Interpolation

Preston Seligman
(Dated: February 21, 2025)

Overview

In this assignment, various Lagrange interpolation methods were examined and compared. These include Global interpolation, Linear interpolation, and Cubic interpolation. Interpolation methods were performed on scattering data to determine cross-sectional width, as well as over a pre-specified function.

Theoretical Framework

For N points of data of the form (x_i, y_i), Lagrange interpolation will determine some polynomial function of degree $N-1$ such that the polynomial function will pass through each defined data point. Here, the polynomial function $f(x)$ should be a close approximation of the true function for the data/system. The general form of the Lagrange interpolation function $f(x)$ can be defined as

$$\lambda_i(x) = \prod_{j=1}^{N} \left(\frac{x - x_j}{x_i - x_j} \right), j \neq i$$

$$f(x) = \sum_{i}^{N-1} y_i \lambda_i(x)$$

For Global interpolation, this polynomial is calculated for all N data points, for Linear interpolation N=2, and for Cubic interpolation N=4. The error of interpolation is generally $f^N(\delta x)^N$. However, the error can vary between different interpolation methods, meaning that for different functions it may be advantageous to use one method over another.

a. Choosing Interpolation Method Global is beneficial as it interpolates over all data points and generates a continuous, smooth function. However, there can be issues when utilizing global interpolation, most notably of unnatural oscillations such as Runge's phenomena. Linear interpolation is easy to perform and does not have any unphysical oscillations, however it's derivative is discontinuous, and it may produce greater error if the true function is not linear. Cubic interpolation provides a happy medium; it has fewer oscillations and is reasonably smooth, however still has discontinuous derivatives at each junction. However, it can provide a better approximation than linear in cases where the true function is not linear.

Code

```
import matplotlib.pyplot as plt
import numpy as np

def legendrepol (x,beg,finish):           # poly interpolation at x
    y = 0.                                # using input points from beg to finish
    for  i in range(beg,finish+1):
       lambd = 1.0;
       for j in range(beg,finish+1):
           if i != j:                        #Lagrange polynom formed here
              lambd = lambd * ((x - xin[j])/(xin[i] - xin[j]))
       y += yin[i] * lambd
    return y

NMAX = 100  # max number of input points

xin = np.zeros(NMAX)  # each is array of length NMAX, all elements set to zero
yin = np.zeros(NMAX)
```

```
# inputfile = open("lagrange.dat","r")  # read in the input x,y values
# r = inputfile.readlines()  # read the whole file into list (one item per line)
# inputfile.close()
        # input has the form: x0 y0
        #                     x1 y1
        #                     ...
m = 0
for i in np.linspace(-1,1, num=21):
    xin[m] = i
    yin[m] = 1.0/(1.0 + (25*i**2))
    print(xin[m],yin[m])
    m += 1



# m = 0
# for line in r:
#     #print(line)
#     s = line.split() # split line and split into list of items(assume items separated
    by spaces)
#     xin[m] = s[0] # first number in each line is the x value
#     yin[m] = s[1]
#     print(xin[m],yin[m])
#     m+=1          # m is total number of input data points
#                   # will be stored in xin[0]..xin[m-1],yin[0].yin[m-1]

xvalues=range(0,201,1)  # At least 200 Steps needed to fully interpolate using our
    function. More will add fidelity
yval_glo = []
yval_lin = []
yval_cub = []

for x in xvalues:
    for i in range(0, m-1):
        if xin[i] <= x and x <= xin[i+1]:
            firstpoint=i
    cubepoint = firstpoint
    if cubepoint > m-4:
         cubepoint = m-4        #Need highest cubpoint to be cubepoint+3=8, so m-4=5 for
             this range
    yval_lin.append(legendrepol(x,firstpoint,firstpoint+1))
    yval_glo.append(legendrepol(x,0,m-1))
    yval_cub.append(legendrepol(x,cubepoint,cubepoint+3))



# for x in xvalues:          # now interpolate
#     yvalues.append(legendrepol(x,firstpoint,firstpoint+numpoints-1))

plt.plot(xin[0:m],yin[0:m],"o",label="Scattering Points")
plt.plot(xvalues,yval_glo,color="orange", linewidth=1, linestyle="dashed", label="Global
    Interpolation")
plt.plot(xvalues,yval_lin, color="green", linewidth=2, linestyle="solid", label="Linear
    Interpolation")
plt.plot(xvalues,yval_cub, color="purple", linewidth=2, linestyle="dashed", label="Cubic
    Interpolation")
plt.legend(loc="upper right")
plt.xlabel("Energy (MeV)")
plt.ylabel("Cross Sectional Area")
plt.title("Lagrange Interpolation")
plt.axis([-1.00, 1.00, -0.2, 1.1])
plt.show()
```

Problems

Problem 1

First, a global interpolation of the data was performed, where interpolation was extrapolated up to 220MeV. The plot for this interpolation can be seen in Figure 1.

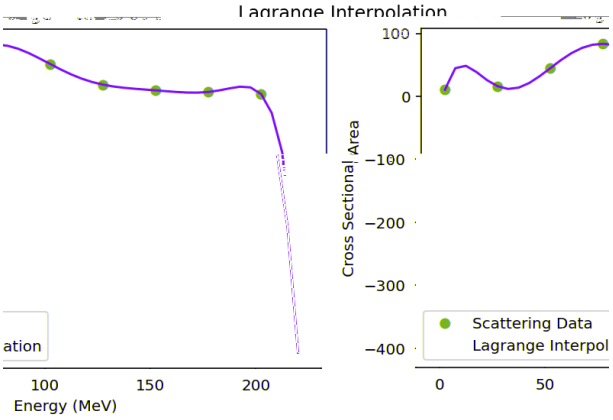


FIG. 1: Global Interpolation Extrapolated to 220MeV

Problem 2

Interpolation was repeated for $0 \leq x \leq 200$ using Linear interpolation. The code for linear interpolation is

```
for x in xvalues:
    for i in range(0, m-1):
        if xin[i] <= x and x <= xin[i+1]:
            firstpoint=i
yval_lin.append(legendrepol(x,firstpoint,firstpoint+1))
```

Here, the number of points being interpolated over is $N = 2$ giving a linear function between all data points. The plot of this interpolation is given in Figure 2. The oscillations present in the global interpolation are no longer present for

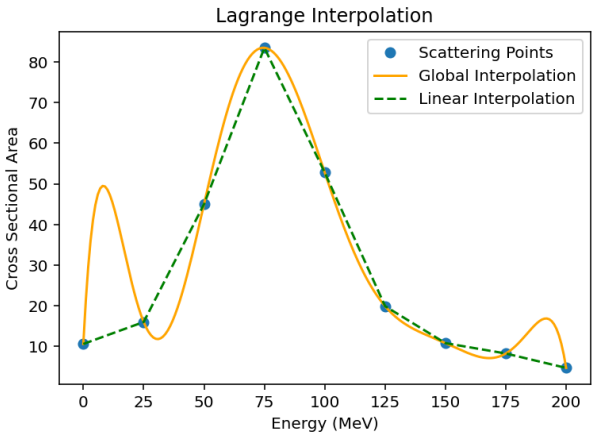


FIG. 2: Linear Interpolation

the linear interpolation. This is especially apparent near the minimum and maximum values for x.

Problem 3

Cubic interpolation was performed over the same range. The code for the cubic interpolation is

```
for x in xvalues:
    for i in range(0, m-1):
        if xin[i] <= x and x <= xin[i+1]:
            firstpoint=i
    cubepoint = firstpoint
    if cubepoint > m-4:
        cubepoint = m-4          #Need highest cubpoint to be cubepoint+3=8, so m-4=5 for
            this range
    yval_lin.append(legendrepol(x,firstpoint,firstpoint+1))
    yval_glo.append(legendrepol(x,0,m-1))
    yval_cub.append(legendrepol(x,cubepoint,cubepoint+3))
```

For the extreme values, the firstpoint has to be set such that $N+3$ is within the range N. That is to say, the cubepoint cannot be greater than $N-4$ as this will mean cubic interpolation is being performed on data outside the range. This check is performed in the function above. The result of cubic interpolation can be seen in Figure 3.

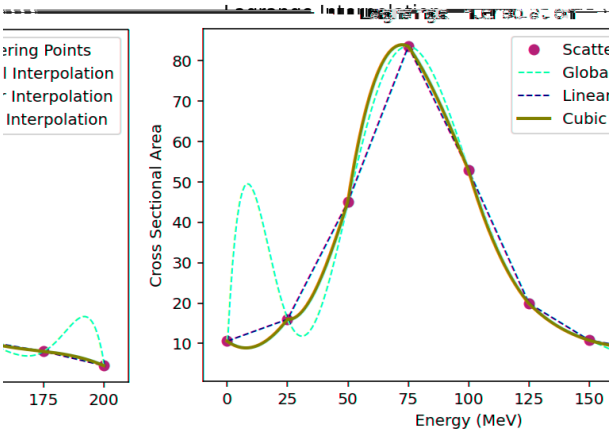
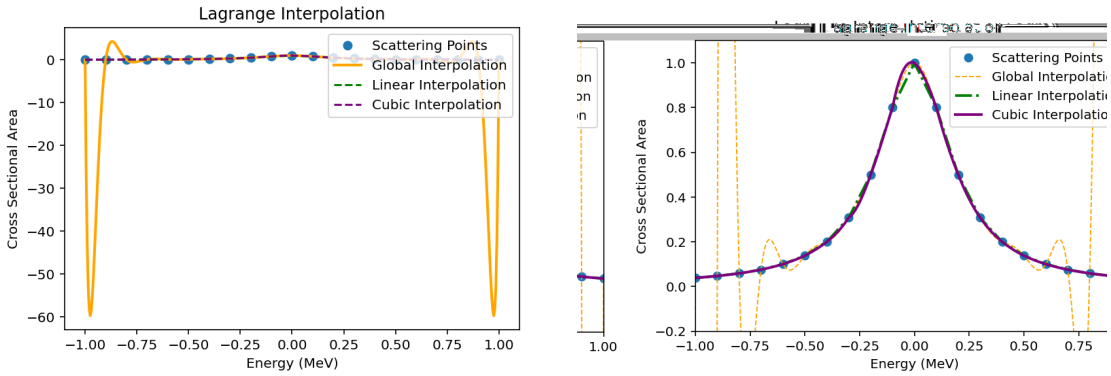


FIG. 3: Cubic Interpolation

Problem 4

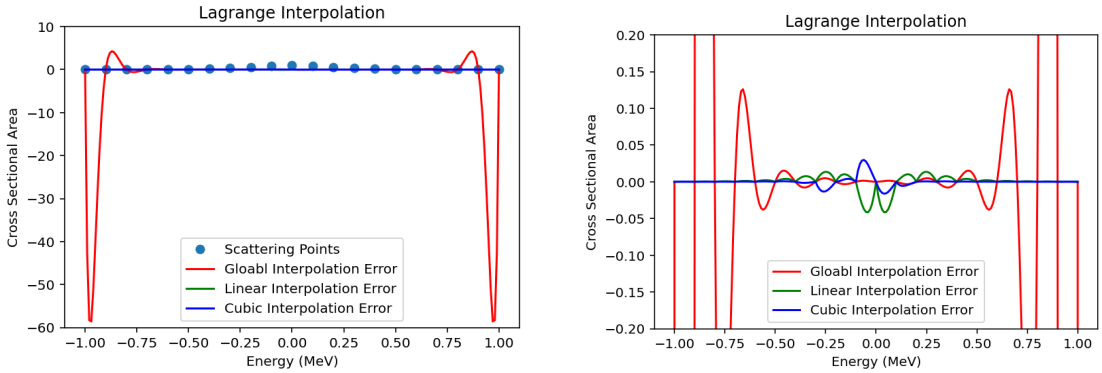
These three methods of interpolation were repeated for the function $\frac{1}{1+25x^2}$ over the range $[-1,1]$ with 21 equally spaced intervals. The code for this function is:

```
m = 0
for i in np.linspace(-1,1, num=21):
    xin[m] = i
    yin[m] = 1.0/(1.0 + (25*i**2))
    print(xin[m],yin[m])
    m += 1

xvalues=np.linspace(-1,1,21)

plt.plot(xvalues, yval_glo-1/(1+25*xvalues**2), linewidth=1.5, label="Gloabl
    Interpolation Error", color="red")
plt.plot(xvalues, yval_lin-1/(1+25*xvalues**2), linewidth=1.5, label="Linear
    Interpolation Error", color="green")
plt.plot(xvalues, yval_cub-1/(1+25*xvalues**2), linewidth=1.5, label="Cubic
    Interpolation Error", color="blue")
```



(a) Full Range (b) Zoomed in to Show Linear and Cubic

FIG. 4: Results for function $\frac{1}{1+25x^2}$



(a) Full Range (b) Zoomed in to Show Linear and Cubic

FIG. 5: Results for function $\frac{1}{1+25x^2}$

Results can be seen in Figure 4. Error in the functions can be seen in Figure 5.

As can be seen, global interpolation has a large amount of oscillations. By examining the error of the functions, it can be seen that global interpolation has the lowest error near the center, however the error increases drastically as x moves further away from 0. Cubic interpolation has the lowest overall error and best approximates the data.