Project in applied finite element method

Fredrik Gustafsson F4.B

November 2018

1 Introduction

Solving partial differential equations, pde, is of a great interest within the fields of science and engineering. They appear in problems such as vibrations, heat transfer and diffusion to name a few. Most are, however, at best time consuming and at worst impossible to solve analytically. A tool to help solve such problems is the finite element method.

The finite element method "works" by descretize the continuous input space into finite elements, nodes, where each element has a fixed coordinate. To solve the pde one then solves the linear system of equations where each equation being constructed by using a basis function in the test function space.

In this project the following problem was to be solved

$$\partial_t u(x) - \alpha \Delta u(x,t) = f(x), \quad u(x,t) \in B \quad \text{x} \quad [0,T] \tag{1}$$

$$u(x,t) = 0, \quad u(x,t) \in \partial B \quad \text{x} \quad [0,T] \tag{2}$$

$$u(x,0) = \begin{cases} \rho, & x \in T \\ 0, & x \in B \backslash T \end{cases}, \quad x \in B \tag{3}$$

where $\alpha = 0.01\text{mm}^2/\text{day}$. The geometry of the problem was a torus with major and minor radii as R and r. To start the problem was solved in 1D followed by 2D and finally the full 3D problem as well as optimization of the parameters ρ, R, and r.

2 Theory

2.1 The finite element method in 1D

A model problem for a pde in 1D is the equation

$$-au'' = f, x \in (0,1) \tag{4}$$

$$u(0) = u(1) = 0 \tag{5}$$

To solve this problem with the finite element method we start by discretize the problem on the interval into n amount of subintervals. Since each subinterval has two end points we get $n+1$ endpoints. In order to solve the problem we have to calculate the value of the function at each intervals endpoint, the value of each node. This approximation of the solution to the function can be written in matrix form as $A\xi = B$ where ξ is the unknown coefficients at each node, A the so called stiffness matrix and B the load vector.

In order to start to try to solve the problem we introduce a test function V_h belonging to the space containing all picewise linear functions on the whole interval. Further we introduce the subspace $V_{h,0}$ which fulfills our boundary conditions. By then multiplying each side of the equation with a test function $v \in V_{h,0}$ integration both sides over the interval and applying our boundary condition we arrive at

$$\int_0^1 u_h' v' dx = \int_0^1 f v dx \tag{6}$$

where u_h is the approximate finite element solution. A basis for $V_{h,0}$ is then obtained by deleting the half hats at the start and end from the set that spans V_h. This sort of method is referred to as the Galerkin method. We can then replace the test function v with the hat function φ_i whose solution is known at the nodes.

Next step is then be to assemble the stiffness matrix and the load vector. The Stiffness matrix A can be assembled by calculating the integrals

$$A_{i,i} = \int_{x_{i-1}}^{x_i} \alpha {\varphi_i'}^2 dx + \int_{x_i}^{x_{i+1}} \alpha {\varphi_i'}^2 dx = \int_{x_{i-1}}^{x_i} \alpha_i \frac{1}{h_i^2} dx + \int_{x_i}^{x_{i+1}} \alpha_{i+1} \frac{(-1)^2}{h_{i+1}^2} dx = \frac{2\alpha}{h} \tag{7}$$

$$A_{i,i+1} = \alpha \int_{x_i}^{x_{i+1}} \varphi_i' \varphi_{i+1}' dx = \alpha \int_{x_i}^{x_{i+1}} \frac{-1}{h} \frac{1}{h} = -\frac{\alpha}{h} \tag{8}$$

The elements $A_{i+1,i}$ can be calculated the same way as equ.8 and becomes $-\frac{\alpha}{h}$. Every other elements in A will be zero due to the properties of the hat function. The load vectors elements B_i can be calculated, however as these elements depends on the function f we approximate them with a quadrature rule.

$$B_i = \int_{x_{i-1}}^{x_i} f\varphi_i dx + \int_{x_i}^{x_{i+1}} f\varphi_i dx \approx f(x_i) \frac{h_i + h_{i+1}}{2} \tag{9}$$

Then we just need to assemble each matrix. For the stiffness matrix we just need to calculate the diagonal and super diagonal elements in order to assemble

it. This is doable due to the stiffness matrix being sparse meaning that we just have to calculate the 2 by 2 matrix containing the diagonal and superdiagonal elements. Repeating it for each subinterval and sum them up to get the matrix for the full interval I.

$$B^I = \frac{1}{2}\begin{bmatrix} f(x_i) \\ f(x_{i+1}) \end{bmatrix} h \tag{10}$$

$$A^I = \frac{\alpha}{h}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} h. \tag{11}$$

The final solution u_h can then be calculated as $A \backslash B$.

In the case where we instead of having an integral over u'v' instead want to integrate over uv we get the mass matrix M instead. The elements of the mass matrix can be calculated with the Simpson's formula as

$$M_{i,i} = \int_a^b \varphi_i^2 dx = \int_{x_{i-1}}^{x_i} \varphi_i^2 dx + \int_{x_i}^{x_{i+1}} \varphi_i^2 dx = \frac{0 + 4\frac{1}{2}^2 + 1}{6} h_i + \frac{1 + 4\frac{1}{2}^2 + 0}{6} h_{i+1} = \frac{h_i}{3} + \frac{h_{i+1}}{3} \tag{12}$$

$$M_{i,i+1} = \int_{x_i}^{x_{i+1}} \varphi_i \varphi_{i+1} = \frac{4\frac{1}{2}^2}{6} h_{i+1} = \frac{h_{i+1}}{6}, \tag{13}$$

$M_{i+1,i}$ is calculated in the same way as equ.13. Assembly of the mass matrix is then done in the same way as the stiffness matrix giving us

$$M^I = \frac{1}{6}\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} h. \tag{14}$$

2.2 Finite element method in 2D

If we want to solve a problem in two dimensions we begin with defining a spatial discretization, in one dimension this was done by "just" putting nodes on a line. However in two dimensions we need something more advanced namely a mesh in 2 dimensions. Such a mesh is done by triangulation, splitting the domain into smaller and smaller triangles while trying to keep them as regular as possible. This then means that each element will be defined by three points located in each corner of the element.

Once we've created our mesh we need a test space, just as in 1D, which consists of all picewise linear polynomials whom are continous between each element triangle.

$$V_h = \{v : v \in C(\Omega), v|_k \in P_1(K) \forall K \in K\} \tag{15}$$

where p_1 is the space of all linear polynomials and K is our mesh. Since problems in 2D will have some boundary condition we also define a subspace V_0 to V_h that full fills said condition.

To start solving problems we need three matrices, the mass matrix,M, stiffness matrix, A, and load vector, b. The complete derivation of each matrix can be found in "The Finite Element Method: Theory, Implementation, and Practice"[1] however in the end the local element mass matrix can be computed as

$$M^K = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} |K| \tag{16}$$

where K is the area of the local element, to assemble the global mass matrix you then use a local to global mapping. The global load vector can be assembled by computing

$$b_i^K = \int_K f\phi_i dx, \quad i = 1, 2, 3 \tag{17}$$

$$b_i^K \approx \frac{1}{3} f(N_i)|K|, \quad i = 1, 2, 3 \tag{18}$$

where we use node quadrature to approximate the integral and then once again use a local to global mapping for the 3 by 1 local load vector. Finally the stiffness matrix can be computed as

$$A_{i,j}^K = \int_K \nabla\phi_i \nabla\phi_j dx, \quad i, j = 1, 2, 3 \tag{19}$$

which, just as for the mass matrix, becomes a local 3 by 3 matrix multiplied by the element area and then to assemble it into a global stiffness matrix a local to global map is used.

2.3 Time discretization, and answer to question B2

In order to solve a time dependant problem, such as a non stationary heat equation, we have to find a way to discretize the time. If we start with the problem

$$\dot{u} - \alpha\Delta u = f(x), \quad u(x,t) \in B \quad \text{x} \quad [0,T] \tag{20}$$

$$u(x,t) = 0, \quad u(x,t) \in \partial B \quad \text{x} \quad [0,T] \tag{21}$$

$$u(x,0) = \begin{cases} \rho, & x \in T \\ 0, & x \in B \backslash T \end{cases}, \quad x \in B \tag{22}$$

and in this case setting $f(x) = 0$. Then discretizing in space is as before, however since we have $\dot{u}$ we also get the mass matrix, compare to having u' which also yields the mass matrix. This then leaves us with a spatial discretization on the form

$$M\dot{\xi}(t) + A\xi(t) = b. \tag{23}$$

In order to discretize equ. 21 with respect to time we'll integrate it from $t-1$ to t and since M and A are constants we get

$$M\int_{t-1}^{t}\dot{\xi}(t)+A\int_{t-1}^{t}\xi(t)=\int_{t-1}^{t}b. \tag{24}$$

The first integral then becomes $M(\xi(t_l)-\xi(t_{l-1})$. The remaining integrals can be evaluated in some different ways. If we start with using right end-point quadrature and rearranging a bit we get

$$(M+k_lA)\xi_l=M\xi_{l-1}+k_lb_l \tag{25}$$

which is the backward Euler scheme. IF we instead use left end point quadrature we get

$$M\xi_l=(M-k_lA)\xi_{l-1}+k_lb_{l-1} \tag{26}$$

which is the forward Euler scheme. Finally adding the two together or equivalently using the trapezoidal quadrature rule we get

$$(2M+k_lA)\xi=(2M-k_lA)\xi_{l-1}+k_l(b_l+b_{l-1} \tag{27}$$

however since the load vector is time independent and by dividing equ.25 by 2 we end up with

$$(M+\frac{k_l}{2}A)\xi_l=(M-\frac{k_l}{2}A)\xi_{l-1}+k_lb_l) \tag{28}$$

which is the crank-nicolson method. This then is another linear system of equations which can be solved in e.g. matlab.

3 Results

In the following section results from the modelling are presenteed. The subsections with A is from the 1D case, those with a B are from the 2D case and C corresponds to the 3D problem solved in fenics. For the 3D part the code was developed in conjunction with Sara Hedar.

3.1 A.1

Since the accuracy in the solution depends on the mesh it becomes interesting to try to refine it in a clever way and not "just add more" nodes. The information required for this can be obtained from a posteriori error estimates. A a posteriori error estimates is computed from the solution u_h to the pde. The first question was to derive

$$||(u-u_h)'||^2_{L^2(I)}\leq C\sum_{i=1}^{n}\eta_i^2 \tag{29}$$

in which $\eta_i = h_i||f +''_h||_{L^2(I_i)}$ and c a constant. Starting with calling $u - u_h = e$ and not writing $L^2(I)$ for simplicity we get

$$||e'||^2 = \int_{-1}^{1} e'^2 dx. \tag{30}$$

Now using Galerkin orthogonality $\int_0^1 (u - u_h)'v'dx = 0, \forall v \in V_{h,0}$, we can subtract an interpolant

$$\int_{-1}^{1} e'(e - \pi e)'dx$$

in which $\pi e \in V_h$. Since we're working in a discrete space from node 1 to node n we'll calculate the integral on the subintervals x_{i-1} to x_i

$$\sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} e'(e - \pi e)'dx \tag{31}$$

$$= \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} (-e'')(e - \pi e)dx + [e'(e - \pi e]_{x_{i-1}}^{x_i} \tag{32}$$

$$= \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} (-e'')(e - \pi e)dx. \tag{33}$$

By seeing $-e'' = -(u - u_h)'' = -u'' - u''_h$ and remembering that $-u'' = f/\alpha$ we get $-e'' = f/\alpha + u''_h$. To continue we'll use Cauchy-Schwartz inequality.

$$= \alpha \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} (f + \alpha u''_h)(e - \pi e) \tag{34}$$

$$\leq \alpha \sum_{i=1}^{n} ||f + \alpha u''_h||||e - \pi e|| \tag{35}$$

$$\leq \alpha \sum_{i=1}^{n} ||f + \alpha u''_h||Ch_i||e'|| \tag{36}$$

where Ch_i is a standard interpolation error, and α a constant

$$= C \sum_{i=1}^{n} h_i||f + \alpha u''_h||||e'|| \tag{37}$$

$$\leq C(\sum_{i=1}^{n} h_i^2 ||f + \alpha u_h''||^2)^{\frac{1}{2}} (\sum_{i=1}^{n} ||e'||^2)^{\frac{1}{2}} \tag{38}$$

$$= C(\sum_{i=1}^{n} h_i^2 ||f + \alpha u_h''||^2)^{\frac{1}{2}} ||e'|| \tag{39}$$

by dividing with $||e'||$ we've derived the a posteriori error.

3.2 A.2

To solve the 1D problem $-\alpha u'' = f$ with f being either ρ if $|R - |x|| \leq r$ or else 0. One could derive a stiffness matrix, A, and a load vector, B. However another way is to replace $-u''$ with the discrete laplacian $\Delta_h u_h$. When doing this one gets a mass matrix, M, in addition to the stiffness matrix and load vector. The nodal values of M can then be calculated as $\zeta = -M^{-1}A\xi$.

However as discussed in A.1 the solution will depend on the mesh, how many intervals, since a finer mesh can model the changes more accurately. This is an important fact when dealing with impulse signals for which we might need a very fine mesh to model the impulse accurately. If we where to keep the mesh uniform we would end up with large matrices which we tries to solve. An improvement to the naive uniform refinement is to use an adaptive mesh.

An adaptive mesh is a mesh in which we refine the mesh size on limited intervals where we measure a large error. This is done by splitting these intervals but leaving the others as before.
This splitting can be done by first calculating η_i for each interval calculating f with a numerical integration. The refinement will then happen at the intervals fulfilling

$$\eta_i > \lambda \max_{i=1,2,\ldots,n} \eta_i. \tag{40}$$

Figure 1 below shows four subplots depicting the finite element solution u_h the error, residual, and mesh distribution.

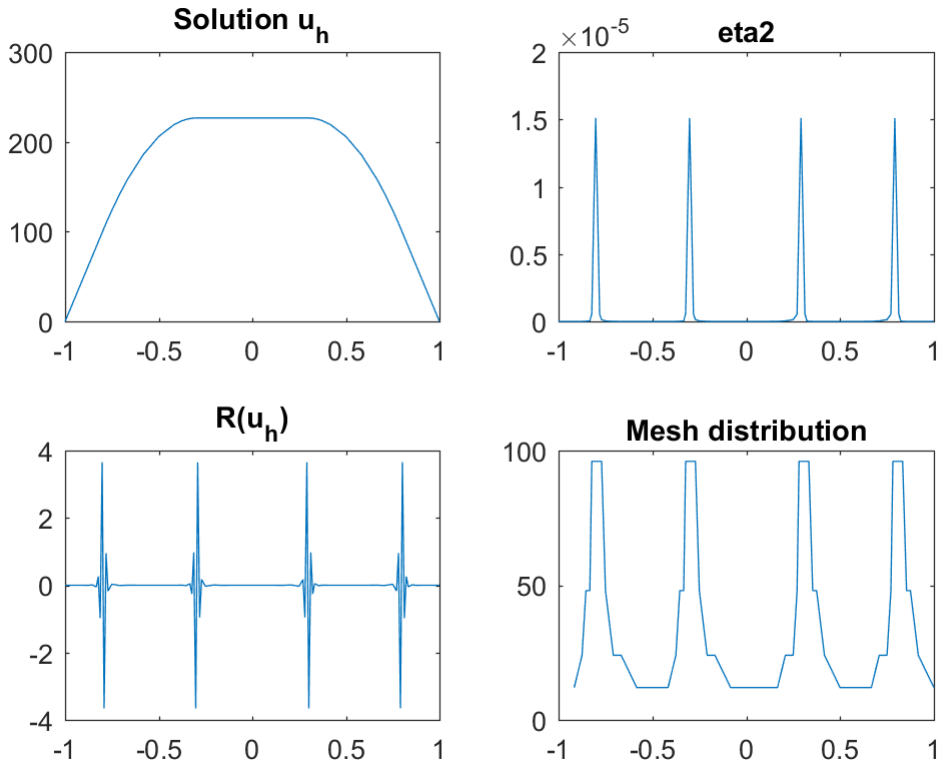

Figure 1. Finite element solution, η^2, the residual, and mesh distribution.

3.3 B.1

The pde in 2D with boundary condition as shown in equ. 41 and 42

$$-\Delta u(x) = f(x), \quad x \in B \tag{41}$$

$$u(x) = u_{exact}(x), \quad x \in \partial B \tag{42}$$

is an example of a problem with an inhomogenous dirichlet boundary condition and can be used to show the rate of convergence and how to compute it. If we start with defining a trial, since the b.c. is inhomogenous, space V_e s.t.

$$V_e = \{v : ||v|| + ||\nabla v|| < \infty, v|_{\partial B} = u_{exact}\}. \tag{43}$$

Multiplying equ.41. with a test function $v \in V_0$ we get

$$\int_B fv dx = -\int_B \Delta uv dx \tag{44}$$

$$= \int_B \nabla \cdot u \nabla v dx - \int_{\partial B} n \nabla u v dx \tag{45}$$

$$= \int_B \nabla u \cdot \nabla v dx. \tag{46}$$

If we now introduce a subspace

$$V_{h,g} = \{v \in V_h : v|_{\partial B} = u_{exact}\} \tag{47}$$

the problem is to find u_h $\in$V$_{h,e}$ s.t

$$\int_B \nabla u_h \cdot \nabla v dx = \int_B f v dx, \quad \forall v \in V_{h,0}. \tag{48}$$

To solve for u_h we write it as $u_h = u_{h,0} + u_{h,e}$. Since $u_{h,g}$ is known, it's the boundary condition, we want to determine $u_{h,0}$. This can be done as

$$\int_B \nabla u_{h,0} \cdot \nabla v dx = \int_B f v dx - \int_B \nabla u_{h,e} \cdot \nabla v dx, \quad \forall v \in Y_{h,0} \tag{49}$$

To implement this we'll start with assuming that we have n_p nodes of which the first n_i are interior nodes while $n_b = n_p - n_i$ are the boundary nodes. If A and b are n_p x n_p and n_p x 1 respectively a linear system can be written as

$$\begin{bmatrix} A_{0,0} & A_{0,e} \\ 0 & I \end{bmatrix} \begin{bmatrix} \xi_0 \\ \xi_e \end{bmatrix} = \begin{bmatrix} b_0 \\ u_{exact} \end{bmatrix} \tag{50}$$

which can be written as

$$A_{0,0}\xi_0 = b_0 - A_0\xi_e = b_0 - A_{0,e}u_{exact} \tag{51}$$

where $A_{0,0}$ is n_ixn_i, $A_{0,e}$ n_ixn_b, b_0 the first n_i entries of b. This is then a linear system of equations which can be solved in matlab. The resulting plots when choosing a mesh with edge lengths of$\frac{1}{2}$ and $\frac{1}{32}$ and $u_{exact} = sin(2\pi x_1)sin(2\pi x_2)$ is shown below in figure 2.

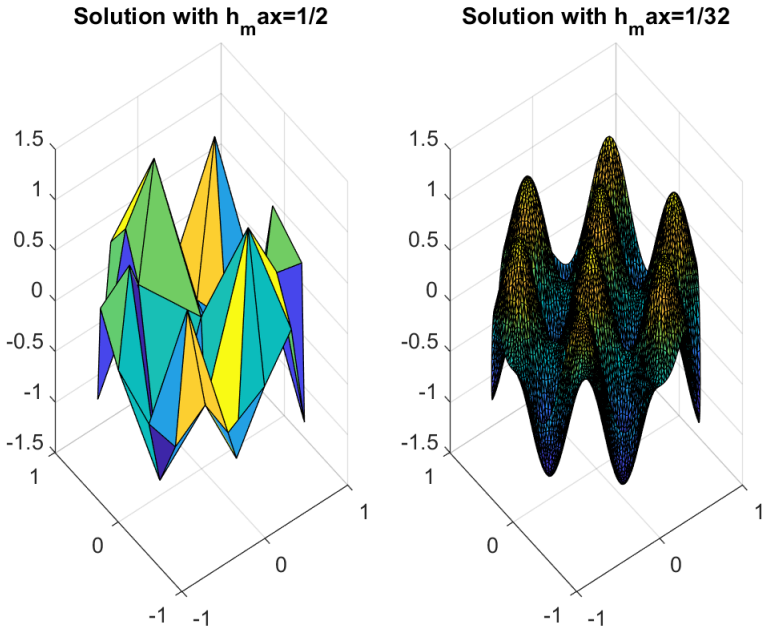

Figure 2. Solution on meshs with h_{max} as $\frac{1}{2}$ and $\frac{1}{32}$

by calculating the energy norm of the error as

$$||u - u_h||_E = (\int_B (\nabla u - \nabla u_h)(\nabla u - \nabla u_h) dx)^{\frac{1}{2}} \tag{52}$$

which in matlab can be calculated with EnE=sqrt($(u - u_h)' * A * (u - u_h)$). Plotting the error vs mesh size in a loglog plot as shown in figure 3 can help us visualize the convergence rate. With the matlab function polyfit we can fit a straight line to our data points and return the slope and where it crosses the y-axis. IN this case the slope, convergence rate, becomes 1.4089.

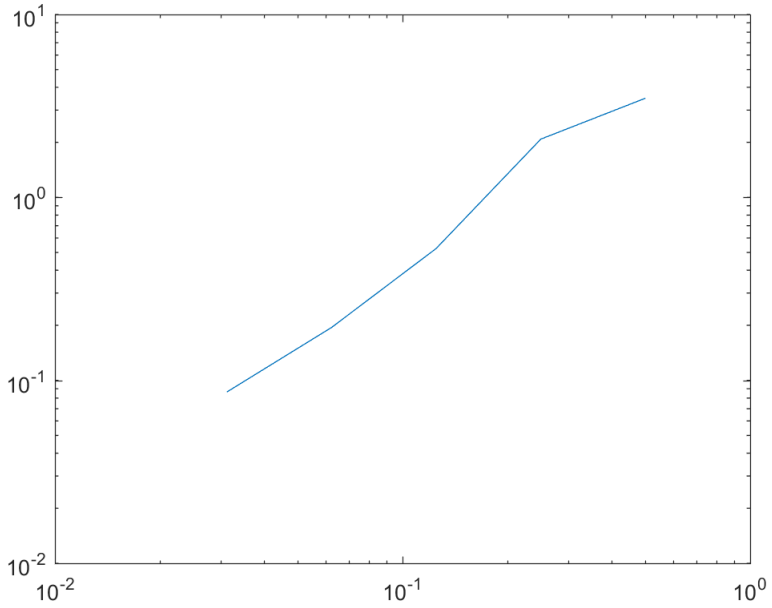

Figure 3. loglog plot of error in energy norm and h_{max}

3.4 B.2 and B.3

To solve the problem presented in the theory section equ.20-22 we solve the system where M,A, and b are the known matrices ξ_l the solution ξ_{l-1} the known solution in the previous time step and k_l the time step size,$t_i - y_{i-1}$. Figure 4 shows the solution at time 0 and time $t = 30$ with $h_{max} = \frac{1}{20}$ where h_{max} is the edge length.

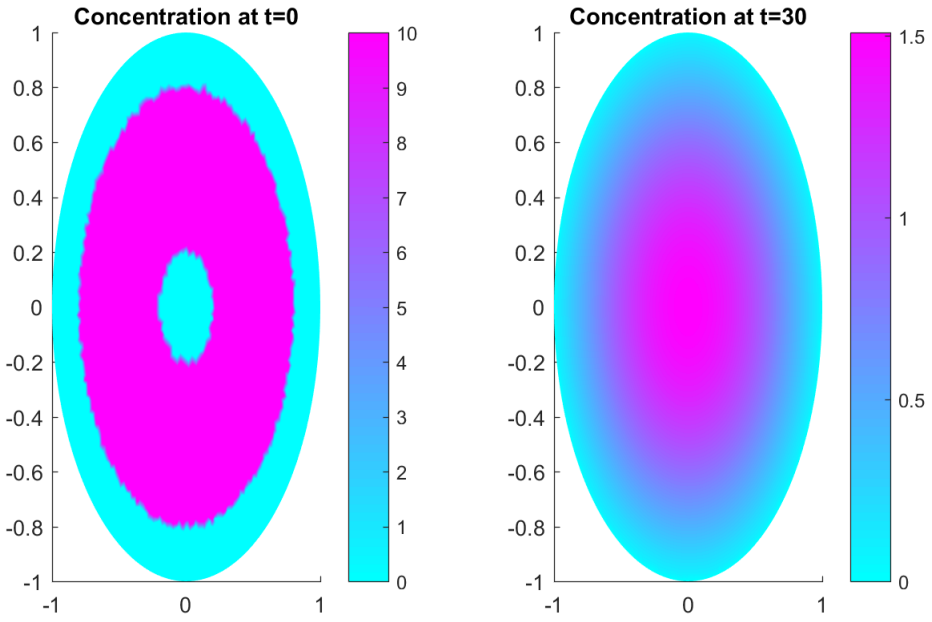

Figure 4. Solution to the time dependant problem at $t = 0$ and $t = 30$

Finally since this problem is a diffusion problem one can calculate the mass loss with the quadrature rule as

$$\int_K g(x)dx \approx \frac{|K|}{3} \sum_{i=1}^{3} g(N_i). \tag{53}$$

Figure 5 shows the emitted hormone vs time for two mesh sizes $h_{max} = \frac{1}{5}$ and $h_{max} = \frac{1}{20}$

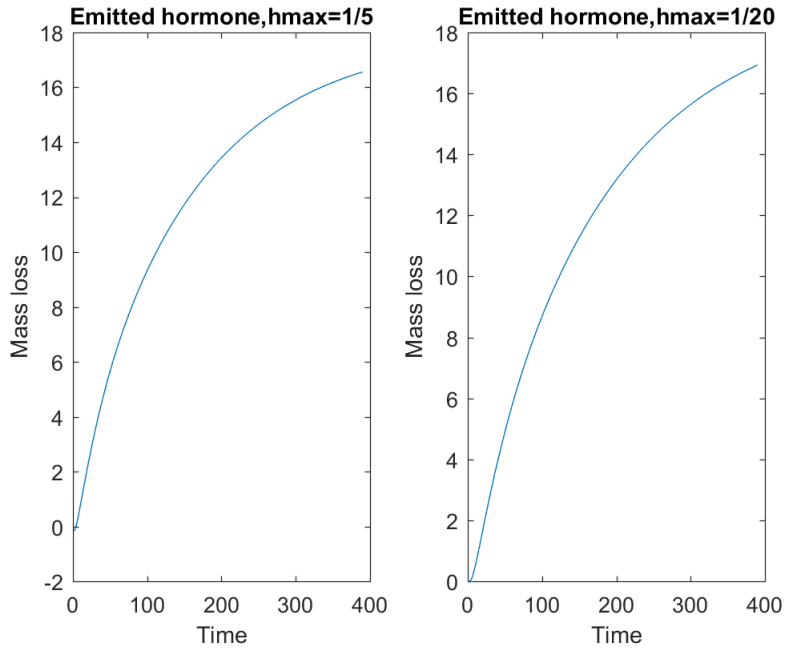

Figure 5. Mass loss as a function of time for mesh with $h_{max} = \frac{1}{5}$ and $h_{max} = \frac{1}{20}$

3.5 C.1

Translating the code from matlab to python/fenics, solving the 2D and 3D problem, where the 3D is solved in the same way as the 2D, and then plotting the solution for time $t = 0$ and $t = 20$ is shown in figure 6 and 7 below for the 2D case while the 3D one is shown in figure 8 and 9.

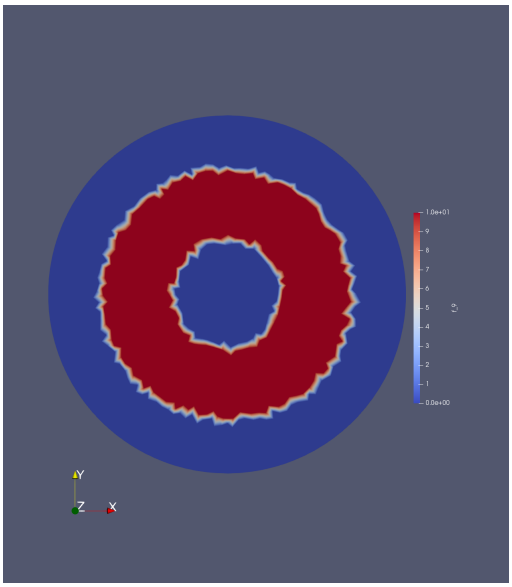

Figure 6. 2D problem at t=0

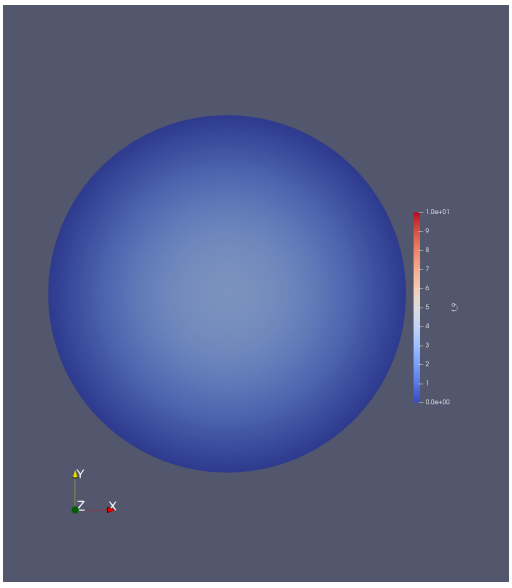

Figure 7. 2D problem at t=20

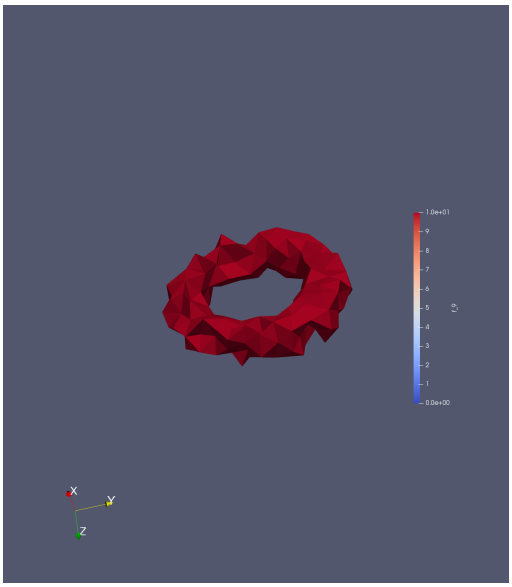

Figure 8. 3D problem at t=0

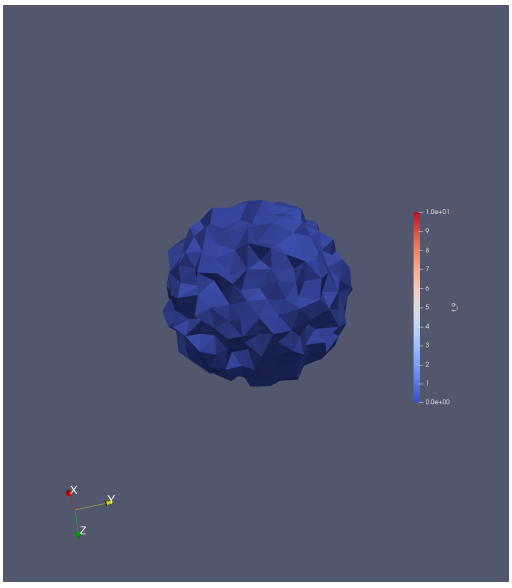

Figure 9. 3D problem at t=20

As we can see the solution for the 2D problem resembles the solution from matlab. The 3D solution looks spiky at the start but that's due to the mesh. We can also see that at the end both solutions have decreased by the same amount from 10 to 1.5.

3.6 C.2

When instead solving the 3D problem to $T = 50$ and varying ρ as 10, 20, and 40, keeping $R = 0.5$ and $r = 0.2$, with a time step of 0.1 and plotting the mass loss as shown in figure 10 below we can see that a higher ρ gives a quicker initial mass loss corresponding to a quicker diffusion, however the appearance of the curves are the same. This makes sense since a higher concentration should mean that, when the other parameters are the same, it will diffuse quicker.

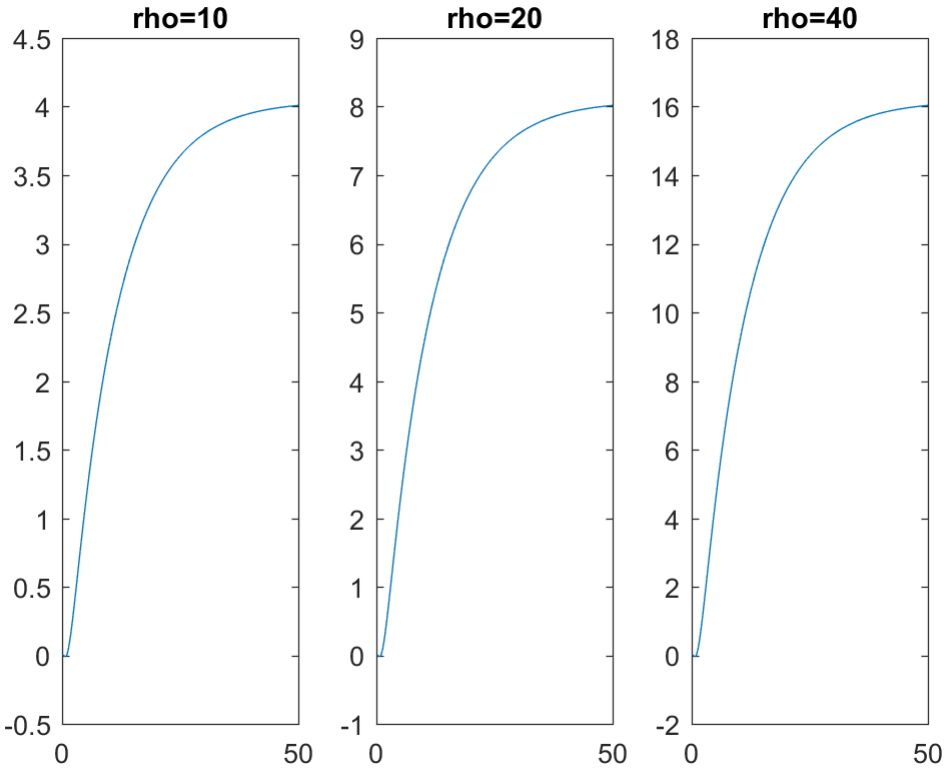

Figure 10. Emitted hormones for varying ρ

3.7 C.3

The final problem was to find the optimal values for ρ, R, and r by solving the optimization problem

$$\min_{\rho,R,r} F(\rho, R, r) = \sum_{i=0}^{2} (M(T_i) - M_i)^2 \tag{54}$$

where $M(T_i)$ is calculated as in c.2 and the values for M_i at the times {5,7,30} was given as {10,15,30}. Further the constraints $R \leq 1$ and $r \leq R$ was to be respected. To solve this problem the optimization package for python was used. The optimization algorithm Nelder-Mead with the initial values as [40,0.65, 0.22] was tested on the finer of the two meshes. Doing this the returned parameters became $\rho = 42.84661778$,$R = 0.5140653$, and $r = 0.27563518$, with a final function value, defined as $\sum_{i=1}^{3}(M(t_i) - M_i)^2$ of 0.55. Plotting the mass loss with these parameters and a time step of 0.1 are shown below in figure 11 together with the three target points. These are then one set of parameters which solves the minimization problem however since we can't say for sure if this is a local minimum we can't know if they are the optimal parameters.

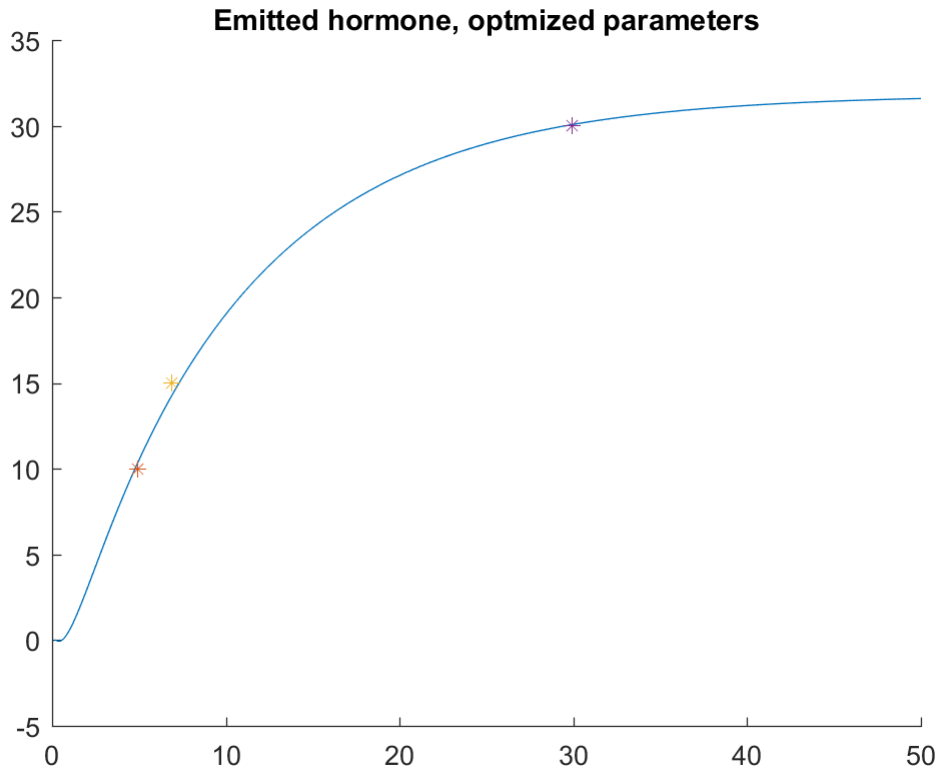

Figure 11. Emitted hormone with optimized parameters and target values

4 References

[1] Larsson Mats G., Bengzon Fredrik. The Finite Element Method: Theory, Implementation, and Practice, Springer, 2010

5 Appendix

Matlab code: %1D problem A.2

```
clear all
%Problem definition
R = 0.5;
roh = 10;
r = 0.3;
alpha = 0.01
a = -1;
b = 1;

N = 12; %Start number of nodes
nN = 1.e+4; %Maximum allowed nodes
h = abs(a-b)/N;
x = [a:h:b];
eta2 = 1; %Initialize eta2 > tol
tol = 1.e-3;

while sum(eta2) >tol || length(x)> nN

    %Mass, Load, Stiff matrix, and forcing function
    Mass = zeros(length(x),length(x));
    Load = zeros(length(x),1);
    Stiff = zeros(length(x),length(x));
    f = Load;
    %Populate function vector

    for i = 1:length(x)-1
        if (abs(x(i))<=0.8&&0.3<=abs(x(i)))
                f(i) = roh;
            else
                f(i) = 0;
        end
    end

    %Assemble matrices
    for i = 1:length(x)-1
        h = x(i+1)-x(i);
        n = [i i+1];
        Stiff(n,n) = Stiff(n,n)+[1 -1; -1 1]/h;
        Load(n) = Load(n)+[f(i);f(i+1)]*(h/2);
        Mass(n,n) = Mass(n,n)+[2 1; 1 2]*h/6;
    end
```

```
    %Boundary condition

    Stiff = alpha*Stiff;
    Stiff(1,1) = 1.e+6;
    Stiff(length(x),length(x)) = 1.e+6;
    Load(1) = 0;
    Load(length(x)) = 0;

    %Solve
    xi = Stiff\Load;
    zeta = -Mass\Stiff*xi;
    px = x; %Store current x vector of node coordinates to plot

    %Mesh refinement
    eta2 = zeros(length(x),1);
    res = zeros(length(x),1);
    res = f+zeta; %Residual

    for i = 1:length(x)-1
        h = x(i+1)-x(i);
        f1 = f(i)+zeta(i);
        f2 = f(i+1)+zeta(i+1);
        eta2(i) = h^2*(((f1^2+f2^2)*h/2)); %Trapezoid integration

    end

    lamda = 0.5; %0 uniform refinement, 1 no refinement
    for i = 1:length(eta2)
        if eta2(i) > lamda*max(eta2) %if large then refine
            x = [x (x(i+1)+x(i))/2];
        end
    end
    x = sort(x);
end
figure
subplot(2,2,1)
plot(px,xi)
title('Solution u_h')
subplot(2,2,2)
plot(px,eta2)
title('eta2')
subplot(2,2,3)
plot(px,res)
title('R(u_h)')
subplot(2,2,4)
plot(px(2:end),1./diff(px))
```

```
title('Mesh distribution')
```

%2D time independent problem, B.1

```
clear all
geometry = @circleg;
hmax = 1/16;
[p,e,t] = initmesh(geometry, 'hmax', hmax);
f = @(x1,x2) 8*(pi^2)*sin(2*pi*x1)*sin(2*pi*x2);
np = size(p,2);

for i = 1:length(p)
    u_exact(i) = sin(2*pi*p(1,i))*sin(2*pi*p(2,i));
end


Stiff = AssembleStiffness2D(p,t);
Load = AssembleLoad2D(p,t,f);
boundaryNode = unique([e(1,:) e(2,:)]);
interiorNode = setdiff(1:length(p),boundaryNode);
Load = Load(interiorNode)-Stiff(interiorNode,boundaryNode)*u_exact(1:length(e))'
u_h(interiorNode) = Stiff(interiorNode,interiorNode)\Load;
u_h(boundaryNode) = u_exact(1:length(e));
err = u_exact-u_h;
EnE = sqrt(err*Stiff*err');
trisurf(t(1:3,:)',p(1,:)',p(2,:)',u_h)
```

%2D time dependent problem B.2

```
%B.3
clear all
close all
t0 = 0.1;
L = 29;
time =  linspace(0,t0,L+1);

a = 0.01;
geometry = @circleg;
hmax = 1/32;
[p,e,t] = initmesh(geometry, 'hmax', hmax);
I = eye(length(p));
np = size(p,2);
nt = size(t,2);
roh = 10;
r = 0.3;
R = 0.5;
f = 0;
bc = 0;
```

```
Mass = AssembleMass2D(p,t);
Stiff = a*AssembleStiffness2D(p,t);
Load = zeros(size(Stiff,1),1);

bN = unique([e(1,:) e(2,:)]);
iN = setdiff(1:length(p),bN);
for i = 1:length(p)
    if abs(R-sqrt(p(1,i)^2+p(2,i)^2)) <= r
        xi(i) = roh;
    else
        xi(i) = 0;
    end
end
xi = xi';
u_init = xi;
for l = 1:L
    k=time(l+1)-time(l);
    xi = (Mass+t0/2*Stiff)\((Mass-t0/2*Stiff)*xi+(t0*Load/2));
    xi(bN) = 0;
    pdeplot(p,e,t,'XYData',xi)
    pause(0.1);
    ml = 0;
    for i = 1:length(t)
        loc2glb = t(1:3,i);
        x1 = p(1,loc2glb);
        x2 = p(2,loc2glb);
        area = polyarea(x1,x2);
        for i = 1:3
           m(i) = u_init(loc2glb(i)) - xi(loc2glb(i));
        end
        ml = ml+(sum(m))/3*area;
    end
    m_loss(l) = ml;
end
plot([1:L],m_loss)
title('Amount of emitted hormone')
ylabel('Mass loss')
xlabel('Time')
```

%Helper functions

```
function [ area,b,c ] = Gradient(x,y)
%UNTITLED7 Summary of this function goes here
%   Detailed explanation goes here
area = polyarea(x,y);
b = [y(2)-y(3); y(3)-y(1); y(1)-y(2)]/2/area;
```

```
c = [x(3)-x(2); x(1)-x(3); x(2)-x(1)]/2/area;
end

function [ B ] = AssembleLoad2D( p,t,f )
%UNTITLED5 Summary of this function goes here
%   Detailed explanation goes here
np = size(p,2);
nt = size(t,2);
B = zeros(np,1);
for i = 1:nt
    loc2glb = t(1:3,i);
    x1 = p(1,loc2glb);
    x2 = p(2,loc2glb);
    area = polyarea(x1,x2);
    Bi = [f(x1(1),x2(1)); f(x1(2),x2(2)); f(x1(3),x2(3))]*area/3;
    B(loc2glb) = B(loc2glb)+Bi;

end
end

function [M] = AssembleMass2D(p,t)
%UNTITLED4 Summary of this function goes here
%   Detailed explanation goes here
np = size(p,2);
nt = size(t,2);
M = sparse(np,np);
for i = 1:nt
    loc2glb = t(1:3,i);
    x1 = p(1,loc2glb);
    x2 = p(2,loc2glb);
    area = polyarea(x1,x2);
    Mi =  [2 1 1; 1 2 1; 1 1 2]*area/12;
    M(loc2glb,loc2glb) = M(loc2glb,loc2glb)+Mi;
end
end

function [A] = AssembleStiffness2D(p,t)
%UNTITLED6 Summary of this function goes here
%   Detailed explanation goes here
np = size(p,2);
nt = size(t,2);
A = sparse(np,np);
for i = 1:nt
    loc2glb = t(1:3,i);
    x1 = p(1,loc2glb);
    x2 = p(2,loc2glb);
```

```
    [area,b,c] = Gradient(x1,x2);
       Ai =(b*b'+c*c')*area;
    A(loc2glb,loc2glb) = A(loc2glb,loc2glb)+Ai;
end
end
```

%3D problem C.1-2

```
from dolfin import *
import numpy as np
mesh = Mesh("sphere2.xml")
Q = FunctionSpace(mesh, "CG", 1)
u = TrialFunction(Q)
v = TestFunction(Q)

class DirichletBoundary(SubDomain):
    def inside (self, x, boundary):
        return boundary

# Optimal parameters
# rho = 42.84661778
# R =  0.5140653
# r = 0.27563518
# T = 50

rho = 10
R = 0.5
r = 0.2
T = 20
alpha = 0.01
theta = 0.5
h = mesh.hmin ()
dt = h
mass = []

# +x[2]*x[2] 3D term
data = Expression("pow(R-sqrt(x[0]*x[0]+x[1]*x[1]), 2)+x[2]*x[2]<=r*r?rho:0.0",
u0 = Function(Q)
u0 = interpolate(data, Q)
u_init = Function(Q)
u_init = interpolate(data, Q)

g = Constant(0.0)
bc = DirichletBC(Q, g, DirichletBoundary())

a = inner(u, v)*dx + dt * theta * alpha * inner(grad(u), grad(v))*dx
L = inner(u0, v)*dx - dt*(1.0 - theta)*alpha*inner(grad(u0), grad(v))*dx
```

```
A = assemble(a)
b = assemble(L)
bc.apply(A, b)

sol = Function(Q)
M = (u_init-u0)*dx
t = 0.0

file = File("Resultc1/Sphere.pvd")
while t<T:
    file << u0
    assemble(M)
    mass.append(assemble(M))
    solve(A, sol.vector(), b)
    u0.assign(sol)
    b = assemble(L)
    bc.apply(b)
    t += dt
```

%Optimization C.3

```
from dolfin import *
import scipy.optimize as optimize
from scipy.optimize import minimize
mesh = Mesh("sphere2.xml")
Q = FunctionSpace(mesh, "CG", 1)
u = TrialFunction(Q)
v = TestFunction(Q)

alpha = 0.01
theta = 0.5
dt = 0.1
rho = 40
R = 0.5
r = 0.2
T = 50
data = [rho, R, r]

class DirichletBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary


def func(data):
    print(data)
    rho = data[0]
    R = data[1]
```

```
    r = data[2]
    indata = Expression("pow(R-sqrt(x[0]*x[0] + x[1]*x[1]), 2)+x[2]*x[2] <= r*r?
    u0 = Function(Q)
    u0 = interpolate(indata, Q)
    u_init = Function(Q)
    u_init = interpolate(indata, Q)

    g = Constant(0.0)
    bc = DirichletBC(Q, g, DirichletBoundary())

    a = inner(u, v)*dx+dt*theta*alpha*inner(grad(u), grad(v))*dx
    L = inner(u0, v)*dx-dt*(1.0 - theta)*alpha*inner(grad(u0), grad(v))*dx
    M = (u_init-u0)*dx

    A = assemble(a)
    b = assemble(L)
    bc.apply(A, b)

    sol = Function(Q)
    t = dt
    mass = []
    while t < T:
        mass.append(assemble(M))
        solve(A, sol.vector(), b)
        u0.assign(sol)
        b = assemble(L)
        bc.apply(b)
        t += dt
    F = (mass[5*10-1]-10)**2+(mass[7*10-1]-15)**2+(mass[30*10-1]-30)**2
    print(F)
    return F

res = minimize(func, [40, 0.65, 0.22], method='Nelder-Mead', tol=1e-3)
print(res)
```