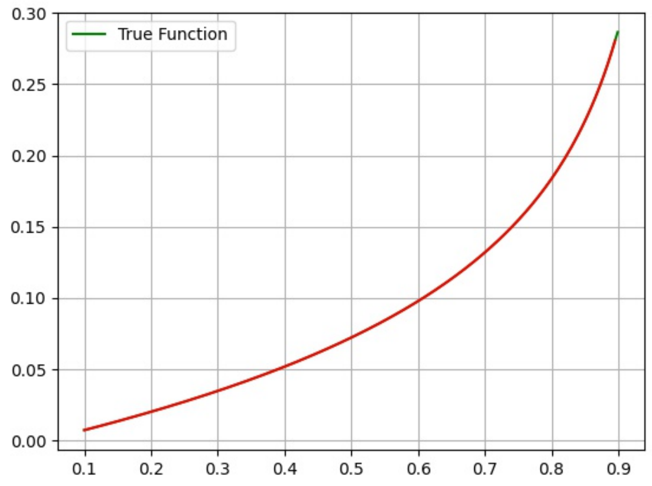
EE5011 Week2 Assignment

ee21s061

September 2021

1. Plot for $\frac{x^{1+J_0(x)}}{\sqrt{1-x+100x^2-100x^3}}$



‘

2.Is it analytic in that region? What is the radius of convergence at x = 0 and at x = 0.9?

The function is analytic in the region [0.1,0.9] since its derivative exists and is continuous on this interval. The function has one singularity when defined on the real numbers $\mathbb{R}$: at x=1 and two additional singularities when defined on the complex numbers $\mathbb{C}$: at x=$\pm 0.1\iota$. Thus, the radius of convergence of the function is 0.1 at both x=0.1 and x=0.9.

3.Program for using Spline interpolation with y”=0 at boundary. To get six order of accuracy with uniformly spaced points we are varying the points from 16 to 800.

```
from matplotlib.pyplot import semilogy
from pylab import *
from os import system
import spline
import numpy as np
from scipy import special
err=[]
till=800
for i in range(16,till):
    xa=np.linspace(0.1,0.9,i)
    fx=[(w**(1+special.j0(w)))/(np.sqrt(1-w+(100*(w**2))-(100*(w**3)))) for w in xa]
    y2a=spline.spline(xa,fx,2*10**30,2*10**30)
    xx=np.linspace(0.1,0.9,1000)
    yya=spline.splintn(xa,fx,y2a,xx)
    fx_1=[(w**(1+special.j0(w)))/(np.sqrt(1-w+(100*(w**2))-(100*(w**3)))) for w in xx]
    err.append(max(abs(yya-fx_1)))
plt.clf()
figure(1)
semilogy(range(16,till),err,'r')
legend(["Max Error"])
grid(True)
figure(1).savefig("q3_plot_1.jpg")
```

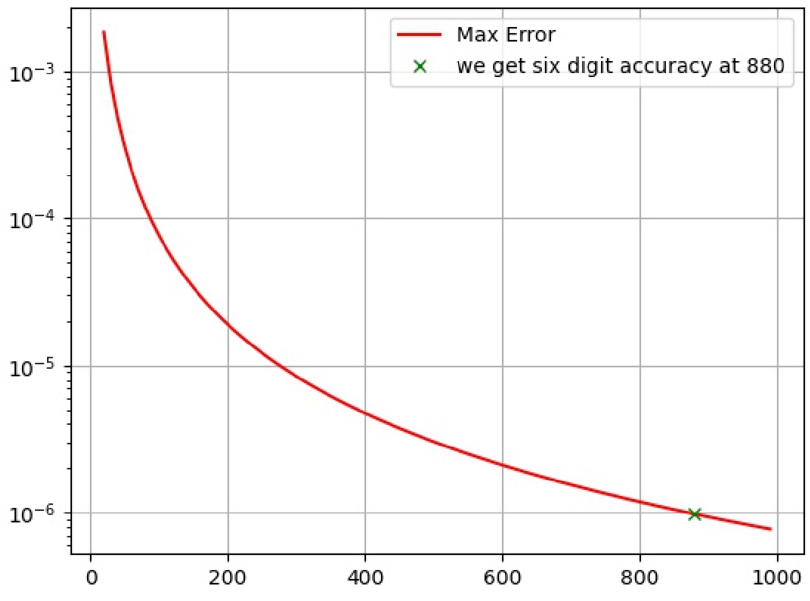


Figure: order of error vs no. of uniformly spaced points.

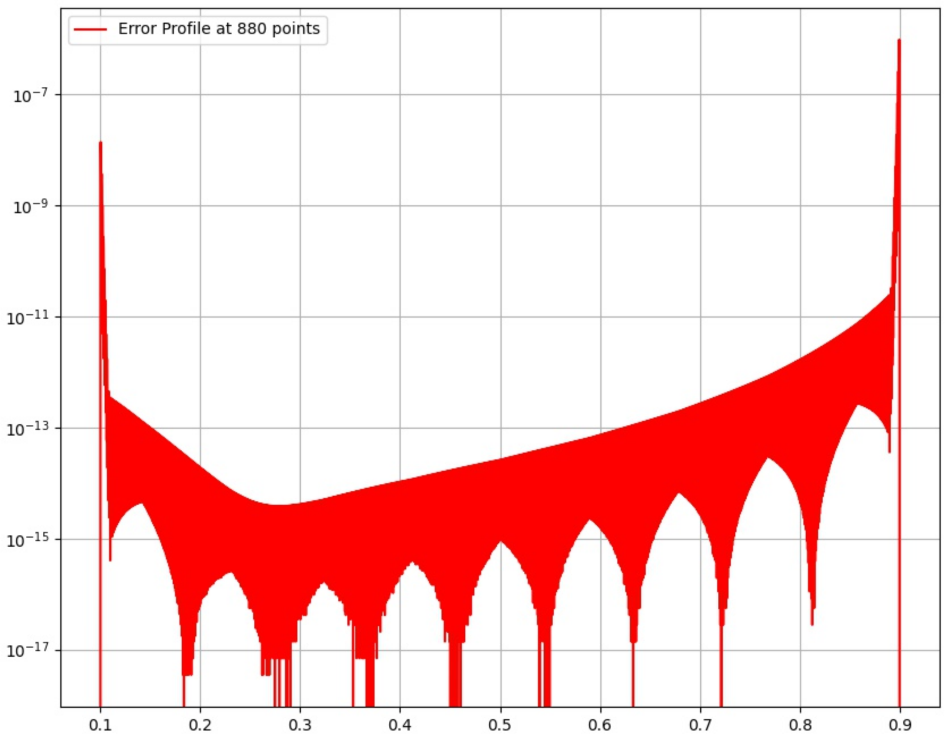


Figure: Error Profile for Natural Spline

Observations:

- As Expected the setting $y_1'' = y_n'' = 0$ gives us the most error and it takes roughly 880 points to get to six digit accuracy.
- Unlike Polynomial Interpolation we don't get to see increase in error eventually as we increase number of points and also the drop in error is significant in beginning but it gradually slows down giving us the graph we see above.

Q4: Program for using Spline interpolation with not a knot condition. To get six order of accuracy with uniformly spaced points we are varying the points from 16 to 800 and a look at error profile for 10^{-6}

Fortran Subroutine to get y" based on not a knot condition. Python code is similar to previous one just with one fucntion call changed.

```
SUBROUTINE splinenak(x,y,n,y2,a,b,c,d)
c
cf2py intent(out) :: y2
cf2py intent(hide) :: n
cf2py intent(hide) :: a
cf2py intent(hide) :: b
cf2py intent(hide) :: c
cf2py intent(hide) :: d
cf2py double precision :: u(n)
cf2py double precision :: x(n)
cf2py double precision :: y(n)
cf2py double precision :: a(n)
cf2py double precision :: b(n)
cf2py double precision :: c(n)
cf2py double precision :: d(n)
c
      INTEGER n
      DOUBLE PRECISION x(n),y(n),y2(n)
      INTEGER i,k,j
      DOUBLE PRECISION b(n),d(n),c(n),a(n),p,q

      b(1)=(x(2)-x(1))+(2.*(x(3)-x(2)))
      c(1)=(x(3)-x(2))-(x(2)-x(1))
      p=((y(3)-y(2))/(x(3)-x(2)))-((y(2)-y(1))/(x(2)-x(1)))
      d(1)=(6.*p*(x(3)-x(2)))/((x(3)-x(1)))
      a(n)=(x(n-1)-x(n-2))-(x(n)-x(n-1))
      b(n)=(2.*(x(n-1)-x(n-2)))+(x(n)-x(n-1))
      p=((y(n)-y(n-1))/(x(n)-x(n-1)))-((y(n-1)-y(n-2))/(x(n-1)-x(n-2)))
      d(n)=(6.*p*(x(n-1)-x(n-2)))/((x(n)-x(n-2)))
      do j=2,n-1
        a(j)=(x(j)-x(j-1))
        c(j)=(x(j+1)-x(j))
        b(j)=2*(a(j)+c(j))
        p=(y(j+1)-y(j))/(x(j+1)-x(j))
        q=(y(j)-y(j-1))/(x(j)-x(j-1))
        d(j)=6*(p-q)
      end do
      do i=1,n-1
```

```
    a(i+1)=x(i+1)-x(i)
    c(i+1)=x(i+1)-x(i)
    p=a(i+1)*c(i)
    b(i+1)=b(i+1)-(p/(b(i)))
    p=a(i+1)*d(i)
    d(i+1)=d(i+1)-(p/(b(i)))
end do
y2(n)=d(n)/b(n)
do k=n-1,1,-1
    y2(k)=(d(k)-(c(k)*y2(k+1)))/b(k)
end do
return
END
```

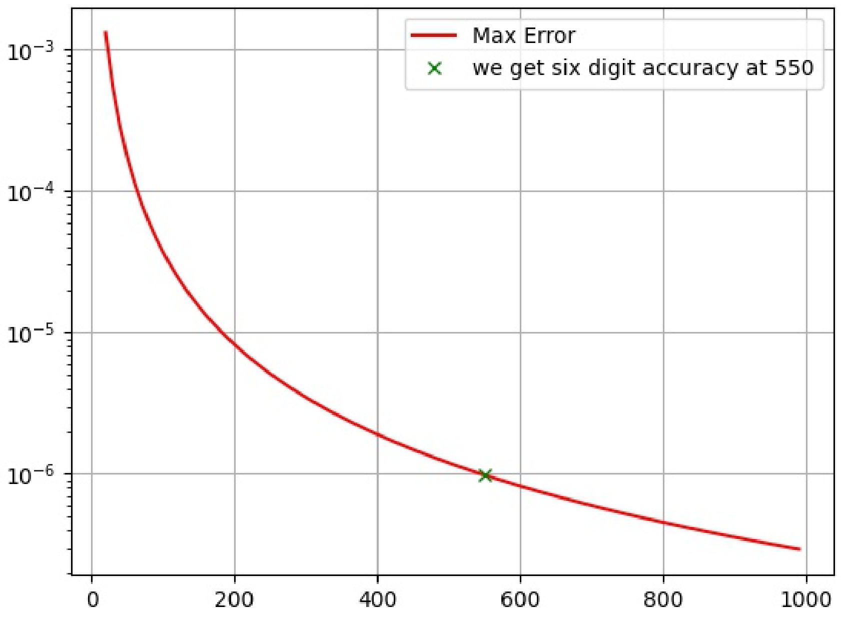


Figure: order of error vs no. of uniformly spaced points.

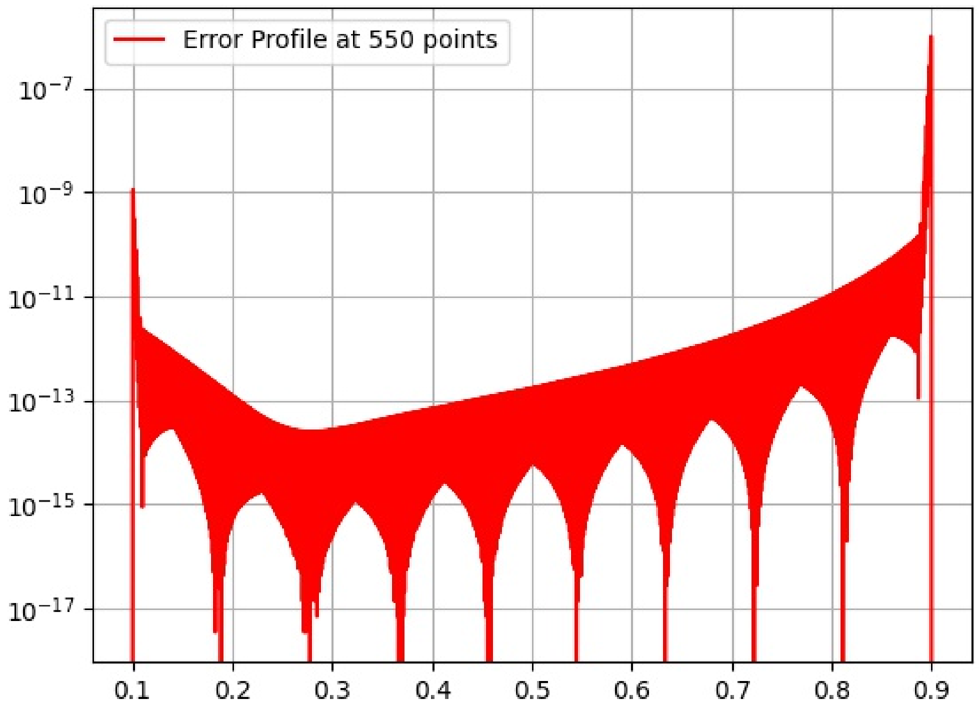


Figure: Error Profile for Not a Knot Condition

Observations

- Not a knot significantly reduces the error and gives us six digit accuracy at 550 points itself which is a pretty good deal. This also makes sense according to theory.
- Not a knot makes a really good option as we don't always have the original function that the sampled values are derived from so this is a good trade between accuracy and data requirements.
- In the right i plotted the error as the function goes from 0.1 to 0.9 and we can see that error is max at the edges and drop significantly at between them. This makes sense as the plot in middle is cubic spline where as the points in the ends are effected by the decision to make y_1'' and y_n'' extrapolation based on previous points.

Q5: Using First derivate of the function to get a even better estimate.

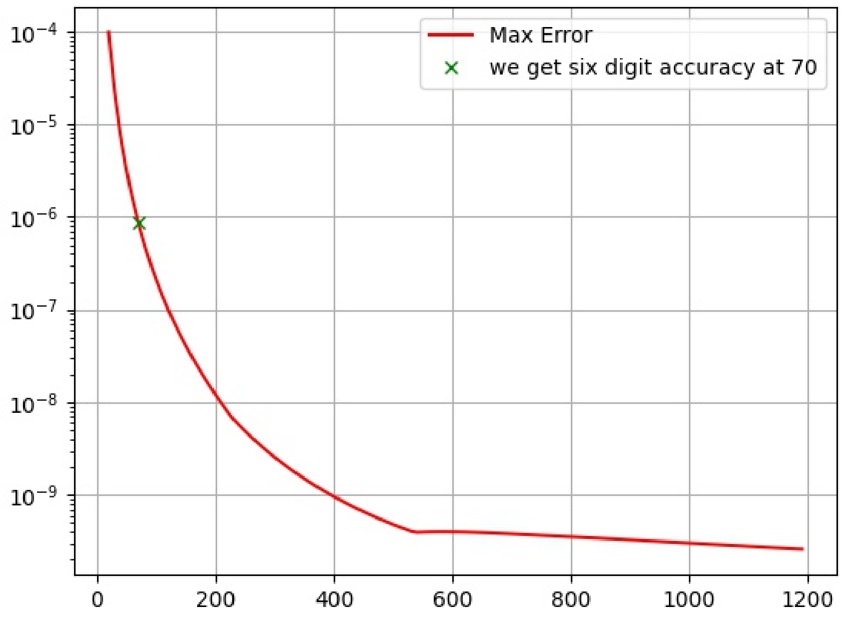


Figure: order of error vs no. of uniformly spaced points.

Observations

- This graph shows significant improvement over any previous method as we get to six digit accuracy at 70 points.
- The error drops significantly as the number of sampled points increase but it reaches a point where the error starts dropping more gradually as the points increase.

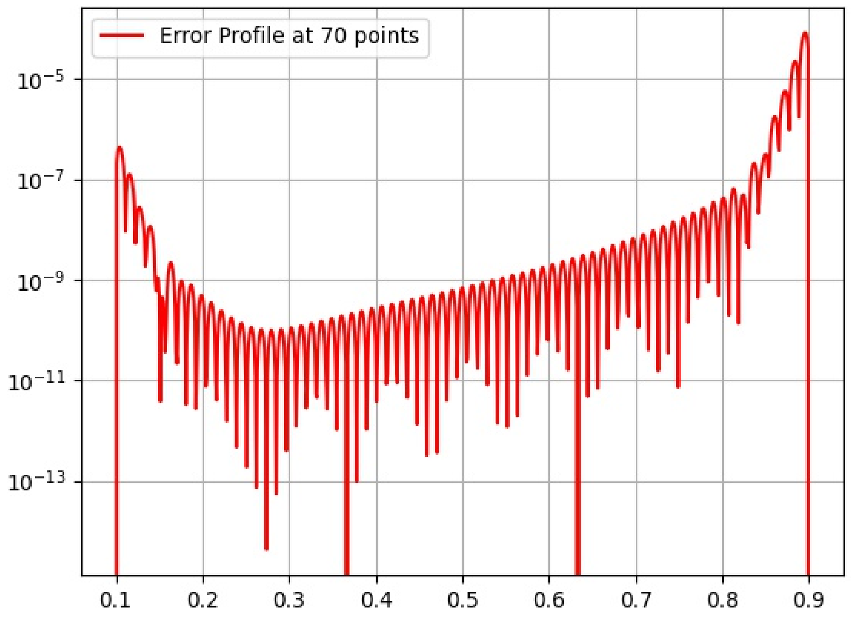
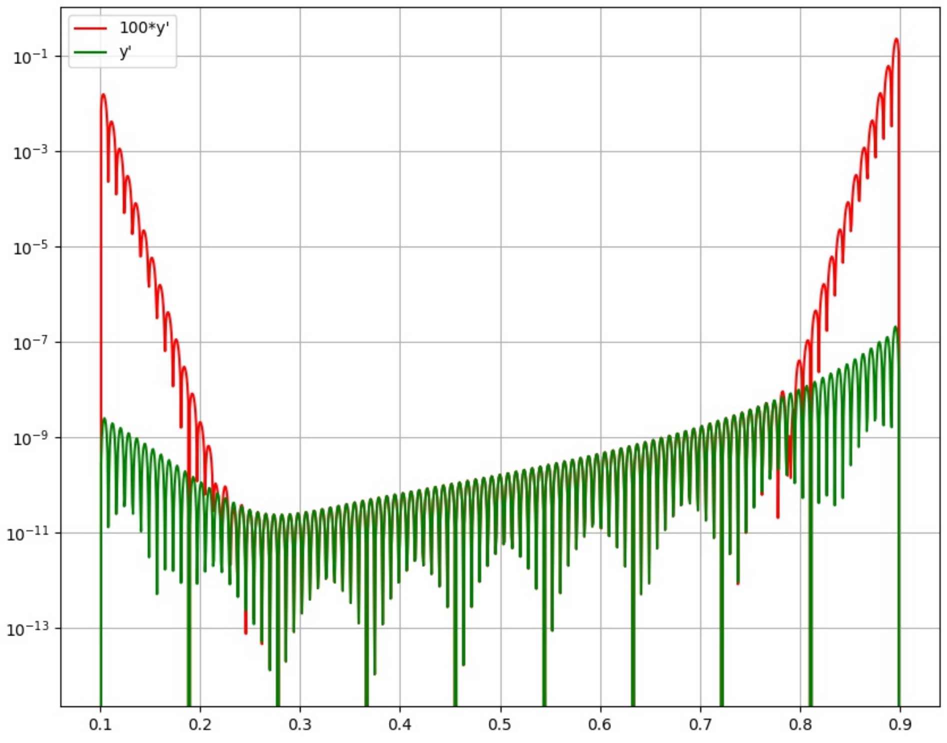


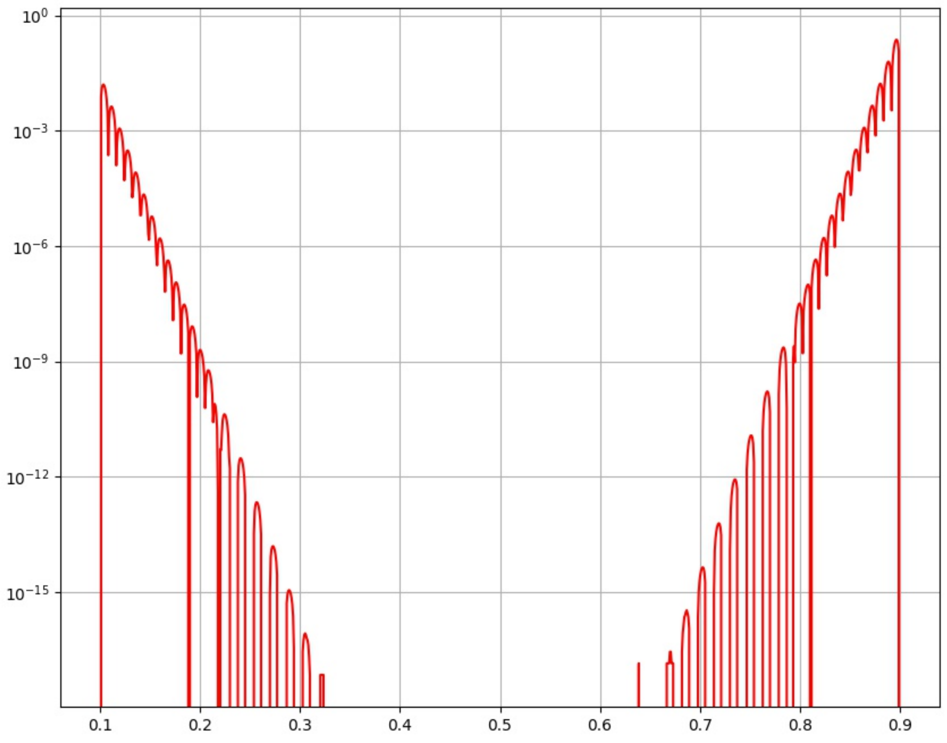
Figure: order of error vs no. of uniformly spaced points.

- We can see in the graph in the above that this scheme follows similar pattern as well which is the error is minimal in between and increases drastically around edges which can be predicted from the theory.

Q6: Using 100 times the first derivate to see how the error varies.



- Giving a fault estimate of y_1'' and y_n'' makes the error go up by an order of 10^8 at the edges this error gradually declines towards the center of the graph where the spline starts following the actual estimate graph.
- This shows the self healing properties of a cubic spline that even with a bad estimate it got to a decent order of accuracy in the middle of the graph.



- This is the graph between the error between the 2 plot shown above. As we can see the error declines significantly and goes away entirely from 0.32 to 0.64.

Q7: Using Non-Uniform Scaling trying to reduce the required number of points to get six digit or better accuracy.

Looking at the error profile we can see that the error is max around the boundary and even more near the right edge of the boundary. My idea is to divide the total number of points in parts and taking smaller steps for points with more error and bigger step for points which less error this can significantly reduce the number of points to get a accurate answer.

```
h=np.linspace(0.1,0.9,5)
parts=10
xa=np.concatenate([np.linspace(0.1,h[2],int(3*i/parts),endpoint=False),
np.linspace(h[2],h[3],int(2*i/parts),endpoint=False),
np.linspace(h[3],h[-1],int(5*i/parts))])
fx=[(w**(1+special.j0(w)))/(np.sqrt(1-w+(100*(w**2))-(100*(w**3)))) for w in xa]
```

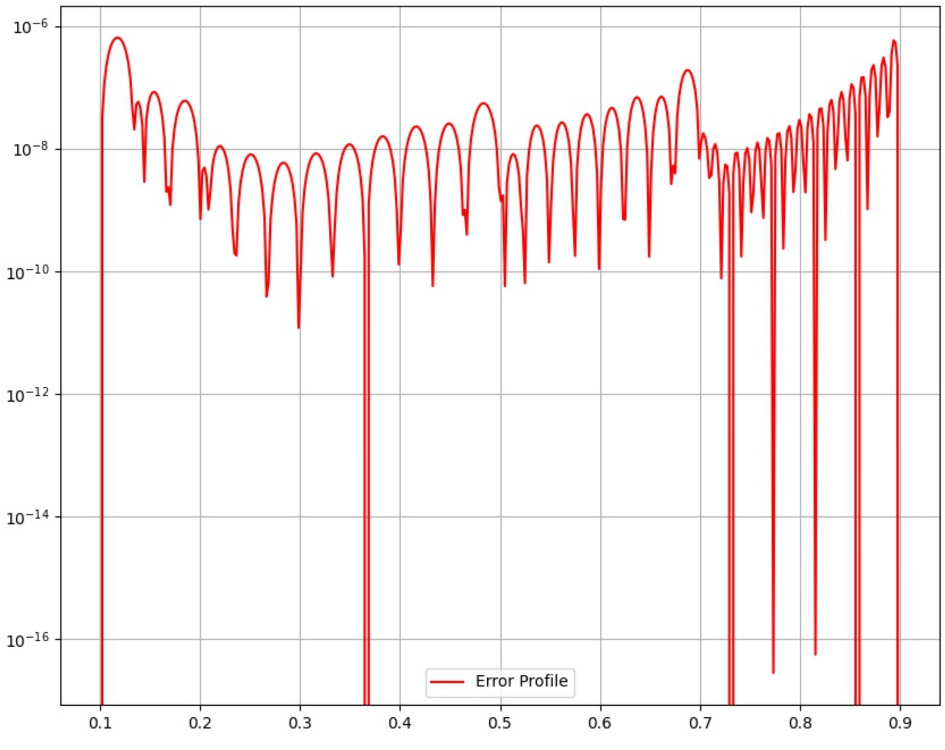


Figure: Error Profile for Natural Spline with non-uniform spacing.

- This graph above is derived when we divide the number of points by 10 and use the points $\frac{3}{10}$ for beginning , $\frac{2}{10}$ for the mid section and $\frac{5}{10}$ for the edge at right side.
- I tried many iteration of this idea with different weightage for each part and best i could manage with this idea is this error profile we see above.

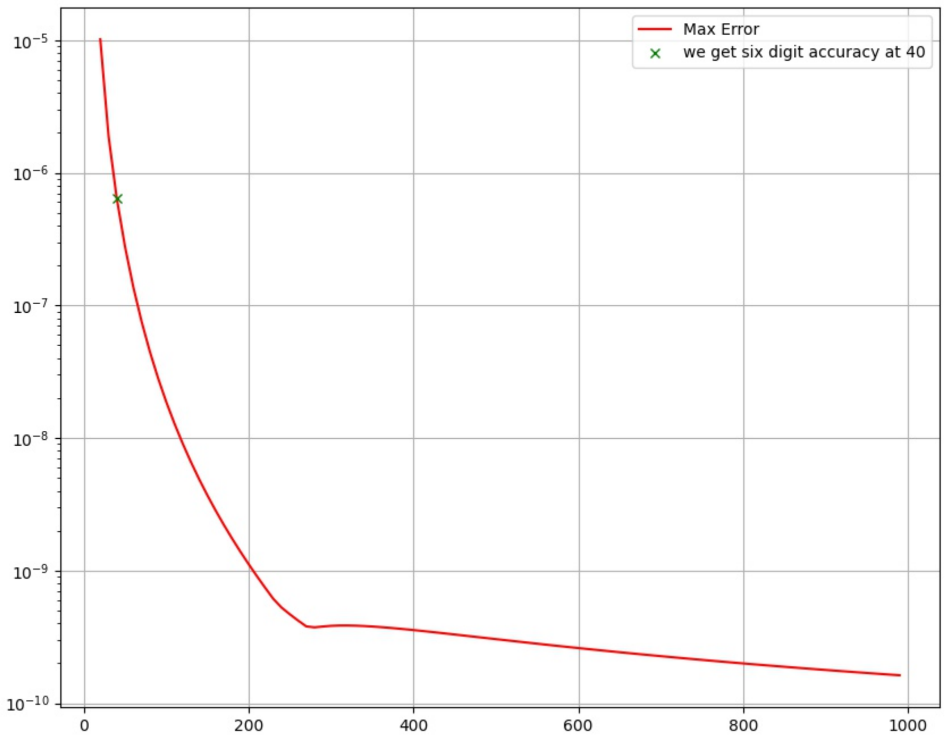


Figure: order of error vs no. of non-uniformly spaced points.

- This allows us to get six digit of accuracy at roughly 40 points better than previous estimate at about 70 points.

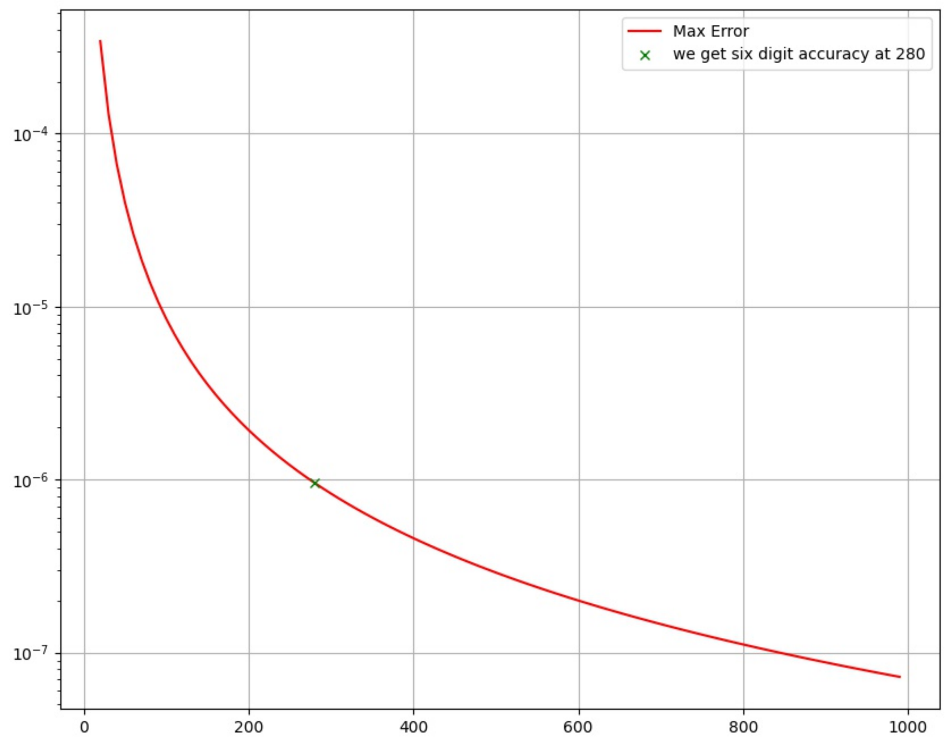


Figure: order of error vs no. of non-uniformly spaced points.

- Based on the same theory i tried it for not a knot condition and could get 6 digit accuracy from 280 points from a previous best of 550 points.

Q8: 2-D interpolation Surface plot for the function $\sin(\pi x)\cos(\pi y)$:

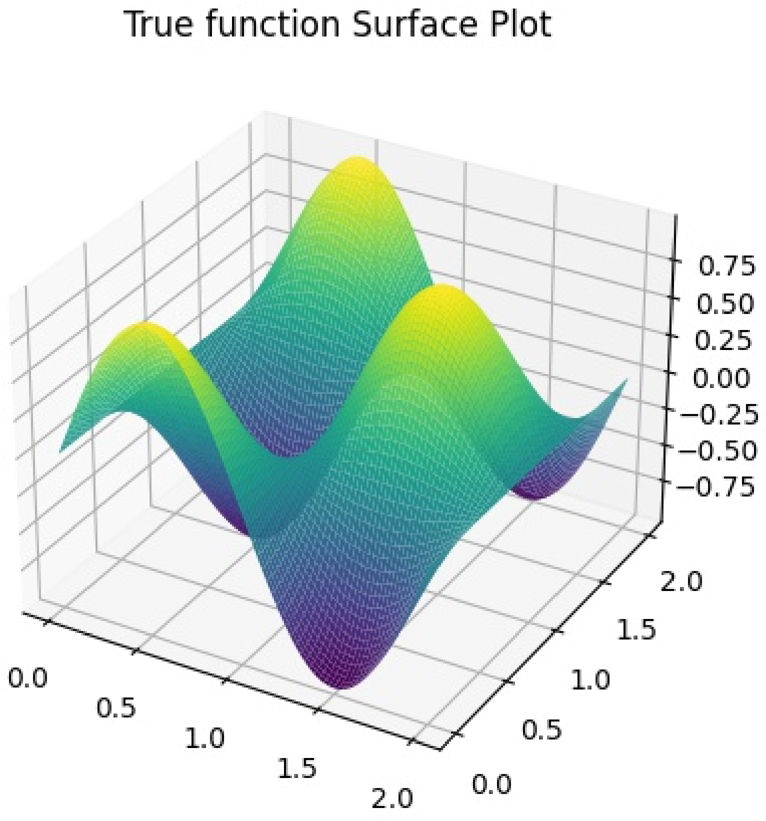


Figure: Surface plot of the original Function.

a): Bi-Linear Interpolation for the function for 50*50 sampled points across 0 to 2.

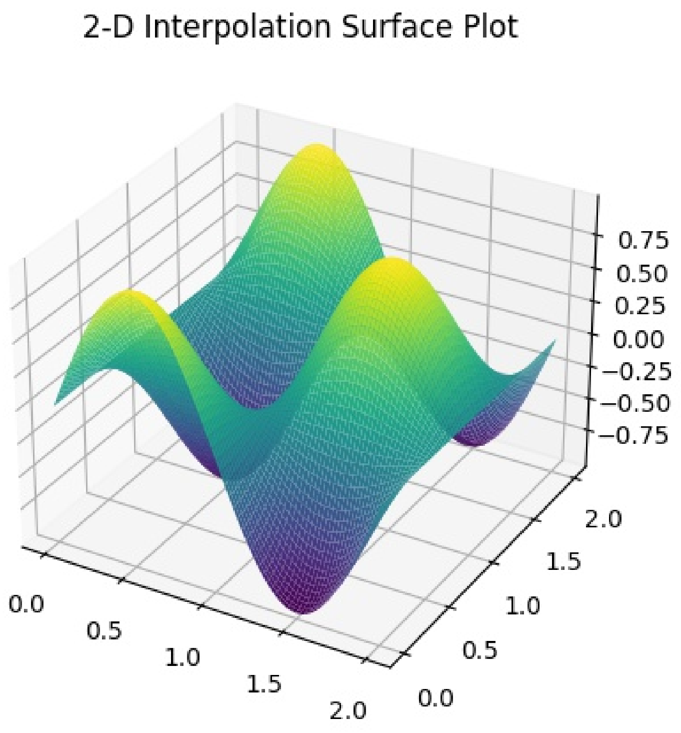


Figure: Plot of the function using Bi-Linear Interpolation

- Using Bi-Linear interpolation the we can see we get a good estimate of the original function looking at the error profile below we can also observe the how much accuracy do we have.

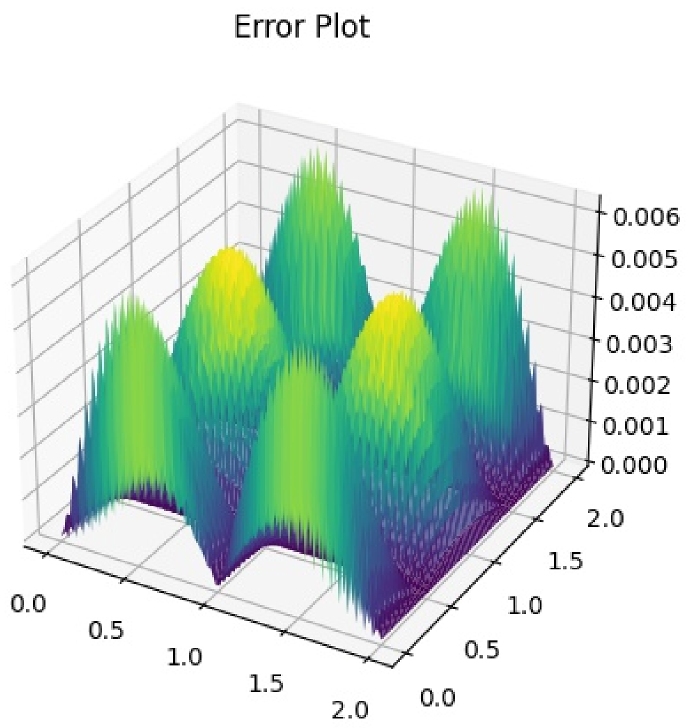


Figure: Error Profile for Bi-Linear Interpolation.

- As we know from theory the cost of 2-D interpolation is significantly higher than that of 1-D interpolation so i could only manage to run this for 200 points sampling and looking at the error graph below we can see some similarity between Linear interpolation and Bi-Linear Interpolation but the best error we can manage is 3 digit accuracy.

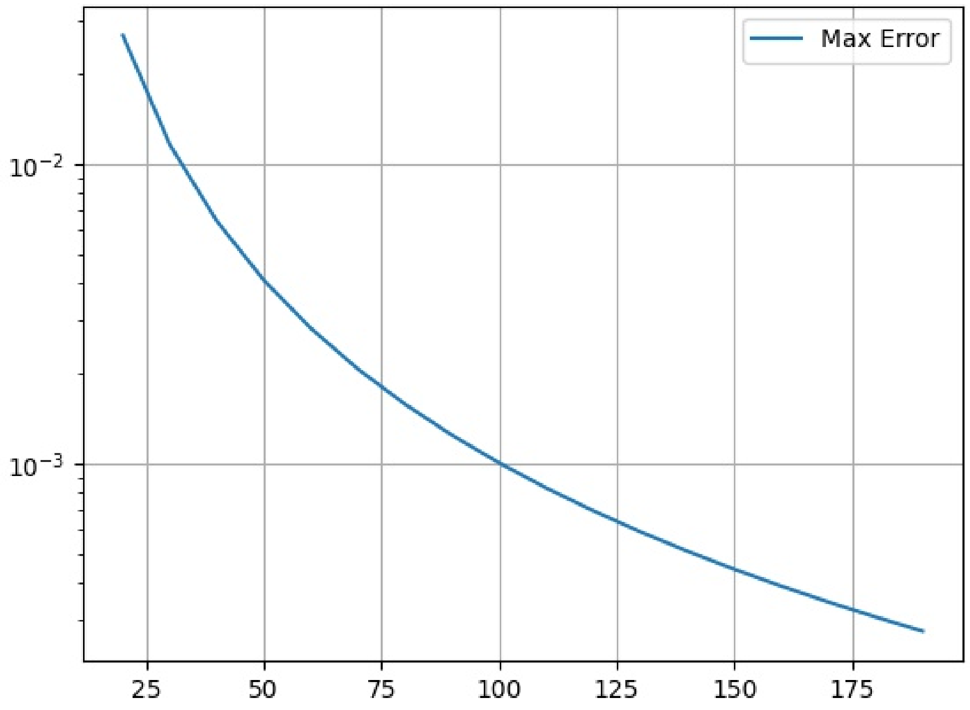


Figure: Error varying with increasing number of sample points.

- The trend seems to similar but the accuracy is not there and with expensive cost it makes the solution infeasible.

a): Bi-Cubic Interpolation for the function for 50*50 sampled points across 0 to 2.

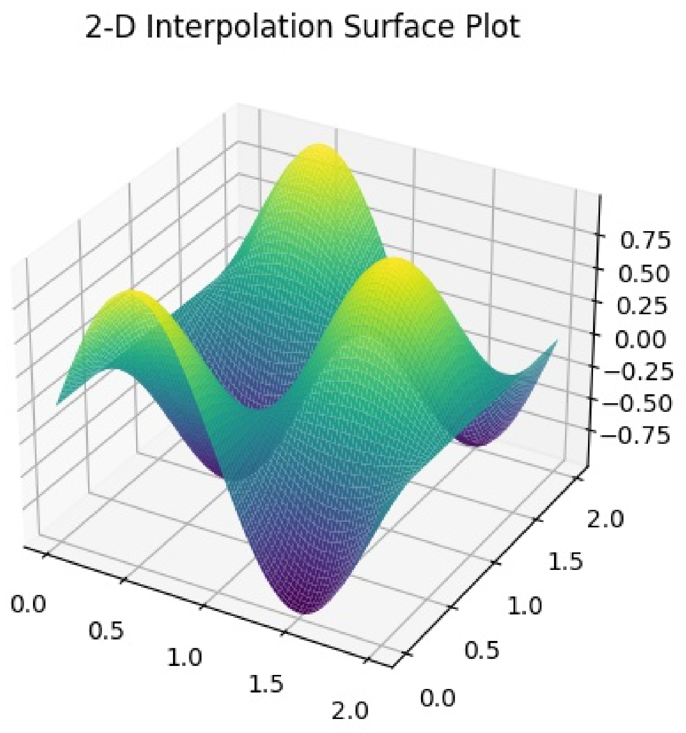


Figure: Plot of the function using Bi-Cubic Interpolation

- Using Bi-Cubic interpolation the we can see we get a good estimate of the original function this looks similar to Bi-Linear Interpolation but looking at the error profile below we can observer that we get a better accuracy for this approach.

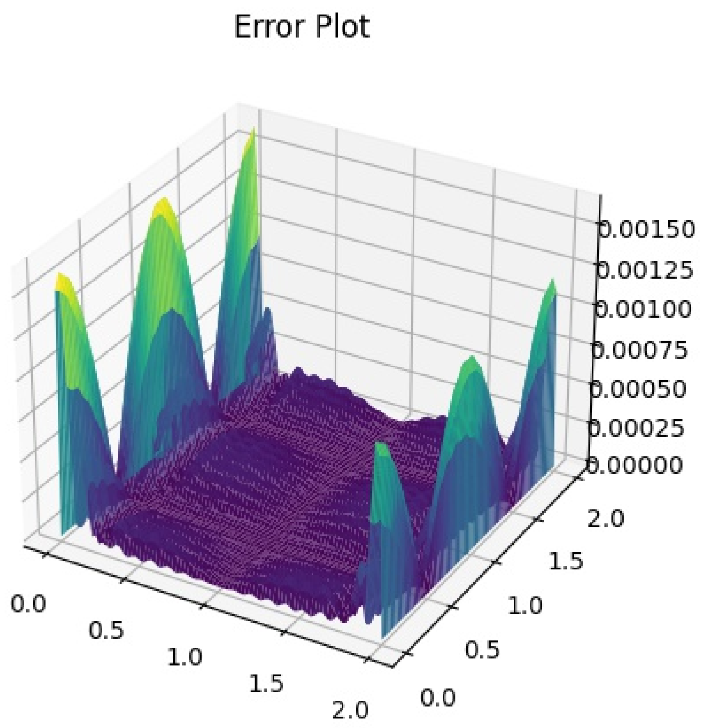


Figure: Error Profile for Bi-Cubic Interpolation.

- The error around the edges is still significant and comparable to Bi-Linear Interpolation but the error profile in the middle we can see is substantially lower so the accuracy we now get is way better than Bi-Linear Interpolation.

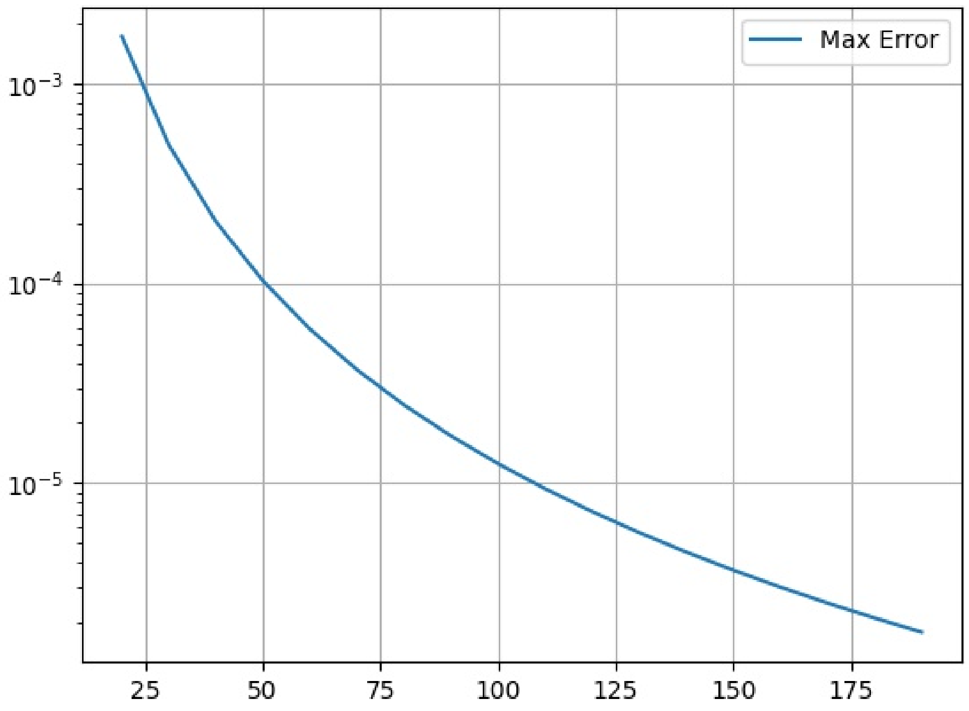


Figure: Max error vs increasing number of sample points.

- The Error plot as the number of points increases shows that trend is similar to Cubic Spline and we can get to 5 digit accuracy around 130 points but the computation cost is significantly higher which makes it harder to scale.