeler app/
% pdepoly(x,y))
% gd = [2,6,-1,1,1,0,0,-1,-1,-1,1,1,0,0]';
% ns = [80;49];
% sf = 'P1';
x1 = [0, 20, 20, 0];
y1 = [0, 0, 10, 10];
r1x = [12, 12.25, 12.25, 12];
```

```
r1y = [0, 0, 5.5, 5.5];
r2x = [12, 12.25, 12.25, 12];
r2y = [6, 6, 10, 10];
gd =  [3, 4, x1, y1;
       3, 4, r1x, r1y;
       3, 4, r2x, r2y]';
ns = ["F1", "F2", "F3"];
sf = 'F1+F2+F3';
geom = decsg(gd,sf,ns);

if flag ~= 0 && flag ~= 1
    error('flag parameter should be either 0 or 1 \n%s', "flag: " + flag)
elseif flag == 0
    return
end

figure()
pdegplot(geom, 'EdgeLabels','on',"FaceLabels","on")
xlabel("x")
ylabel("y")
title("Resonator mesh")
end
```

5.2.10 triangulation2d

```
% author: Mauro Morini
% last modified: 06.11.23
function [p, t, e] = triangulation2d(p)
% creates a 2d triangulation with given points p
% Outputs :
% p :   coordinate matrix nPx2 containing points (x,y) in rows,
%       representing the verteces of the triangles
% t :   connectivity matrix nTx3 each row representing one element
%       and each column representing the local numbering (0,1,2) and the
%       entry the global numbering
% e :   connectivity matrix nEx2 for edges on the boundary
%
% Input :
% p :   same as output but with possible duplicates
DT = delaunayTriangulation(p);

% eliminate duplicates
p = DT.Points;
t = DT.ConnectivityList;
e = freeBoundary(DT);

end
```

5.2.11 generateMesh2dUnitSquare

```
% author: Mauro Morini
% last modified: 06.11.23
function [p, t, e] = generateMesh2dUnitSquare(h)
% creates an equidistant mesh and a triangularization on the unit square
% (0,1)x(0,1)
```

```
%
% Outputs :
% p :   coordinate matrix nPx2 containing points (x,y) in rows,
%       representing the verteces of the triangles
% t :   connectivity matrix nTx3 each row representing one element
%       and each column representing the local numbering (0,1,2) and the
%       entry the global numbering
% e :   connectivity matrix nEx2 for edges on the boundary of the form
%       (i,j), where i and j are indexes of points on the boundary
%
% Inputs :
% h :   meshsize (equidistant)

[X, Y] = meshgrid(0:h:1, 1:-h:0);
N = size(X, 1);

% construct points p
p = zeros(N^2, 2);
k = 1;
for i = 1:N
    for j = 1:N
        % x coordinate
        p(k, 1) = X(i,j);

        % y coordinate
        p(k, 2) = Y(i,j);
        k = k + 1;
    end
end

[p, t, e] = triangulation2d(p);
end
```