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

Juan Esteban Aristizabal Zuluaga
Instituto de Física, Universidad de Antioquia.
(Dated: 14 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 gananó 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.

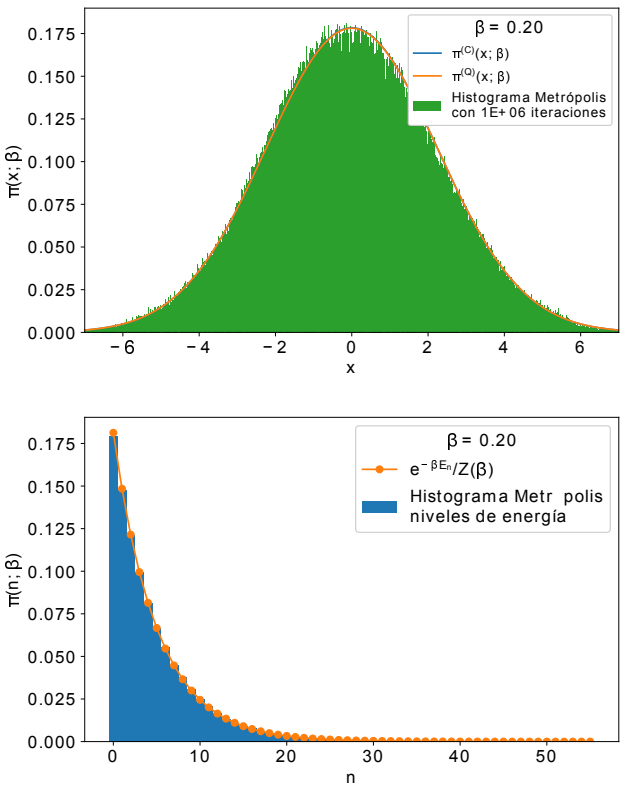

Figura 1. Arriba: densidad de probabilidad de encontrar a la «partícula» cuántica en una posición dada, cuando está en presencia de un potencial armónico y de baño térmico a temperatura definida por $\beta = 1/T = 0.2$. Mostramos los resultados teóricos clásico y cuántico como línea continua y el histograma 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).

[11] Encyclopedia, *Journal of Chemical Information and Modeling*, Vol. 53 (2019) pp. 1689–1699, arXiv:arXiv:1011.1669v3.

Apéndice A: Código 1: Matrix Squaring

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

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):
    """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.
    """
```

```
33      return (np.tanh(beta/2.)/np.pi)**0.5 * np.exp(- x**2 * np.tanh(beta/2.))
34
35  def rho_trotter(x_max = 5., nx = 101, beta=1, potential=harmonic_potential):
36      """
37      Uso:    devuelve matriz densidad en aproximación de Trotter para altas temperaturas
38              y bajo influencia del potencial "potential".
39
40      Recibe:
41          x_max: float    -> los valores de x estarán en el intervalo (-x_max,x_max).
42          nx: int         -> número de valores de x considerados (igualmente espaciados).
43          beta: float     -> inverso de temperatura en unidades reducidas.
44          potential: func -> potencial de interacción. Debe ser función de x.
45
46      Devuelve:
47          rho: numpy array, shape=(nx,nx)     ->  matriz densidad en aproximación de Trotter para
48                                                  altas temperaturas y  potencial dado.
49          grid_x: numpy array, shape=(nx,)    ->  valores de x en los que está evaluada rho.
50          dx: float                           ->  separación entre valores contiguos de grid_x
51      """
52      # Valor de la discretización de posiciones según x_max y nx dados como input
53      dx = 2. * x_max / (nx - 1)
54      # Lista de valores de x teniendo en cuenta discretización y x_max
55      grid_x = np.array([i*dx for i in range(-int((nx-1)/2), int(nx/2 + 1))])
56      # Construcción de matriz densidad dada por aproximación de Trotter
57      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])
58      return rho, grid_x, dx
59
60  def density_matrix_squaring(rho, grid_x, N_iter = 1, beta_ini = 1, print_steps=True):
61      """
62      Uso:    devuelve matriz densidad luego de aplicarle algoritmo matrix squaring N_iter veces.
63              En la primera iteración se usa matriz de densidad dada por el input rho (a
64              temperatura inversa beta_ini); en las siguientes iteraciones se usa matriz densidad
65              generada por la iteración inmediatamente anterior. El sistema asociado a la matriz
66              densidad obtenida (al final de aplicar el algoritmo) está a temperatura inversa
67              beta_fin = beta_ini * 2**(N_iter).
68
69      Recibe:
70          rho: numpy array, shape=(nx,nx)     ->  matriz densidad discretizada en valores dados
71                                                  por x_grid.
72          grid_x: numpy array, shape=(nx,)    ->  valores de x en los que está evaluada rho.
73          N_iter: int                         ->  número de iteraciones del algoritmo.
74          beta_ini: float                     ->  valor de inverso de temperatura asociado a la
75                                                  matriz densidad rho dada como input.
76          print_steps: bool                   ->  decide si muestra valores de beta en cada
77                                                  iteración.
78
79      Devuelve:
80          rho: numpy array, shape=(nx,nx)     ->  matriz densidad de estado rho a temperatura
81                                                  inversa igual a beta_fin.
82          trace_rho: float                    ->  traza de la matriz densidad a temperatura
    ↪  inversa
83                                                  igual a beta_fin. Por la definición que tomamos
84                                                  de rho, ésta es equivalente a la función
85                                                  partición a dicha temperatura.
86          beta_fin: float                     ->  temperatura inversa del sistema asociado a rho.
87      """
88      # Valor de discretización de las posiciones
```

```
    dx = grid_x[1] - grid_x[0]
    # Cálculo del valor de beta_fin según valores beta_ini y N_iter dados como input
    beta_fin = beta_ini * 2 **  N_iter
    # Imprime infromación relevante
    print('\nbeta_ini = %.3f'%beta_ini,
          '\n----------------------------------------------------------------')
    # Itera algoritmo matrix squaring
    for i in range(N_iter):
        rho = dx * np.dot(rho,rho)
        # Imprime información relevante
        if print_steps==True:
            print(u'Iteración %d)  2^%d * beta_ini --> 2^%d * beta_ini'%(i, i, i+1))
    # Calcula traza de rho
    trace_rho = np.trace(rho)*dx
    return rho, trace_rho, beta_fin

def save_pi_x_csv(grid_x, x_weights, file_name, relevant_info, print_data=True):
    """
    Uso: guarda datos de la distribución de probabilidad pi(x;beta) en un archivo .csv

    Recibe:
        grid_x: numpy array, shape=(nx,)    ->  valores de x en los que está evaluada
↪   pi(x;beta).
        x_weights: numpy array, shape=(nx,) ->  valores de pi(x;beta) para cada x en grid_x
        file_name: str                      ->  nombre del archivo en el que se guardarán datos.
        relevant_info: str                  ->  información que se agrega como comentario en
                                                primera línea.
        print_data: bool                    ->  decide si imprime datos guardados, en pantalla.

    Devuelve:
        pi_x_data: pd.DataFrame             ->  valores de pi(x;beta) para x en grid_x en
↪   formato
                                                "pandas".
    """
    # Almacena datos de probabilifad en diccionario: grid_x para posiciones y x_weights para
    # valores de densidad de probabilidad.
    pi_x_data = {'Position x': grid_x,
                 'Prob. density': x_weights}
    # Pasamos datos a formato DataFrame de pandas.
    pi_x_data = pd.DataFrame(data=pi_x_data)
    # Crea archivo .csv y agrega comentarios relevantes dados como input
    with open(file_name,mode='w') as rho_csv:
        rho_csv.write(relevant_info+'\n')
    rho_csv.close()
    # Usamos pandas para escribir en archivo en formato csv.
    with open(file_name,mode='a') as rho_csv:
        pi_x_data.to_csv(rho_csv)
    rho_csv.close()
    # Imprime en pantalla datos de posiciones y probabilidades.
    if print_data==True:
        print(pi_x_data)
    return pi_x_data

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, plot=True, save_plot=True, show_plot=True):
    """
    Uso:    corre algoritmo matrix squaring iterativamente (N_iter veces). En la primera
```

```
145             iteración se usa una matriz densidad en aproximación de Trotter a temperatura
146             inversa beta_ini = beta_fin * 2**(-N_iter) para potencial dado por potential;
147             en las siguientes iteraciones se usa matriz densidad generada por la iteración
148             inmediatamente anterior. Además ésta función guarda datos de pi(x;beta) vs. x
149             en archivo de texto y grafica pi(x;beta) comparándolo con teoría para el oscilador
150             armónico cuántico.
151
152         Recibe:
153             x_max: float        ->  los valores de x estarán en el intervalo (-x_max,x_max).
154             nx: int             ->  número de valores de x considerados.
155             N_iter: int         ->  número de iteraciones del algoritmo matrix squaring.
156             beta_ini: float     ->  valor de inverso de temperatura que queremos tener al final de
157                                     aplicar el algoritmo matrix squaring iterativamente.
158             potential: func     ->  potencial de interacción usado en aproximación de trotter. Debe
159                                     ser función de x.
160             potential_string: str   ->  nombre del potencial (con éste nombramos los archivos que
161                                         se generan).
162             print_steps: bool   ->  decide si imprime los pasos del algoritmo matrix squaring.
163             save_data: bool     ->  decide si guarda los datos en archivo .csv.
164             plot: bool          ->  decide si grafica.
165             save_plot: bool     ->  decide si guarda la figura.
166             show_plot: bool     ->  decide si muestra la figura en pantalla.
167
168         Devuelve:
169             rho: numpy array, shape=(nx,nx)     ->  matriz densidad de estado rho a temperatura
170                                                     inversa igual a beta_fin.
171             trace_rho: float                    ->  traza de la matriz densidad a temperatura
    ↪ inversa
172                                                     igual a beta_fin. Por la definición que tomamos
173                                                     de "rho", ésta es equivalente a la función
174                                                     partición en dicha temperatura.
175             grid_x: numpy array, shape=(nx,)    ->  valores de x en los que está evaluada rho.
176         """
177         # Cálculo del valor de beta_ini según valores beta_fin y N_iter dados como input
178         beta_ini = beta_fin * 2**(-N_iter)
179         # Cálculo de rho con aproximación de Trotter
180         rho, grid_x, dx = rho_trotter(x_max, nx, beta_ini, potential)
181         # Aproximación de rho con matrix squaring iterado N_iter veces.
182         rho, trace_rho, beta_fin_2 = density_matrix_squaring(rho, grid_x, N_iter,
183                                                              beta_ini, print_steps)
184         print('------------------------------------------------------------\n' + \
185               u'beta_fin = %.3f   Z(beta_fin) = Tr(rho(beta_fin)) = %.3E \n'%(beta_fin_2,trace_rho))
186         # Normalización de rho a 1 y cálculo de densidades de probabilidad para valores en grid_x.
187         rho_normalized = np.copy(rho)/trace_rho
188         x_weights = np.diag(rho_normalized)
189         # Guarda datos en archivo .csv.
190         if save_data==True:
191             # Nombre del archivo .csv en el que guardamos valores de pi(x;beta_fin).
192             file_name = u'pi_x-%s-x_max_%.3f-nx_%d-N_iter_%d-beta_fin_%.3f.csv'\
193                                             %(potential_string,x_max,nx,N_iter,beta_fin)
194             # Información relevante para agregar como comentario al archivo csv.
195             relevant_info = u'# %s   x_max = %.3f   nx = %d   '%(potential_string,x_max,nx) + \
196                             u'N_iter = %d   beta_ini = %.3f   '%(N_iter,beta_ini,) + \
197                             u'beta_fin = %.3f'%beta_fin
198             # Guardamos valores  de pi(x;beta_fin) en archivo csv.
199             pi_x_data = save_pi_x_csv(grid_x, x_weights, file_name, relevant_info, print_data=0)
200         # Gráfica y comparación con teoría
201         if plot == True:
```

```
202         plt.figure(figsize=(8,5))
203         plt.plot(grid_x, x_weights, label = 'Matrix squaring +\nfórmula de Trotter.\n$N=%d$
            ↪  iteraciones\n$dx=%.3E$'%(N_iter,dx))
204         plt.plot(grid_x, QHO_canonical_ensemble(grid_x,beta_fin), label=u'Valor teórico QHO')
205         plt.xlabel(u'x')
206         plt.ylabel(u'$\pi^{(Q)}(x;\\beta)$')
207         plt.legend(loc='best',title=u'$\\beta=%.2f$'%beta_fin)
208         plt.tight_layout()
209         if save_plot==True:
210             plot_name = u'pi_x-plot-%s-x_max_%.3f-nx_%d-N_iter_%d-beta_fin_%.3f.eps'\
211                                             %(potential_string,x_max,nx,N_iter,beta_fin)
212             plt.savefig(plot_name)
213         if show_plot==True:
214             plt.show()
215         plt.close()
216     return rho, trace_rho, grid_x
217
218 # Agranda letra en texto de figuras generadas
219 plt.rcParams.update({'font.size':15})
220 # Corre el algoritmo
221 rho, trace_rho, grid_x = run_pi_x_sq_trotter( potential = harmonic_potential,
222                                             potential_string =  'harmonic_potential',
223                                             save_data=True, save_plot=True, show_plot=True)
```

Apéndice B: Código 2: Naive Path Integral Montecarlo Sampling