CHALMERS UNIVERSITY OF TECHNOLOGY

Simulations of waves in a bath tub

TMA690 Partiella differentialekvationer: uppgift 3, rapport

Boström Isac isacb@student.chalmers.se
Lindqvist Jakob jaklindq@student.chalmers.se

Wednesday 21st December, 2016

1 Theory

1.1 The physical problem

We wish to model waves in a bath tub or more specifically the water surface of a bath tub under different conditions. We will do this by solving the wave equation in 2 spatial dimensions by means of the finite element method and for different times via a linearisation. Furthermore we will model different kind of behaviours of this water surface by imposing certain conditions. The success of the model will largely be evaluated using graphical observations.

1.2 Weak formulation

We begin by transforming the original analytic wave equation into something that we can use. The wave equation:

$$\begin{cases} \ddot{u} - \Delta u = f, & \text{on } [0,T] \times \Omega \subset \mathbb{R}_+ \times \mathbb{R}^2 \\ \vec{n}\nabla u = 0, & \text{on } [0,T] \times \partial\Omega \\ u(0,\vec{x}) = u_0, & \dot{u}(0,\vec{x}) = \dot{u_0}. \end{cases} \tag{D}$$

Multiplying (D) with a test function $v \in V = \{v : v \in L^2(\Omega), \nabla v \in L^2(\Omega)\}$ and integrating over the space Ω we obtain the weak formulation

$$\int_\Omega (\ddot{u} - \Delta u) v d\vec{x} = \int_\Omega f v d\vec{x}, \qquad \forall v \in V \tag{1}$$

Integrating the space integral of the left hand side by parts using Green's formula we loose the second order derivatives

$$\int_\Omega \Delta u v d\vec{x} = \int_\Omega \nabla u \nabla v d\vec{x} + \int_{\partial\Omega} \vec{n}\nabla u v dS \tag{2}$$

Where the last term vanishes due to the boundary conditions in (D). Using this in (1) we obtain the variational formulation

$$\int_\Omega (\ddot{u}v + \nabla u \nabla v) d\vec{x}, \qquad \forall v \in V. \tag{V}$$

1.3 Discretisation

Using the weak formulation we wish to formulate an approximation of the solution on a discretized space Ω_h. For this task Ω_h will be Ω divided into a finite

number of triangular subspaces called *elements* (further explained in section 1.4). The vertices of the elements are called *nodes*. We define N to be the number of nodes and on these nodes we define piece wise linear basis functions

$$\{\varphi_i\}_{i=1}^{N} = \begin{cases} 1, & \vec{x} = \vec{x}_i \\ 0, & \vec{x} \neq \vec{x}_i \end{cases} \tag{3}$$

where $\vec{x}_i$ is the is the coordinates of the node i. Using this basis we can approximate u with a linear combination

$$u(t,\vec{x}) \approx u_h(t,\vec{x}) = \sum_{i=1}^{N} \xi_i(t)\varphi_i(\vec{x}). \tag{4}$$

Since φ_i is 1 in the node i and 0 everywhere else the coefficient ξ_i is simply the value of u_h at $\vec{x}_i$ at time t, this means that if we can determine these coefficients at every time step we have a complete approximation of u. Furthermore, φ_i is constant in time which renders the derivatives in time easy to calculate:

$$\ddot{u}_h(t,\vec{x}) = \sum_{i=1}^{N} \ddot{\xi}_i(t)\varphi_i(\vec{x}). \tag{5}$$

We substitute u for u_h and choose every φ_j as our test functions v. This transforms eq. (V) into a system of equations

$$\sum_{i=1}^{N} \ddot{\xi}_i(t) \int_{\Omega_h} \varphi_i(\vec{x})\varphi_j(\vec{x})d\vec{x} + \sum_{i=1}^{N} \xi_i(t) \int_{\Omega_h} \nabla\varphi_i(\vec{x})\nabla\varphi_j(\vec{x}))d\vec{x} = \int_{\Omega_h} f\varphi_j \tag{6}$$

This equation can be written substantially more pleasant by defining the square matrices M and S and vectors $\vec{F}$ and $\vec{\xi}$ with elements

$$\begin{cases} M_{ji} = \int\limits_{\Omega_h} \varphi_j(\vec{x})\varphi_i(\vec{x})d\vec{x} \\ S_{ji} = \int\limits_{\Omega_h} \nabla\varphi_j(\vec{x})\nabla\varphi_i(\vec{x}))d\vec{x}, \\ \vec{\xi} = \{\xi_1, \ldots, \xi_M\}^T \\ F_j = \int\limits_{\Omega_h} f\varphi_j(\vec{x}), \qquad j = 1, \ldots, N \end{cases} \tag{7}$$

for which we get the matrix equation:

$$M\ddot{\vec{\xi}} + S\vec{\xi} = \vec{F}. \tag{8}$$

In order to solve this numerically we must also approximate $\ddot{\xi}_i$ in terms of ξ_i and discrete time sequence with step length Δt

$$\ddot{\xi}_i \approx \frac{\xi_i(t_{l+1}) - 2\xi_i(t_l) + \xi_i(t_{l-1})}{\Delta t^2} [1, \text{eq. } 8.21a]. \tag{9}$$

where $l = 0, \ldots, T$ which means that for this scheme to work we actually need two initial values, namely $\xi(t_0)$ and $\xi(t_{-1})$. The former is easy enough since it corresponds to $u_0(0,\vec{x}_i)$. For the latter we need to extrapolate using a backwards Taylor expansion of sorts:

$$\begin{cases} Ma_0 = f_0 - Su_0 \\ u_(- \Delta t,\vec{x}_i) = \xi(t_{-1}) = u_0 - \Delta t\dot{u}_0 + \frac{\Delta t^2}{2} a_0. \end{cases} \tag{10}$$

With the initial values calculated we arrive to a recursive equation for a discrete evolution in time of the coefficients ξ_i:

$$\begin{aligned} &\frac{M}{\Delta t^2}\left(\vec{\xi}(t+\Delta t) - 2\vec{\xi}(t) + \vec{\xi}(t-\Delta t)\right) + S\vec{\xi} = \vec{F} \iff \\ &\vec{\xi}(t+\Delta t) = 2\vec{\xi}(t) - \vec{\xi}(t-\Delta t) + \Delta t^2 M^{-1}(\vec{F} - S\vec{\xi}) \end{aligned} \tag{11}$$

1.4 Geometry and numerical methods

The partition of Ω is called Ω_h and it is built up by small elements. This partition or *tesselation* is auto generated by the function `initmesh` from Matlab's PDE toolbox. The fineness of the tesselation is measured by $h_{\max}$ which is the largest side of an element in the mesh. One such tesselation can be seen in figure 1.

Given a mesh we define N piece wise linear basis functions as stated in the previous section. Due to the triangular shape of the elements we get pyramidal $\varphi_i(x)$ which means that each function has a nonzero part that extends from its node to, but not exceeding, the closest surrounding nodes, see figure 2.

This however makes the calculations of the matrix elements M_{ij} and S_{ij} (eq. (7)) rather tedious since we need to consider interactions between multiple basis functions for every node. We work our way around this by, instead of considering one node at a time, considering the elements separately. That is, we calculate the integral in question on the triangular parts of the domain where φ_i is nonzero. Then we add these element integrals to build up the entire domain where φ_i is nonzero.

The advantage of this scheme is that for another basis function φ_l, overlapping with φ_i, the exact same integral on the same element will appear again and we can simply assign the same term to the sum that evaluates to *that* matrix element, see figure 3.

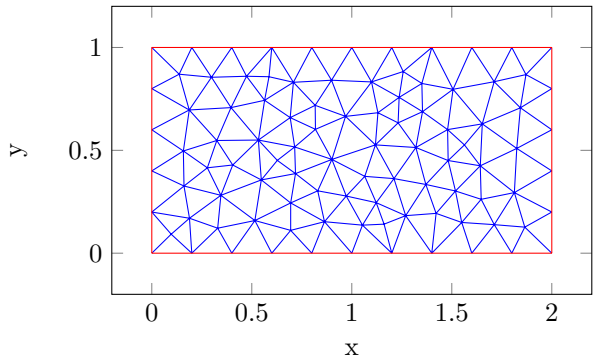

Figure 1: Bath tub meshed using `initmesh`. The small triangles are called elements and the vertices of the elements are called nodes. $h_{\max}$ is defined as the longest side of one element. Note that a finer mesh is used for the simulations presented later.

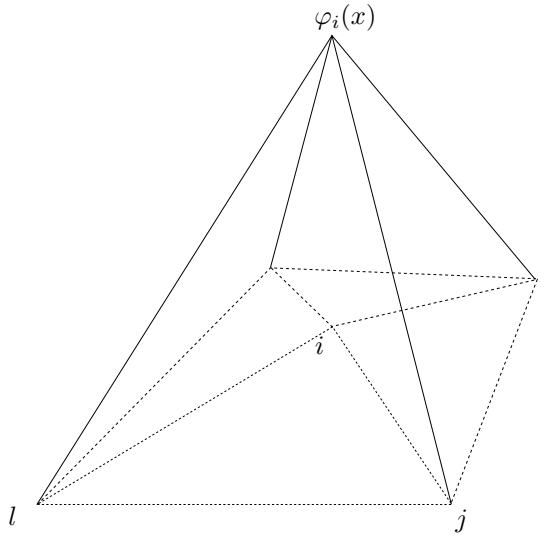

Figure 2: The nonzero parts of a basis function $\varphi_i(x)$. The function takes the value 1 at the node i and goes linearly to zero on the surrounding elements and remains zero throughout the rest of Ω_h.

For the actual calculation of the integrals we will use some approximations. For the $M_{ji}-$integral on element k, we will use the mid point approximation [1, p. 246]

$$\int_{\Delta_k} \varphi_i(x)\varphi_j(x)dx \approx \frac{area(\Delta_k)}{3} \sum_{i=1}^{3} |ij|/2 = \frac{area(\Delta_k)}{4}, \tag{12}$$

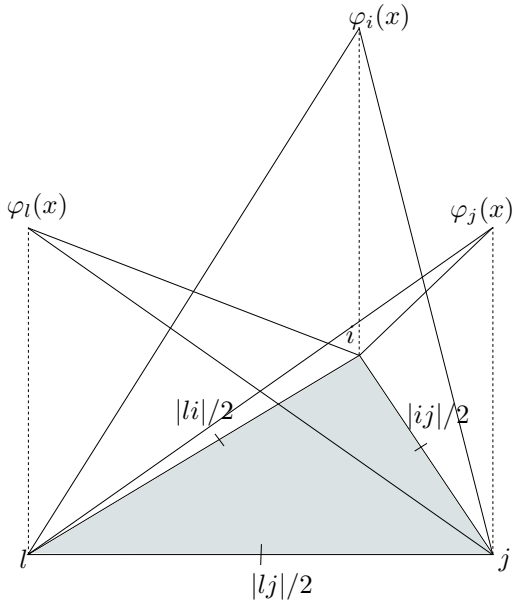

Figure 3: An element in the Ω_h with the the basis functions that are nonzero on the element. Note how they overlap and each function take the value $1/2$ on the mid point $|nn'|/2$ of each side.

since the value at the midpoint ($|ij|/2$) is always $\frac{1}{2}\frac{1}{2} = \frac{1}{4}$ (for the diagonal elements we get this value twice), see figure 3.

The S_{ji}'s involve gradients of the linear basis functions which are constant on every element, so these can actually be calculated exactly and are simply a function of the coordinates of the element vertices.

The F_j's are done with a centre point approximation [1, p. 246]

$$\int_{\Delta_k} \varphi_j f(c_k) dx \approx area(\Delta_k)\varphi_j f(c_k), \tag{13}$$

where c_k is the centre point of element k.

2 Results

In this section we show results for simulations of different conditions. For comparison we have used the same parameters for all simulations, i.e. $h_{\max} = 0.05$ and $\Delta t = 0.001$ have remained constant for each part of the problem posed.

2.1 Non-trivial initial conditions u_0 with source term $f(t,\vec{x}) \equiv 0$

First we consider the case where the source term f is zero for every $\vec{x}$ and t. This require non-trivial initial value u_0 if we want to achieve something different than a flat surface. Here we use

$$u_0 = \frac{cos(5\pi|\vec{x}|)}{1+10|x|}. \tag{14}$$

The initial conditions should be viewed as the initial distribution of the surface; a frozen snapshot which we start from whence the water becomes unfrozen. The surface will then change with each iteration according to the wave equation. Note that we don't have any friction or other energy decreasing phenomenon which means that the waves should continue forever but we still expect it to reach some kind of equilibrium in the sense that the waves and particularly their amplitudes should be evenly distributed in the tub.

In figure 4 we include some snapshots of the surface for different time steps and we confirm that the system behaves according to our expectations.

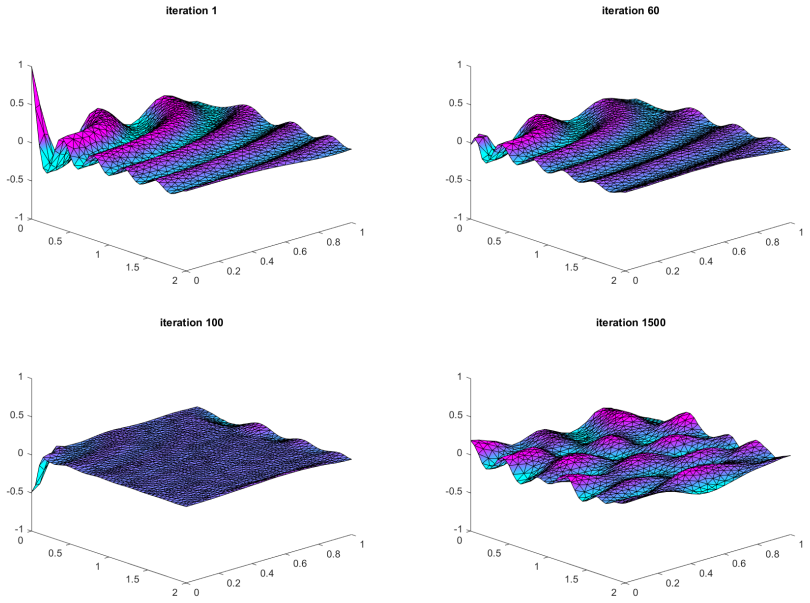

Figure 4: Snapshots of the simulation at four different iterations. In the top-left corner we have the initial distribution of the surface. In the top right and bottom left we see how the largest wave top fluctuates the most and note also that for a large number of iterations (bottom right) the heights of the wave tops are more uniformly distributed.

2.2 Non-trivial source term $f(t,\vec{x})$ $(u_0 \equiv 0)$

Now we seek to implement a source term corresponding to some physical interpretation. Viewing this as a literal source of, in this case, water can be helpful in understanding whats going on. We sought to model drips of water dripping in to the tub at certain points (actually nodes) at certain times. We model this numerically by having the source term

$$f(\vec{x},t) = \begin{cases} f_0 < 0, & \vec{x} = \vec{x}_{drop}, t = t_{drop} \\ 0, & \text{otherwise .} \end{cases} \tag{15}$$

This will only affect the loadvector F which will be 0 for all entries except the dripping point which in turn will be equal to $\int_{\Omega_h} \varphi_{drop} f = \frac{1}{2}(\text{ area under } \varphi_{drop}) f_0$. This means that we actually have to recalculate this for every iteration. In figure 5 we see the results of this simulation and we observe a rather nice result with a surface behaviour that accurately resembles a drop falling into water.

2.3 Non-trivial initial velocity condition $\dot{u}_0$, $(u_0, f \equiv 0)$

If we repeat the basic simulation but instead of u_0 take a non-trivial distribution of $\dot{u}_0$ we get the result shown in figure 6. By reasoning that we wanted $\dot{u}_0$ to be the velocities of a wave at a certain time we chose

$$\dot{u}_0(\vec{x}) = cos(5\pi|\vec{x}|) \tag{16}$$

The simulations are somewhat similar and that is reasonable since the height and vertical velocity are linked. We can think of it like it isn't an initial condition at all but rather that we have chosen some u_0 and then we freeze the simulation at a certain iteration when u_0 happens to be zero for every $\vec{x}$, even though the surface is in a non static state, and then we resume the simulation. This could also be used to model external disturbances on the tub, e.g. if the tub is shifted or rocked in any way this would not display in the surface at that exact same time but rather offset the velocities which would then affect the surface in subsequent iterations.

2.4 Dirichlet initial conditions

We now impose Dirichlet boundary condition, i.e.

$$u(t,\vec{x}) = 0, \vec{x} \in \partial\Omega. \tag{17}$$

The Neumann condition is no longer active but the equation (V) remains the same as we impose the Dirichlet condition on the test functions (basis functions) and this instead causes the integral over the boundary to vanish. The matrix

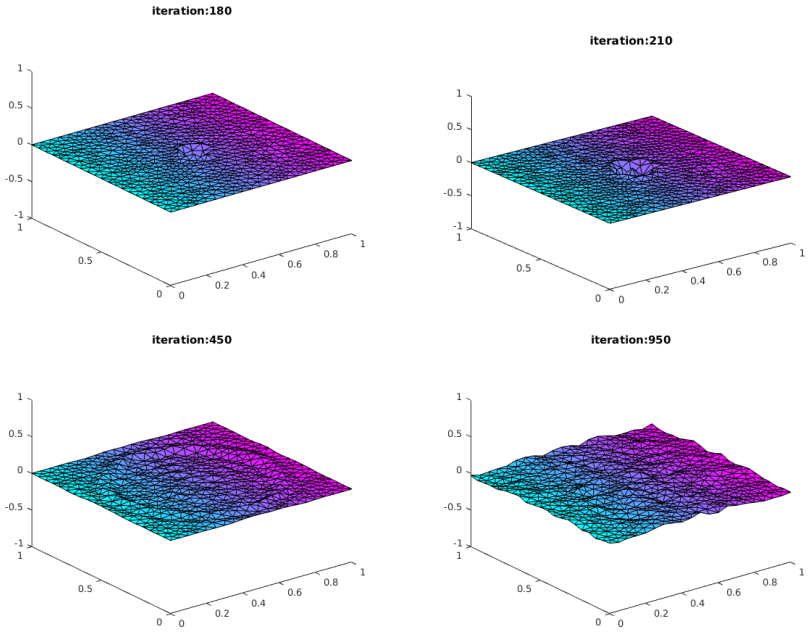

Figure 5: Snapshots of the simulation, for four different iterations with $f_0 = -20$. Here we see the impact of a simulated drop. At iteration 80 the surface is first disturbed and a circular wave begins to propagate and in iteration 210 we also see the recoil in the centre where the drop hit, causing secondary waves. In iteration 450 the surface has recognisable pattern of rings of water. Iteration 950 shows that we reach the same kind of equilibrium with this setup as well and we see no discernible circular waves as these have been blurred out by interference. Note that there is a clear disk of influence, outside this disk the surface is completely unaffected by the drop. This means that the waves propagate with a finite (uniform) speed.

equation is changed in the sense that we omit all rows and all rows and columns for F and M and S respectively corresponding to vertices on the boundary ($\vec{x}_i \in \partial\Omega_h$). This of course reduces the dimension of the vector containing all coefficients ξ_i but since the boundary conditions explicitly states that these omissions should always be zero we simply add these zero entries later. All the information of elements on the boundary are given by the `initmesh` function. Some snapshots of the simulation under the Dirichlet condition are shown in figure 7.

These conditions obviously offers a poor simulation of a bathtub since it is physically nonsense to have a fixed boundary of a non static liquid surface. This should rather be interpreted as some sort of vibrating membrane, e.g. a drum skin or a loudspeaker.

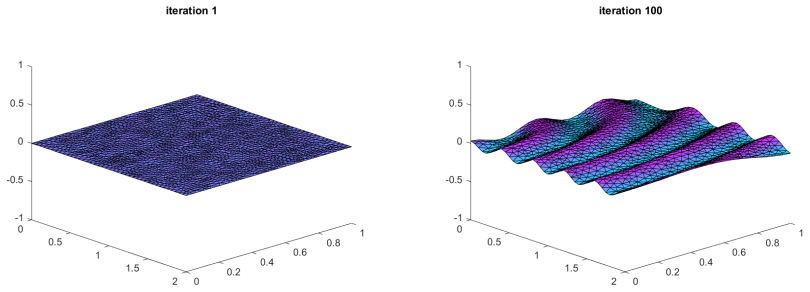

Figure 6: Snapshots of the simulation, with trivial initial distribution, u_0, but non-trivial initial velocities, $\dot{u}_0$, at two different iterations. Here we see that the surface is still even though $\dot{u}_0(t_1) \neq 0$.

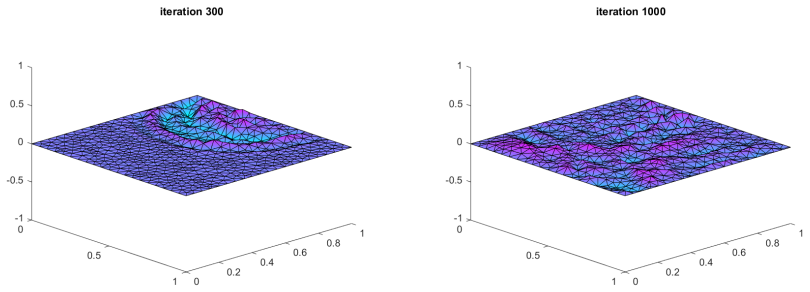

Figure 7: Snapshots of the simulation with the Dirichlet boundary condition as well as time dependent source function. We chose the same source as the one in figure 5 for comparison not that the surface behaves seemingly similar to that of the Neumann condition in the interior but that it has to adapt to the fixed boundary. This is a clear analogy to how a vibrating membrane would behave.

3 Numerical errors

Throughout this process we have repeatedly discovered that there exists clear numerical limitations. One obvious thing is that the mesh needs to be fine enough ($h_{\max}$ small enough) for there to be any resemblance at all to a wave. This is not an error, strictly speaking, but it needs to be addressed since we assess our simulations through ocular inspection and thus our conclusion requires that we get some physical feasibility. A more interesting phenomenon is that the geometry and mesh combined set an upper bound for how large time steps we can use in the simulation. There is a very strict phase shift from a stable to an extremely unstable solution (diverges greatly in just a few iterations) once you traverse this bound ever so slightly. We take this to be a consequence of the fact that the wave has a finite velocity. This means that if we sample to seldom

(large time steps) the disk of influence expands two fast and can affect outer nodes at the same time as an inner one which causes these numerical errors.

References

[1] C Johnson, *Numerical Solution of Partial Differential Equations by the Finite Element Method* (1987), Cambridge University Press

```
cd ~/Chalmers/TMA690_PDE/
% addpath ~/Script/matlab2tikz/src/
clear all
clc
%%%%%%%%%%%%%%
%% Basic Part
%%%%%%%%%%%%%%

hmax = 0.05;
tubGeom =[  3 %Specify rectangle
    4
    0 % x-coord of vertices
    2
    2
    0
    0 % y-coord of vertices
    0
    1
    1];

% tubGeom =[  4 %Specify rectangle
%     1
%     0.5 % x-coord of vertices
%     1
%     0.5
%     0
%     0
%     ];

[g,bt] = decsg(tubGeom);
[p,e,t] = initmesh(g,'hmax',hmax);
% figure(1)
% clf
% pdemesh(p,e,t)
% axis([-.5 2.5 -.5 1.5])
% xlabel('x')
% ylabel('y')
% legend('Discretisation')
% axis equal

%
% Given the mesh; generate matrices M, S, and vector F.
m = size(p,2);
dt = 0.001;
M = MassMatrix(p,t);
Minv = inv(M);
S = StiffnessMatrix(p,t);
F = zeros(m,1);
xi = InitialXi(Minv,S,p,dt);

A = dt^2*Minv*F;
B = dt^2*Minv*S;


fig = figure(2);
tid = 0;
speedParam = 8;
```

```
while ishandle(fig)
    tid = tid + 1;
    xi =[xi (2*xi(:,end)-xi(:,end-1)+A-B*xi(:,end))];

    xi = xi(:,end-2:end); % Crop xi to reduce memory load; Comment out
    % if you wish to obtain whole solution.
    % Plot each speedParam time step.
    if mod(tid,speedParam) == 0

        pdeplot(p, e, t, 'zdata', xi(:,end), 'xydata', xi(:, end),...
            'mesh','on','colorbar','off');
        set(gca,'ZLim',[-1 1])
        drawnow

    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% NON TRIVIAL SOURCE TERM f
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
cd ~/Chalmers/TMA690_PDE/

clear all
hmax = 0.05;
tubGeom =[  3 %Specify rectangle
    4
    0 % x-coord of vertices
    1
    1
    0
    0 % y-coord of vertices
    0
    1
    1];

% tubGeom =[  4 %Specify ellipse
%     1
%     0.5
%     1
%     0.5
%     0
%     0
%     ];

% Generate geometry for the rectangle (pre-built functions)
[g,bt] = decsg(tubGeom);
[p,e,t] = initmesh(g,'hmax',hmax);


% Parameters and constants
dt = 0.001;
T = 2e3;
m = size(p, 2);
% Place drop in centre
% dropInd = randi(m)
[~,dropInd] = min(sum(([0.5; 0.5] - p).^2,1));
dropTime = 150;
```

```
f = -20 % Value of source term (Diracesque)

% Matrices and and preallocation
xi = zeros(m, T);
F = LoadVector(p,t,hmax,f,dropInd,dropTime,T);
S = StiffnessMatrix(p, t);
M = MassMatrix(p, t);


% Iterate for xi(t+1)
for tIter = 3:T
    b = dt^2 * F(:, tIter) + M*(2*xi(:, tIter - 1) - xi(:, tIter-2)) - dt^2 * S * xi(:, tIter-1);
    xi(:, tIter) = M\b;
end

% Plotting
fig = figure(2);
clf
for tIter = 1:2:T
    pdeplot(p, e, t, 'zdata', xi(:,tIter), 'xydata', p,...
        'mesh','on','colorbar','off');
    set(gca,'ZLim',[-1 1])
    str = strcat('Time until drop: ', num2str((dropTime-tIter)*(dropTime>tIter)));
    text(0.9,0.9,0.9,str)
    drawnow
end

%% Plotting certain iterations
figure(3)
clf
iter = dropTime + 800;
pdeplot(p, e, t, 'zdata', xi(:,iter), 'xydata', p,...
        'mesh','on','colorbar','off');
    set(gca,'ZLim',[-1 1])
    title(strcat('iteration: ',num2str(iter)))
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% DIRICHLET BOUNDARY CONDITIONS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
cd ~/Chalmers/TMA690_PDE/

clear all
hmax = 0.05;
tubGeom =[  3 %Specify rectangle
    4
    0 % x-coord of vertices
    1
    1
    0
    0 % y-coord of vertices
    0
    1
    1];

% tubGeom =[  4 %Specify ellipse
%     1
%     0.5clear
%     1
```

```
%     0.5
%     0
%     0
%     ];

% Generate geometry for the rectangle (pre-built functions)
[g,bt] = decsg(tubGeom);
[p,e,t] = initmesh(g,'hmax',hmax);


% Parameters and constants
dt = 0.001;
T = 2e3;
m = size(p, 2);


% Matrices and and preallocation
xi = zeros(m, T);
F = zeros(m, T);
S = StiffnessMatrix(p, t);
M = MassMatrix(p, t);

% Remove boundary vertices from the problem.
[F,S,M,BoundNodesCompl] = Dirichlet(F,S,M,e);

% Generate load vector corresponding to source term (Diracesque)
% dropInd = randi(m)
dropInd = 280
F(dropInd, 20) = -40;


% Iterate for xi(t+1)
for tIter = 3:T
    b = dt^2 * F(:, tIter) + M*(2*xi(BoundNodesCompl, tIter - 1) - xi(BoundNodesCompl, tIter-2)) - d
    xi(BoundNodesCompl, tIter) = M\b;
end

fig = figure(2);
clf
for tIter = 1:T
    pdeplot(p, e, t, 'zdata', xi(:,tIter), 'xydata', p,...
        'mesh','on','colorbar','off');
    set(gca,'ZLim',[-1 1])
    drawnow
end


function [M] = MassMatrix(p,t)

% Generating the mass matrix for a given mesh and base functions-
m = size(p, 2);
M = zeros(m, m);
for el = 1 : size(t, 2)
    dM = ElemMassMatrix(p, t, el);
    nn = t(1:3, el);
    M(nn, nn) = M(nn, nn) + dM;
end
```

```
end

function dM = ElemMassMatrix(p, t, el)
    % returns the element diffusion matrix dD for element number
    % el defined by t(1:4, el) and p(t(1:3, el).
    NodeCoords = p(:, t(1:3, el));
    dx = ElemArea(NodeCoords);
    dM = dx/12 * [2,1,1;1,2,1;1,1,2];
end


function S = StiffnessMatrix(p, t)
    n = size(p, 2);
    S = zeros(n, n);
    for el = 1 : size(t, 2)
        dD = ElemStiffnessMatrix(p, t, el);
        nn = t(1:3, el);
        S(nn, nn) = S(nn, nn) + dD;
    end
end

function dS = ElemStiffnessMatrix(p, t, el)
    NodeCoords = p(:, t(1:3, el));
    Dphi = ElementPhiGradients(NodeCoords);
    dx = ElemArea(NodeCoords);
    dS = Dphi' * Dphi * dx;
end

function Dphi = ElementPhiGradients(Nodes)
    % returns the gradients of the three element basis functions phi on a
    % triangle with nodes Nodes
    v = Nodes(:,3)-Nodes(:,1);
    w = Nodes(:,2)-Nodes(:,1);
    gr = v - dot(v,w)*w/norm(w)^2;
    Dphi3 = gr/norm(gr)^2;
    gr = w - dot(w,v)*v/norm(v)^2;
    Dphi2 = gr/norm(gr)^2;
    Dphi1 = - (Dphi2 + Dphi3);
    Dphi = [Dphi1 Dphi2 Dphi3];
end


function [F] = LoadVector(p,t,hmax,sourceValue,dropInd,dropTime,endTime)
    % Generating the load vector for a given function f(t,x), mesh and
    % base functions. Will have to be updated each time for a time dependent f



    m = size(p, 2);
    F = zeros(m, endTime);
        for el = 1 : size(t, 2) % el = triangel
            dF = ElemLoadVector(p, t,el,hmax,sourceValue,dropInd);
            F(dropInd, dropTime) = F(dropInd, dropTime) + dF;
        end
end
```

```
function dF =  ElemLoadVector(p,t,el,hmax,sourceValue,dropInd)
    f = sourceValue/hmax^2; % Scale source with geometry.
    nodes = p(:,t(1:3,el) );

    % Element containing drop index
    isDropInd = sum(t(1:3,el)==dropInd);
    dF = isDropInd*f*ElemArea(nodes)/2;
end


function [xi0] = InitialXi(Minv,S,p,dt)
xi0 = zeros(size(p,2),1);
% Generating the first two vectors of the xi matrix. Which are needed for
% the recursive temporal evolution of xi(t).
% The entries are: xi(1) = xi(t_-1) = u_0(:,x_i) and xi(:,2) = xi_0

% f0 = SourceFunction(p,0,pi,1);
f0 = 0;
[u0,v0] = InitialConditions(p);
a0 = Minv*(f0 -S*u0);


xi0(:,1) = u0 - dt*v0 + dt^2/2*a0;

xi0(:,2) = u0;

end

function [u0,v0] = InitialConditions(x)
n = size(x,2);
%u0 = cos( 5*pi*sqrt( sum(x.^2,1) ) ) ./ ( 1 + 10* sqrt( sum(x.^2,1) ) ); %PART I
u0 = zeros(1,size(x,2)); %PART II, FOR DROP AND NON TRIVIAL V0
%v0 =  zeros(1,n);
v0 = cos( 5*pi*sqrt( sum(x.^2,1) ) ); %NON TRIV V0

u0 = u0';
v0 = v0';
end


function ar=ElemArea(NodeCoords)

% returns the area of a triangular element with given node coordinates

  x1 = NodeCoords(1,:);
  x2 = NodeCoords(2,:);
  ar = (x1(2)-x1(1))*(x2(3)-x2(1))-(x1(3)-x1(1))*(x2(2)-x2(1));
  ar = ar/2;
end


function [F,S,M,BoundNodesCompl] = Dirichlet(F,S,M,e)
% Truncates all the rows (and columns for S,M) of the vector (and matrices)
% corresponding to vertices on the boundary dOmega. Also returns indices
% for all elements not on the boundary to be used to single out which xi:s
% to be updated in the iteration.
```

```
m = size(M,1);
BoundNodes = [e(1,:) e(2,:)];
BoundNodes = unique(BoundNodes);

BoundNodesCompl = comple(BoundNodes,m);

S(:,BoundNodes) = [];
S(BoundNodes,:) = [];
M(:,BoundNodes) = [];
M(BoundNodes,:) = [];
F(BoundNodes,:)= [];


end
```