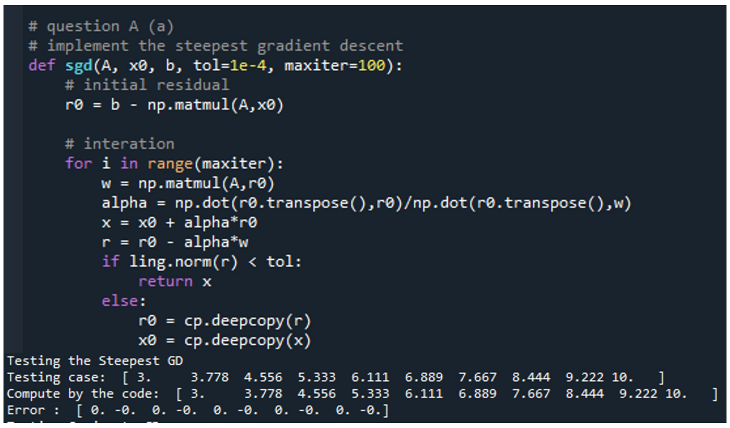
HOMEWORK 2

Boyuan Lu
blu38@wisc.edu

Solution A

Solution (a)

See my figure and my code:



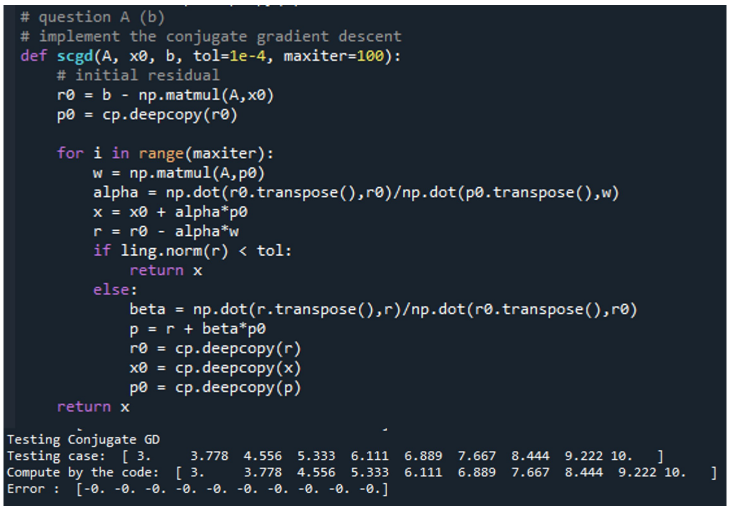
```
# question A (a)
# implement the steepest gradient descent
def sgd(A, x0, b, tol=1e-4, maxiter=100):
    # initial residual
    r0 = b - np.matmul(A,x0)

    # interation
    for i in range(maxiter):
        w = np.matmul(A,r0)
        alpha = np.dot(r0.transpose(),r0)/np.dot(r0.transpose(),w)
        x = x0 + alpha*r0
        r = r0 - alpha*w
        if ling.norm(r) < tol:
            return x
        else:
            r0 = cp.deepcopy(r)
            x0 = cp.deepcopy(x)
Testing the Steepest GD
Testing case:  [ 3.     3.778  4.556  5.333  6.111  6.889  7.667  8.444  9.222 10.   ]
Compute by the code:  [ 3.     3.778  4.556  5.333  6.111  6.889  7.667  8.444  9.222 10.   ]
Error :  [ 0. -0.  0. -0.  0. -0.  0. -0.  0. -0.]
```

Figure 1. implementation of Steepest Gradient Descent and testing case.

Solution (b)

See my figure and my code:



```
# question A (b)
# implement the conjugate gradient descent
def scgd(A, x0, b, tol=1e-4, maxiter=100):
    # initial residual
    r0 = b - np.matmul(A,x0)
    p0 = cp.deepcopy(r0)

    for i in range(maxiter):
        w = np.matmul(A,p0)
        alpha = np.dot(r0.transpose(),r0)/np.dot(p0.transpose(),w)
        x = x0 + alpha*p0
        r = r0 - alpha*w
        if ling.norm(r) < tol:
            return x
        else:
            beta = np.dot(r.transpose(),r)/np.dot(r0.transpose(),r0)
            p = r + beta*p0
            r0 = cp.deepcopy(r)
            x0 = cp.deepcopy(x)
            p0 = cp.deepcopy(p)
    return x
Testing Conjugate GD
Testing case:  [ 3.     3.778  4.556  5.333  6.111  6.889  7.667  8.444  9.222 10.   ]
Compute by the code:  [ 3.     3.778  4.556  5.333  6.111  6.889  7.667  8.444  9.222 10.   ]
Error :  [-0. -0. -0. -0. -0. -0. -0. -0. -0. -0.]
```

Figure 2. implementation of the conjugate gradient descent and test case.

Solution (c)

According to my plot, the convergent rate stays constant no matter the size of matrix as long as the conditioning number remains the same.

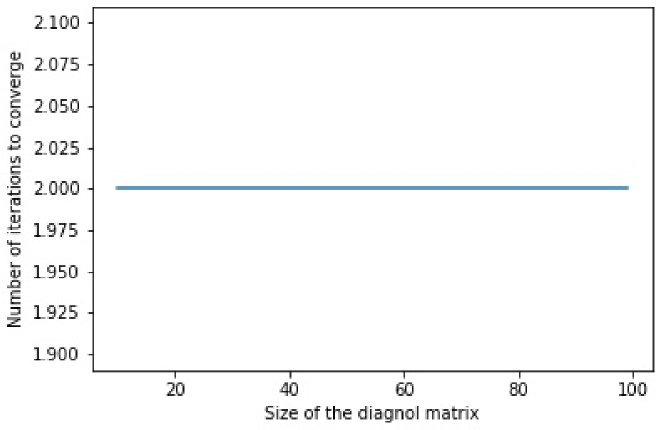


Figure 3. convergent rate of conjugate gradient descent with different matrix size.

If I change the conditioning number in the matrix, the number of iteration will be the consistent as the conditioning number increase. See my plot below:

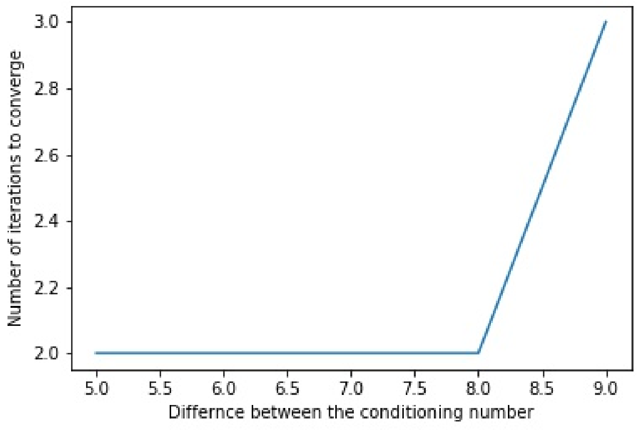


Figure 4. implementation of the conjugate gradient descent and test case.

The x label here is the power of ratio between the largest value to smallest vale. For example, x = 5 means $\frac{\lambda_N}{\lambda_1} = \frac{e5}{e^{-5}}$. Notice that the iteration increase by 1 after different increase 8, I believe this cause by the precision of float and the result should keep consistent. However, it is against what we conclude that convergence rate should be affected by the condition number. The major reason for that is the matrix I calculate is too sparse so that it always converges quickly and you won't discover the impact of condition number. I will state more details of this in next question.

Solution (d)

The matrix with sparse eigenvalue will have less convergence rate then the matrix with uniform eigenvalue (you can check the coding output, I won't give screenshot here.)
Here is the plot (I see the post in piazza, prof thinks the plot is better than presenting in table) for the norm of the residual at each iteration:

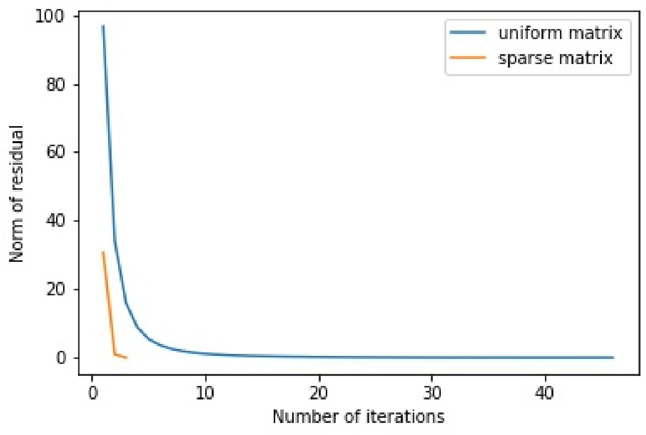
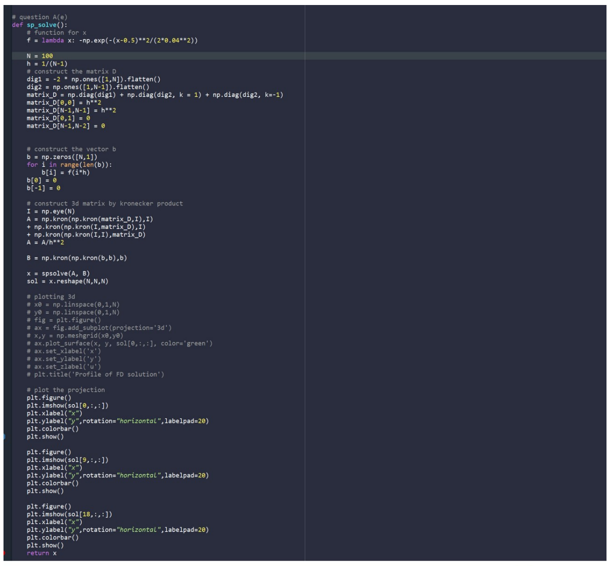


Figure 5. norm of residual at each iteration for steepest conjugate GD.The orange line is convergence curve for really sparse matrix which converge at 3 iteration, while the blue line is for a uniform matrix with 45 iterations of convergence.

Although, the convergence rate is quite different between sparse matrix and uniform matrix. However, both of them are bounded by O(m) since they have the same condition number. The sparse matrix has faster converge rate because it would be easy to do matrix multiplication for a sparse matrix. When doing the SCGD, we need to compute the matrix A with A-conjugate of search direction p at every iteration. $A * p$ can really easy to compute for a large matrix if it is sparse matrix. This explain why in previous question, the convergence rate won't respond to the condition number since the matrix is too sparse and too easy to converge.

Solution (e)

1. the implementation of 3D and output of my solver:



```python
# question A(e)
def sp_solve():
    # function for x
    f = lambda x: -np.exp(-(x-0.5)**2/(2*0.04**2))

    N = 100
    h = 1/(N-1)
    # construct the matrix D
    dig1 = -2 * np.ones([1,N]).flatten()
    dig2 = np.ones([1,N-1]).flatten()
    matrix_D = np.diag(dig1) + np.diag(dig2, k = 1) + np.diag(dig2, k=-1)
    matrix_D[0,0] = h**2
    matrix_D[N-1,N-1] = h**2
    matrix_D[0,1] = 0
    matrix_D[N-1,N-2] = 0


    # construct the vector b
    b = np.zeros([N,1])
    for i in range(len(b)):
        b[i] = f(i*h)
    b[0] = 0
    b[-1] = 0

    # construct 3d matrix by kronecker product
    I = np.eye(N)
    A = np.kron(np.kron(matrix_D,I),I)
    + np.kron(np.kron(I,matrix_D),I)
    + np.kron(np.kron(I,I),matrix_D)
    A = A/h**2

    B = np.kron(np.kron(b,b),b)

    x = spsolve(A, B)
    sol = x.reshape(N,N,N)

    # plotting 3d
    # x0 = np.linspace(0,1,N)
    # y0 = np.linspace(0,1,N)
    # fig = plt.figure()
    # ax = fig.add_subplot(projection='3d')
    # x,y = np.meshgrid(x0,y0)
    # ax.plot_surface(x, y, sol[0,:,:], color='green')
    # ax.set_xlabel('x')
    # ax.set_ylabel('y')
    # ax.set_zlabel('u')
    # plt.title('Profile of FD solution')

    # plot the projection
    plt.figure()
    plt.imshow(sol[0,:,:])
    plt.xlabel("x")
    plt.ylabel("y",rotation="horizontal",labelpad=20)
    plt.colorbar()
    plt.show()

    plt.figure()
    plt.imshow(sol[9,:,:])
    plt.xlabel("x")
    plt.ylabel("y",rotation="horizontal",labelpad=20)
    plt.colorbar()
    plt.show()

    plt.figure()
    plt.imshow(sol[18,:,:])
    plt.xlabel("x")
    plt.ylabel("y",rotation="horizontal",labelpad=20)
    plt.colorbar()
    plt.show()
    return x
```

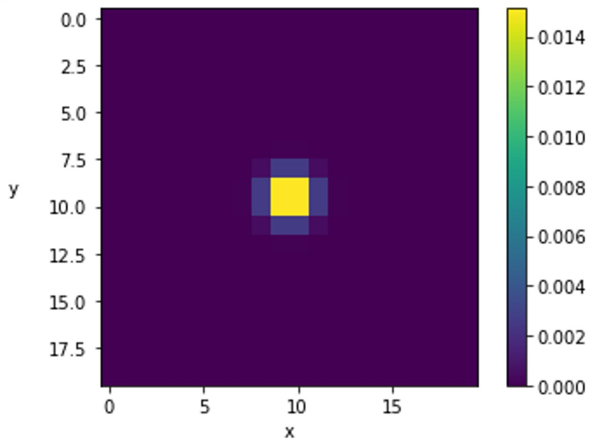
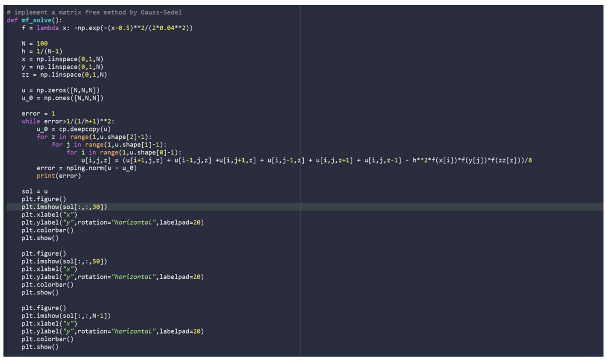


Figure 5. implementation of the spsolve by constructing block matrix using Kronecker product (up picture), and the plot for the cross-section of 3d poisson equation.

Notice that my computation was conducted in grid size of 20, which is really sparse. Therefore, the poisson dot in the center doesn't look like a circle. However, I cannot increase my grid size too much (when I try around 40) as my console report the warning that computation takes too much memory. The reason for it is the way of spsolve work. The spsolve solve the linear equation by LU factorization which costs too much resource from computer if size is too large or matrix is not sparse.

2. the implementation of 3D Laplacian using the iterative method (Gausse-Sadel):



```python
# implement a matrix free method by Gauss-Sedel
def mf_solve():
    f = lambda x: -np.exp(-(x-0.5)**2/(2*0.04**2))

    N = 100
    h = 1/(N-1)
    x = np.linspace(0,1,N)
    y = np.linspace(0,1,N)
    zz = np.linspace(0,1,N)

    u = np.zeros([N,N,N])
    u_0 = np.ones([N,N,N])

    error = 1
    while error>1/(1/h+1)**2:
        u_0 = cp.deepcopy(u)
        for z in range(1,u.shape[2]-1):
            for j in range(1,u.shape[1]-1):
                for i in range(1,u.shape[0]-1):
                    u[i,j,z] = (u[i+1,j,z] + u[i-1,j,z] +u[i,j+1,z] + u[i,j-1,z] + u[i,j,z+1] + u[i,j,z-1] - h**2*f(x[i])*f(y[j])*f(zz[z]))/8
        error = nplng.norm(u - u_0)
        print(error)

    sol = u
    plt.figure()
    plt.imshow(sol[:,:,30])
    plt.xlabel("x")
    plt.ylabel("y",rotation="horizontal",labelpad=20)
    plt.colorbar()
    plt.show()

    plt.figure()
    plt.imshow(sol[:,:,50])
    plt.xlabel("x")
    plt.ylabel("y",rotation="horizontal",labelpad=20)
    plt.colorbar()
    plt.show()

    plt.figure()
    plt.imshow(sol[:,:,N-1])
    plt.xlabel("x")
    plt.ylabel("y",rotation="horizontal",labelpad=20)
    plt.colorbar()
    plt.show()
```

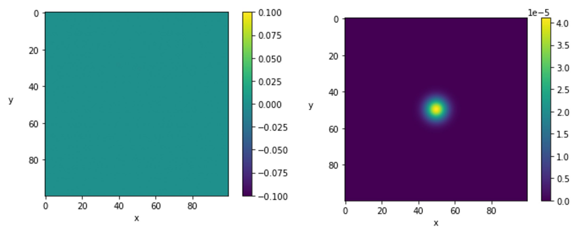
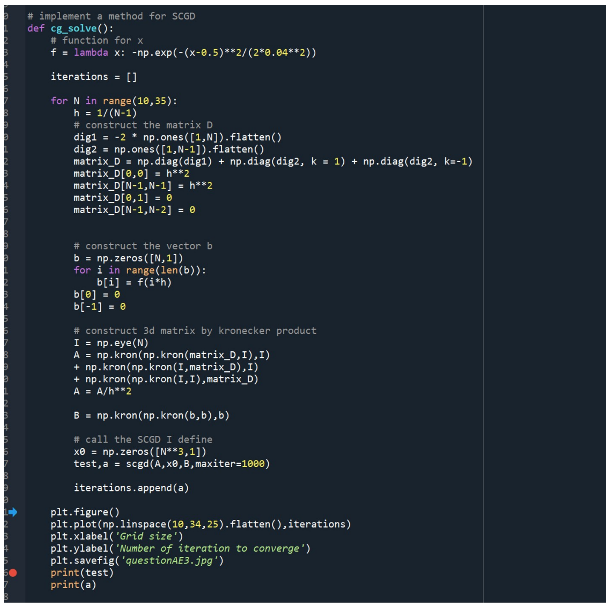


Figure 6. implementation (top) of the matrix-free fashion way (iterative method - Gausse/Sedel method) to calculate 3D Laplacian equation in 100^3 grid size, and the results for 3D equation at boundary(bottom left) and at middle of space(bottom right) of $z = 0.5$

See, if I increase the resolution of grid, the circle looks nicer. Iteration really works well for large grid size. :)

3. the implementation of conjugate GD:
   This method is still matrix heavy task. My computing ranges from 20 to 35 (quite the same as the spsolver, since them both involve with matrix operation), which is the grid size m. For larger grid size, the computer shows warning for allocation of memory.



```python
# implement a method for SCGD
def cg_solve():
    # function for x
    f = lambda x: -np.exp(-(x-0.5)**2/(2*0.04**2))

    iterations = []

    for N in range(10,35):
        h = 1/(N-1)
        # construct the matrix D
        dig1 = -2 * np.ones([1,N]).flatten()
        dig2 = np.ones([1,N-1]).flatten()
        matrix_D = np.diag(dig1) + np.diag(dig2, k = 1) + np.diag(dig2, k=-1)
        matrix_D[0,0] = h**2
        matrix_D[N-1,N-1] = h**2
        matrix_D[0,1] = 0
        matrix_D[N-1,N-2] = 0


        # construct the vector b
        b = np.zeros([N,1])
        for i in range(len(b)):
            b[i] = f(i*h)
        b[0] = 0
        b[-1] = 0

        # construct 3d matrix by kronecker product
        I = np.eye(N)
        A = np.kron(np.kron(matrix_D,I),I)
        + np.kron(np.kron(I,matrix_D),I)
        + np.kron(np.kron(I,I),matrix_D)
        A = A/h**2

        B = np.kron(np.kron(b,b),b)

        # call the SCGD I define
        x0 = np.zeros([N**3,1])
        test,a = scgd(A,x0,B,maxiter=1000)

        iterations.append(a)

    plt.figure()
    plt.plot(np.linspace(10,34,25).flatten(),iterations)
    plt.xlabel('Grid size')
    plt.ylabel('Number of iteration to converge')
    plt.savefig('questionAE3.jpg')
    print(test)
    print(a)
```

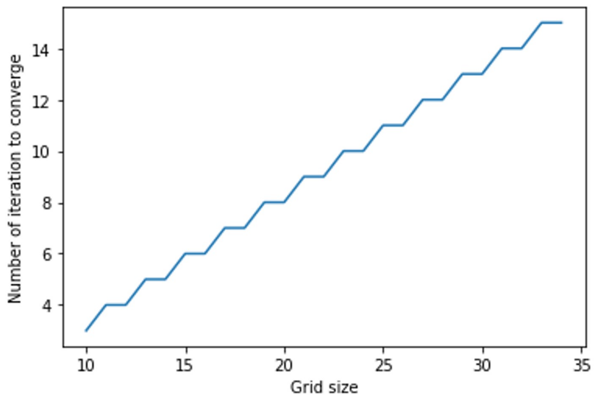


Figure 7. implementation (top image) of the steepest conjugate GD to calculate 3D Laplacian equation with varied grid size, and convergence speed (bottom figure) at different grid size (m,one-dimensional).

The speed of convergence relative to grid size m turns out to be a linear line, which proves that the iteration rate of conjugate gradient descent is bounded by $O(m)$.

Solution B

For this problem, I utilize the python library Scipy.interpolate to compute the different sampling size for the target equation. Here is the figure I got varying from 94 to 104.

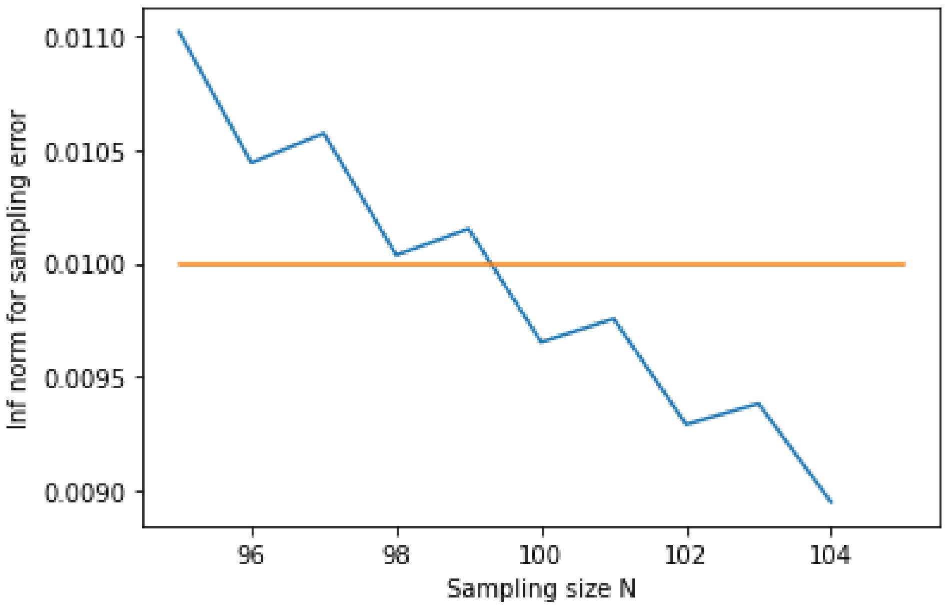


Figure 8. uniform norm between sampling size N and N+1

According to the figure 1, sampling size 100 should be the smallest size at most 0.01 uniform norm.

Solution C

Solution (a)

To solve 2D wave equation which is time derivatives hypobolic equation, I wrote the partial differential equations in point-wise equation:

$$\frac{u_{i,j}^{t+1} - 2u_{i,j}^{t} + u_{i,j}^{t-1}}{\Delta t^2} = \frac{u_{i+1,j}^{t} + u_{i-1,j}^{t} + u_{i,j+1}^{t} + u_{i,j-1}^{t} - 4u_{i,j}^{t}}{\Delta x^2}$$

The initial condition:

$$u_{i,j}^{0} = 0$$

$$\frac{u_{i,j}^{1} - u_{i,j}^{0}}{\Delta t} = f(x_i)f(y_j)$$

The boundary condition is the Dirichlet boundary condition which can be derived from the equation in problem B. To meet with the stability requirement, I set $\Delta t = 1/2\Delta x$, ranging from 0 to 1, therefore $T = 1$. To obtain the "analytical solution", I compute one scenario with high resolution of $\Delta x = 1024$. Here are results I made.

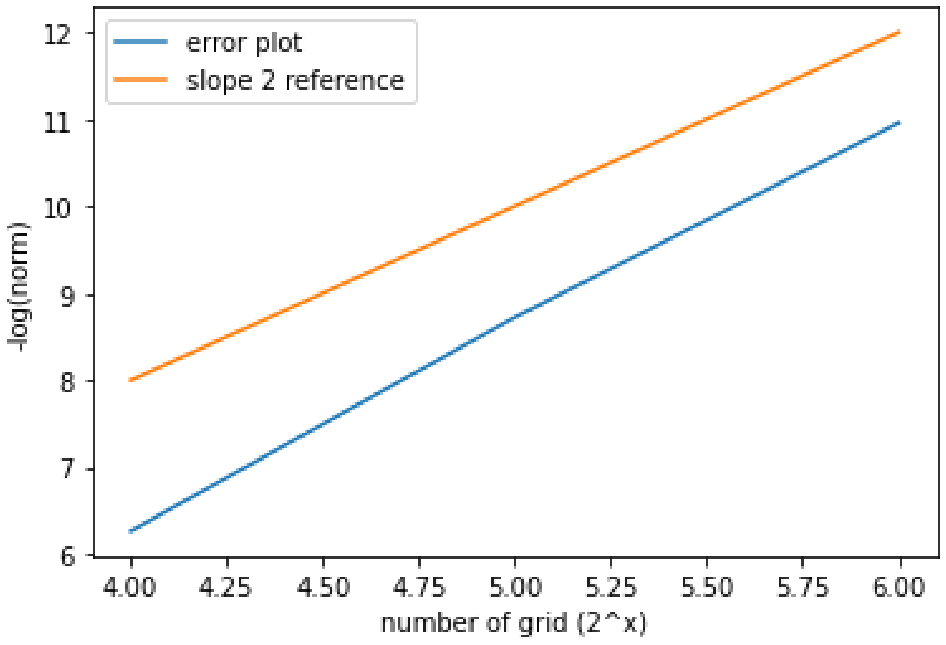


Figure 9. error plot with different grid size. The blue line is the plot from code, the orange line is the reference line with slope of 2.

The result I have perfectly shows the second-order accuracy.

Solution (b)

Since I applied the 3-point discretization for second time derivatives, the point-wise equation is shown as:

$$\frac{y_{i,j}^{t+1} - 2y_{i,j}^{t} + y_{i,j}^{t-1}}{\Delta t^2} = \lambda y^t$$

Rewrite the equation as following:

$$y_{i,j}^{t+1} - (2 + \lambda \Delta x^2) y_{i,j}^{t} + y_{i,j}^{t-1} = 0$$

Substitute $y^t = \rho^t$:

$$\rho^2 - (2 + \lambda \Delta x^2)\rho + 1 = 0$$

To obtain the strong stability, we need to have $||\rho|| \leq 1$. Solutions for the equation are:

$$\rho = \frac{2 + \lambda \Delta x^2 \pm \sqrt{(2 + \lambda \Delta x^2)^2 - 4}}{2} \leq 1$$

Rearrange it, we have the real and image part of field:

$$-4 \leq Re(\lambda \Delta x^2) \leq 0$$

$$Im(\lambda \Delta x^2) = 0$$

The stable complex plain are:

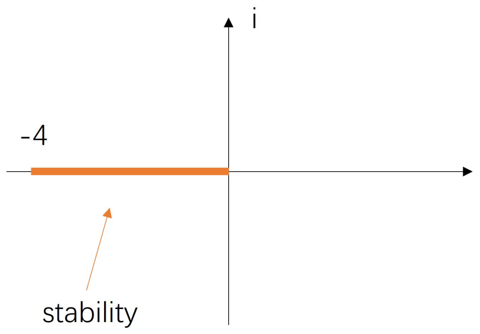


Figure 10. the complex plane and stability zone (mark in orange)

Solution (c)

Here, to easily compute the eigenvalues of matrix, I just directly applied the solution from last homework 1 so that I can rewrite the problem in Kronecker product:

$$u_{tt} = \Delta u = \frac{(A \otimes I_n + I_n \otimes B)}{\Delta x^2} U = \frac{(A \otimes B)}{\Delta x^2} U$$

where,

$$A = \begin{pmatrix} -2 & 1 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & \cdots & 0 \\ \vdots & \cdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & \cdots & 1 & -2 \end{pmatrix}$$

From homework 1, we obtained that the eigenvalues of $A \otimes I_n + I_n \otimes B$ should be $\lambda_i \lambda_j$, where λ_i and λ_j are the eigenvalues for matrix A and B, respectively. In this case, we have symmetric field so that $A = B$. For tridiagnoal matrix A, its eigenvalues ranges: $-4 \leq \lambda_i \leq 0$. Then the eigenvalues of $A \otimes A$ ranges: $0 \leq \lambda_i \lambda_j \leq 16$. Combine with the solution in (b), we have the following stability condition:

$$\frac{\Delta t^2}{\Delta x^2} \geq -1 \geq 0$$

So, it is always stable.

Solution (d)

Replace $u_{i,j}$ in part (a) with $\rho \exp{(ik_1 j_1 \Delta x)} \exp{(ik_2 j_2 \Delta x)}$

$$\rho^{n+1} \exp{(ik_1 j_1 \Delta x)} \exp{(ik_2 j_2 \Delta x)} - 2\rho^n \exp{(ik_1 j_1 \Delta x)} \exp{(ik_2 j_2 \Delta x)} + \rho^{n-1} \exp{(ik_1 j_1 \Delta x)} \exp{(ik_2 j_2 \Delta x)} =$$
$$\frac{\rho^n \Delta t^2}{\Delta x^2} (\exp{(ik_1 (j_1+1)\Delta x)} \exp{(ik_2 j_2 \Delta x)} + \exp{(ik_1 (j_1-1)\Delta x)} \exp{(ik_2 j_2 \Delta x)} + \exp{(ik_1 j_1 \Delta x)} \exp{(ik_2 (j_2+1)\Delta x)}$$
$$+ \exp{(ik_1 j_1 \Delta x)} \exp{(ik_2 (j_2-1)\Delta x)} - 4\exp{(ik_1 j_1 \Delta x)} \exp{(ik_2 j_2 \Delta x)})$$

Rearrange the equation:

$$\begin{aligned} g(k) - 2 + \frac{1}{g(k)} &= \frac{\Delta t^2}{\Delta x^2} (2\exp{(ik\Delta x)} + 2\exp{(-ik\Delta x)} - 4) \\ &= \frac{\Delta t^2}{\Delta x^2} (4\cos{(k\Delta x)} - 4) \end{aligned}$$

Then,

$$\begin{aligned} g(k) &= \frac{\Delta t^2}{\Delta x^2} (4\cos{(k\Delta x)} - 4) - 1/g(k) + 2 \\ &= -1/8 \frac{\Delta t^2}{\Delta x^2} \sin^2(k\Delta/2) - 1/g(k) + 2 \end{aligned}$$

To meet with the stability $g(k) \leq 1$, we need to have:

$$\frac{\Delta t^2}{\Delta x^2} \leq \frac{1}{2}$$

Solution (e)

Apply the FT to the equation in part (a), we have the following series:

$$u(x,y,t) = \sum_{n=1}^{\infty}\sum_{m=1}^{\infty} \sin(m\pi x)\sin(n\pi y)(B_{mn}\cos(\lambda_{mn}t) + B_{mn}\sin(\lambda_{mn}t))$$

where,

$$\lambda_{mn} = \sqrt{(mx)^2 + (ny)^2}$$

and B_{mn} is the sparse matrix determined by the boundary condition. The physical extra terms are dispersive.

Solution D (Bonus question)

Proving the Lax equivalence theorem for two time derivatives:

$$y'' = \lambda y$$

where the scheme of y'' is $\frac{y_{i,j}^{t+1} - 2y_{i,j}^{t} + y_{i,j}^{t-1}}{\Delta t^2}$

1. Consistency
   We need to prove following condition:

   $$||\tau^{\Delta x, \Delta t}|| \to 0 \text{ as } \Delta x, \Delta t \to 0$$

   We don't have the spatial discretization so only discretized in time:

   $$\tau = \frac{y(t+\Delta t) - 2y(t) + y(t+\Delta t)}{\Delta t^2} - y''$$

   By Taylor Expansion

   $$y(t+\Delta t) = y(t) + y'(t)\Delta t + \frac{y''(t)}{2}\Delta t^2 + O(\Delta t^3)$$

   $$y(t-\Delta t) = y(t) - y'(t)\Delta t + \frac{y''(t)}{2}\Delta t^2 + O(\Delta t^3)$$

   Combine with the previous equation, we have:

   $$\tau = \frac{y''(t)\Delta t^2 - O(\Delta t^3)}{\Delta t^2} - y'' = O(\Delta t)$$

   Therefore, $\Delta t \to 0$, $||\tau^{\Delta t}|| \to 0$ Consistency maintain.

2. Weak stability

Github link

You can find colorful image here: https://github.com/lubyant/CS714.git
:)