

Solución Numérica por Python de la Relación de Dispersión para un Cristal Unidimensional con el Modelo Kronig Penney

LUIS ALONSO DOMÍNGUEZ VILLANUEVA¹ Y ALEJANDRO GALO ROLDÁN²

¹Escuela de Física - UNAH, mail: villanuevadovi@gmail.com

²Escuela de Física - UNAH, mail: alejandrogalaroldan@gmail.com

Recibido: 14 de Diciembre de 2013 / Aceptado: 30 de Julio de 2016

Resumen

In this paper a numerical method is presented along with scheduling a script to calculate solutions of the dispersion relation Kronig-Penney model as a continuation of the work presented with the Green's functions and the dispersion relation [3]. The basic packages of Python program [6], [5] will be used without using the existing numerical analysis functions in python. At the same time allowed energy bands depending on the wave vector representing the results using a graphical interface is calculated.

Keywords: dispersion relation, Bandas de energía, Numerical method, Python.

En este artículo se presentará un método numérico junto con la programación de un script para calcular soluciones de la relación de dispersión del modelo Kronig-Penney como una continuación del trabajo presentado con las funciones de Green y la relación de dispersión [3]. Se utilizará los paquetes básicos del programa Python [6], [5] sin utilizar las funciones de análisis numérico existentes en python. Al mismo tiempo se calculará las bandas de energía permitidas en función del vector de onda representando los resultados mediante una interfaz gráfica.

Palabras clave: Relación de dispersión, Bandas de energía, Método numérico, Python.

I. INTERFAZ GRÁFICA UTILIZANDO PYTHON

LA deducción de la relación de dispersión [4] utilizando las funciones de Green fue el trabajo principal presentado por Domínguez y Galo [3]. La solución de esta relación será presentada utilizando análisis numérico debido a que es una ecuación trascendental, ver ecuación 1, donde su solución no puede ser representada analíticamente.

$$\cos(Ka) = \cos(Ka) + \frac{P}{Ka} \sin(Ka) \quad (1)$$

Sin embargo se puede obtener una noción de la solución cuando se grafican ambos miembros de la ecuación, como se puede observar en la figura 1.

La gráfica de la figura 1 muestra claramente que existen únicamente *intervalos* de valores permitidos para K , los mostrados de color gris. Los otros valores de K no son permitidos, debido a que no son soluciones de la forma de Bloch [1].

Una vez determinadas las soluciones de la relación de dispersión se puede obtener la gráfica de la energía en función del valor k , ya que $E = (\hbar^2/2m)K^2$ y $K(k)$, es decir, K está en función del vector de onda del espacio recíproco. Estos valores de $E(k)$ son presentados utilizando la interfaz gráfica, ver figura 2

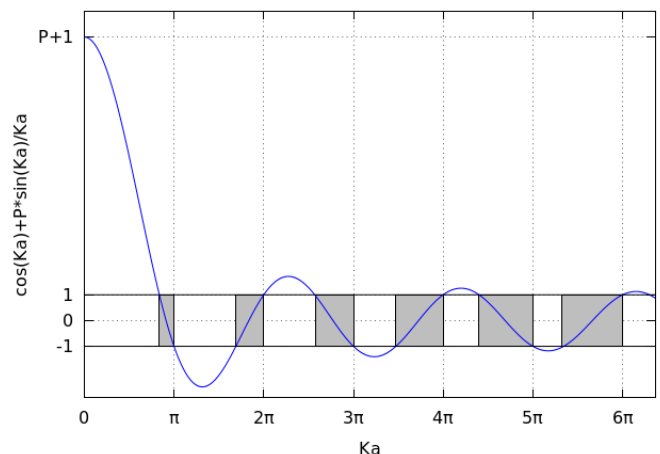


Figura 1: Gráfica de la relación de dispersión

de igual forma se gráfica la tercera banda de energía permitida en la zona reducida, ver figura 3, y en la zona extendida, ver figura 4.

Un caso interesante es cuando se toma $P = 0$, físicamente significa que no existe potencial, por lo tanto el problema tratado en esta forma no es más que el problema del electrón libre. Debido a esto la relación de la energía

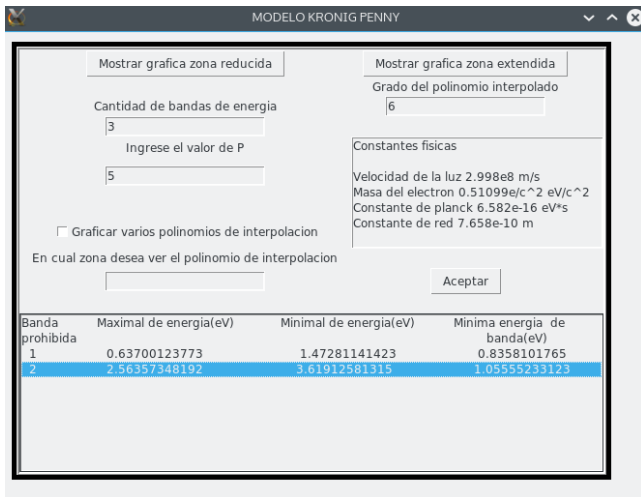


Figura 2: Interfaz gráfica programada para el modelo Kronig Penney

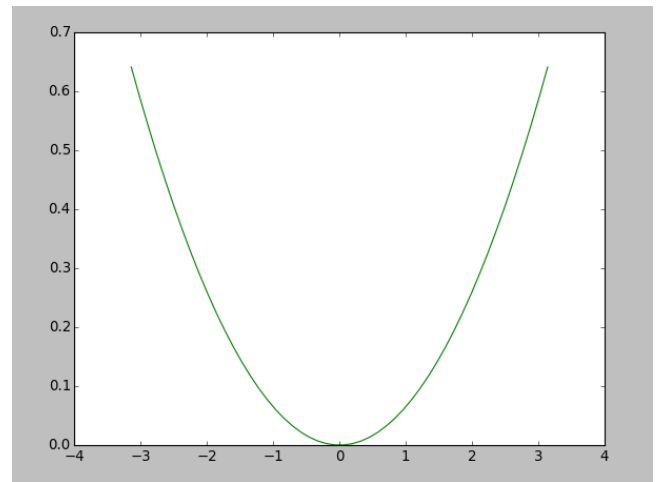


Figura 5: Espectro de energía permitida en la zona extendida para el electrón libre

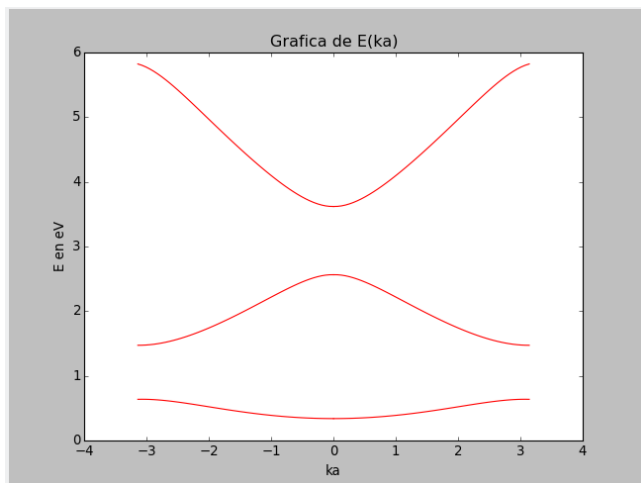


Figura 3: Espectro de energía permitidas en la zona reducida con potencial presente

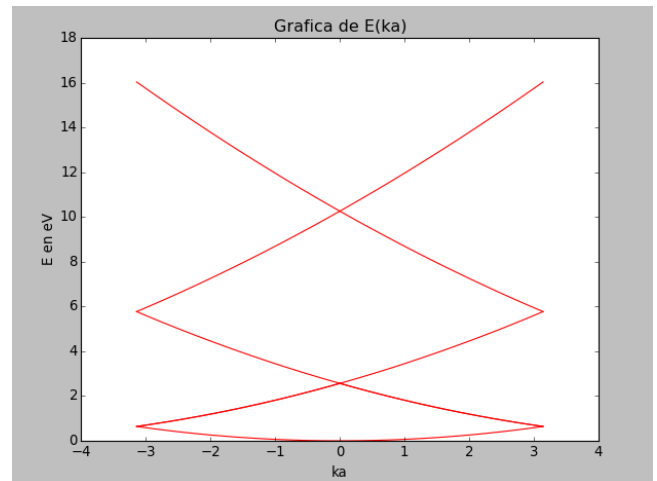


Figura 6: Espectro de energía permitida en la zona reducida para el electrón libre

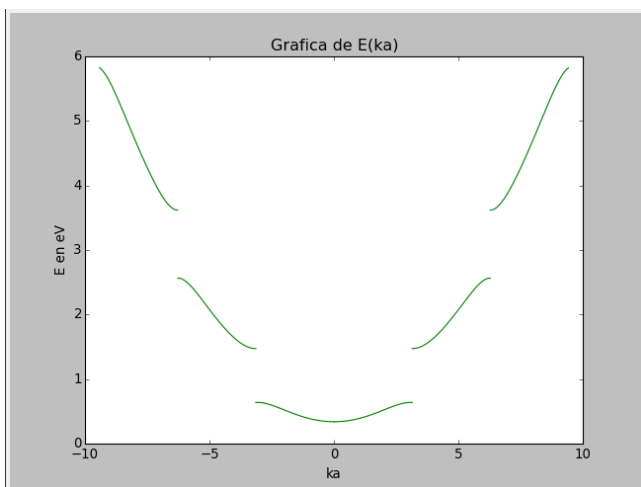


Figura 4: Espectro de energía permitidas en la zona extendida con potencial presente

con el vector de onda está descrita por medio de una parábola tal como lo ilustra la figura 5 y 6.

Al aumentar el valor de P se tiene que las bandas de

energía tiende a líneas rectas horizontales en el esquema de zona reducida, y las bandas de energía prohibidas aumenta. Es decir el modelo hace una representación del enlace fuerte y los electrones presentaría una tendencia a ocupar orbital cercanos a los átomos próximos y al aumentar P hacia un valor infinito la única forma que la relación de dispersión tenga soluciones es que $\sin(Ka)/Ka = 0$ es decir, $Ka = n\pi$ y $E = \pi^2 \hbar^2 n^2 / 2ma^2$ que no es más que la energía permitida del electrón a un único átomo.

II. SCRIPT UTILIZANDO PYTHON PARA REPRESENTAR LAS BANDAS DE ENERGÍA

El script presentado a continuación determina las soluciones de K de la relación de dispersión, para expresar la energía en función del valor de onda k utilizando el método de Newton-Raphson, mínimos cuadrados y la factorización LU [2].

```

1 import os
2 import sys
3 import math
4 from matplotlib.pyplot import *
5 from numpy import *
6 from Tkinter import *
7 from mpl_toolkits.axes_grid1.inset_locator
  import zoomed_inset_axes
8 from mpl_toolkits.axes_grid1.inset_locator
  import mark_inset
9
10
11
12 #-----
13 #Definicion de los valores de las
  constantes fisicas
14 #-----
15
16 _c=2.998e8          # velocidad de
  la luz en m/s
17 _me=0.510999e6/_c**2      # masa del
  electron en unidades de eV/c^2
18 _hb=6.582e-16        # constante de
  planck h/2pi en eV*s
19 _a=7.658e-10         # constante de
  la red de atomos del cristal
20
21
22 #-----
23 zw=math.pi         # valor de la zona de
  Brillown
24 # funcion que proporciona el salto de
  punto flotante
25 #-----
26
27 def frange(x,y,salto):
28     while x < y:
29         yield x
30         x += salto
31
32 #-----
33 # Selecciona los valores decimales segun
  el punto flotante
34 #-----
35
36 def punto_flotante(solu):
37     solu=str(solu)
38     solu=solu[:8]
39     solu=float(solu)
40     return solu
41
42 def punto_flotante1(solu):
43     solu=str(solu)
44     solu=solu[:13]
45     solu=float(solu)
46     return solu
47
48 #-----
49 # Determina la raice utilizando el metodo
  Newton-raphson con una
50 # tolerancia de 10^-10 para el infimo
51 #-----
52
53
54
55 def f_raiceinf(x,p,j):
56     i=0
57     toleran=1
58     y=0
59     while toleran > 10**-10:
60
61         D=p*x*math.cos(x)-(x**2+p)*math.sin(x)
62         if (D!=0 and x > zw*(j-1) and
63             x < zw*j+0.01):
64
65             x=x-(x**2*(math.cos(x)+1)+p*x*math.sin(x))/D
66             toleran=abs(x-y)
67             y=x
68         else:
69             return 0
70         i=i+1
71         if i > 200 or toleran <
72             10**-10:
73             return x
74 #-----
75 # Determina la raice utilizando el metodo
  Newton-raphson con una
76 # tolerancia de 10^-10 para el supremo
77 #-----
78
79 def f_raicesup(x,p,j):
80     i=0
81     toleran=1
82     y=0
83     while toleran > 10**-10:
84
85         D=p*x*math.cos(x)-(x**2+p)*math.sin(x)
86         if (D!=0 and x > zw*(j-1) and
87             x < zw*j+0.01):
88
89             x=x-(x**2*(math.cos(x)-1)+p*x*math.sin(x))/D
90             toleran=abs(x-y)
91             y=x
92         else:
93             return 0
94         i=i+1
95         if i > 200 or toleran <
96             10**-10:
97             return x
98 #-----
99 # Compara y determina raices repetidas
100 #-----
101
102 def depuracion(sol1,sol2):
103     lista=[sol1,sol2]
104     lista.sort()
105     medida=len(sol_dispersion)
106     if medida==0:
107         return lista
108     else:
109         for k in range(0,medida):

```

```

106         if lista ==
107             sol_dispersion[k]:
108                 return
109             return lista
110
111 #-----
112 sol_dispersion=[] #lista de las soluciones
113 #cuerpo principal del programa para
114 #ecuacion de dispersion
115 #-----
116
117 def d_raices(p,n):
118     sol_dispersion=[[0,0] for i in
119         range(n)]
120     for j in range(1,n+1):
121         a=[0,0]
122         for i in
123             frange(zw*(j-1),zw*j,1.5):
124
125             sol1=punto_flotante(f_raiceinf(i,p,j))
126             sol2=punto_flotante(f_raicesup(i,p,j))
127             if int(sol1) != 0 and
128                 int(sol2) != 0:
129                 sol=
130                 depuracion(sol1,sol2)
131                 if sol != None:
132                     sol_dispersion[j-1]=sol
133                     elif int(sol1) !=0 and
134                         int(sol2) == 0:
135                             a[0]=sol1
136                             else:
137                                 a[1]=sol2
138
139                             if a[0] != 0 and a[1]
140                                 != 0:
141
142                                 sol=depuracion(a[0],a[1])
143                                 if sol != None:
144
145                                 sol_dispersion[j-1]=sol
146                                 return sol_dispersion
147
148 #-----
149 #Determina la matriz de Hilber, los
150 #coeficientes de la interpolacion
151 # del polinomio grado n por minimos
152 #cuadrados
153 #-----
154 def hilberm(k1,k2,p,gp):
155     salto=float(abs(k1-k2))/10
156     H=[[0 for i in range(gp)] for i in
157         range(gp)]
158     n=-1
159     for j in range(gp): # gp determina el
160         grado del polinomio
161         n=n+1
162
163         for s in range(gp):# gp
164             determina el grado del polinomio
165             for i in
166                 frange(k1,k2,salto):
167
168                 bb=math.cos(i)+p*math.sin(i)/i
169                 if bb > 1:
170
171                     H[j][s]=(math.acos(1))**(s+n)+H[j][s]
172                     elif bb < -1:
173
174                         H[j][s]=(math.acos(-1))**(s+n)+H[j][s]
175                         else:
176
177                             H[j][s]=(math.acos(bb))**(s+n)+H[j][s]
178                             return H
179
180 #-----
181 #El vector b del sistema de ecuacion Hx=b
182 #para la interpolacion
183 #-----
184 def vectorb(k1,k2,p,gp):
185     salto=abs(k1-k2)/10
186     b=[[0] for i in range(gp)] # gp
187     determina el grado del polinomio
188     for j in range(gp): # gp
189         determina el grado del polinomio
190         for i in frange(k1,k2,salto):
191
192             bb=math.cos(i)+p*math.sin(i)/i
193             if bb > 1:
194
195                 b[j][0]=i*(math.acos(1))**j+b[j][0]
196                 elif bb < -1:
197
198                     b[j][0]=i*(math.acos(-1))**j+b[j][0]
199                     else:
200
201                         b[j][0]=i*(math.acos(bb))**j+b[j][0]
202                         return b
203
204 #-----
205 #Resuelve el sistema Hx=b por el metodo de
206 #factorizacion LU
207 #*****
208 #-----
209 #Factoriza la matriz de Hilbert en LU
210 #-----
211 def matrizLU(aa):
212     n=len(aa)
213     s=0
214     for i in range(n):
215         LU=[[0 for i in range(n)] for
216             i in range(n)]
217         for i in range(n):
218             LU[i]=aa[i][:]
219             for i in range(1,n):

```

```

198 LU[i][0]=float(LU[i][0])/float(LU[0][0])
199     for i in range(1,n):
200         for j in range(i,n):
201             for k in
range(0,i):
202                 s=s+LU[i][k]*LU[k][j]
203                     LU[i][j]=LU[i][j]-s
204                         s=0
205                 for j in range(i+1,n):
206                     for k in
range(0,i):
207                         s=s+float(LU[j][k])*float(LU[k][i])
208
LU[j][i]=(LU[j][i]-s)/float(LU[i][i])
209                         s=0
210     return LU
211
212
213 class matriz:
214
215     def __init__(LU,aa):
216         n=len(aa)
217         for i in range(n):
218             LU.m=[[0] for i in
range(n)]
219                 for i in range(n):
220                     LU.m[i]=aa[i][:]
221     def matrizL(LU):
222         n=len(LU.m)
223         for i in range(n):
224             lu=[[0] for i in range(n)]
225             for i in range(n):
226                 lu[i]=LU.m[i][:]
227                 for i in range(0,n):
228                     for j in
range(i,n):
229                         if i==j:
230
lu[i][j]=1
231
232                         else:
233
lu[i][j]=0
234                 return lu
235     def matrizU(LU):
236         n=len(LU.m)
237         for i in range(n):
238             lu=[[0] for i in range(n)]
239             for i in range(n):
240                 lu[i]=LU.m[i][:]
241                 for i in range(n-1,0,-1):
242                     for j in
frange(i-1,-1,-1):
243                         lu[i][j]=0
244                 return lu
245 #-----
246 #Proporciona la solucion del sistema
utilizando la matriz factorizada
247 #-----
248
249
250 def solu(L,U,b):
251     n=len(L)
252     Y=[[0] for i in range(0,n)]
253     X=[[0] for i in range(0,n)]
254     s=0
255     for i in range(0,n):
256         for j in range(0,i):
257             s=s+L[i][j]*Y[j][0]
258         Y[i][0]=b[i][0]-s
259         s=0
260     for i in range(n-1,-1,-1):
261         for j in range(i+1,n):
262             s=s+U[i][j]*X[j][0]
263         X[i][0]=(Y[i][0]-s)/U[i][i]
264         s=0
265     return X
266
267
268 #-----
269 #Define el polinomio de la interpolacion
utilizando los coeficientes
270 #calculados
271 #-----
272
273 def valorE(x,solcoef):
274     n=len(solcoef)
275     nn=len(x)
276     for i in range(n):
277         coefi=[0 for i in range(n)]
278     for i in range(n):
279         coefi=solcoef[:]
280     y=[0 for i in range(nn)]
281     for i in range(n):
282         y=coefi[i][0]*abs(x)**i+y
283     return y
284
285
286
287
288
289
290 #-----
291 #cuerpo principal del programa para
interpolacion la relacion E (Energia)
292 #vs k (vector disper)
293 #-----
294
295
296
297 def principal1(entradab,entradap,gp):
298     nn=int(entradab)
299     p=float(entradap)
300     ver=d_raices(p,nn)
301     mm=len(ver)
302     matplotlib.pyplot.clf()
303     energia1=[0,0]
304     energia2=[0,0]
305     lista=[0 for i in range(nn-1)]
306     if p == 0:
307         ver[0]=[math.pi,2*math.pi]
308         x=linspace(-math.pi,math.pi,50)
309         y=(_hb**2)/(2*_me*_a**2)*x**2

```

```

310         plot(x,y,'r')
311
312     for i in range(0,nn):
313         k1=ver[i][0]
314         k2=ver[i][1]
315         HH=hilberm(k1,k2,p,gp)
316         bb=vectorb(k1,k2,p,gp)
317         M=matriz(matrizLU(HH))
318         U=M.matrizU()
319         L=M.matrizL()
320         solcoeficientes=solu(L,U,bb)
321         x=linspace(0,math.pi,50)
322         y=valorE(x,solcoeficientes)
323         y=(_hb**2)/(2*_me*_a**2)*y**2
324         if p!=0:
325             a=(2+i)%2
326         if a==0:
327
328     energia1[a]=punto_flotante1(y[49])
329
330     energia2[a]=punto_flotante1(y[0])
331     else:
332
333     energia2[a]=punto_flotante1(y[49])
334
335     energia1[a]=punto_flotante1(y[0])
336
337     if p!=0 and i!=0 and a!=0:
338
339     resta=energia2[1]-energia1[0]
340     lista[i-1]=" "+str(i)+"
341     "+str(energia1[0])+"
342     "+str(energia2[1])+"
343     "+str(abs(resta))
344     if p!=0 and i!=0 and a==0:
345
346     resta=energia2[0]-energia1[1]
347     lista[i-1]=" "+str(i)+"
348     "+str(energia1[1])+"
349     "+str(energia2[0])+"
350     "+str(abs(resta))
351
352     plot(x,y,'r')
353     x=linspace(-math.pi,0,50)
354     y=valorE(x,solcoeficientes)
355     y=(_hb**2)/(2*_me*_a**2)*y**2
356     plot(x,y,'r')
357     xlabel("ka")
358     ylabel("E en eV")
359     title("Grafica de E(ka)")
360     return lista
361
362 def principal2(entradab,entradap,gp):
363     nn=int(entradab)
364     p=float(entradap)
365     ver=d_raices(p,nn)
366     mm=len(ver)
367     matplotlib.pyplot.clf()
368     energia1=[0,0]
369     energia2=[0,0]
370     lista=[0 for i in range(nn-1)]
371     if p == 0:
372
373         ver[0]=[math.pi,2*math.pi]
374
375         x=linspace(-math.pi,math.pi,50)
376
377         y=(_hb**2)/(2*_me*_a**2)*x**2
378         plot(x,y,'g')
379     else:
380         for i in range(0,nn):
381             k1=ver[i][0]
382             k2=ver[i][1]
383             HH=hilberm(k1,k2,p,gp)
384             bb=vectorb(k1,k2,p,gp)
385             M=matriz(matrizLU(HH))
386             U=M.matrizU()
387             L=M.matrizL()
388
389             solcoeficientes=solu(L,U,bb)
390
391             x=linspace(0,math.pi,50)
392
393             y=valorE(x,solcoeficientes)
394
395             y=(_hb**2)/(2*_me*_a**2)*y**2
396
397             if p!=0:
398                 a=(2+i)%2
399             if a==0:
400
401                 energia1[a]=punto_flotante1(y[49])
402
403                 energia2[a]=punto_flotante1(y[0])
404             else:
405
406                 energia2[a]=punto_flotante1(y[49])
407
408                 energia1[a]=punto_flotante1(y[0])
409
410                 if p!=0 and i!=0 and
411                 a!=0:
412
413                     resta=energia2[1]-energia1[0]
414
415                     lista[i-1]=" "+str(i)+"
416                     "+str(energia1[0])+"
417                     "+str(energia2[1])+"
418                     "+str(abs(resta))
419                     if p!=0 and i!=0 and
420                     a==0:
421
422                         resta=energia2[0]-energia1[1]
423
424                         lista[i-1]=" "+str(i)+"
425                         "+str(energia1[1])+"
426                         "+str(energia2[0])+"
427                         "+str(abs(resta))
428
429                     if (2+i)%2==0:
430
431                         k=linspace(i*math.pi,(i+1)*math.pi)
432                         plot(k,y,'g')
433             else:

```

```

400 k=linspace(-(i+1)*math.pi,-i*math.pi) 449
401 plot(k,y,'g') 450
402 451 nn=int(entradab)
403 x=linspace(-math.pi,0,50) 452 p=float(entradap)
404 y=valorE(x,solcoeficientes) 453 num=int(num)
405 y=(_hb**2)/(2*_me*_a**2)*y**2 454 ver=d_raices(p,nn)
406 if (2+i)%2 ==0: 455 matplotlib.pyplot.clf()
407 k=linspace(-(i+1)*math.pi,-i*math.pi) 456 if p==0:
408 plot(k,y,'g') 457 print("NO existe bandas
409 else: 458 permitidas o prohibidas, por lo tanto
410 k=linspace(i*math.pi,(i+1)*math.pi) 459 no existe bandas que interpolar")
411 plot(k,y,'g') 460 else:
412 xlabel("ka") 461 if varl==0:
413 ylabel("E en eV") 462 k1=ver[num][0]
414 title("Grafica de 463 k2=ver[num][1]
415 E(ka)") 464 print(k1,k2)
416 return lista 465 HH=hilberm1(k1,k2,p,gp)
417 466 bb=vectorb1(k1,k2,p,gp)
418 def 467 M=matriz(matrizLU(HH))
419 graficapolino(entradab,entradap,num,varl, 468 U=M.matrizU()
420 gp): 469 L=M.matrizL()
421 def hilberm1(k1,k2,p,gp): 470 solcoeficientes=solu(L,U,bb)
422 salto=float(abs(k1-k2))/10 471 x=linspace(-p/6-p/3,p/3+p/3,50)
423 H=[[0 for i in range(gp)] for i in 472 y=valorE1(x,solcoeficientes)
424 range(gp)] 473 print(x[0],x[49])
425 n=-1 474 fig=subplot(1,1,1)
426 for j in range(gp): 475 x1=linspace(0.00001,k2+3*math.pi,k2*50)
427 n=n+1 476 xlim(0.0000001,k2+2*pi)
428 for s in range(gp): 477 fig.plot(x1,cos(x1)+p*sin(x1)/x1,y,x)
429 for i in 478 figzoom=zoomed_inset_axes(fig,1.8+p*0.01,
430 frange(k1,k2,salto): 479 loc=2,
431 bb=math.cos(i)+p*math.sin(i)/i 480 bbox_to_anchor=(0.7,0.9),bbox_transform=(
432 H[j][s]=(bb)**(s+n)+H[j][s] 481 fig.figure.transFigure))
433 return H 482 #zomm =1.8+algo
434 def vectorb1(k1,k2,p,gp): 483 figzoom.plot(y,x)
435 salto=abs(k1-k2)/10 484 x1,x2=k1,k2
436 b=[[0] for i in range(gp)] 485 y1,y2=-p/4,p/4
437 for j in range(gp): 486 figzoom.set_xlim(x1,x2)
438 for i in frange(k1,k2,salto): 487 figzoom.set_ylim(y1,y2)
439 bb=math.cos(i)+p*math.sin(i)/i 488 mark_inset(fig,figzoom,
440 b[j][0]=i*(bb)**j+b[j][0] 489 loc1=1,loc2=3,fc="none",ec="0.5")
441 return b 490 xticks(visible=False)
442 def valorE1(x,solcoef): 491 yticks(visible=True)
443 n=len(solcoef) 492 figzoom.plot(x,y)
444 nn=len(x) 493 print(k1,k2)
445 for i in range(n): 494 show()
446 coefi=[0 for i in range(n)] 495 else:
447 for i in range(n): 496 for i in range(num+1):
448 y=coefi[i][0]*x**i+y 497 k1=ver[i][0]
498 k2=ver[i][1]
499 print(k1,k2)
500 HH=hilberm1(k1,k2,p,gp)
501 bb=vectorb1(k1,k2,p,gp)
502 M=matriz(matrizLU(HH))
503 U=M.matrizU()
504 L=M.matrizL()
505 solcoeficientes=solu(L,U,bb)

```

```

501 x=linspace(-p/6-p/3,p/3+p/3,50)
502
503 y=valorE1(x,solcoeficientes)
504     plot(y,x,'b')
505
506 x=linspace(0.000001,k2+3*math.pi,k2*50)
507     xlim(0.000001,k2+2*pi)
508
509 plot(x,cos(x)+p*sin(x)/x,'r')
510     show()

```

Código 1: *Análisis numérico usando Python*

El siguiente script contiene las líneas de comando que entrelazan el programa de análisis numérico presentado anteriormente con la interfaz gráfica.

```

1
2
3 #-----
4 # Interfaz grafica.
5 #-----
6
7
8 from Tkinter import *
9 from cuerpo import *
10 import sys
11
12
13 def prueb1():
14     try:
15         var1=float(entradab.get())
16         var2=float(entradap.get())
17         var3=float(entradagp.get())
18         if (var1-int(var1))>0.00001 or
19             int(var1)< 0:
20             etiqueta3.config(text="Las
21                 bandas a graficar tiene que ser un
22                 entero positivo")
23             elif int(float(var2))< 0:
24                 etiqueta3.config(text="El
25                     valor de P tiene que ser un numero
26                     positivo")
27             elif (var3-int(var3))>0.00001 or
28                 int(float(var3))< 0:
29                 etiqueta3.config(text="El
30                     grado del polinomio tiene que ser
31                     entero positivo")
32             else:
33                 etiqueta3.config(text="")
34
35         a=principal1(entradab.get(),entradap.get(),
36             int(entradagp.get()+1))
37         lista2.delete(2,END)
38         for elem in a:
39             lista2.insert(END,elem)
40
41         etiqueta3.config(text="")
42         show()
43     except:
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62 def grafpoli():
63     try:
64         etiqueta5.config(text="")
65         graficapolino(entradab.get(),(
66             entradap.get(),
67
68             int(entradanum.get()-1,var.get()),
69             int(entradagp.get()+1))
70         except:
71             etiqueta5.config(text="El valor
72                 tiene que ser un numero entero entre:
73                 1 y " +
74                 str(int(float(entradab.get()))))

```



```

72
73
74 root=Tk()
75 root.title("MODELO KRONIG PENNEY")
76 root.geometry("660x540")
77 vp=Frame(root,bd=3,
    relief="ridge",highlightbackground="green",
    highlightcolor="black",
    highlightthickness=5)
78 vp.grid(column=0, row=0, padx=(5,5),
    pady=(10,10))
79 vp.columnconfigure(0, weight=1)
80 vp.rowconfigure(0, weight=1)
81
82 v1=Toplevel(vp)
83 v1.geometry("400x400")
84 v1.title("Grafica de E(ka) en la zona
    reducida")
85 v1.protocol("WM_DELETE_WINDOW","onexit")
86 v1.resizable(0,0)
87 v1.withdraw()
88
89 v2=Toplevel(vp)
90 v2.geometry("400x400")
91 v2.title("grafica de E(ka) en la zona
    extendida")
92 v2.protocol("WM_DELETE_WINDOW","onexit")
93 v2.withdraw()
94 v2.resizable(0,0)
95
96 valorp=""
97 valorb=""
98 entradap=Entry(vp, textvariable=valorp)
99 entradap.grid(column=1, row=8)
100 entradab=Entry(vp, textvariable=valorb)
101 entradab.grid(column=1, row=5)
102
103
104
105 lista1=Listbox(vp,width=30, height=7)
106 lista1.insert(END,"Constantes fisicas")
107 lista1.insert(END,"")
108 lista1.insert(END,"Velocidad de la luz
    2.998e8 m/s")
109 lista1.insert(END,"Masa del electron
    0.51099e/c^2 eV/c^2")
110 lista1.insert(END,"Constante de planck
    6.582e-16 eV*s")
111
112
113 lista1=Listbox(vp,width=30, height=7)
114 lista1.insert(END,"Constantes fisicas")
115 lista1.insert(END,"")
116 lista1.insert(END,"Velocidad de la luz
    2.998e8 m/s")
117 lista1.insert(END,"Masa del electron
    0.51099e/c^2 eV/c^2")
118 lista1.insert(END,"Constante de planck
    6.582e-16 eV*s")
119 lista1.insert(END,"Constante de red
    7.658e-10 m")
120 lista1.grid(column=2, columnspan=3, row=6,
    rowspan=8, sticky=EW)
121
122
123 lista2=Listbox(vp, width=75 )
124 lista2.insert(END,"Banda "
    "Maximal de energia(eV)
    Minimal de energia(eV) Minima
    energia de")
125 lista2.insert(END,"prohibida " "
    banda(eV)")
126 lista2.grid(column=1,columnspan=3,row=19)
127
128
129
130
131 BS1=Button(vp, text="Mostrar grafica zona
    reducida",
    command=prueb1).grid(column=1, row=1)
132 BS2=Button(vp, text="Mostrar grafica zona
    extendida",
    command=prueb2).grid(column=2, row=1)
133 etiqueta1=Label(vp, text="Ingrese el valor
    de P").grid(column=1, row=6)
134 etiqueta2=Label(vp, text="Cantidad de
    bandas de energia").grid(column=1,
    row=4)
135 etiqueta3=Label(vp, text="")
136 etiqueta3.grid(column=1, row=9,
    sticky=(W,E))
137 etiqueta5=Label(vp, text="")
138 etiqueta5.grid(column=1, row=17,
    sticky=(W,E))
139
140 valorgp=""
141 etiqueta4=Label(vp, text="Grado del
    polinomio interpolado").grid(column=2,
    row=3)
142 entradagp=Entry(vp, textvariable=valorgp)
143 entradagp.grid(column=2, row=4)
144
145 valornum=""
146 var=IntVar()

```

Código 2: *Interfaz gráfica para el modelo Kronig Penney*

III. CONCLUSIÓN

Se demostró que la relación de dispersión para el modelo Kronig-Penney en una dimensión tiene su representación gráfica utilizando análisis numérico para cualquier valor diferente de P y además si $P = 0$ no es más que la relación existente para el electrón libre y cuando P se tiende al infinito el modelo tiene soluciones si el $\sin(Ka) = 0$ por lo tanto $Ka = n\pi$ lo cual es una representación al modelo de enlace fuerte, es decir el electrón estaría ligado únicamente a los átomos próximos, que en una dimensión representaría el electrón confinado en una barrera de potenciales infinitos.

REFERENCIAS

- [1] Ashcroft, N. W. y Mermin, D. N. (1976). Solid state physics. *Thomson Learning, Toronto, 1*.
- [2] Burden, R. y Faires, J. (2002). Análisis numérico. *Math Learning Mexico, 7nd*.
- [3] Domínguez, L. y Galo, A. (2016). Determinación de la relación de dispersión de un cristal unidimensional con el modelo de kronig penney utilizando funciones de green. *Revista de Física, UNAH*, 4(1), 5.
- [4] Kronig, R. y Penney, W. (1931). Quantum mechanics of electrons in crystal lattices. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 130(814), 499–513, doi:10.1098/rspa.1931.0019. URL <http://dx.doi.org/10.1098/rspa.1931.0019>.
- [5] Ramirez, A. O. (2010). Python como primer lenguaje de programación.
- [6] Severance, C. (2015). Python para informáticos: Explorando la información. *Sue Blumenberg, 2.7.2*.