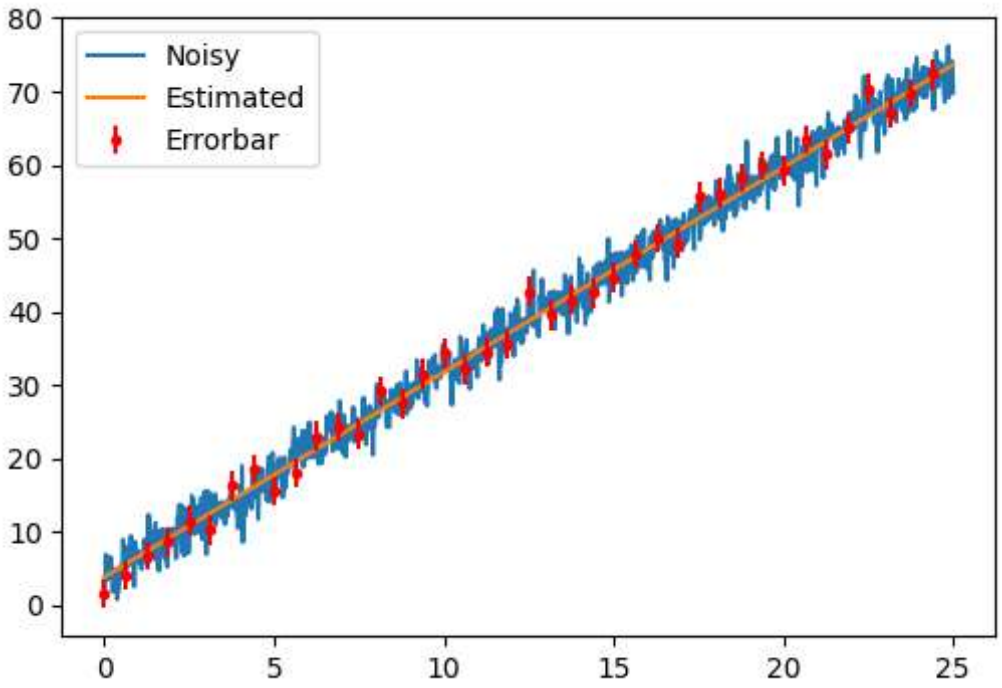
ASSIGNMENT 3 - Curve Fitting

Dataset 1 (Estimation of Straight line from noisy function)

In [ ]:
```
import csv
import numpy as np
import matplotlib.pyplot as plt
x=[]            #initialisation of x
y=[]            #initialisation of y
with open("D:\\vscode\\a3-data\\a3-data\\dataset1.txt", "r") as file:
    data = csv.reader(file,delimiter=' ')
    for row in data:    #reads each line into x,y
        x.append(float(row[0]))
        y.append(float(row[1]))
x=np.array(x)
y=np.array(y)
M = np.column_stack([x, np.ones(len(x))])   #construction of M matrix
(p1, p2), _, _, _ = np.linalg.lstsq(M, y, rcond=None)   #least square fitting
print(f"The estimated equation is {p1} x + {p2}")
y1=[p1*i+p2 for i in x]                     # estimated values
noise=(y1-y)                                #noise extraction
plt.plot(x,y)                               #plotting original data
plt.plot(x,y1)                              #plotting estimated data
plt.errorbar(x[::25], y[::25], np.std(noise), fmt='r.')
plt.legend(["Noisy","Estimated","Errorbar"],loc='upper left')
```

```
The estimated equation is 2.791124245414918 x + 3.8488001014307427
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x2221a38d130>
```



Construction of M

Data is extracted using the CSV format. The initial step involves appending the first number in each line to the variable `x` and the second number to the variable `y` . Then, matrix `M` is created by column-stacking the `x` values along with a column of ones. This modified matrix is then passed to the `np.linalg.lstsq()` . This procedure could have been implemented manually as follows:

$$\mathbf{y} \equiv \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{pmatrix} \begin{pmatrix} m \\ c \end{pmatrix} \equiv \mathbf{M}\mathbf{p}$$

Now we know y and M. We need to solve for p. If it were a square matrix, we could directly use gaussian elimination. So we multiply by transposed matrix to convert to square matrix. Then it could be solved through linalg.solve() which is implemented by gaussian elimination.

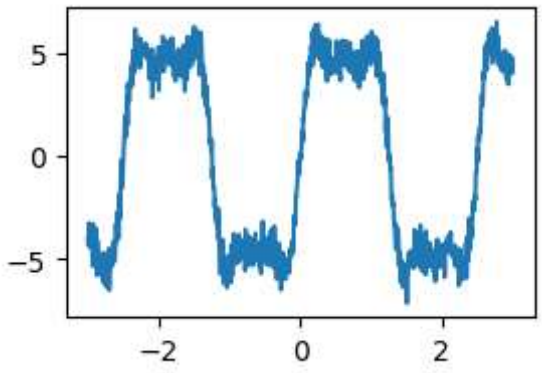
$$\mathbf{M^T y} = (\mathbf{M^T M})\,\mathbf{p}$$

Plotting

- `y1` , the estimated line is computed and plotted.
- The `plt.errorbar()` is used to plot errorbar for 1 in 25 values(A matrix `noise` is created by subtracting estimated and original data)
- `plt.legend()` is used to describe legend

Dataset 2 (Finding amplitude of sine waves from their sum corrupted with noise)

In [ ]:
```
t=[]            #initialisation of t
fnc=[]          #initialisation of fnc
with open("D:\\vscode\\a3-data\\a3-data\\dataset2.txt", "r") as file:
    data = csv.reader(file,delimiter=' ')
    for row in data:    #reading each line
        t.append(float(row[0]))
        fnc.append(float(row[1]))
plt.plot(t,fnc)     #plotting original data
t=np.array(t)
fnc=np.array(fnc)
```



Data is extracted using the CSV format. The first number in each line is appended to the variable `t` and the second number to the variable `fnc` .

Periodicity estimate

The plot clearly shows that a phase shift of π leads to the same values.

In [ ]:
```
def period(t,y):
    m=-np.inf
    maxy=(max(y))
    find=0
    ini = 0  # Initialize the start index
    end = 0  # Initialize the end index
    for i in range(len(y)):
        if(find==0):
            if(abs(y[i]-maxy)<=0.5 and y[i]>y[i - 1] and y[i+1]<y[i]):
                m=y[i]
                ini=i       #finding first maximum
                find=1
        elif(find==1):
            if abs(y[i]+m)<0.001:
                end=i       #finding first minimum, negative of 1st maximum
                break
    return 2*(t[end]-t[ini])
```

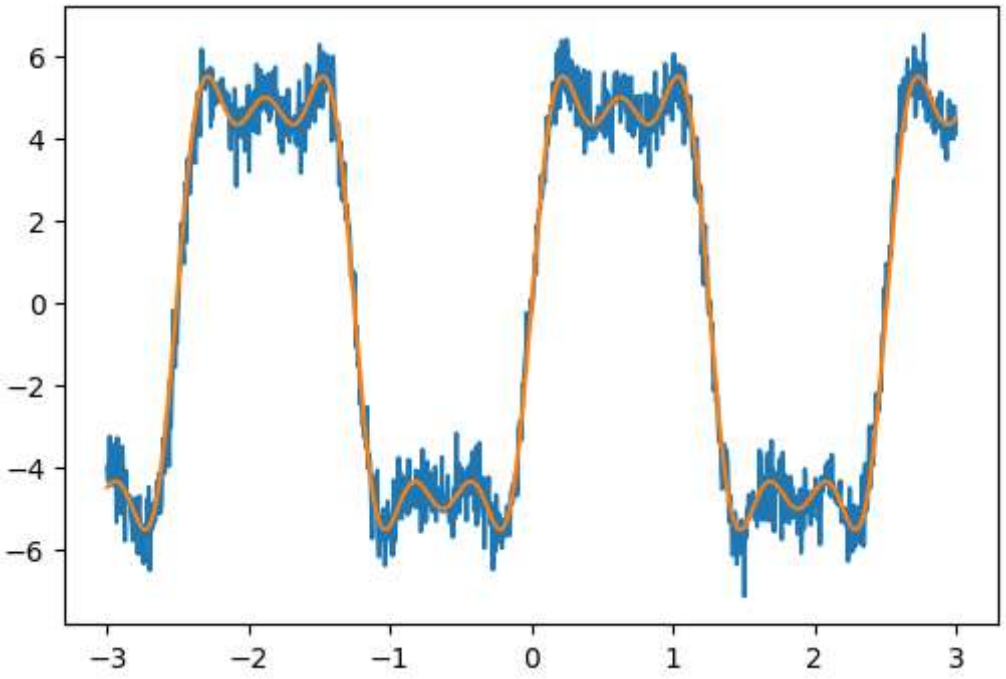
- We know that sin(π+x)=-sinx. Only sin(2n+1)x follows this pattern. So I have odd multiples of certain base frequency

  Since it is symmetric about points that differ by T/2, we the first local maxima(as there are multiple peaks indicating a minimum of 3 sinusoids of different frequencies). Then we find the first minima that corresponds to negative of maxima and find half time-period. (The minima should correspond to negative of maxima as some oscillating curves will have local minima not corresponding to required minima)

  I have set condition for the maxima to not vary much from global maxima(This is because the noise would produce fake local maximas and we know from graph that first maxima is close to max(y). A more general method would curve smoothening followed by finding maxima)

In [ ]:
```
T=period(t,fnc)         #finding time period
print(f"The time period is {T}")
t1=(2*np.pi *t/T)
for c in range(2,20,3):
    M1 = np.column_stack([np.sin(t1),np.sin(3*t1),np.sin(c*t1)]) #forming M matrix
    p, _, _, _ = np.linalg.lstsq(M1, fnc, rcond=None)
    fnc1=p[0]*np.sin(t1)+p[1]*np.sin(3*t1)+p[2]*np.sin(c*t1)
    n=(fnc1-fnc)
    if(np.std(n)<=0.6):  #ideal frequencies have std of about 0.5, others around 0.9
        break
af=2*np.pi/T
print(f"""The estimated parameters are: {p[0]:.3f} sin(t*{af:.3f})
      +{p[1]:.3f} sin(t*{3*af:.3f} +{p[2]:.3f} sin(t*{c*af:.3f})""")
plt.plot(t,fnc,t,fnc1)
print(f"The standard deviation in using  is np.linalg.lstsq is {np.std(n)}")
```

```
The time period is 2.5105105105105108
The estimated parameters are: 6.009 sin(t*2.503)
     +1.997 sin(t*7.508 +0.989 sin(t*12.514)
The standard deviation in using  is np.linalg.lstsq is 0.5183058212251715
```



Construction of M

Here I procede by assuming values for third frequency and then doing entire computation and checking if error is below a certain criteria.

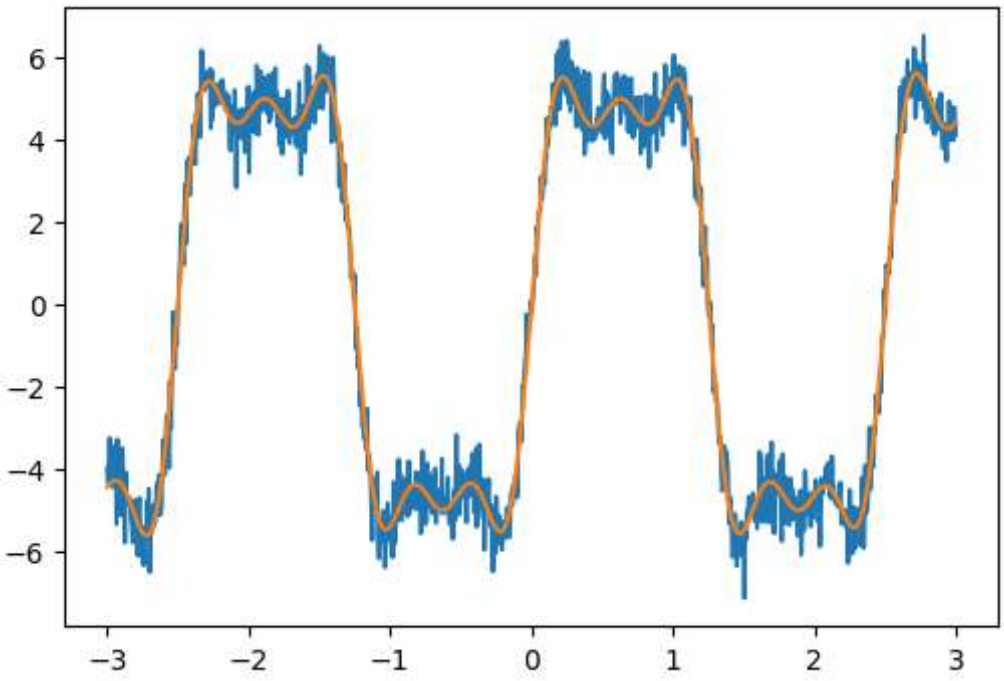
M is constructed using np.linalg.lstsq after column stacking ω1.t,ω2.t,ω3.t where ω1=2π/T,ω2=3.2π/T

(We know that LCM(t1,t2,t3)=T and one time period is thrice the other. We assume t1=T,t2=T/3 and look for t3=T/c through iteration where c is odd

A more general assumption would be t1=T/b,t2=T/3b,t3=T/c. But given that all the amplitudes are unknown making the frequencies also unknown isn't the best option. However in curve_fit we could implement it)

In [ ]:
```
from scipy.optimize import curve_fit
def sinfunc(t, p1, p2,p3,c,b):
    return p1* np.sin(b*t1)+p2*np.sin(3*b*t1)+p3* np.sin(c*t1)
guess=[10,2,3,5,1]
(sp1, sp2,sp3,c1,b1), _ = curve_fit(sinfunc, t, fnc, guess)
print(f"""The estimated parameters are: {sp1:.3f} sin(t*{b1*af}) +
       {sp2:.3f} sin(t*{3*b1*af}+{sp3:.3f} sin(t*{c1*af}""")
# Regenerate data
sest = sinfunc(t, sp1, sp2,sp3,c1,b1)
plt.plot(t, fnc, t, sest)
y=fnc
n=[(sest[i]-y[i]) for i in range(len(sest))]
print(f"The standard deviation in using scipy.optimize.curve_fit() is {np.std(n)}")
```

```
The estimated parameters are: 6.012 sin(t*2.5148341716933134) +
       1.996 sin(t*7.544502515079941+0.980 sin(t*12.532036421030677
The standard deviation in using scipy.optimize.curve_fit() is 0.5008948424546209
```



Using `curve_fit`

Curve fit gives a curve with lesser error though it needs an ideal starting guess to work well. The third frequency computed from `curve_fit` and `lstsq` are same atleast to first decimal digit. The amplitudes are also matching well

Fourier Transform

I later colaborated with EE22B143 on fourier transforming the function. Here I have increased data points by reconstructing y to get accurate fourier.

In [ ]:
```
def fourier(x, y):
    l=len(x)
    f = np.arange(-20, 20, 0.1)  # Create an array of frequencies
    transform = np.zeros(len(f))  # Initialize the array for the Fourier transform
    for k in range(len(f)):
        for i in range(l):
            transform[k] += (y[i] * np.sin(f[k] * x[i]))/l  # Compute the Fourier transf
    return f, transform
```

In [ ]:
```
x = []
y = []
for i in range(len(t)):
    if 0 <= t[i] <= T:
        x.append(t[i])
        y.append(fnc[i])
    elif t[i] > T:
        break
x1 = []
y1 = []
for n in range(-5, 5):
```

```
    x1.extend([i + n * T for i in x])
    y1.extend(y)
f, transform = fourier(x1, y1)
plt.plot(f, transform)
plt.title('Fourier Transform')
plt.show()
```

This confirms that frequencies of 2.5,7.5 and 12.5 occur with amplitudes 6,2 and 1 respectively

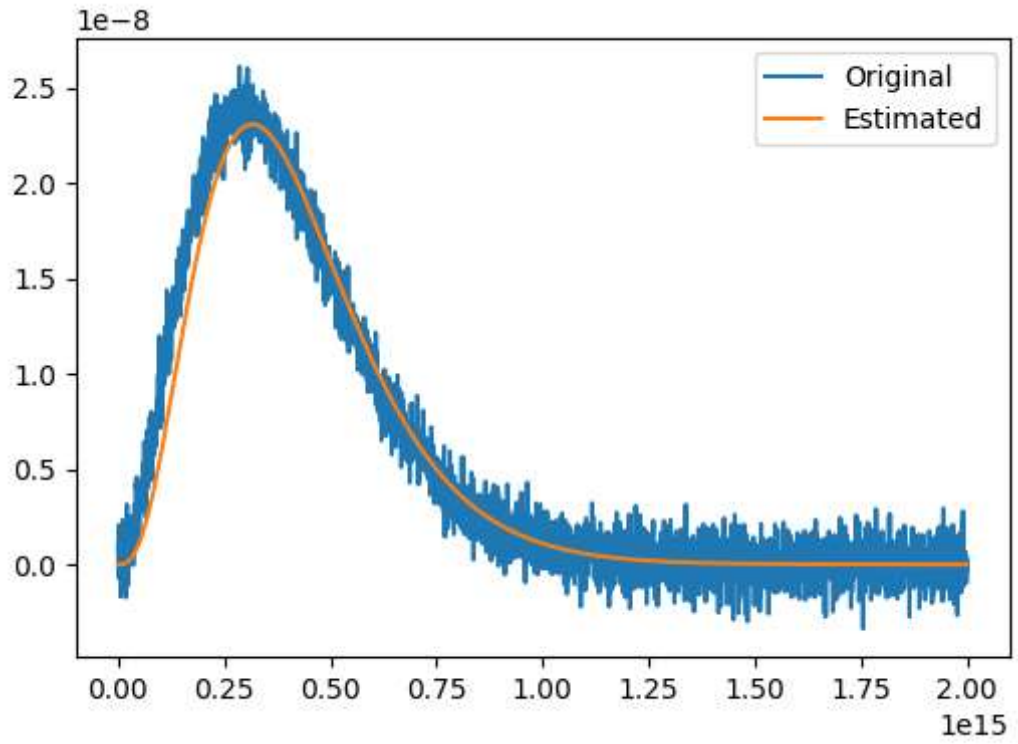
Dataset 3(Non-linear curve fitting)

First solution:

In [ ]:
```
kB = 1.3806452e-23      #constant parameters
h = 6.62607015e-34
c = 299792458
f = []
y1 = []
with open("D:\\vscode\\a3-data\\a3-data\\dataset3.txt", "r") as file:
    data = csv.reader(file, delimiter=' ')
    for row in data:  #reading data
        f.append(float(row[0]))
        y1.append(float(row[1]))
f=np.array(f)
y1=np.array(y1)
def plank(f, T):    #radiation intensity function
    return (2 * h * (f**3) / ((c**2) * np.exp((h * f / (kB * T)))-1))
guess = 2e3 #initial guess
sp1, _ = curve_fit(plank, f, y1,guess) #curve fitting
print(f"{sp1}")
y2 = plank(f, sp1)          #estimated values
plt.plot(f, y1)             #plotting
plt.plot(f, y2)
plt.legend(["Original","Estimated"])
```

```
[5050.40275679]
```

Out[ ]: <matplotlib.legend.Legend at 0x2221a176d30>



The dataset is extracted using the CSV format. The constants values are supplied, and the `curve_fit` function is invoked.

- For initial assumptions within the range of 14 to 2.44e13, a temperature estimate of 5050 K is obtained.
- Assumptions below this range result in a temperature estimate same as guess, as the value of h * x / (kB * T) becomes excessively high, leading to its exponent in denominator being high and consequently, a small value of radiation and a dead straight line is plotted for radiation. (If no assumption is specified 1K gets printed as algorithm starts with guess of 1)
- Conversely, assumptions exceeding this range cause the initial guess to be rejected, and the radiation plot exhibits cubic growth with frequency, as h * x / (kB * T) is excessively small(e^0 = 1), denominator becoming almost constant
- Also setting bounds decreses chances of getting right value as it either gets struck in initial guess or returns value error. This is because we have a noisy data. Non-convex datasets as these may have multiple local minima, and the optimization algorithm can get stuck in a region where it can't satisfy both the bounds and the initial guess.

In [ ]:
```
T=14
print(np.exp((h * f [0]/ (kB * T))))
```

```
inf
```

Denominator becomes too high for values below 14, for some frequencies

In [ ]:
```
T=2.44e13
print(np.exp((h * f [-1]/ (kB * T)))-1)
```

```
0.0
```

Denominator becomes too low for some frequencies for values above 2.44e13

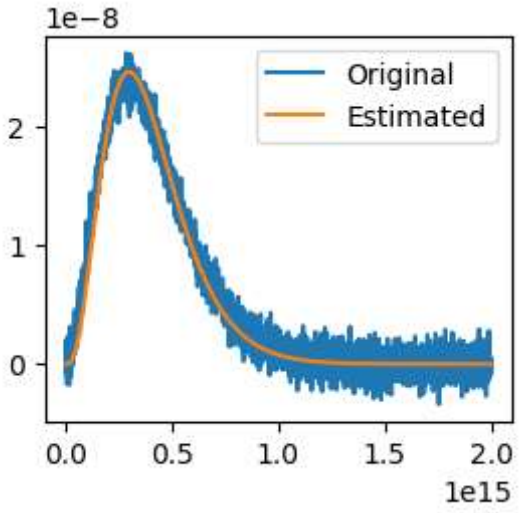
Second solution:

Assume we need to find k=k1.k2 where we know k. k1 and k2 could essentially lie anywhere and we could get lots of combinations. Thus we take T1=kB.T,c1=c^2(which isn't essential),h as variables

In [ ]:
```
f = []
y1 = []
with open("D:\\vscode\\a3-data\\a3-data\\dataset3.txt", "r") as file:
    data = csv.reader(file, delimiter=' ')
    for row in data:        #reading data
        f.append(float(row[0]))
        y1.append(float(row[1]))
def plank(f, T1,h,c1):      #radiation intensity function
    return [(2 * h * (x**3) / ((c1) * np.exp((h * x / (T1)))-1)) for x in f]
guess =[6.97e-20,6.626e-34,8.99e+16] #initial guess
sp1, _ = curve_fit(plank, f, y1, guess)  #curve fitting
print(f"kB.T={sp1[0]}\nh={sp1[1]}\nc={np.sqrt(sp1[2])}")
sest = plank(f, sp1[0],sp1[1],sp1[2])    #estimated values
plt.plot(f, y1)                          #plotting
plt.plot(f, sest)
plt.legend(["Original","Estimated"])
```

```
kB.T=7.729069887845371e-15
h=7.796471196549027e-29
c=91209316030.82516
```

Out[ ]: <matplotlib.legend.Legend at 0x2221a18a7c0>

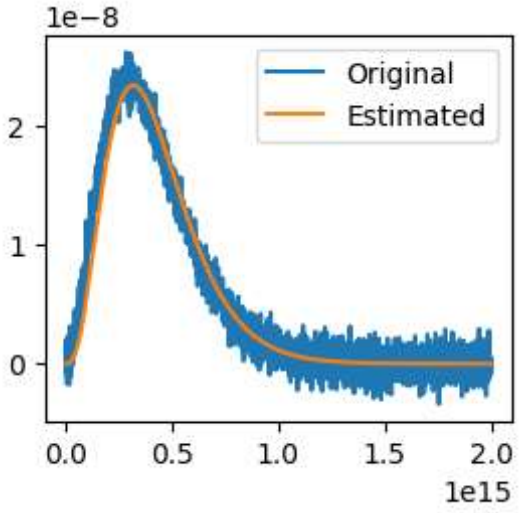


- The `curve_fit` tries to optimze further even when perfect values are supplied leading to deviation. Thus the ideal values around `[6.97e-20,6.626e-34,2.99e+8]` are not obtained
- Trying to give bounds makes algorithm get struck in initial values.(eg: bounds=([1e-22, 1e-36, 1e14], [1e-18, 1e-32, 1e19]))
- Reducing data points doesn't help and even data smoothening won't help
- One could give ftol=1e-1(or greater) to control the relative error in the sum of squares of the residuals and give close to ideal values in guess. At its best it gives, `[7.06e-18, 6.69e-32, 3e9]`

- Using `trf(Trust Region Reflective)` as our algorithm instead of `lm (Levenberg-Marquardt)` which is highly efficient, could decreses error in value of constants even in absence of guess.(Bounds of small range should be given)

In [ ]:
```
sp1, _ = curve_fit(plank, f, y1, bounds=([4e-20, 3.25e-34, 8e16], [1e-19, 1e-33, 1e17]),
#curve fitting after changing algorithm
sest = np.array(plank(f, sp1[0],sp1[1],sp1[2]))  #estimated values
print(f"kB.T={sp1[0]}\nh={sp1[1]}\nc={np.sqrt(sp1[2])}")
plt.plot(f, y1)                                  #plotting
plt.plot(f,sest)
plt.legend(["Original","Estimated"])
plt.show()
```

```
kB.T=7e-20
h=6.6250000000000005e-34
c=300000000.0
```



In the absence of accurate values, we have gotten a good approximation of constants by giving close bounds

Now,we could further split kB*T only if we know either of values.

In [ ]:
```
T=sp1[0]/kB
print(f"T={T:.3f}")
print(f"kB={kB}")
```

```
T=5070.093
kB=1.3806452e-23
```

- We could also try using least_squares in scipy.optimize instead of curve_fit. There we could control curve_fit through `loss='soft_l1', f_scale=0.1` where loss parameter is altered to make the optimization more robust(close to guess) and reduce scaling factor of algorithm through f_scale.