Homework assignment 1

Question 1:

Section a:

$We\ will\ denote\ the\ discrete\ function\ as\ \boldsymbol{u}, and\ we\ use\ it\ as\ a\ vector\ of\ length\ n = \left\lfloor\frac{\pi}{h}\right\rfloor, such\ thst\ \boldsymbol{u_i} + \mathrm{O(h)} = \mathrm{u(x_i)}\ for\ i = 0, \dots n.\ Now, from\ the\ Dirichlet\ BC\ we\ know\ that\ \boldsymbol{u_0} = 1.\ Using\ central\ differance\ for\ seconed\ derivative\ we\ know\ that\colon \frac{\boldsymbol{u_{i+1}} - 2\boldsymbol{u_i} + \boldsymbol{u_{i-1}}}{h^2} + O(h^2) = -4\cos(2x).$

$Using\ Neuwmann\ BC\ with\ we\ can\ calculate\colon$

$\frac{\boldsymbol{u_{n+1}} - \boldsymbol{u_n}}{\boldsymbol{h}} = \boldsymbol{0} \Rightarrow \boldsymbol{u_n} = \boldsymbol{u_{n+1}}.From\ here\ we\ can\ define\ the\ Laplacian\ operator\ L'$

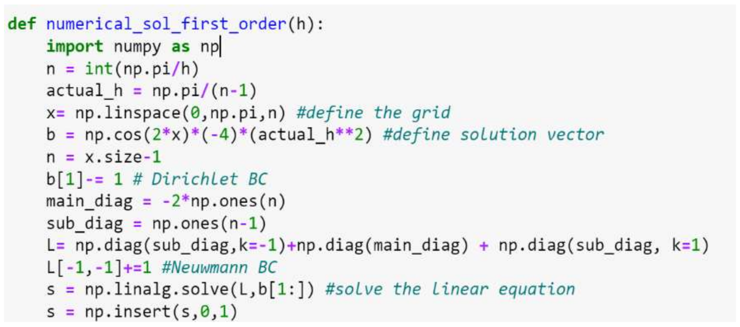
$$to\ be\colon L' = \begin{pmatrix} -2 & 1 & 0 & \dots & \dots & 0 \\ 1 & -2 & 1 & 0 & \dots & 0 \\ 0 & 1 & \ddots & & & \\ \vdots & & \ddots & \ddots & & \\ \vdots & & & 1 & -2 & 1 \\ 0 & \dots & \dots & 0 & 1 & -2 \end{pmatrix}.$$

$$We\ want\ to\ solve\ the\ linear\ equation\colon L' \begin{pmatrix} \boldsymbol{u_1} \\ \boldsymbol{u_2} \\ \vdots \\ \boldsymbol{u_{n-2}} \\ \boldsymbol{u_n} \end{pmatrix} = \begin{pmatrix} -4\cos(2x_1)h^2 - 1 \\ -4\cos(2x_2)\,h^2 \\ \vdots \\ -4\cos(2x_{n-2})\,h^2 \\ -4\cos(2x_{n-1})h^2 - \boldsymbol{u_n} \end{pmatrix}.$$

$$So\ now\ we\ get\colon L \begin{pmatrix} \boldsymbol{u_1} \\ \boldsymbol{u_2} \\ \vdots \\ \boldsymbol{u_{n-1}} \end{pmatrix} = \begin{pmatrix} -4\cos(2x_1)\,h^2 - 1 \\ -4\cos(2x_2)\,h^2 \\ \vdots \\ -4\cos(2x_{n-1})\,h^2 \end{pmatrix}$$

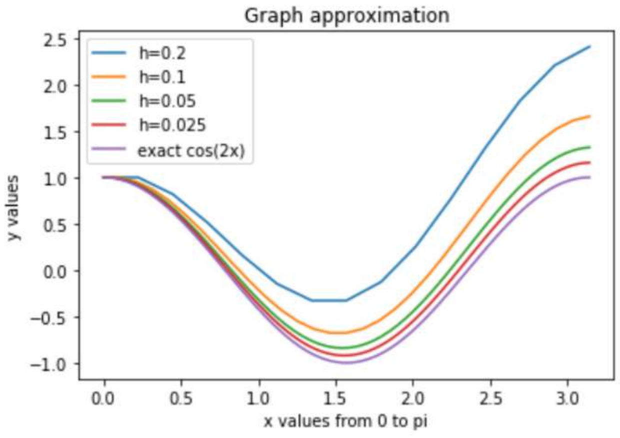
$$Where\ L = \begin{pmatrix} -2 & 1 & 0 & \dots & \dots & 0 \\ 1 & -2 & 1 & 0 & \dots & 0 \\ 0 & 1 & \ddots & & & \\ \vdots & & \ddots & \ddots & & \\ \vdots & & & 1 & -2 & 1 \\ 0 & \dots & \dots & 0 & 1 & \boldsymbol{-1} \end{pmatrix}.(notice\ the\ difference\ in\ L[n,n])$$

The code in Python will be as follows: (full code in zip file)

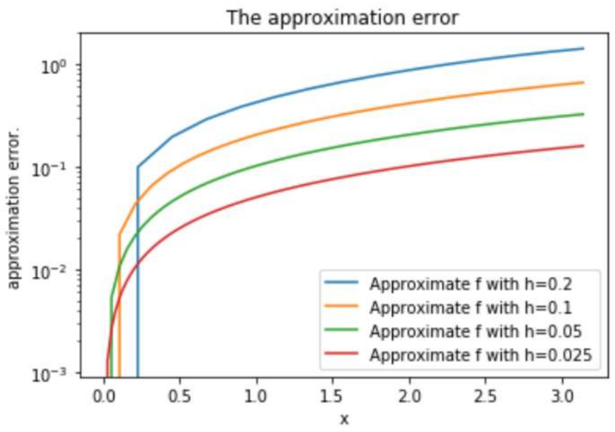


```python
def numerical_sol_first_order(h):
    import numpy as np
    n = int(np.pi/h)
    actual_h = np.pi/(n-1)
    x= np.linspace(0,np.pi,n) #define the grid
    b = np.cos(2*x)*(-4)*(actual_h**2) #define solution vector
    n = x.size-1
    b[1]-= 1 # Dirichlet BC
    main_diag = -2*np.ones(n)
    sub_diag = np.ones(n-1)
    L= np.diag(sub_diag,k=-1)+np.diag(main_diag) + np.diag(sub_diag, k=1)
    L[-1,-1]+=1 #Neuwmann BC
    s = np.linalg.solve(L,b[1:]) #solve the linear equation
    s = np.insert(s,0,1)
```

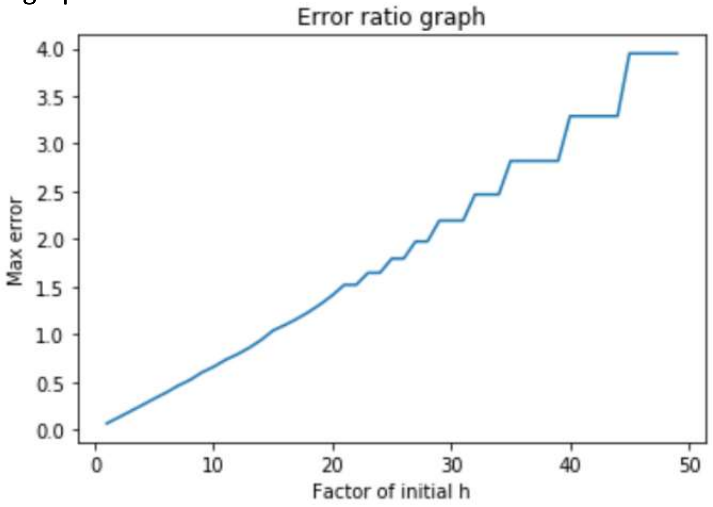
:Plotting the results we get:



Error graph in semi-logarithmic scale



Error ratio graph with initial h=0.01:



We can see the error grows in approximately linearity scale as h gets bigger, since we used first order approx.

Section b:

$We\ will\ use\ second\ order\ central\ finite\ difference\ for\ Neumann\ BC.:$

$From\ the\ second\ derivative\ we\ know\ that: -4\cos(2x_n) = \frac{\boldsymbol{u_{n+1}} - 2\boldsymbol{u_n} + \boldsymbol{u_{n-1}}}{h^2}$

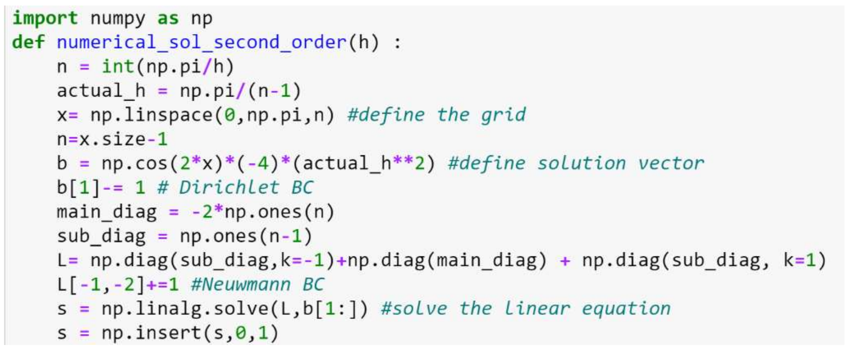
$and\ from\ the\ seconed\ order\ to\ Neumann\ BC\ we\ get\ 0 = \frac{\boldsymbol{u_{n+1}} - \boldsymbol{u_{n-1}}}{2h} \Rightarrow \boldsymbol{u_{n+1}} = \boldsymbol{u_{n-1}}$

$Combining\ these\ equations\ we\ get: -4\cos(2x_n)\,h^2 = 2\boldsymbol{u_{n-1}} - 2\boldsymbol{u_n}.$

So now we can define our system of equations as follows:

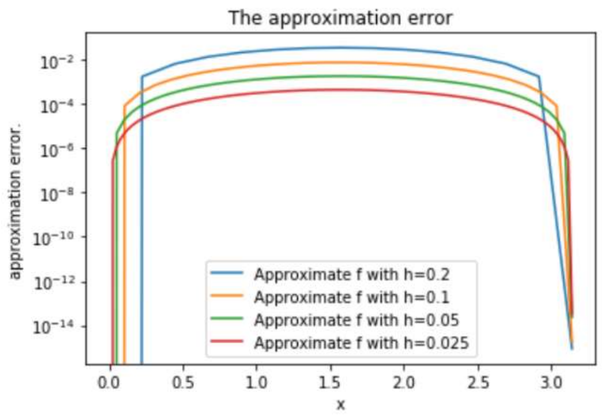
$$\begin{pmatrix} -2 & 1 & 0 & \dots & \dots & 0 \\ 1 & -2 & 1 & 0 & \dots & 0 \\ 0 & 1 & \ddots & & & \\ \vdots & & \ddots & \ddots & & \\ \vdots & & & 1 & -2 & 1 \\ 0 & \dots & \dots & 0 & \mathbf{2} & -2 \end{pmatrix} \begin{pmatrix} \boldsymbol{u_1} \\ \boldsymbol{u_2} \\ \vdots \\ \boldsymbol{u_{n-1}} \\ \boldsymbol{u_n} \end{pmatrix} = \begin{pmatrix} -4\,cos(2x_1)h^2 - 1 \\ -4\,cos(2x_2)\,h^2 \\ \vdots \\ -4\,cos(2x_{n-1})\,h^2 \\ -4\,cos(2x_n)h^2 \end{pmatrix}$$

The code in Python will be as follows:

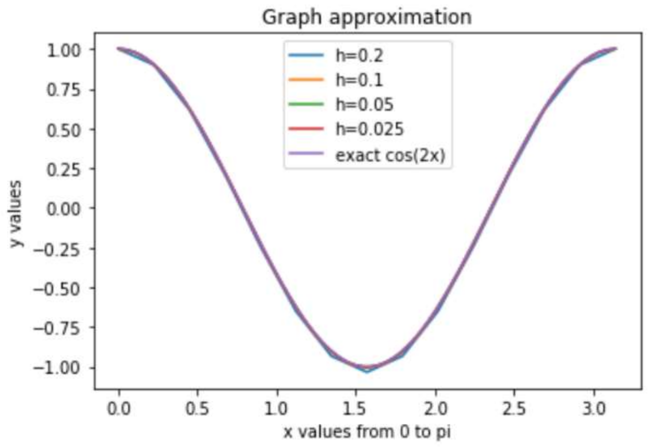


```
import numpy as np
def numerical_sol_second_order(h) :
    n = int(np.pi/h)
    actual_h = np.pi/(n-1)
    x= np.linspace(0,np.pi,n) #define the grid
    n=x.size-1
    b = np.cos(2*x)*(-4)*(actual_h**2) #define solution vector
    b[1]-= 1 # Dirichlet BC
    main_diag = -2*np.ones(n)
    sub_diag = np.ones(n-1)
    L= np.diag(sub_diag,k=-1)+np.diag(main_diag) + np.diag(sub_diag, k=1)
    L[-1,-2]+=1 #Neuwmann BC
    s = np.linalg.solve(L,b[1:]) #solve the linear equation
    s = np.insert(s,0,1)
```

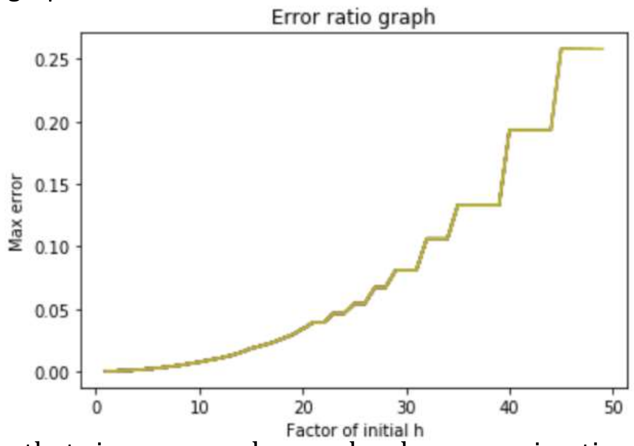
Error graph in semi logarithmic scale:



Plotting the results



Error ratio graph with initial h=0. 01:



We can see that since we used second order approximation, the error grows in approximately square scale as h gets bigger.

Comparing the error graphs in semi logarithmic scale at both sections we can see that the in the second order approximation the solution is much more accurate. For instance, the max error in the first order for h=0.025 is bigger then the max order in the second order for only h=0.2.

Section c:

First, we will solve the ODE analytically:

$$By\ the\ laws\ of\ derivative\ we\ know: \frac{\partial\left(\sigma(x)\left(\frac{\partial u}{\partial x}\right)\right)}{\partial x} = \sigma'(x)\frac{\partial u}{\partial x} + \sigma(x)\frac{\partial^2 u}{\partial^2 x} = q(x)$$

$$so\ now\ we\ have: \sigma'(x)\frac{\partial u}{\partial x} + \sigma(x)\frac{\partial^2 u}{\partial^2 x} = \sigma'(x)(-4\cos(2x)) - \sigma(x)(2\sin(2x))$$

$$From\ here\ we\ can\ see\ that\ \frac{\partial u}{\partial x} = -2\sin(2x)\ and\ \frac{\partial^2 u}{\partial^2 x} = -4\cos(2x)\ satisfies$$

$$the\ solution.\ Now\ we\ get\ \int \partial u = \int -2\sin(2x)\partial x\ \ so\ u = \cos(2x) + c.$$

$$Putting\ together\ with\ the\ BC\ we\ get: 1 = \cos(0) + c \Rightarrow c = 0.$$

$So\ the\ solution\ to\ the\ ODE\ is\ u(x) = \cos(2x).$

Using the guide we calculate: $\frac{\partial u}{\partial x}(x_i) \approx \frac{\boldsymbol{u}_{i+\frac{1}{2}} - \boldsymbol{u}_{i-\frac{1}{2}}}{h}$, now we multiply the result with $\sigma(x)$ and apply the operator on the result:

$$\frac{\partial\left(\sigma(x)\left(\frac{\partial u}{\partial x}\right)\right)}{\partial x}(x_i) \approx \frac{\sigma\left(x_{i+\frac{1}{2}}\right)(\boldsymbol{u}_{i+1} - \boldsymbol{u}_i)}{h^2} - \frac{\sigma\left(x_{i-\frac{1}{2}}\right)(u_i - u_{i-1})}{h^2} =$$

$$= \frac{1}{h^2}\left(\sigma\left(x_{i-\frac{1}{2}}\right)\boldsymbol{u}_{i-1} + \left(-\sigma\left(x_{i+\frac{1}{2}}\right) - \sigma\left(x_{i-\frac{1}{2}}\right)\right)\boldsymbol{u_i} + \sigma\left(x_{i+\frac{1}{2}}\right)\boldsymbol{u}_{i+1}\right).$$

Using Dirichlet BC can approximate: $h^2\cos(2x) = \sigma\left(x_{\frac{1}{2}}\right)\boldsymbol{u}_0 + \left(-\sigma\left(x_{\frac{3}{2}}\right) - \sigma\left(x_{\frac{1}{2}}\right)\right)\boldsymbol{u_1} + \sigma\left(x_{\frac{3}{2}}\right)\boldsymbol{u}_2$

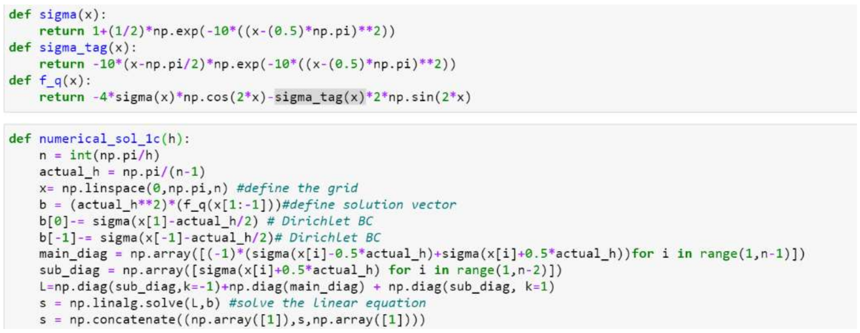
$$\Rightarrow \left(-\sigma\left(x_{\frac{3}{2}}\right) - \sigma\left(x_{\frac{1}{2}}\right)\right)\boldsymbol{u_1} + \sigma\left(x_{\frac{3}{2}}\right)\boldsymbol{u}_2 = h^2 q(x_1) - \sigma\left(x_{\frac{1}{2}}\right)$$

And similarly: $\sigma\left(x_{n-\frac{3}{2}}\right)\boldsymbol{u}_{n-2} + \left(-\sigma\left(x_{n-\frac{1}{2}}\right) - \sigma\left(x_{n-\frac{3}{2}}\right)\right)\boldsymbol{u_{n-1}} = h^2 q(x_{n-1}) - \sigma\left(x_{n-\frac{1}{2}}\right)$

So we want to solve the following linear equation:

$$L\begin{pmatrix} \boldsymbol{u_1} \\ \boldsymbol{u_2} \\ \vdots \\ \boldsymbol{u_{n-2}} \\ \boldsymbol{u_{n-1}} \end{pmatrix} = \begin{pmatrix} q(x_1) - \sigma\left(x_{\frac{1}{2}}\right) \\ q(x_2)h^2 \\ \vdots \\ q(x_{n-2})h^2 \\ q(x_{n-1})h^2 - \sigma\left(x_{n-\frac{1}{2}}\right) \end{pmatrix} \text{For } [L]_{i,j} = \begin{cases} -\sigma\left(x_{i+\frac{1}{2}}\right) - \sigma\left(x_{i-\frac{1}{2}}\right) & if\ i = j \\ \sigma\left(x_{i-\frac{1}{2}}\right) & if\ i = j+1 \\ \sigma\left(x_{i+\frac{1}{2}}\right) & if\ i = j-1 \\ 0 & else \end{cases}$$

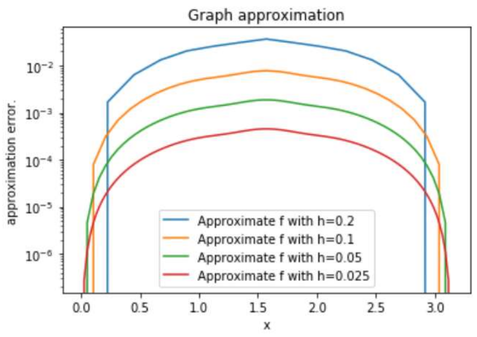
The code in python will be as follows:



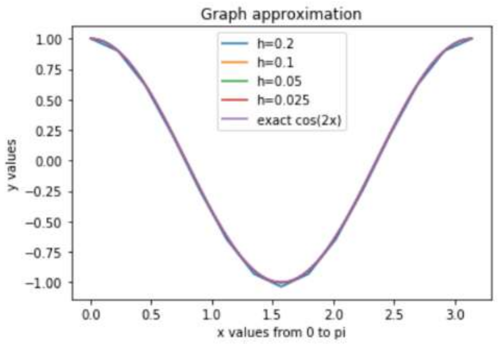
```python
def sigma(x):
    return 1+(1/2)*np.exp(-10*((x-(0.5)*np.pi)**2))
def sigma_tag(x):
    return -10*(x-np.pi/2)*np.exp(-10*((x-(0.5)*np.pi)**2))
def f_q(x):
    return -4*sigma(x)*np.cos(2*x)-sigma_tag(x)*2*np.sin(2*x)
```

```python
def numerical_sol_1c(h):
    n = int(np.pi/h)
    actual_h = np.pi/(n-1)
    x= np.linspace(0,np.pi,n) #define the grid
    b = (actual_h**2)*(f_q(x[1:-1]))#define solution vector
    b[0]-= sigma(x[1]-actual_h/2) # Dirichlet BC
    b[-1]-= sigma(x[-1]-actual_h/2)# Dirichlet BC
    main_diag = np.array([(-1)*(sigma(x[i]-0.5*actual_h)+sigma(x[i]+0.5*actual_h))for i in range(1,n-1)])
    sub_diag = np.array([sigma(x[i]+0.5*actual_h) for i in range(1,n-2)])
    L=np.diag(sub_diag,k=-1)+np.diag(main_diag) + np.diag(sub_diag, k=1)
    s = np.linalg.solve(L,b) #solve the linear equation
    s = np.concatenate((np.array([1]),s,np.array([1])))
```

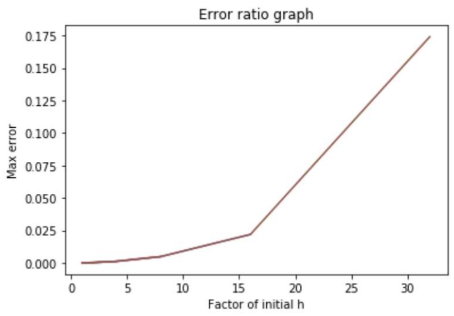
Error graph in semi logarithmic scale:



Plotting the results



Error ratio with initial h=0.01:


Error ratio graph
Max error
Factor of initial h