TRABAJO DE FIN DE GRADO

FORMALISMO GEOMÉTRICO DE LA MECÁNICA CUÁNTICA Y SUS APLICACIONES: ANEXOS

Autor:
Carlos Bouthelier Madre

Director:
Dr. Jesús Clemente-Gallardo

UNIVERSIDAD DE ZARAGOZA

28 Junio 2017

Índice general

Capítulo 1

La dinámica como curvas integrales del campo vectorial Hamiltoniano

Consideremos la función asociada al operador Hamiltoniano

$$f_H(\phi) = \tfrac{1}{2}\langle \phi \mid H\phi \rangle.$$

Consideremos ahora el campo vectorial definido mediante

$$X_H = \hbar^{-1}\Omega(df_H, \cdot).$$

Este campo vectorial sobre la variedad M_Q codifica la ecuación de Schrödinger en lenguaje geométrico, como veremos a continuación. Nótese que estamos trabajando sobre M_Q, no sobre $\mathcal{P}$.

Consideremos $\mathcal{H}$ como $\mathbb{C}^n$ y un Hamiltoniano $H : \mathbb{C}^n \to \mathbb{C}^n$ que se suele expresar en forma matricial como H^i_j donde i y j se corresponden con los índices de vectores coordenadas complejos $\{\Psi_1, ...\Psi_n\}$. Si lo consideramos como una matriz HR sobre M_Q, sus elementos vendrán dados por HR^μ_ν, donde μ y ν se corresponden con los índices de vectores de coordenadas reales $\{q^1, ..., q^n, p_1, ..., p_n\}$. La hermiticidad del Hamiltoniano se traduce en:

$$HR^{q^k}_{q^k} = H^k_k = HR^{p_k}_{p_k} \qquad HR^{q^k}_{p_k} = HR^{p_k}_{q^k} = 0$$
$$HR^{q^j}_{q^k} = HR^{p_j}_{p_k} = Re(H^j_k) \qquad HR^{q^j}_{p_k} = -HR^{p_j}_{q^k} = -Im(H^j_k)$$

Recordando la definición de f_H, donde H se sustitue por HR y $\psi \in M_Q$, podemos ver que las curvas integrales del campo vectorial hamiltoniano, que es

$$X_H = \hbar^{-1} \sum_k \left(\frac{\partial f_H}{\partial p_k} \frac{\partial}{\partial q^k} - \frac{\partial f_H}{\partial q^k} \frac{\partial}{\partial p_k} \right),$$

son precisamente la expresión de la ecuación de Schrödinger cuando lo escribimos de nuevo en notación compleja, de modo que vemos que: $\dfrac{d}{dt}x^\mu = -\hbar J^\mu_\nu HR^\nu_\rho x^\rho$, que es, escrita en notación matricial con la notación de Einstein, la siguiente ecuación diferencial

$$\dot{\psi}(q,p) = -\hbar J HR \psi(q,p),$$

donde J es la estructura compleja, que cumple, en su representación coordenada, $J^{\mu}_{\nu} = 0$ excepto $J^{q^n}_{p_n} = -1$ y $J^{p_n}_{q^n} = 1$. Donde además se ha identificado el elemento de M_Q con su vector de coordenadas, $\psi = x^{\mu}$.

Por lo tanto, hemos descrito la dinámica como las curvas integrales del campo vectorial Hamiltoniano $X_H = \hbar^{-1}\{H, \cdot\}$.

Capítulo 2

El espacio de proyectores y su relación con el espacio proyectivo

La acción unitaria $U(\mathcal{H})$ sobre $\mathcal{H}$ induce una acción simpléctica sobre la variedad simpléctica (M_Q, ω). F actuando sobre un par estado-operador da lugar a la medida de i veces dicho operador antihermítico.

$$F : M_Q \times u(\mathcal{H}) \to \mathbb{R} \qquad (\phi, A) \mapsto \langle \psi \mid iA\psi \rangle = f_{iA}(\phi)$$

Si fijamos A da la funcion f_{iA} que puede actuar sobre un estado ψ reproduciendo el resultado anterior:

$$F_A := f_{iA} : M_Q \mapsto \mathbb{R}$$

que cumplen $\{F(A), F(B)\} = iF([A, B])$. Por lo que hemos definido una aplicación $\mu^* : \hat{A} = f_A$, donde $\hat{A} = iA \in u(\mathcal{H})$.

Análogamente, si fijamos ψ, tenemos $F(\psi) : u(\mathcal{H}) \to \mathbb{R}$, de modo que estos elementos $F(\psi) \in u^*(\mathcal{H})$. De este modo hay una aplicación que identifica cada elemento ψ de M_Q con un elemento $F(\psi) \in u^*(\mathcal{H})$, que constituye una aplicación momento

$$\mu : \mathcal{H} \to u^*(\mathcal{H}) \qquad \mu(\psi) = \mid \psi\rangle\langle\psi \mid$$

Análogamente podemos considerar la acción proyectada, sustituyendo M_Q por $\mathcal{P}$ y $f_{iA} \in F : (\mathcal{H})$ por $e_{iA} \in \mathcal{O}(\mathcal{P})$, y obviamente $\phi \in M_Q$ por $[\phi] \in \mathcal{P}$. De este modo, $F_{\mathcal{P}}$ pertenece a $u^*(\mathcal{H})$, de modo que al actuar sobre $u(\mathcal{H})$ envía sus elementos al cuerpo $\mathbb{R}$.

En este caso, la aplicación $\mu_{\mathcal{P}}([\phi]) = \dfrac{\mid \phi\rangle\langle\phi \mid}{\langle\phi \mid \phi\rangle}$, que identificamos con lo que en cuántica llamamos un proyector, por lo que hemos probado que los elementos del espacio proyectivo $\mathcal{P}$ se encuentran en correspondencia uno-a-uno con el subconjunto de elementos de $u^*(\mathcal{H})$ que son proyectores de rango uno, es decir, el subconjunto de elementos $\{\rho_k\}$ que satisface $\rho_k^2 = \rho_k \qquad Tr\rho_k = 1$.

Este subconjunto es el espacio de estados puros físicos, por dicha correspondencia uno-a-uno. Sin embargo, sabemos que físicamente, al construir una estadística, podemos encontrar estados mezcla, que por tanto $\rho^2 \neq \rho$. Una combinación lineal convexa de proyectores de rango uno, i.e. estados puros, dará lugar generalmente a un estado mezcla.

Capítulo 3

Expresión coordenada de g y ω

Si analizamos brevemente el producto hermítico, vemos algunas de las propiedades estructurales fundamentales que tendremos que traducir a objetos geométricos:

$$\langle \phi \mid \psi \rangle = \sum_{k=1}^{n} (\phi_k^R - i\phi_k^I)(\psi_k^R + i\psi_k^I) = \sum_{k=1}^{n} \left(\phi_k^{R^2} \psi_k^{R^2} + \phi_k^{I^2} \psi_k^{I^2} \right) + i \sum_{k=1}^{n} \left(\phi_k^R \psi_k^I - \psi_k^R \phi_k^I \right)$$

Por lo tanto, si ahora consideramos que $\mid \psi\rangle, \mid \phi\rangle \in M_Q$ existen unos tensores sobre M_Q llamados g y ω que reproducen la estructura anterior de modo que:

$$\omega_\varphi : M_Q \times M_Q \to \mathbb{R} \qquad g_\varphi : M_Q \times M_Q \to \mathbb{R}$$
$$\langle \phi \mid \psi \rangle = g_\varphi(\mid \phi\rangle, \mid \psi\rangle) + i\omega_\varphi(\mid \phi\rangle, \mid \psi\rangle)$$

De este modo identificamos que actúan localmente del siguiente modo:

$$g_\varphi(\mid \phi\rangle, \mid \psi\rangle) = g_\varphi(X_\phi(\varphi), X_\psi(\varphi)) = \sum_{k=1}^{n} \left(\phi_k^{R^2} \psi_k^{R^2} + \phi_k^{I^2} \psi_k^{I^2} \right),$$
$$\omega_\varphi(\mid \phi\rangle, \mid \psi\rangle) = \omega_\varphi(X_\phi(\varphi), X_\psi(\varphi)) = \sum_{k=1}^{n} \left(\phi_k^R \psi_k^I - \psi_k^R \phi_k^I \right)$$

Análogamente $J_\varphi(\mid \phi\rangle) = g_\varphi(X_\phi(\varphi)) =\mid i\phi\rangle$, donde con $\mid i\phi\rangle$ denotamos que se ha producido el siguiente cambio en coordenadas: $J_\varphi : \phi(q^k, p_k) \mapsto \phi(-p_k, q^k)$.J es pues un campo tensorial, que localmente actúa como un tensor (1,1).

Estos campos tensoriales son parte de la estructura de Kähler presente en M_Q, que en el cuerpo principal del trabajo se construyen sin necesidad de una base. Se ve así que los campos tensoriales actúan del siguiente modo:

$$g = \sum_{k=1}^{n} dq_k \otimes dq_k + dp_k \otimes dp_k \qquad \omega = \sum_{k=1}^{n} dq_k \wedge dp_k \qquad J = \sum_{k=1}^{n} dq_k \otimes \frac{\partial}{\partial p_k} - dp_k \otimes \frac{\partial}{\partial q_k}$$

Como G y Ω son los vectores contravariantes asociados a g y ω, respectivamente, y estos son constantes, podemos escribir que:

$$\Omega = \sum_{k=1}^{n} \left(\frac{\partial}{\partial q^k} \wedge \frac{\partial}{\partial p_k} \right) \qquad G = \sum_{k=1}^{n} \left(\frac{\partial}{\partial q^k} \otimes \partial q^k + \frac{\partial}{\partial p_k} \otimes \partial p_k \right)$$

Capítulo 4

Aplicación del Teorema de Darboux

Teorema 1 *Para cada punto de una variedad simpléctica existe una carta local* $\phi : U_P \to \mathbb{R}^{2n}$ *tal que si* ω *es el pullback de la forma simpléctica canónica* ω_0 *en* $\mathbb{R}^{2n}$ *entonces* $\omega = \phi^* \omega_0$.

La carta local U_P es la carta local de Darboux alrededor de P. La variedad simpléctica (M,ω) puede ser descrita mediante un atlas formado por cartas de Darboux. En el caso de la mecánica Hamiltoniana llamamos coordenadas de Darboux a las coordenadas canónicas q^k y p_k, que nos permiten definir del modo que indica el teorema la forma simpléctica. La aplicación ϕ, en el caso de la mecánica cuántica geométrica ϕ_{M_Q}, sería la identificación del par parte real-parte imaginaria, de las coordenadas en una base del espacio de Hilbert complejo.

En el caso particular de M_Q, como cualquier variedad lineal, un solo mapa asociado a unas coordenadas de Darboux en concreto conforman un atlas que recubre todo el espacio, que es el dado por las coordenadas q^k, p_k escogidas mediante ϕ_{M_Q}. Es por esto por lo que afirmamos que una sola base en $\mathcal{H}$ nos define un conjunto de coordenadas que sirve para toda la variedad y que por tanto mantenemos a lo largo de todo el trabajo.

Capítulo 5

Breve repaso de formulación geométrica de la Mecánica Clásica

Sea M_C el espacio de fases del sistema, que se trata de una variedad simpléctica. Los estados vienen determinados por un conjunto de posiciones $\{\vec{q}_k\}$ y su momentos asociados $\{\vec{p}_k\}$, de modo que podemos considerar, sin introducir restricciones al espacio de fases, $M_C \sim \mathbb{R}^{2n}$, donde n son los grados de libertad (por ejemplo, en 3D, $n = 3 \cdot N_{part}$).

Un observable viene dado por una función $f \in C^\infty(M_C)$ tal que $f : M_C \to \mathbb{R}$, que a cada estado le asignan un determinado valor, reflejando así la medida. Sobre el espacio de observables $C^\infty(M_C)$ se puede introducir el corchete de Poisson $\{\cdot,\cdot\}$ como una operación interna bilineal

$$\{\cdot,\cdot\} : C^\infty(M_C) \times C^\infty(M_C) \to C^\infty(M_C)$$

que cumple las siguientes condiciones:

- Antisimetría: $\{f,g\} = -\{g,f\} \qquad \forall f,g \in C^\infty(M_C)$
- Satisfacción de la identidad de Jacobi. $\{f,\{g,h\}\} + \{h,\{f,g\}\} + \{g,\{h,f\}\} = 0 \qquad \forall f,g,h \in C^\infty(M_C)$
- Satisfacción de la regla de Leibniz $\{f,gh\} = \{f,g\}h + g\{f,h\} \qquad \forall f,g,h \in C^\infty(M_C)$

De este modo, el corchete de Poisson introduce una estructura de Poisson y el espacio de observables $(C^\infty(M_C), \{\cdot,\cdot\})$ forma un álgebra de Poisson.

Si los pares posición momento (q^k, p_k) son coordenadas de Darboux (ver capítulo anterior de este anexo), podemos construir el corchete de Poisson a partir de la forma simpléctica canónica en $\mathbb{R}^{2n}$, de modo que:

$$\{f,g\} = \sum_{k=1}^{n} \frac{\partial f}{\partial p_k}\frac{\partial g}{\partial q^k} - \frac{\partial g}{\partial p_k}\frac{\partial f}{\partial q^k}$$

Teniendo este corchete de Poisson y una función $f \in C^\infty(M_Q)$ definimos el campo vectorial hamiltoniano de la función f como $X_f = \{f,\cdot\}$ que en coordenadas canónicas será $X_f = \sum_{k=1}^{n} \frac{\partial f}{\partial p_k}\frac{\partial}{\partial q^k} - \frac{\partial f}{\partial q^k}\frac{\partial}{\partial p_k}$. De este modo, el sistema dinámico hamiltoniano será el espacio de fases M_C, el corchete

de poisson $\{\cdot,\cdot\}$ y una función $H \in C^\infty(M_C)$, que se denomina Hamiltoniano y para la cual las curvas integrales de su campo vectorial hamiltoniano X_H reproducen la dinámica del sistema.

La dinámica de cualquier observable $\forall f \in C^\infty(M_C)$ vendrá dada entonces por:

$$\frac{df}{dt} = \{H, f\}$$

Al considerar como observables las coordenadas canónicas posición q^i y momento p^i y la expresión en dichas coordenadas del corchete de Poisson, se ve que efectivamente se obtienen las ecuaciones de Hamilton que describen la dinámica:

$$\dot{q}^i = \frac{\partial H}{\partial p_i} \qquad \dot{p}_i = -\frac{\partial H}{\partial q^i}$$

Capítulo 6

Código sympy para obtener las funciones termodinámicas

```
import sympy as sym #Importanto importarlos así y no from sympy import *,
#pues la incompatibilidad de python simbólico y numérico da problemas
import numpy as num#Importanto importarlos así y no from numpy import *,
#pues la incompatibilidad de python simbólico y numérico da problemas
from sympy import cos
from sympy.tensor.array import Array
from sympy import I, Matrix, symbols,var
from sympy.physics.quantum import TensorProduct
from sympy import init_printing
from sympy import init_session
from sympy import plot

from IPython.display import display, Math, Latex


var('Tita1 Tita2 Tita3 Tita4 Tita5 En0 En1 En2 En3 E4 E5 E6 E7  Beta eps Zt', real=True)
def delta(n):#Define los niveles de energía en concreto del hamiltoniano escogido
    f='Tita'+str(n)
    return sqrt(cos(f)*cos(f)*eps*eps+1)

def EF(i):#Devuele el H diagonalizado para una sola partícula y en función del gap de energía
#(mínimo autovalor llevado a 0 como referencia de energías)
    return Matrix([[0,0],[0,2*delta(i)]])

ID=Matrix([[1,0],[0,1]])

def ExtDim(X,n):#Extiende un tensor X a n dimensiones, multiplicando tensorialmente por n-1 veces la identidad
    if n==1:
        return X
    else:
        return TensorProduct(ExtDim(X,n-1),ID)

def n_energy(n):#Construye H  para n partículas a partir del de una partículas, pues son no interactuantes
    if n==1:
        return EF(n)
    else:
        Etemp=TensorProduct(ID,n_energy(n-1))
        Etemp+=ExtDim(EF(n),n)
        return Etemp
```

```
def AsignaDiag(E,n):#asigna los autovalores del hamiltoniano a un vector
    for i in range(2**n):
        Evec[i]=simplify(E[i,i]-E[2**n-1,2**n-1]/2)

def Z_Q(n): #Devuelve la función de partición clásica para cualquier hamiltoniano con niveles de energía no degenerados
#guardados en Evec[k], para cualquier n
    Zq=0

    for i in range(2**n):
        prod=1
        for j in range(2**n):
            if j!=i:
                prod=prod*((Evec[j]-Evec[i]))
        Zq+=exp(-Beta*Evec[i])/prod
        #F.write("Elemento"+str(i)+"="+str(exp(-Beta*Evec[i])/prod)+"\n\n")
    return Zq*Beta#*(2*3.14159265358979323846/Beta)**(2**n) #factor ausente debido a que dividir ro(ji) entre Z se cancelan,
    #también ausente en la función para ro(ji)
def


def Rho_Q(n): #Esta funcion devuelve ro(xi) según los niveles de energía E[j],
# para cualquier N y cualquier hamiltoniano diagonalizado
    x=[0]*(2**n)
    for i in range (2**n):#eLEMENTO DE MATRIZ
        for k in range (2**n):
            Prod=1
            for j in range(2**n):
                if (j!=k):
                    Prod=Prod*simplify(Evec[j]-Evec[k])
            if (k!=i):
                x[i]+=exp(-Beta*Evec[k])/(simplify(Evec[i]-Evec[k])*Prod)
            if (k==i):
                Sumprodi=0
                for l in range(2**n):
                     if (l!=i):
                         Prodi=1
                         for j in range(2**n):
                            if ((j!=l)and(j!=i)):
                                Prodi=Prodi*simplify(Evec[j]-Evec[i])
                         Sumprodi+=Prodi

                x[i]+=(Beta/Prod-Sumprodi/(Prod*Prod))*exp(-Beta*Evec[i])

        x[i]*=(1/Zt)#*(2*3.14159265358979323846/Beta)**(2**n)
        #factor ausente debido a que dividir entre Z lo elimina, también ausente en la función para Z

    return x

def S_VN(n,x): # para calcular densidad de entropia de von-Neuman
    S=0
    for i in range (2**n):
        S+=x[i]*log(x[i])
    return -S
def Traza(n,x):
    t=0
    for i in range (2**n):
        t+=x[i]
    return t
```

```
def IntegraFuncionPos(F,n,Intervalos):#Para integrar numéricamente funciones F con n variables clásicas
# sobre el espacio de variables clásicas Tita_j, para funciones definidas positivas
    Delta=3.14159265358979323846/Intervalos/2
    #Integra en los Tita_i de 0 a pi/2 dividiendo en Intervalos
    Suma=0
    if n==1:
        for i in range(Intervalos):
            fs=F.subs("Tita1",Delta/2+i*Delta)
            if(fs>0):
                Suma+=fs
            else:
                print(fs,"ERROR, sumando negativo")
        return 4*Suma*Delta
    if n==2: #Hay que evitar un entorno de los puntos degenerados que tienen medida 0
        for i in range(1,Intervalos):
                f=F.subs("Tita1",Delta/2+i*Delta)
                for  j in range(i):
                    if (abs(cos(i*Delta+Delta/2)**2-(cos(j*Delta+Delta/2)**2))>0.0004):
                        fs=f.subs("Tita2",Delta/2+j*Delta)
                        if (fs>0):#AQUI LO DE DEFINIDO POSITIVO
                           Suma+=fs
                        else:
                            print(fs,"ERROR, sumando negativo")

        return 2*16*Suma*Delta*Delta
    if n==3: #Hay que evitar un entorno de los puntos degenerados que tienen medida 0
        for i in range(2,Intervalos):
                f=F.subs("Tita1",Delta/2+i*Delta)
                for j in range(1,i):
                    if (abs(cos(i*Delta+Delta/2)**2-(cos(j*Delta+Delta/2)**2))>0.0004):#Tita1 evita a Tita2
                        f2=f.subs("Tita2",Delta/2+j*Delta)
                        for k in range(j):
                            if (abs(cos(i*Delta+Delta/2)**2-(cos(k*Delta+Delta/2)**2))>0.0004):#Tita1 evita a Tita3
                                if (abs(cos(k*Delta+Delta/2)**2-(cos(j*Delta+Delta/2)**2))>0.0004):#Tita2 evita a Tita3
                                    fs=f2.subs("Tita3",Delta/2+k*Delta)

                                    if (fs>0):#AQUI LO DE DEFINIDO POSITIVO
                                       Suma+=fs
                                    else:
                                        print(fs,"ERROR, sumando negativo")

        return 6*64*Suma*Delta*Delta*Delta
    if n==4: #Hay que evitar un entorno de los puntos degenerados que tienen medida 0
        for i in range(3,Intervalos):
                f=F.subs("Tita1",Delta/2+i*Delta)
                for j in range(2,i):
                    if (abs(cos(i*Delta+Delta/2)**2-(cos(j*Delta+Delta/2)**2))>0.0004):#Tita1 evita a Tita2
                        f2=f.subs("Tita2",Delta/2+j*Delta)
                        for k in range(1,j):
                            if (abs(cos(i*Delta+Delta/2)**2-(cos(k*Delta+Delta/2)**2))>0.0004)and
                               (abs(cos(k*Delta+Delta/2)**2-(cos(j*Delta+Delta/2)**2))>0.0004):
                               #Tita3 evita aTita 1 y Tita 2
                                    f3=f2.subs("Tita3",Delta/2+k*Delta)
                                    for l in range(k):
                                        if (abs(cos(i*Delta+Delta/2)**2-(cos(l*Delta+Delta/2)**2))>0.0004)
                                        and(abs(cos(l*Delta+Delta/2)**2-(cos(j*Delta+Delta/2)**2))>0.0004)
                                        and(abs(cos(k*Delta+Delta/2)**2-(cos(l*Delta+Delta/2)**2))>0.0004):
                                            #todos se evitan
```

```
                                        fs=f3.subs("Tita4",Delta/2+l*Delta)
                                        if (fs>0):#AQUI LO DE DEFINIDO POSITIVO
                                           Suma+=fs
                                        else:
                                            print(fs,"ERROR, sumando negativo")

        return 24*256*Suma*Delta*Delta*Delta*Delta
def IntegraFuncion(F,n,Intervalos):#Para integrar numéricamente funciones F con n variables clásicas
#sobre el espacio de variables clásicas Tita_j
    Delta=3.14159265358979323846/Intervalos/2
    #Integra en los Tita_i de 0 a pi/2 dividiendo en Intervalos
    Suma=0
    if n==1:
        for i in range(Intervalos):
            fs=F.subs("Tita1",Delta/2+i*Delta)
            Suma+=fs
        return 4*Suma*Delta
    if n==2: #Hay que evitar un entorno de los puntos degenerados que tienen medida 0
        for i in range(1,Intervalos):
                f=F.subs("Tita1",Delta/2+i*Delta)
                for  j in range(i):
                    if (abs(cos(i*Delta+Delta/2)**2-(cos(j*Delta+Delta/2)**2))>0.0004):
                        fs=f.subs("Tita2",Delta/2+j*Delta)
                        Suma+=fs

        return 2*16*Suma*Delta*Delta
    if n==3: #Hay que evitar un entorno de los puntos degenerados que tienen medida 0
        for i in range(2,Intervalos):
                f=F.subs("Tita1",Delta/2+i*Delta)
                for j in range(1,i):
                    if (abs(cos(i*Delta+Delta/2)**2-(cos(j*Delta+Delta/2)**2))>0.0004):

                        f2=f.subs("Tita2",Delta/2+j*Delta)
                        for k in range(j):
                            if (abs(cos(i*Delta+Delta/2)**2-(cos(k*Delta+Delta/2)**2))>0.0004):
                                if (abs(cos(k*Delta+Delta/2)**2-(cos(j*Delta+Delta/2)**2))>0.0004):
                                    fs=f2.subs("Tita3",Delta/2+k*Delta)
                                    Suma+=fs

        return 6*64*Suma*Delta*Delta*Delta
    if n==4: #Hay que evitar un entorno de los puntos degenerados que tienen medida 0
        for i in range(3,Intervalos):
                f=F.subs("Tita1",Delta/2+i*Delta)
                for j in range(2,i):
                    if (abs(cos(i*Delta+Delta/2)**2-(cos(j*Delta+Delta/2)**2))>0.0004):#Tita1 evita a Tita2
                        f2=f.subs("Tita2",Delta/2+j*Delta)
                        for k in range(1,j):
                            if (abs(cos(i*Delta+Delta/2)**2-(cos(k*Delta+Delta/2)**2))>0.0004)and
                               (abs(cos(k*Delta+Delta/2)**2-(cos(j*Delta+Delta/2)**2))>0.0004):
                                   #Tita3 evita aTita 1 y Tita 2
                                    f3=f2.subs("Tita3",Delta/2+k*Delta)
                                    for l in range(k):
                                        if (abs(cos(i*Delta+Delta/2)**2-(cos(l*Delta+Delta/2)**2))>0.0004)and
                                           (abs(cos(l*Delta+Delta/2)**2-(cos(j*Delta+Delta/2)**2))>0.0004)and
                                           (abs(cos(k*Delta+Delta/2)**2-(cos(l*Delta+Delta/2)**2))>0.0004):
                                            fs=f3.subs("Tita4",Delta/2+l*Delta)
                                            Suma+=fs
        return 24*256*Suma*Delta*Delta*Delta*Delta
```

```
def divideTraza(n,ro):#devuelve una ro(xi) dividida por tr(ro(xi)) que en nuestro caso reproduce la probabilidad cuántica condic
    rox=[None]*2**n
    Norma=Traza(n,ro)
    for i in range(2**n):
        rox[i]=ro[i]/Norma
    return rox
def Energia(n,ro): #Densidad de energía sobre el espacio clásico, a partir de tr(H(xi) ro(xi))
    E=0
    for i in range(2**n):
        E+=Evec[i]*ro[i]
    return E

init_session()


for N in range(1,4):#Este bucle calcula Z, S y U para distintas T
    E=n_energy(N)
    Evec=[None]*(2**N)
    AsignaDiag(E,N)
    ro=[None]*2**N
    #roEnt=[None]*2**N
    ro=Rho_Q(N)
    Q=Z_Q(N)
    Q=Q.subs(eps,1/sqrt(2))
    Stot=S_VN(N,ro)
    Sentretraza=S_VN(N,divideTraza(N,ro))*Traza(N,ro)
    U=Energia(N,ro)
    Stot=Stot.subs(eps,1/sqrt(2))
    Sentretraza=Sentretraza.subs(eps,1/sqrt(2))
    U=U.subs(eps,1/sqrt(2))

    print("Allá vamos\n")
    F=open("EntropyEnergyResN"+str(N)+".txt","w")
    for B in range(1,100):#Bucle en temperaturas
        Bet=1/(B*0.01+0.005)
        faux=Stot.subs(Beta,Bet)
        faux1=U.subs(Beta,Bet)
        faux2=Sentretraza.subs(Beta,Bet)
        Z=IntegraFuncionPos(Q.subs(Beta,Bet),N,50)

        S=IntegraFuncionPos(faux.subs(Zt,Z),N,50)

        Squantprom=IntegraFuncionPos(faux2.subs(Zt,Z),N,50)

        En=IntegraFuncion(faux1.subs(Zt,Z),N,50)

        F.write(str(B*0.01+0.005)+"\t"+str(Z)+"\t"+str(S)+"\t"+str(Squantprom)+"\t"+str(En)+"\n")
        #Squant prom es la entropia promedio asociada  ala distribución cuántica tradicional,
        #no se usa en el trabajo, es para otros estudios
    F.close()
```

Para calcular el entanglement las definiciones de funciones se mantienen, cambia el cuerpo principal:

```
for N in range(2,3):
    E=n_energy(N)
    Evec=[None]*(2**N)
    AsignaDiag(E,N)
    r=[None]*2**N
    ro=[None]*2**N
```

```
roEnt=[None]*2**N
r=Rho_Q(N)
t=Traza(N,r)
ro=divideTraza(N,r)

ro1=[ro[0]+ro[2],ro[1]+ro[3]]
ro2=[ro[0]+ro[1],ro[2]+ro[3]]
Q=Z_Q(N)
roent=[(ro[0]-(ro[0]+ro[2])*(ro[0]+ro[1])),(ro[1]-(ro[0]+ro[2])*(ro[2]+ro[3])),
       (ro[2]-(ro[1]+ro[3])*(ro[0]+ro[1])),(ro[3]-(ro[1]+ro[3])*(ro[2]+ro[3]))]
TrEnt=Traza2(N,roent)
Q=Q.subs(eps,1/sqrt(2))
TrEnt=TrEnt.subs(eps,1/sqrt(2))
TrEnt=TrEnt.subs(Zt,1)*t.subs(eps,1/sqrt(2))
Purity=Traza2(N,ro)-Traza2(1,ro1)*Traza2(1,ro2)#Traza2 es traza al cuadrado
Purity=Purity.subs(eps,1/sqrt(2))
Purity=Purity.subs(Zt,1)*t.subs(eps,1/sqrt(2))
#Hay dos medidas de entrelazamiento, Pure o Purity es la usada en el trabajo. La otra es para otras cosas.
print("Allá vamos\n")
F=open("EntanglementN"+str(N)+".txt","w")
for B in range(1,10):
    Bet=1/(B*0.01+0.005)
    f=TrEnt.subs(Beta,Bet)
    f1=Purity.subs(Beta,Bet)
    Z=IntegraFuncionPos(Q.subs(Beta,Bet),N,100)
    Ent=IntegraFuncionPos(f.subs(Zt,Z),N,100)
    Pure=IntegraFuncion(f1.subs(Zt,Z),N,100)
    F.write(str(B*0.01+0.005)+"\t"+str(Z)+"\t"+str(Ent)+"\t"+str(Pure)+"\n")
```