EE5011:Spline Interpolation and Bicubic spline

Arunima Sarkar-EE21S062

September 2021

1 Introduction

The given function is : $f(x) = \frac{x^{1+J_0}}{\sqrt{(1-x)(1+100x^2)}}$

1.1 Code for Q1 Q2 Q3:

Initially the function has been sampled at 15 points between 0.1 and 0.9 and plotted(Figure 1).

```
import matplotlib.pyplot as plt
import numpy as np
import math as m
import scipy.special as sp
from os import system
system("f2py -c -m spline spline.f")
import spline as s
```

```
def deri1(x):
 y=0.0
 y=-(((x**sp.j0(x))*(-100*x**3 + 2*(100*x**3 - 100*x**2+x-1)
   *sp.j0(x) - (2*(100*x**3 - 100*x**2 + x-1)*x*m.log10(x)
   *sp.j1(x)) +x-2))/(2*(-100*x**3 + 100*x**2 -x +1)**1.5))
    return y

def func(x):
    y1=0.0
    y1=x**(1+sp.j0(x))/(np.sqrt(1-x+100*x**2-100*x**3))
```

```
    return y1

x=list(np.linspace(0.1,0.9,700))
y=[]
for i in range(len(x)):
    y.append(func(x[i]))

plt.clf()
plt.plot(x,y,'r+')
plt.grid()
plt.xlabel("Sample points")
plt.ylabel("function values")
plt.savefig("Function at sample values")

y2a=[0]*len(y)

y1a=[0]*len(y)
for i in range(len(y)):
    y1a[i]=deri1(x[i])

y2a=s.spline(x,y,1.e30,1.e30) #method 1 : Natural spline


xx=list(np.linspace(0.1,0.9,1000))

yt=[]
for i in range(len(xx)):
    yt.append(func(xx[i]))

yy=s.splintn(x,y,y2a,xx)

plt.clf()
plt.semilogy(xx,abs(yy-yt),'r')
plt.xlabel("Sample points")
plt.ylabel("Order of error: Logarithmic")
plt.grid()
plt.savefig("Error profile_M1 Natural spline")
```

The below plot(Figure 1) shows function at 15 Sample points:

The function is analytic for $0.1 <= x <= 0.9$ as all of it's derivatives exist there. However its ROC at 0.9 and at 0.1 is 0.1.

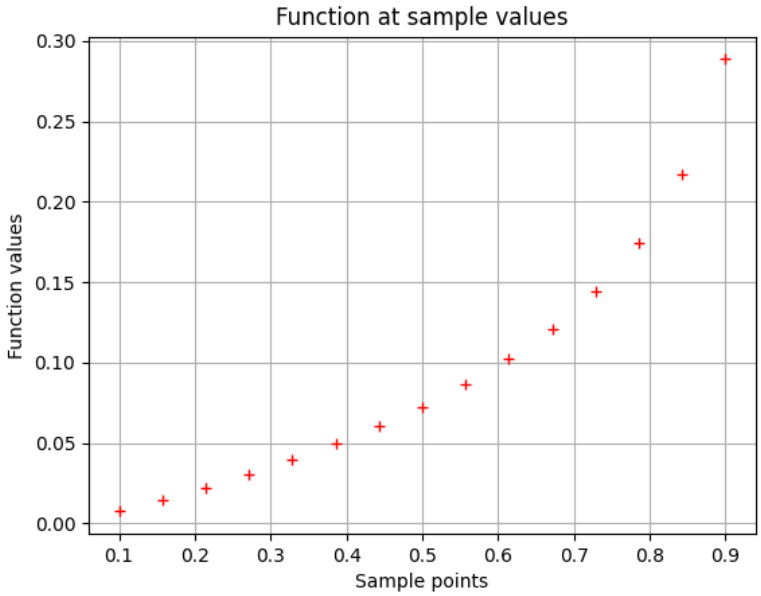

Figure 1: function values

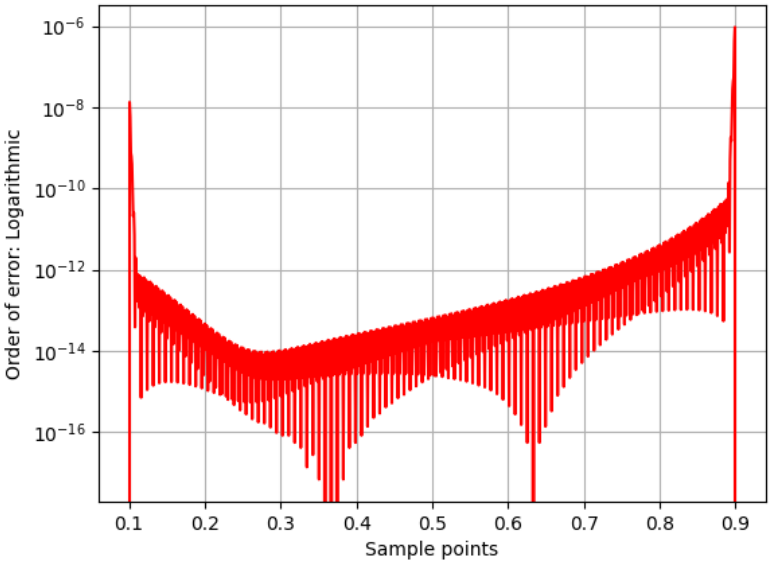

Figure 2: Error curve for 700 points

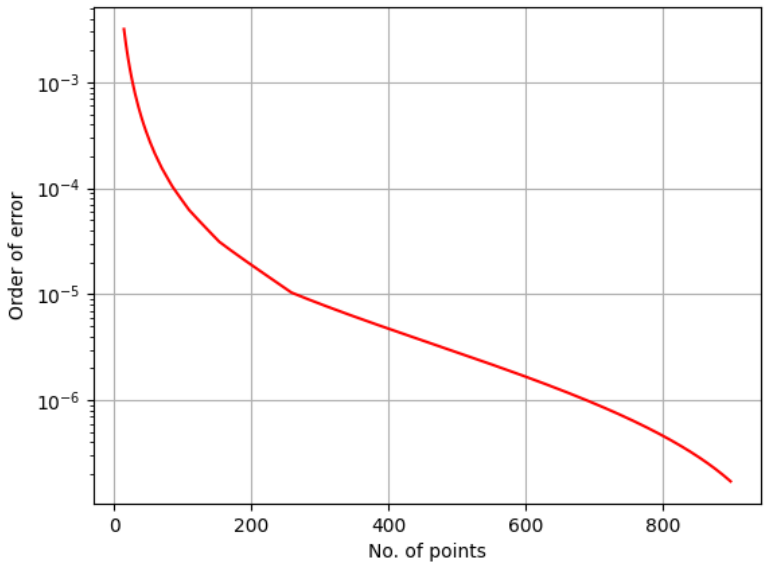

Figure 3: Error profile for increasing no. of points

The error reduces with increasing no. of points and eventually reaches order 6 at 700 points.

1.2 Code for Q4

Here we introduced the cubic spline using not-a-knot method.

```
import matplotlib.pyplot as plt
import numpy as np
import math as m
import scipy.special as sp
from os import system
system("f2py -c -m spline spline.f")
import spline as s


def deri1(x):
 y=0.0
 y=-(((x**sp.j0(x))*(-100*x**3 + 2*(100*x**3 - 100*x**2+x-1)
   *sp.j0(x) - (2*(100*x**3 - 100*x**2 + x-1)*x*m.log10(x)
   *sp.j1(x)) +x-2))/(2*(-100*x**3 + 100*x**2 -x +1)**1.5))
    return y
```

```
def func(x):
    y1=0.0
    y1=x**(1+sp.j0(x))/(np.sqrt(1-x+100*x**2-100*x**3))
    return y1

x=list(np.linspace(0.1,0.9,685))
y=[]
for i in range(len(x)):
    y.append(func(x[i]))
n=len(x)
arr=np.zeros((n-2,n-2))
h=[]
d=[]
for i in range(n-1):
    h.append(x[i+1]-x[i])
for i in range(n-2):
    d.append(6*((y[i+2]-y[i+1])-(y[i+1]-y[i]))/h[0])

for i in range(1,n-3):
    arr[i,i]=2*(h[i-1]+h[i])
    arr[i,i-1]=h[i-1]
    arr[i,i+1]=h[i]
b1=h[0]+2*h[1]
c1=h[1]-h[0]
d[0]=(d[0]*h[1])/2
an=h[n-2]-h[n-3]
bn=2*h[n-3]-h[n-2]
d[n-3]=(d[n-3]*h[n-3])/2
arr[0,0]=b1
arr[0,1]=c1
arr[n-3,n-4]=an
arr[n-3,n-3]=bn

y2a=np.linalg.solve(arr,d)

y3a=[0]*len(y)
y3a[0]=((h[0]+h[1])/h[1])*y2a[0] - (h[0]/h[1])*y2a[1]
y3a[n-1]=((x[n-1]-x[n-3])/(x[n-2]-x[n-3]))*y2a[-1]
          - ((x[n-1]-x[n-2])/(x[n-2]-x[n-3]))*y2a[-2]
for i in range(len(y2a)):
    y3a[i+1]=y2a[i]
xx=list(np.linspace(0.1,0.9,1000))
yy=s.splintn(x,y,y3a,xx)
yt=[]
```

```
for i in range(len(xx)):
    yt.append(func(xx[i]))

plt.clf()
plt.semilogy(xx,abs(yt-yy),'r')
plt.grid()
plt.xlabel('Sample points')
plt.ylabel('Order of error')
plt.savefig("not a knot")
```

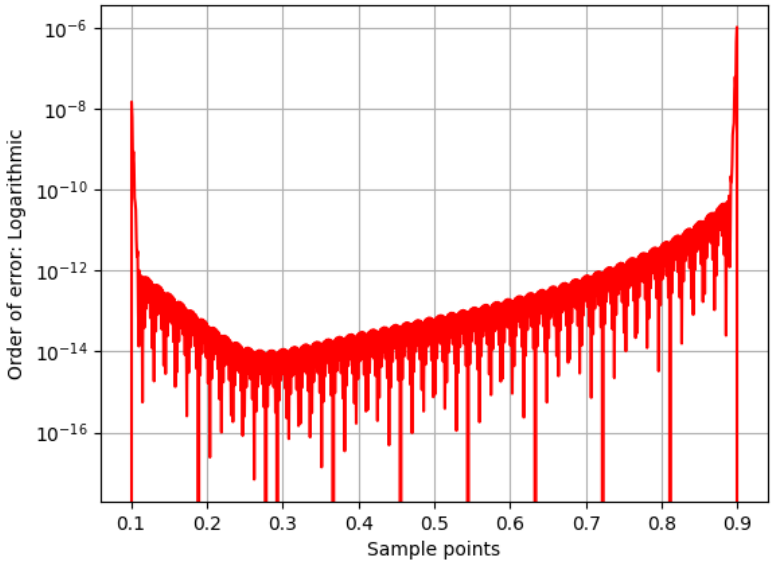

Figure 4: Error profile for not-a-knot, N=685

1.3 Code for Q5

Here we introduced the cubic spline where first order derivatives are given.

```
import matplotlib.pyplot as plt
import numpy as np
import math as m
import scipy.special as sp
from os import system
system("f2py -c -m spline spline.f")
import spline as s
```

```
def deri1(x):
 y=0.0
 y=-(((x**sp.j0(x))*(-100*x**3 + 2*(100*x**3 - 100*x**2+x-1)
   *sp.j0(x) - (2*(100*x**3 - 100*x**2 + x-1)*x*m.log10(x)
   *sp.j1(x)) +x-2))/(2*(-100*x**3 + 100*x**2 -x +1)**1.5))
    return y

def func(x):
    y1=0.0
    y1=x**(1+sp.j0(x))/(np.sqrt(1-x+100*x**2-100*x**3))
    return y1
x=list(np.linspace(0.1,0.9,665))
y=[]
for i in range(len(x)):
    y.append(func(x[i]))

y2a=[0]*len(y)

y1a=[0]*len(y)
for i in range(len(y)):
    y1a[i]=deri1(x[i])

y2a=s.spline(x,y,deri1(0.1),deri1(0.9)) #method 2 : Using derivatives at
                                                    the end


xx=list(np.linspace(0.1,0.9,1000))

yt=[]
for i in range(len(xx)):
    yt.append(func(xx[i]))

yy=s.splintn(x,y,y2a,xx)

plt.clf()
plt.semilogy(xx,abs(yy-yt),'b')
plt.xlabel("Sample points")
plt.ylabel("Order of error: Logarithmic")
plt.grid()
plt.savefig("Error profile_M2")
```

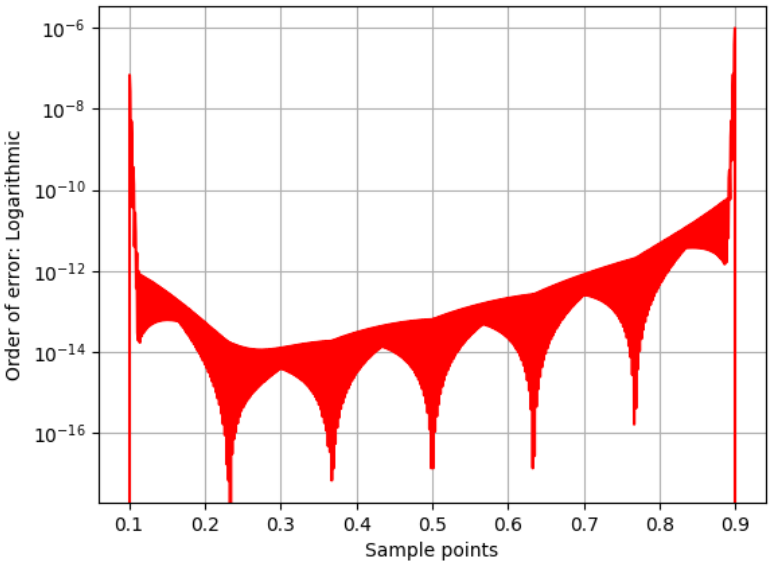

Figure 5: Error curve for 660 points

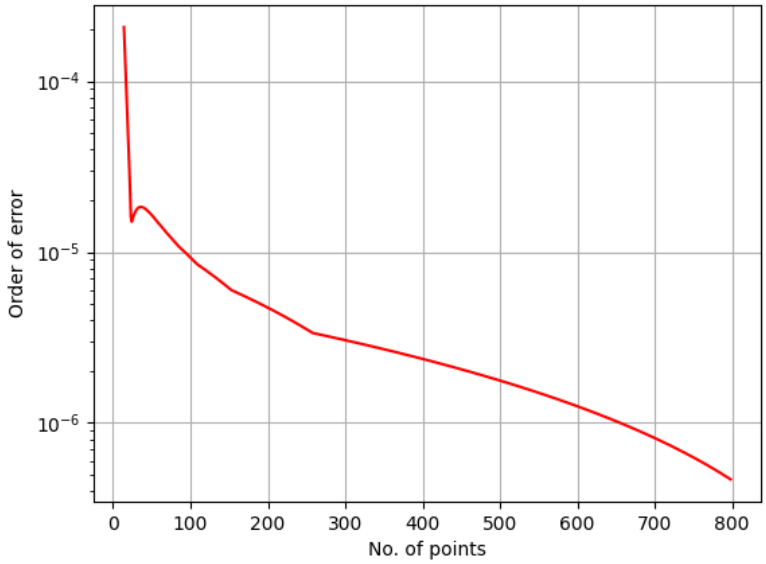

Figure 6: Error profile for increasing no. of points

Here(Figure 5 and Figure 6) the order of error reaches 6 at around 660 points.

1.4 Code for Q6

```
import matplotlib.pyplot as plt
import numpy as np
import math as m
import scipy.special as sp
from os import system
system("f2py -c -m spline spline.f")
import spline as s

def deri1(x):
 y=0.0
 y=-(((x**sp.j0(x))*(-100*x**3 + 2*(100*x**3 - 100*x**2+x-1)
   *sp.j0(x) - (2*(100*x**3 - 100*x**2 + x-1)*x*m.log10(x)
   *sp.j1(x)) +x-2))/(2*(-100*x**3 + 100*x**2 -x +1)**1.5))
    return y

def func(x):
    y1=0.0
    y1=x**(1+sp.j0(x))/(np.sqrt(1-x+100*x**2-100*x**3))
    return y1

x=list(np.linspace(0.1,0.9,12000))
y=[]
for i in range(len(x)):
    y.append(func(x[i]))
y2a=[0]*len(y)
y3a=s.spline(x,y,100*deri1(0.1),100*deri1(0.9))
xx=list(np.linspace(0.1,0.9,15000))
yt=[]
for i in range(len(xx)):
    yt.append(func(xx[i]))
yy=s.splintn(x,y,y3a,xx)
plt.clf()
plt.semilogy(xx,abs(yy-yt),'b')
plt.xlabel("Sample points")
plt.ylabel("Order of error: Logarithmic")
plt.grid()
plt.savefig("Error profile_100xDerivatives")
err=[]
xval=[]
for i in range(50,800):
    x=list((np.linspace(0.1,0.9,i)))
```

```
        y=[]
        for i in range(len(x)):
            y.append(func(x[i]))
        y2a=s.spline(x,y,100*deri1(0.1),100*deri1(0.9))
        xx=list(np.linspace(0.1,0.9,m.floor(i/8)))
        yy=s.splintn(x,y,y2a,xx)
        xval.append(i)
        yt=[]
        for i in range(len(xx)):
            yt.append(func(xx[i]))
        err.append(max(abs(yt-yy)))
plt.clf()
plt.semilogy(xval,err,"r")
plt.ylabel("Order of error")
plt.grid()
plt.xlabel("No. of points")
plt.savefig("E6")
```

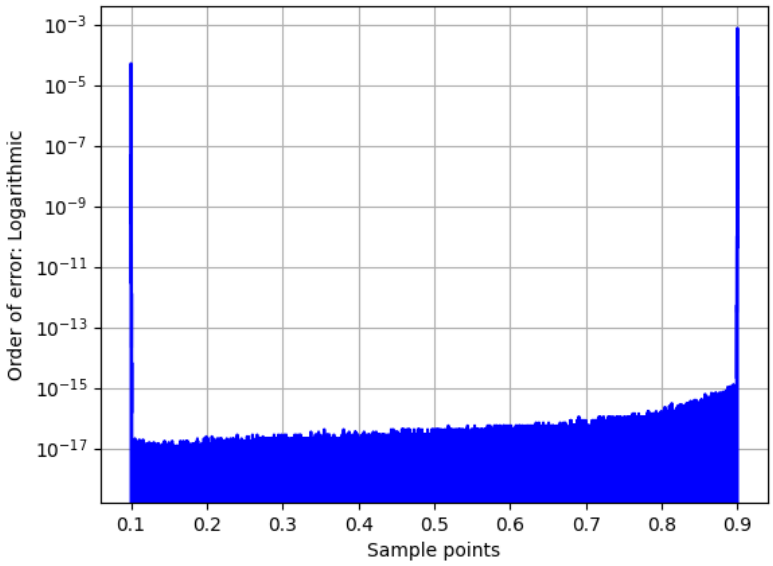

Figure 7: Error at N=12000, interpolated at 15000 points

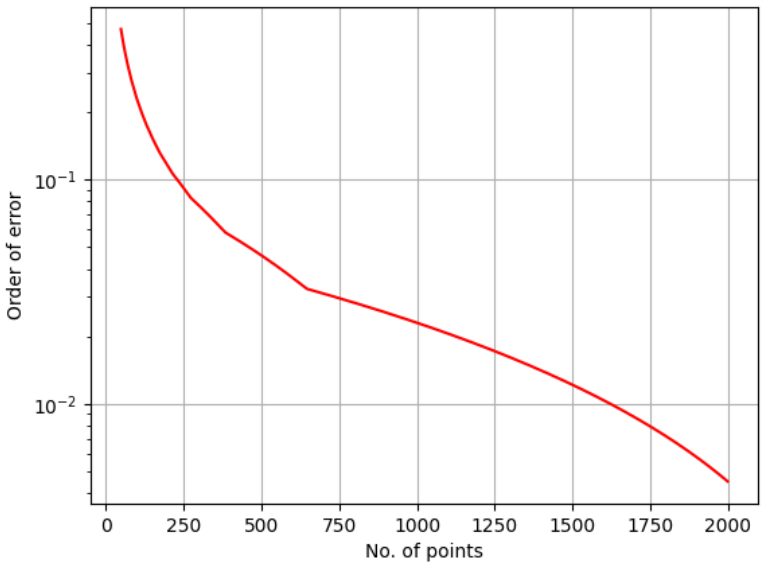

Figure 8: Maximum Error profile for wrong derivative

As we can see that if wrong derivatives are given at the end then the error drastically increases and the order of error doesn't even reaches any close to order 6 even for 12000 points.

1.5 Code for Q7

```
import matplotlib.pyplot as plt
import numpy as np
import math as m
import scipy.special as sp
from os import system
system("f2py -c -m spline spline.f")
import spline as s

def deri1(x):
  y=0.0
  y=-(((x**sp.j0(x))*(-100*x**3 + 2*(100*x**3 - 100*x**2+x-1)
   *sp.j0(x) - (2*(100*x**3 - 100*x**2 + x-1)*x*m.log10(x)
   *sp.j1(x)) +x-2))/(2*(-100*x**3 + 100*x**2 -x +1)**1.5))
    return y

def func(x):
    y1=0.0
    y1=x**(1+sp.j0(x))/(np.sqrt(1-x+100*x**2-100*x**3))
    return y1


x=list(np.linspace(0.1,0.2,10))
x2=list(np.linspace(0.21,0.4,25))

x3=list(np.linspace(0.41,0.6,50))
x4=list(np.linspace(0.61,0.79,45))

x5=list(np.linspace(0.8,0.9,10))
x=x+x2+x3+x4+x5
y=[]
for i in range(len(x)):
    y.append(func(x[i]))
y2a=[0]*len(y)

y2a=s.spline(x,y,deri1(0.1),deri1(0.9))


xx=list(np.linspace(0.1,0.9,100))
yt=[]
for i in range(len(xx)):
    yt.append(func(xx[i]))
yy=s.splintn(x,y,y2a,xx)
err=max(abs(yy-yt))
plt.clf()
```

```
plt.grid()
plt.semilogy(xx,abs(yy-yt),"b")
plt.savefig("aru_10 25 45 50 10")
```

The above code shows how non uniform spacing affects the error. The following conclusions are made from the error plots received:

1) Upon comparing Figure 9 with Figure 10, Figure 11 we can see that maximum error changes as it drops a bit. However the error profile for $0.2 \leq x \leq 0.8$ can't be made uniform with certain distribution of points.

2) Here upon doing trial and error I have reached to the conclusion that for about 665 points I can have partially uniform error plot but that too for certain regions only.

3) 4 intervals: Code for point distribution:

```
x=list(np.linspace(0.1,0.2,100))
x2=list(np.linspace(0.21,0.4,180))
x3=list(np.linspace(0.41,0.79,285))
x4=list(np.linspace(0.8,0.9,100))
x=x+x2+x3+x4
```

In Figure 10, the error between $0.2 \leq x \leq 0.4$ remains almost constant and for $0.4 \leq x \leq 0.8$ it keeps on linearly increasing like the uniform distribution.

4)5 intervals: Code for point distribution:

```
x=list(np.linspace(0.1,0.2,100))
x2=list(np.linspace(0.21,0.4,155))
x3=list(np.linspace(0.41,0.6,125))
x4=list(np.linspace(0.61,0.79,185))
x5=list(np.linspace(0.8,0.9,100))
x=x+x2+x3+x4+x5
```

Figure 10 and Figure 11 shows that the error envelope for $0.2 \leq x \leq 0.4$ remains almost flat which is around order 14 but for $0.4 \leq x \leq 0.8$ the envelope has positive slope. Meanwhile in Figure 11 the error envelope doesn't change much for $0.2 \leq x \leq 0.4$ but between $0.4 \leq x \leq 0.6$ it shows poor error and between $0.6 \leq x \leq 0.8$ it decreased a bit as higher number of points were taken showing the similar behaviour as observed from Figure 11.

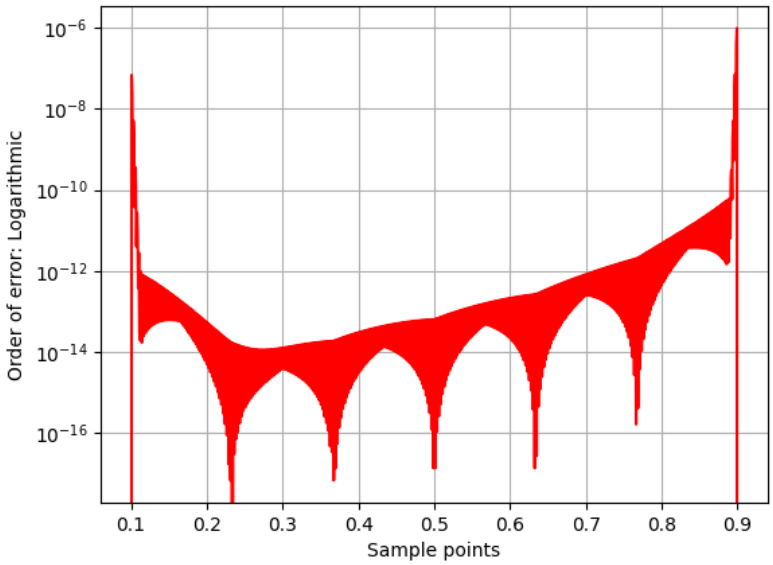

Figure 9: Uniform spacing with 665 points

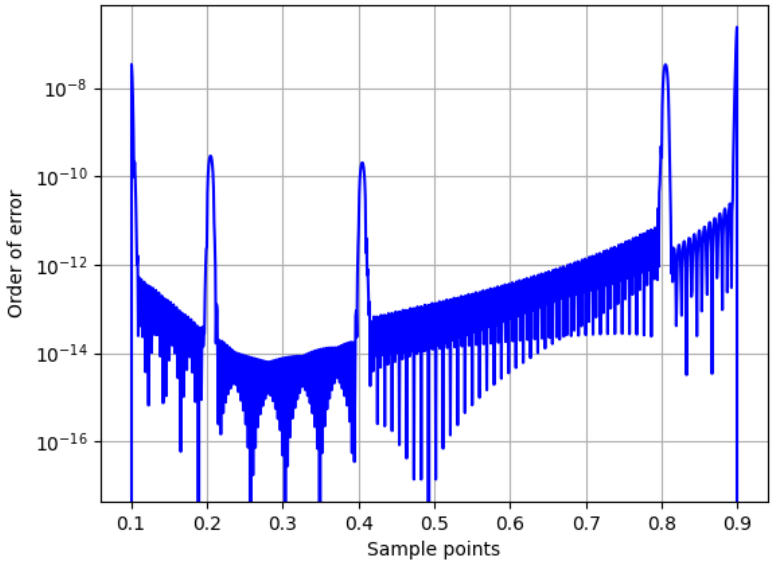

Figure 10: Non Uniform spacing 4 interval;Point distribution:100 180 285 100

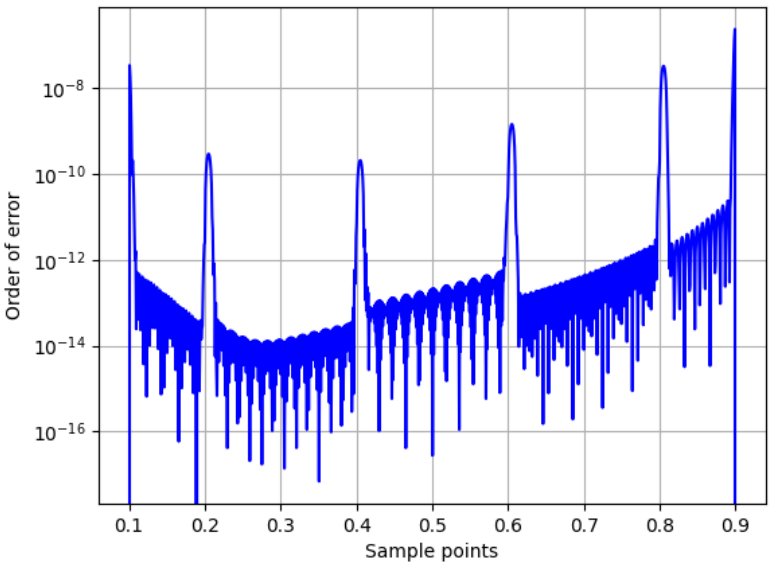

Figure 11: Non Uniform spacing 5 interval;Point distribution:100 155 125 185 100

Thus for around 665 points we are getting somewhat uniform envelope of error for a certain range but behaviour or error remains same.Thus uneven distribution doesn't give uniform error.

NB: Points at the end are taken less as the errors will always be high there and its difficult make it uniform with the other errors

1.6 Code for Q8 (a)

```
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import scipy.special as sp


def func(x,y):
    z=0.0
    z=(np.sin(np.pi*x))*(np.cos(np.pi*y))
    return z

def locate(x,var):
    jl=jm=jl=j=0

    ju=len(x)

    while((ju-jl)>1):
        jm= (ju+jl) >>1
        if(var>=x[jm]):
            jl=jm
        else:
            ju=jm
    if (var==x[0]):
         j=0
    if(var==x[-1]):
         j=len(x)-2
    else:
        j=jl
    return j


x=list(np.linspace(0,2,5))
y=list(np.linspace(0,2,5))
actual=np.zeros((50,50))

xx=list(np.linspace(0,2,50))
yy=list(np.linspace(0,2,50))

for i in range(len(xx)):
    for j in range(len(yy)):
        actual[i,j]=func(xx[i],yy[j])
```

```
interpol=np.zeros((50,50))

den=1
for m in range(len(xx)):
    for n in range(len(yy)):
    i=locate(x,xx[m])
    j=locate(y,yy[n])
    den=(x[i]-x[i+1])*(y[j]-y[j+1])
    interpol[m,n]=((func(x[i],y[j]))*(xx[m]-x[i+1])*(yy[n]-y[j+1])/den) +
    ((func(x[i+1],y[j]))*(x[i]-xx[m])*(yy[n]-y[j+1])/den) +
    ((func(x[i],y[j+1]))*(xx[m]-x[i+1])*(y[j]-yy[n])/den) +
    ((func(x[i+1],y[j+1]))*(x[i]-xx[m])*(y[j]-yy[n])/den)


error=np.zeros((50,50))

for i in range(50):
    for j in range(50):
        error[i,j]=abs(actual[i,j]-interpol[i,j])

x=np.arange(0,2,0.04)
y=np.arange(0,2,0.04)
x,y=np.meshgrid(x,y)
z=interpol
z1=error
fig = plt.figure()
ax = fig.gca(projection='3d')
surf = ax.plot_surface(x, y, z1, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)
fig.colorbar(surf, shrink=0.5, aspect=5)
plt.savefig("3dplot-error")
```

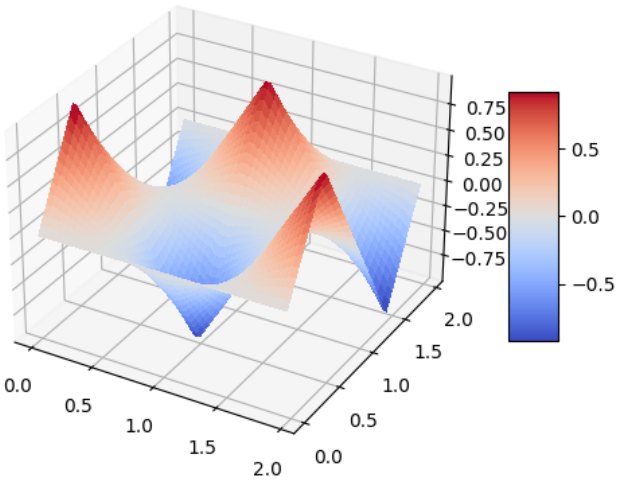

Figure 12: Plot for function for 2D-Bilinear Interpolation

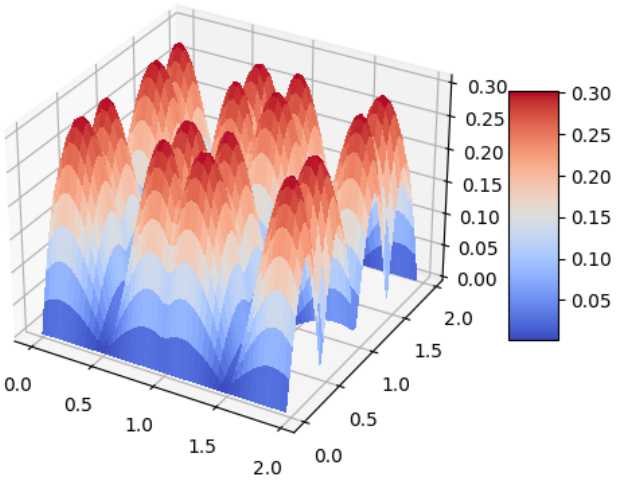

Figure 13: Plot for Error in Bilinear-Interpolation

1.7 Code for Q8 (b)

```
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import scipy.special as sp
from os import system
system("f2py -c -m spline spline.f")
import spline as s

def func(x,y):
    z=0.0
    z=(np.sin(np.pi*x))*(np.cos(np.pi*y))
    return z

def deri2(x,y):
    z=0.0
    z=-(np.pi**2)*(np.sin(np.pi*x))*(np.cos(np.pi*y))
    return z

x=list(np.linspace(0,2,5))
y=list(np.linspace(0,2,5))


sec_deri=np.zeros((5,5))
fval=np.zeros((5,5))
for i in range(5):
    for j in range(5):
        sec_deri[i,j]=deri2(x[j],y[i])
        fval[i,j]=func(x[j],y[i])

xx=list(np.linspace(0,2,50))
yy=list(np.linspace(0,2,50))
yval=np.zeros((5,50))
yfinal=np.zeros((50,50))
for i in range(5):
    yval[i,:]=s.splintn(x,fval[i,:],sec_deri[i,:],xx)

ftrue=np.zeros((5,50))

for i in range(5):
    for j in range(50):
        ftrue[i,j]=func(xx[j],y[i])
```

```
for i in range(50):
    y2a=s.spline(y,yval[:,i],1.e40,1.e40)
    yfinal[i,:]=s.splintn(y,yval[:,i],y2a,yy)

error=np.zeros((50,50))
for i in range(50):
    for j in range(50):
        error[i,j]=abs(yfinal[i,j]-(func(xx[i],yy[j])))

x=np.arange(0,2,0.04)
y=np.arange(0,2,0.04)
x,y=np.meshgrid(x,y)
z=yfinal
z1=error
fig = plt.figure()
ax = fig.gca(projection='3d')
surf = ax.plot_surface(x, y, z1, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)
fig.colorbar(surf, shrink=0.5, aspect=5)
plt.savefig("Cubic spline 3D-error")
```

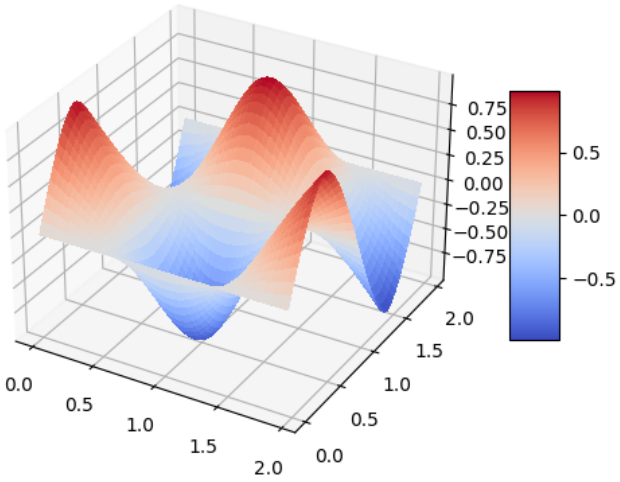

Figure 14: Plot for function for Bi-cubic Interpolation

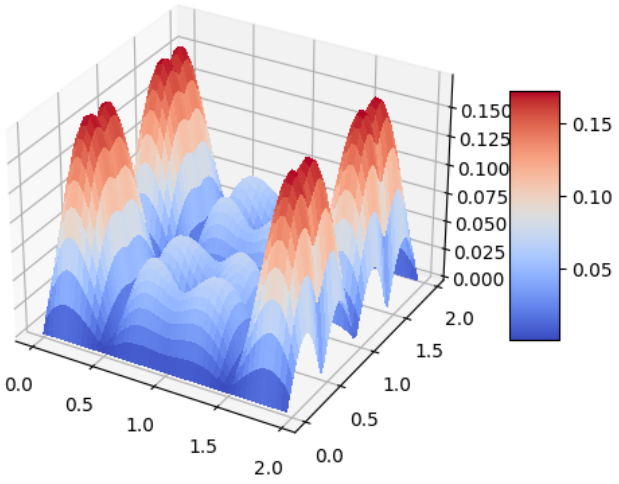

Figure 15: Plot of error for Bi-Cubic Interpolation

1.8 Q8 C

The following diagrams,(Figure 16 and Figure 17) shows the error profile for 20 sample points.For Bi-linear Interpolation the error considerably higher but for Bi-Cubic they fall down and even the graph for Bi-cubic is smoother.

Conclusion: Upon increasing the number of points the error scales down nicely. For Bi-linear interpolation the maximum error with 5 sample value is 0.3 but with 20 sample values it drops 12 times to 0.025. Similarly the error for Bi-cubic interpolation also falls, for 5 sample values the maximum error is close to 0.15 but with 20 sample values the max error reaches 0.005, which means a drop of 30 times.

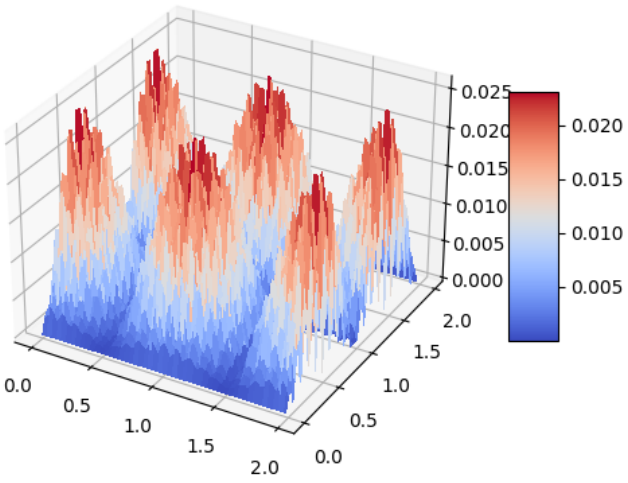

Figure 16: Error plot for Bilinear with 20 points

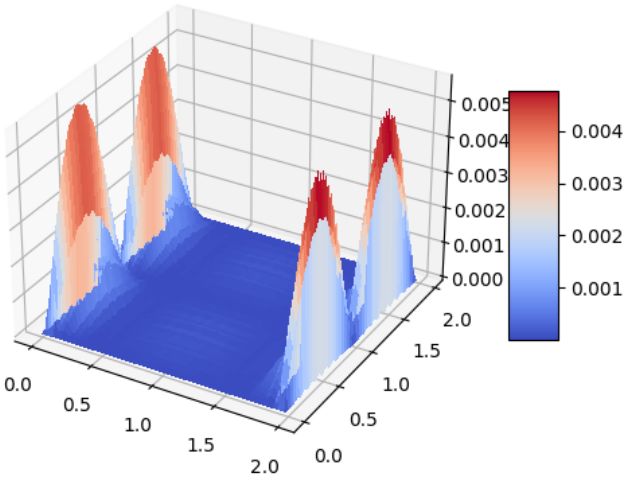

Figure 17: Error plot for Bi-Cubic with 20 points

Here in Figure 18 and 19 I have increased the number of points and for just 350 points the Bi-Cubic spline hits error order or 6 or more and its even more smoother in middle and for Bilinear the maximum error reaches order 6 by 350 points however error at other points are not so good. Hence 2D interpolation in overall is better then Linear interpolation and Bicubic is better than Bi-Linear interpolation.

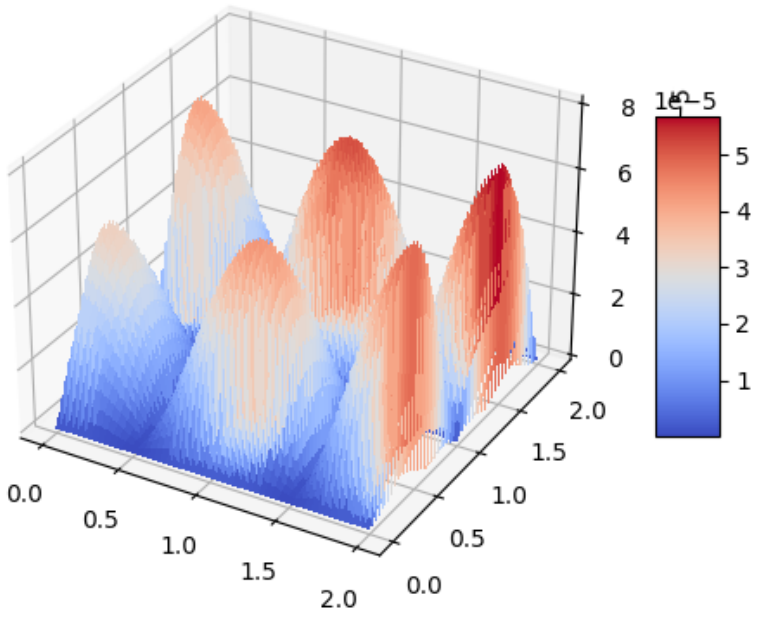

Figure 18: Error plot for Bilinear with 350 points

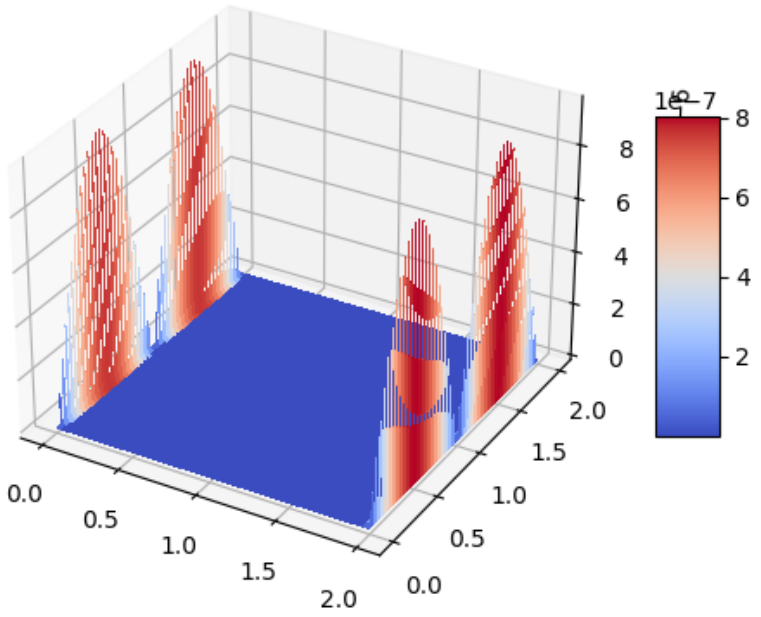

Figure 19: Error plot for Bi-Cubic with 350 points