Ruiz Muñoz, Juan Manuel - PRÁCTICA 2

May 10, 2024

En esta práctica implementaremos el método variacional para diferentes pozos de potencial.

[109]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import quad
from scipy.optimize import minimize
from scipy.misc import derivative
```

Si necesitamos determinar la energía del estado base de un sistema gobernado por el hamiltoniano H, pero no podemos resolver la ecuación de Schrödinger para obtener una solución analítica, podemos recurrir al principio variacional. Este principio nos permite encontrar un límite superior para la energía del estado base y, si hacemos una elección adecuada, podemos obtener un valor muy próximo al valor real. Para lograr esto, seleccionamos una función de onda de prueba normalizada y calculamos el valor esperado del hamiltoniano (energía), que siempre será mayor que la energía del estado base que buscamos, y si es la correcta, será igual. Además, si dejamos este valor en función de algún parámetro, podemos minimizarlo para obtener la energía más baja posible.

[145]:
```
m=1
h=1

N = 8

a= np.ones(N+1,  dtype=int)

E_theoric = (np.pi**2)/4

print("E_theoric = ", E_theoric)
```

```
E_theoric =  2.4674011002723395
```

En términos de programación, lo que haremos en nuestro caso es establecer los valores iniciales para los 'an' en 1, y minimizaremos la función con respecto a estos valores. Para hacer esto, crearemos una función que sea equivalente a los polinomios simetrizados n(x), y luego construiremos manualmente las sumas (una alternativa sería crear un bucle que genere las sumas proporcionándoles un parámetro, lo que optimizaría el programa pero no hemos tenido tiempo para ello dado los errores que hemos ido teniendo). En este caso, tendremos cuatro funciones, para N=1, 2, 3 y 4 (probaremos para 8 también) respectivamente:

Definimos primero algunas funciones que utilizaremos. La función energía En(n) y el hamiltoni-

ano H(fun), donde sus entradas serán el valor de n, que será entera natural y la función de onda.

[119]:
```
def En(n):
    return ((n*np.pi*h)**2)/(2*m*a**2)

def H(fun):
    derivada_primera=derivative(fun,0)
    derivada_segunda=derivative(derivada_primera,0)
    return -(h**2)/(2*m)*derivada_segunda
```

Ahora definimos la función de onda phi:

[120]:
```
def Psi_n(x,n,a):
    return ((2/a[n])**0.5)*np.sin((n*np.pi(x-a[n]/2))/(a[n]))

def polPhi(x,N,n):
    return (1-x)**(N-n+1)*(x+1)**(n+1)
```

[121]:
```
def fi1(x,a):
    return a[0]*polPhi(x,1,0)+a[1]*polPhi(x,1,1)

def fi2(x,a):
    return a[0]*polPhi(x,2,0)+a[1]*polPhi(x,2,1)+a[2]*polPhi(x,2,2)

def fi3(x,a):
    return␣
 ↪a[0]*polPhi(x,3,0)+a[1]*polPhi(x,3,1)+a[2]*polPhi(x,3,2)+a[3]*polPhi(x,3,3)

def fi4(x,a):
    return␣
 ↪a[0]*polPhi(x,4,0)+a[1]*polPhi(x,4,1)+a[2]*polPhi(x,4,2)+a[3]*polPhi(x,4,3)+a[4]*polPhi(x,4,4

def fi8(x,a):
    return␣
 ↪a[0]*polPhi(x,8,0)+a[1]*polPhi(x,8,1)+a[2]*polPhi(x,8,2)+a[3]*polPhi(x,8,3)+a[4]*polPhi(x,8,4
```

[122]:
```
def derivada2(fun,x,a):
    derivada=derivative(fun,x,dx=0.0001,n=2,args=(a,))
    return derivada
```

[123]:
```
def func1_1(x,a):
    return fi1(x,a)*derivada2(fi1,x,a)

def func1_2(x,a):
    return fi2(x,a)*derivada2(fi2,x,a)

def func1_3(x,a):
    return fi3(x,a)*derivada2(fi3,x,a)
```

```
def func1_4(x,a):
    return fi4(x,a)*derivada2(fi4,x,a)

def func1_8(x,a):
    return fi8(x,a)*derivada2(fi8,x,a)
```

[124]:
```
def func2_1(x,a):
    return fi1(x,a)*fi1(x,a)

def func2_2(x,a):
    return fi2(x,a)*fi2(x,a)

def func2_3(x,a):
    return fi3(x,a)*fi3(x,a)

def func2_4(x,a):
    return fi4(x,a)*fi4(x,a)

def func2_8(x,a):
    return fi8(x,a)*fi8(x,a)
```

[126]:
```
def En1(a):
    num=quad(func1_1,-1,1,args=(a,))
    den=quad(func2_1,-1,1,args=(a,))
    return -num[0]/den[0]

def En2(a):
    num=quad(func1_2,-1,1,args=(a,))
    den=quad(func2_2,-1,1,args=(a,))
    return -num[0]/den[0]

def En3(a):
    num=quad(func1_3,-1,1,args=(a,))
    den=quad(func2_3,-1,1,args=(a,))
    return -num[0]/den[0]

def En4(a):
    num=quad(func1_4,-1,1,args=(a,))
    den=quad(func2_4,-1,1,args=(a,))
    return -num[0]/den[0]

def En8(a):
    num=quad(func1_8,-1,1,args=(a,))
    den=quad(func2_8,-1,1,args=(a,))
    return -num[0]/den[0]
```

```
#Resultado aplicando el principio variacional
res1=minimize(En1,a,method='Powell')
res2=minimize(En2,a,method='Powell')
res3=minimize(En3,a,method='Powell')
res4=minimize(En4,a,method='Powell')
res8=minimize(En8,a,method='Powell')
print(res1.x)
print(En1(res1.x))
print(res2.x)
print(En2(res2.x))
print(res3.x)
print(En3(res3.x))
print(res4.x)
print(En4(res4.x))
print(res8.x)
print(En8(res8.x))
```

```
[0.99999554 0.99995225 3.58792896 3.58792896 3.58792896 3.58792896
 3.58792896 3.58792896 3.58792896]
2.499999995949568
[0.8954391  2.80136615 0.89244062 5.97486439 5.97486439 5.97486439
 5.97486439 5.97486439 5.97486439]
2.4674394929631642
[ 0.83825756  3.46473025  3.4636823   0.83787085 13.68726639 13.68726639
 13.68726639 13.68726639 13.68726639]
2.467437454229422
[0.96899154 4.49749506 8.42395066 4.52488299 0.96922681 8.95545884
 8.95545884 8.95545884 8.95545884]
2.4676659676121173
[  35.497067    324.14984416 1173.85654648 2589.00266272 3316.68506475
 2549.04856123 1186.33623742  326.95423621   34.90243976]
2.467490017027919
```

Arriba podemos observar los valores numéricos obtenidos para cada una de las aproximaciones, a medida que cambiamos N. Es de esperar, que este valor no sea exactamente el teórico. Nos basta con que sea cercano, ya que no hay que olvidar que la motivaciñon de realizar el método variacional es que resolver la ecuación de Schodinger es tarea complicada.

Veamos el error numérico entre el valor teórico y el numérico:

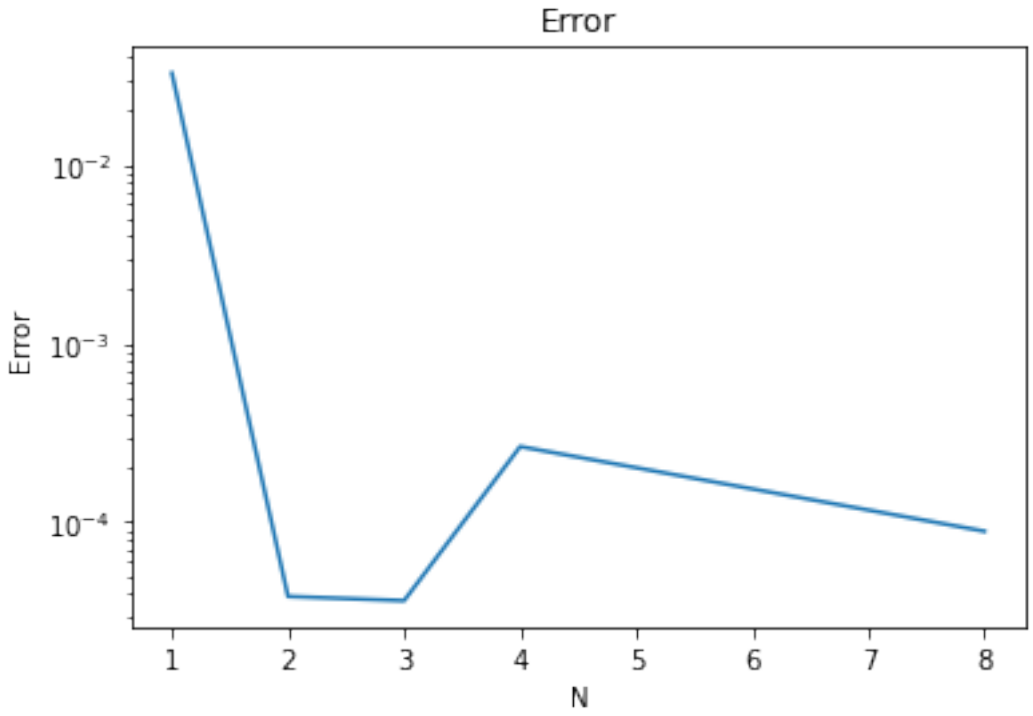
[149]:
```
error = [En1(res1.x) - E_theoric, En2(res2.x) - E_theoric, En3(res3.x) -␣
 ↪E_theoric, En4(res4.x) - E_theoric, En8(res8.x) - E_theoric]
print(error)
```

```
[0.03259889567722851, 3.839269082472896e-05, 3.635395708245781e-05,
0.00026486733977781896, 8.891675557931578e-05]
```

Observamos la diferencia entre el valor teórico y el numérico, observando que esta es pequeña, lo cual es buena señal. Es de esperar también, fluctuaciones a medida que N cambia. No hemos generalizado para muchos valores, por la propia estructura de nuestro código pero vemos que el

error parece disminuir pero justo para N=4 aumenta para luego volver a bajar para N=8. Si esto lo realizáramos para muchos N observariamos de forma más continuada dichas fluctuaciones.

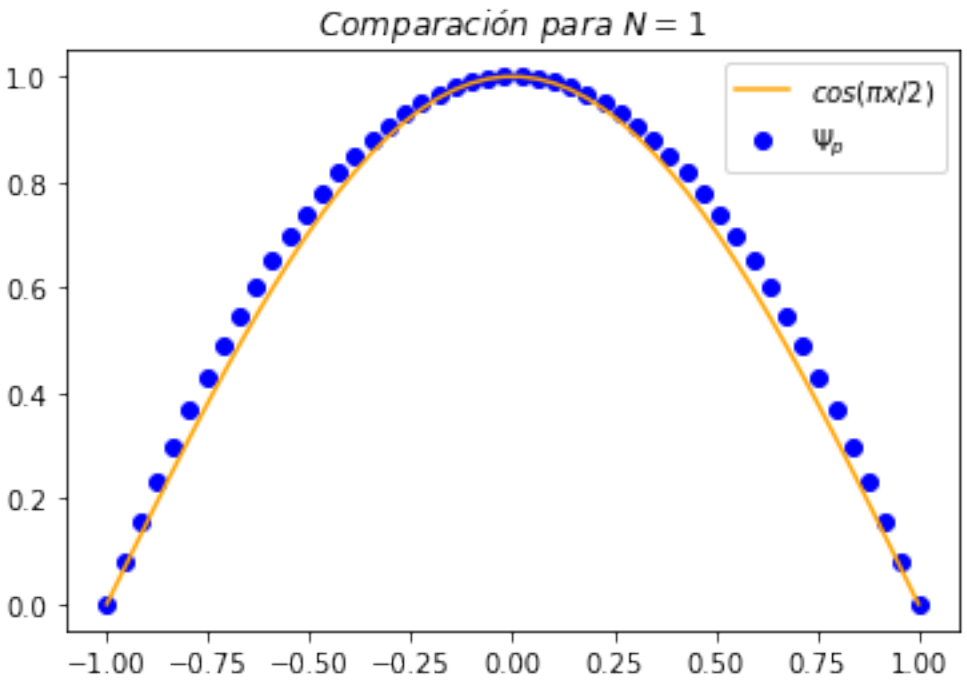
[160]:
```
plt.figure()
plt.plot(np.array([1,2,3,4,8]),error)
plt.yscale('log')
plt.title('Error')
plt.xlabel('N')
plt.ylabel('Error')
plt.show()
```



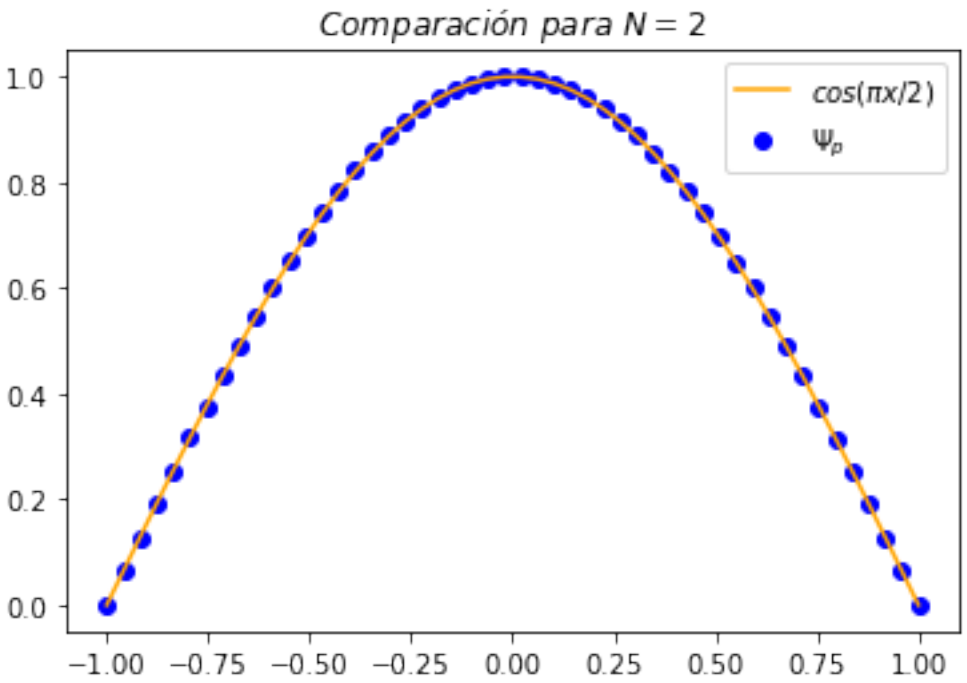
Podemos observar arriba, una gráfica para el error numérico hasta N = 8. Hemos utilizado escala logaritmica para que se marque mejor las fluctuaciones, ya que son pequeñas. Podemos comprobar ahora, la fluctuación para N=4 respecto al resto devalores estudiados. También, que para N = 1 el error es mayor que para ordenes mayores, cosa que tiene sentido.

[156]:
```
x=np.linspace(-1,1)
fi1f=fi1(x,res1.x)/max(fi1(x,res1.x))
fi2f = fi2(x, res2.x)/max(fi2(x, res2.x))

plt.figure()
plt.plot(x,np.cos(np.pi*x/2),color="orange",label="$cos( \pi x /2)$")
plt.scatter(x,fi1f,color="blue",label="$\Psi _p$")
plt.legend(loc="best")
```

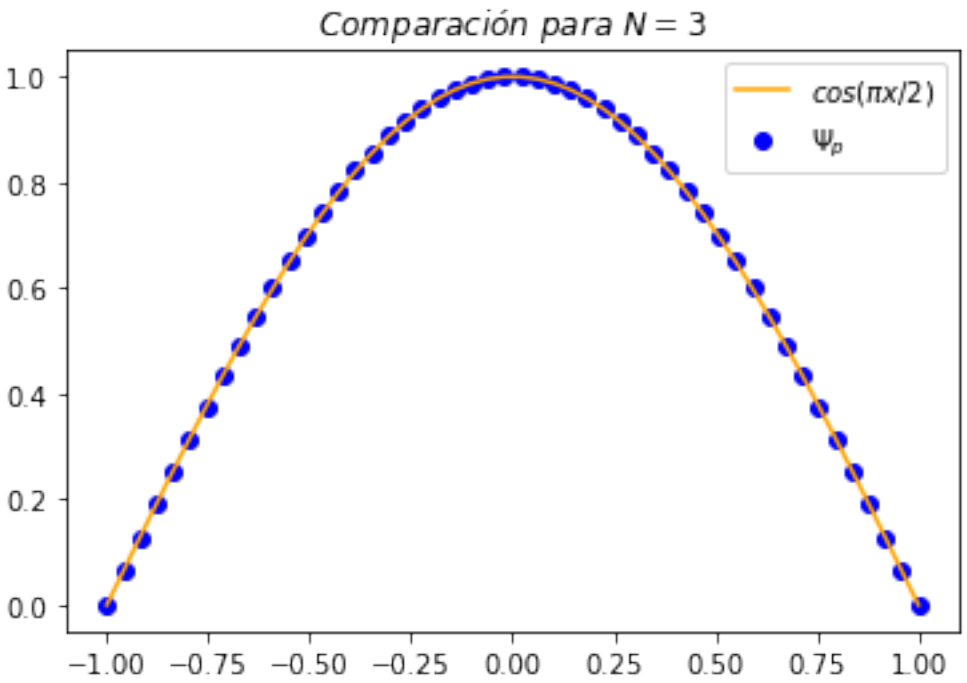
```
plt.title("$Comparación$ $para$ $N=1$")
plt.show()
```



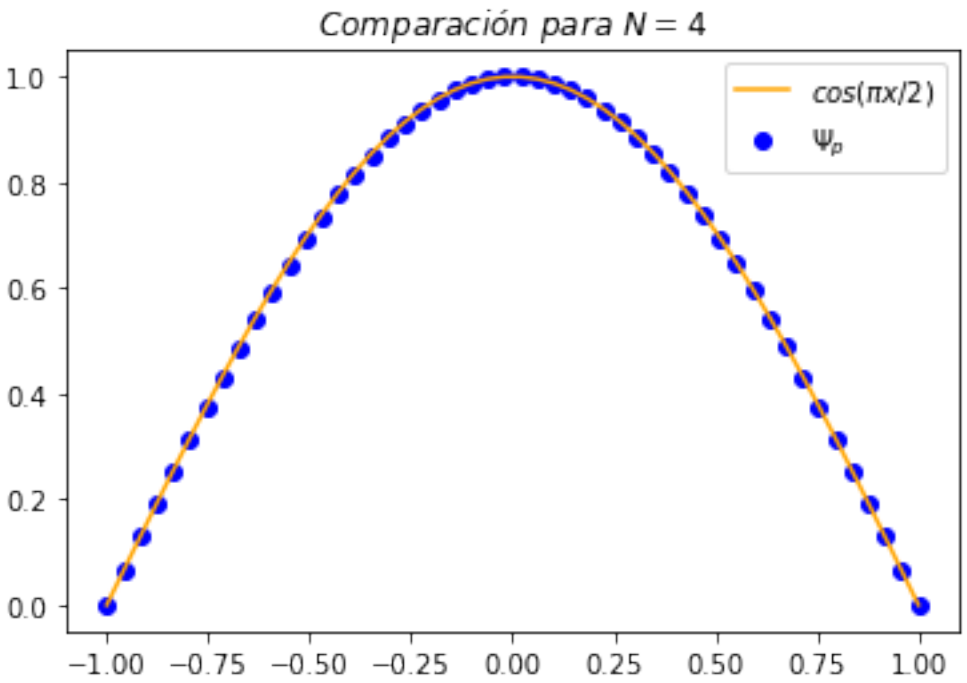
[132]:
```
fi2f=fi2(x,res2.x)/max(fi2(x,res2.x))

plt.figure()
plt.plot(x,np.cos(np.pi*x/2),color="orange",label="$cos( \pi x /2)$")
plt.scatter(x,fi2f,color="blue",label="$\Psi _p$")
plt.legend(loc="best")
plt.title("$Comparación$ $para$ $N=2$")
plt.show()
```



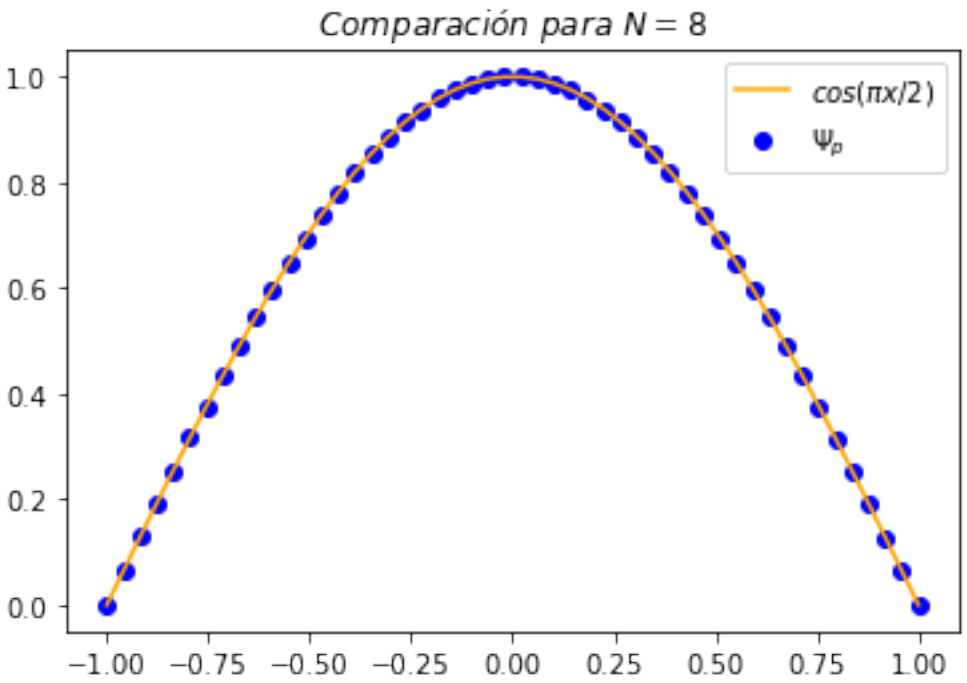
[134]:
```
fi3f=fi3(x,res3.x)/max(fi3(x,res3.x))

plt.figure()
plt.plot(x,np.cos(np.pi*x/2),color="orange",label="$cos( \pi x /2)$")
plt.scatter(x,fi3f,color="blue",label="$\Psi _p$")
plt.legend(loc="best")
plt.title("$Comparación$ $para$ $N=3$")
plt.show()
```



[135]:
```
fi4f=fi4(x,res4.x)/max(fi4(x,res4.x))

plt.figure()
plt.plot(x,np.cos(np.pi*x/2),color="orange",label="$cos( \pi x /2)$")
plt.scatter(x,fi4f,color="blue",label="$\Psi _p$")
plt.legend(loc="best")
plt.title("$Comparación$ $para$ $N=4$")
plt.show()
```



[137]:
```
fi8f=fi8(x,res8.x)/max(fi8(x,res8.x))

plt.figure()
plt.plot(x,np.cos(np.pi*x/2),color="orange",label="$cos( \pi x /2)$")
plt.scatter(x,fi8f,color="blue",label="$\Psi _p$")
plt.legend(loc="best")
plt.title("$Comparación$ $para$ $N=8$")
plt.show()
```



Lo que se observa en las gráficas anteriores es que a medida que N aumenta la solución numérica (puntos azules) se ajustan mejor a la teórica (linea amarilla.)

Mostramos a continuación, el ajuste para el primer excitado, el procedimiento es el mismo que anteriormente pero con diferencia que cambia nuestra función. Definamos las funciones para el primer excitado:

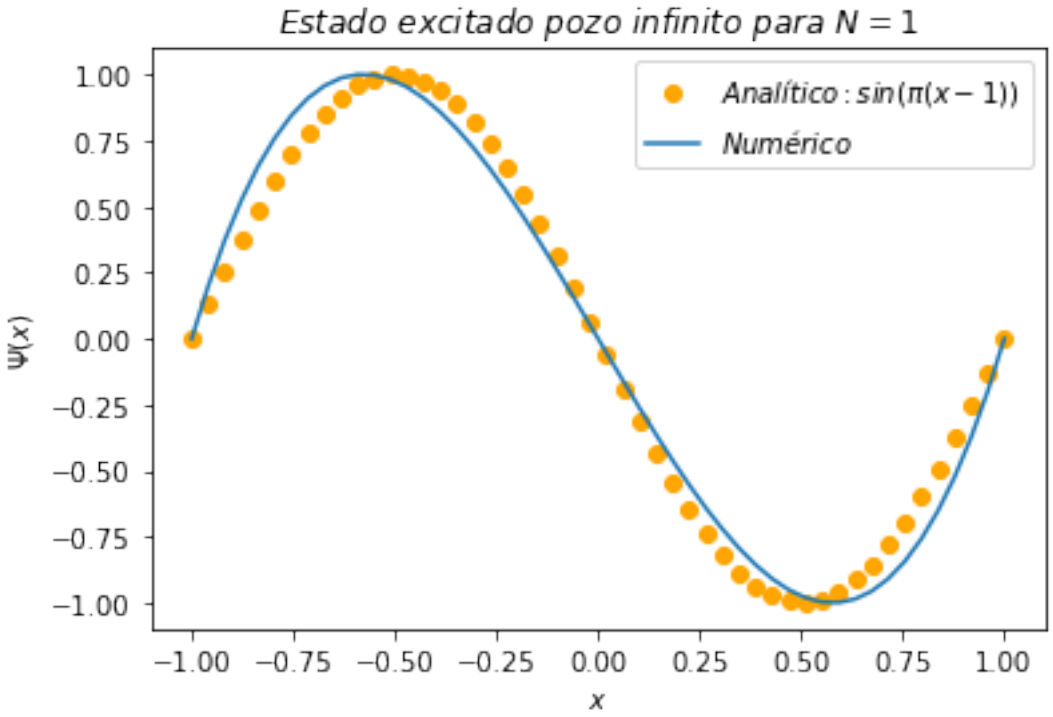
[ ]:
```
def En1(a):
    num=quad(func1_1,-1,1,args=(a,))
    den=quad(func2_1,-1,1,args=(a,))
    return -num[0]/den[0]

def En2(a):
    num=quad(func1_2,-1,1,args=(a,))
    den=quad(func2_2,-1,1,args=(a,))
    return -num[0]/den[0]

def En3(a):
    num=quad(func1_3,-1,1,args=(a,))
    den=quad(func2_3,-1,1,args=(a,))
    return -num[0]/den[0]

def En4(a):
    num=quad(func1_4,-1,1,args=(a,))
    den=quad(func2_4,-1,1,args=(a,))
```

```
    return -num[0]/den[0]

def En8(a):
    num=quad(func1_8,-1,1,args=(a,))
    den=quad(func2_8,-1,1,args=(a,))
    return -num[0]/den[0]

#Resultado aplicando el principio variacional
res1=minimize(En1,a,method='Powell')
res2=minimize(En2,a,method='Powell')
res3=minimize(En3,a,method='Powell')
res4=minimize(En4,a,method='Powell')
res8=minimize(En8,a,method='Powell')
print(res1.x)
print(En1(res1.x))
print(res2.x)
print(En2(res2.x))
print(res3.x)
print(En3(res3.x))
print(res4.x)
print(En4(res4.x))
print(res8.x)
print(En8(res8.x))
```

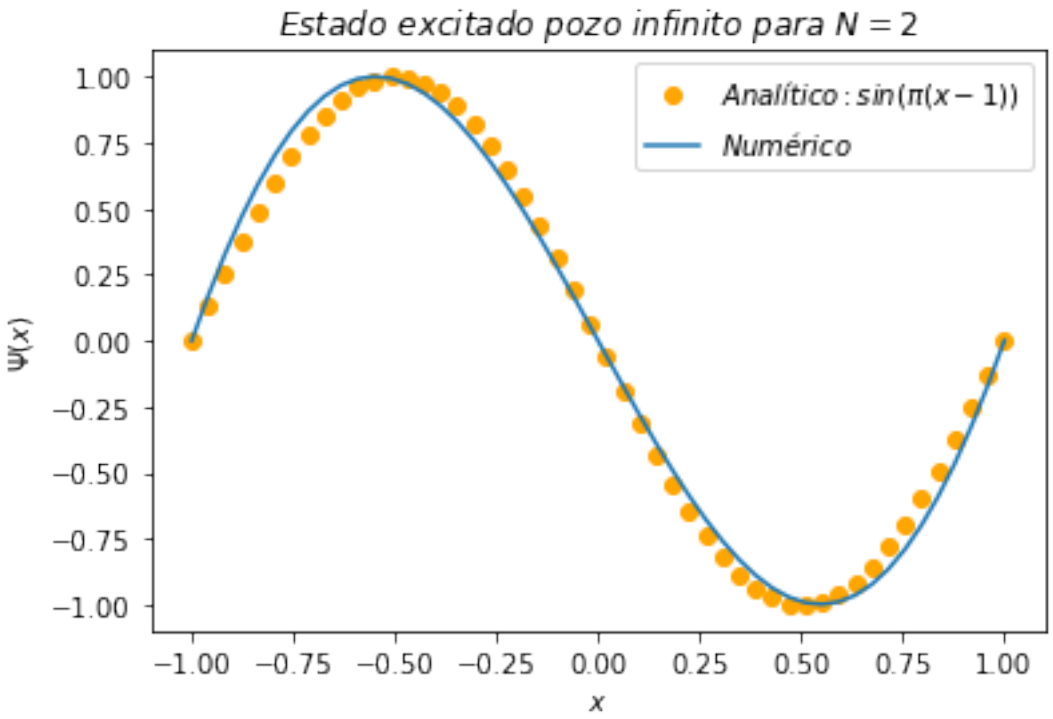
```
[0.99999554 0.99995225 3.58792896 3.58792896 3.58792896 3.58792896
 3.58792896 3.58792896 3.58792896]
2.499999995949568
[0.8954391  2.80136615 0.89244062 5.97486439 5.97486439 5.97486439
 5.97486439 5.97486439 5.97486439]
2.4674394929631642
[ 0.83825756  3.46473025  3.4636823   0.83787085 13.68726639 13.68726639
 13.68726639 13.68726639 13.68726639]
2.467437454229422
[0.96899154 4.49749506 8.42395066 4.52488299 0.96922681 8.95545884
 8.95545884 8.95545884 8.95545884]
2.4676659676121173
[  35.497067    324.14984416 1173.85654648 2589.00266272 3316.68506475
 2549.04856123 1186.33623742  326.95423621   34.90243976]
2.467490017027919
```

[139]:
```
ex1=fi1(x,res1.x)*(-x)
exf1=ex1/max(ex1)

plt.figure()
plt.scatter(x,np.sin(np.pi*(x-1)),color="orange",label="$Analítico:sin( \pi␣
↪(x-1))$")
plt.plot(x,exf1,label="$Numérico$")
```

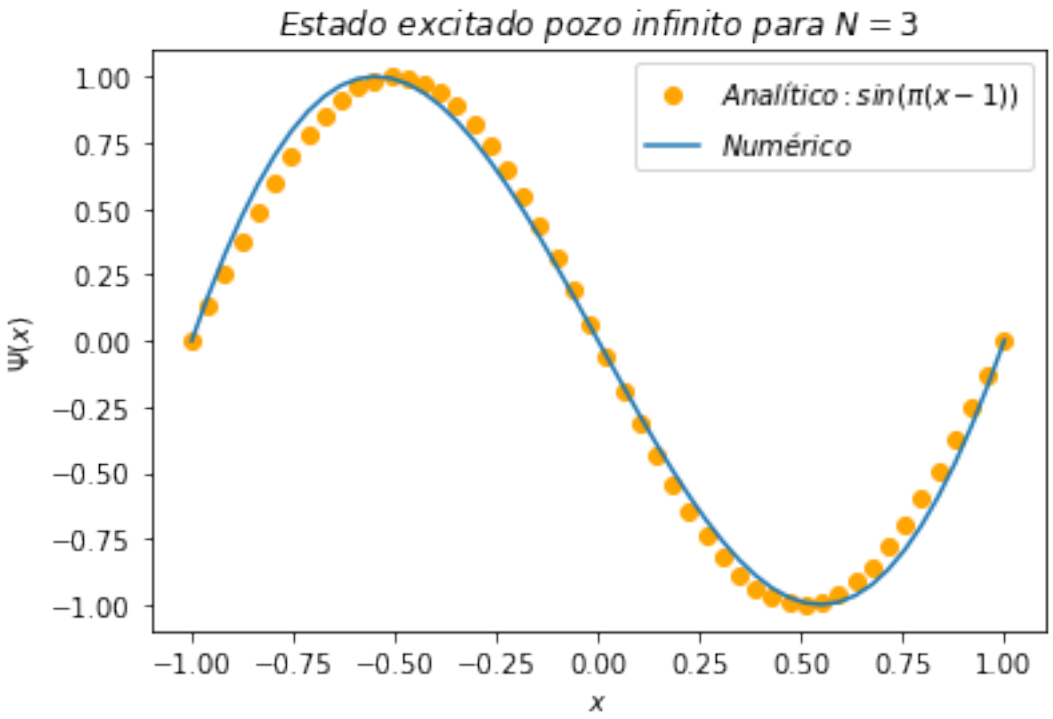
```
plt.xlabel("$x$")
plt.ylabel("$\Psi (x)$")
plt.title("$Estado$ $excitado$ $pozo$ $infinito$ $para$ $N=1$")
plt.legend(loc="best")
plt.show()
```



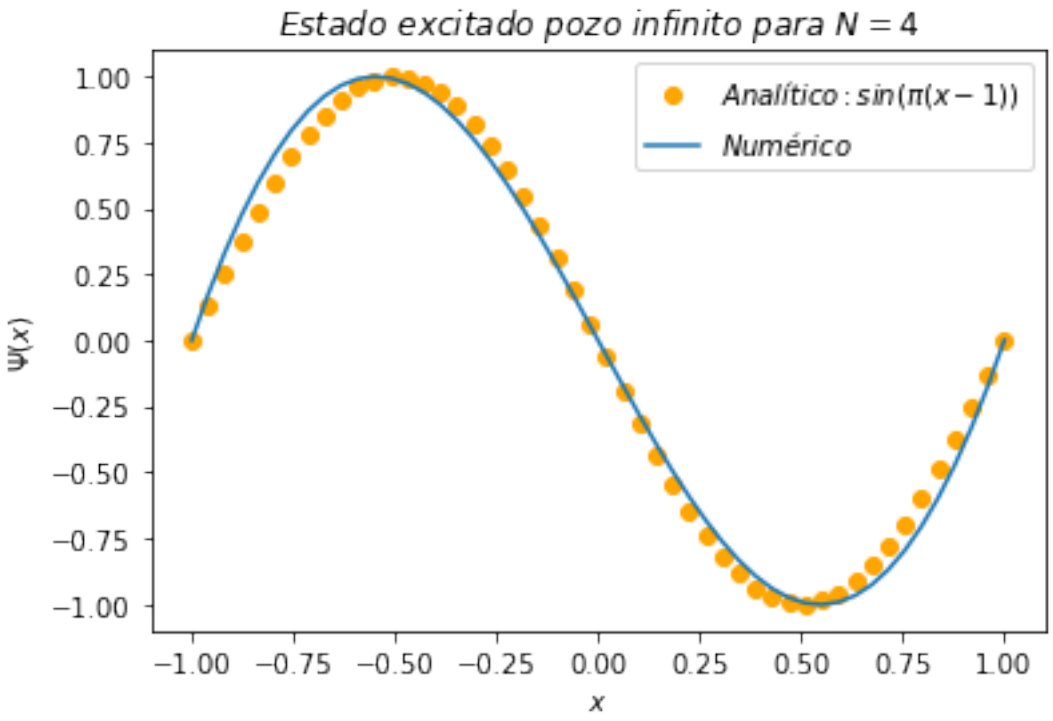
[140]:
```
ex2=fi2(x,res2.x)*(-x)

exf2=ex2/max(ex2)

plt.figure()
plt.scatter(x,np.sin(np.pi*(x-1)),color="orange",label="$Analítico:sin( \pi␣
↪(x-1))$")
plt.plot(x,exf2,label="$Numérico$")
plt.xlabel("$x$")
plt.ylabel("$\Psi (x)$")
plt.title("$Estado$ $excitado$ $pozo$ $infinito$ $para$ $N=2$")
plt.legend(loc="best")
plt.show()
```



[141]:
```
ex3=fi3(x,res3.x)*(-x)

exf3=ex3/max(ex3)

plt.figure()
plt.scatter(x,np.sin(np.pi*(x-1)),color="orange",label="$Analítico:sin( \pi␣
↪(x-1))$")
plt.plot(x,exf3,label="$Numérico$")
plt.xlabel("$x$")
plt.ylabel("$\Psi (x)$")
plt.title("$Estado$ $excitado$ $pozo$ $infinito$ $para$ $N=3$")
plt.legend(loc="best")
plt.show()
```



[142]:
```
ex4=fi4(x,res4.x)*(-x)

exf4=ex4/max(ex4)

plt.figure()
plt.scatter(x,np.sin(np.pi*(x-1)),color="orange",label="$Analítico:sin( \pi␣
↪(x-1))$")
plt.plot(x,exf4,label="$Numérico$")
plt.xlabel("$x$")
plt.ylabel("$\Psi (x)$")
plt.title("$Estado$ $excitado$ $pozo$ $infinito$ $para$ $N=4$")
plt.legend(loc="best")
plt.show()
```



Como hemos visto, el principio variacional es una herramienta extremadamente útil para estimar la energía del estado base de un sistema, como hemos demostrado en esta práctica con un sistema cuyo resultado ya conocíamos. De hecho, las ecuaciones que hemos utilizado son las más comúnmente empleadas en la práctica de la mecánica cuántica. Además, una vez que conocemos la solución del estado base, en muchos casos podremos proponer una función de onda que nos permita determinar el primer estado excitado de energía.