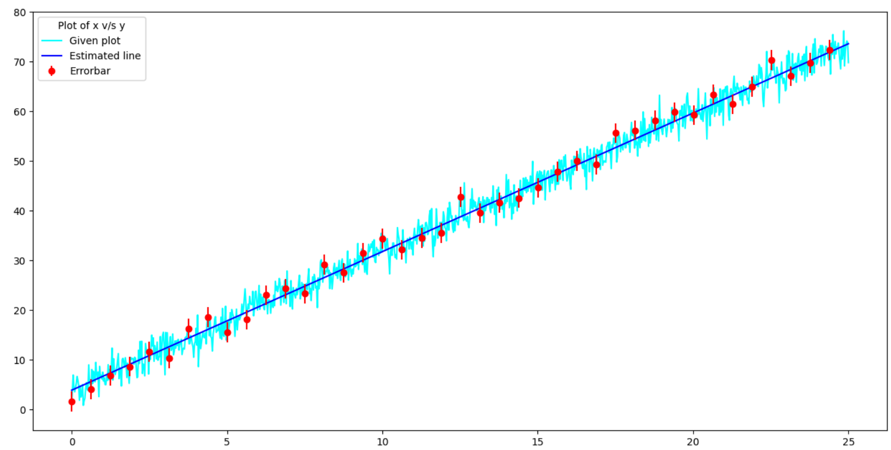
EE22B143 - Shah Dhruvanshi Kamlesh

Dataset - 1

- Firstly, importing the required libraries and storing data in `numpy` arrays `x` and `y`.
- Given function is linear and of the form $y(x, p_1, p_2) = p_1 x + p_2$.
- Collum stacking `x` and ones to form the `M` matrix(as shown below) which will be passed as argument to `np.linalg.lstsq` function.

$$\mathbf{Y} \equiv \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \end{pmatrix} \equiv \mathbf{Mp}$$

- Calling `np.linalg.lstsq` with `M`, `y` and `rcond = "None"`
- Out of the return values of `lstsq`, storing slope and inteceot as `p1` and `p2`.
- Calling `stline` function with `p1`, `p2` and `x` to store the estimated line points in `yest`.
- Finally, I am plotting the errorbars(for every 25 points) alongside the original and noisy curve as shown.



Dataset - 2

- Importing necessary libraries and storing data in `x` and `y`.
- I am then proceeding to smoothen out the graph using moving average algoritm Refer here to smoothen the curve.
- The main issue to find the time period of the given data was that the curve was noisy, but with smoothened curve, it becomes much simpler to calculate time

period.

- I am using the simple fact that time period is the distance beween two points that have same values. However, since the data is averaged and still not exact, we cannot guarantee that the same values might occur on differences of time period, so I am using the a limit (0.01), do find the points having value 0 except the point near origin.
- Taking the difference of these two time periods, we can find the time period. Please note that this won't work for every curve, but for this particular curve (having frequencies f, 3f and 5f as shown later), this method will indeed work well for given dataset.
- Finding the two x-values wherein the y-values become equal(0, to specific), and subtracting them, we can get the time period, that will be a good approximation to the original time-period.
- The time period is then used to find the frequencies of individual sine components, the exact relation between them is given below.
- After finding the frequencies the curve is plotted alongside the original curve.

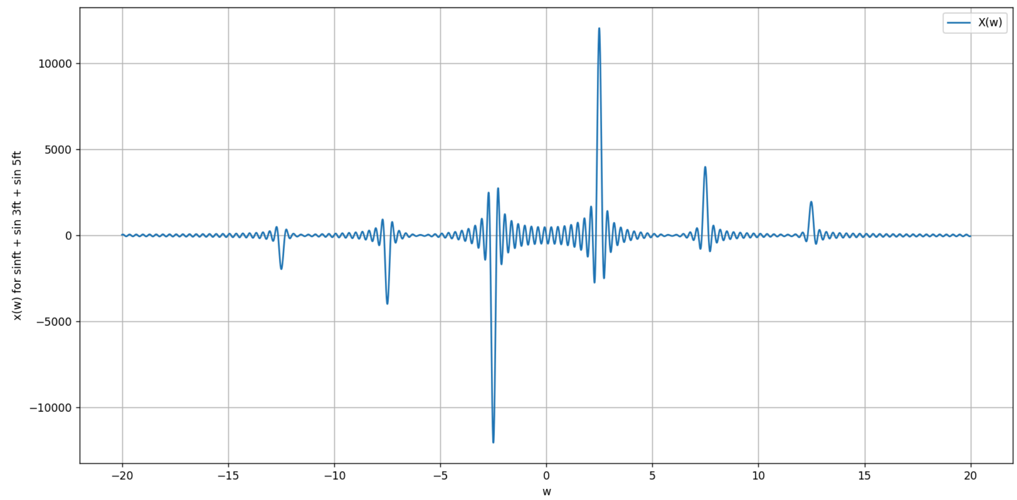
Explaination of why frequencies are f, 3f and 5f

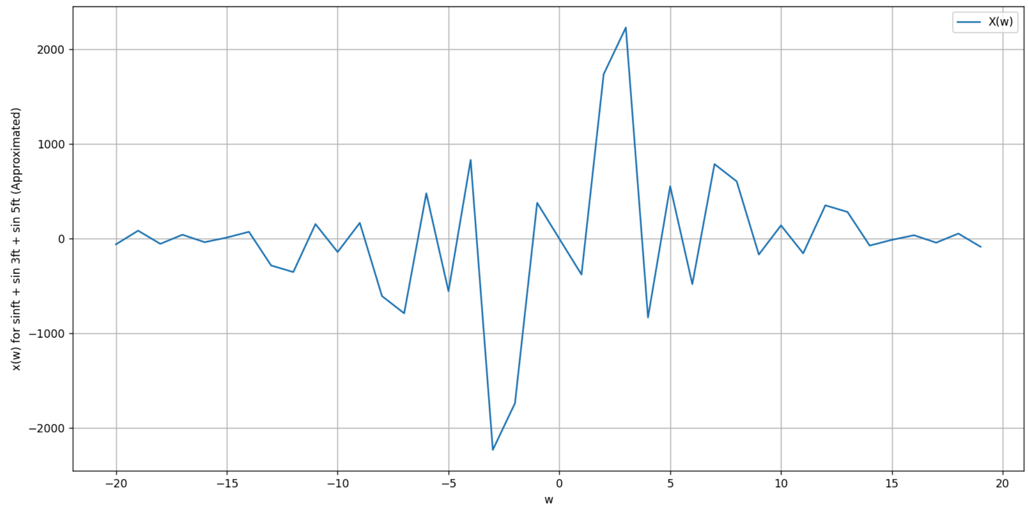
- Intuitively, we can look into the graph and conclude, the frequencies are odd mutiples of f since the graph appears to be odd. However, we can also conclude that frequencies be f, 2f and 3f with 2f having extremely low amplitude. This is indeed a valid solution, but we'll see further that comparing the std deviation of this with former case will lead us to confirm that former is better approximation.
- The best way to figure out frequencies is to perform fourier analysis. Since this is a noisy data, we may not get exact fourier transform but we can get peaks in the X(jω) curve and conclude the frequencies.

In [ ]:

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

T = 2.5165165165165164

data = np.loadtxt("dataset2.txt")
x = data[:, 0]
y = data[:, 1]

t = np.arange(-20, 20, 0.01)

y_1 = 6*np.sin(2*np.pi*t/T) + 2*np.sin(3*2*np.pi*t/T) + np.sin(5*2*np.pi*t/T)
def f2(a, b, c):
    return np.sum(c*np.sin((b)*a))
ynn = []
ynn1 = []
hello = np.arange(-40, 40, 1)
for y0 in t:
    ynn.append(f2(t, (y0), y_1))
for y0 in hello:
    ynn1.append(f2(x, (y0), y))
plt.plot(t, ynn, label = "X(w)")
plt.xlabel("w")
plt.ylabel("x(w) for sinft + sin 3ft + sin 5ft")
```

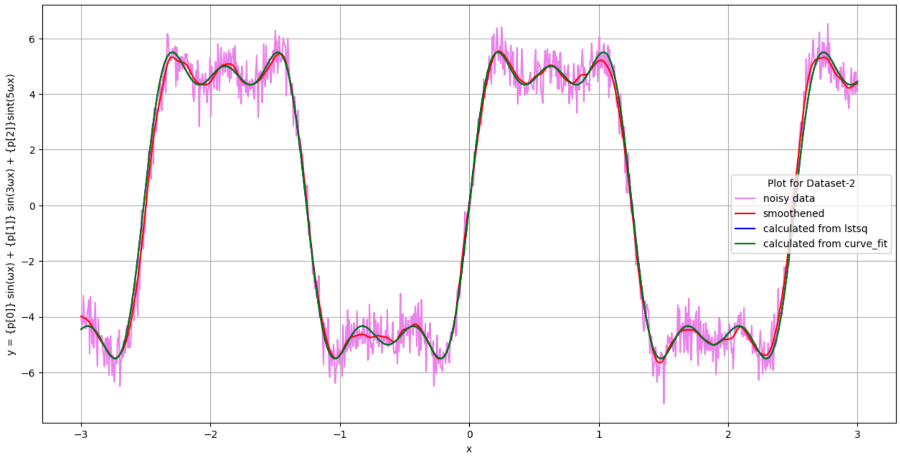
```
plt.grid("True")
plt.legend()
plt.show()
plt.plot(hello, ynn1, label = "X(w)")
plt.xlabel("w")
plt.ylabel("x(w) for sinft + sin 3ft + sin 5ft (Approximated)")
plt.grid("True")
```





- However, there's one ambiguity, as noticed in the approximated graph for our data: The peaks of 2f(~5) and 5f(~12.5) are nearly same and in fact the peak for 5f is greater, which may force us to conclude that f, 2f and 3f is the possible relation between frequencies, but we also need to coonsider the fact that our data is less, however, as the amount of points increases, we'll have a graph similar to first one. SInce, the two amplitudes, son't differ much, it's better to consoder both, one at a time.
- Also, if we accept for a moment that f, 2f and 3f are frequencies and calculate the amplitudes, we'll get the amplitude of 2f more than 50 times smaller than the amplitude of sine wave with frequency f but the amplitude of sine wave with frequncy 5f is comparable to that of frequncy f .
- Moreover, the standard deviation when f, 3f and 5f are taken as frequencies is less(`0.5416040238963677`) than the standard deviation when frequencies f, 2f and

3f(`0.8851425479242195.`) are taken.



Dataset - 3

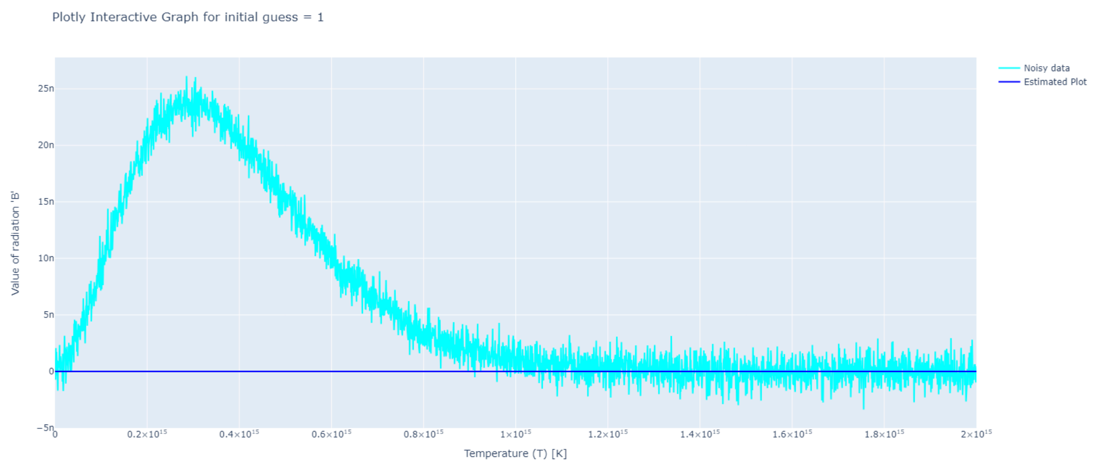
Solution - 1

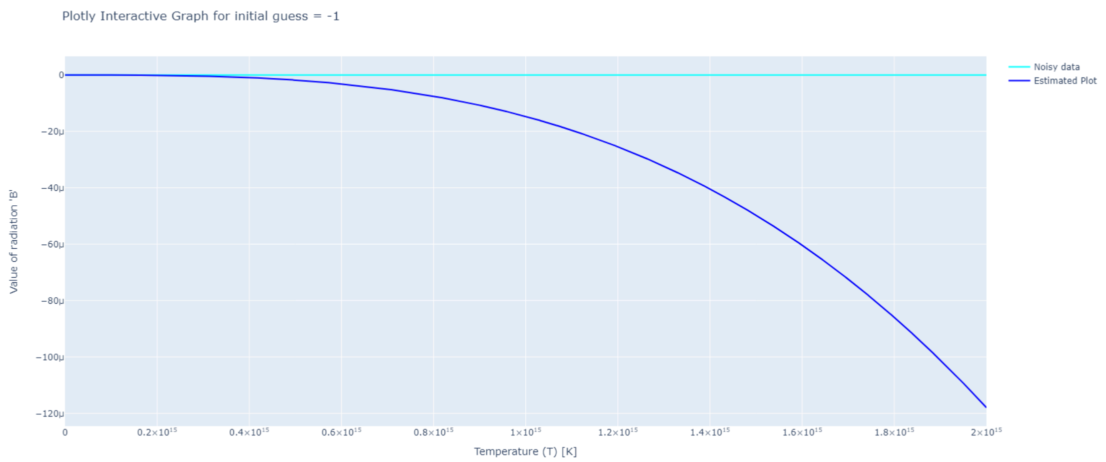
- After importing the required modules and diabling warnigs (since calculating radiation intensity will cause overflow), I have defined `PLANCK_CONSTANT` , `SPEED_OF_LIGHT` and `BOLTZMANN_CONSTANT` and then data is read and stored in `f` and `B` numpy arrays.
- Then, I have defined mapping function which takes all constants and temperature as parameters, and the function is then passed into `curve_fit` to optimize with a valid initial guess(`initial_guess`), the explaination of which is given.
- After calculating the most optimal `cuve_fit` parameters, graph is plotted to compare the given noisy data and the original data.

Explaination of Initial Guess

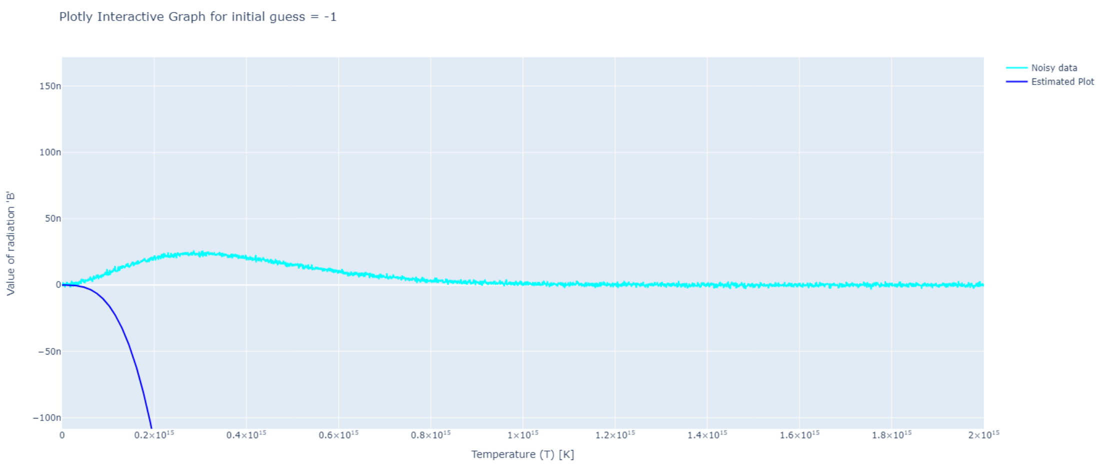
- According to Sci-Py docs(Refer here), the `Sci-Py` works on Non-linear least Square model(Refer here), wherein the initial guess given determines the solution that the algorithm is going to provide.
- Non-Linear least squares model optimizes by reducing the residual error(Refer here%20is,underlying%20the%20NLS%20regression%20model.)), that is the algorithm will try to find the minima in the error or loss function using an algorithm similar to Gradient Descent(Refer here). However, there might be many issues, as to why gradient descent might fail in some non linear cases. Some off them include:
  - As shown in the video, the matrix (AA^T) can become non-invertible(singular), which can explain the fact that we get `Runtime Error: division by zero encountered` and `OptimizeWarning` when an invalid value is given.
  - Secondly, since the curve is non-linear we can't assure that the minimum we get is Global Minimum, like it would have been the case, if for example, the loss

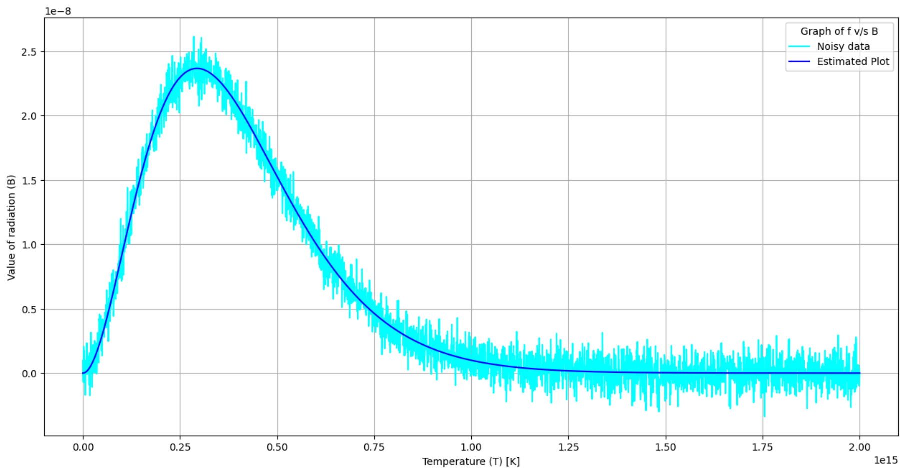
    function is a Gaussian, which it is in many cases. Therefore, some `initial_guess`, might not give the correct curve.
    - Moreover, the algorithm could also depend on learning rate encountered in Gradient descent algorithm. If the learning rate is not high enough, it could pass the optimum, and return the incorrect solution, if the minima is too steep.
- Hence, giving valid initial guess works on trial and error basis. For the given code, I encountered straight lines for extremely low values of initial temperature as shown. This couldd be explained by the fact that for extremely low values, the denominator tends to infinity since, the exponential term's denominator(k_BT) tends to 0.
- Moreover, if we randomly enter negative values of temperature, similar situation like above will arise, and our graph will look something like this. The noisy data is looks linear due to scaling.





Zoomed in image indicating the effect of compressed axis





Solution 2

- Importing the required libraries disabling warnings, and defining constants, the code proceeds to taking data from user.
- It then defines the mapping and initial guess(exact method of finding optimal initial guess is given below).
- The code then prints the arguments obtained by `curve_fit` function and plots them.

Explaination for optimal guess

- curve_fit will return the parameters that trace the curve that traces the noisy data, but we also need to ensure that the sum of relative error is minimum since all the parameters are known(except temperature, everything is a constant and temperature is known from solution).
- Hence, I am iterating thorught various initial values of T while keeping all other parameters constant to get that initial value of t whhich will give least error(which is 2.56%) and then printing the corresponding function values.

In [ ]:
```
from scipy.optimize import curve_fit
import numpy as np
import matplotlib.pyplot as plt
import plotly.graph_objs as go

import warnings
warnings.simplefilter("ignore")

PLANCK_CONSTANT = np.float64("6.62607015e-34")
SPEED_OF_LIGHT = np.float64("299792458")
BOLTZMANN_CONSTANT = np.float64("1.380649e-23")

data = np.loadtxt("dataset3.txt")
f = data[:, 0]
B = data[:, 1]

def mapping(x, T, h, c, k_b):
    return (2*h*(x**3)/((c**2)*(np.exp((h*x)/(k_b*T)) - 1)))
initial_guess = [5246, PLANCK_CONSTANT, SPEED_OF_LIGHT, BOLTZMANN_CONSTANT]
t0 = 1000
min = 100
for t in range(1, 10000):
    initialguess =  [t, PLANCK_CONSTANT, SPEED_OF_LIGHT, BOLTZMANN_CONSTANT]
    args, pcov = curve_fit(mapping, f, B, initialguess)
    err = (abs(args[0] - 4997.341993826246)/4997.341993826246) + abs((args[1] -
    if err < min:
        min = err
        t0 = t
print(t0, min)
# 1984 0.9270305358077109 ; error for range(1000, 2000)
# 2803 0.47443222099388194 ; error for range(2000, 3000)
# 3382 0.30172617272840807 ; error for range(3000, 4000)
# 4996 0.02914523111316886 ; error for range(4000, 5000)
# 5246 0.025654352118527048 ; error for range(5000, 6000)
# 6149 4.356972793229201 ; error for range(6000, 7000)
```

```
5246 0.025654352118527048
```