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

Juan Esteban Aristizabal Zuluaga
Instituto de Física, Universidad de Antioquia.
(Dated: 23 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 formalizmo de intrgrales 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 ámpliamente 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 dad 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

Consideraremos los sistemas en unidades reducidas, es decir, con sus variables adimensionalizadas.

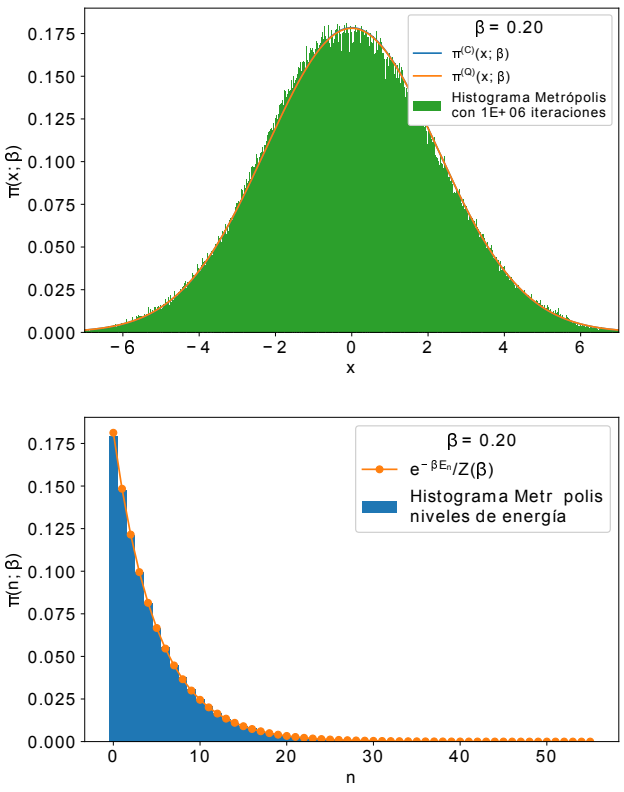

Figura 1. Arriba: densidad de probabilidad de encontrar a la «partícula» cuántica en una posición dada, cuando está en presencia de un potencial armónico y de baño térmico a temperatura definida por $\beta = 1/T = 0.2$. Mostramos los resultados teóricos clásico y cuántico como línea continua y el histograma que resulta del algoritmo Metrópolis usando 10^6 iteraciones y $\delta x = 0.5$. Observamos que éste es un límite de alta temperatura ya que las distribuciones teóricas clásica y cuántica se solapan en gran medida y son muy similares. Abajo: histograma de niveles de energía obtenido con algoritmo Metrópolis y los respectivos valores teóricos. Notamos que muchos niveles de energía contribuyen en este caso que hemos considerado de alta temperatura. Además, los valores calculados por el algoritmo se acercan en gran medida a los teóricos

A. Caso Clásico

B. Caso cuántico

III. RESULTADOS Y DISCUSIÓN

A. Límite de muy baja temperatura $T \to 0$

B. Temperatura finita $T \neq 0$

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 probablididad cuántica mencionada tiende a la clásica, conforme se espera también desde la física estadística.

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 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 suficientemente 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. Newmanand G. T. Barkema, Monte Carlo Methods in Statistical Physics, Oxford University Press , 1 (1999).

Apéndice A: Código 1: Matrix Squaring

```
# -*- coding: utf-8 -*-
from __future__ import division
import os
import numpy as np
import matplotlib.pyplot as plt
from time import time
import pandas as pd

# Author: Juan Esteban Aristizabal-Zuluaga
# date: 20200414

def rho_free(x,xp,beta):
    """Uso:    devuelve elemento de matriz dsnsidad para el caso de una partícula libre en
    un toro infinito.
    """
    return (2.*np.pi*beta)**(-0.5) * np.exp(-(x-xp)**2 / (2 * beta))

def harmonic_potential(x):
    """Uso: Devuelve valor del potencial armónico para una posición x dada"""
    return 0.5*x**2

def anharmonic_potential(x):
    """Devuelve valor de potencial anarmónico para una posición x dada"""
    # return np.abs(x)*(1+np.cos(x)) #el resultado de este potencial es interesante
    return 0.5*x**2 - x**3 + x**4

def QHO_canonical_ensemble(x,beta):
    """
    Uso:    calcula probabilidad teórica cuántica de encontrar al oscilador armónico
            (inmerso en un baño térmico a temperatura inversa beta) en la posición x.

    Recibe:
        x: float            -> posición
        beta: float         -> inverso de temperatura en unidades reducidas beta = 1/T.

```

```
    Devuelve:
        probabilidad teórica cuántica en posición x para temperatura inversa beta.
    """
    return (np.tanh(beta/2.)/np.pi)**0.5 * np.exp(- x**2 * np.tanh(beta/2.))

def Z_QHO(beta):
    """Uso: devuelve valor de función de partición para el QHO unidimensional"""
    return 0.5/np.sinh(beta/2)

def E_QHO_avg_theo(beta):
    """Uso: devuelve valor de energía interna para el QHO unidimensional"""
    return 0.5/np.tanh(0.5*beta)

def rho_trotter(x_max=5., nx=101, beta=1, potential=harmonic_potential):
    """
    Uso:    devuelve matriz densidad en aproximación de Trotter para altas temperaturas
            y bajo influencia del potencial "potential".

    Recibe:
        x_max: float    -> los valores de x estarán en el intervalo (-x_max,x_max).
        nx: int         -> número de valores de x considerados (igualmente espaciados).
        beta: float     -> inverso de temperatura en unidades reducidas.
        potential: func -> potencial de interacción. Debe ser solo función de x.

    Devuelve:
        rho: numpy array, shape=(nx,nx)     ->  matriz densidad en aproximación de Trotter para
                                                altas temperaturas y  potencial dado.
        grid_x: numpy array, shape=(nx,)    ->  valores de x en los que está evaluada rho.
        dx: float                           ->  separación entre valores contiguos de grid_x
    """

    nx = int(nx)

    # Si nx es par lo cambiamos al impar más cercano para incluir al 0 en valores de x
    if nx%2 == 0:
        nx = nx + 1

    # Valor de la discretización de posiciones según x_max y nx dados como input
    dx = 2 * x_max/(nx-1)

    # Lista de valores de x teniendo en cuenta discretización y x_max
    grid_x = [i*dx for i in range(-int((nx-1)/2),int((nx-1)/2 + 1))]

    # Construcción de matriz densidad dada por aproximación de Trotter
    rho = np.array([[rho_free(x , xp, beta) * np.exp(-0.5*beta*(potential(x)+potential(xp)))
                     for x in grid_x]
                     for xp in grid_x])

    return rho, grid_x, dx

def density_matrix_squaring(rho, grid_x, N_iter=1, beta_ini=1, print_steps=True):
    """
    Uso:    devuelve matriz densidad luego de aplicarle algoritmo matrix squaring N_iter veces.
            En la primera iteración se usa matriz de densidad dada por el input rho (a
            temperatura inversa beta_ini); en las siguientes iteraciones se usa matriz densidad
            generada por la iteración inmediatamente anterior. El sistema asociado a la matriz
            densidad obtenida (al final de aplicar el algoritmo) está a temperatura inversa
            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
```

```
                                                primeras líneas. Cada elemento de esta lista
                                                se agrega como una nueva línea.
        print_data: bool                    ->  decide si imprime datos guardados, en pantalla.

    Devuelve:
        data_pdDF: pd.DataFrame             ->  archivo con datos formato "pandas data frame".
        guarda archivo con datos e inforamación relevante en primera línea.
    """

    if file_name==None:
        #path completa para este script
        script_dir = os.path.dirname(os.path.abspath(__file__))
        file_name = script_dir + '/' + 'file_name.csv'

    data_pdDF = pd.DataFrame(data, columns=data_headers, index=data_index)

    # Crea archivo CSV y agrega comentarios relevantes dados como input
    if relevant_info is not None:

        # Agregamos información relevante en primeras líneas
        with open(file_name,mode='w') as file_csv:
            for info in list(relevant_info):
                file_csv.write('# '+info+'\n')
        file_csv.close()

        # Usamos pandas para escribir en archivo en formato csv.
        with open(file_name,mode='a') as file_csv:
            data_pdDF.to_csv(file_csv)
        file_csv.close()

    else:
        with open(file_name,mode='w') as file_csv:
            data_pdDF.to_csv(file_csv)
        file_csv.close()

    # Imprime datos en pantalla.
    if print_data==True:
        print(data_pdDF)

    return data_pdDF

def run_pi_x_sq_trotter(x_max=5., nx=201, N_iter=7, beta_fin=4, potential=harmonic_potential,
                        potential_string='harmonic_potential', print_steps=True,
                        save_data=True, file_name=None, relevant_info=None,
                        plot=True, save_plot=True, show_plot=True):
    """
    Uso:    corre algoritmo matrix squaring iterativamente (N_iter veces). En la primera
            iteración se usa una matriz densidad en aproximación de Trotter a temperatura
            inversa beta_ini = beta_fin * 2**(-N_iter) para potencial dado por potential;
            en las siguientes iteraciones se usa matriz densidad generada por la iteración
            inmediatamente anterior. Además ésta función guarda datos de pi(x;beta) vs. x
            en archivo de texto y grafica pi(x;beta) comparándolo con teoría para el oscilador
            armónico cuántico.

    Recibe:
        x_max: float        ->  los valores de x estarán en el intervalo (-x_max,x_max).
        nx: int             ->  número de valores de x considerados.
        N_iter: int         ->  número de iteraciones del algoritmo matrix squaring.
```

```
        beta_ini: float     ->  valor de inverso de temperatura que queremos tener al final de
                                aplicar el algoritmo matrix squaring iterativamente.
        potential: func     ->  potencial de interacción usado en aproximación de trotter. Debe
                                ser función de x.
        potential_string: str   ->  nombre del potencial (con éste nombramos los archivos que
                                    se generan).
        print_steps: bool   ->  decide si imprime los pasos del algoritmo matrix squaring.
        save_data: bool     ->  decide si guarda los datos en archivo .csv.
        file_name: str      ->  nombre de archivo CSV en que se guardan datos. Si valor es None,
                                se guarda con nombre conveniente según parámetros relevantes.
        plot: bool          ->  decide si grafica.
        save_plot: bool     ->  decide si guarda la figura.
        show_plot: bool     ->  decide si muestra la figura en pantalla.

    Devuelve:
        rho: numpy array, shape=(nx,nx)     ->  matriz densidad de estado rho a temperatura
                                                inversa igual a beta_fin.
        trace_rho: float                    ->  traza de la matriz densidad a temperatura
                                                inversa igual a beta_fin. Por la definición que
                                                tomamos de "rho", ésta es equivalente a la
                                                función partición en dicha temperatura.
        grid_x: numpy array, shape=(nx,)    ->  valores de x en los que está evaluada rho.
    """
    # Cálculo del valor de beta_ini según valores beta_fin y N_iter dados como input
    beta_ini = beta_fin * 2**(-N_iter)

    # Cálculo de rho con aproximación de Trotter
    rho, grid_x, dx = rho_trotter(x_max, nx, beta_ini, potential)
    grid_x = np.array(grid_x)

    # Aproximación de rho con matrix squaring iterado N_iter veces.
    rho, trace_rho, beta_fin_2 = density_matrix_squaring(rho, grid_x, N_iter,
                                                         beta_ini, print_steps)
    print('------------------------------------------------------------'
          + '------------------------------------------------------------\n'
          + u'Matrix squaring: beta_ini = %.3f --> beta_fin = %.3f'%(beta_ini, beta_fin_2)
          + u'   N_iter = %d   Z(beta_fin) = Tr(rho(beta_fin)) = %.3E \n'%(N_iter,trace_rho)
          + '------------------------------------------------------------'
          + '------------------------------------------------------------'
          )

    # Normalización de rho a 1 y cálculo de densidades de probabilidad para valores en grid_x.
    rho_normalized = np.copy(rho)/trace_rho
    x_weights = np.diag(rho_normalized)

    # Guarda datos en archivo CSV.
    script_dir = os.path.dirname(os.path.abspath(__file__)) #path completa para este script

    if save_data:

        # Nombre del archivo .csv en el que guardamos valores de pi(x;beta_fin).
        if file_name is None:
            csv_file_name = (script_dir
                             + u'/pi_x-ms-%s-beta_fin_%.3f-x_max_%.3f-nx_%d-N_iter_%d.csv'
                             %(potential_string,beta_fin,x_max,nx,N_iter))
        else:
            csv_file_name = script_dir + '/' + file_name + '.csv'

```

```
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.
```

```
        Z_file_name: str        ->  nombre del archivo en el que se guardan datos de Z. Si valor
                                    es None, se guarda con nombre conveniente según parámetros
                                    relevantes.
        relevant_info_Z: list   ->  infrmación relevante se añade a primeras líneas del archivo.
                                    Cada str separada por una coma en la lista se añade como una
                                    nueva línea.
        print_Z_data: bool      ->  imprime datos de Z en pantalla.
        *args: tuple            ->  argumentos de run_pi_x_sq_trotter

    Devuelve:
        Z_data: list, shape=(3,)
        Z_data[0]: list, shape(N_temp,) -> contiene valores de beta en los que está evaluada Z.
        Z_data[1]: list, shape(N_temp,) -> contiene valores de T en los que está evaluada Z.
        Z_data[2]: list, shape(N_temp,) -> contiene valores de Z.
                                           Z(beta) = Z(1/T) =
                                           Z_data[0](Z_data[1]) = Z_data[0](Z_data[2])
    """

    # Transforma valores de beta en valores de T y calcula lista de beta.
    beta_max = 1./temp_min
    beta_min = 1./temp_max
    N_temp = int(N_temp)
    beta_array = np.linspace(beta_max,beta_min,N_temp)
    Z = []

    # Calcula valores de Z para valores de beta especificados en beta_array.
    for beta_fin in beta_array:
        rho, trace_rho, grid_x = run_pi_x_sq_trotter(x_max, nx, N_iter, beta_fin, potential,
                                                     potential_string, print_steps,
                                                     save_pi_x_data, file_name,
                                                     relevant_info, plot, save_plot, show_plot)
        Z.append(trace_rho)

    # Calcula el output de la función.
    Z_data = np.array([beta_array.copy(), 1./beta_array.copy(), Z.copy()], dtype=float)

    # Guarda datos de Z en archivo CSV.
    if save_Z_csv == True:

        script_dir = os.path.dirname(os.path.abspath(__file__))

        if Z_file_name is None:
            Z_file_name = ('Z-ms-%s-beta_max_%.3f-'%(potential_string,1./temp_min)
                           + 'beta_min_%.3f-N_temp_%d-x_max_%.3f-'%(1./temp_max,N_temp,x_max)
                           + 'nx_%d-N_iter_%d.csv'%(nx, N_iter))

        Z_file_name = script_dir + '/' + Z_file_name

        if relevant_info_Z is None:
            relevant_info_Z = ['Partition function at several temperatures',
                               '%s   beta_max = %.3f   '%(potential_string,1./temp_min)
                               + 'beta_min = %.3f   N_temp = %d   '%(1./temp_max,N_temp)
                               + 'x_max = %.3f   nx = %d   N_iter = %d'%(x_max,nx, N_iter)]

        Z_data_headers = ['beta', 'temperature', 'Z']
        Z_data = save_csv(Z_data.transpose(), Z_data_headers, None, Z_file_name, relevant_info_Z,
                          print_data=False)

```

```
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 
404     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 exluyentes. 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\nnaive 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 alcoritmo (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).
```

```
        nx_sampling: int        ->  se generan nx mediante range(nx_max,nx_min,-1*nx_sampling).
        N_iter_min: int
        N_iter_max: int         ->  se relaciona con beta_ini = beta_fin **(-N_iter). Se gereran
                                    valores de N_iter con range(N_iter_max,N_iter_min-1,-1).
        save_opt_data: bool     ->  decide si guarda datos de optimización en archivo CSV.
        opt_data_file_name: str ->  nombre de archivo para datos de optimización.
        plot: bool              ->  decide si grafica optimización.
        show_plot: bool         ->  decide si muestra optimización.
        save_plot: bool         ->  decide si guarda optimización.
        opt_plot_file_name: str ->  nombre de gráfico de optimización. Si valor es None, se
                                    guarda con nombre conveniente según parámetros relevantes.

    Devuelve:
        error: list, shape=(nb,ndx) ->  valores de calc_error para diferentes valores de dx y
                                        beta_ini. dx incrementa de izquierda a derecha en lista
                                        y beta_ini incrementa de arriba a abajo.
        dx_grid: list, shape=(ndx,)         -> valores de dx para los que se calcula error.
        beta-ini_grid: list, shape=(nb,)    -> valores de beta_ini para los que calcula error.
    """

    t_0 = time()

    # Decide si genera o lee datos.
    if generate_opt_data:
        N_iter_min = int(N_iter_min)
        N_iter_max = int(N_iter_max)
        nx_min = int(nx_min)
        nx_max = int(nx_max)

        if nx_min%2==1:
            nx_min -= 1
        if nx_max%2==0:
            nx_max += 1

        # Crea valores de nx y N_iter (equivalente a generar valores de dx y beta_ini)
        nx_values = range(nx_max,nx_min,-1*nx_sampling)
        N_iter_values = range(N_iter_max,N_iter_min-1,-1)

        dx_grid = [2*x_max/(nx-1) for nx in nx_values]
        beta_ini_grid = [beta_fin * 2**(-N_iter) for N_iter in N_iter_values]

        error = []

        # Calcula error para cada valor de nx y N_iter especificado
        # (equivalentemente dx y beta_ini).
        for N_iter in N_iter_values:
            row = []
            for nx in nx_values:
                rho,trace_rho,grid_x = run_pi_x_sq_trotter(x_max, nx, N_iter, beta_fin,
                                                           potential, potential_string,
                                                           False, False, None, None, False,
                                                           False, False)
                grid_x = np.array(grid_x)
                dx = grid_x[1]-grid_x[0]
                rho_normalized = np.copy(rho)/trace_rho
                pi_x = np.diag(rho_normalized)
                theoretical_pi_x = QHO_canonical_ensemble(grid_x,beta_fin)
                error_comp_theo = calc_error(pi_x,theoretical_pi_x,dx)
```

```
                row.append(error_comp_theo)
            error.append(row)

    elif read_opt_data:
        error =  pd.read_csv(opt_data_file_name, index_col=0, comment='#')
        dx_grid = error.columns.to_numpy()
        beta_ini_grid = error.index.to_numpy()
        error = error.to_numpy()

    else:
        raise Exception('Escoja si generar o leer datos en optimization(.)')

    # Toma valores de error  en cálculo de Z (nan e inf) y los remplaza por
    # el valor de mayor error en el gráfico.
    try:
        error = np.where(np.isinf(error),0,error)
        error = np.where(np.isnan(error),0,error)
        nan_value = 1.3*np.max(error)
        error = np.where(error==0, float('nan'), error)
    except:
        nan_value = 0
    error = np.nan_to_num(error, nan=nan_value, posinf=nan_value, neginf=nan_value)

    script_dir = os.path.dirname(os.path.abspath(__file__))

    # Guarda datos (solo si fueron generados y se escoje guardar)
    if generate_opt_data and save_opt_data:

        if opt_data_file_name is None:
            opt_data_file_name = ('pi_x-ms-opt-%s-beta_fin_%.3f'%(potential_string, beta_fin)
                                  + '-x_max_%.3f-nx_min_%d-nx_max_%d'%(x_max, nx_min, nx_max)
                                  + '-nx_sampling_%d-N_iter_min_%d'%(nx_sampling, N_iter_min)
                                  + '-N_iter_max_%d.csv'%(N_iter_max))

        opt_data_file_name = script_dir + '/' + opt_data_file_name
        if opt_relevant_info is None:
            opt_relevant_info = ['Optimization of parameters dx and beta_ini of matrix squaring'
                        + ' algorithm', '%s   beta_fin = %.3f    '%(potential_string, beta_fin)
                        + 'x_max = %.3f    nx_min = %d    nx_max = %d    '%(x_max, nx_min, nx_max)
                        + 'nx_sampling = %d N_iter_min = %d    '%(nx_sampling, N_iter_min)
                        + 'N_iter_max = %d'%(N_iter_max)]

        save_csv(error, dx_grid, beta_ini_grid, opt_data_file_name, opt_relevant_info)

    t_1 = time()

    # Grafica.
    if plot:

        fig, ax = plt.subplots(1, 1)

        DX, BETA_INI = np.meshgrid(dx_grid, beta_ini_grid)
        cp = plt.pcolormesh(DX,BETA_INI,error)
        plt.colorbar(cp)

        ax.set_ylabel(u'$\\beta_{ini}$')
        ax.set_xlabel('$dx$')
        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
637     return error, dx_grid, beta_ini_grid, comp_time
638
639
640 ##########################################################################################################
641 # PANEL DE CONTROL
642 #
643 # Decide si corre algoritmo matrix squaring
644 run_ms_algorithm = False
645 # Decide si corre algoritmo para cálculo de energía interna
646 run_avg_energy = False
647 # Decide si corre algoritmo para optimización de dx y beta_ini
648 run_optimization = True
649 #
650 #
651 ##########################################################################################################
652
653
654
655 ##########################################################################################################
656 # PARÁMETROS GENERALES PARA LAS FIGURAS
657 #
658 # Usar latex en texto de figuras y agrandar tamaño de fuente
659 plt.rc('text', usetex=True)
660 plt.rcParams.update({'font.size':15,'text.latex.unicode':True})
661 # Obtenemos path para guardar archivos en el mismo directorio donde se ubica el script
662 script_dir = os.path.dirname(os.path.abspath(__file__))
663 #
664 ##########################################################################################################
665
666
667
668 ##########################################################################################################
669 # CORRE ALGORITMO MATRIX SQUARING
670 #
671
672 # Parámetros físicos del algoritmo
673 x_max = 5.
```

```
nx = 201
N_iter = 7
beta_fin = 4
potential, potential_string =  harmonic_potential, 'harmonic_potential'

# Parámetros técnicos
print_steps = False
save_data = False
file_name = None
relevant_info = None
plot = True
save_plot = False
show_plot = True

if run_ms_algorithm:
    rho, trace_rho, grid_x = run_pi_x_sq_trotter(x_max, nx, N_iter, beta_fin, potential,
                                                 potential_string, print_steps, save_data,
                                                 file_name, relevant_info, plot,
                                                 save_plot, show_plot)

#
#
###############################################################################################################



###############################################################################################################
# CORRE ALGORITMO PARA CÁLCULO DE ENERGÍA INTERNA
#

# Parámetros técnicos función partición y cálculo de energía
read_Z_data = False
generate_Z_data = True
Z_file_name = None
plot_energy = True
save_plot_E = True
show_plot_E = True
E_plot_name = None

# Parámetros físicos para calcular Z y <E>
temp_min = 1./10
temp_max = 1./2
N_temp = 10
potential, potential_string = harmonic_potential, 'harmonic_potential'

# Más parámetros técnicos
save_Z_csv = True
relevant_info_Z = None
print_Z_data = False
x_max = 7.
nx = 201
N_iter = 7
print_steps = False
save_pi_x_data = False
pi_x_file_name = None
relevant_info_pi_x = None
plot_pi_x = False
save_plot_pi_x = False
```

```
show_plot_pi_x = False

if run_avg_energy:
    average_energy(read_Z_data, generate_Z_data, Z_file_name, plot_energy, save_plot_E,
                   show_plot_E, E_plot_name,
                   temp_min, temp_max, N_temp, save_Z_csv, relevant_info_Z, print_Z_data,
                   x_max, nx, N_iter, potential, potential_string, print_steps, save_pi_x_data,
                   pi_x_file_name, relevant_info_pi_x,plot_pi_x, save_plot_pi_x, show_plot_pi_x)

#
#
###########################################################################################################



###########################################################################################################
# CORRE ALGORITMO PARA OPTIMIZACIÓN DE DX Y BETA_INI
#

# Parámetros físicos
beta_fin = 4
x_max = 5
potential, potential_string =  harmonic_potential, 'harmonic_potential'
nx_min = 10
nx_max = 310
nx_sampling = 60
N_iter_min = 8
N_iter_max = 20

# Parámetros técnicos
generate_opt_data = True
read_opt_data = False
save_opt_data = True
opt_data_file_name = None
opt_relevant_info = None
plot_opt = True
show_opt_plot = True
save_plot_opt = True
opt_plot_file_name = None

if run_optimization:
    error, dx_grid, beta_ini_grid, comp_time = \
        optimization(generate_opt_data, read_opt_data, beta_fin, x_max, potential,
                     potential_string, nx_min, nx_max, nx_sampling, N_iter_min,
                     N_iter_max, save_opt_data, opt_data_file_name,opt_relevant_info,
                     plot_opt, show_opt_plot, save_plot_opt, opt_plot_file_name)
    print('----------------------------------------'
          + '----------------------------------------\n'
          + 'Optimization:  beta_fin=%.3f,   x_max=%.3f,   potential=%s\n \
              nx_min=%d, nx_max=%d,   N_iter_min=%d,   N_iter_max=%d\n \
              computation time = %.3f sec.\n'%(beta_fin,x_max,potential_string,nx_min,
                                               nx_max,N_iter_min,N_iter_max,comp_time)
          + '----------------------------------------'
          + '----------------------------------------')

#
#
###########################################################################################################
```

Apéndice B: Código 2: Naive Path Integral Montecarlo Sampling