

Oscilador armónico cuántico unidimensional en baño térmico usando el algoritmo Metrópolis.

Juan Esteban Aristizabal Zuluaga
Instituto de Física, Universidad de Antioquia.
(Dated: 29 de abril de 2020)

En este artículo presentamos un estudio del oscilador armónico cuántico unidimensional en un baño térmico. En particular, nos interesamos por calcular la probabilidad de encontrar el sistema en una posición dada. Para esto, presentamos los cálculos teóricos cuánticos y su contraparte clásica, con el fin de comparar los resultados. Especialmente nos enfocamos en usar el algoritmo Metrópolis para reconstruir histogramas que representan las distribuciones de probabilidad cuánticas, tanto en el espacio de posiciones como en los niveles de energía. Estos resultados cuánticos los usamos para contrastarlos con los clásicos y los tres casos presentados –baja, media y alta temperatura– concuerdan claramente con los cálculos teóricos. Se presenta también la implementación del algoritmo Metrópolis en el lenguaje de programación «Python».

Palabras clave: Oscilador armónico, física estadística cuántica, baño térmico, ensamble canónico, algoritmo Montecarlo.

I. INTRODUCCIÓN

El oscilador armónico ha sido históricamente para la física un sistema simple pero del que se puede extraer gran cantidad de información y con el que se han descubierto muchos nuevos métodos y hasta teorías completas, basadas en los razonamientos y el conocimiento obtenido de éste. Por citar un ejemplo, está la cuantización del campo electromagnético que se puede reducir a un sistema «osciladores armónicos» no acoplados y en general las teorías de segunda cuantización en la base número usan gran parte del formalismo del oscilador armónico cuántico, aunque con un significado muy diferente al que se le da en el sistema que nos compete[1, 2].

En nuestro caso hemos tomado el oscilador armónico unidimensional inmerso en un baño térmico y hemos estudiado su comportamiento a diferentes temperaturas y contrastado los resultados cuántico y clásico para la probabilidad de encontrarlo en una posición dada. Para ello hemos revisado los resultados teóricos. Entre diferentes alternativas presentadas en la literatura para llegar al resultado cuántico, entre ellas el formalismo de integrales de camino [3, 4], propagadores [5] y métodos más heurísticos como el de Feynman [6], hemos decidido presentar el formalismo desarrollado por Cohen-Tannoudji [7], el cual deriva una ecuación diferencial parcial para encontrar los elementos de matriz diagonales del operador densidad, los cuales corresponden con la probabilidad en la que estamos interesados. El método que presentamos tiene la ventaja de que requiere de cálculos básicos y no de métodos avanzados, que pueden ser un poco más confusos.

Por otro lado, como sabemos, en la física estadística las herramientas computacionales han permitido un mejor entendimiento de diversos problemas. En particular, el algoritmo Metrópolis ha sido ampliamente usado desde que Metropolis *et al.* publicaron el artículo que lo propone [8], en el año 1953, que posteriormente ganó más popularidad con la generalización hecha en 1970 por Hastings [9]. El algoritmo es útil especialmente en problemas

de alta dimensionalidad para muestrear distribuciones de probabilidad en el que otros métodos no son igual de eficientes o simplemente no funcionan –uno de los ejemplos más comunes es la implementación de este algoritmo en sistemas tipo Ising [10]–, aunque en teoría se puede usar para sistemas con cualquier dimensionalidad.

En nuestro caso, a pesar de tener un sistema de baja dimensionalidad, usamos el algoritmo metrópolis para obtener los histogramas de las densidades de probabilidad para el caso cuántico tanto para $T = 0$ como varios valores de $T \neq 0$. Por otro lado, usando el mismo algoritmo, encontramos histogramas para los niveles de energía en cada caso y comprobamos que corresponden con la distribución de Boltzmann *i.e.* la distribución de probabilidad dada por el ensamble canónico de la física estadística.

La estructura del artículo es la siguiente: en la sección II presentamos los resultados teóricos para la densidad de probabilidad cuántica y clásica de encontrar el oscilador armónico unidimensional en una posición dada cuando éste está inmerso en un baño térmico. En la parte III contrastamos los resultados teóricos clásico y cuántico con simulaciones usando el algoritmo Montecarlo para la parte cuántica, para diferentes valores de temperatura. En esta sección también comprobamos los resultados y los límites de alta y baja temperatura que obtuvimos en II. En IV presentamos la conclusión del trabajo y, finalmente, en los apéndices A y B escribimos las implementaciones de los algoritmos de metrópolis usados (en Python3) para generar las figuras y para los análisis de la sección III.

II. CONSIDERACIONES TEÓRICAS

Los algoritmos *Matrix Squaring* y *Path Integral Naive Sampling* están basados en un hecho muy sencillo. Nos aprovechamos de que la matriz densidad en el ensamble

canónico se escribe como

$$\hat{\rho}(\beta) = e^{-\beta \hat{H}}. \quad (1)$$

De este modo es muy sencillo ver que

$$\begin{aligned} \hat{\rho}(\beta) &= e^{-\beta \hat{H}/N} \cdot e^{-\beta \hat{H}/N} \dots e^{-\beta \hat{H}/N} \rightarrow N \text{ veces en total} \\ &= \hat{\rho}(\beta/N) \cdot \hat{\rho}(\beta/N) \dots \hat{\rho}(\beta/N) \end{aligned} \quad (2)$$

El significado físico de la ecuación anterior físico es que podemos construir matrices densidad a temperatura $1/\beta$ usando matrices densidad a temperaturas más altas N/β .

Por otro lado, en estos algoritmos también usamos la conocida aproximación de Trotter para la matriz densidad a altas temperaturas **CITAAAAAAAAA**

$$\rho(x, x', \beta) \xrightarrow{\beta \rightarrow 0} e^{-\frac{1}{2}\beta V(x)} \rho^{\text{free}}(x, x', \beta) e^{-\frac{1}{2}\beta V(x')}. \quad (3)$$

Con esto en mente, veamos cómo se construyen los algoritmos mencionados.

Como comentario final de esta sección, es importante también mencionar que los algoritmos se ejecutaron en Python3 v3.6.

IV. CONCLUSIÓN

En este trabajo estudiamos el problema del oscilador armónico en un baño térmico, tanto de forma clásica como cuántica y con un tratamiento teórico y computacional –éste último en el marco del algoritmo Metrópolis.

Pudimos calcular para el oscilador armónico cuántico en un baño térmico los elementos diagonales del operador densidad en la base de posiciones, $\rho(x, x; \beta)$. Estos elementos diagonales los interpretamos como la densidad de probabilidad de encontrar a la «partícula» en la posición x : $\pi^{(Q)}(x; \beta)$. En el caso clásico calculamos esta probabilidad con ayuda de la función de distribución en el espacio de fase definida por el ensamble canónico. Encontramos que el límite de baja temperatura para el caso clásico es una delta de Dirac centrada en el origen, mientras que en el caso cuántico este límite corresponde con la densidad de probabilidad de la autofunción de energía del estado base del oscilador armónico, conforme se espera. De igual modo pudimos notar que en el límite de altas temperaturas la densidad de probabilidad cuántica mencionada tiende a la clásica, conforme se espera también desde la física estadística.

A. Matrix Squaring

Para entender la idea de este algoritmo, primero consideramos la ecuación (2) con $N = 2$ para construir, a partir de dos matrices densidad a temperatura $2/\beta$, una matriz densidad a temperatura $1/\beta$.

$$\hat{\rho}(\beta) = \hat{\rho}(\beta/2) \cdot \hat{\rho}(\beta/2). \quad (4)$$

En el algoritmo lo que hacemos es tomar
En el espacio de posiciones tenemos

$$\rho(x, x'; \beta) = \int dx_1 \rho(x, x_1; \frac{\beta}{2}) \rho(x_1, x'; \frac{\beta}{2}). \quad (5)$$

B. Path Integral Naive Sampling

III. RESULTADOS Y DISCUSIÓN

A. Matrix Squaring

B. Path Integral Naive Sampling

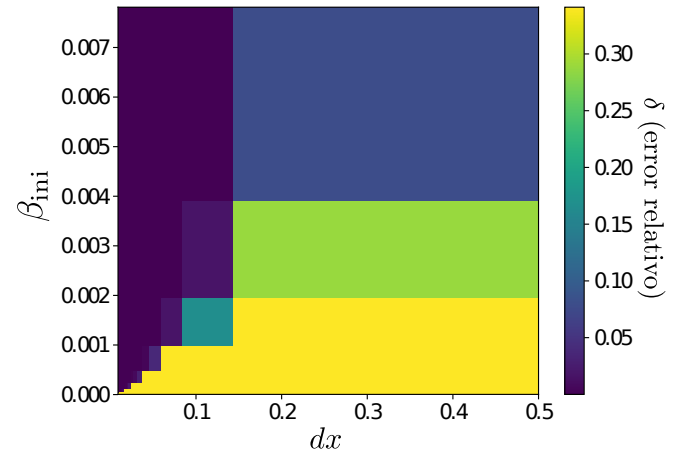


Figura 1. **caption**

Para contrastar los resultados teóricos usamos el algoritmo Metrópolis para reconstruir los histogramas del sistema cuántico en el espacio de las posiciones y en los niveles de energía. Para los casos de β evaluados encontramos que uno corresponde a un límite de alta temperatura ya que las distribuciones cuántica y clásica eran muy

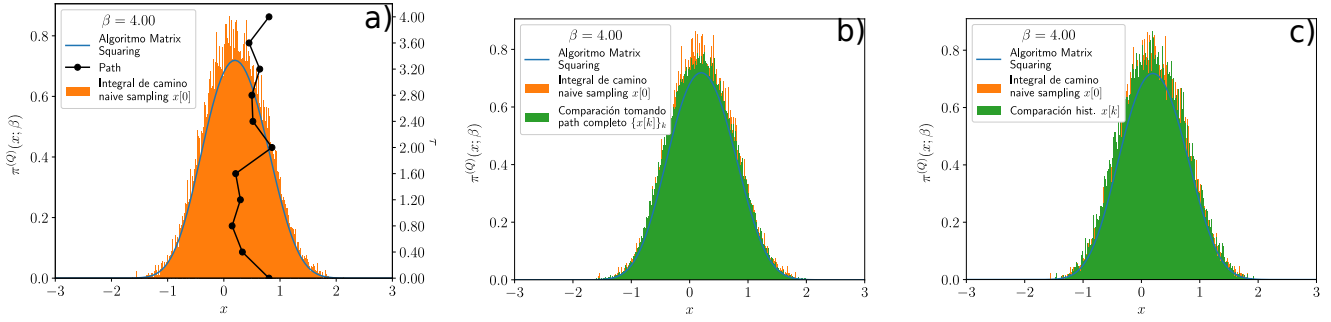


Figura 2. caption

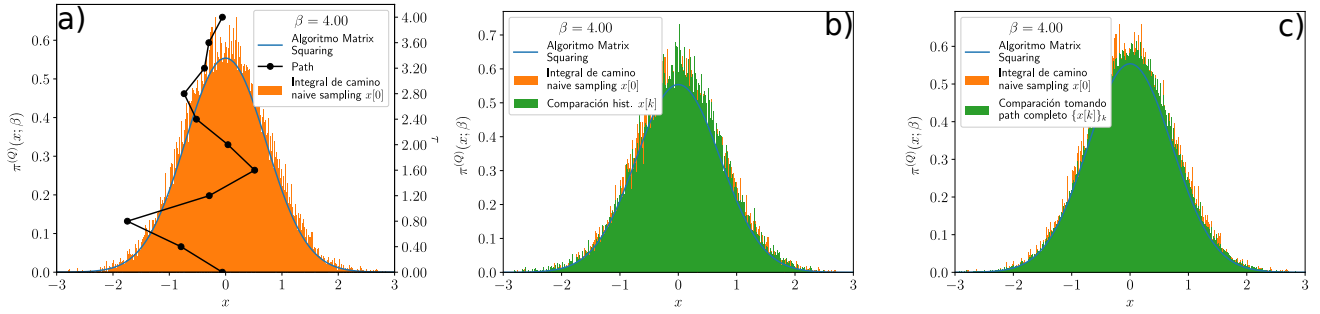


Figura 3. caption

parecidas, también tenemos un caso intermedio entre alta y baja temperatura y uno de baja temperatura. Esas conclusiones las soportamos tanto en las comparaciones de las curvas teóricas como en los histogramas generados. Siempre los histogramas de los niveles de energía corresponden con el límite que tratamos: altas temperaturas implican contribuciones apreciables de muchos niveles de energía, mientras que para bajas temperaturas contribuyen solo niveles muy próximos al estado base.

Las implementaciones de los algoritmos usados son su-

ficientemente generales y se podrían adaptar con cierta facilidad a otros sistemas de interés que sean objeto de estudio.

AGRADECIMIENTOS

Agradezco a mis compañeros de clase con los que tuve discusiones que ayudaron en la implementación del algoritmo y en las conclusiones presentadas.

-
- [1] G. Grynberg, A. Aspect, and C. Fabre, *Introduction to Quantum Optics: From the Semi-classical Approach to Quantized Light*, 1st ed. (Cambridge University Press, 2010).
 - [2] M. D. Schwartz, *Quantum Field Theory and Standard Model*, 1st ed. (Cambridge University Press, 2014) [arXiv:arXiv:1011.1669v3](#).
 - [3] F. A. Barone, H. Boschi-Filho, and C. Farina, Three methods for calculating the Feynman propagator, *American Journal of Physics* **71**, 483 (2003), [arXiv:0205085 \[quant-ph\]](#).
 - [4] B. R. Holstein, The harmonic oscillator propagator, *American Journal of Physics* **66**, 583 (1998).
 - [5] F. Kheirandish, Exact density matrix of an oscillator-bath system: Alternative derivation, *Physics Letters, Section A: General, Atomic and Solid State Physics* **382**, 3339 (2018).
 - [6] Richard P. Feynmann, *Statistical Mechanics: a Set of Lectures*, 2nd ed. (THE BENJAMIN/CUMMINGS PUBLISHING COMPANY, INC., 1972) pp. 49–51.
 - [7] C. Cohen-Tannoudji, B. Diu, and F. Laloë, *Quantum mechanics* (Wiley, New York, NY, 1977) pp. 628–631.
 - [8] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, Equation of state calculations by fast computing machines, *The Journal of Chemical Physics* **21**, 1087 (1953).
 - [9] W. K. Hastings, Monte carlo sampling methods using Markov chains and their applications, *Biometrika* **57**, 97 (1970).
 - [10] M. E. J. Newman and G. T. Barkema, *Monte Carlo Methods in Statistical Physics*, Oxford University Press, 1 (1999).

Apéndice A: Código 1: Matrix Squaring

```

1  # -*- coding: utf-8 -*-
2  from __future__ import division
3  import os
4  import numpy as np
5  import matplotlib.pyplot as plt
6  from time import time
7  import pandas as pd
8
9  # Author: Juan Esteban Aristizabal-Zuluaga
10 # date: 20200414
11
12 def rho_free(x, xp, beta):
13     """Uso: devuelve elemento de matriz dsnsidad para el caso de una partícula libre en
14     un toro infinito.
15     """
16     return (2.*np.pi*beta)**(-0.5) * np.exp(-(x-xp)**2 / (2 * beta))
17
18 def harmonic_potential(x):
19     """Uso: Devuelve valor del potencial armónico para una posición x dada"""
20     return 0.5*x**2
21
22 def anharmonic_potential(x):
23     """Devuelve valor de potencial anarmónico para una posición x dada"""
24     # return np.abs(x)*(1+np.cos(x)) #el resultado de este potencial es interesante
25     return 0.5*x**2 - x**3 + x**4
26
27 def QHO_canonical_ensemble(x, beta):
28     """
29     Uso: calcula probabilidad teórica cuántica de encontrar al oscilador armónico
30     (inmerso en un baño térmico a temperatura inversa beta) en la posición x.
31
32     Recibe:
33         x: float          -> posición
34         beta: float       -> inverso de temperatura en unidades reducidas beta = 1/T.
35
36     Devuelve:
37         probabilidad teórica cuántica en posición x para temperatura inversa beta.
38     """
39     return (np.tanh(beta/2.)/np.pi)**0.5 * np.exp(- x**2 * np.tanh(beta/2.))
40
41 def Z_QHO(beta):
42     """Uso: devuelve valor de función de partición para el QHO unidimensional"""
43     return 0.5/np.sinh(beta/2)
44
45 def E_QHO_avg_theo(beta):
46     """Uso: devuelve valor de energía interna para el QHO unidimensional"""
47     return 0.5/np.tanh(0.5*beta)
48
49 def rho_trotter(x_max=5., nx=101, beta=1, potential=harmonic_potential):
50     """
51     Uso: devuelve matriz densidad en aproximación de Trotter para altas temperaturas
52     y bajo influencia del potencial "potential".
53
54     Recibe:
55         x_max: float      -> los valores de x estarán en el intervalo (-x_max, x_max).
56         nx: int           -> número de valores de x considerados (igualmente espaciados).

```

```

57     beta: float      -> inverso de temperatura en unidades reducidas.
58     potential: func -> potencial de interacción. Debe ser solo función de x.
59
60 Devuelve:
61     rho: numpy array, shape=(nx,nx)      -> matriz densidad en aproximación de Trotter para
62                                             altas temperaturas y potencial dado.
63     grid_x: numpy array, shape=(nx,)      -> valores de x en los que está evaluada rho.
64     dx: float                             -> separación entre valores contiguos de grid_x
65     """
66
67 nx = int(nx)
68
69 # Si nx es par lo cambiamos al impar más cercano para incluir al 0 en valores de x
70 if nx%2 == 0:
71     nx = nx + 1
72
73 # Valor de la discretización de posiciones según x_max y nx dados como input
74 dx = 2 * x_max/(nx-1)
75
76 # Lista de valores de x teniendo en cuenta discretización y x_max
77 grid_x = [i*dx for i in range(-int((nx-1)/2),int((nx-1)/2 + 1))]
78
79 # Construcción de matriz densidad dada por aproximación de Trotter
80 rho = np.array([[rho_free(x , xp, beta) * np.exp(-0.5*beta*(potential(x)+potential(xp)))
81                  for x in grid_x]
82                  for xp in grid_x])
83
84 return rho, grid_x, dx
85
86 def density_matrix_squaring(rho, grid_x, N_iter=1, beta_ini=1, print_steps=True):
87     """
88     Uso:     devuelve matriz densidad luego de aplicarle algoritmo matrix squaring N_iter veces.
89             En la primera iteración se usa matriz de densidad dada por el input rho (a
90             temperatura inversa beta_ini); en las siguientes iteraciones se usa matriz densidad
91             generada por la iteración inmediatamente anterior. El sistema asociado a la matriz
92             densidad obtenida (al final de aplicar el algoritmo) está a temperatura inversa
93             beta_fin = beta_ini * 2**(N_iter).
94
95     Recibe:
96     rho: numpy array, shape=(nx,nx)      -> matriz densidad discretizada en valores dados
97                                             por x_grid.
98     grid_x: numpy array, shape=(nx,)      -> valores de x en los que está evaluada rho.
99     N_iter: int                           -> número de iteraciones del algoritmo.
100    beta_ini: float                        -> valor de inverso de temperatura asociado a la
101                                             matriz densidad rho dada como input.
102    print_steps: bool                      -> decide si muestra valores de beta en cada
103                                             iteración.
104
105 Devuelve:
106     rho: numpy array, shape=(nx,nx)      -> matriz densidad de estado rho a temperatura
107                                             inversa igual a beta_fin.
108     trace_rho: float                     -> traza de la matriz densidad a temperatura
109                                             inversa igual a beta_fin. Por la definición que
110                                             tomamos de rho, ésta es equivalente a la función
111                                             partición a dicha temperatura.
112     beta_fin: float                      -> temperatura inversa del sistema asociado a rho.
113     """
114

```

```

115     # Valor de discretización de las posiciones
116     dx = grid_x[1] - grid_x[0]
117
118     # Cálculo del valor de beta_fin según valores beta_ini y N_iter dados como input
119     beta_fin = beta_ini * 2 ** N_iter
120
121     # Itera algoritmo matrix squaring
122     if print_steps:
123         print('\nbeta_ini = %.3f'%beta_ini,
124               '\n-----')
125     for i in range(N_iter):
126         rho = dx * np.dot(rho,rho)
127         # Imprime información relevante
128         if print_steps:
129             print(u'Iteración %d)  2~%d * beta_ini --> 2~%d * beta_ini'%(i, i, i+1))
130     if print_steps:
131         print('-----\n' +
132               u'beta_fin = %.3f'%beta_fin)
133
134     # Calcula traza de rho
135     trace_rho = np.trace(rho)*dx
136
137     return rho, trace_rho, beta_fin
138
139 def save_csv(data, data_headers=None, data_index=None, file_name=None,
140             relevant_info=None, print_data=True):
141     """
142     Uso:      data debe contener listas que serán las columnas de un archivo CSV que se guardará
143              con nombre file_name. relevant_info agrega comentarios en primeras líneas del
144              archivo.
145
146     Recibe:
147         data: array of arrays, shape=(nx,ny)    -> cada columna es una columna del archivo.
148         data_headers: numpy array, shape=(ny,) -> nombres de las columnas
149         data_index: numpy array, shape=(nx,)    -> nombres de las filas
150         file_name: str                          -> nombre del archivo en el que se guardarán datos.
151         relevant_info: list of str               -> información que se agrega como comentario en
152                                                    primeras líneas. Cada elemento de esta lista
153                                                    se agrega como una nueva línea.
154         print_data: bool                       -> decide si imprime datos guardados, en pantalla.
155
156     Devuelve:
157         data_pdDF: pd.DataFrame                -> archivo con datos formato "pandas data frame".
158         guarda archivo con datos e información relevante en primera línea.
159     """
160
161     if file_name==None:
162         #path completa para este script
163         script_dir = os.path.dirname(os.path.abspath(__file__))
164         file_name = script_dir + '/' + 'file_name.csv'
165
166     data_pdDF = pd.DataFrame(data, columns=data_headers, index=data_index)
167
168     # Crea archivo CSV y agrega comentarios relevantes dados como input
169     if relevant_info is not None:
170
171         # Agregamos información relevante en primeras líneas
172         with open(file_name,mode='w') as file_csv:

```

```

173         for info in list(relevant_info):
174             file_csv.write('# '+info+'\n')
175     file_csv.close()
176
177     # Usamos pandas para escribir en archivo en formato csv.
178     with open(file_name,mode='a') as file_csv:
179         data_pdDF.to_csv(file_csv)
180     file_csv.close()
181
182 else:
183     with open(file_name,mode='w') as file_csv:
184         data_pdDF.to_csv(file_csv)
185     file_csv.close()
186
187 # Imprime datos en pantalla.
188 if print_data==True:
189     print(data_pdDF)
190
191 return data_pdDF
192
193 def run_pi_x_sq_trotter(x_max=5., nx=201, N_iter=7, beta_fin=4, potential=harmonic_potential,
194                        potential_string='harmonic_potential', print_steps=True,
195                        save_data=True, file_name=None, relevant_info=None,
196                        plot=True, save_plot=True, show_plot=True):
197     """
198     Uso:     corre algoritmo matrix squaring iterativamente (N_iter veces). En la primera
199              iteración se usa una matriz densidad en aproximación de Trotter a temperatura
200              inversa beta_ini = beta_fin * 2**(-N_iter) para potencial dado por potential;
201              en las siguientes iteraciones se usa matriz densidad generada por la iteración
202              inmediatamente anterior. Además ésta función guarda datos de pi(x;beta) vs. x
203              en archivo de texto y grafica pi(x;beta) comparándolo con teoría para el oscilador
204              armónico cuántico.
205
206     Recibe:
207         x_max: float      -> los valores de x estarán en el intervalo (-x_max,x_max).
208         nx: int           -> número de valores de x considerados.
209         N_iter: int       -> número de iteraciones del algoritmo matrix squaring.
210         beta_ini: float   -> valor de inverso de temperatura que queremos tener al final de
211                             aplicar el algoritmo matrix squaring iterativamente.
212         potential: func    -> potencial de interacción usado en aproximación de trotter. Debe
213                             ser función de x.
214         potential_string: str -> nombre del potencial (con éste nombramos los archivos que
215                             se generan).
216         print_steps: bool  -> decide si imprime los pasos del algoritmo matrix squaring.
217         save_data: bool    -> decide si guarda los datos en archivo .csv.
218         file_name: str     -> nombre de archivo CSV en que se guardan datos. Si valor es None,
219                             se guarda con nombre conveniente según parámetros relevantes.
220         plot: bool        -> decide si grafica.
221         save_plot: bool    -> decide si guarda la figura.
222         show_plot: bool    -> decide si muestra la figura en pantalla.
223
224     Devuelve:
225         rho: numpy array, shape=(nx,nx) -> matriz densidad de estado rho a temperatura
226                                             inversa igual a beta_fin.
227         trace_rho: float -> traza de la matriz densidad a temperatura
228                             inversa igual a beta_fin. Por la definición que
229                             tomamos de "rho", ésta es equivalente a la
230                             función partición en dicha temperatura.

```



```

231     grid_x: numpy array, shape=(nx,)    -> valores de x en los que está evaluada rho.
232 """
233 # Cálculo del valor de beta_ini según valores beta_fin y N_iter dados como input
234 beta_ini = beta_fin * 2**(-N_iter)
235
236 # Cálculo de rho con aproximación de Trotter
237 rho, grid_x, dx = rho_trotter(x_max, nx, beta_ini, potential)
238 grid_x = np.array(grid_x)
239
240 # Aproximación de rho con matrix squaring iterado N_iter veces.
241 rho, trace_rho, beta_fin_2 = density_matrix_squaring(rho, grid_x, N_iter,
242                                                     beta_ini, print_steps)
243 print('-----')
244 + '-----\n'
245 + u'Matrix squaring: beta_ini = %.3f --> beta_fin = %.3f'%(beta_ini, beta_fin_2)
246 + u'  N_iter = %d   Z(beta_fin) = Tr(rho(beta_fin)) = %.3E \n'%(N_iter,trace_rho)
247 + '-----'
248 + '-----'
249 )
250
251 # Normalización de rho a 1 y cálculo de densidades de probabilidad para valores en grid_x.
252 rho_normalized = np.copy(rho)/trace_rho
253 x_weights = np.diag(rho_normalized)
254
255 # Guarda datos en archivo CSV.
256 script_dir = os.path.dirname(os.path.abspath(__file__)) #path completa para este script
257
258 if save_data:
259
260     # Nombre del archivo .csv en el que guardamos valores de pi(x;beta_fin).
261     if file_name is None:
262         csv_file_name = (script_dir
263                         + u'/pi_x-ms-%s-beta_fin_%.3f-x_max_%.3f-nx_%d-N_iter_%d.csv'
264                         %(potential_string,beta_fin,x_max,nx,N_iter))
265     else:
266         csv_file_name = script_dir + '/' + file_name + '.csv'
267
268     # Información relevante para agregar como comentario al archivo csv.
269     if relevant_info is None:
270         relevant_info = ['pi(x;beta_fin) computed using matrix squaring algorithm and'
271                         + ' Trotter approximation. Parameters:',
272                         u'%s   x_max = %.3f   nx = %d  '%(potential_string,x_max,nx)
273                         + u'N_iter = %d   beta_ini = %.3f  '%(N_iter,beta_ini,)
274                         + u'beta_fin = %.3f'%beta_fin]
275
276     # Guardamos valores de pi(x;beta_fin) en archivo csv.
277     pi_x_data = np.array([grid_x.copy(),x_weights.copy()])
278     pi_x_data_headers = ['position_x','prob_density']
279     pi_x_data = save_csv(pi_x_data.transpose(),pi_x_data_headers, None, csv_file_name,
280                         relevant_info, print_data=0)
281
282 # Gráfica y comparación con teoría
283 if plot:
284
285     plt.figure(figsize=(8,5))
286     plt.plot(grid_x, x_weights,
287             label = 'Matrix squaring +\nfórmula de Trotter.\n$N=%d$ iteraciones\n$dx=%.3E$'
288                   %(N_iter,dx))

```



```

289 plt.plot(grid_x, QHO_canonical_ensemble(grid_x,beta_fin), label=u'Valor teórico QHO')
290 plt.xlabel(u'x')
291 plt.ylabel(u'$\pi^{\{Q\}}(x;\\beta)$')
292 plt.legend(loc='best',title=u'$\\beta$=%.2f'%beta_fin)
293 plt.tight_layout()
294
295 if save_plot:
296     if file_name is None:
297         plot_file_name = (script_dir
298             + u'/pi_x-ms-plot-%s-beta_fin_%.3f-x_max_%.3f-nx_%d-N_iter_%d.eps'
299             %(potential_string,beta_fin,x_max,nx,N_iter))
300     else:
301         plot_file_name = script_dir+u'/pi_x-ms-plot-'+file_name+'.eps'
302     plt.savefig(plot_file_name)
303
304 if show_plot:
305     plt.show()
306 plt.close()
307
308 return rho, trace_rho, grid_x
309
310 def Z_several_values(temp_min=1./10, temp_max=1/2., N_temp=10, save_Z_csv=True,
311     Z_file_name = None, relevant_info_Z = None, print_Z_data = True,
312     x_max=7., nx=201, N_iter=7, potential = harmonic_potential,
313     potential_string = 'harmonic_potential', print_steps=False,
314     save_pi_x_data=False, pi_x_file_name=None, relevant_info_pi_x=None,
315     plot=False, save_plot=False, show_plot=False):
316     """
317     Uso:      calcula varios valores para la función partición, Z, usando operador densidad
318               aproximado aproximado por el algoritmo matrix squaring.
319
320     Recibe:
321         temp_min: float          -> Z se calcula para valores de beta en (1/temp_min,1/temp_max)
322                                   con N_temp valores igualmente espaciados.
323         temp_max: float.
324         N_temp: int.
325         save_Z_csv: bool         -> decide si guarda valores calculados en archivo CSV.
326         Z_file_name: str         -> nombre del archivo en el que se guardan datos de Z. Si valor
327                                   es None, se guarda con nombre conveniente según parámetros
328                                   relevantes.
329         relevant_info_Z: list    -> información relevante se añade a primeras líneas del archivo.
330                                   Cada str separada por una coma en la lista se añade como una
331                                   nueva línea.
332         print_Z_data: bool       -> imprime datos de Z en pantalla.
333         *args: tuple             -> argumentos de run_pi_x_sq_trotter
334
335     Devuelve:
336         Z_data: list, shape=(3,)
337         Z_data[0]: list, shape(N_temp,) -> contiene valores de beta en los que está evaluada Z.
338         Z_data[1]: list, shape(N_temp,) -> contiene valores de T en los que está evaluada Z.
339         Z_data[2]: list, shape(N_temp,) -> contiene valores de Z.
340                                   Z(beta) = Z(1/T) =
341                                   Z_data[0](Z_data[1]) = Z_data[0](Z_data[2])
342     """
343
344     # Transforma valores de beta en valores de T y calcula lista de beta.
345     beta_max = 1./temp_min
346     beta_min = 1./temp_max

```

```

347 N_temp = int(N_temp)
348 beta_array = np.linspace(beta_max,beta_min,N_temp)
349 Z = []
350
351 # Calcula valores de Z para valores de beta especificados en beta_array.
352 for beta_fin in beta_array:
353     rho, trace_rho, grid_x = run_pi_x_sq_trotter(x_max, nx, N_iter, beta_fin, potential,
354                                                potential_string, print_steps,
355                                                save_pi_x_data, file_name,
356                                                relevant_info, plot, save_plot, show_plot)
357     Z.append(trace_rho)
358
359 # Calcula el output de la función.
360 Z_data = np.array([beta_array.copy(), 1./beta_array.copy(), Z.copy()], dtype=float)
361
362 # Guarda datos de Z en archivo CSV.
363 if save_Z_csv == True:
364
365     script_dir = os.path.dirname(os.path.abspath(__file__))
366
367     if Z_file_name is None:
368         Z_file_name = ('Z-ms-%s-beta_max_%.3f-%s'%(potential_string,1./temp_min)
369                       + 'beta_min_%.3f-N_temp_%d-x_max_%.3f-%s'%(1./temp_max,N_temp,x_max)
370                       + 'nx_%d-N_iter_%d.csv'%(nx, N_iter))
371
372     Z_file_name = script_dir + '/' + Z_file_name
373
374     if relevant_info_Z is None:
375         relevant_info_Z = ['Partition function at several temperatures',
376                           '%s    beta_max = %.3f    '%(potential_string,1./temp_min)
377                           + 'beta_min = %.3f    N_temp = %d    '%(1./temp_max,N_temp)
378                           + 'x_max = %.3f    nx = %d    N_iter = %d'%(x_max,nx, N_iter)]
379
380     Z_data_headers = ['beta', 'temperature', 'Z']
381     Z_data = save_csv(Z_data.transpose(), Z_data_headers, None, Z_file_name, relevant_info_Z,
382                      print_data=False)
383
384     if print_Z_data == True:
385         print(Z_data)
386
387     return Z_data
388
389 def average_energy(read_Z_data=True, generate_Z_data=False, Z_file_name = None,
390                  plot_energy=True, save_plot_E=True, show_plot_E=True,
391                  E_plot_name=None,
392                  temp_min=1./10, temp_max=1/2., N_temp=10, save_Z_csv=True,
393                  relevant_info_Z=None, print_Z_data=True,
394                  x_max=7., nx=201, N_iter=7, potential=harmonic_potential,
395                  potential_string='harmonic_potential', print_steps=False,
396                  save_pi_x_data=False, pi_x_file_name=None, relevant_info_pi_x=None,
397                  plot_pi_x=False, save_plot_pi_x=False, show_plot_pi_x=False):
398     """
399     Uso:    calcula energía promedio, E, del sistema en cuestión dado por potential.
400     Se puede decidir si se leen datos de función partición o se generan,
401     ya que  $E = - (d/d \beta) \log(Z)$ .
402
403     Recibe:

```

```

405     read_Z_data: bool      -> decide si se leen datos de Z de un archivo con nombre
406                             Z_file_name.
407     generate_Z_data: bool  -> decide si genera datos de Z.
408     Nota: read_Z_data y generate_Z_data son excluyentes. Se analiza primero primera opción
409     Z_file_name: str      -> nombre del archivo en del que se leerá o en el que se
410                             guardarán datos de Z. Si valor es None, se guarda con nombre
411                             conveniente según parámetros relevantes.
412     plot_energy: bool     -> decide si gráfica energía.
413     save_plot_E: bool     -> decide si guarda gráfica de energía. Nótese que si
414                             plot_energy=False, no se generará gráfica.
415     show_plot_E: bool     -> decide si muestra gráfica de E en pantalla
416     E_plot_name: str      -> nombre para guardar gráfico de E.
417     *args: tuple          -> argumentos de Z_several_values
418
419     Devuelve:
420         E_avg: list        -> valores de energía promedio para beta especificados por
421                             beta__read
422         beta_read: list
423     """
424
425     # Decide si lee o genera datos de Z.
426     if read_Z_data:
427         Z_file_read = pd.read_csv(Z_file_name, index_col=0, comment='#')
428     elif generate_Z_data:
429         t_0 = time()
430         Z_data = Z_several_values(temp_min, temp_max, N_temp, save_Z_csv, Z_file_name,
431                                   relevant_info_Z, print_Z_data, x_max, nx, N_iter, potential,
432                                   potential_string, print_steps, save_pi_x_data, pi_x_file_name,
433                                   relevant_info_pi_x, plot_pi_x, save_plot_pi_x, show_plot_pi_x)
434         t_1 = time()
435         print('-----\n'
436               + '%d values of Z(beta) generated --> %.3f sec.'%(N_temp,t_1-t_0))
437         Z_file_read = Z_data
438     else:
439         print('Elegir si se generan o se leen los datos para la función partición, Z.\n'
440               + 'Estas opciones son mutuamente excluyentes. Si se seleccionan las dos, el'
441               + 'algoritmo escoge leer los datos.')
442
443
444     beta_read = Z_file_read['beta']
445     temp_read = Z_file_read['temperature']
446     Z_read = Z_file_read['Z']
447
448     # Calcula energía promedio.
449     E_avg = np.gradient(-np.log(Z_read), beta_read)
450
451     # Grafica.
452     if plot_energy:
453         plt.figure(figsize=(8,5))
454         plt.plot(temp_read, E_avg, label=u'$\langle E \rangle$ via path integral\ naive sampling')
455         plt.plot(temp_read, E_QHO_avg_theo(beta_read), label=u'$\langle E \rangle$ teórico')
456         plt.legend(loc='best')
457         plt.xlabel(u'$T$')
458         plt.ylabel(u'$\langle E \rangle$')
459         if save_plot_E:
460             script_dir = os.path.dirname(os.path.abspath(__file__))
461             if E_plot_name is None:
462                 E_plot_name = ('E-ms-plot-%s-beta_max-%.3f'%(potential_string, 1./temp_min)

```

```

463         + 'beta_min_%.3f-N_temp_%d-x_max_%.3f-%(1./temp_max,N_temp,x_max)
464         + 'nx_%d-N_iter_%d.eps'%(nx, N_iter))
465     E_plot_name = script_dir + '/' + E_plot_name
466     plt.savefig(E_plot_name)
467     if show_plot_E:
468         plt.show()
469     plt.close()
470
471     return E_avg, beta_read.to_numpy()
472
473 def calc_error(x, xp, dx):
474     """
475     Uso:     error acumulado en cálculo computacional de  $\pi(x; \beta)$  comparado
476             con valor teórico
477     """
478     x, xp = np.array(x), np.array(xp)
479     N = len(x)
480     if N != len(xp):
481         raise Exception('x y xp deben ser del mismo tamaño.')
482     else:
483         return np.sum(np.abs(x-xp))*dx
484
485 def optimization(generate_opt_data=True, read_opt_data=False, beta_fin=4, x_max=5,
486                 potential=harmonic_potential, potential_string='harmonic_potential',
487                 nx_min=50, nx_max=1000, nx_sampling=50, N_iter_min=1, N_iter_max=20,
488                 save_opt_data=False, opt_data_file_name=None, opt_relevant_info=None,
489                 plot=True, show_plot=True, save_plot=True, opt_plot_file_name=None):
490     """
491     Uso:     calcula diferentes valores de error usando calc_error() para encontrar valores de
492             dx y beta_ini óptimos para correr el algoritmo (óptimos = que minimicen error)
493
494     Recibe:
495         generate_opt_data: bool -> decide si genera datos para optimización.
496         read_opt_data: bool -> decide si lee datos para optimización.
497         Nota: generate_opt_data y read_opt_data son excluyentes. Se evalúa primero la primera.
498         nx_min: int
499         nx_max: int -> se relaciona con  $dx = 2*x_{max}/(nx-1)$ .
500         nx_sampling: int -> se generan nx mediante range(nx_max, nx_min, -1*nx_sampling).
501         N_iter_min: int
502         N_iter_max: int -> se relaciona con  $\beta_{ini} = \beta_{fin} * (-N_{iter})$ . Se generan
503             valores de N_iter con range(N_iter_max, N_iter_min-1, -1).
504         save_opt_data: bool -> decide si guarda datos de optimización en archivo CSV.
505         opt_data_file_name: str -> nombre de archivo para datos de optimización.
506         plot: bool -> decide si grafica optimización.
507         show_plot: bool -> decide si muestra optimización.
508         save_plot: bool -> decide si guarda optimización.
509         opt_plot_file_name: str -> nombre de gráfico de optimización. Si valor es None, se
510             guarda con nombre conveniente según parámetros relevantes.
511
512     Devuelve:
513         error: list, shape=(nb, ndx) -> valores de calc_error para diferentes valores de dx y
514             beta_ini. dx incrementa de izquierda a derecha en lista
515             y beta_ini incrementa de arriba a abajo.
516         dx_grid: list, shape=(ndx,) -> valores de dx para los que se calcula error.
517         beta_ini_grid: list, shape=(nb,) -> valores de beta_ini para los que calcula error.
518     """
519
520     t_0 = time()

```

```

521
522 # Decide si genera o lee datos.
523 if generate_opt_data:
524     N_iter_min = int(N_iter_min)
525     N_iter_max = int(N_iter_max)
526     nx_min = int(nx_min)
527     nx_max = int(nx_max)
528
529     if nx_min%2==1:
530         nx_min -= 1
531     if nx_max%2==0:
532         nx_max += 1
533
534     # Crea valores de nx y N_iter (equivalente a generar valores de dx y beta_ini)
535     nx_values = range(nx_max,nx_min,-1*nx_sampling)
536     N_iter_values = range(N_iter_max,N_iter_min-1,-1)
537
538     dx_grid = [2*x_max/(nx-1) for nx in nx_values]
539     beta_ini_grid = [beta_fin * 2**(-N_iter) for N_iter in N_iter_values]
540
541     error = []
542
543     # Calcula error para cada valor de nx y N_iter especificado
544     # (equivalentemente dx y beta_ini).
545     for N_iter in N_iter_values:
546         row = []
547         for nx in nx_values:
548             rho,trace_rho,grid_x = run_pi_x_sq_trotter(x_max, nx, N_iter, beta_fin,
549                                                         potential, potential_string,
550                                                         False, False, None, None, False,
551                                                         False, False)
552             grid_x = np.array(grid_x)
553             dx = grid_x[1]-grid_x[0]
554             rho_normalized = np.copy(rho)/trace_rho
555             pi_x = np.diag(rho_normalized)
556             theoretical_pi_x = QHO_canonical_ensemble(grid_x,beta_fin)
557             error_comp_theo = calc_error(pi_x,theoretical_pi_x,dx)
558             row.append(error_comp_theo)
559         error.append(row)
560
561 elif read_opt_data:
562     error = pd.read_csv(opt_data_file_name, index_col=0, comment='#')
563     dx_grid = error.columns.to_numpy()
564     beta_ini_grid = error.index.to_numpy()
565     error = error.to_numpy()
566
567 else:
568     raise Exception('Escoja si generar o leer datos en optimization(.)')
569
570 # Toma valores de error en cálculo de Z (nan e inf) y los reemplaza por
571 # el valor de mayor error en el gráfico.
572 try:
573     error = np.where(np.isinf(error),0,error)
574     error = np.where(np.isnan(error),0,error)
575     nan_value = 1.3*np.max(error)
576     error = np.where(error==0, float('nan'), error)
577 except:
578     nan_value = 0

```

```

579 error = np.nan_to_num(error, nan=nan_value, posinf=nan_value, neginf=nan_value)
580
581 script_dir = os.path.dirname(os.path.abspath(__file__))
582
583 # Guarda datos (solo si fueron generados y se escoje guardar)
584 if generate_opt_data and save_opt_data:
585
586     if opt_data_file_name is None:
587         opt_data_file_name = ('pi_x-ms-opt-%s-beta_fin_%.3f'%(potential_string, beta_fin)
588                               + '-x_max_%.3f-nx_min_%d-nx_max_%d'%(x_max, nx_min, nx_max)
589                               + '-nx_sampling_%d-N_iter_min_%d'%(nx_sampling, N_iter_min)
590                               + '-N_iter_max_%d.csv'%(N_iter_max))
591
592     opt_data_file_name = script_dir + '/' + opt_data_file_name
593     if opt_relevant_info is None:
594         opt_relevant_info = ['Optimization of parameters dx and beta_ini of matrix squaring'
595                               + ' algorithm', '%s beta_fin = %.3f'%(potential_string, beta_fin)
596                               + 'x_max = %.3f nx_min = %d nx_max = %d'%(x_max, nx_min, nx_max)
597                               + 'nx_sampling = %d N_iter_min = %d'%(nx_sampling, N_iter_min)
598                               + 'N_iter_max = %d'%(N_iter_max)]
599
600     save_csv(error, dx_grid, beta_ini_grid, opt_data_file_name, opt_relevant_info)
601
602 t_1 = time()
603
604 # Grafica.
605 if plot:
606
607     fig, ax = plt.subplots(1, 1)
608
609     DX, BETA_INI = np.meshgrid(dx_grid, beta_ini_grid)
610     cp = plt.pcolormesh(DX, BETA_INI, error)
611     plt.colorbar(cp)
612
613     ax.set_ylabel(u'$\\beta_{ini}$')
614     ax.set_xlabel('$dx$')
615     plt.tight_layout()
616
617     if save_plot:
618
619         if opt_plot_file_name is None:
620             opt_plot_file_name = \
621                 ('pi_x-ms-opt-plot-%s-beta_fin_%.3f'%(potential_string, beta_fin)
622                 + '-x_max_%.3f-nx_min_%d-nx_max_%d'%(x_max, nx_min, nx_max)
623                 + '-nx_sampling_%d-N_iter_min_%d'%(nx_sampling, N_iter_min)
624                 + '-N_iter_max_%d.eps'%(N_iter_max))
625
626             opt_plot_file_name = script_dir + '/' + opt_plot_file_name
627
628             plt.savefig(opt_plot_file_name)
629
630         if show_plot:
631             plt.show()
632
633     plt.close()
634
635 comp_time = t_1 - t_0
636

```

[illegible]


```

695 #
696 #####
697
698
699
700 #####
701 # CORRE ALGORITMO PARA CÁLCULO DE ENERGÍA INTERNA
702 #
703
704 # Parámetros técnicos función partición y cálculo de energía
705 read_Z_data = False
706 generate_Z_data = True
707 Z_file_name = None
708 plot_energy = True
709 save_plot_E = True
710 show_plot_E = True
711 E_plot_name = None
712
713 # Parámetros físicos para calcular Z y <E>
714 temp_min = 1./10
715 temp_max = 1./2
716 N_temp = 10
717 potential, potential_string = harmonic_potential, 'harmonic_potential'
718
719 # Más parámetros técnicos
720 save_Z_csv = True
721 relevant_info_Z = None
722 print_Z_data = False
723 x_max = 7.
724 nx = 201
725 N_iter = 7
726 print_steps = False
727 save_pi_x_data = False
728 pi_x_file_name = None
729 relevant_info_pi_x = None
730 plot_pi_x = False
731 save_plot_pi_x = False
732 show_plot_pi_x = False
733
734 if run_avg_energy:
735     average_energy(read_Z_data, generate_Z_data, Z_file_name, plot_energy, save_plot_E,
736                   show_plot_E, E_plot_name,
737                   temp_min, temp_max, N_temp, save_Z_csv, relevant_info_Z, print_Z_data,
738                   x_max, nx, N_iter, potential, potential_string, print_steps, save_pi_x_data,
739                   pi_x_file_name, relevant_info_pi_x, plot_pi_x, save_plot_pi_x, show_plot_pi_x)
740
741 #
742 #
743 #####
744
745
746
747 #####
748 # CORRE ALGORITMO PARA OPTIMIZACIÓN DE DX Y BETA_INI
749 #
750
751 # Parámetros físicos
752 beta_fin = 4

```

```

753 x_max = 5
754 potential, potential_string = harmonic_potential, 'harmonic_potential'
755 nx_min = 10
756 nx_max = 310
757 nx_sampling = 60
758 N_iter_min = 8
759 N_iter_max = 20
760
761 # Parámetros técnicos
762 generate_opt_data = True
763 read_opt_data = False
764 save_opt_data = True
765 opt_data_file_name = None
766 opt_relevant_info = None
767 plot_opt = True
768 show_opt_plot = True
769 save_plot_opt = True
770 opt_plot_file_name = None
771
772 if run_optimization:
773     error, dx_grid, beta_ini_grid, comp_time = \
774         optimization(generate_opt_data, read_opt_data, beta_fin, x_max, potential,
775                     potential_string, nx_min, nx_max, nx_sampling, N_iter_min,
776                     N_iter_max, save_opt_data, opt_data_file_name, opt_relevant_info,
777                     plot_opt, show_opt_plot, save_plot_opt, opt_plot_file_name)
778     print('-----'
779           + '-----\n'
780           + 'Optimization:  beta_fin=%.3f,  x_max=%.3f,  potential=%s\n \
781             nx_min=%d, nx_max=%d,  N_iter_min=%d,  N_iter_max=%d\n \
782             computation time = %.3f sec.\n'%(beta_fin,x_max,potential_string,nx_min,
783                                               nx_max,N_iter_min,N_iter_max,comp_time)
784           + '-----'
785           + '-----')
786
787 #
788 #
789 #####

```

Apéndice B: Código 2: Naive Path Integral Montecarlo Sampling
