Lab Assignment

TU Delft, Faculty of Electrical Engineering, Mathematics and Computer Science,
Minor Computational Science and Engineering

Modelling Tumor Growth in the Brain

Authors:
Kaj Dockx
(4586425)
Matheus Coelho Silva
(4537823)

Supervisor:
H. Kraaijevanger

Delft
February 20, 2019

Contents

1 Introduction

This paper deals with the modelling of tumor growth in a brain using numerical methods. In Section 2, the model equations and initial and boundary conditions are discussed. These equations describe the growth and diffusion of the tumor cells in the brain through the processes of cell mitosis and proliferation. In Section 3, the discretization of the problem through a cell-centered finite-volume method is discussed. The time-stepping method is also covered. Lastly, in Section 4 the numerical results of the simulation are presented and discussed in detail. The code written in Python is presented in full in the Appendix.

This report was completed as part of the course Numerical Methods for Differential Equations of the minor Computational Science and Engineering from the Delft University of Technology.

2 Model Equations

The goal of this exercise is to model tumor growth in a brain. The brain is modelled in xy-space as a flat 2D disk with grey and white matter regions as seen in Figure 1 of the prompt. The simplified model used here to describe the growth of a brain tumor consists of a partial differential equation, an initial condition, and a boundary condition. The partial differential equation is given as follows:

$$\frac{\partial c}{\partial t} = D\ div(grad\ c) + \rho c \qquad \text{in } \Omega \tag{1}$$

This equation states that the change in concentration of tumor cells over time at a given point in the domain depends on a diffusion term (which expresses the diffusion of tumor cells from the surrounding area) and a growth term (which expresses exponential growth from the local concentration).

The mitotic rate coefficient ρ depends on the location $\mathbf{x}$ as:

$$\rho(\mathbf{x}) = \begin{cases} \rho_g & \mathbf{x} \in \Omega_g \\ \rho_w & \mathbf{x} \in \Omega_w \end{cases} \tag{2}$$

The parameter values are given by $\rho_w = 0.01\ day^{-1}$, $\rho_g = 0.005\ day^{-1}$, $D = 0.01\ cm^2/day$, and $\hat{c} = 4 * 10^4\ cells/cm^2$.

The brain dimensions are $R_1 = 2\ cm$, $R_2 = 4\ cm$, $R_3 = 6\ cm$, $R_4 = 8\ cm$, $R_5 = 10\ cm$, and $\phi = \pi/8$.

Furthermore, the initial condition is given by the equation below.

$$c(\mathbf{x}, 0) = \hat{c} \cdot exp(-||x||^2) \tag{3}$$

At $t = 0$ there is some concentration of tumor cells everywhere in $\overline{\Omega}$, but the largest concentration is found at the origin (where c = $\hat{c}$). From the origin, the concentration decays exponentially towards the boundary. It seems that this reflects an early stage in the development of the tumor, as the tumor cells are still mostly concentrated in one place and very little has diffused outwards. It can be expected that over time the tumor will spread to a more uniform concentration throughout the brain due to diffusion (cell proliferation). The death of the patient sets in when $c \geq \hat{c}$ over at least 25% of the brain area.

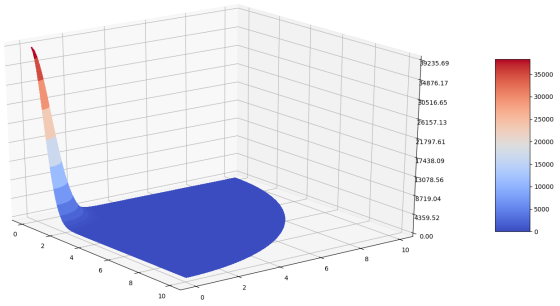


Figure 1: Representation of the tumor at time = 0.

As for the boundary condition, it can be ascertained that no tumor cells diffuse through the skull wall and leave the brain area. In other words, there is no outward diffusion of tumor cells at the boundary. Therefore the following Neumann condition can be used:

$$D\frac{\partial c}{\partial \mathbf{n}} = 0 \tag{4}$$

This model is better suited to the polar coordinate system than the cartesian one; the former allows the domain's boundaries and white/grey matter regions to be more easily defined in terms of r and θ, and the symmetry of the domain to be easily exploited. The domain can easily be converted into polar coordinates (r, θ), where:

$$x = r\cos\theta$$
$$y = r\sin\theta$$

Using the symmetry of the problem, the domain can be reduced to one fourth of the total size through the following choice of periodic boundary conditions:

$$\begin{cases} c(r, \frac{\pi}{2}) = c(r, 0) \\ \frac{\partial c}{\partial \theta}(r, \frac{\pi}{2}) = \frac{\partial c}{\partial \theta}(r, 0) \end{cases} \tag{5}$$

In polar coordinates, the PDE can be reformulated into:

$$\begin{cases} \frac{\partial c}{\partial t} = D\left(\frac{\partial^2 c}{\partial r^2} + \frac{1}{r}\frac{\partial c}{\partial r} + \frac{1}{r^2}\frac{\partial^2 c}{\partial \theta^2}\right) + \rho c & \text{in } \Omega \\ D\frac{\partial c}{\partial r} = 0 & \text{on } \partial\Omega \end{cases} \tag{6}$$

The resulting virtual domain $\Omega_{r\theta}$ in the $r-\theta$ coordinate space is pictured below. The reduced domain spanning only $\left[0, \frac{\pi}{2}\right)$ will be referred to as $\omega_{r\theta}$ in the subsequent discussion.

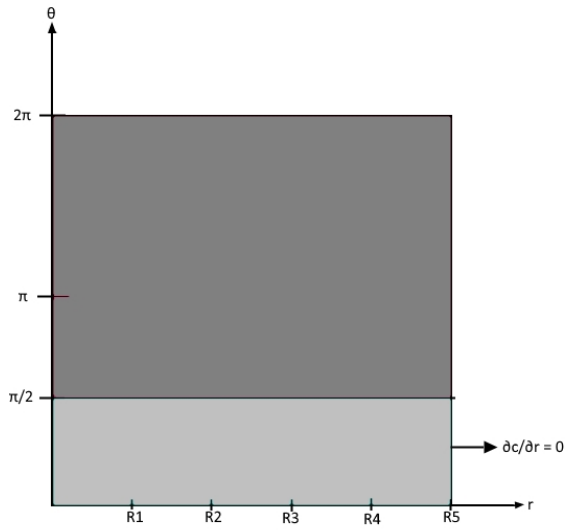


Figure 2: An illustration of the virtual domain $\Omega_{r\theta}$ in the $r-\theta$ coordinate space, with $\omega_{r\theta}$ highlighted in a lighter tone of gray.

3 Numerical Method

In order to solve the problem numerically, Equation 1 must be discretized in both space and time. The method of lines is used, where the PDE is first discretized in space and then a time-stepping algorithm is used to calculate each value of c in the grid for $t > 0$.

3.1 The Spatial Discretization

In order to discretize the the partial differential equation in space, the integral over a single control volume v in the r, θ domain is taken. Note that because the transformation for the x, y domain to the r, θ domain results in an extra term, r, in the integrals. The integral is then rewritten using Gausses Theorem. This all results in the following expression for the integral of the PDE over a single control volume.

$$\begin{aligned} \int_v c' \, r \, d\Omega &= \int_v \left(D \left(\frac{\partial^2 c}{\partial r^2} + \frac{1}{r}\frac{\partial c}{\partial r} + \frac{1}{r^2}\frac{\partial^2 c}{\partial \theta^2} \right) + \rho c \right) r \, d\Omega \\ &= \int_{\partial v} \begin{pmatrix} r \, \partial u/\partial r \\ (1/r) \, \partial u/\partial \theta \end{pmatrix} \cdot \begin{pmatrix} n_r \\ n_\theta \end{pmatrix} d\Gamma + \int_{\partial v} \rho c \, r \, d\Omega \end{aligned} \tag{7}$$

In order to simply this expression, the values of n_r and n_θ are required. Because of the shape of the control volume, these can easily be obtained for each boundary of the control volume. In figure 3 the value of $\mathbf{n}$ is shown for each boundary. As these values differ, the integral containing the inner product is split into 4 the different integrals, one for each boundary. Substituting the values of n_r and n_θ in these separate integral yields the following expression:

$$-\int_{\Gamma_S} \frac{1}{r}\frac{\partial c}{\partial \theta} \, d\Gamma + \int_{\Gamma_E} r\frac{\partial c}{\partial r} \, d\Gamma + \int_{\Gamma_N} \frac{1}{r}\frac{\partial c}{\partial \theta} \, d\Gamma + \int_{\Gamma_W} r\frac{\partial c}{\partial r} \, d\Gamma \tag{8}$$

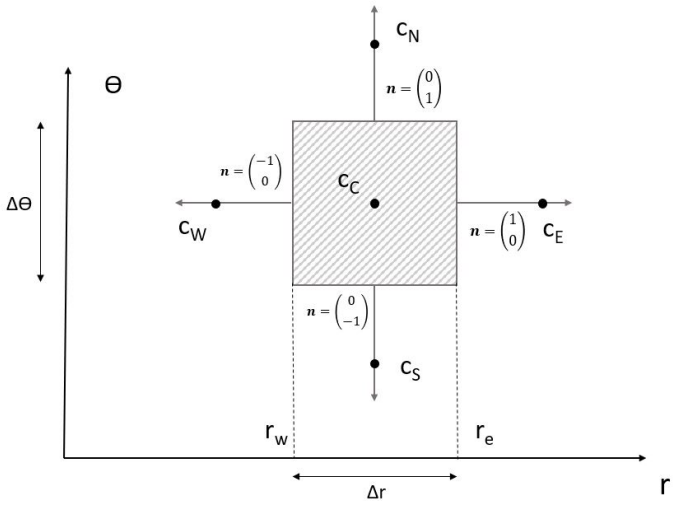


Figure 3: A control volume in the cell-centered Finite Volume discretization of $\Omega_{r\theta}$

Finally the integrals are discretized using the known methods. The spatial discretization of the PDE thus equals:

$$c'_C \, r_C \Delta r \Delta\theta = \frac{1}{r_C}\frac{c_S - c_C}{\Delta\theta}\Delta r + r_e\frac{c_E - c_C}{\Delta r}\Delta\theta + \frac{1}{r_C}\frac{c_N - c_C}{\Delta\theta}\Delta r + r_w\frac{c_W - c_C}{\Delta r}\Delta\theta + r_C\rho_C c_C \Delta r \Delta\theta \tag{9}$$

where r_e and r_w are the values of r at the east and west boundary of the control volume as shown in figure 3. Note that ρ_C either equals ρ_g or ρ_w, depending on whether the central node lies in grey or white brain tissue. However, if the grid node lies exactly on the interface of a gray and white area, another solution is required. In this paper, the choice is made to use the average of ρ_g and ρ_w. The value of ρ_C at an interface thus equals:

$$\rho_C = \frac{\rho_w + \rho_g}{2} \tag{10}$$

3.2 Boundary Conditions

If the side Γ_E of a cell is also the east boundary of the domain $\omega_{r\theta}$, then the term $\int\limits_{\Gamma_E} r\frac{\partial c}{\partial r}\, d\Gamma$ associated with that side in equation 8 would be set to zero. This is because $\frac{\partial c}{\partial r} = 0$ along the boundary. Other terms in the finite volume equation for the boundary cell remain the same. And, since this discretization is cell-centered, the area of the cell also remains the same. The effect of this in equation 9 is that the $r_e\frac{c_E - c_C}{\Delta r}\Delta\theta$ term is neglected when solving the discretized PDE along the eastern boundary of $\omega_{r\theta}$.

If the side Γ_W of a cell is the west boundary of the domain $\omega_{r\theta}$, then the western node relative to that cell is actually the central node of the same cell in the reflected domain spanning $\left[\frac{\pi}{2}, \frac{3\pi}{4}\right)$. Because of the symmetry of Ω which allows it to be split into four sections of angle $\frac{\pi}{2}$ radians, the concentration at this western node will in fact be the same as the concentration at the central node of the cell. That is, $c_W = c_C$ in equation 9.

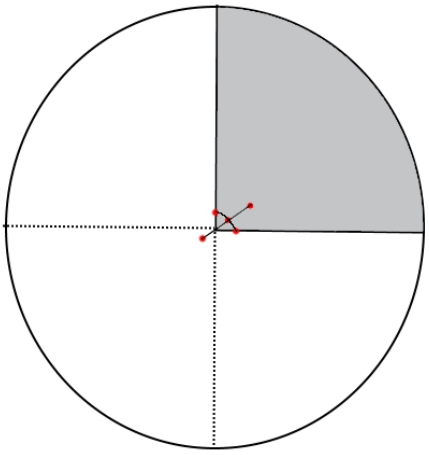


Figure 4: A cell on the west boundary of the reduced domain, shown in the physical x-y space.

Due to the symmetry of Ω, a periodic boundary condition is set at the northern and southern boundaries of $\omega_{r\theta}$. In practice, θ values only go until $\frac{\pi}{2} - \Delta\theta$ because $c(r, \frac{\pi}{2}) = c(r, 0)$. For nodes at $\theta = \frac{\pi}{2} - \Delta\theta$, $c_N = c(r_c, 0)$. And for nodes at $\theta = 0$, $c_S = c(r_c, \frac{\pi}{2} - \Delta\theta)$.

Cells at the corners of $\omega_{r\theta}$ are a combination of a east or west boundary case and a north-south periodicity case. A cell at the northeastern boundary will have its $r_e\frac{c_E - c_C}{\Delta r}\Delta\theta$ term set to zero as well as having $c_N = c(r_c, 0)$. A cell at the southwestern boundary will have its $r_w\frac{c_W - c_C}{\Delta r}\Delta\theta$ term set to zero as well as having $c_S = c(r_c, \frac{\pi}{2} - \Delta\theta)$.

The discretized PDE can furthermore be expressed in Matrix form:

$$M\mathbf{c'} = S\mathbf{c} = (Z + G)\mathbf{c} \tag{11}$$

Where:

$$M = \begin{bmatrix} r_c\Delta r\Delta\theta & 0 & \cdots & 0 \\ 0 & r_c\Delta r\Delta\theta & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & r_c\Delta r\Delta\theta \end{bmatrix}$$

$$G = \begin{bmatrix} \rho r_c\Delta r\Delta\theta & 0 & \cdots & 0 \\ 0 & \rho r_c\Delta r\Delta\theta & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \rho r_c\Delta r\Delta\theta \end{bmatrix}$$

And Z contains the terms of the discretized PDE correponding to the diffusion term (and accounting for boundary conditions).

3.3 Time-Stepping

The problem is integrated in time with the Implicit Euler method. This method was chosen because the matrix $M^{-1}G$, where G is the matrix consisting of only the growth terms in the discretized PDE, contains positive real eigenvalues. The Explicit Euler method would therefore not work, even though it might have been suitable to a problem with only the diffusion term. The Implicit Euler is a very robust and reasonably easy to implement time-stepping algorithm, and was therefore deemed suitable for this problem.

For a general time-dependent problem of the form $\frac{\partial c}{\partial t} = f(x,t)$, the Implicit Euler algorithm can be expressed as:

$$c_{n+1} = c_n + \Delta t f(t_{n+1}, c_{n+1}) \tag{12}$$

And in this specific problem:

$$\mathbf{c'} = M^{-1}(Z+G)\mathbf{c} = (M^{-1}Z + R)\mathbf{c}$$

where R is the matrix with the (position-dependent) values of ρ along its diagonal. Combined with Equation 12, this yields the following relationship:

$$\mathbf{c}_{n+1} = \mathbf{c}_n + \Delta t(M^{-1}Z + R)\mathbf{c}_{n+1} \tag{13}$$

or, explicitly:

$$\mathbf{c}_{n+1} = [(I - \Delta t(M^{-1}Z + R)]^{-1}\mathbf{c}_n \tag{14}$$

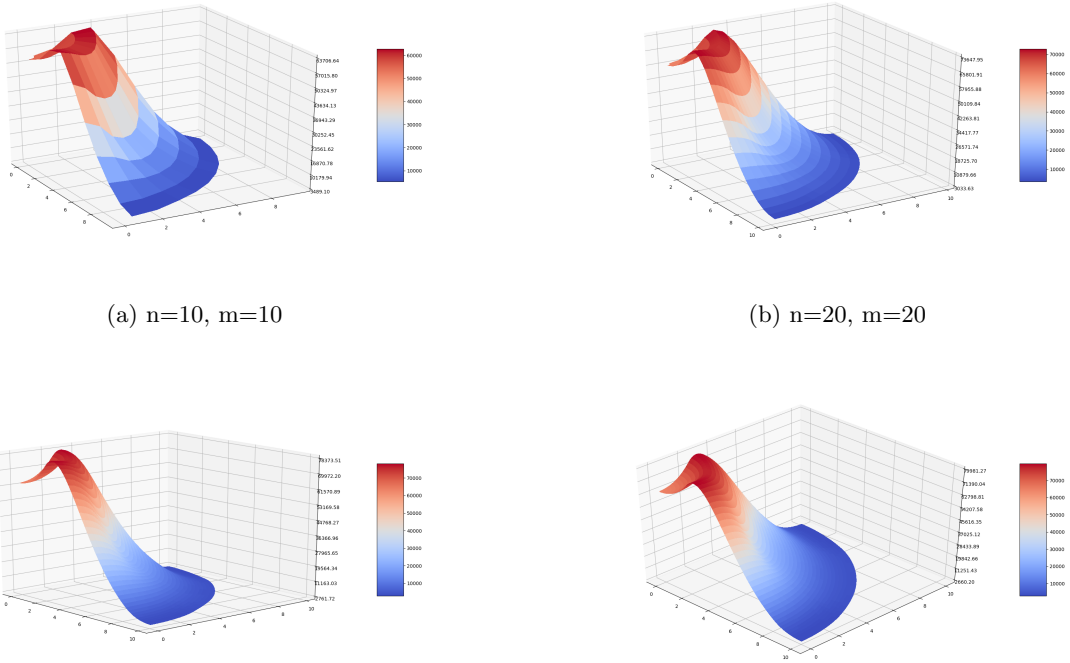
The time step Δt was chosen empirically through experimentation with the code. However, as upper and lower bounds it can be stated that $0 < \Delta t \leq 1/\lambda$ where λ is the largest positive eigenvalue of the matrix $(M^{-1}Z+R)$.

4 Numerical results

Before the results can be plotted and analyzed, the results in r, θ space must be coordinate transformed back to the x,y space. This is important for the visualization of the solution in the physical space.

The death condition is checked at each time interval by summing all the cell areas which have a concentration higher than $\hat{c}$. If this area is larger than 1/16 of the area of the entire domain Ω (1/4 of the truncated domain ω), the patient has passed away.

In order to choose a suitable grid size, plots of the tumor at the time of death are made using different grid sizes. These are shown figure 5. Note that $\Delta r = \frac{10}{n}$ and $\Delta\theta = \frac{\pi}{2m}$.



(a) n=10, m=10

(b) n=20, m=20

(c) n=50, m=50

(d) n=100, m=100

Figure 5: Representation of tumor at time of death with varying grid sizes.

This figure shows how the coarseness of the surface depends on the grid size. The two graphs with the smallest m and n clearly do not suffice to model the tumor correctly as they do not have a high enough resolution. When m and n are at least 50, the surface becomes much more smooth. For the final estimation of the time of death a grid size of 100x100 will be used to guarantee a high resolution.

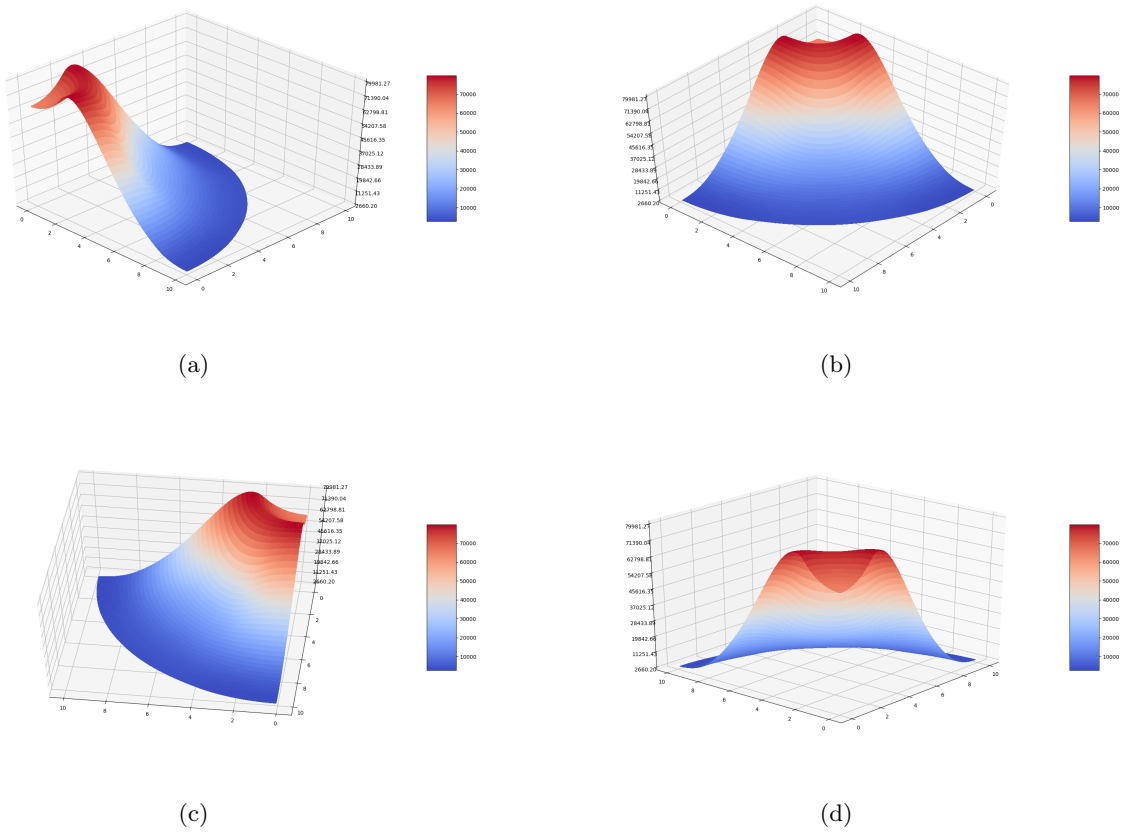
Besides the grid size, the value of Δt will also influence the result. In order to choose a good time step, the time of death is estimated using varying time steps for different grid sizes. These results are shown in table 1:

Table 1: Estimated time of death using varying grid sizes and time steps

n	m	Δt	time of death (days)
10	10	0.5	529.50
10	10	0.05	530.20
10	10	0.01	530.25
20	20	0.5	532.00
20	20	0.1	532.70
20	20	0.05	532.80
50	50	1	526.00
50	50	0.5	527.50
50	50	0.1	528.00
50	50	0.05	528.10

When keeping grid size constant and decreasing Δt, the final time of death eventually converges to a single value. In other words, at a certain point decreasing Δt stops changing the final result significantly. It appears that when using a larger grid size, the final result converges faster. This can be observed in the slope defined as the change in the result over the change in Δt. For example, when using a grid size of 10x10, this slope is 0.1 at a Δt of 0.05. When using a grid size of 50x50, this same slope is already met at a Δt of 0.1. Therefore it is safe to assume that for the final computation with a grid size of 100 by 100, a Δt of 0.1 will yield a slope of at most 0.1.

When running the code with a grid size of 100x100 and a Δt of 0.1, the estimated time of death amounts 526.8 days. The surface plots corresponding to this computation are shown in figure 6.



(a) (b)

(c) (d)

Figure 6: Representation of tumor at time of death using a grid size of 100x100 and a Δt of 0.1

Note that, because of the choice of a truncated domain ω, the tumor is only evaluated and plotted in a quarter of the brain. Due to the symmetry of the full domain Ω and of the initial condition, the same concentration distribution will be seen in the other three quarters of the brain.

5 Conclusion

The best estimated time of death given these initial conditions is 526.8 days. This estimation was calculated using the method of lines with cell-centered FVM spatial discretization and a Backward Euler time-stepping method. The finest parameters used for these methods were a grid size of 100x100 and a time step of 0.1 days. The time step was found to be small enough to allow for convergence. It is possible that a finer grid size would yield more accurate results.

Appendix

```
import numpy as np
import math as math
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter

rhow = 0.01 #day^-1
rhog = 0.005 #day^-1
D = 0.01 #cm^2/day
chat = 4.00e4
R1 = 2.0 #cm
R2 = 4.0 #cm
R3 = 6.0 #cm
R4 = 8.0 #cm
R5 = 10.0 #cm
phi = math.pi/8.0

n = 100 # number r nodes
m = 100 # number theta nodes

deltaR = R5/n #cm
deltaT = (math.pi /2.0)/m #rad
#m = m -1 # eliminate north border ( equals south border )

#we use horizontal numbering
def rowChecker(alpha, n, m):
    rowNumber = m-1-(alpha//n)
    return rowNumber;

def columnChecker(alpha, n, m):
    columnNumber = alpha%n
    return columnNumber;

def colorChecker(rowNumber, columnNumber, deltaR, deltaT, R1, R2, R3, R4, phi):
    r = deltaR*columnNumber;
    theta = deltaT*rowNumber;
    if ((r >= R4) and ((theta <= phi) or (theta >= math.pi/2 - phi))):
        if ((r == R4) or (theta == phi) or (theta == math.pi/2 - phi)):
            rho = (rhow + rhog)/2.0
        else:
            rho = rhog

    elif ((r >= R2) and (r <= R3) and (theta >= phi) and (theta <= math.pi/2 - phi)):
        if ((r==R2) or (r==R3) or (theta==phi) or (theta == math.pi/2 - phi)):
            rho = (rhow + rhog)/2.0
        else:
            rho = rhog

    elif (r <= R1):
        if (r==R1):
            rho = (rhow + rhog)/2.0
        else:
            rho = rhog

    else:
```

```
        rho = rhow

    return rho;

rc = np.zeros(n)
re = np.zeros(n)
rw = np.zeros(n)
M = np.zeros((n*m,n*m))
S = np.zeros((n*m,n*m))

for i in range(0,n):
    rc[i] = deltaR*i + deltaR/2.0
    re[i] = rc[i] + deltaR/2.0
    rw[i] = rc[i] - deltaR/2.0

for alpha in range(0,n*m):
    row = rowChecker(alpha, n, m)
    column = columnChecker(alpha, n, m)
    rho = colorChecker(row, column, deltaR, deltaT, R1, R2, R3, R4, phi)

    # M-matrix
    M[alpha,alpha] = rc[column]*deltaR*deltaT
    # S-matrix

    #internal nodes
    if ((row > 0) and (row < m - 1) and (column > 0) and (column < n - 1)):
        S[alpha, alpha] = rho*rc[column]*deltaR*deltaT - 2.0*D*deltaR/(rc[column]*deltaT) - D*(re[colum
        S[alpha, alpha - 1] = D*rw[column]*deltaT/deltaR #west
        S[alpha, alpha + 1] = D*re[column]*deltaT/deltaR #east
        S[alpha, alpha + n] = D*deltaR/(rc[column]*deltaT) #north
        S[alpha, alpha - n] = D*deltaR/(rc[column]*deltaT) #south
    #east boundary
    elif (column == n-1):
        if (row > 0) and (row < m-1): #not a corner
            S[alpha, alpha] = rho*rc[column]*deltaR*deltaT - 2*D*deltaR/(rc[column]*deltaT) - D*rw[colu
            S[alpha, alpha - 1] = D*rw[column]*deltaT/deltaR
            S[alpha, alpha + n] = D*deltaR/(rc[column]*deltaT)
            S[alpha, alpha - n] = D*deltaR/(rc[column]*deltaT)
        elif (row == 0): # top right corner
            S[alpha, alpha] = rho*rc[column]*deltaR*deltaT - 2*D*deltaR/(rc[column]*deltaT) - D*rw[colu
            S[alpha, alpha - 1] = D*rw[column]*deltaT/deltaR
            S[alpha, n - 1] = D*deltaR/(rc[column]*deltaT)
            S[alpha, alpha - n] = D*deltaR/(rc[column]*deltaT)
        else: #bottom right corner
            S[alpha, alpha] = rho*rc[column]*deltaR*deltaT - 2*D*deltaR/(rc[column]*deltaT) - D*rw[colu
            S[alpha, alpha - 1] = D*rw[column]*deltaT/deltaR
            S[alpha, alpha + n] = D*deltaR/(rc[column]*deltaT)
            S[alpha, n*m-1] = D*deltaR/(rc[column]*deltaT)
    #west boundary
    elif (column == 0):
        if (row > 0) and (row < m-1): #not a corner
            S[alpha, alpha] = rho*rc[column]*deltaR*deltaT - 2.0*D*deltaR/(rc[column]*deltaT) -D*re[col
            S[alpha, alpha + 1] = D*re[column]*deltaT/deltaR
            S[alpha, alpha + n] = D*deltaR/(rc[column]*deltaT)
            S[alpha, alpha - n] = D*deltaR/(rc[column]*deltaT)
        elif (row == 0): #top left corner
             S[alpha, alpha] = rho*rc[column]*deltaR*deltaT - 2*D*deltaR/(rc[column]*deltaT) - D*re[col
             S[alpha, alpha + 1] = D*re[column]*deltaT/deltaR
```

```
            S[alpha, 0] = D*deltaR/(rc[column]*deltaT)
            S[alpha, alpha - n] = D*deltaR/(rc[column]*deltaT)
        else: #bottom left corner
            S[alpha, alpha] = rho*rc[column]*deltaR*deltaT - 2*D*deltaR/(rc[column]*deltaT) - D*re[colu
            S[alpha, alpha + 1] = D*re[column]*deltaT/deltaR
            S[alpha, alpha + n] = D*deltaR/(rc[column]*deltaT)
            S[alpha, (m-1)*n] = D*deltaR/(rc[column]*deltaT)
    #bottom boundary without corners
    elif ((row == m - 1) and (column > 0) and (column < n - 1)):
        S[alpha, alpha] = rho*rc[column]*deltaR*deltaT - 2*D*deltaR/(rc[column]*deltaT) - D*(re[column]
        S[alpha, alpha - 1] = D*rw[column]*deltaT/deltaR
        S[alpha, alpha + 1] = D*re[column]*deltaT/deltaR
        S[alpha, alpha + n] = D*deltaR/(rc[column]*deltaT)
        S[alpha, alpha + (m-1)*n] = D*deltaR/(rc[column]*deltaT)
    #top boundary without corners
    elif ((row == 0) and (column > 0) and (column < n - 1)):
        S[alpha, alpha] = rho*rc[column]*deltaR*deltaT - 2*D*deltaR/(rc[column]*deltaT) - D*(re[column]
        S[alpha, alpha - 1] = D*rw[column]*deltaT/deltaR
        S[alpha, alpha + 1] = D*re[column]*deltaT/deltaR
        S[alpha, alpha - (m-1)*n] = D*deltaR/(rc[column]*deltaT)
        S[alpha, alpha - n] = D*deltaR/(rc[column]*deltaT)

# time intergration Backward Euler



inverseM = np.zeros((n*m,n*m))
I = np.identity(n*m)
A = np.zeros((n*m,n*m))



#Initial condition!
c0 = np.zeros((n*m,1))
c1 = np.zeros((n*m,1))
dt = 0.1;

for alpha in range(0,n*m):
    column = columnChecker(alpha, n, m)
    c0[alpha] = chat*math.exp(-rc[column]*rc[column])

#
for i in range(0,n*m):
    inverseM[i,i] = 1.0/M[i,i]

A = I - dt*(inverseM@S)

eigvals = np.linalg.eigvals(inverseM@S)


area = 0
t = 0.0

while (area < 0.25*0.25*math.pi*R5*R5):
    area = 0
    t = t + dt
    c1 = np.linalg.solve(A,c0)
    c0 = c1
```

```
    for i in range(0,n*m):
        if c1[i] > chat:
            column = columnChecker(i,n,m)
            area = area + rc[column]*deltaR*deltaT;



print(t)

#plot

fig = plt.figure()
ax = fig.gca(projection='3d')

ax1 = np.linspace(deltaR/2.0,R5-deltaR/2.0,n)
ax2 = np.linspace(deltaT/2,4*phi-deltaT/2.0,m)
ax1, ax2 = np.meshgrid(ax1, ax2)
xx = np.zeros((n,m))
yy = np.zeros((n,m))

for i in range(0,n):
    for j in range(0,m):
        xx[i,j] = ax1[i,j]*math.cos(ax2[i,j])
        yy[i,j] = ax1[i,j]*math.sin(ax2[i,j])

ax1 = xx
ax2 = yy

Z = np.zeros((m,n))

for i in range(0,n):
    for j in range(0,m):
        Z[j][i] = c0[i + j*n,0]


# Plot the surface.
surf = ax.plot_surface(ax1, ax2, Z, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)

# Customize the z axis.
#ax.set_zlim(-1.01, 1.01)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))

# Add a color bar which maps values to colors.
fig.colorbar(surf, shrink=0.5, aspect=5)

plt.show()
```