Física Numérica

Julio César Avila Torreblanca

02 de diciembre del 2021

Tarea 7

1. Ecuación de Poisson

- Resuelva numéricamente la ecuación de Poisson:

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = f(x,y)$$

con:

$$f(x,y) = \cos(3x+4y) - \cos(5x-2y)$$

sujeta a las condiciones de frontera periódica:

$$\phi(x,0) = \phi(x,2\pi), \quad \phi(0,y) = \phi(2\pi,y)$$

Indique el método elegido (Jacobi o Gauss-Seidel), presente su resultado en forma gráfica.

Solución.
En clase se desarrollo la ecuación de Poisson en 2D de forma numérica:

$$\nabla^2\phi(x,y) = -\frac{\rho}{\varepsilon_0}$$

Llegando a una solución de la forma discretizada:

$$\phi_{i,j} = \frac{\rho_{i,j}}{4\varepsilon_0} + \frac{1}{4}\left[\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1}\right]$$

Para este problema la densidad ρ junto con la constante ε_0 serán sustituidas por la función $f(x,y)$ dada. Además, utilizaremos el método de Gauss-Seidel para obtener la solución. En este método debemos utilizar los últimos valores del potencial calculados. Es decir, que en cada iteración deberemos reescribir los antiguos valores del potencial por los nuevos y con ello la velocidad de convergencia será mayor. Haciendo estos ajustes, nuestra ecuación discretizada nos queda:

$$\phi^{new}_{i,j} = \frac{1}{4}\left[\phi^{old}_{i+1,j} + \phi^{new}_{i-1,j} + \phi^{old}_{i,j+1} + \phi^{new}_{i,j-1} + f(x,y)\Delta^2\right]$$

Donde $\Delta^2 = 2\pi/N$ con N el número de nodos en un lattice de $N \times N$.

Para comenzar con el algoritmo requeriremos de las siguientes librerías:

```
import numpy as np
import matplotlib.pylab as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
```

Lo siguiente a realizar es crear una función que nos calcule la parte $f(x,y) = \cos(3x+4y) - \cos(5x - 2y)$ en el problema.

```
#Funciones
def f(x, y):
    return np.cos(3*x + 4*y) - np.cos(5*x - 2*y)
```

Ahora debemos definir dos arreglos de longitud N llenos de números aleatorios. Además, se deben crear dos lattices para ir guardando las soluciones viejas y nuevas. El lattice viejo lo podemos llenar de números aleatorios. Finalmente, se crea otro arreglo donde guardaremos los valores para la función $f(x,y)$.

```
#Parámetros del lattice
N = 50 # N0. de nodos
x = np.linspace(0.0, 2.0*np.pi, N) # No. aleatorios en los arreglos
y = np.linspace(0.0, 2.0*np.pi, N)
Delta = 2.0*np.pi/len(x) # Longitud de los pasos
lat = np.random.rand(N, N)
lat_new = np.zeros((N, N), float)
F= np.zeros((N, N), float) #Genera un arreglo de NxN
for index, y_val in enumerate(y):
    valor = f(x, y_val)
    F[index] = valor
```

Ya que tenemos los lattices es necesario establecer nuestras condiciones de frontera al lattice viejo. Nosotros fijaremos estos puntos a 5 V.

```
# Condiciones de fronetera:
cond_fron = 5.0
for i in range(0, N):
    lat[0, i] = cond_fron
    lat[N - 1, i] = cond_fron
    lat[i, 0] = cond_fron
    lat[i, N - 1] = cond_fron
```

Enseguida implementaremos el método de Gauss-Seidel. Para ello es necesario tener un parámetro de tolerancia `eps` que servirá para comparar punto a punto el lattice nuevo del viejo. Es decir, en cada iteración se obtendrá la diferencia punto a punto del lattice nuevo menos el viejo, y si la diferencia de cada punto es menor a `eps` habremos terminado con la recursión. Esto se observa en el siguiente algoritmo:

```
eps = 1.e-3 #Condición de tolerancia
count = 1
for i in range(N):
        for j in range(N):
            lat_new[i, j] = lat[i, j]

# Implementación del algoritmo de Gauss-Seidel
while True:
    for i in range(1, len(x) - 1):
        for j in range(1, len(y) - 1):
            lat_new[i, j] = (lat[i+1, j] + lat_new[i-1, j] + lat[i, j+1]
                             + lat_new[i, j-1] + F[i, j]*Delta)/4.0
    # Verifica la condición de tolerancia
    if np.allclose(lat_new, lat, rtol=eps):
        break

    #Actualizamos los puntos del lattice
    for i in range(N):
        for j in range(N):
            lat[i, j] = lat_new[i, j]

    count += 1
    print(f"Iteración: {count}")
```

Hasta esta parte ya habremos generado el lattice final que contendrá los valores adecuados para nuestra solución. Solo resta determinar las alturas y graficar el resultado.

```
# Graficación
x = list(range(0, N))
y = list(range(0, N))

#Creamos la maya en el plano XY
X , Y = plt.meshgrid(x, y)

def altura(lat):
    z = lat[X, Y]
    return z

#Encontremos la altura en cada punto (X,Y)
Z = altura(lat_new)

fig = plt.figure(1)
ax = Axes3D(fig)
surf1 = ax.contour(X, Y, Z, N, cmap=cm.CMRmap)
fig.colorbar(surf1, shrink=0.5, aspect=8)
ax.contour(X, Y, Z, zdir='z', offset=-0)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('$\phi(x,y)$')
plt.show()
```

Como producto final obtenemos:

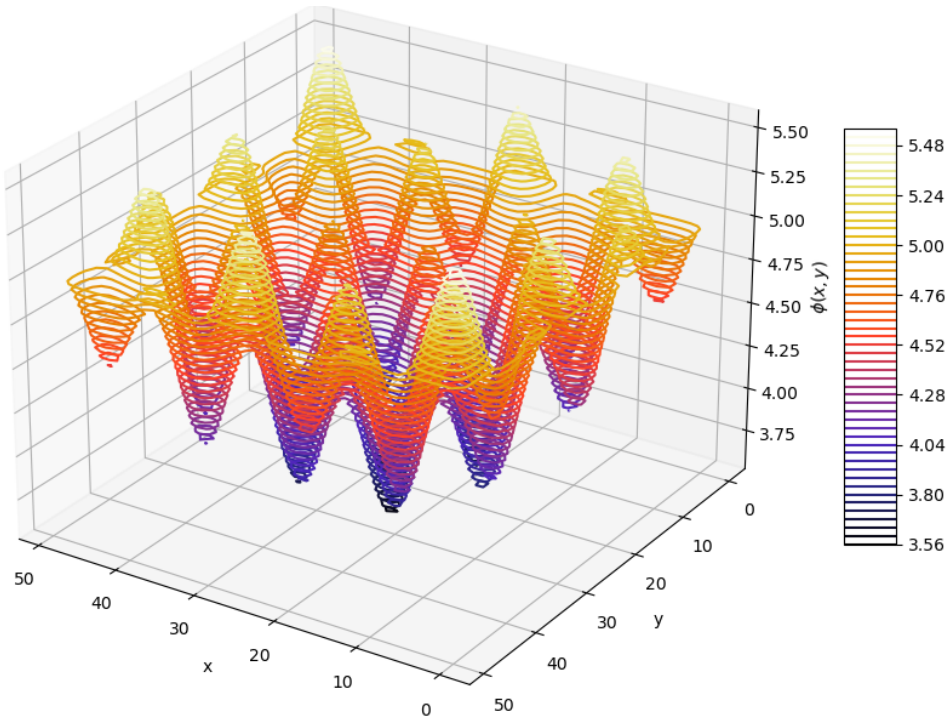


Figura 1: Solución a la ecuación de Poisson dada.

---

2. Estados ligados en un potencial arbitrario

- Una partícula cuántica en un estado estacionario de energía definida E se encuentra ligada por un potencial $1D$. Su función de onda se determina por la ecuación de Schrödinger independiente del tiempo:

$$\frac{d^2\psi(x)}{dx^2} - \frac{2m}{\hbar^2}V(x)\psi(x) = \kappa^2\psi(x), \quad \kappa^2 = -\frac{2m}{\hbar^2}E$$

Cuando la partícula está ligada, tenemos que está confinada a cierta región *finita* del espacio, por lo que $\psi(x)$ es normalizable. Esto nos dice que la energía debe ser negativa para que $\psi(x)$ decaiga exponencialmente cunado $x \to \pm\infty$:

$$\psi(x) \to \begin{cases} e^{-\kappa x}, & \text{cuando} x \to +\infty \\ e^{+\kappa x}, & \text{cuando} x \to -\infty \end{cases}$$

Aunque la ecuación puede resolverse con algún método numérico (Runge Kutta, por ejemplo), el extra aquí es que también debemos tener la solución $\psi(x)$ debe satisfacer las condiciones de frontera anteriores. Esta condición adicional convierte al problema de la EDO en un problema de eigenvalores en el cual existe solución solo para ciertos valores del eigenvalor E. La solución, si existe, se sigue de encontrar una energía permitida, resolver la ecuación y después variar la energía como parte de un problema de prueba y error (búsqueda de raíces) para la función de onda que satisfaga las condiciones de frontera.

Solución.
La ecuación que queremos resolver es:

$$\frac{d^2\psi(x)}{dx^2} = \frac{2m}{\hbar^2}\left(V(x) - E\right)\psi(x) \tag{1}$$

Donde hemos sustituido el valor de κ. Notemos que es una EDO de segundo orden, por lo que debemos transformarla usando la forma estándar.
Sean:

$$y^{(0)}(x) = \psi(x) \quad ; \quad y^{(1)}(x) = \frac{d\psi(x)}{dx}$$

Por lo que:

$$y^{(1)}(x) = \frac{dy^{(0)}(x)}{dx} \tag{2}$$

$$\frac{dy^{(1)}(x)}{dx} = \frac{2m}{\hbar^2}(V(x) - E)y^{(0)}(x) \tag{3}$$

Estas serán las EDO's a resolver en nuestro algoritmo.
Para nuestro programa utilizaremos las siguientes librerías:

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
```

En la ecuación (3) utilizaremos el siguiente valor para la constante:

$$\frac{2m}{\hbar^2} = \frac{2mc^2}{(\hbar c)^2} = \frac{2(0.511\,\text{MeV})}{(0.197\,\text{MeV}\cdot\text{pm})^2} \approx 26.33\,\text{MeV}^{-1}\text{pm}^{-2}$$

Donde hemos usado la masa del electrón $m = 0.511\,\text{MeV/c}^2$. Definimos esta constante en nuestro algoritmo como `Cte`:

```
In [2]: ###############Constante 2m/(hc)^2
        m = 0.511 # masa del electrón [MeV /c^2]
        hc = 0.197 # Cte Planck*c [MeV * pm
        Cte = 2.*m/(hc*hc) #Constente en [MeV^-1 pm^-2]
```

Siguiente con el orden sugerido, crearemos una función llamada `f(x,y)` cuyo propósito es calcular las funciones $f^{(0)}$ y $f^{(1)}$ cuya forma estándar están dadas por:

$$f^{(0)}(x) = y^{(1)}(x)$$
$$f^{(1)}(x) = \frac{2m}{\hbar^2}(V(x) - E)y^{(0)}$$

Asumiremos que E está dado. Este valor más adelante se obtendrá por el método de Bisección. La función `f(x,y)` con su respectiva documentación (comentada) está dada por:

In [4]:
```
def f(x, y):
    """Esta función calcula las funciones f^(0) y f^(1) de la forma estándar
    en la solución de la ODE.

    Parámetros:
        x: punto a ser evaluado
        y: arreglo que contiene los valores de y^(0) y y^(1) de a forma estándar
            en la solición de la EDO

    Salida:
        f: arregloq ue contiene las funciones f^(0) y f^(1) evaluadas
    """
    f = np.zeros(2,float)
    f[0] = y[1]               #Forma estándar f^(0)
    f[1] = Cte*(V(x) -E)*y[0] #Forma estándar f^(1)
    return f
```

La función contiene comentarios donde se definen los parámetros y la salida.
Lo siguiente a realizar es definir el potencial $V(x)$. Tomaremos un potencial constante de la siguiente forma:

$$V(x) = \begin{cases} -10\,\text{MeV}, & |x| \leq a \\ 0 & |x| > a \end{cases} \tag{4}$$

Donde consideraremos $a = 10\,\text{pm}$ como la mitad del ancho del pozo, de forma que en el eje x el pozo de potencial finito empieza en $-a$ y termina en a. La función que contiene esta parte está dada por:

In [5]:
```
#Constantes para el potencial
a = 10. #Longitud del pozo en [pm]
Vx = -10. #Potencial en [Mev]

def V(x: float)->float:
    """Esta función calcula el potencial cumpliendo con la condición
    en el infinito

    Parámetros:
        x: punto donde se quiere obtener el potencial
    Salida:
        Valor del potencial
    """

    if abs(x) <= a :    return Vx
    else:   return 0.
```

Lo que sigue es resolver la EDO, para ello nos apoyaremos del algoritmo de Runge-Kutta de cuarto orden (rk4). Este algoritmo se vio en clase y está dado por:

In [11]:
```
def rk4Algor(x,h,N,y,f):
    """Esta función resulve la ODE por runge kutta orden 4

    Parámetros:
        x: puntos donde se evaluará
        h: longitud del salto en el cálculo de la integral
        N: longitud de los arreglos para el cálculo en varias variables
        y: variable donde se guardaran los resultados
        f: funcion que calcula las formas estándares f^(0) y f^(1) en el algoritmo

    Salida:
        y: un arreglo que contiene todos los puntos evaluados en la solución de la EDO

    """
    k1=np.zeros(N); k2=np.zeros(N); k3=np.zeros(N); k4=np.zeros(N)
    k1 = h*f(x,y)
    k2 = h*f(x+h/2.,y+k1/2.)
    k3 = h*f(x+h/2.,y+k2/2.)
    k4 = h*f(x+h,y+k3)
    y=y+(k1+2*(k2+k3)+k4)/6.
    return y
```

Ya con el algoritmo de rk4 procederemos a crear una función llamada `diff` cuyo objetivo será iniciar la función de onda por izquierda y por derecha en el X que consideraremos como infinito y enseguida se usará el algoritmo de Runge-Kutta para integrar desde el "infinito" hasta el punto de pegado y ahí evaluaremos la condición:

$$\Delta(E,x) = \frac{\frac{\psi'_{izq}(x)}{\psi_{izq}(x)} - \frac{\psi'_{dcha}(x)}{\psi_{dcha}(x)}}{\frac{\psi'_{izq}(x)}{\psi_{izq}(x)} + \frac{\psi'_{dcha}(x)}{\psi_{dcha}(x)}}$$

Tomaremos a $X = 5a$ nuestro infinito (a es la mitad del ancho del pozo). Además, para la función de onda ψ_{izq} por la izquierda en menos infinito consideraremos la aproximación:

$$\psi_{izq}(x = -X) = e^{-\kappa(x=X)} = e^{-\kappa X}$$

Por lo que la derivada en "infinito" es:

$$\psi'_{izq}(x = -X) = \kappa e^{-\kappa X}$$

En el siguiente fragmento del algoritmo se ve implementada esta primer parte:

```
In [ ]: Xinf = a*5 #Infinito -> no hacer muy grande porque si no tocamos overflow

        def diff(h, E):
            kappa = np.sqrt(-E*Cte)
            f_onda = 1/np.exp(kappa*Xinf)

            y = np.zeros(2, float)      #Arreglo que contendrá la función de onda y su derivada
            x_m = Nsteps//3            #Encajando el x_m de pegado para los pasos
            nL = x_m + 1
            y[0] = f_onda       #Inicialización de la función de onda izq en infinito
            y[1] = kappa*y[0]   #Derivada de la función de onda izq en infinito
            for ix in range(0, nL+1):  #Va de -X_inf hasta el punto de pegado
                x = h * (ix - Nsteps/2)
                y = rk4Algor(x, h, 2, y, f)
            izq= y[1]/y[0]        #Derivada logarítmica por izq
```

Notemos que hemos pedido como parámetros la energía E que se encontrará más adelante por bisección y la constante h que contendrá la longitud de los pasos para la integración por Runge-Kutta. Esta parte de la función solo hace el cálculo para la derivada logarítmica $\psi'_{izq}(x)/\psi_{izq}(x)$ en el punto de pegado. Hace falta calcular por la derecha.
Para la parte derecha es muy similar, solamente que en lugar de integrar hacia delante debemos integrar hacia atrás. Esto lo modificaremos en la función `rk4Algor` considerando la longitud de paso como $-h$ para ir hacia atrás. Haciendo esta modificación y ajustando el número de pasos para la parte derecha nos queda la función completa como:

```
In [ ]: Xinf = a*5 #Infinito -> no hacer muy grande porque si no tocamos overflow

def diff(h, E):
    kappa = np.sqrt(-E*Cte)
    f_onda = 1/np.exp(kappa*Xinf)

    #Cálculo por la izquierda
    y = np.zeros(2, float)      #Arreglo que contendrá la función de onda y su derivada
    x_m = Nsteps//3            #Encajando el x_m de pegado para los pasos
    nL = x_m + 1
    y[0] = f_onda       #Inicialización de la función de onda izq en infinito
    y[1] = kappa*y[0]   #Derivada de la función de onda izq en infinito
    for ix in range(0, nL+1):  #Va de -X_inf hasta el punto de pegado
        x = h * (ix - Nsteps/2)
        y = rk4Algor(x, h, 2, y, f)
    izq= y[1]/y[0]        #Derivada logarítmica por izq

    #Cálculo por la derecha
    y[0] = f_onda         #Inicializa la función de onda dcha en infinito
    y[1] = -kappa*y[0]    #Derivada de la fun. de onda derecha en infinito
    for ix in range (Nsteps, nL + 1, -1):  #Va de X_inf hasta el x_m en reversa
        x = h*(ix + 1 - Nsteps/2)
        y = rk4Algor(x, -h, 2, y, f)    #Integración en reversa
    dcha = y[1]/y[0]              #Derivadas logaritmicas

    |
    return ((izq- dcha)/(izq + dcha))
```

Notemos que la función anterior nos regresa el cálculo de $\Delta(E, x)$ en el punto de pegado. Más adelante buscaremos que este valor sea menor que un ε dado.
La siguiente parte que sigue para el algoritmo es generar los calcular las funciones de onda normalizadas por izquierda y por derecha para después pegarlas juntas en un gráfico. La función que calculará estas funciones de onda normalizadas se llamará `plot(h,E)` con parámetros h la longitud del paso y E la energía. La primer parte de la función que calcula la parte izquierda es la siguiente:

```
In [10]: def plot(h,E):
             Lwf = []    #Función de onda por la izquierda
             Rwf = []    #Función de onda por la derecha
             xR = []     #x para Rwf
             xL = []     #x para Lwf
             Nsteps = 1501       #No. de pasos para la integración de la ODE
             y = np.zeros(2, float) #Arreglo apr ala fun de onda y su derivada
             yL = np.zeros ((2,505), float) #Arreglo auxiliar para calcular las fun de onda
             i_match = 500   #Radio de pegado
             nL = i_match + 1

             #Cálculo de la fun. de onda en el infinito
             kappa = np.sqrt(-E*Cte)
             f_onda = 1/np.exp(kappa*Xinf)


             #Cálculo de la fun de onda por la izquierda
             y[0] = f_onda      #Función de onda izquierda inicial
             y[1] = kappa*y[0]  #Derivada valuada en x<0

             for ix in range(0, nL + 1):
                 yL[0][ix] = y[0]
                 yL[1][ix] = y[1]
                 x = h * (ix - Nsteps/2)
                 y = rk4Algor(x, h, 2, y, f)
```

Dentro de la función se tuvo que generar un arreglo `yL` auxiliar para guardar los valores de la función de onda izquierda y su derivada. Notemos que este algoritmo es muy parecido al de la función `diff`, solamente que aquí se usaron más arreglos auxiliares.
La otra parte de la función está dada por:

```
    #Cálculo función de onda por derecha
    y[0] = f_onda
    y[1] = kappa*y[0]
    for ix in range(Nsteps - 1, nL + 2, -1):    #Función de onda derecha
        x = h * (ix + 1 - Nsteps/2)     #Integración
        y = rk4Algor(x, -h, 2, y, f)
        xR.append(x)
        Rwf.append(y[0])
    #Normalización para que la onda izquierda encaje con la derecha
    norm = y[0]/yL[0][nL]
    for ix in range(0,nL + 1):  #Normaliza la función de onda izquierda y la deriva
        x = h*(ix - Nsteps/2 +1)
        y[0] = yL[0][ix]*norm
        y[1] = yL[1][ix]*norm
        xL.append(x)
        Lwf.append(y[0])
    return xL,Lwf,xR,Rwf
```

El cálculo de la onda por la derecha es análogo. Luego, para hacer que coincidiera la onda izquierda y derecha se hizo una normalización para la onda izquierda. Simplemente hay que multiplicar por un factor de escala que contiene a la función de onda por derecha en el punto de pegado entre la función de onda izquierda en el punto de pegado. El algoritmo completo es el siguiente:

```
In [10]: def plot(h,E):
             """Esta función realiza el cálculo de la fun. de onda por izquierda y
             por derecha con las condiciones de frontera y las normaliza.

             Parámetros:
                 h: longitud de los pasos para la integración
                 E: energía para el cálculo de la conste
             Salidas:
                 xL: arreglo con los puntos x para la fun de onda izq
                 Lwf: arreglo con los puntos de la fun de onda izq valuada en cada punto
                     de xL
                 xR: arreglo con los puntos x para la fun de onda dcha
                 Rwf: arreglo con los puntos de la fun de onda dcha valuada en cada punto
                     de xR
             """
             Lwf = []    #Función de onda por la izquierda
             Rwf = []    #Función de onda por la derecha
             xR = []     #x para Rwf
             xL = []     #x para Lwf
             Nsteps = 1501       #No. de pasos para la integración de la ODE
             y = np.zeros(2, float) #Arreglo apr ala fun de onda y su derivada
             yL = np.zeros ((2,505), float) #Arreglo auxiliar para calcular las fun de onda
             i_match = 500   #Radio de pegado
             nL = i_match + 1

             #Cálculo de la fun. de onda en el infinito
             kappa = np.sqrt(-E*Cte)
             f_onda = 1/np.exp(kappa*Xinf)


             #Cálculo de la fun de onda por la izquierda
             y[0] = f_onda      #Función de onda izquierda inicial
             y[1] = kappa*y[0]  #Derivada valuada en x<0

             for ix in range(0, nL + 1):
                 yL[0][ix] = y[0]
                 yL[1][ix] = y[1]
                 x = h * (ix - Nsteps/2)
                 y = rk4Algor(x, h, 2, y, f)

             #Cálculo función de onda po derecha
             y[0] = f_onda
             y[1] = kappa*y[0]
             for ix in range(Nsteps - 1, nL + 2, -1):    #Función de onda derecha
                 x = h * (ix + 1 - Nsteps/2)     #Integración
                 y = rk4Algor(x, -h, 2, y, f)
                 xR.append(x)
                 Rwf.append(y[0])


             normL = y[0]/yL[0][nL]
             for ix in range(0,nL + 1):  #Normaliza la función de onda izquierda y la deriva
                 x = h*(ix - Nsteps/2 +1)
                 y[0] = yL[0][ix]*normL
                 y[1] = yL[1][ix]*normL
                 xL.append(x)
                 Lwf.append(y[0])
             return xL,Lwf,xR,Rwf
```

Ya con todas estas funciones podremos realizar la parte central del código donde determinaremos

la energía por bisección. En el siguiente algoritmo se realiza la parte de bisección junto con la graficación de cada aproximación obtenida en cada una de las iteraciones.

In [21]:
```
####Parte principal
fig = plt.figure()
ax = fig.add_subplot(111)
ax.grid()

################ Constantes
eps = 1e-2; Nsteps = 501; h = 0.04; Nmax = 100 #Parámetros
E = -17.0; Emax = 1.2*E; Emin = E/1.2

for i in range (0, Nmax):
    E = (Emax + Emin)/2                #Rango de bisección
    #Algoritmo de bisección
    Diff = diff(h, E)
    Eaux = E
    E = Emax
    diffMax = diff(h,E)
    E = Eaux
    if (diffMax * Diff > 0):
        Emax = E
    else:
        Emin = E
    print(f'Iteración {i}, E = {E:.4f}')

    fig.clear()
    x_izq, onda_izq, x_dcha, onda_dcha = plot(h,E)
    plt.plot(x_izq, onda_izq)
    plt.plot(x_dcha,onda_dcha)
    plt.pause(0.8)  #Pausa entre figuras

    if (abs(Diff) < eps ):  break


plt.xlabel('x')
plt.ylabel('$\psi(x) $', fontsize = 18)
plt.title('Funciones de onda Izq. y Dcha. pegadas en x = -a')

print(f'Eigenvalor final E = {E:.4f}')
print(f'Iteraciones = {_}, max = {Nmax}')
plt.show()
```

La variable `eps` contiene nuestra la tolerancia para la aproximación. Además hemos colocando un número `Nmax` de iteraciones para no entrar en un bucle infinito si no hay solución.
La primer parte del ciclo `for` contiene la parte del algoritmo de bisección e imprime en cada iteración la energía estimada. Notemos que por sugerencia $|E| < V_0$ para tener una buena aproximación. Así que hay que tener cuidado con esta condición y con los puntos `Emax` y `Emin` que contendrán el intervalo donde se realizará el algoritmo de bisección.
La última parte del algoritmo se encarga crear una gráfica para cada iteración y reescribirla para observar el cambio. Además nos arroja el eigenvalor final para la energía.
Pongamos a prueba nuestro programa con la siguiente definición de constantes:

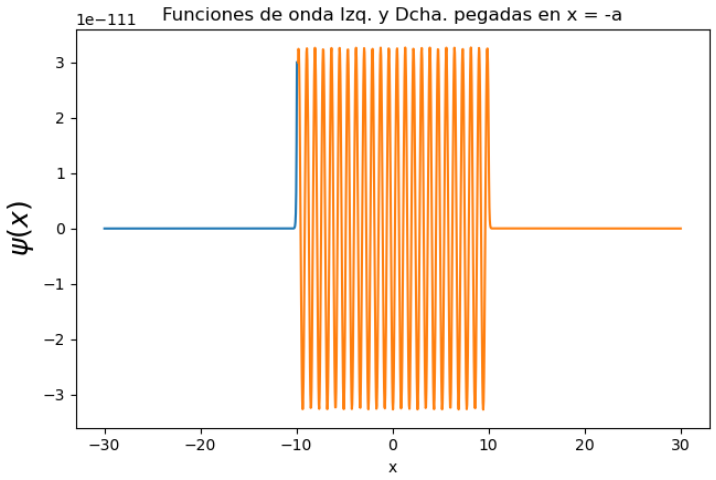
```
################ Constantes
eps = 1e-2; Nsteps = 501; h = 0.04; Nmax = 100 #Parámetros
E = -17.0; Emax = 1.1*E; Emin = E/1.1

################Constante 2m/(hc)^2
m = 0.511 #masa del electrón [MeV /c^2]
hc = 0.197 #0.4829 Planck*c [MeV * pm]
Cte = 2.*m/(hc*hc)

################Constantes del pozo de potencial
a = 10. #Longitud del pozo en [pm]
Vx = -20. #Potencial en [Mev]
Xinf = a*3#Infinito -> no hacer muy grande porque si no tocamos overflow
```

Aquí hemos usado la masa de un electrón como ya se mencionó antes. Esto nos produce el siguiente

eigenvalor junto con el número de iteraciones:

```
Eigenvalor final E = -17.9459
Iteraciones = 13, max = 100
```

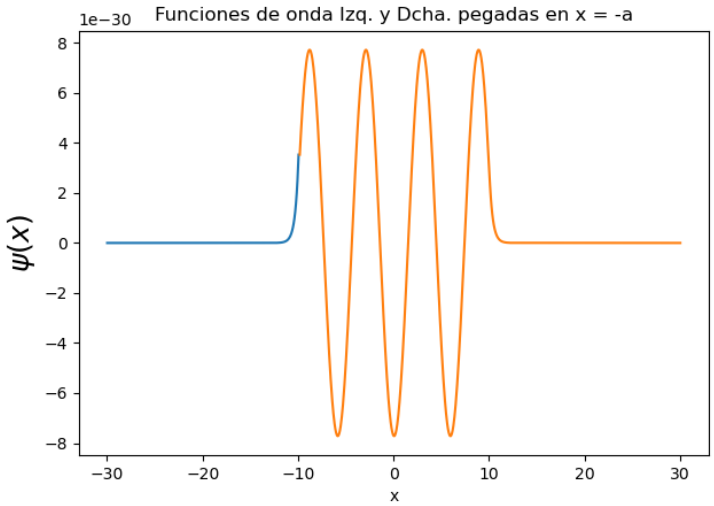
Y el gráfico final es el siguiente:



Se ve que la función oscila mucho dentro del pozo. Esto se debe a que hemos considerado la masa del electrón. Si modificamos el valor de la variable `Cte` en nuestro algoritmo por `Cte`$= 0.4829$ (un valor relativamente adecuado) obtenemos el siguiente resultado para el eigenvalor:

```
Eigenvalor final E = -17.6509
Iteraciones = 9, max = 100
```

Y para la solución gráfica:



En esta última se aprecia mucho mejor el comportamiento de la función de onda.