Física Numérica

Julio César Avila Torreblanca

09 de noviembre del 2021

Tarea 6

1. Interpolación de Lagarange

(a) Escriba un programa que ajuste un polinomio según el algoritmo de Lagrange a un conjunto de n puntos.

Solución.
Aquí haremos uso de las siguientes librerías. Recordemos que en el algoritmo de interpolación de

```
In [ ]: import pandas as pd
        import sympy as sp
        import numpy as np
```

Lagrange, tenemos una variable independiente x y n valores de ella: x_i $(i = 1, 2, ..., n)$. Para esta variable independiente existe una función $g(x)$ con valores $g_i = g(x_i)$. Con esto se busca aproximar a $g(x)$ mediante un polinomio de grado $n-1$, dado por:

$$g(x) \approx \sum_{i=1}^{n} g_i L_i(x) \tag{1}$$

Donde:

$$L_i(x) = \prod_{j(\neq i)=1}^{n} \frac{x - x_j}{x_i - x_j} = \frac{x - x_1}{x_i - x_1}\frac{x - x_2}{x_i - x_2} \cdots \frac{x - x_n}{x_i - x_n} \tag{2}$$

Con base en esto, crearemos un algoritmo que realice interpolación de Lagrange.

Lo primero que haremos será crear una función llamada `Lx`, la cual calculará los polinomios $L_i(x)$ usando la ecuación (2). Notemos que estos polinomios son de grado $n-1$. Para esta función nos apoyaremos del cálculo simbólico integrado por la librería `sympy`. Como parámetros se piden el subíndice `i` del polinomio $L_i(x)$ por calcular y una lista `values` que contendrá los datos conocidos de la variable independiente x: $(x_1, x_2, ..., x_n)$.
El código de la función es el siguiente (los comentarios contienen una explicación de qué se está realizando en cada sentencia):

```
In [ ]: def Lx(i:int , values: list):
            """
            Esta función calcula el coeficiente de gragange L_i(x) de forma simbólica
            para la imterpolación de Lagrange.

            Parámetros:
                i = número de índice de L_i(x) po obtener
                x = lista que contiene los datos experimentales para obtener L_i(x)
            Salida:
                Regresa un polinomio de grado n-1, con n el número de datos. Este
                polinomio será el término L_i(x) en la interpolación de Lagrange.
            """

            n = len(values)# Asigna el número de datos
            L, x , y = sp.symbols('L, x, y') #Creamos las variables para el cálculo simbólico

            #Inicializamos las variables
            y = 1
            L = 1

            #Este ciclo calcula L_i(x) haciendo los productos en la fórmula
            for j in range(n):
                if j != i-1:
                    y = (x - values[j])/float((values[i-1] - values[j]))
                    L *= y
            #Expandimos el resultado haciendo todos los productos
            L = sp.expand(L)
            return L
```

Ya con esta función será muy sencillo calcular el polinomio de Lagrange a partir de la fórmula (1) y completar el algoritmo para realizar interpolación de Lagrange.
Dentro del código debemos importar los datos experimentales usando la librería `pandas`. Ya con esos datos, guardaremos los puntos de interés $(x_1, x_2, ..., x_n)$ y $(g_1, g_2, ..., g_n)$ en dos arreglos distintos. Para no tener problemas con el manejo de estos datos, los transformaremos en tipo `float64`. Después guardaremos en una variable `n` el número de datos que coincide con la longitud de cualquier arreglo, esto es necesario saber ya que en la fórmula (1) se necesita. Enseguida definimos las variables simbólicas `g` y `x` para obtener $g(x)$ a través del cálculo simbólico usando `sympy`. Por último, a través de un ciclo `for` haremos los cálculos correspondientes a la fórmula (1) para generar nuestro polinomio $g(x)$. Este proceso se aprecia en el siguiente código:

```
In [15]: # Importamos los datos de un archivo txt
         data = pd.read_csv(r'C:\Users\cesar.avila\Documents\IPN\Semestre 7\Física Numérica\Tareas\Tarea6\DatosLagrange.txt',
                            header=0,delim_whitespace = True)

         # Recopilamos los datos de las dos columnas
         E = data.iloc[:,0]
         f = data.iloc[:,1]

         # Transformamos los datos a dos arreglos x,y para su mejor manejo
         data_x = np.float64(E)
         data_y = np.float64(f)

         #Número de datos
         n = len(data_x)

         # Definimos variables para la obtendión del polinomio y los inicializamos
         g, x = sp.symbols('g, x')
         g = 0.

         #Calculamos el polinomio completo de Lagrange de grado n-1
         for i in range(n):
             g += data_y[i]*Lx(i+1,data_x)
```

El código anterior calcula el polinomio de Lagrange para cualquier conjunto de n puntos, solo hace falta cambiar la ruta del archivo en el renglón 2 por el archivo que se desee. En este caso hemos

usado la ruta del archivo que se pide en el siguiente inciso. Lo único que resta es poner aprueba el algoritmo y graficar para ver el resultado. Esto se hará en el siguiente inciso.

(b) Utilice su programa para ajustar un polinomio al conjunto de datos: Utilice si ajuste para

$i =$	1	2	3	4	5	6	7	8	9
$E_i\,(MeV)$	0	25	50	75	100	125	150	175	200
$f\,(E_i)\,(MeV)$	10.6	16.0	45.0	83.5	52.8	19.9	10.8	8.25	4.7
$\sigma_i\,(MeV)$	9.34	17.9	41.5	85.5	51.5	21.5	10.8	6.29	4.14

graficar la sección eficaz en pasos de 5 MeV

Solución.
Primeramente creamos un archivo .txt que contenga los datos experimentales. Estos datos los leemos con el algoritmo del inciso anterior para generar el polinomio de Lagrange correspondiente. El polinomio generado para estos datos es el siguiente:

```
In [4]: sp.pprint(g)
                          8                         7                        6
- 1.15581968253968e-13·x  + 9.98107428571427e-11·x  - 3.54436266666667e-8·x  +

                        5                 4                       3
 6.63364266666666e-6·x  - 0.0006944452·x  + 0.0395631066666667·x  - 1.09353836

       2
507937·x  + 11.4094238095238·x + 10.6
```

Lo siguiente a realizar es graficar este polinomio junto con los datos experimentales para hacer la comparación y evaluar el ajuste obtenido.
Para esta parte usaremos las siguientes librerías.

```
In [ ]: import numpy.polynomial.polynomial as poly
        import matplotlib.pyplot as plt
```

En el algoritmo primero convertimos el polinomio `g` a un objeto tipo polinomio de la librería de `numpy` y extraemos los coeficientes en un arreglo llamado `coef`. Luego reacomodamos los coeficientes para crear el polinomio adecuado en una variable llamada `pol`. Enseguida creamos un arreglo llamado `valores_E` que contendrá los puntos a ser evaluados en el eje x, notemos que en la definición de este arreglo contendrá la longitud de los pasos de 5 MeV que se piden. Lo siguiente a realizar es definir la gráfica y evaluar en los puntos que se tienen. Esto se observa en el siguiente algoritmo:

```
In [9]: #Convertimos el g a un objeto tipo polinomio y guardamos sus coeficientes en una lista
p = sp.Poly(g)
coef = p.rep.rep

#Creamos el polinomio con numpy para un mejor manejo
pol = poly.Polynomial(coef[::-1])

#Creamos un arrgelo que contendrá los puntos a ser evaluados  en el eje x
valores_E = np.arange(-2.0, 205, 5) #El último parámetro contiene la longitud de los pasos

#Graficamos el polinomio y los ptos experimentales
fig = plt.figure(figsize = (10,8))
plt.plot(valores_E, pol(valores_E), 'b', data_x, data_y, 'ro')
plt.title('Interpolación de Lagrange')
plt.grid()
plt.xlabel('$E_i$  [MeV]')
plt.ylabel('$f(E_i)$  [MeV]')
plt.show()
```

A partir de esto generamos el siguiente gráfico donde la recta azul muestra el polinomio de Lagrange obtenido y los puntos rojos son los puntos experimentales.

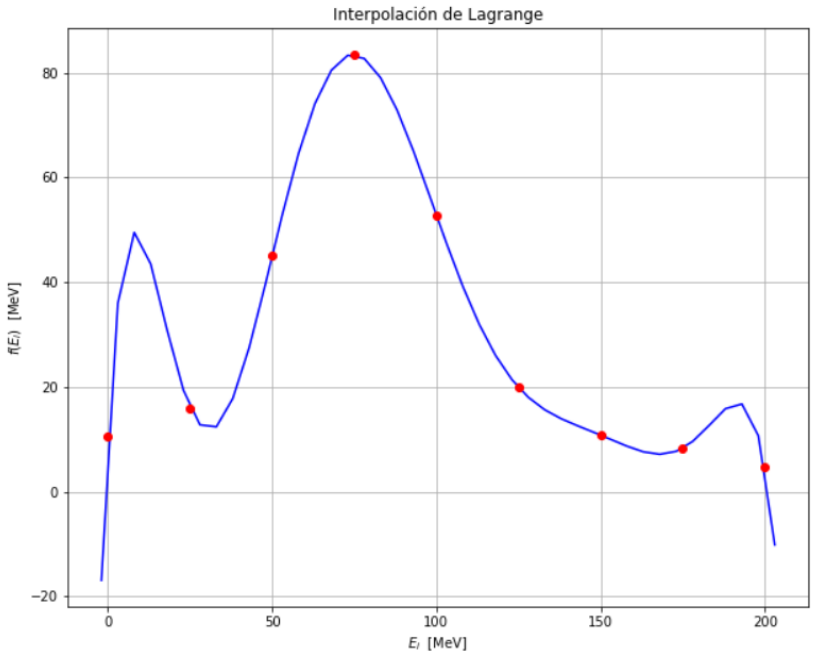


Figura 1: Gráfica del polinomio de Lagrange con pasos de 5 MeV.

El polinomio de Lagrange pasa por todos los puntos experimentales. La razón por la que las curvas no son suaves es porque se graficó en pasos de 5 MeV. Si se hace más pequeño el paso, se tendrán curvas más suaves. Por último, notemos que este tipo de ajuste, a pesar de que pasa por todos los puntos, no es correcto. Esto se debe a que hay zonas donde se crean picos o curvas que se desvían de la tendencia y no nos dicen nada en esa sección. Por lo que este método no es adecuado para ajustar nuestros datos.

(c) Utilice su gráfica para estimar la energía de resonancia E_r y Γ el ancho a la mitad del máximo (*full-width at half-maximun*). Compare sus resultados con el valor predicho por la teoría $(E_r, \Gamma) = (78\,\text{MeV}, 55\,\text{MeV})$.

Solución.
Para obtener la energía de resonancia, necesitamos el máximo absoluto de nuestro polinomio de Lagrange. Notemos que este punto está cercano al punto experimental rojo con $f(E_i) = 75$ MeV. Para determinarE_r, realizaremos el criterio de la primera derivada y obtendremos las raíces del polinomio resultante. Estas raíces serán los candidatos a ser máximos o mínimos, y como sabemos que el máximo absoluto es cercano a $f(E_i) = 75$ MeV, tomaremos la raiz más cercana a ese valor. En el siguiente algoritmo se realiza esta parte:

```
In [10]: #Obtengamos el máximo del polinomio de Lagrange con el criterio de la primer derivada
         df = sp.diff(g,x) #1ra derivada

         critic_points = list(sp.solveset(sp.Eq(df,0))) #Obtenemos las raices de la 1ra derivada

         print('Las raíces de la 1ra derivada son:\n',critic_points)#Imprimimos las raíces
```

```
Las raíces de la 1ra derivada son:
 [8.29188710073911, 30.7769085373451, 74.5802207708932, 168.097363948498, 191.673104120887, 141.093133587003 - 14.6352020117278
*I, 141.093133587003 + 14.6352020117278*I]
```

Notemos que el valor buscado está en la posición 2 de la lista. Así nuestra energía de resonancia para el polinomio de Lagrange obtenido está dada por:

$$E_r = 74.580\,220\,770\,893\,2\,\text{MeV}$$

Por otro lado, para determinar Γ basta resolver la ecuación $g(x) = \frac{g(E_r)}{2}$, donde $g(x)$ es el polinomio de Lagrange. Si analizamos la figura (1), las dos soluciones que nos interesan deben ser las más cercanas a la energía de resonancia E_r. La diferencia de estos dos valores será el valor de Γ. En el siguiente código se muestra este proceso.

```
In [12]: #obtengamos la energía de resonancia y gamma
         E_r = critic_points[2]
         mitad_max = sp.sympify(g).subs(x,E_r) /2. #Obtenemos el máximo por la mitad

         #Raices de la ecuación g(x) = E_r
         candidatos = list(sp.solveset(sp.Eq(g,mitad_max)))
         print('Soluciones a la ecuación g(x) = g(E_r)/2\n',candidatos)
```

```
Soluciones a la ecuación g(x) = g(E_r)/2
 [4.15253976518238, 13.7637238439646, 48.7495359122139, 106.439115386978, 148.814112055004 - 36.4474827181311*I, 148.8141120550
04 + 36.4474827181311*I, 196.408145720752 - 12.0172541150965*I, 196.408145720752 + 12.0172541150965*I]
```

Los valores que nos interesan son los elementos en las posiciones 2 y 3 de la lista. Así la diferencia de estos nos da el valor de Γ.

```
In [14]: #Valor de Gamma
         gamma = candidatos[3]- candidatos[2]
```

Comparando con los valores teóricos tenemos los siguientes resultados:

```
In [15]: #Errores porcentuales
         errorE_r = 100.*abs(78 - E_r)/78.
         errorGamma = 100.*abs(55-gamma)/55.

         print(f'Energía de resonancia: {E_r} | Con un error de: {errorE_r} %')
         print(f'Gamma: {gamma} | Con un error de: {errorGamma} %')

Energía de resonancia: 74.5802207708932 | Con un error de: 4.38433234500873 %
Gamma: 57.6895794747641 | Con un error de: 4.89014449957113 %
```

Podemos ver que los valores que se determinaron para E_r y Γ son muy cercanos a los valores teóricos.

---

2. Interpolación vía splines cúbicos

(a) Para los datos del problema anterior, ajuste splines cúbicos utilizando una rutina de las bibliotecas de Python. Estime los valores para la energía de resonancia E_r y Γ el ancho a la mitad del máximo (*full-width at half-maximum*).

Solución.
Para este ejercicio utilizaremos la rutina `CubicsSpline` de la librería de `scipy`. En el siguiente algoritmo se crea el spline cúbico. Al igual contiene comentarios que explican que hace cada parte.

```
In [16]: from scipy.interpolate import CubicSpline
         import matplotlib.pyplot as plt
         import pandas as pd
         import numpy as np

         # Importamos los datos de un archivo txt
         data = pd.read_csv(r'C:\Users\cesar.avila\Documents\IPN\Semestre 7\Física Numérica\Tareas\Tarea6\DatosLagrange.txt'
                            ,header=0,delim_whitespace = True)

         # Recopilamos los datos de las dos columnas
         E = data.iloc[:,0]
         f = data.iloc[:,1]

         # Transformamos los datos a dos arreglos x,y para su mejor manejo
         x = np.float64(E)
         y = np.float64(f)

         #Creamos el CubicSpline
         cs = CubicSpline(x, y)

         #Creamos un arrgelo que contendrá los puntos a ser evaluados  en el eje x
         eje_x = np.arange(-2.0, 205, 5)
         eje_y = cs(eje_x)

         fig, ax = plt.subplots()
         ax.plot(x,y, 'og', label = 'Datos experimentales')
         ax.plot(eje_x, eje_y,  label = 'Spline Cúbico')
         ax.legend(loc='upper right')
         plt.show()
```

El ajuste obtenido fue el siguiente:

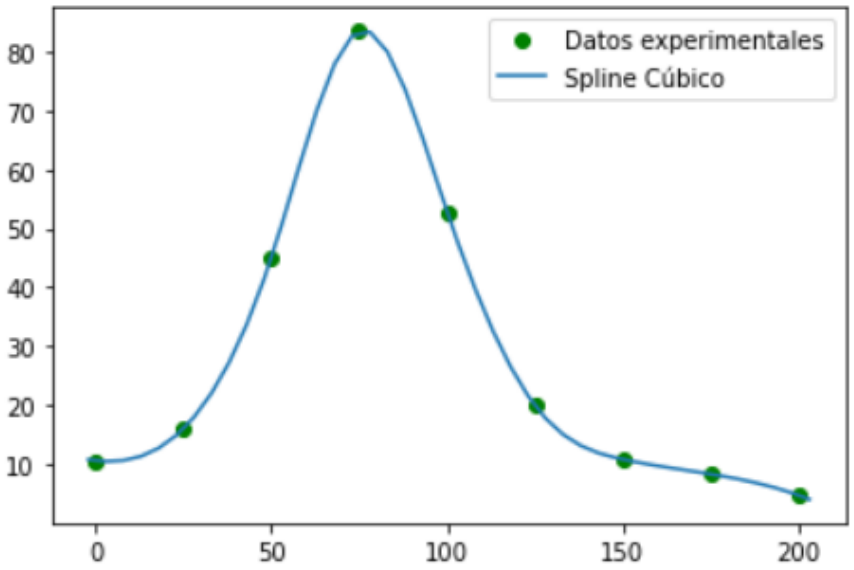


Figura 2: Ajuste por Splines Cúbicos con pasos de 5 MeV.

Podemos apreciar que este ajuste es mucho mejor que la interpolación de Lagrange, pues se aproxima mejor a la tendencia de los datos.
Para determinar el valor de la energía de resonancia, usaremos un `while` en el cual se evaluará de 1 en 1 nuestro ajuste $g(x)$ de Spline Cúbico y guardará el valor x_0 para el cual $g(x_0)$ es máxima. Esto se aprecia en el siguiente algoritmo:

```
In [18]: #Esta parte evalúa de 1 en 1 nuestro ajuste para determinar el máximo y así obtener E_r
while i != 200:
    Antes = cs.__call__(i-1)
    Desp = cs.__call__(i)
    if Desp > Antes:
        E_r = i
    i += 1

print(f'La energía de resonancia es: {E_r} MeV')
```

```
La energía de resonancia es: 76 MeV
```

Por lo tanto, para el ajuste por Spline Cúbicos obtenemos:

$$E_r = 76\,\text{MeV}$$

Ahora para determinar Gamma recorreremos nuestro spline cúbico en el eje y el valor de la mitad del máximo y obtendremos las raíces. Esto se realiza en el siguiente algoritmo:

```
In [20]: from scipy.interpolate import sproot, splrep

         #Maximo dividido por la mitad
         mitad_maximo = cs.__call__(E_r)/2.

         #Creamos el spline recorrido como tipo splrep para poder sacar sus raices
         cs_recorrido = splrep(x, y- mitad_maximo)
         #sacamos las raices
         r1, r2 = sproot(cs_recorrido)

         #Valor de Gamma
         gamma = r2 - r1
         print(f'Gamma: {gamma} MeV')
```

```
Gamma: 58.41018191655321 MeV
```

Cuando se recorre nuestro spline cúbico se obtiene lo siguiente:

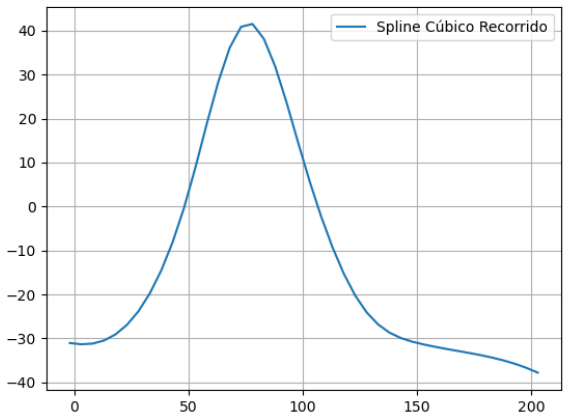


Figura 3: Spline Cúbico recorrido.

Notemos que las raíces reales serán los valores que determinarán Γ. El valor obtenido fue $\Gamma = 58.410\,181\,916\,553\,21$ MeV.
Comparando con los valores teóricos tenemos los siguientes resultados:

```
In [21]: #Errores porcentuales
         errorE_r = 100.*abs(78 - E_r)/78.
         errorGamma = 100.*abs(55-gamma)/55.

         print('-'*50)
         print(f'Energía de resonancia: {E_r} MeV| Con un error de: {errorE_r} %')
         print(f'Gamma: {gamma} MeV| Con un error de: {errorGamma} %')
```

```
--------------------------------------------------
Energía de resonancia: 76 MeV| Con un error de: 2.5641025641025643 %
Gamma: 58.41018191655321 MeV| Con un error de: 6.200330757369479 %
```

---

3. Ajuste de la fórmula de resonancia de Breit-Wigner

(a) Para la teoría indica que la fórmula de Breit - Wigner debe ajustar a los datos de los dos ejercicios anteriores:

$$f(E) = \frac{f_r}{(E - E_r)^2 + \frac{\Gamma^2}{4}} \tag{3}$$

Su problema consiste en determinar los valores para los parámetros E_r, f_r y Γ. Se sugiere renombrar los parámetros haciendo:

$$a_1 = f_r, \quad a_2 = E_r, \quad a_3 = \frac{\Gamma^2}{4}, \quad x = E$$

Para escribir:

$$g(x) = \frac{a_1}{(x - a_2)^2 + a_3} \tag{4}$$

y encontrar los parámetros a partir de minimizar χ^2, encuentre estas ecuaciones.

Solución.
Las ecuaciones que minimizan a χ^2 son:

$$f_1(a_1, a_2, a_3) = \sum_{i=1}^{9} \frac{y_i - g(x_i)}{\left[(x_i - a_2)^2 + a_3\right] \sigma_i^2} = 0 \tag{5}$$

$$f_1(a_1, a_2, a_3) = \sum_{i=1}^{9} \frac{(y_i - g(x_i))(x_i - a_2)}{\left[(x_i - a_2)^2 + a_3\right]^2 \sigma_i^2} = 0 \tag{6}$$

$$f_1(a_1, a_2, a_3) = \sum_{i=1}^{9} \frac{(y_i - g(x_i))}{\left[(x_i - a_2)^2 + a_3\right]^2 \sigma_i^2} = 0 \tag{7}$$

Obtengamos estas ecuaciones en python. Para ello usemos las siguientes librerías:

In [ ]:
```python
import pandas as pd
import sympy as sp
import numpy as np
```

Para generar $g(x)$ crearemos la siguiente función:

In [ ]:
```python
def gx(i:int , values: list):
    """
    Esta funcion nos sirve para determinar el valor de g(x_i) utilizando el calculo
    simbolico de sympy.

    -------------
    Parámetros:
        i -> int: número de índice de x_i.
        x -> list:  lista que contiene los datos experimentales

    -------------

    Salida:
        Regresa la ecuación g(x_i)
    """

    #Creamos las variables para el cálculo simbólico
    g, a1, a2, a3 = sp.symbols('g, a1, a2, a3')

    #Inicializamos funcion con 0
    g = 0

    # Usamos la definicion de g para evaluar en x_i
    g = a1/((values[i] - a2)**2 + a3)
    g = sp.expand(g)

    # regresamos el valor de g(x_i)
    return g
```

Ya con esta función podremos generar las funciones f_1, f_2, f_3. Primeramente leamos los datos experimentales y transformemoslos a flotantes:

```
# Importamos los datos de un archivo txt
data = pd.read_csv(r'C:\Users\cesar.avila\Documents\IPN\Semestre 7\Física Numérica\Tareas\Tarea6\DatosLagrange.txt'
                   ,header=0,delim_whitespace = True)

# Recopilamos los datos de las dos columnas
E = data.iloc[:,0]
f = data.iloc[:,1]
sigma = data.iloc[:,2]

# Transformamos los datos a dos arreglos x,y para su mejor manejo
data_x = np.float64(E)
data_y = np.float64(f)
sigma_y = np.float64(sigma)
```

Ya con esto definamos las variables simbólicas con la librería de `sympy` y calculemos las funciones f_1, f_2, f_3.

```
#Guardamos el número de datos
n = len(data_x)
# Definimos variables para la obtendión g(x) y los inicializamos
g, a1, a2, a3 = sp.symbols('g, a_1, a_2, a_3')
g = 0.

# Definimos variables para la obtendión f_1, f2, f3 y los inicializamos
f1, f2, f3, aux = sp.symbols('f_1, f_2, f_3, aux')
f1 = 0.
f2 = 0.
f3 = 0.

for i in range(n):
    #Cálculo de g(x_i)
    g = gx(i,data_x)

    #Cálculo de  f1
    aux = (data_y[i] - g) / (((data_x[i] - a2)**2 + a3) * sigma_y[i]**2)
    #aux = sp.expand(aux)
    f1 += aux

    #Cálculo de  f2
    aux = (data_y[i] - g) * (data_x[i] - a2) / (((data_x[i] - a2)**2 + a3)**2 * sigma_y[i]**2)
    #aux = sp.expand(aux)
    f2 += aux

    #Cálculo de  f3
    aux = (data_y[i] - g) / (((data_x[i] - a2)**2 + a3)**2 * sigma_y[i]**2)
    #aux = sp.expand(aux)
    f3 += aux
```

(b) Las ecuaciones que obtuvo en el inciso anterior NO son lineales, elabore un programa que utilice el método de Newton-Raphson multidimensional para la búsqueda de raíces.

Solución.
Las semillas que usaremos para el método de Newton-Raphson serán los valores teóricos del ejercicio 1 : $(E_r, \Gamma) = (78\,\text{MeV}, 55\,\text{MeV})$. Esto con el objetivo de acelerar la convergencia.

```
In [ ]: #Semillas de a1,a2,a3
        x = 1.
        y = 84.
        z = 756.25
```

EL método de Newton-Raphson de forma multidimensional, nos dice que dado un vecor de funciones

$\vec{f}$ el cual tiene por entradas a las funciones f_i determinadas en el inciso (a), se cumple lo siguiente:

$$\vec{f} + \vec{J}\Delta\vec{x} = 0$$

Donde $\vec{J}$ es la matriz jacobiana de las f_i's y $\Delta\vec{x}$ es el vector de los desplazamientos que se realizan con este algoritmo. De esta forma, encontraremos una solución de la forma:

$$\Delta\vec{x} = -\vec{J}^{-1}\vec{f}$$

Este proceso se realiza en el siguiente código:
Primero definimos las funciones que nos generarán las matrices $\vec{f}$ y la matriz jacobiana $\vec{J}$.

```
def f_vector(f1,f2, x:float, y:float):
    """Esta función calcula el f vector en la solución de Newton-Rapghson y
    evalúa a las funciones f1, f2 en los puntos x,y.
    Regresa una matriz."""

    a1, a2 = sp.symbols('a1, a2')

    valor1 = f1.subs([(a1,x), (a2,y)])
    valor2 = f2.subs([(a1,x), (a2,y)])


    return np.array([[valor1], [valor2]])

def jacobiano(f1,f2, x:float, y:float):
    """Esta función calcula la matriz Jacobiana en la solución de Newton-Rapghson
    y evalúa a las derivadas parciales de f1, f2, f3 en los puntos x,y,z.
    Regresa una matriz."""

    df, a1, a2 = sp.symbols('df, a1, a2')

    variables = [a1, a2]
    funciones = [f1, f2]

    #Saca la derivada y evalua los valores de a1,a2 = x,y
    J = np.array([[np.float64(sp.diff(funciones[i], variables[j]).subs([(a1,x), (a2,y)]))
                for j in range(len(variables))] for i in range(len(variables))])

    return J
```

Luego definamos las siguientes variables que nos ayudaran a realizar las iteraciones para Newton-Raphson:

```
# Longitud a1+Delta_a1,a2+Delta_a2, a3+Delta_a3

a = [0., 0., 0.]
a_new = [x, y, z]

# Definimos el error para detener la iteración de Newton-Raphson
error = 10**(-5)
i=1

condicion = abs(a_new[0] - a[0])
```

Ya con las matrices definidas, podremos iterar para obtener las estimaciones para a_1, a_2, a_3.

```
#Iteraciones Newton-Raphson:
while  condicion > error:
    print(f'No. iteración: {i}')
    print(f'Condición: {condicion}')
    A = f_vector(f1, f2, f3, a_new[0], a_new[1], a_new[2])
    B = jacobiano(f1, f2, f3, a_new[0], a_new[1], a_new[2])
    inv_B = np.linalg.inv(B)

    Delta_a = -inv_B.dot(A)

    #Guardamos los viejos valores
    a[0] = a_new[0]
    a[1] = a_new[1]
    a[2] = a_new[2]

    #Asignamos los nuevos valores
    a_new[0] += Delta_a[0][0]
    a_new[1] += Delta_a[1][0]
    a_new[2] += Delta_a[2][0]
    condicion = abs(a_new[0] - a[0])
    i+=1

print('-'*50)
print(f'Valores:\na_1 = {a_new[0]}\na_2 = {a_new[1]} \na_3 = {a_new[2]}')
```

El código anterior nos produce los siguientes resultados:

```
No. iteración: 1
Condición: 1.0
No. iteración: 2
Condición: 67655.4330261934
No. iteración: 3
Condición: 24614.9900253153
No. iteración: 4
Condición: 28442.6096127858
No. iteración: 5
Condición: 3793.96340011013
No. iteración: 6
Condición: 434.129151612229
No. iteración: 7
Condición: 1788.56911727491
No. iteración: 8
Condición: 982.129104051593
No. iteración: 9
Condición: 163.747680600121
No. iteración: 10
Condición: 4.04242087905004
No. iteración: 11
Condición: 0.00249124612309970
--------------------------------------------
Valores:
a_1 = 70995.3968045506
a_2 = 78.2055214202417
a_3 = 877.575841650900
```

4. Decaimiento de una fuente de voltaje

Cuando una fuente de voltaje se conecta a tavés de una resistencia y un inductor en serie, el voltaje a trabés del inductor $V_i(t)$ obedece la ecuación:

$$V(t) = V_0 e^{-\Gamma t} \tag{8}$$

donde t es el tiempo y $\Gamma = \frac{R}{L}$ es el cociente de la resistencia R y la inductancia L del circuito. Con base en los datos experimentales:

(a) Encuentre el mejor estimado para los valores de Γ y V_0.

Solución.
Sea $a_1 = V_0$ y $a_2 = \Gamma$. Así nuestra ecuación para hacer ajuste por mínimos cuadrados resulta:

$$g(t) = a_1 e^{-a_2 t} \tag{9}$$

Luego:

$$\frac{\partial g}{\partial a_1} = e^{-a_2 t}$$
$$\frac{\partial g}{\partial a_2} = -t a_1 e^{-a_2 t}$$

Así nuestras funciones f_1 y f_2 que minimizan chi cuadrada resultan:

$$f_1(a_1, a_2) = \sum_{i=1}^{16} \frac{[y_i - a_1 e^{-a_2 t_i}]}{\sigma_i^2} e^{-a_2 t_i}$$
$$f_2(a_1, a_2) = \sum_{i=1}^{16} \frac{[y_i - a_1 e^{-a_2 t_i}]}{\sigma_i^2} (-t_i a_1 e^{-a_2 t_i})$$

Igualándolas a cero se reducen a:

$$f_1(a_1, a_2) = \sum_{i=1}^{16} \frac{[y_i - a_1 e^{-a_2 t_i}]}{\sigma_i^2} e^{-a_2 t_i} = 0 \tag{10}$$

$$f_2(a_1, a_2) = \sum_{i=1}^{16} \frac{[y_i - a_1 e^{-a_2 t_i}]}{\sigma_i^2} t_i e^{-a_2 t_i} = 0 \tag{11}$$

Para determinar a_1 y a_2 usaremos el método de Newton-Raphson para dos variables.
Notemos que este problema es muy similar al ejercicio anterior. Por lo que reciclaremos el código del ejercicio 3, pero lo ajustaremos para las ecuaciones (9),(10) y (11).
Las librerías usadas fueron:

In [1]:
```
import pandas as pd
import sympy as sp
import numpy as np
import numpy.polynomial.polynomial as poly
import matplotlib.pyplot as plt
```

Las funciones definidas fueron:

```
def gx(i:int , values: list):
    """
    Esta función calcula el valor de g(x_i) usando el cálculo simbólico
    de sumpy

    Parámetros:
        i = número de índice de x_i, i =0,1,2,...n-1 con n = no. de datos
        x = lista que contiene los datos experimentales para obtener g(x_i)
    Salida:
        Regresa la ecuación g(x_i)
    """

    g, a1, a2 = sp.symbols('g, a1, a2') #Creamos las variables para el cálculo simbólico

    #Inicializamos las variables
    g = 0.

    #Esta parte calculo g(x_i) usando la fórmula

    g = a1*sp.exp(-a2*values[i])
    g = sp.expand(g)
    return g
```

```
def f_vector(f1,f2, x:float, y:float):
    """Esta función calcula el f vector en la solución de Newton-Rapghson y
    evalúa a las funciones f1, f2 en los puntos x,y.
    Regresa una matriz."""

    a1, a2 = sp.symbols('a1, a2')

    valor1 = f1.subs([(a1,x), (a2,y)])
    valor2 = f2.subs([(a1,x), (a2,y)])


    return np.array([[valor1], [valor2]])

def jacobiano(f1,f2, x:float, y:float):
    """Esta función calcula la matriz Jacobiana en la solución de Newton-Rapghson
    y evalúa a las derivadas parciales de f1, f2, f3 en los puntos x,y,z.
    Regresa una matriz."""

    df, a1, a2 = sp.symbols('df, a1, a2')

    variables = [a1, a2]
    funciones = [f1, f2]

    #Saca la derivada y evalua los valores de a1,a2 = x,y
    J = np.array([[np.float64(sp.diff(funciones[i], variables[j]).subs([(a1,x), (a2,y)]))
                for j in range(len(variables))] for i in range(len(variables))])

    return J
```

Lo siguiente es el cuerpo del código principal el cual se encarga de estimar a_1 y a_2 por Newton-Raphson:

```
# Importamos los datos de un archivo txt
data = pd.read_csv(r'C:\Users\cesar.avila\Documents\IPN\Semestre 7\Física Numérica\Tareas\Tarea6\DatoVoltage.txt'
                   ,header=0,delim_whitespace = True)

# Recopilamos los datos de las dos columnas
t = data.iloc[:,0]
V = data.iloc[:,1]
sigma = data.iloc[:,2]

# Transformamos los datos a dos arreglos x,y para su mejor manejo
data_x = np.float64(t)
data_y = np.float64(V)
sigma_y = np.float64(sigma)


#Guardamos el número de datos
n = len(data_x)

# Definimos variables para la obtendión g(x) y los inicializamos
g, a1, a2 = sp.symbols('g, a1, a2')
g = 0.

# Definimos variables para la obtendión f_1, f2, f3 y los inicializamos
f1, f2, aux = sp.symbols('f1, f2, aux')
f1 = 0.
f2 = 0.
```

```
for i in range(n):
    #Cálculo de g(x_i)
    g = gx(i,data_x)
    #Cálculo de  f1
    aux = (data_y[i] - g) * sp.exp(-a2*data_x[i]) / sigma_y[i]**2
    aux = sp.expand(aux)
    f1 += aux

    #Cálculo de  f2
    aux = (data_y[i] - g) * data_x[i] * sp.exp(-a2*data_x[i]) / sigma_y[i]**2
    aux = sp.expand(aux)
    f2 += aux

#Semillas de a1,a2,
x = 1.
y = 0.00001

# Longitud a1+Delta_a1,a2+Delta_a2, a3+Delta_a3

a = [0., 0.]
a_new = [x, y]

# Definimos el error para detener la iteración de Newton-Raphson
error = 10**(-20)
i=1

condicion1 = abs(a_new[0] - a[0])
```

Esta parte calcula iteradamente por Newton-Raphson:

```
#Iteraciones Newton-Raphson:
while  (condicion1 > error):
    A = f_vector(f1, f2, a_new[0], a_new[1])
    B = jacobiano(f1, f2, a_new[0], a_new[1])
    inv_B = np.linalg.inv(B)

    Delta_a = -inv_B.dot(A)

    #Guardamos los viejos valores
    a[0] = a_new[0]
    a[1] = a_new[1]

    #Asignamos los nuevos valores
    a_new[0] += np.float64(Delta_a[0][0])
    a_new[1] += np.float64(Delta_a[1][0])

    condicion1 = abs(a_new[0] - a[0])
    print(f'No. iteración: {i}')
    print(f'Condición: {condicion1}')
    i+=1

print('-'*50)
print(f'Valores:\na_1 = {a_new[0]}\na_2 = {a_new[1]}')
```

Los resultados fueron:

```
No. iteración: 1
Condición: 0.18833957009718816
No. iteración: 2
Condición: 0.23662530371362678
No. iteración: 3
Condición: 0.5614541732550196
No. iteración: 4
Condición: 0.8751231332017051
No. iteración: 5
Condición: 0.9606525935785575
No. iteración: 6
Condición: 0.757634102786926
No. iteración: 7
Condición: 0.4757632327253978
No. iteración: 8
Condición: 0.24266713986132515
No. iteración: 9
Condición: 0.0813043061961558
No. iteración: 10
Condición: 0.010600811190355941
No. iteración: 11
Condición: 0.00020256647960081864
No. iteración: 12
Condición: 8.230561210353926e-08
No. iteración: 13
Condición: 1.4210854715202004e-14
No. iteración: 14
Condición: 0.0
--------------------------------------------
Valores:
a_1 = 5.013687875197109
a_2 = 0.01221930618762891
```

Por lo tanto:

$$a_1 = V_0 = 5.0136 \, \text{Volts}$$
$$a_2 = \Gamma = 0.0122$$

(b) Encuentre el valor de χ^2 para su ajuste. ¿Tiene sentido?

Solución.
Recordemos que:

$$\chi^2 = \sum_{i=1}^{16} \left[\frac{y_i - a_1 e^{-a_2 t_i}}{\sigma_i} \right]^2$$

Donde usaremos los valores de a_1 y a_2 determinados en el ejercicio anterior. Este cálculo se realiza en el siguiente código:

```
In [2]: # Cálculo de chi^2 con los valores de a_1 = a_new[0] y a_2 = a_new[1]

chi = sp.symbols('chi') # Creamos las variables para el cálculo simbólico

#Inicializamos las variables
chi = 0.

for i in range(len(data_x)):
    aux = ((data_y[i] - a_new[0]*np.exp(- a_new[1] *data_x[i])) / sigma_y[i])**2
    chi += aux

print('-'*50)
print(f'Valor de chi^2: {chi}')
```

```
--------------------------------------------------
Valor de chi^2: 11.653063986177251
```

De esa forma $\chi^2 \approx 12$, lo cual es menor a nuestro número de datos 16. Como es un valor cercano al número de datos, podemos considerarlo como un buen ajuste.

(c) Realice una gráfica semi-log para los datos y el ajuste.

Solución.
Notemos que de (8) obtenemos:

$$\underbrace{\ln(V(t))}_{y} = \underbrace{-\Gamma}_{m} t + \underbrace{\ln(V_0)}_{b}$$

La cual es una ecuación de la forma $y = mx + b$. Para obtener m y b realizaremos un ajuste de recta por mínimos cuadrados.
Lo primero a realizar será obtener los valores logarítmicos y la dispersión de las incertidumbres dadas por:

$$\sigma_y = \frac{\partial y}{\partial V}\sigma_V = \frac{\sigma_V}{V}$$

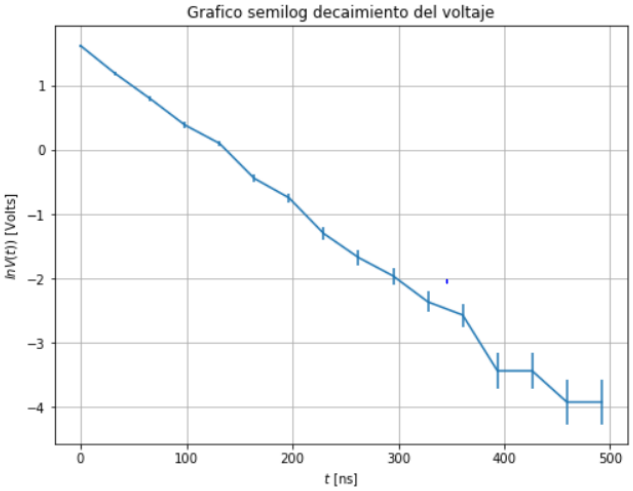
Esto se realiza en el siguiente código:

```
In [3]: # Hagamos una gráfica semilog para obtener ajuste por mínimos cuadrados a una recta

logaritmos = np.log(data_y)
incertidumbre_ln = sigma_y/data_y
```

Lo siguiente a realizar es graficar junto con las incertidumbres:

In [4]:
```
# Graficquemos y obtengamos el ajuste
fig, ax = plt.subplots(figsize=(8, 6))
ax.errorbar(data_x, logaritmos, yerr=incertidumbre_ln) # Grafica la incertidumbre en cada punto
ax.set_ylabel('$ln V(t))$ [Volts]')
ax.set_xlabel('$t$ [ns]')
ax.set_title('Grafico semilog decaimiento del voltaje')
ax.grid()
plt.show()
```



Podemos ver que se tiene una tendencia lineal, por lo que realizaremos el ajuste de recta por mínimos cuadrados. En el siguiente algoritmo se realiza esta parte:

In [5]:
```
# Obtengamos el ajuste de recta por mínimos cuadrados
S = 0.
Sx = 0.
Sy = 0.
Sxx = 0.
Sxy = 0.

for i in range(len(logaritmos)):
    S += 1/incertidumbre_ln[i]**2
    Sx += data_x[i]/incertidumbre_ln[i]**2
    Sy += logaritmos[i]/incertidumbre_ln[i]**2
    Sxx += data_x[i]**2 / incertidumbre_ln[i]**2
    Sxy += data_x[i]*logaritmos[i] / incertidumbre_ln[i]**2

# Solución al sistema de ecuaciones
matriz1 = np.array([[S, Sx],[Sx, Sxx]])
matriz2 = np.array([[Sy], [Sxy]])

inversa = np.linalg.inv(matriz1)
resultado = inversa.dot(matriz2)
# b = ordenada al origen y m = pendiente
b, m = float(resultado[0]), float(resultado[1])

print('-'*50)
print(f'Los valores de Gamma y V_0 por este ajuste son:')
print(f'Gamma: {-m}')
print(f'V_0: {np.exp(b)}')
```

```
--------------------------------------------------
Los valores de Gamma y V_0 por este ajuste son:
Gamma: 0.012139925808506362
V_0: 5.002014936967898
```

Por lo tanto, los valores determinados por este ajuste de recta son:

$$V_0 = 5.0020\,\text{Volts}$$
$$\Gamma = 0.01213$$

Los cuales coinciden mucho con el inciso (a).

5. Ajustando el espectro de un cuerpo negro

La Mecánica Cuántica inición con el espectro de radiación de un cuerpo negro de Planck:

$$I(\nu, T) = \frac{2h\nu^3}{c^2}\frac{1}{\exp(\frac{h\nu}{kT}) - 1} \tag{12}$$

donde $I(\nu, T)$ es la energía por unidad de tiempo de radiación con freceucnai nu emitida en la superficie por unidad de área, por unidad de ángulo sólido y por unidad de frecuencia por un cuerpo negro a temperatura T. El parámetro h es la constante de Panck, c es la velocidad de la luz en el vacío y k es la constante de Boltzmann. El proyecto COBE midió la radiación cósmica de fonde, obteniendo algunos datos experimentales.

(a) Grafique los datos del COBE y vea si tienen una forma similar a la radiación de cuerpo negro dada por Planck.

Solución.
Para graficar los datos usaremos el siguiente algoritmo, el cual se encarga de recopilar los datos en tres arreglos distintos. Note que las incertidumbres se han dividido entre 1000 con el objetivo de usar las mismas unidades de medición.

In [23]:
```
import pandas as pd
import numpy as np

#Leemos el archivo
data = pd.read_csv(r'C:\Users\cesar.avila\Documents\IPN\Semestre 7\Física Numérica\Tareas\Tarea6\COBE.txt'
                   ,header=0,delim_whitespace = True)

# Extraccion de datos
Nu = data.iloc[:,0]
I = data.iloc[:,1]
Error = data.iloc[:,2]

# Manejo de los datos a flotantes
nu_float = np.float64(Nu)
I_float = np.float64(I)
Error_float = np.float64(Error)/1000

fig, ax = plt.subplots()
ax.errorbar(nu_float, I_float, yerr=Error_float)
ax.set_xlabel('$\\nu$ [1/cm]')
ax.set_ylabel('$I(\\nu, T)$ [MJy/sr]')
ax.set_title('Experimento COBE')
ax.grid()
plt.show()
```

Produciendo el siguiente gráfico:

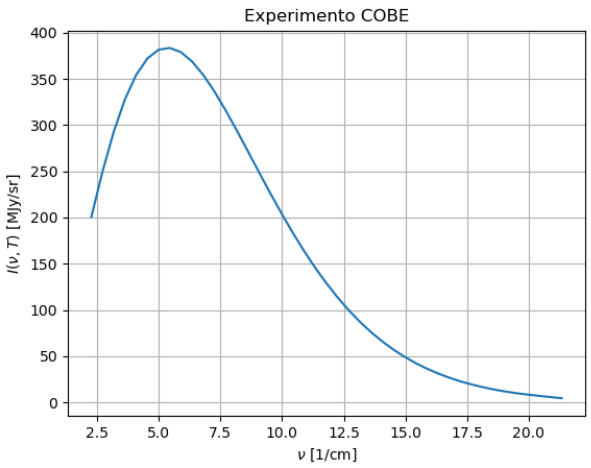


Figura 4: Dispersión de los datos experimentales de COBE.

Por otro lado, para la radiación de cuerpo negro de Planck se tiene que:

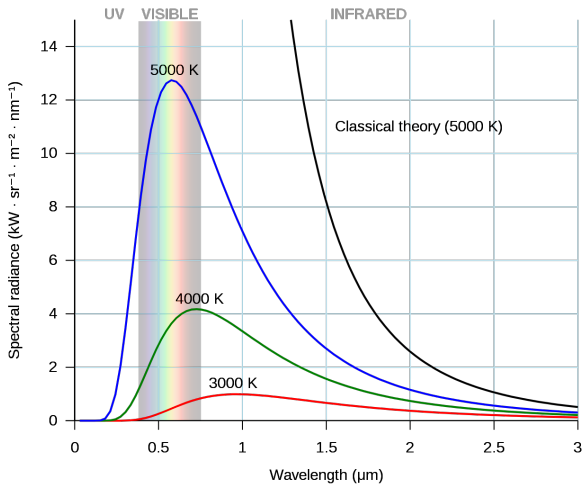


Figura 5: Radiación de cuerpo negro de Planck.

Podemos ver que la dispersión de los datos de COBE tienen la misma forma que la radiación de cuerpo negro de Planck.

(b) Utilice estos datos para estimar la temperatura T de la radiación cósmica de fondo, ajustando la curva por mínimos cuadrados.

Solución.
Para realizar el ajuste, de la ecuación (12) la modificaremos de forma que podamos obtener un ajuste lineal. La expresión obtenida es:

$$\underbrace{\ln\left(\frac{2h}{c^2}\frac{\nu^3}{I}+1\right)}_{y} = \frac{1}{T}\underbrace{\frac{h\nu}{k}}_{x} \tag{13}$$

Esta ecuación es de la forma $y = ax$. Nuestro objetivo será determinar la pendiente $a = \frac{h}{kT}$ para poder determinar el valor de T.

Lo primero que haremos será definir las constantes y hacer las conversiones adecuadas. Usaremos unidades del SI. De esa forma:

$$h = 6.6261 \times 10^{-34}\,\mathrm{J}\cdot\mathrm{s} \quad \text{(Cte. de Planck)}$$
$$c = 2.9979 \times 10^{8}\,\mathrm{m/s} \quad \text{(Vel. de la luz)}$$
$$k = 1.3806 \times 10^{-23}\,\mathrm{J/K} \quad \text{(Cte. de Boltzmann)}$$

Por otro lado, las unidades Jansky (Jy) están definidas de la siguiente forma:

$$1\,\mathrm{Jy} = 10^{-26}\frac{\mathrm{W}}{\mathrm{m}^2\cdot\mathrm{Hz}}$$

De esa forma, las mediciones para la irradiancia cuyas unidades son [MJy/sr] deberán multiplicarse por un factor de 1×10^{-20} para tener unidades del SI. Por otro lado, notemos que las mediciones para ν tienen como unidad [1/cm]. Esto se debe a que en ocasiones la frecuencia se toma como $\nu = \frac{1}{\lambda}$ donde λ es la longitud de onda. Para modificarlo y tener a la frecuencia en [Hz] tal y como la conocemos, recordemos que $\nu = \frac{c}{\lambda}$. De esta forma, si nuestra medición dice que tenemos $\nu = 10\,(\mathrm{cm})^{-1}$ en realidad tenemos $\nu = c_{cgs}\cdot 10\,\mathrm{Hz}$, donde $c_{cgs} = 2.9979 \times 10^{10}\,\mathrm{cm/s}$ (Vel. de la luz en cgs).
En las siguientes lineas de código realizamos este cambio:

```
In [26]: # Valores de las contantes (Usaremos unidades del SI)
         h = 6.6261E-34 # Cte de Planck en J*s
         c = 2.9979E8 # Vel. luz en m/s
         k =1.3806E-23 # Cte de Boltzmann en J/Kelvin

         #Transformamos los valores de la irradiancia y la frecuencia a unidades SI
         nu_SI = nu_float*2.9979E10
         I_SI = I_float*1E-20
         SigmaI_SI = Error_float*1E-20
```

Lo siguiente a realizar será crear tres listas: `logaritmos`, `incertidumbres_ln` y `eje_x`. En la lista `logaritmos` guardaremos los valores de y de la ecuación (13) para el ajuste lineal, en `eje_x` guardaremos los valores de x. Por último, en `incertidumbres_ln` guardaremos los valores de la dispersión de las incertidumbres usando la siguiente fórmula:

$$\sigma_y = \left|\frac{\partial y}{\partial I}\right|\sigma_I = \frac{\frac{2h\nu^3}{c^2 I^2}}{\frac{2h}{c^2}\frac{\nu^3}{I} + 1}$$

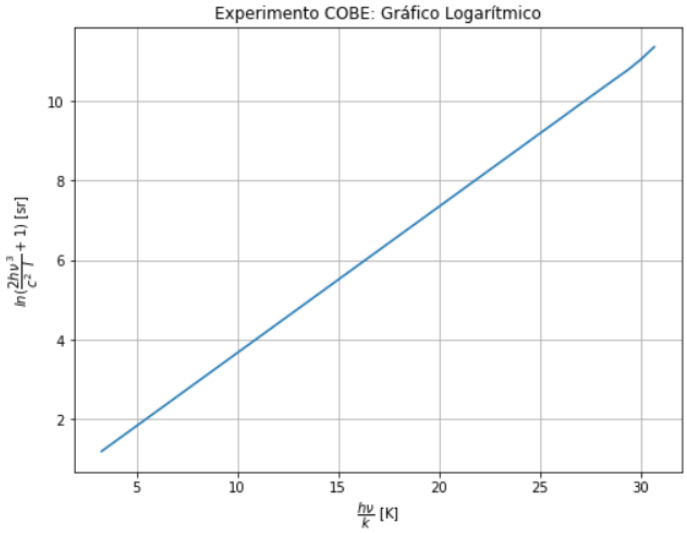
Para guardar los valores en estas tres listas se utilizó un ciclo `for` tal como se aprecia en la siguiente imagen:

```
In [ ]: #Guadamos los valores para el ajuste de recta en tres listas distintas
        logaritmos = []
        incertidumbre_ln = []
        eje_x = np.float64(h*nu_SI/k)

        for i in range(len(nu_SI)):
            valor1 = np.log((2*h*nu_SI[i]**3)/(c**2 * I_SI[i]) +1 )
            valor2 = (2*h*nu_SI[i]**3 / c**2 * I_SI[i]**2) *SigmaI_SI[i] / ((2*h*nu_SI[i]**3)/(c**2 * I_SI[i]) +1)
            logaritmos.append(np.float64(valor1))
            incertidumbre_ln.append(np.float64(valor2))
```

Con estos valores ya podremos realizar un ajuste lineal. Para ver la tendencia de los datos, grafiquemos y vs x (`logartimos` vs `eje_x`).

In [29]:
```
fig, ax = plt.subplots(figsize=(8, 6))
ax.errorbar(eje_x, logaritmos, yerr=incertidumbre_ln) # Grafica la incertidumbre en cada punto
ax.set_xlabel('$\dfrac{h \\nu}{k}$ [K]')
ax.set_ylabel('$ln (\dfrac{2 h}{c^2} \dfrac{\\nu^3}{I} + 1)$ [sr]')
ax.set_title('Experimento COBE: Gráfico Logarítmico')
ax.grid()
plt.show()
```



Podemos ver que la tendencia es lineal, por lo que será posible realizar un ajuste por mínimos cuadrados. DE la teoría sabemos que el ajuste propuesto es de la forma:

$$g(x) = a_1 + a_2 x$$

Determinaremos a_1 y a_2. De esta forma la temperatura buscada es $T = 1/a_2$. Para ello requeriremos obtener S, S_x, S_y, S_{xx} y S_y tal y como se definieron en clase. En la siguiente parte de código se realiza esta parte:

In [30]:
```
# Obtengamos el ajuste por mínimos cuadrados
S = 0.
Sx = 0.
Sy = 0.
Sxx = 0.
Sxy = 0.

for i in range(len(logaritmos)):
    S += 1/incertidumbre_ln[i]**2
    Sx += eje_x[i]/incertidumbre_ln[i]**2
    Sy += logaritmos[i]/incertidumbre_ln[i]**2
    Sxx += eje_x[i]**2 / incertidumbre_ln[i]**2
    Sxy += eje_x[i]*logaritmos[i] / incertidumbre_ln[i]**2
```

Con estos valores recordemos también que se tienen el siguiente sistema de ecuaciones:

$$S_y - Sa_1 - S_x a_2 = 0$$
$$S_{xy} - S_x a_1 - S_{xx} a_2 = 0$$

De forma matricial:

$$\begin{bmatrix} S & S_x \\ S_x & S_{xx} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} S_y \\ S_{xy} \end{bmatrix}$$

Basta despejar la segunda matriz para obtener los valores de a_1 y a_2. Esto se realiza en el siguiente código:

In [31]:
```
# Solución al sistema de ecuaciones
matriz1 = np.array([[S, Sx],[Sx, Sxx]])
matriz2 = np.array([[Sy], [Sxy]])

inversa = np.linalg.inv(matriz1)
resultado = inversa.dot(matriz2)

a1, a2 = float(resultado[0]), float(resultado[1])
print(a1, a2)
```

```
-0.6969042846340017 0.3915557263619007
```

Por lo que la temperatura resultante es:

In [32]:
```
print(f'La temperatura es: {1/a2} K')
```

```
La temperatura es: 2.553914890458622 K
```