Universidad de Valladolid

FACULTAD DE CIENCIAS

TRABAJO FIN DE GRADO

Grado en Física

# ALGORITMOS CUÁNTICOS VARIACIONALES: UN PRIMER PASO EN LA COMPUTACIÓN CUÁNTICA ADIABÁTICA

Autor: Miguel Santos Pascual
Tutores: Fernando J. Gómez-Ruiz
y Luis M. Nieto Calzada
Año 2024

# Índice general

# RESUMEN

Con el auge de la computación cuántica aparece la necesidad de desarrollar nuevos algoritmos para poder trabajar con esta nueva tecnología. De entre la gran variedad de ellos, este documento se centra en estudiar y explicar los Algoritmos de Optimización Cuántica Aproximada conocidos en inglés como *Quantum Approximate Optimization Algorithm* (QAOA). Estos son un subtipo que forma parte de la Computación Cuántica Adiabática, *Variational Quantum Algorithms* (VQA) en inglés, y su objetivo es la optimización de funciones dicotómicas basándose en la lenta evolución temporal de un sistema cuántico hasta el punto de equilibrio buscado.

Para analizar su funcionamiento, primero se lleva a cabo una aproximación teórica de las bases e ideas sobre las que se sustentan estos tipos de algoritmos. Posteriormente, y con la ayuda de un ordenador convencional, se han realizado diferentes simulaciones de complejidad creciente. De esta forma, ha sido posible extraer conclusiones sobre los resultados que estos algoritmos nos proporcionan y sus ventajas e inconvenientes a la hora de ser aplicados a problemas reales.

# ABSTRACT

The current emergence of quantum computing leads to a need to develop new algorithms to work with this rising technology. Among the wide variety of them, this document focuses on studying and explaining the Quantum Approximate Optimization Algorithm (QAOA), a subtype which is part of the Variational Quantum Algorithms (VQA) and whose goal is the optimization of dichotomous functions based on the slow temporal evolution of a quantum system to the desired equilibrium state.

In order to analyze its performance, a theoretical approximation of the foundations on which these types of algorithms are based is firstly carried out. Subsequently, making use of conventional computers, different simulations of increasing complexity have been conducted. In this way, it has been possible to draw conclusions about the results that these algorithms provide and their advantages and disadvantages when applied to real-world problems.

# Capítulo 1

# Introducción

Tareas como simular sistemas físicos complejos o resolver problemas de álgebra lineal a gran escala suponen un desafío considerable para los ordenadores clásicos debido al alto coste computacional que estos conllevan. Por esta razón, los ordenadores cuánticos empiezan a cobrar gran importancia ya que se espera que sean la solución a estos problemas reduciendo enormemente el tiempo de computación.

Dentro del desarrollo de este nuevo ámbito aparecen gran variedad de ramas a desarrollar para asegurar el buen funcionamiento de esta tecnología. Entre estos avances encontramos el desarrollo de *hardware* que permita trabajar a bajas temperaturas, estudios de sistemas cuánticos que permitan interactuar con estos ordenadores o el uso de campos magnéticos de forma rápida y eficiente. Sin embargo, de entre todas ellas, la que va a ser objeto de estudio en este documento consiste en la necesidad de ingeniar nuevos algoritmos con los que estos ordenadores puedan trabajar.

Debido a las enormes diferencias que existen entre la computación clásica que conocemos hoy en día y los ordenadores cuánticos, estos nuevos algoritmos no se asemejan en absoluto a los que se han ido usando hasta la fecha. Es por ello que hay que desarrollar estos en un paradigma completamente diferente.

Dentro del abanico de algortimos cuánticos que existen, este documento se centra en los Algoritmos de Optimización Cuántica Aproximada conocidos en inglés como *Quantum Approximate Optimization Algorithm* (QAOA), un tipo de algoritmo que se usa para la optimización de funciones en las que las variables involucradas puedan tomar únicamente dos valores. Estas funciones reciben el nombre de funciones dicotómicas y aunque a primera vista parezcan sencillas de minimizar, cuando aumenta el número de variables involucradas, la complejidad y tiempos de computación clásicos aumentan exponencialmente. Por esta razón, los ordenadores cuánticos buscan llevar estas tareas en un tiempo inferior.

Sin embargo, este tipo de algoritmo se incluye a su vez dentro de otros dos grupos más grandes que engloban un abanico de algoritmos mucho más amplios. El primero de estos grupos en los que los QAOA se incluyen recibe el nombre de computación cuántica adiabática, llamado así por su relación con el teorema cuántico adiabático que permite relacionar estados de mínima energía de hamiltonianos diferentes siempre y cuando se lleve en un tiempo suficientemente lento. Y finalmente, el segundo gran grupo se conoce como Algoritmos Cuánticos Variacionales y su principal

característica es la utilización en segundo plano de un ordenador clásico para llevar a cabo una optimización paramétrica de una serie de variables que se reflejan en el hamiltoniano del sistema cuántico de estudio.

## 1.1. Descripción de los contenidos

El documento se encuentra divido en cinco capítulos bien diferenciados, además consta de dos anexos complementarios para ayudar a una mejor comprensión del análisis de datos que se ha llevado a cabo.

En primer lugar, este capítulo sirve de introducción al propio documento y de cómo este plantea el estudio de los Algoritmos de Optimización Cuántica Aproximada. Pero debido a la gran diferencia que plantea la computación cuántica frente a la computación clásica, y debido también a la necesidad de comprender como funcionan estos ordenadores y así poder entender los algoritmos que se aplican, este capítulo contiene además una breve sección que permite al lector familiarizarse con los conceptos que este nuevo paradigma computacional presenta.

En el siguiente capítulo nos adentramos en los propios algoritmos cuánticos, explicando en primer lugar los Algoritmos Cuánticos Variacionales junto a la Computación Cuántica Adiabática. Esto consiste en una primera explicación esquematizada de su funcionamiento seguida de un desarrollo teórico que se basa en el Teorema Adiabático y que, tras una serie de cálculos y adaptaciones, nos permiten llegar a entender el funcionamiento de los QAOA.

A continuación, presentamos también un pequeño capítulo que explica la aplicabilidad de estos algoritmos en relación a la física, en concreto al estudio del sistemas magnéticos, junto a otro tipo de problemas de optimización mucho más amplio y su relación con los hamiltonianos de la física cuántica.

El último tramo del documento conlleva una aplicación numérica de este tipo de algoritmos. Para ello hemos llevado a cabo una serie simulaciones en un ordenador convencional con ayuda del lenguaje de programación *Python*, en el entorno *Visual Studio Code*. Estas simulaciones están compuestas de tres partes: la primera se encarga de simular el trabajo de un ordenador cuántico, puesto que no se ha dispuesto de una a lo largo del trabajo, la segunda refleja la optimización clásica mientras que la tercera trata el análisis de los resultados obtenidos. Estos códigos pueden encontrarse en el Apéndice correspondiente.

Las simulaciones se han llevado en orden creciente de dificultad. En primer lugar, se trabaja con una función dependiente de una única variable, lo cual nos permite sacar conclusiones extrapolables a cualquier caso más complejo. En segundo lugar se trabaja para casos más generales que nos permiten ya aplicar el algoritmo a problemas reales y, finalmente, se estudia la eficacia de estos algoritmos aumentando el número de operaciones a lo largo del circuito. Todo ello ha sido realizado mediante código desarrollado por nosotros mismos y aparece también relacionado con los resultados teóricos que los respaldan.

Como se ha mencionado también, al final del documento, siguiendo a la bibliografía, aparecen dos anexos: en el primero se explican una serie de conceptos estadísticos utilizados para el análisis estadísticos de los datos mientras que en el segundo aparecen explícitamente los códigos utilizados.

## 1.2. Bases de la computación cuántica

Tras una primera aproximación a lo que va a ser objeto de estudio a lo largo del documento es necesario comenzar con lo que compone las base de la computación cuántica y de cómo esta funciona para que el lector pueda seguir adecuadamente los algortimos cuánticos implicados [1, 27, 12, 28, 29].

Este nuevo modelo de computación, completamente distinto a lo que conocemos hoy en día como informática, se basa en dos principios fundamentales de la mecánica cuántica: la superposición cuántica y el entrelazamiento cuántico.

### 1.2.1. Principio de superposición cuántica

Dado un sistema cuántico que pueda existir en una serie de autoestados diferentes $\{|\psi_i\rangle\}_{i\in I}$, donde $I$ representa el número de autoestados ya sea finito o infinito, cualquier combinación lineal de esos estados que cumpla la normalización del estado es también un estado permitido del sistema:

$$|\phi\rangle = \sum_{i\in I} a_i \,|\psi_i\rangle\,, \qquad |\,\langle\phi|\phi\rangle\,|^2 = 1. \tag{1.1}$$

Generalmente, para describir estos sistemas se usa uno de los grados de libertad intrínseco de las partículas atómicas conocido como spin. Usualmente y por simplicidad experimental se usan partículas de spin $\frac{1}{2}$ que tiene dos autoestados posibles $|1\rangle$ y $|0\rangle$ con momento angular en el eje $z$ igual a $\frac{\hbar}{2}$ y $-\frac{\hbar}{2}$ respectivamente. De esta forma, teniendo en cuenta el principio de superposición cuántica, el sistema puede ser descrito de la siguiente forma:

$$|\phi\rangle = \alpha\,|1\rangle + \beta\,|0\rangle\,, \qquad |\alpha|^2 + |\beta|^2 = 1. \tag{1.2}$$

Es importante también notar que aun estando el sistema en este estado, a la hora de medir el spin, solo se pueden obtener esos valores de $\frac{\hbar}{2}$ y $-\frac{\hbar}{2}$ con probabilidades $|\alpha|^2$ y $|\beta|^2$ respectivamente. Por simplicidad del modelo y con intención de no sobrecargar la notación, se establece a lo largo de todo el documento que $\hbar = 1$.

### 1.2.2. Entrelazamiento cuántico

Dado dos sistemas que inicialmente no interactuaban entre si y que se encuentran respectivamente en los siguientes estados, $|\phi_1\rangle$ y $|\phi_2\rangle$, la descripción global de ambos sistemas se puede describir en términos del producto tensorial, $|\psi\rangle = |\phi_1\rangle \otimes |\phi_2\rangle = |\phi_1, \phi_2\rangle$. Este tipo de estados recibe el nombre de **estados producto tensorial**. Son muy cómodos de estudiar ya que la medición de uno de los sistemas no altera el restante.

Por otro lado, si los dos sistemas interactúan entre si y el sistema global evoluciona, el estado final dependerá de esta evolución conjunta y podría dar lugar a un tipo de estado cuántico distinto al inicial que **no** sea un estado producto tensorial. Los estados que no pueden ser escritos de forma de producto tensorial se conocen con el nombre de **estados entrelazados**. La importancia de estos estados radica en que aunque estos se separen y comiencen a ser independientes nuevamente, las mediciones que se hagan sobre uno de ellos afectará el estado del otro.

### 1.2.3. Qubits

El siguiente paso es entender las bases de la computación cuántica y cómo los dos conceptos previos se ven involucrados. Para ello, haremos un comparativa con la computación clásica como se conoce hoy en día.

La primera diferencia hace referencia a la unidad básica de información. Mientras que clásicamente se usa el **bit**, que puede tomar exactamente dos valores entre 0 o 1 que corresponden por ejemplo al paso o no de corriente en un transistor respectivamente. En la computación cuántica aparece el **qubit**. Un qubit es un sistema cuántico con dos estados propios que puede ser manipulado arbitrariamente y únicamente tiene dos estados bien distinguibles mediante medidas físicas. Sin embargo, lo que los diferencia enormemente de los bits es la superposición cuántica, es decir, un circuito clásico de 2 bits puede estar únicamente en cuatro estados diferentes: 00, 01, 10 o 11, mientras que un sistema cuántico podrá estar simultáneamente en una combinación lineal de estos.

Siguiendo este ejemplo, aparece la idea de **base computacional**, que será la base que usaremos para describir el estado cuántico de nuestro ordenador. Esta base para los dos qubits es de la siguiente manera: $\{|00\rangle = |0\rangle \otimes |0\rangle, |10\rangle, |01\rangle, |11\rangle\}$. Generalizado para $N$ qubits, nuestro sistema será un espacio de Hilbert de dimensión $2^N$, de forma que la base estará compuesta de $2^N$ elemento:

$$|\psi\rangle = \bigotimes_{i=1}^{N} |\alpha_i\rangle, \quad \alpha_i \in \{0, 1\}. \tag{1.3}$$

Por tanto, cualquier estado cuántico de dos niveles se puede utilizar para representar un qubit o incluso los sistemas de niveles múltiples se pueden utilizar también si poseen dos estados que se puedan desemparejar con eficacia del resto. Hay varias opciones de este tipo de sistemas que se han puesto en práctica con diferentes grados de éxito como son las trampa de iones o de átomos [21], defectos cristalinos en diamante [22], puntos cuánticos [23] o el que usaremos nosotros a partir de aquí que son los espines nucleares o más exactamente la polarización de la magnetización de esos núcleos en un vasto número de moléculas idénticas ya que estas son fácilmente manejables mediante campos magnéticos para así conseguir el circuito que se desea [24].

### 1.2.4. Puertas cuánticas

La otra gran diferencia entre paradigmas informáticos son las puerta lógicas que se usan para hacer operaciones con estos bits. En la informática clásica reciben el nombre de puerta lógicas. Estas reciben un input en forma de ceros y unos y a la salida expulsan un único output que será o bien un 1 o bien un 0. Haciendo combinaciones de millones y millones de estas puerta lógicas se construye cualquier algoritmo clásico que conocemos hoy en día.

En el otro lado se encuentran las puerta lógicas cuánticas. Estás también reciben un input que es el estado en el que se encuentran los diferentes qubits que participan en la puerta y, a diferencia de las puerta lógicas, los output vuelven a ser los qubits anteriores pero en un nuevo estado. Este nuevo estado se ha generado por la interacción de todos los qubits que han sido incluidos en la puerta cuántica junto con la propia puerta de forma que el estado a la salida depende de todos los qubits involucrados, es aquí donde cobra importancia el entrelazamiento cuántico, es decir, lo que después se haga en cada uno de los qubits influencia también a todos los demás.

Estas puerta lógicas cuánticas pueden ser descritas mediante operadores unitarios para que el estado de salida siga siendo un estado cuántico. Esto nos permite representar estas puertas en la base computacional mediante matrices de orden $2^N$ donde $N$ es el número de qubits que componen el ordenador. Algunos de los ejemplos de las puertas más comúnmente utilizadas son las siguientes:

**Puerta Hadamard**

Esta puerta actúa sobre un único qubit, y descrita en la base computacional tiene la siguiente forma:

$$\hat{\mathbf{H}} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}. \tag{1.4}$$

Se observa que el estado final tras actuar sobre los elementos de la base $\{|1\rangle, |0\rangle\}$ son $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ y $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ respectivamente. De esta forma, lo que hace este operador es igualar las probabilidades de ambos estados de la base. Es de resaltar que la compuerta Hadamard recibe un estado puro y lo convierte en estado entrelazado.

**Puertas de Pauli**

Las puertas de Pauli están asociadas a las tres matrices de Pauli ($\hat{\sigma}_x$, $\hat{\sigma}_y$, $\hat{\sigma}_z$) que actúan sobre un sólo qubit. Las puertas de Pauli ejecutan una rotación de $\pi$ radianes alrededor de los ejes $x$, $y$, $z$ de la esfera de Bloch. Las cuales pueden ser representadas en la base computacional o del eje $z$ como:

$$\hat{\sigma}_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \hat{\sigma}_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad \hat{\sigma}_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{1.5}$$

La importancia de estas matrices se reduce a su utilización para definir las rotaciones.

**Rotaciones**

De igual manera que se definen en mecánica cuántica los operadores de rotación, tanto de momento angular orbital como de spin se definen las puertas de rotación, en este caso, haciendo referencia únicamente al spin, que es lo que esta en juego en nuestros ordenadores cuánticos [25, 26].

$$\hat{R}_{\mathbf{u}}(\alpha) = e^{-\frac{i}{\hbar}\alpha \hat{\mathbf{S}}\mathbf{u}} = e^{-i\frac{\alpha}{2}\hat{\sigma}\mathbf{u}}. \tag{1.6}$$

Donde $\hat{\mathbf{S}} = \hat{S}_x u_x + \hat{S}_y u_y + \hat{S}_z u_z$, $\sigma = \hat{\sigma}_x u_x + \hat{\sigma}_y u_y + \hat{\sigma}_z u_z$, $u = (u_x, u_y, u_z)$ es el vector unitario en la dirección del eje de giro y $\alpha$ representa el ángulo de giro.

Este operador se puede escribir también de la siguiente manera de forma que la matriz que representa la puerta cuántico sea,

$$\hat{R}_{\mathbf{u}}(\alpha) = \cos\left(\frac{\alpha}{2}\right) - i\sigma\mathbf{u}\sin\left(\frac{\alpha}{2}\right) = \begin{pmatrix} \cos(\frac{\alpha}{2}) - iu_z\sin(\frac{\alpha}{2}) & (-iu_x + u_y)\sin(\frac{\alpha}{2}) \\ (-iu_x + u_y)\sin(\frac{\alpha}{2}) & \cos(\frac{\alpha}{2}) + iu_z\sin(\frac{\alpha}{2}) \end{pmatrix}. \tag{1.7}$$

**Puertas de desplazamiento de fase**

Esta puerta actúa sobre un único qubit y deja intacto al estado $|0\rangle$ mientras que a $|1\rangle$ se le asigna un desfase $e^{i\phi}$.

$$\hat{R}(\phi) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{bmatrix}. \tag{1.8}$$

Para $\phi = \pi$ equivale a la matriz de Pauli en el eje $z$.

**Puerta SWAP**

Esta puerta actúa sobre dos qubits y como su propio nombre indica intercambia los estados de los qubits, de forma que si el estado inicial es $|01\rangle$ el final será $|10\rangle$. Su representación matricial en la base computacional $B = \{|11\rangle, |01\rangle, |01\rangle, |00\rangle\}$ es la siguiente:

$$\mathbf{S\hat{W}AP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{1.9}$$

**Puerta controlada**

Estas puertas actúan sobre dos qubits siendo el primero de ellos el controlador y el segundo de ellos sobre el que actúa un operador unitario $\hat{U}$, de forma que si el primero es $|0\rangle$ no ocurrirá nada, y si es $|1\rangle$ el operador $\hat{U}$ actúa sobre el segundo qubit. En la base computacional mencionada previamente:

$$\hat{C}(\hat{U}) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & x_{00} & x_{01} \\ 0 & 0 & x_{10} & x_{11} \end{bmatrix}, \quad \text{siendo} \quad \hat{U} = \begin{bmatrix} x_{00} & x_{01} \\ x_{10} & x_{11} \end{bmatrix}. \tag{1.10}$$

De esta forma, una vez conocido la representación matemática de un circuito cuántico, será necesario definir una forma de dibujar y esquematizar estos circuitos. En primer lugar, estos se leerán de izquierda a derecha de forma que a la izquierda del todo se encuentren los qubits en su estado inicial que, salvo que se indique lo contrario, estarán todos en el estado $|0\rangle$.

Por otro lado, a diferencia de un circuito clásico en el que una puerta lógica con varios inputs solo tiene un output, el número de qubits antes y después de cualquier puerta es el mismo y por lo tanto, cada qubit y su registro[1] se representa por una línea horizontal una debajo de otra. Las puerta lógicas que actúan en uno o varios registros de qubits se indican como un cuadro en el que en su interior se indica el operador unitario que representan.

Finalmente la operación restante que se va a visualizar en los diagramas de los circuitos es la acción de la medida. Esta toma un único registro de qubit individualmente, lo mide y genera el resultado como información clásica.

[1]El registro de un qubit es el nombre que recibe su evolución a lo largo del circuito.

En la Figura 1.1 se pueden apreciar alguna de las puerta lógicas más conocidas.

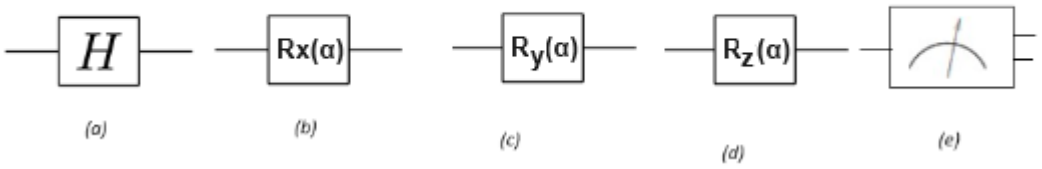



Figura 1.1: **Representación esquemática de las puerta lógicas**: (a) Puerta Hadamard, (b) Puerta de rotación entrono al eje x un ángulo $\alpha$, (c) Puerta de rotación entrono al eje y un ángulo $\alpha$, (d) Puerta de rotación entrono al eje z un ángulo $\alpha$ y (e) medidor del estado cuántico de un qubit.

# Capítulo 2

# Algoritmos cuánticos

Una vez se tienen los conocimientos básicos sobre circuitos cuánticos el siguiente paso será diseñar algoritmos cuánticos intentando solventar cualquier problema de forma mucho más eficiente de lo que lo haría un ordenador clásico.

Sin embargo, las versiones de los ordenadores cuánticos actuales y las que se prevé que existirán a corto plazo se enfrentan a varios problemas. El más importante de ellos se conoce como **decoherencia cuántica** que, a grandes rasgos, es el proceso en el que un sistema físico, bajo ciertas condiciones específicas, pasa a exhibir un comportamiento clásico, sin los efectos típicos de la mecánica cuántica como son la superposición y entrelazamiento. Por esto, los qubits deben estar en condiciones muy especificas como por ejemplo a muy bajas temperaturas o totalmente aislados de campos externos. Estos tiempos en los que un qubit pierde la coherencia cuántica dependen de muchos factores como es esa temperatura o aislamiento. Estos se llaman tiempos de coherencia y están entorno a unos 70 microsegundos, por lo tanto es necesario que los cálculos que lleven a cabo estos ordenadores se hagan en menos de ese tiempo lo que supone una limitación en el número de qubits a utilizar y en la longitud de los circuitos.

Los Algoritmos Cuánticos Variacionales (*Variational Quantum Algorithms*, VQA en inglés) surgen como una respuesta para lidiar con estas limitaciones.

## 2.1. Algoritmos Cuánticos Variacionales (VQA)

Este tipo de algoritmos recibe este nombre de las siglas en inglés, *Variational Quantum Algorithms* y se caracterizan a grandes rasgos por emplear un ordenador clásico como optimizador de forma que nos ayude a entrenar un circuito cuántico dependiente de una serie de parámetros. Actualmente, existen VQAs para prácticamente todas las aplicaciones que los investigadores han concebido para los ordenadores cuánticos y parecen ser la esperanza para el uso de esta nueva tecnología. No obstante, aún hay mucho en lo que mejorar con lo que respecta a la capacidad de entrenamiento, la precisión y la eficiencia de estos algoritmos [4].

De esta forma, un VQA se puede ver como un algoritmo híbrido en el que se combina por un lado la actuación de un circuito cuántico junto con la optimización clásica de los ordenadores actuales.

Su metodología consiste en utilizar un circuito cuántico que refleje esa función a minimizar y que sea dependiente de una serie de parámetros. Los inputs serían así la función de coste y una aproximación inicial de lo que valen estos. Una vez se tiene el circuito este se puede evaluar cuando se quiera y, con ayuda del ordenador clásico y usando los resultados del ordenador cuántico, encontrar los parámetros óptimos dependiendo de lo que se quiera. De forma esquemática, el proceso que llevan estos ordenadores se puede visualizar en la Figura 2.1. Esto supone así un algoritmo cíclico en el que se va mejorando la actuación del ordenador cuántico con cada ciclo al variar los parámetros.

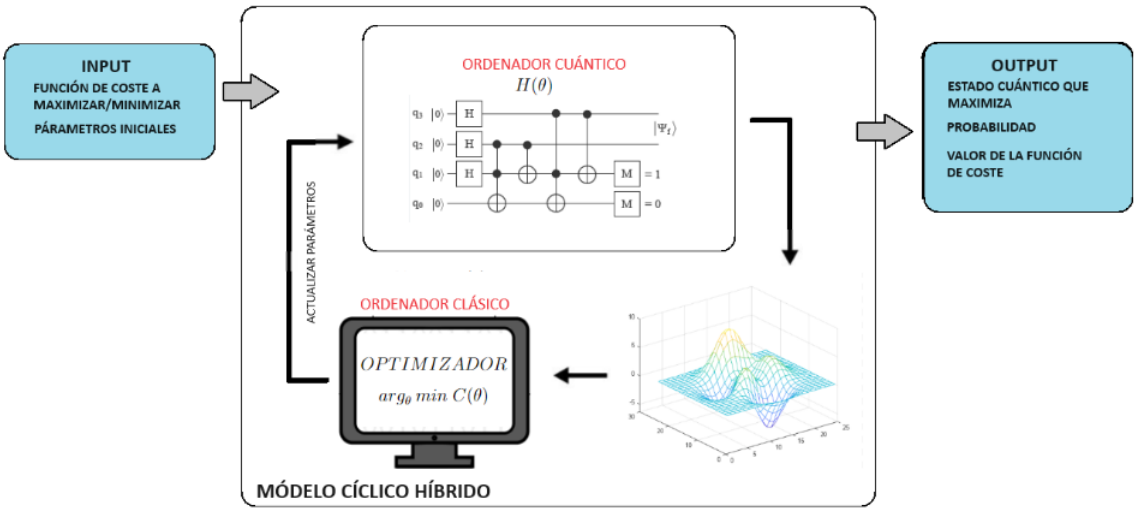



Figura 2.1: **Esquema del funcionamiento básico de un VQA.** En el recuadro central de la figura se representa un circuito cuántico de 5 qubits, en el cual los qubits están siendo operados por compuertas y medidores al final del circuito. La configuración o disposición de dichas puertas cuánticas generan una operación global sobre los qubits denotada por $H(\theta)$ y dicha operación global define la función de coste a optimizar, que depende de ese único parámetro a optimizar $\theta$. La función de coste definirá en el espacio de parámetros una superficie con mínimos y máximos, el trabajo del proceso de optimización clásico sera encontrar ese mínimo o máximo según sea la necesidad del problema. De forma que esta será la configuración final de parámetros que se requiere el ordenador cuántico para la solución del problema.

A su vez, en estos tipos de algoritmos, los inputs pueden incluir desde funciones de coste simples hasta funciones complejas que incluyen además *training data*. Lo mismo ocurre con los outputs o el propio circuito cuántico en sí, existe gran variedad de alternativas. Por eso dentro de los VQAs se distinguen gran cantidad de subtipos, entre los que se encuentran los Algoritmos Cuánticos de Optimización Aproximada o, como se conocen en inglés, *Quantum Approximate Optimization Algorithms* (QAOA) en los que nos centraremos nosotros [2, 5].

## 2.2. Algoritmos de Optimización Cuántica Aproximada (QAOA)

Como se ha dicho en el apartado anterior, QAOA es un tipo de VQA que están además destinado a la optimización de funciones discretas. Estas funciones a las que llamaremos funciones de coste

$C$, se escribirán de la forma:

$$C(\mathbf{z}) = \sum_{k=1}^{2^n} a_k C_k(\mathbf{z}), \tag{2.1}$$

donde $\mathbf{z} = z_1 z_2 z_3 .. z_n$ es la secuencia de bits, $a_k$ son constantes arbitrarios que caracterizan nuestro problema y $C_k$ es un valor que valdrá o bien 0 o bien 1 dependiendo de los valores de los $z_i$. Se comporta así como una delta de Dirac que vale 1 únicamente en el estado número k de la base computacional.

Ahora bien, dado que la función de coste definida por Ec. (2.1) define un problema especifico clásico, es necesario traducir dicho problema de optimización a un formato que un algoritmo cuántico pueda entender. La función de coste puede ser generalmente implementada como un Hamiltoniano. A partir de aquí, esta función $C$, será el hamiltoniano problema de nuestro sistema, $\hat{C}$ y sus autofunciones y autovalores serán de la siguiente forma:

$$\hat{C}\,|\mathbf{z}\rangle = C(\mathbf{z})\,|\mathbf{z}\rangle\,, \tag{2.2}$$

siendo así C un operador diagonal en la base computacional.

Por otro lado, este tipo de algoritmo se caracteriza sobretodo por presentar un circuito cuántico formado por un número determinado de capas al que denotaremos por $p$ (Este valor tendrá gran importancia en los resultados que se obtengan a lo largo del documento). Además, estas capas tienen una estructura común, dos puertas cuánticas definidas como sigue:

1. Un operador evolución temporal definido por el hamiltoniano $\hat{C}$ y una constante $\gamma_i$:

$$\hat{U}(\hat{C}, \gamma_i) = e^{-i\gamma_i \hat{C}} = \prod_{\alpha=1}^{m} e^{-i\gamma \hat{C}_\alpha}, \tag{2.3}$$

   donde el subíndice i que acompaña a $\gamma_i$ indica la capa en la que nos encontramos

2. Una rotación de un ángulo $\theta_i$ entorno al eje x:

$$\hat{U}(\hat{B}, \theta_i) = e^{-i\theta_i \hat{B}} = \prod_{j=1}^{n} e^{-i\beta\hat{\sigma}_x^j} = \prod_{j=1}^{n} \hat{R}_x^j(\theta_i)\,, \qquad \hat{B} = \sum_{j=1}^{n} \hat{\sigma}_x^j\,, \tag{2.4}$$

   donde $\hat{\sigma}_x^j$ es el operador del spin en la dirección x del spin $j$ representado por la matriz de Pauli correspondiente.

Además, al inicio del circuito cuántico, el estado inicial del ordenador cuántico siempre será el estado $|\mathbf{z}\rangle = |000...,00\rangle$. Este pasará primero por una puerta que actúan en cada qubit individualmente de forma que igualen la probabilidad del estado 1 y 0. Esta puerta se ha mencionado anteriormente, recibe el nombre de puerta Hadamard y nos permiten que el estado al entrar en la capa inicial sea:

$$|\mathbf{s}\rangle = \frac{1}{\sqrt{2^n}} \sum_{z_i} |\mathbf{z_i}\rangle\,, \tag{2.5}$$

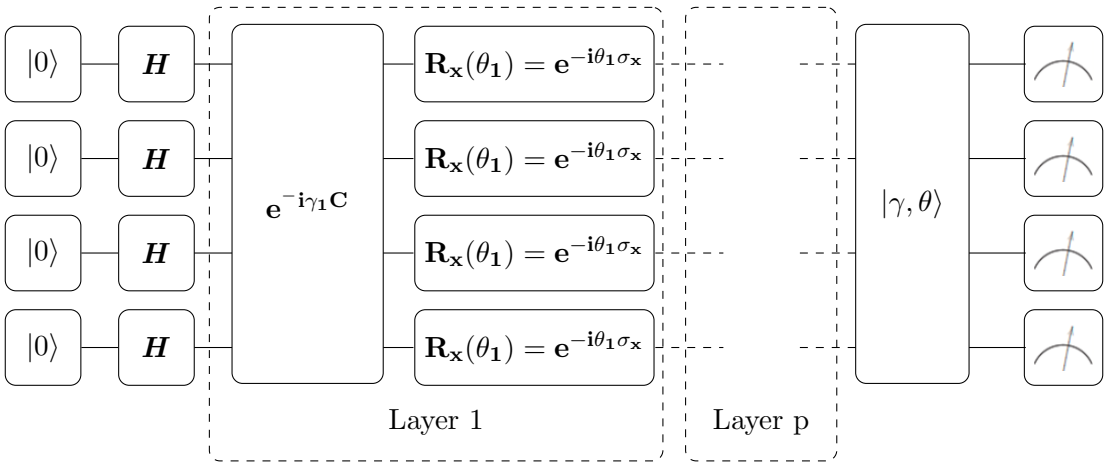
donde $\mathbf{z_i}$ representa cada uno de los estados de la base computacional.

Teniendo todo esto en cuenta, ya podemos escribir nuestro estado al final del circuito:

$$|\gamma, \theta\rangle = \hat{U}(\hat{B}, \theta_p)\hat{U}(\hat{C}, \gamma_p) \cdots \hat{U}(\hat{B}, \theta_1)\hat{U}(\hat{C}, \gamma_1)|s\rangle. \tag{2.6}$$

Y por tanto el valor esperado de la función de coste será $\langle \gamma, \theta | \hat{C} | \gamma, \theta \rangle$.

De forma esquemática se representa como sigue:




Es importante darse cuenta que si al final del circuito medimos el estado final, o el valor de $\hat{C}$, no obtendremos el estado $|\gamma, \theta\rangle$ sino un estado de la base computacional que dependerá de las amplitudes de probabilidad que conformen ese estado final. Por tanto, si queremos obtener ese valor esperado de $\hat{C}$, habrá que medir varias veces y tomar el valor medio.

### 2.2.1. Origen del algoritmo QAOA: Computación Cuántica Adiabática (QAA)

Para entender de donde viene este algoritmo y porque funciona es imprescindible conocer y entender el teorema adiabático.

**Teorema adiabático** (Max Born y Vladimir Fock, 1928) [14, 25, 28]: "*Un sistema físico permanece en su estado propio instantáneo si una perturbación dada actúa sobre él lo suficientemente lento y si existe una brecha entre el autovalor asociado y el resto del espectro del Hamiltoniano.*"

Dicho de otra forma, si el sistema comienza en un estado propio del Hamiltoniano inicial, terminará en el estado propio correspondiente del Hamiltoniano final siempre y cuando las condiciones que cambian gradualmente de un hamiltoniano a otro, permiten que el sistema adapte su configuración.

Nuestro hamiltoniano dependiente del tiempo será el siguiente:

$$\hat{H}(t) = \frac{t}{T}\hat{C} + \left(1 - \frac{t}{T}\right)\hat{B}, \tag{2.7}$$

donde para $t = 0$, el hamiltoniano inicial será $\hat{H}(0) = \hat{B}$ que se ha definido anteriormente según las matrices de Pauli como $\hat{B} = \sum_{j=1}^{n} \hat{\sigma}_x^j$. De esta forma, los estados iniciales de los qubits antes

de entrar a las capa $p = 0$ y tras atravesar las puertas Hadamard, se encuentran en el estado $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$, correspondiente al estado base de menor energía. Y tras un tiempo $t = T$ el sistema estará dirigido por el hamiltoniano $\hat{C}$, que corresponde al hamiltoniano mapeado correspondiente a nuestra función a minimizar. Si $T$ es suficientemente grande como para ser considerado un proceso lentamente variable, el teorema adiabático afirma que el estado final del sistema será el estado base del hamiltoniano $\hat{C}$ y por tanto el valor esperado de $\hat{C}$ al final del circuito será el valor del mínimo buscado.

Esta idea de porque la evolución del sistema por si solo nos lleva al estado de mínima energía que buscamos es la base de lo que se conoce como **Computación cuántica adiabática** o en inglés, **Quantum Adiabatic Algorithm (QAA)** [14, 15].

Sin embargo, como se vio previamente, existe un tiempo de coherencia en el que el sistema cuántico se ve afectado por la decoherencia y la información codificada en los qubits se ve perturbada. Esto implica que el tiempo $T$, que en principio se busca lo más grande posible, tiene que ser a su vez pequeño para no perder la coherencia. Por esto, aplicar esta idea a un ordenador cuántico es una tarea complicada. Surge una alternativa que consiste la discretización del modelo. En primera aproximación, dividimos el espacio de tiempos $[0, T]$ en partes no necesariamente iguales de forma que en esos intervalos de tiempo se pueda considerar que el hamiltoniano es constante:

$$(0, T) = (0, t_1) \cup (t_1, t_2) \cup ... \cup (t_{k-1}, t_k) = \cup_{j=1}^{k} (t_{i-i}, t_i) \tag{2.8}$$

tal que $\hat{H}(t) = (1 - \frac{t_i}{T})\hat{C} + \frac{t_i}{T}$ si $t \in (t_{i-1}, t_i)$. Esta idea es equivalente a las que se usan para resolver numéricamente las ecuaciones diferenciales.

De esta forma, al final de cada intervalo, el estado vendrá dado por la resolución de la ecuación de Schrödinger dependiente del tiempo:

$$|\psi(t_i)\rangle = e^{-i\hat{H}(t_i)(t_i - t_{i-1})} |\psi(t_{i-1})\rangle . \tag{2.9}$$

Y si aplicamos esta idea de forma recurrente el estado final del sistema se puede escribir como sigue:

$$|\psi(T)\rangle = \prod_{i=1}^{k} e^{-i\hat{H}(t_i)(t_i - t_{i-1})} |\psi(0)\rangle . \tag{2.10}$$

Sin embargo, estos hamiltonianos aún no son manejables ya que cada uno de los $\hat{H}(t_i)$ depende a su vez de $\hat{C}$ y $\hat{B}$, por tanto, buscaremos separarlos. Entra así en juego lo que se conoce como **troterización de la evolución temportal** o **trotterized aproximation of time evolution** [28, 11, 13]. Esta se basa en la fórmula del **producto de Trotter** que establece que para dos matrices complejas arbitrarias de orden m, se cumple:

$$e^{A+B} = \lim_{n \to \infty} \left( e^{\frac{A}{n}} e^{\frac{B}{n}} \right)^n . \tag{2.11}$$

Y por tanto se puede tomar la siguiente aproximación para $r$ suficientemente grande:

$$e^{A+B} \approx \left( e^{\frac{A}{r}} e^{\frac{B}{r}} \right)^r . \tag{2.12}$$

Si particularizamos al caso del intervalo $(t_i, t_{i-1})$ llegamos a:

$$e^{-i\hat{H}(t_i)(t_i-t_{i-1})} = e^{-i\left[\left(\frac{1}{r_i}-\frac{t_i}{T}\right)\hat{B}+\frac{t_i}{T}\hat{C}\right](t_{i+1}-t_i)} \approx \left(e^{-i\left(1-\frac{t_i}{Tr_i}\right)\hat{B}(t_{i+1}-t_i)}e^{-i\frac{t_i}{Tr_i}\hat{C}(t_{i+1}-t_i)}\right)^{r_i}, \quad (2.13)$$

donde $r_i$ corresponde al $r$ elegido en cada subintervalo $(t_{i+1,}t_i)$ para llevar a cabo la troterización.

Si juntamos ahora las aproximaciones (2.10) y (2.13):

$$\begin{aligned}
|\psi(T)\rangle &= \prod_{i=1}^{k}\left(e^{-i(\frac{1}{r_i}-\frac{t_i}{Tr_i})\hat{B}(t_{i+1}-t_i)}e^{-i\frac{t_i}{Tr_i}\hat{C}(t_{i+1}-t_i)}\right)^{r_i}|\psi(0)\rangle \\
&\sim \prod_{l=1}^{p}\left(e^{-i\theta_l\hat{B}}e^{-i\gamma_l\hat{C}}\right)|\psi(0)\rangle\ . \\
|\psi(T)\rangle &\sim \prod_{l=1}^{p}U(\hat{B},\theta_l)U(\hat{C},\gamma_l)\,|\mathbf{s}\rangle\,. \qquad (2.14)
\end{aligned}$$

Donde se ha realizado una agrupación de subíndices para englobar en el mismo productorio tanto la partición del intervalo $t_i$ como las exponenciales $r_i$ siendo $p$ el número de capas finales.

Es necesario remarcar que este resultado final es en si una aproximación que en el limite cuando el número de capas $p$ tiende a infinito coincidirá con el valor real del estado final. Además, según el procedimiento que se ha llevado a cabo, las constantes que aparecen en la formula final $\gamma_l$ y $\theta_l$ tienen una fórmula fija en función de los $t_i$ y los $r_i$. Sin embargo, teniendo en cuenta que estos son arbitrarios y buscan en si ser elegidos de forma que la aproximación sea lo mejor posible, podemos de manera más directa trabajar con los $\gamma_l$ y $\theta_l$ en lugar de los $t_i$ y los $r_i$.

Es por estas dos cosas que este tipo de algoritmos lleva en su nombre la palabra aproximación ya que no se busca como en los algoritmos cuanticos adiabaticos (QAA) llegar a una solución exacta, sino a una buena aproximación que nos de el valor máximo o mínimo de la función.

Una vez llegados a este punto, nos surge la duda de cuales serán esos parámetro $\theta$ y $\gamma$ que hagan que nuestro algoritmo funcione y sobretodo, cómo calcularlos. La respuesta a esto no es cerrada, sino que depende del algoritmo, de su complejidad y del problema en concreto, hay una infinidad de posibles soluciones. Por ejemplo, [4] nos presenta una infinidad de métodos entre los que destaca el usar el ordenador cuántico para calcular los valores en función de esos ángulos y con ayuda de un ordenador clásico llevar a cambo métodos de *gradient descent* para llegar a lo más óptimo. Sin embargo, nosotros lo que haremos es o bien calcular analíticamente los valores de esos máximos y mínimos en caso de que sea posible, o por otro lado, usar un ambiente de programación clásico que nos lo resuelva.

# Capítulo 3

# Resolución de problemas con QAOA

Como se ha visto anteriormente, el objetivo de este tipo de algoritmos es minimizar funciones discretas donde las variables pueden tomar dos únicos valores (variables dicotómicas). Además, estas funciones se han de poder convertir en un hamiltoniano para que puedan interactuar los qubits con el ordenador cuántico, aparecen así lo que se conocen como **hamiltonianos tipo Ising** [16].

## 3.1. Hamiltoniano de Ising

El modelo de Ising clásico es un modelo matemático utilizado en física estadística para describir el comportamiento de los sistemas magnéticos. Este tipo de hamiltonianos reciben el nombre en honor al físico *Ernst Ising* por sus investigaciones en las transiciones de fase, siendo uno de los pocos modelos que existen de partículas que interactúan entre si con solución exacta y siendo además el más sencillo a resolver. Desde una mirada simple el modelo de Ising clásico se compone de los siguientes elementos claves:

- Spin: Cada sitio de una red (generalmente una red cuadrada en 2D o una red cúbica en 3D) contiene un spin que puede tomar uno de dos valores: $+1$ o $-1$, representando estados de "spin up o spin down".
- Interacciones: Los spins interactúan con sus vecinos más cercanos. La fuerza de interacción entre dos spinS adyacentes $i$ y $j$ está cuantificada por un peso de interacción $J_{i,j}$.

El objetivo en estudios del modelo de Ising clásico es analizar cómo los spins se alinean a distintas temperaturas y campos magnéticos, especialmente cerca de la temperatura crítica donde ocurre una transición de fase. Su papel ha sido clave en el desarrollo histórico de la comprensión del ferromagnetismo y de las transiciones de fase y actualmente se utiliza este modelo para explicar una gran variedad de fenómenos, no solo físicos, sino también en diversas áreas de la biología.

Tomaremos como punto de partida la explicación del ferromagnetismo. Este se debe a que una fracción importante de los momentos magnéticos asociados a los espines de los átomos se alinean en la misma dirección debido a la interacción entre los mismos, dando lugar a la imanación de la

muestra. Además, por encima de cierta temperatura característica llamada temperatura de Curie, esta alineación se pierde produciéndose una transición de fase, con una fenomenología diferente.

Ahora por otro lado, El modelo de Ising cuántico es una extensión del modelo clásico que incorpora efectos cuánticos que lo caracterizan. Las diferencias principales son:

- Spins Cuánticos: En lugar de ser variables clásicas que pueden tomar valores $+1$ o $-1$, los spin ahora son operadores cuánticos que actúan sobre estados de qubits. Estos operadores están representados por matrices de Pauli.
- Hamiltoniano Cuántico: El Hamiltoniano del modelo de Ising cuántico incluye términos que describen interacciones clásicas entre spins, así como términos que introducen fluctuaciones cuánticas.

En el modelo de Ising cuántico, el objetivo es entender cómo las interacciones y las fluctuaciones cuánticas afectan el estado del sistema, especialmente a bajas temperaturas y en presencia de campos transversales. Este modelo es fundamental en el estudio de transiciones de fase cuánticas y en aplicaciones de computación cuántica y simulación cuántica.

Ambos modelos, el clásico y el cuántico, son esenciales para comprender fenómenos en física de la materia condensada y en el desarrollo de algoritmos cuánticos como el QAOA, donde el modelo de Ising cuántico a menudo se utiliza para formular problemas de optimización.

El modelo de Ising parte de una red regular, que imita la red cristalina, en cuyos sitios se colocan un momento magnético de espín que puede tomar exclusivamente los valores +1 ó -1. De esta forma, la interacción entre dos momentos se puede escribir con el siguiente hailtoniano:

$$\hat{H}_{ij} = -J_{ij}\hat{\sigma}_i^z\hat{\sigma}_j^z\,, \tag{3.1}$$

donde si $J_{ij} > 0$ se favorece que los espines estén alineados a largo de la dirección $z$ ya que si $\sigma_i = \sigma_j$ la energía disminuye en una cantidad $-J_{ij}$. El caso contrario, $J_{ij} < 0$, favorece que los espines se encuentren en oposición, $\sigma_i \neq \sigma_j$.

Además, si existe un campo magnético externo en el eje z $B_0$, los momentos magnéticos interaccionan con él con una energía $-\mu_i B_0\hat{\sigma}_i^z$ donde $\mu$ es el momento dipolar magnético que tendrá una orientación u otra dependiendo del valor de $\sigma_i$. El hamiltoniano total es entonces:

$$\hat{H} = -\sum_{i,j} J_{i,j}\hat{\sigma}_i^z\hat{\sigma}_j^z - \sum_i h_i\hat{\sigma}_i^z - \sum_i g_i\hat{\sigma}_i^x, \tag{3.2}$$

donde $h_i$ y $g_i$ son los campos magnéticos longitudinal y transversal. Resolviendo estos hamiltonianos, es decir, encontrando el nivel de menos energía, se han resulto a lo largo de la historia muchos problemas relacionados con la interacción de partículas.

Lo sorprendente de todo esto es que teniendo como objetivo este tipo de hamiltonianos, se pueden resolver cualquier tipo de problema combinatorio en el que las variables tomen valores binarios cualesquiera. Este tipo de problemas reciben el nombre de **Problemas de Optimización Binaria Cuadrática No Restringida**.

## 3.2. Problemas de Optimización Binaria Cuadrática No Restringida (QUBO)

Este tipo de problemas cuyo objetivo es la optimización de funciones discretas donde las variables son variables dicotómicas se conocen en inglés como **QUBO**, que son las siglas de ***Quadratic unconstrained binary optimization***, también conocido como ***unconstrained binary quadratic programming (UBQP)*** y tienen gran variedad de aplicaciones en finanzas, teoría de grafos, economía o machine learning [20].

De manera formal, este tipo de problemas queda definido de la siguiente forma: dado un conjunto de vectores binarios de longitud fija $n > 0$, es decir pertenecientes a $\mathbb{B}^n$, donde $\mathbb{B} = \{0, 1\}$ es el conjunto de valores binarios posibles, se busca además optimizar una función dada de la siguiente forma:

$$f_A(\mathbf{x}) = \mathbf{x^T A x} = \sum_{i=1}^{n}\sum_{j=i}^{n} A_{ij}x_i x_j = \sum_{i=1}^{n}\sum_{j=i+1}^{n} A_{ij}x_i x_j + \sum_{i=1}^{n} A_{ii}x_i^2 \,. \tag{3.3}$$

Para un mejor manejo de la expresión, es posible trabajar con una matriz $Q$ simétrica de forma que la expresión se pueda escribir como sigue:

$$f_Q(\mathbf{x}) = \sum_{i=1}^{n}\sum_{\substack{j=1\\ j\neq i}}^{n} Q_{ij}x_i x_j + \sum_{i=1}^{n} Q_{ii}x_i^2 \,. \tag{3.4}$$

A lo largo del documento vamos a generar estos hamiltonianos a partir de grafos por lo que la matriz con la que vamos a trabajar tendrá ceros en la diagonal. Esto se debe a que las entradas de las matrices que representan los grafos representan las uniones entre vértices y no nos interesan grafos que conecten un vértice consigo mismo. Por tanto nuestra expresión será:

$$f_Q(\mathbf{x}) = \sum_{i=1}^{n}\sum_{\substack{j=1\\ j\neq i}}^{n} Q_{ij}x_i x_j \,. \tag{3.5}$$

Finalmente, la resolución del problema QUBO consiste en encontrar un vector binario $\mathbf{x_0}$ que haga mínimo el valor de $f_Q$, es decir,

$$\forall \mathbf{x} \in \mathbb{B}^n : \quad f_Q(\mathbf{x_0}) \leq f_Q(\mathbf{x}) \,. \tag{3.6}$$

Además, el problema QUBO se puede definir también con el objetivo de maximizar $f_Q$, lo cual es equivalente a minimizar $-f_Q = f_{-Q}$.

Sin embargo, si queremos trabajar con este tipo de problemas en un ordenador cuántico donde se trabaja con valores discretos del spin ($\{-1, 1\}$) es necesario trabajar con los operadores correspondientes, es decir, las matrices de Pauli. Este cambio de variable en el que se pasa la función problema escrita en términos de bits clásicos ($\{0, 1\}$) a una formulación cuántica con la que poder trabajar recibe el nombre de **mapeo** y se realiza de la siguiente manera:

$$x_i = \frac{1}{2}\left(\mathbb{I} + \hat{\sigma}_z^i\right) \,, \tag{3.7}$$

donde $\sigma_z$ es el operador matriz de Pauli en el eje z tomando autovalores $\{-1, 1\}$ para los estados $|0\rangle$ y $|1\rangle$ respectivamente. Aplicamos este cambio a la función inicial (3.5):

$$\begin{aligned} f_Q(\mathbf{x}) &= \sum_{i=1}^{n}\sum_{j=1}^{n} Q_{ij}x_i x_j \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} \frac{Q_{ij}}{4}\left(\mathbb{I}+\hat{\sigma}_z^i\right)\left(\mathbb{I}+\hat{\sigma}_z^j\right) \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} \frac{Q_{ij}}{4}\left(\mathbb{I}+\hat{\sigma}_z^i+\hat{\sigma}_z^j+\hat{\sigma}_z^i\hat{\sigma}_z^j\right) \\ &= \left(\sum_{i=1}^{n}\sum_{j=1}^{n}\frac{Q_{ij}}{4}\right)\mathbb{I} + \sum_{i=1}^{n}\left(\sum_{j=1}^{n}\frac{Q_{ij}}{4}+\sum_{j=1}^{n}\frac{Q_{ji}}{4}\right)\hat{\sigma}_z^i + \sum_{i=1}^{n}\sum_{j=1}^{n}\left(\frac{Q_{ij}}{4}\right)\hat{\sigma}_z^i\hat{\sigma}_z^j\,. \end{aligned} \tag{3.8}$$

Como el objetivo es encontrar la secuencia de $\{-1, 1\}^n$ que minimice esa función, se puede prescindir del primer termino al ser una constante y lo mismo ocurre con el factor $1/4$ que se extrae como factor común. Teniendo todo ello en cuenta, la función con la que se va a trabajar queda de la siguiente forma.

$$f_Q = \sum_{i=1}^{n}\sum_{j=i}^{n} J_{ij}\hat{\sigma}_z^i\hat{\sigma}_z^j + \sum_{i=1}^{n} H_i\hat{\sigma}_z^i \quad \text{donde} \quad H_i = \sum_{j=1}^{n} Q_{ij} + \sum_{j=1}^{n} Q_{ji} \quad \text{y} \quad J_{i,j} = Q_{ij}\,. \tag{3.9}$$

Se llega así a un hamiltoniano tipo Ising (3.2) manejable por un ordenador cuántico.

Por otro lado, uno repara en que el número de vectores binarios que existen es $|\mathbb{B}^n| = 2^n$, de forma que la complejidad del problema crece exponencialmente con $n$. Por ello, este problema es considerado un problema de clase NP y se busca con estos nuevos algoritmos darle una solución más rápida y eficiente. [20]

# Capítulo 4

# Aplicación numérica del método

Una vez se ha visto el fundamento teórico detrás de este tipo de algoritmos vamos a realizar ciertas pruebas numéricas para comprobar y ver como desempeñan estos algoritmos. Para ello, lo primero que haremos será el caso más sencillo de todos, un único qubit sometido a una única capa de QAOA que a pesar de su aparente simplicidad, nos permite sacar muchas conclusiones extrapolables a cualquier tipo de algoritmo de este tipo.

## 4.1. Iniciación en los algoritmos QAOA: *1 qubit y 1 capa*

### 4.1.1. Primera aproximación teórica

Es conocido que los hamiltonianos tipo Ising con los que vamos a trabajar son de la siguiente forma:

$$f_Q = \sum_{i=1}^{n}\sum_{j=i}^{n} J_{ij}\hat{\sigma}_z^i\hat{\sigma}_z^j + \sum_{i=1}^{n} H_i\hat{\sigma}_z^i\,. \tag{4.1}$$

Sin embargo, al tener un único qubit, no existe ningún término $\sigma_i^z\sigma_j^z$, el hamiltoniano será más sencillo aun:

$$f_Q = H_1\hat{\sigma}_z^1 \quad \hat{H} = \begin{pmatrix} H_1 & 0 \\ 0 & -H_1 \end{pmatrix}. \tag{4.2}$$

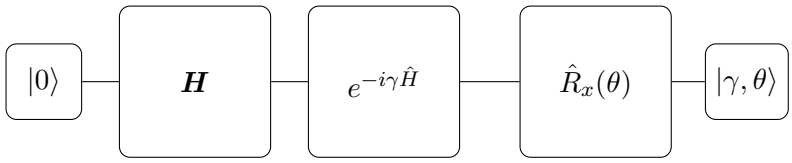
Lo primero en lo que uno repara es en la simplicidad del problema ya que se ve claramente cual es el mínimo de la función $f_Q$ (Si $H_1 > 0$, el mínimo se alcanza en en el estado $|0\rangle$ con una energía mínima de -$H_1$ y si $H_1 > 0$, con $|1\rangle$ y $E = H_1$). Esto quiere decir que con un único *if* sería suficiente para minimizar está función y no sería necesario todo el proceso cuántico. Pero esto que parece tan rápido para $N = 1$, escala exponencialmente de forma que si se tiene una función dependiente de $N = 20$ variables binarias, lo que equivale a $2^{20} = 1048576$ combinaciones posibles. Esto con los algoritmos tradicionales que se usan para ordenar muestras, tienen un orden de computación de $O(n^2)$ siendo n el número de combinaciones. En su totalidad, la complejidad de computación crece con $O(2^N)$ frente a la complejidad lineal que ofrecen los ordenadores cuánticos $O(N)$ [30].

Sea entonces nuestro Hamiltoniano, la matriz de rotación y la puerta Hadamard para este caso de un único qubit:

$$\hat{H} = \begin{pmatrix} E_{x1} & 0 \\ 0 & E_{x0} \end{pmatrix}, \quad \hat{\mathbf{H}} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \tag{4.3}$$

$$\hat{R}_x(\theta) = e^{-i\frac{\theta}{2}\sigma^x} = \mathbb{I}\cos\left(\frac{\theta}{2}\right) - i\hat{\sigma}_x \sin\left(\frac{\theta}{2}\right) = \begin{pmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}. \tag{4.4}$$

Nuestro circuito queda representado de la siguiente forma:




Y el estado final se calcula como sigue:

$$\begin{aligned}
|\gamma,\theta\rangle &= \frac{1}{\sqrt{2}} \begin{pmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix} \begin{pmatrix} e^{-iE_{x1}\gamma} & 0 \\ 0 & e^{-iE_{x0}\gamma} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\
&= \frac{1}{\sqrt{2}} \begin{pmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix} \begin{pmatrix} e^{-iE_{x1}\gamma} & 0 \\ 0 & e^{-iE_{x0}\gamma} \end{pmatrix} \begin{pmatrix} 1 \\ -1 \end{pmatrix} \\
&= \frac{1}{\sqrt{2}} \begin{pmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix} \begin{pmatrix} e^{-iE_{x1}\gamma} \\ -e^{-iE_{x0}\gamma} \end{pmatrix} \\
|\gamma,\theta\rangle &= \frac{1}{2} \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) e^{-iE_{x1}\gamma} + i\sin\left(\frac{\theta}{2}\right) e^{-iE_{x0}\gamma} \\ -\cos\left(\frac{\theta}{2}\right) e^{-iE_{x0}\gamma} - i\sin\left(\frac{\theta}{2}\right) e^{-iE_{x1}\gamma} \end{pmatrix}.
\end{aligned} \tag{4.5}$$

Esto se puede escribir en función de sus amplitudes de probabilidad finales a las que llamaremos $N_1$ y $N_0$:

$$\begin{pmatrix} N_1 \\ N_0 \end{pmatrix} = \frac{e^{-i\gamma E_{x0}}}{\sqrt{2}} \begin{pmatrix} \cos\frac{\theta}{2} + ie^{-i\gamma F_{x1,x0}}\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} - e^{-i\gamma F_{x1,x0}}\cos\frac{\theta}{2} \end{pmatrix}, \tag{4.6}$$

donde se ha definido $F_{x_1,x_0} := E_{x1} - E_{x0}$, como la diferencia de energías entre el primer y el segundo nivel. La razón por la que se saca esta fase fuera de la matriz, es para escribirlo en función de la diferencia entre los niveles de energía ya que esta fase común no altera el estado del sistema. Las probabilidades y valores medios serán los mismos.

$$\begin{aligned}
N_1^2 &= \frac{1}{2}\left(\cos\frac{\theta}{2} - ie^{i\gamma F_{x1,x0}}\sin\frac{\theta}{2}\right)\left(\cos\frac{\theta}{2} + ie^{-i\gamma F_{x1,x0}}\sin\frac{\theta}{2}\right) \\
&= \frac{1}{2}\left(\cos^2\frac{\theta}{2} + \sin^2\frac{\theta}{2} + i\cos\frac{\theta}{2}\sin\frac{\theta}{2}\left(e^{-i\gamma F_{x1,x0}} - e^{i\gamma F_{x1,x0}}\right)\right) \\
&= \frac{1}{2}\left(1 + \sin(\theta)\sin(\gamma F_{x1,x0})\right).
\end{aligned} \tag{4.7}$$

$$N_0^2 = \frac{1}{2}\left(1 - \sin(\theta)\sin(\gamma F_{x1,x0})\right). \tag{4.8}$$

donde se han usado la fórmula del ángulo doble $\sin(2x) = \frac{1}{2}\sin(x)\cos(x)$ y la expresión del seno como resta de exponenciales complejas $\sin(x) = \frac{e^{ix}+e^{-ix}}{2}$.

Una vez se tiene el estado final, el objetivo es minimizar la función de coste que equivale a minimizar la energía media:

$$\langle\gamma,\theta|\, H\,|\gamma,\theta\rangle = \begin{pmatrix} N_1 & N_2 \end{pmatrix} \begin{pmatrix} a & 0 \\ 0 & -a \end{pmatrix} \begin{pmatrix} N_1 \\ N_2 \end{pmatrix} = aN_1^2 - aN_0^2 = a\left(1 + \frac{1}{2}\sin(\theta)\sin(2\gamma F_{x1,x0})\right) . \quad (4.9)$$

Lo primero que nos damos cuenta antes de resolver este problema de optimización en dos variables es que minimizar la energía en este caso es equivalente a maximizar la probabilidad de que el estado final este en el estado base o *ground state*. Esto es lógico ya que este estado es el de mínima energía, entonces si se encuentra en este estado, la energía del sistema será la mínima posible.

Esto nos permite ver un resultado clave que se generaliza fácilmente a cualquier tamaño de sistema cuántico siempre y cuando la matriz que representa el Hamiltoniano sea diagonal (esta hipótesis que parece muy estricta ocurre siempre para los problemas QUBO que se tratan): a la hora de buscar los ángulos $\theta$ y $\gamma$ que minimicen nuestra función de coste es equivalente maximizar la probabilidad de que el sistema se encuentre en el *ground state*.

A pesar de esta aparente ventaja, para llevar a cabo la optimización numérica de la probabilidad de que el sistema este en estado de mínima energía es necesario conocer previamente cual es ese estado y por ello, los algoritmos van destinados a minimizar la energía.

Esto no es sorprendente ya que el objetivo de los algoritmos adiabáticos es a grandes rasgos, llevar lentamente un estado de mínima energía de un Hamiltoniano conocido a otro estado también de mínima energía pero en este caso desconocido. Todo ello con el objetivo de averiguar esa energía. Por eso, si conociéramos inicialmente ese estado final al que queremos llegar ya no sería necesario hacer nada porque la energía asociada a este estado es lo que se busca.

Por otro lado, entre las variables a optimizar encontramos $\theta$ que es un ángulo de giro. Esto quiere decir que, al menos, podemos reducir nuestro intervalo de estudio que inicialmente sería $\mathbb{R}^2$ al intervalo $\mathbb{R} \times [0, 2\pi]$. El caso del término $\gamma$ es algo diferente ya que viene acompañado de la diferencia de energías entre el nivel máximo y mínimo: $\gamma F_{x_1,x_0} = \gamma(E_{x_1} - E_{x_0})$. Por eso, se puede trabajar en el intervalo $\gamma \in [0, \frac{2\pi}{F_{x_1,x_0}}]$.

Si a esto le añadimos que para maximizar la energía, los senos deberían tener signos contrarios, es decir, dentro del intervalo dado anteriormente $[0, 2\pi] \times [0, \frac{2\pi}{F_{x_1,x_0}}]$ hay dos valores donde se alcanza el máximo. Esto nos permite reducir aún más nuestro intervalo y hacer más eficiente la optimización:

$$n\pi \leq \theta < (n+1)\pi \quad \text{y} \quad |\gamma| < \frac{\pi}{F_{x_0,x_1}} \sim O(\|F_{x_0,x_1}\|^{-1}) \quad con \quad n \in \mathbb{Z} . \quad (4.10)$$

Esto ha sido una primera aproximación de la importancia de la limitación de los intervalos con el fin de aumentar la eficacia. La complejidad de este tipo de análisis varía en función de la complejidad del sistema. Léase [6, 3] para mayor información sobre este tema.

Pasamos ahora a la optimización explicita de la energía media:

$$\frac{\partial\langle H\rangle}{\partial\gamma} = 2F_{x_0,x_1}\sin(\theta)\cos(2\gamma F_{x_0,x_1}) = 0 , \quad (4.11)$$

$$\frac{\partial\langle H\rangle}{\partial\theta} = \cos(\theta)\sin(2\gamma F_{x_0,x_1}) = 0\,, \tag{4.12}$$

Ambas ecuaciones se pueden anular cuando se anulen los senos simultáneamente $\sin(\theta) = \sin(2\gamma F_{x_0,x_1}) = 0$ que corresponde a $(\theta,\gamma) = (0,0)$ o cuando se anulen los cosenos $\cos(\theta) = \cos(2\gamma F_{x_0,x_1}) = 0$ que corresponde a dos puntos $(\theta,\gamma) = (\frac{\pi}{2}, \frac{\pi}{4F_{x_0,x_1}})$ o $(\theta,\gamma) = (\frac{\pi}{2}, -\frac{\pi}{4F_{x_0,x_1}})$. La opción de los senos se puede despreciar ya que anula también la parte variable de la función por lo que no representa ni un máximo ni un mínimo. Nos quedamos con dos opciones y como se ha dicho anteriormente, para minimizar la energía es necesario que los senos sean de signo contrario, por tanto, optamos lo opción $(\theta,\gamma) = (\frac{\pi}{2}, -\frac{\pi}{4F})$.

Estos resultados teóricos pueden parecer no ser útiles ya que se va a llevar una optimización numérica para los distintos escenarios. Sin embargo, nos permite llegar a varias conclusiones importantes que nos serán útiles en situaciones con mayor número de capas y qubits:

- A la hora de optimizar numéricamente la función, se puede reducir el intervalo de trabajo lo que hará más rápido este proceso. En concreto, el intervalo de los valores de $\theta$ que no dependen de la diferencia de energías entre niveles más alejados.
- En este mismo sentido pero con respecto al factor $\gamma$, nos dice que si las energías son muy similares entre si, será necesario un mayor rango de valores de $\gamma$ que si las energías difieren enormemente. Esto quiere decir que si por ejemplo tenemos algún grafo con pesos muy pequeños, sería recomendable aumentar los intervalos de trabajo.
- Casi todos los métodos numéricos de optimización requieren de unas condiciones iniciales. Estas habrá que elegirlas con mucho cuidado ya que en caso de ser estas por ejemplo (0,0) para este sistema, el ordenador lo identificará como un mínimo y no obtendremos ningún resultado concluyente.

Una vez visto esto, contrastaremos las conclusiones teóricas con resultados numéricos generados con ordenador por nosotros mismos.

### 4.1.2. Desarrollo computacional

Se ha trabajado para distintos valores de $a := E_{x1} = -E_{x0}$ que representan los niveles de energía. Para cada uno de ellos, representamos en función de los ángulo $\gamma$ y $\theta$ los valores finales de las probabilidades en el estado de mínima energía y el estado excitado junto con el valor de la energía esperado. Los resultados obtenidos se pueden visualizar en las Figuras 4.1 y 4.2.

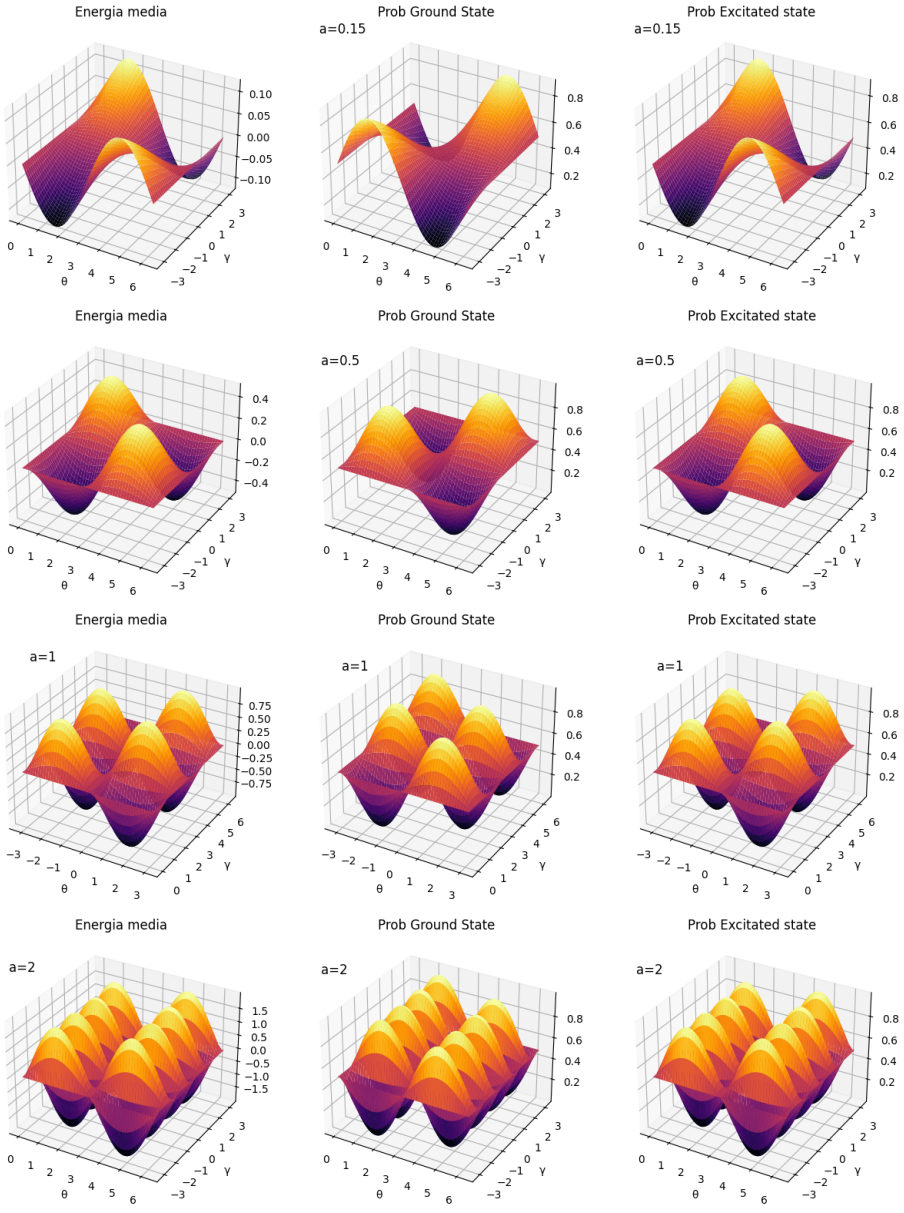



Figura 4.1: Representación en tres dimensiones de la energía media, de la probabilidad de que el estado final se encuentre en el estado base y de la probabilidad de que el estado acabe en el estado excitado, todo ello, en las columnas uno, dos y tres respectivamente. A su vez, se ha representado para diferentes valores de a para visualizar el contraste.

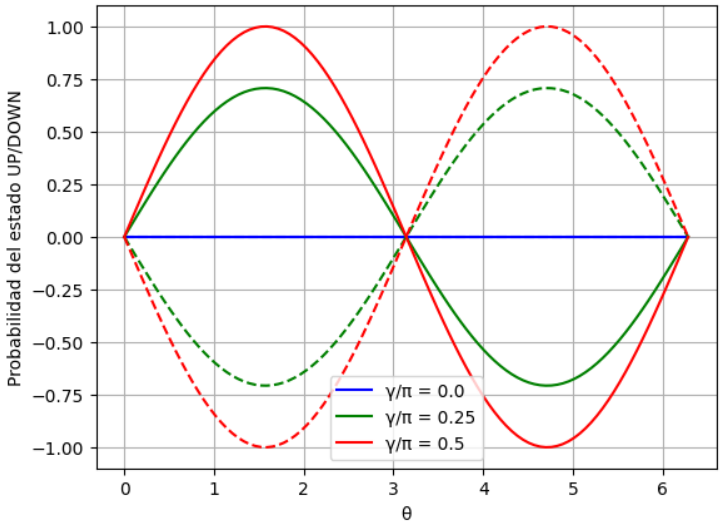



Figura 4.2: Representación de las probabilidades de encontrar al estado final $|\gamma, \theta\rangle$ en los estados $|0\rangle$ (rayado) o $|1\rangle$ (en continua) en función del ángulo $\theta$ para tres valores distintos de $\gamma$. Esto nos permite ver esa gran relación que hay entre magnitudes representadas.

Lo primero que uno repara es que, evidentemente, al tener un único qubit con dos estados posibles cuya probabilidad tiene que sumar uno $1 = |\langle 0|\gamma, \theta\rangle|^2 + |\langle 1|\gamma, \theta\rangle|^2$, da lugar a que las gráficas de la columna 2 en la Figura 4.1 sean simétricas a las gráficas de la columna 3 de la Figura 4.1. También nos fijamos en que las gráficas de la Figura 4.1 en las que se representa la energía son idéntica a las del primer estado excitado salvo en un factor de escala.

Otra cosa de menor importancia, es que las gráficas no varían respecto al valor de $\theta$ en ninguno de los casos ya que este es una rotación fija que no depende de los niveles de energía del hamiltoniano.

Por otro lado, se ve claramente la importancia del intervalo de trabajo. Por ejemplo, para $a = 0{,}15$ (primera fila), se tiene que $F_{x_0,x_1} = 2a = 0{,}30$ y por tanto $\frac{\pi}{2F_{x_0,x_1}} > \pi$. Esto implica que el máximo no se encuentra dentro del intervalo en $\gamma$ por lo que no se llegaría al resultado deseado.

En el otro lado de la balanza, se encuentra el caso $a = 2$ (fila cuarta). El intervalo de las $\gamma$ se podría haber reducido enormemente y haber hecho muchos menos cálculos de los que se han llevado a cabo. Esto, para este caso de un solo qubit puede parecer una tontería porque el tiempo de computación es prácticamente inapreciable, pero en el momento en que el número de qubits comience a aumentar puede resultar en una gran perdida de eficiencia.

Todo esto es fruto de que el operador unitario de la exponencial del hamiltoniano, es al fin y al cabo el operador temporal de la mecánica cuántica, es decir, hay que hacer que evolucione un tiempo suficiente para que pueda alcanzar el máximo pero a su vez, es bueno que el tiempo de evolución no sea demasiado alto como para perder eficiencia.

Si nos centramos ahora en el caso $a = 0{,}5$ , la Figura 4.3 recoge los resultados del algoritmo QAOA para diferentes condiciones iniciales en el proceso de optimización. Se puede observar como el algoritmo de optimización nos lleva siempre a ese punto máximo calculado teóricamente salvo en el caso que $\gamma$ o $\theta$ sean nulos inicialmente o iguales a la unidad.

| Condiciones iniciales ($\gamma$, $\theta$) | $\gamma_{opt}$ | $\theta_{opt}$ | $\|\langle 0\|\psi_{final}\rangle\|^2$ | $\|\langle 1\|\psi_{final}\rangle\|^2$ | $E_{final}$ |
|---|---|---|---|---|---|
| (0,0) | 0.0 | 0.0 | 0.5 | 0.5 | 0.0 |
| (1,0) | 0.0 | 1.0 | 0.5 | 0.5 | 0.0 |
| (2,0) | 0 | 1.0 | 0.5 | 0.5 | 0.0 |
| (1,1) | -1.57079633 | 1.57079633 | 1.0 | $3.11\text{x}10^{-18}$ | -0.5000 |
| (2,1) | 1.57079633 | 0.0 | 0.5 | 0.5 | 0.0 |
| (3,3) | 1.57079633 | 3.14159265 | 0.5 | 0.5 | 0.0 |

Figura 4.3: Resultados obtenidos para la optimización numérica llevada a cabo por ordenador con el método *'L-BFGS-B'* para condiciones iniciales controladas

Estos resultados nos indican que las condiciones iniciales determinan enormemente el llegar o no al resultado correcto. Para corroborar este hecho, buscamos llevar a cabo una serie de pruebas con condiciones iniciales aleatorias. Estas se recogen en la Figura 4.4.

| Condiciones iniciales ($\gamma$, $\theta$) | $\gamma_{opt}$ | $\theta_{opt}$ | $E_{final}$ | $\|\langle 0\|\psi_{final}\rangle\|^2$ | $\|\langle 1\|\psi_{final}\rangle\|^2$ |
|---|---|---|---|---|---|
| (2.0822, 0.5356) | 1.5708 | 0 | 0 | 0.5 | 0.5 |
| (2.2620, 2.7922) | 1.5708 | 3.1416 | 0 | 0.5 | 0.5 |
| (1.1856, 1.7782) | -1.5708 | 1.5708 | -0.5 | 1 | 0 |
| (0.5306, 1.8324) | -1.5708 | 1.5708 | -0.5 | 1 | 0 |
| (5.8551, 0.0238) | 1.5708 | 0 | 0 | 0.5 | 0.5 |
| (5.8087, 2.8972) | 1.5708 | 3.1416 | 0 | 0.5 | 0.5 |
| (0.8718, 1.7175) | -1.5708 | 1.5708 | -0.5 | 1 | 0 |
| (0.0944, -0.6425) | 0.0944 | 0 | -0.5 | 1 | 0 |

Figura 4.4: Resultados obtenidos para la optimización numérica llevada a cabo por ordenador con el método *'L-BFGS-B'* para condiciones iniciales aleatorias dentro de los límites estudiados

Y ahora, aún habiendo elegido las condiciones iniciales de manera aleatoria, solo en una pequeña parte de los casos se obtiene el resultado deseado. Esto que ahora nos resulta muy irrelevante ya que identificar que resultados podemos tomar por buenos y cuales no es muy sencillo al conocer la teoría, en casos más complejos nos obliga a ser crítico con los resultados obtenidos y determinar si nos son concluyentes o no. Como recomendación respecto a este tema, ya que nuestro estudio no se basa en la comprobación del funcionamiento de la optimización en este tipo de algoritmos, se recomiendo hacer un primer barrido que mezcle tanto tipos de optimización numérica como condiciones iniciales para que el método elegido sea lo mas fiable posible.

## 4.2. Algoritmos QAOA para un mayor número de qubits: $N$ qubits and 1 layer

### 4.2.1. Aproximación teórica

Sea un ordenador cuántico compuesto por $N$ qubits y denotamos por $F(\mathbf{x})$ a la amplitud de probabilidad del estafo final $|\gamma, \theta\rangle$ con respecto al estado $\mathbf{x}$, siendo este uno de los estados de la base

computacional:

$$F(\mathbf{x}) \equiv \langle \mathbf{x}|\gamma,\theta\rangle \ . \tag{4.13}$$

Este estado final se escribe como

$$|\gamma,\theta\rangle = \hat{R}_x(\theta)\ e^{i\gamma\hat{C}}\ |\mathbf{s}\rangle \ , \tag{4.14}$$

donde $|\mathbf{s}\rangle = \frac{1}{2^{N/2}}\sum_{\mathbf{x}'}|\mathbf{x}'\rangle$, $\mathbf{x}'$ representa cada uno de los elementos de la base computacional, y $\hat{C}$ representa el hamiltoniano problema. Si además tenemos en cuenta que el hamiltoniano Ising es diagonal en la base computacional y por tanto, su exponencial también lo será, se tiene que:

$$|\gamma,\theta\rangle = \hat{R}_x(\theta)\ e^{i\gamma\hat{H}}\frac{1}{2^{N/2}}\sum_{\mathbf{x}'}|\mathbf{x}'\rangle = \sum_{\mathbf{x}'}\frac{e^{-i\gamma E_{x'}}}{2^{N/2}}\left(\prod_{j=1}^{N}\mathbb{I}\cos(\theta)+\hat{\sigma}_x^j\sin(\theta)\right)|\mathbf{x}'\rangle \ . \tag{4.15}$$

Y finalmente se llega a

$$F(\mathbf{x}) = \sum_{\mathbf{x}'}\frac{e^{-i\gamma E_{x'}}}{2^{N/2}}\langle\mathbf{x}|\prod_{j=1}^{N}\mathbb{I}\cos(\theta)-i\hat{\sigma}_x^j\sin(\theta)\,|\mathbf{x}'\rangle \ . \tag{4.16}$$

A partir de aquí, usaremos un concepto conocido como ***Distancia de Hamming*** o en inglés ***Hamming distance***.

## HAMMING DISTANCE

La **distancia de Hamming**, nombrada así en honor al matemático estadounidense *Richard Hamming*, es una de las varias métricas que miden la distancia entre dos secuencias.

Se define como el número de posiciones en las que difieren los símbolos correspondientes de dos cadenas o vectores de igual longitud. En otras palabras, representa el mínimo de sustituciones necesarias para transformar una cadena en otra.

La notación que usaremos para la distancia entre dos cadenas, *cadena 1* y *cadena 2*, es $H_{cadena1,cadena2}$ y para entenderlo será útil ver varios ejemplos:

- **"MIGUEL"** y **"MANUEL"**: es importante que las palabras tengan igual longitud, este caso, de las cinco letras, difieren en dos y por tanto $H_{MIGUEL,MANUEL} = 2$.
- **"25818935"** Y **"25328995"**: en este caso, nos damos cuenta que lo único que nos importa para calcular la distancia de Hamming es la posición de los caracteres. Por ejemplo, ambas cadenas tienen un 3, pero es indiferente ya que nos fijamos en que si en la posición del primer 3 hay otro 3 o no. Además, el valor numérico no tiene ninguna relevancia, solo nos importa si coinciden o no. Por todo esto, en este caso, $H_{25818935,25328995} = 4$.
- El último ejemplo será el que usaremos nosotros. Al estar los estados de la base computacional definidos por una secuencia de unos y ceros, se puede definir una distancia entre estados. Sea por ejemplo un ordenador cuántico con 8 qubits y sean dos estados cualesquiera $\mathbf{x}' = |\mathbf{10001101}\rangle$ y $\mathbf{x}' = |\mathbf{01100101}\rangle$, se tiene que $H_{\mathbf{x},\mathbf{x}'} = 4$.

Una propiedad sobre esta distancia que usaremos más adelante consiste en la caracterización de la distancia de Hamming para una concatenación de dos o más cadenas en función de las distancias individuales:

*Aditividad de las distancias de Hamming:* sean n pares de cadenas de caracteres a las que denotamos por $(\mathbf{x_i}, \mathbf{x'_i})$ con $i \in \mathbb{N}$ y donde $\mathbf{x_i}$ y $\mathbf{x'_i}$ son cadenas de longitud $l_i$ con distancia de Hamming $H_{\mathbf{x_i},\mathbf{x'_i}}$ respectivamente. La cadena resultante de la concatenación de todas ellas en un mismo orden, $\mathbf{x} = \mathbf{x_1 x_2 ... x_n}$ y $\mathbf{x'} = \mathbf{x'_1 x'_2 ... x'_n}$, cumple que tiene longitud $L = \sum_{i=1}^{n} l_i$ y su distancia de Hamming $H_{\mathbf{x},\mathbf{x'}} = \sum_{i=1}^{n} H_{\mathbf{x_i},\mathbf{x'_i}}$.

Teniendo esto propiedad en cuenta, se puede reescribir la amplitud de probabilidad de la siguiente forma:

$$\begin{aligned} F(\mathbf{x}) &= \sum_{\mathbf{x'}} \frac{e^{-i\gamma E_{x'}}}{2^{N/2}} \langle \mathbf{x}| \prod_{j=1}^{N} \mathbb{I}\cos(\theta) - i\hat{\sigma}_x^j \sin(\theta) |\mathbf{x'}\rangle \ , \\ &= \frac{1}{2^{N/2}} \sum_{\mathbf{x'}} e^{-i\gamma E_{x'}} \left(\cos(\theta)\right)^{N - H_{\mathbf{x},\mathbf{x'}}} \left(-i\sin(\theta)\right)^{H_{\mathbf{x},\mathbf{x'}}} \ . \end{aligned} \tag{4.17}$$

Para ver de donde viene este último paso, es necesario demostrar la siguiente igualdad:

$$\langle \mathbf{x}| \prod_{j=1}^{N} \mathbb{I}\cos(\theta) - i\hat{\sigma}_x^j \sin(\theta) |\mathbf{x'}\rangle = \left(\cos(\theta)\right)^{N - H_{\mathbf{x},\mathbf{x'}}} \left(-i\sin(\theta)\right)^{H_{\mathbf{x},\mathbf{x'}}} \ . \tag{4.18}$$

*Demostración*

Comenzamos demostrándolo para $N = 1$ y teniendo en cuenta que para este caso $H_{\mathbf{x},\mathbf{x'}} = \delta_{\mathbf{x}\mathbf{x'}}$:

$$\begin{aligned} \langle \mathbf{x}| \mathbb{I}\cos(\theta) - i\hat{\sigma}_x^1 \sin(\theta) |\mathbf{x'}\rangle &= \cos(\theta) \langle \mathbf{x}|\mathbf{x'}\rangle - i\sin(\theta) \langle \mathbf{x}| \hat{\sigma}_x^1 |\mathbf{x'}\rangle \ , \\ &= \cos(\theta)\delta_{\mathbf{x}\mathbf{x'}} - i\sin(\theta)(1 - \delta_{\mathbf{x}\mathbf{x'}}) \, , \end{aligned} \tag{4.19}$$

donde se ha usado que

$$\hat{\sigma}_x^j |1\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |0\rangle \ , \tag{4.20}$$

$$\hat{\sigma}_x^j |0\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = |1\rangle \ , \tag{4.21}$$

$$\langle \mathbf{x}|\mathbf{x'}\rangle = \delta_{\mathbf{x}\mathbf{x'}} \, . \tag{4.22}$$

Se observa fácilmente la conclusión a la que queríamos llegar:

Si $\mathbf{x} = \mathbf{x'}$ $\quad \cos(\theta)\delta_{\mathbf{x}\mathbf{x'}} - \mathbf{i}\sin(\theta)(\mathbf{1} - \delta_{\mathbf{x}\mathbf{x'}}) = \cos(\theta) = \left(\cos(\theta)\right)^{\mathbf{1}-\mathbf{H}_{\mathbf{x},\mathbf{x'}}} \left(-\mathbf{i}\sin(\theta)\right)^{\mathbf{H}_{\mathbf{x},\mathbf{x'}}}$ .

Si $\mathbf{x} \neq \mathbf{x'}$ $\quad \cos(\theta)\delta_{\mathbf{x}\mathbf{x'}} - \mathbf{i}\sin(\theta)(\mathbf{1} - \delta_{\mathbf{x}\mathbf{x'}}) = -\mathbf{i}\sin(\theta) = \left(\cos(\theta)\right)^{\mathbf{1}-\mathbf{H}_{\mathbf{x},\mathbf{x'}}} \left(-\mathbf{i}\sin(\theta)\right)^{\mathbf{H}_{\mathbf{x},\mathbf{x'}}}$ .

Pasamos ahora a demostrarlo para $N$ arbitrario. Usaremos en este caso la descripción tensorial de la base computacional $|\mathbf{x}\rangle = |x_1\rangle \otimes |x_2\rangle \otimes ... \otimes |x_N\rangle$ y que el operador $\hat{\sigma}_x^j$ actúa únicamente sobre el elemento j de la base computacional. Por tanto:

$$\begin{aligned}
\langle \mathbf{x}| \prod_{j=1}^{N} \mathbb{I}\cos(\theta) - i\hat{\sigma}_x^j \sin(\theta) \, |\mathbf{x}'\rangle &= \prod_{j=1}^{N} \langle x_j| \, \mathbb{I}\cos(\theta) - i\hat{\sigma}_x^j \sin(\theta) \, |x_j\rangle \ , \\
&= \prod_{j=1}^{N} \left(\cos(\theta)\right)^{1-H_{x_j,x'_j}} \left(-i\sin(\theta)\right)^{H_{x_j,x'_j}} \ , \\
&= \left(\cos(\theta)\right)^{\sum_{j=1}^{N} 1-H_{x_j,x'_j}} \left(-i\sin(\theta)\right)^{\sum_{j=1}^{N} H_{x_j,x'_j}} \ , \\
&= \left(\cos(\theta)\right)^{N-\sum_{j=1}^{N} H_{x_j,x'_j}} \left(-i\sin(\theta)\right)^{\sum_{j=1}^{N} H_{x_j,x'_j}} \ , \\
&= \left(\cos(\theta)\right)^{N-H_{\mathbf{x},\mathbf{x}'}} \left(-i\sin(\theta)\right)^{H_{\mathbf{x},\mathbf{x}'}} .
\end{aligned}$$

□

Finalizada esta demostración, es posible finalmente escribir la amplitud de probabilidad como sigue:

$$F(\mathbf{x}) = \frac{1}{2^{N/2}} \sum_{\mathbf{x}'} e^{-i\gamma E_{x'}} \left(\cos(\theta)\right)^{N-H_{\mathbf{x},\mathbf{x}'}} \left(-i\sin(\theta)\right)^{H_{\mathbf{x},\mathbf{x}'}} . \tag{4.23}$$

El siguiente paso consiste en hacer un cambio de variable. Para ello, y teniendo en cuenta lo visto en el caso de un qubit y una sola capa, en el que era posible reducir el intervalo de trabajo de $\theta_{opt}$, podemos escribir de forma única las funciones trigonométricas como

$$\cos(\theta) = e^{r/2} \quad \text{y} \quad \sin(\theta) = e^{-r/2} , \tag{4.24}$$

$$F(\mathbf{x}) = \left(\frac{e^r}{2}\right)^{N/2} \sum_{\mathbf{x}'} e^{\left[-H_{xx'}\left(i\frac{\pi}{2}+r\right)-i\gamma E_{\mathbf{x}'}\right]} . \tag{4.25}$$

De esta forma, tenemos escrito el estado final del sistema en función de todos los estados de la base computacional codificados en función de sus energías y de las distancia de Hamming entre ellas.

Para seguir con el estudio del sistema es conveniente introducir una distribución de probabilidad que dado un estado fijo $\mathbf{x}$ relacione las distancias en la base computacional con el espectro de energía.

$$\rho(H, E|\mathbf{x}) = \frac{1}{2^N} \sum_{\mathbf{x}'} \delta(H - H_{\mathbf{x}\mathbf{x}'}) \delta(E - E_{\mathbf{x}'}) . \tag{4.26}$$

Esta función de densidad es una función de densidad discreta que se obtiene simplemente añadiendo los puntos que cumplan igualdad de energía y distancia. Representa así el número relativo de estados de la base computacional que, dada un estado de referencia $\mathbf{x}$, tienen una distancia de Hamming $H$ con respecto a ese estado y una energía igual a $E$. En otras palabras, teniendo en

cuenta todas las distribuciones correspondientes a todos los $\mathbf{x}$ se captura las correlaciones internas entre estados y energías que tiene un determinado modelo tipo Ising.

La idea principal al introducir esta densidad es estudiar como se comporta de forma que al pasar a sistemas de grandes dimensiones, pueda sustituirse por una distribución continua y sacar conclusiones importantes con lo que respecta a las amplitudes de probabilidad.

Además, al tratarse de una función compuesta por deltas de Dirac, la función $F(\mathbf{x})$ (4.25) puede reescribirse de la siguiente forma:

$$F(\mathbf{x}) = \left(\frac{e^r}{2}\right)^{N/2} \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{\left[-H\left(i\frac{\pi}{2}+r\right)-i\gamma E\right]}\rho(H,E|\mathbf{x})\,dH\,dE\,. \tag{4.27}$$

Por tanto, para obtener conclusiones sobre la distribuciones de las probabilidades, lo primero será analizar como se comportan esas funciones de densidad $\rho(H,E|\mathbf{x})$[7, 8].

**Estudio de la distribución de probabilidad de energías y distancias de Hamming.**

Trabajaremos en un sistema de 10 qubits, generando un hamiltoniano tipo Ising aleatorio. Para ello, en primer lugar generamos un grafo sin pesos en las aristas para posteriormente añadirles un peso según una distribución normal de media 0 y desviación típica igual a 1 ($\mathcal{N}(\mu,\sigma^2)=\mathcal{N}(0,1)$). Posteriormente, le añadimos un vector aleatorio procedente de la misma distribución para los términos independientes del hamiltoniano.

Una vez tenemos el hamiltoniano, que sera una matriz diagonal de orden $2^{10} = 1024$, nos centraremos en la función de probabilidad $\rho(H,E|\mathbf{x})$ donde tomaremos $\mathbf{x}=\mathbf{x_0}$ como el estado base o ground state ya que este es el estado de mínima energía y al fin y al cabo, es este el que nos interesa estudiar. Se obtiene la siguiente distribución mostrada en la Figura 4.5 para el caso no degenerado. (El caso de un hamiltoniano degenerado es algo más complejo de estudiar y requeriría de un desarrollo considerablemente más extenso).

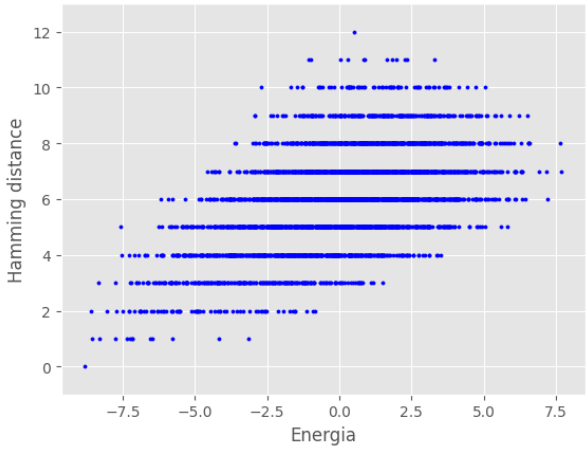



Figura 4.5: Representación de la distribución de los pares $(E_{\mathbf{x},\mathbf{x}'}, H_{\mathbf{x},\mathbf{x}'})$ donde $\mathbf{x}$ representa el ground state para un hamiltoniano generado aleatoriamente en el que no hay degeneración.

Primero de todo, es evidente que la distribución se encuentra totalmente discretizada al tratarse de la distancia de Hamming que toma únicamente valores entero entre 0 (para el propio auto estado) y 10 (para el estado del que más se diferencia). También nos damos cuenta en que para cada valor fijo en el eje de las y's, es decir, para cada distancia de Hamming, las energías están distribuidas como si se tratará de una distribución normal cuya media aumenta conforme aumenta la distancia.

El siguiente paso que se puede dar consiste en estimar la distribución empírica que siguen esta serie de datos. Esto se muestra en la Figura 4.6 y nos permite visualizarlo tener una visión más amplia al tratarse de una distribución continua.

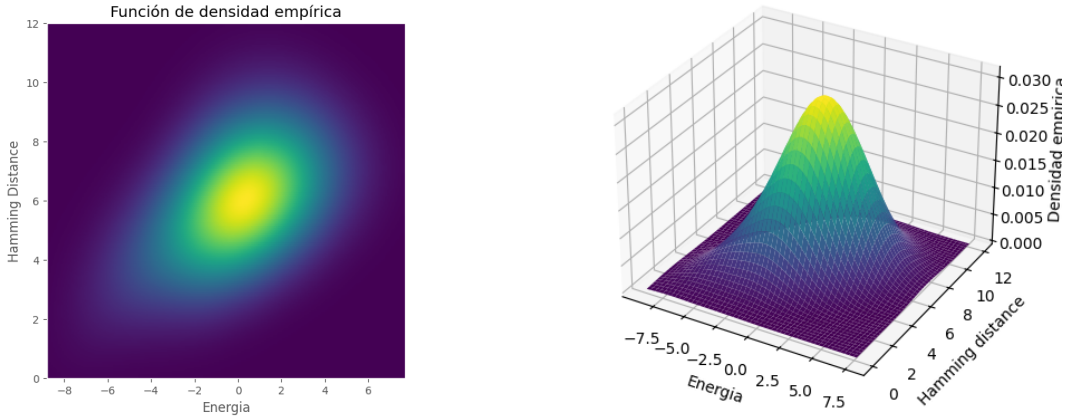



Figura 4.6: A la izquierda, la representación de la distribución continua estimada a través de los pares $(E_{\mathbf{x},\mathbf{x}'}, H_{\mathbf{x},\mathbf{x}'})$ donde $\mathbf{x}$ representa el ground state para un hamiltoniano generado aleatoriamente en el que no hay degeneración y a la derecha la representación en tres dimensiones de la distribución continua estimada a través de los pares $(E_{\mathbf{x},\mathbf{x}'}, H_{\mathbf{x},\mathbf{x}'})$ donde $\mathbf{x}$ representa el ground state para un hamiltoniano generado aleatoriamente en el que no hay degeneración .

Se puede observar en el mapa de color como forman elipses concéntricas algo difuminadas al tratarse al fin y al cabo de una densidad obtenida a través de unos datos discretos. Esta forma nos recuerda a una distribución normal bivariante que tiene la siguiente forma:

$$f_{\mathbf{X}}(\mathbf{x}) = \frac{1}{2\pi|\mathbf{\Sigma}|^{\mathbf{1/2}}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\top}\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu})\right), \tag{4.28}$$

donde en este caso $x = (E, H)^T$ y la matriz de covarianzas es:

$$\begin{pmatrix} \sigma_E^2 & \sigma_{EH} \\ \sigma_{EH} & \sigma_H^2 \end{pmatrix} \tag{4.29}$$

Si definimos también el coeficiente de correlación como $\rho = \frac{\sigma_{EH}}{\sigma_E \sigma_H}$, podemos llegar a una expresión más manejable:

$$\rho(E, H|\mathbf{x}) = \frac{1}{2\pi\sigma_E\sigma_H\sqrt{1-\rho^2}} \exp\left(-\frac{1}{2(1-\rho^2)}\left(\frac{(E-\mu_E)^2}{\sigma_E^2} + \frac{(H-\mu_H)^2}{\sigma_H^2} - \frac{2\rho(E-\mu_E)(H-\mu_H)}{(\sigma_E\sigma_H)}\right)\right). \tag{4.30}$$

Es conocido que al tratarse de un Hamiltoniano generado aleatoriamente con media cero, la media de las energías será $\mu_E = 0$. Por otro lado, la media y desviación de las distancia de Hamming, no depende del Hamiltoniano, sino que son fijas dependen únicamente del número de quibits N. Estas son $\mu_H = \frac{N}{2}$ y $\sigma_H = \frac{\sqrt{N}}{2}$. Por tanto, la función de densidad a la que queremos aproximar nuestros datos será la siguiente, con $\sigma_E$ desconocido:

$$\rho(E, H|\mathbf{x}) = \frac{1}{\pi\sigma_E\sqrt{N}\sqrt{1-\rho^2}} \exp\left(-\frac{1}{2(1-\rho^2)}\left(\frac{E^2}{\sigma_E^2} + \frac{2(H-N/2)^2}{\sqrt{N}} - \frac{4\rho E(H-N/2)}{(\sigma_E\sqrt{N})}\right)\right). \tag{4.31}$$

Para corroborar que se cumple esto ajustamos los datos a un modelo Gaussiana Bivariante, obteniendo los siguientes datos para la media y covarianzas estimados:

$$\mu = \begin{pmatrix} 0{,}0 & 5{,}0 \end{pmatrix} \quad \mathbf{\Sigma} = \begin{pmatrix} 16{,}23630141 & 2{,}96835055 \\ 2{,}96835055 & 2{,}49756673 \end{pmatrix} \tag{4.32}$$

Visualmente, en la Figura 4.7 se superponen estos datos con unos generados aleatoriamente según esta distribución gaussiana obtenida. Esto nos permite cerciorarnos de ese parecido.

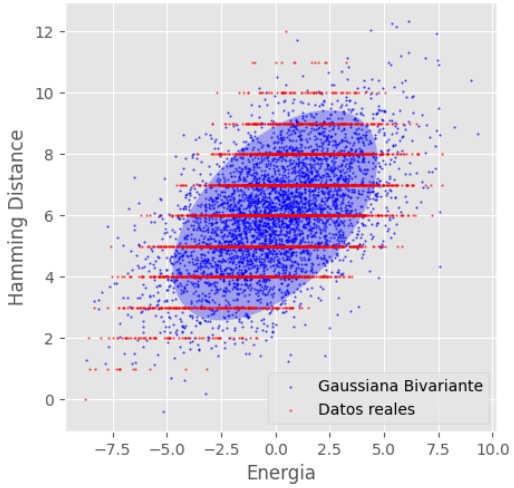



Figura 4.7: Representación conjunta de los datos reales *(en rojo)* frente a los datos de la distribución Gaussiana bivariante *(en azul)*. Se representa también la elipse de confianza al 90 % de la distribución Gaussiana.

Sin embargo, esta visualización no es suficiente para poder afirmar que se trata en si de una distribución Gaussiana multivariante. Para ello, vamos a trabajar con **la distancia de de Mahalanobis** que, como se menciona en el apéndice correspondiente, mide la distancia de cada uno de los puntos al centro de masas, es decir, a la media de la supuesta distribución Gaussiana. En caso afirmativo, la distribución de estas distancias se a de corresponder con una distribución $\chi^2_2(p)$. Para comprobar esto, realizamos el histograma de las distancias asociadas a nuestro datos y lo comparamos con el de la distribución Gaussiana teórica estimada. Este análisis se muestra en la Figura 4.8, Figura 4.9 y Figura 4.10 donde se ha incluido una tercera distribución correspondiente a la distribución uniforme para tener una referencia de lo que podría ocurrir en caso de que nuestras hipótesis no fueran correctas.

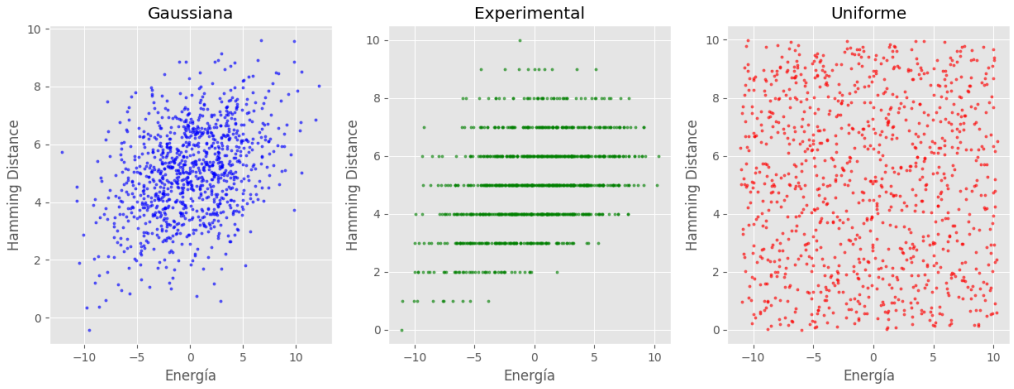



Figura 4.8: De izquierda a derecha: muestreo aleatorio de la distribución Gaussiana bivariante estimada a partir de nuestros datos simulados, nuestros datos simulados y muestreo aleatorio de una distribución uniforme con el objetivo de ver el contraste.

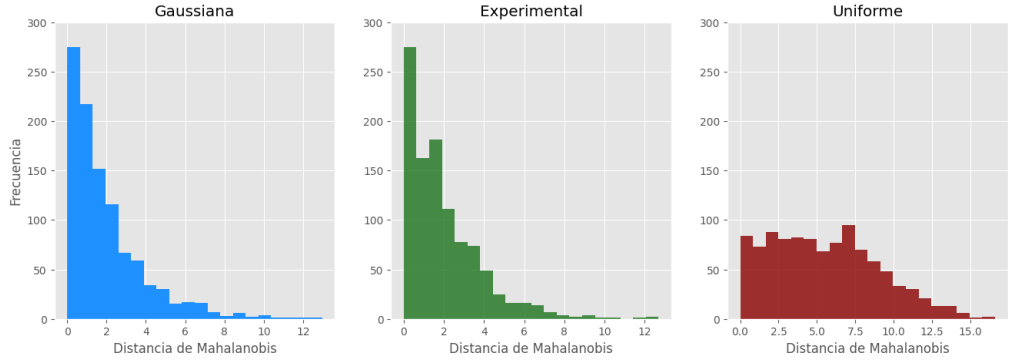



Figura 4.9: De izquierda a derecha: histograma de la distancia de de Mahalanobis de la distribución Gaussiana bivariante estimada a partir de nuestros datos simulados, histograma de la distancia de de Mahalanobis de nuestros datos simulados y histograma de la distancia de de Mahalanobis de una muestra de una distribución uniforme con el objetivo de ver el contraste.

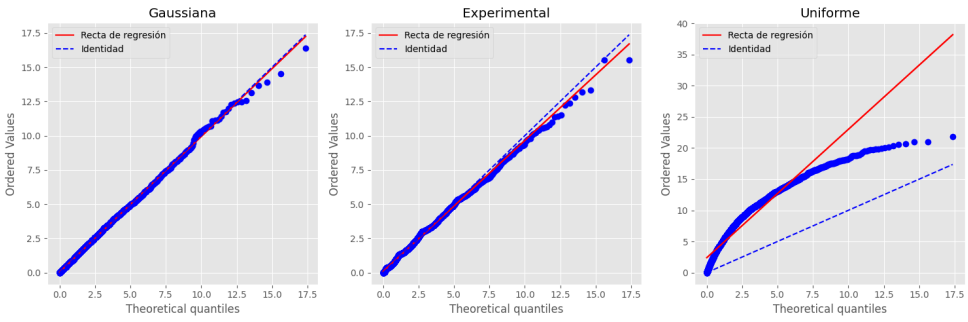



Figura 4.10: Representación de los cuantiles de las figuras anteriores comparados a su vez con los cuantiles de la distribución $\chi^2_2(p)$ en discontinua. De izquierda a derecha: distribución Gaussiana bivariante estimada a partir de nuestros datos simulados, datos simulados y distribución uniforme con el objetivo de ver el contraste.

Analizando finalmente estos datos, en concreto la Figura 4.10, podemos confirmar que se trata de una distribución normal bivariante con las medias y matriz de covarianzas obtenidas. La amplitud de probabilidad del estado base en su forma integral queda entonces descrita por

$$F(\mathbf{x}) = \left(\frac{e^r}{2}\right)^{N/2} \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{\left[-H\left(i\frac{\pi}{2}+r\right)-i\gamma E\right]} p(H,E|\mathbf{x})\, dH\, dE\,, \tag{4.33}$$

$$\rho(E,H|\mathbf{x}) = \frac{1}{\pi\sigma_E\sqrt{N}\sqrt{1-\rho^2}} \exp\left(-\frac{1}{2(1-\rho^2)}\left(\frac{E^2}{\sigma_E^2} + \frac{2(H-N/2)^2}{\sqrt{N}} - \frac{4\rho E(H-N/2)}{(\sigma_E\sqrt{N})}\right)\right). \tag{4.34}$$

Con el objetivo de calcular esta integral, se va a utilizar la estandarización de nuestro par de variables $(E,H)$. Esta consiste en encontrar una matriz $A$ tal que

$$\begin{pmatrix} E \\ H \end{pmatrix} = A\begin{pmatrix} x \\ y \end{pmatrix} + \mu \quad \text{donde A es la matriz que cumple } \mathbf{\Sigma} = \mathbf{A}\mathbf{A}^{\mathbf{T}} \text{ y } \mu = \begin{pmatrix} \mu_E \\ \mu_H \end{pmatrix} = \begin{pmatrix} 0 \\ \mu_H \end{pmatrix}. \tag{4.35}$$

El primer paso será calcular la matriz $A$ que al tratarse de una matriz $2\times 2$ se puede realizar de la siguiente forma:

$$\Sigma = \begin{pmatrix} \sigma_E^2 & \sigma_{E,H} \\ \sigma_{E,H} & \sigma_H^2 \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} a & c \\ b & d \end{pmatrix} \Rightarrow \begin{cases} a^2+b^2 = \sigma_E^2\,. \\ ac+bd = \sigma_{EH}\,. \\ d^2+c^2 = \sigma_H^2\,. \end{cases} \tag{4.36}$$

Consiste en un sistema de tres ecuaciones no lineales con cuatro incógnitas por lo que es evidente que la solución no será única. Aún así, la elección de la solución del sistema no afectará las conclusiones finales. Por simplicidad se elige $c=0$ y se obtiene:

$$A = \begin{pmatrix} \sqrt{\sigma_E\sigma_H - \sigma_{EH}^2} & \frac{\sigma_{EH}}{\sigma_H} \\ 0 & \sigma_H \end{pmatrix}. \tag{4.37}$$

Con este cambio de variable se llega a la siguiente función de distribución,

$$\rho(x,y) = \frac{1}{2\pi} e^{-\frac{1}{2}\left(x^2+y^2\right)}. \tag{4.38}$$

Finalmente, llamando $a = -\left(i\frac{\pi}{2}+r\right)$ y $b = -i\gamma$ para facilitarnos los cálculos y completando

cuadrados,

$$\begin{aligned}
F(\mathbf{x}) &= \left(\frac{e^r}{2}\right)^{N/2} \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{aH+bE} p(H,E|\mathbf{x})\, dH\, dE \\
&= \left(\frac{e^r}{2}\right)^{N/2} \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{a\sigma_H y + a\mu_H + bx\sqrt{\sigma_E\sigma_H-\sigma_{EH}^2}+by\frac{\sigma_{EH}}{\sigma_H}} \frac{1}{2\pi} e^{-\frac{1}{2}\left(x^2+y^2\right)} dxdy \\
&= \left(\frac{e^r}{2}\right)^{N/2} e^{a\mu_H+\frac{b^2\sigma_E\sigma_H}{2}+\frac{a^2\sigma_H^2}{2}+2ab\sigma_{EH}} \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \frac{1}{2\pi} e^{-\frac{1}{2}\left((x-b\sqrt{\sigma_E\sigma_H-\sigma_{EH}^2})^2+(y-(a\sigma_H+b\frac{\sigma_{EH}}{\sigma_h}))^2\right)} dxdy \\
&= \left(\frac{e^r}{2}\right)^{N/2} e^{a\mu_H+\frac{b^2\sigma_E\sigma_H}{2}+\frac{a^2\sigma_H^2}{2}+ab\sigma_{EH}} \\
&= \left(\frac{e^r}{2}\right)^{N/2} e^{-\left(i\frac{\pi}{2}+r\right)\mu_H+\frac{(i\gamma)^2\sigma_E\sigma_H}{2}+\frac{(i\frac{\pi}{2}+r)^2\sigma_H^2}{2}+\left(i\frac{\pi}{2}+r\right)(i\gamma)\sigma_{EH}}\,. \qquad (4.39)
\end{aligned}$$

Donde en el último paso, el valor de la integral se iguala a uno al ser una distribución de probabilidad normalizada.

Se escribe finalmente la probabilidad del estado final $x$ como:

$$|F(\mathbf{x})|^2 = \left(\frac{e^r}{2}\right)^{N} e^{-2r\mu_H+\left(r^2-\frac{\pi^2}{4}\right)\sigma_H^2-(\gamma\pi\sigma_{EH})}\,. \qquad (4.40)$$

Si uno se fija, el único término que depende de $\mathbf{x}$ es $\sigma_{EH}(\mathbf{x})$ por lo que el siguiente paso, es estudiar como se comporta este término para extraer más conclusiones acerca del funcionamiento de estos algoritmos.

El valor empírico de $\sigma_{EH}(\mathbf{x})$ es $\frac{1}{2^N}\sum_{x'}(H_x - E[htb])(E_x - E[E])$. Es por eso que se puede intuir que el espectro de energía está altamente correlacionado con $\sigma_{EH}(\mathbf{x})$. Concretamente, en espacios no degenerados, cuando x es el estado de energía más bajo es más probable encontrar estados de baja energía y similar $(E_{\mathbf{x}'} - E[E]) < 0$ cerca del estado de referencia $(H_{\mathbf{x}'} - H[htb]) > 0$ y estados de alta energía $(E_{\mathbf{x}'} - E[E]) > 0$ lejos $(H_{\mathbf{x}'} - H[htb]) > 0$, por lo que conduce a una covarianza positiva. Razonando de manera análoga, cuando $\mathbf{x}$ sea un estado de alta energía, dará lugar a una covarianza negativa y conforme vayamos moviéndonos de un estado a otro, los términos positivos pasarán a ser positivos a ser negativos poco a poco. Con todo esto, podemos esperar tener una relación lineal entre covarianza y estados, aunque por simplicidad ya que el estado de mínima energía puede ir variando de un problema a otro, vamos a centrarnos en la relación entre covarianza y energía del estado fijado.

Si se representa el valor de $\sigma_{EH}(\mathbf{x})$ que se obtiene para cada $\mathbf{x}$ fijo de los pares $(E_{\mathbf{x}'}, H_{\mathbf{x}\mathbf{x}'})$, pero en lugar de representarlo en función de $\mathbf{x}$, lo representamos en función de la energía $E_{\mathbf{x}}$ normalizada de forma que podamos hacer la superposición en la misma gráfica de los resultados obtenidos para diferentes $x$, se obtiene la Figura 4.11 junto a un $R^2 = 0{,}9263$ asociado a esa regresión lineal.

Por todo ello, podemos suponer una dependencia lineal de la covarianza

$$\sigma_{EH}(\mathbf{x}) = -cE_{\mathbf{x}} \pm \omega\,, \qquad (4.41)$$

donde $c$ es una constante real mayor que cero y $\omega$ es un término estocástico que representa ese error respecto a la regresión lineal.

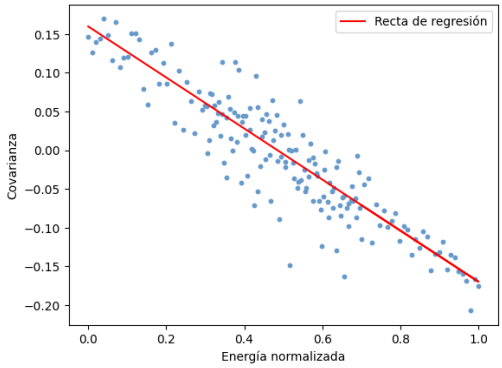



Figura 4.11: Representación de la correlación entre la covarianza de los pares $(E_{\mathbf{x}'}, H_{\mathbf{x}\mathbf{x}'})$ para $\mathbf{x}$ fijo en función de la energía $E_{\mathbf{x}}$.

Teniendo esto último en cuenta y llamando $\beta = c\pi\gamma$, la probabilidad de que el estado final este en el estado $\mathbf{x}$ viene dada por el cuadrado de la amplitud, es decir,

$$|F(\mathbf{x})|^2 = Ke^{-\beta E_{\mathbf{x}}} \text{ donde } K = \left(\frac{e^r}{2}\right)^N e^{-2r\mu_H + \left(r^2 - \frac{\pi^2}{4}\right)\sigma_H^2 \pm \frac{\beta\omega}{c}}. \tag{4.42}$$

Por lo que las probabilidades de que el estado final este en un estado concreto sigue una **distribución de Boltzmann**. La constante de Boltzmann, $\beta$, depende tanto del $\gamma_{opt}$ elegido como del problema que se este estudiando ya que entra en juego de esa covarianza entre energía y distancia entre estados.

Además, uno se da cuenta que ya no trabajamos con los estados en si, sino con la energía que cada uno de ellos tiene asociada. Nuestro objetivo, es así obtener la distribución de probabilidad de la energía en el estado final $|\gamma, \theta\rangle$.

Ahora falta simplemente añadirle la distribución de las energías, que depende del hamiltoniano del problema y de como se ha generado este. Por tanto, la función de distribución de la energía en nuestro estado final se puede escribir como,

$$P(E) \sim d(E)e^{-\beta E}, \tag{4.43}$$

donde $d(E)$ es la distribución de energías. Esta depende del problema que se este estudiando.

Para el caso en el que estamos trabajando, un problemas tipo Ising generado aleatoriamente a partir de una distribución normal, la distribución $d(E)$ será esa misma distribución normal.

### 4.2.2. Aplicación a un hamiltoniano de Ising concreto

Con el objetivo de visualizar esta dependencia anterior, se ha llevado a cabo un ejemplo de este proceso para un Hamiltoniano tipo Ising generado a partir de un grafo tipo Erdös–Rényi con probabilidad 0.5 y pesos obtenidos a partir de una distribución normal estándar. Para ello se

han representado las probabilidades del estado final en función de su energía todo ello en escala logarítmica. Además, se ha llevado a cabo una regresión lineal de los datos obtenidos con el objetivo de encontrar esa constante de Boltzmann que caracteriza el modelo.

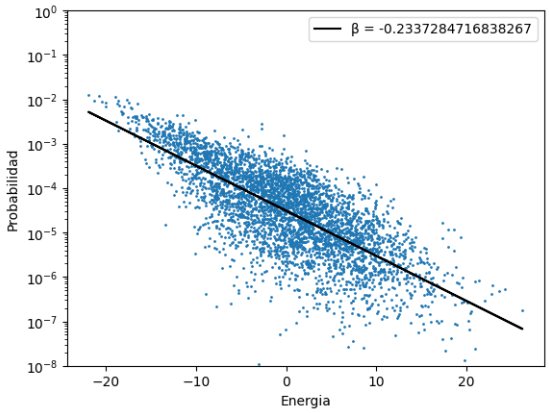



Figura 4.12: Representación logarítmica las probabilidades del estado final $|\gamma, \theta\rangle$ de un problema tipo QUBO generado a partir de un gráfico de Erdös–Rényi y cuyos pesos siguen una distribución normal estándar. Junto a ella se ha representado la pendiente correspondiente con la constante de Boltzmann.

Los resultados obtenidos se pueden observar en la Figura 4.12, la cual nos permite sacar una seria de conclusiones respecto a como se comportan estos sistemas ante un algoritmo QAOA:

- La dependencia de la probabilidad con $\beta$ al ir esta disminuyendo conforme aumenta la energía.
- La aportación de la distribución normal del espectro de energías al existir una mayor agrupación de valores en torno a esta media del espectro que es cero en este caso.
- No se puede olvidar uno tampoco de ese parte estocástica que deriva de la aproximación lineal de la covarianza. Es por esto que los puntos no siguen exactamente esa linea recta sino que tienen una cierta desviación. Aparentemente mayor en el centro simplemente por el hecho de existen más valores en esta zona.
- Si recapitulamos en cual era el objetivo de los algoritmos QAOA, la idea principal era obtener el mínimo de la función y en que estado se obtenía. Si observamos atentamente el gráfico, nuestro estado de menor energía es el que mayor probabilidad tiene y por tanto, realizando varias mediciones, el estado que más veces se observe, será el que buscamos. Y aún no obteniendo este estado porque al fin y al cabo es probabilístico, seguramente se obtenga otro de los estados próximos que aún no siendo la mejor opción, estará muy cerca por esa correlación entre energía y distancia entre estados que se ha visto.

Lo más sorprendente de todo esto es que el proceso deductivo que se ha llevado a cabo, nos permite ampliar la utilidad de estos algoritmos. No solo tiene como aplicación la optimización de funciones sino que también constituye un gran avance en el muestreo de distribuciones de probabilidad.

De esta forma, los estados finales a la salida de un algoritmo variacional, cuando se proyectan en la base computacional, se convierten esencialmente en distribuciones de probabilidad clásicas

y medirlos equivale a tomar muestras de estas distribuciones, conocido también como **random sampling o muestreo aleatorio**. En este sentido, una computadora cuántica puede considerarse como una máquina capaz de generar distribuciones de probabilidad complejas.

Aunque los algoritmos de Monte Carlo de Cadena de Markov (MCMC)[17] se utilizan clásicamente para simular estas distribuciones de Bolztmann, su eficacia disminuye drásticamente a temperaturas muy bajas. Y es por eso los Quantum Approximate Optimization Algorithm (QAOA) cobran gran importancia al demostrarse que estos pueden aproximar estas distribuciones incluso más allá de los límites teóricos de los métodos clásicos [18, 19].

Además, el reciente auge de estos algoritmos QAOA se debe sobretodo a la variedad de aplicaciones que estas distribuciones tienen en diversos campos muy presentes en la actualidad:

- Aparece en primer lugar en muchos campos de la física. Como por ejemplo la descripción de la distribución de velocidades de las moléculas en un gas, el modelado de la energía de los átomos en un material sólido o el estudio de la distribución de energía entre los átomos o moléculas en un sólido, lo que influye en propiedades como la conductividad térmica y eléctrica.
- Por otro lado, se ha convertido en un pilar de inteligencia artificial para el aprendizaje no supervisado y la exploración en algoritmos de aprendizaje por refuerzo. Se emplea por ejemplo en técnicas como las redes neuronales restringidas de Boltzmann para modelar la distribución conjunta de variables observadas y no observadas, lo que permite aprender representaciones útiles de los datos y generar nuevos datos similares a los de entrenamiento ahorrando tiempo y recursos.
- Incluso en economía, se emplea para entender cómo se distribuyen las elecciones y decisiones de consumo entre los individuos, considerando factores como la utilidad esperada y las restricciones de recursos lo que puede tener implicaciones importantes para la formación de políticas económicas y la predicción de tendencias económicas.

En resumen, este estudio y otros similares están abriendo nuevas perspectivas en la interpretación inicial de la computación cuántica que era una simple optimización dando paso a una comprensión más profunda de los fenómenos físicos subyacentes aplicables en muchos otros ámbitos.

# Capítulo 5

# Algoritmo QAOA para múltiples capas

Como se vio en la introducción de los **QAOA**, el objetivo principal de estos consistía en ir de un estado de mínima energía conocido, a otro desconocido y el cual buscamos. Todo ello de manera adiabática y lenta. La forma que estos algoritmos tenían de hacerlo consistía en dividir la evolución continua del Hamiltoniano en partes pequeñas llamadas capas, de forma que al someter al sistema a una infinidad de estas, el estado final fuera una aproximación del estado buscado. Y llegado a este punto, nos entran preguntas como cuál es el número de capas necesario para obtener el resultado aceptable o como de fiable es usar este tipo de algoritmos para un número reducido de capas.

Hasta ahora, solo se han realizado experiencias con una única capa y a pesar de lo que uno pueda esperar, los resultados obtenidos son bastante acertados. Esto se debe a que al fin y al cabo uno esta forzando los ángulos que minimicen la energía final y por tanto, el estado final tendrá una fuerte componente correspondiente al estado de mínima energía y esto nos permite calcular ese mínimo. Sin embargo, el estado final al que se llega no se acerca prácticamente al estado base, sino que las amplitudes de probabilidad que lo componen son mayores cuanto menos energía tenga el estado de la base computacional como se puede ver en la Figura 4.12.

$$\text{Estado de mínima energía buscado: } |\gamma,\theta\rangle = \mathbf{x}_0\,,$$
$$\text{Estado al que se llega: } |\gamma,\theta\rangle = \epsilon_0\mathbf{x}_0 + \sum_{\mathbf{x}'\neq\mathbf{x}_0} \epsilon_{\mathbf{x}'}\mathbf{x}' \text{ con } \epsilon_0 > \epsilon_{\mathbf{x}'}\,.$$

Entonces, el objetivo de este capítulo es ver como se comporta el algoritmo para múltiples capas y compararlo con los resultados obtenidos para una única capa.

## 5.1. Comparación de los resultados obtenidos para algoritmos QAOA compuestos por varias capas

En este apartado compararemos los resultados obtenidos para un algoritmo QAOA de una capa, frente a los resultados obtenidos para dos, tres y cuatro capas. Todo ello aplicado al mismo problema

tipo Ising generado aleatoriamente de la misma forma que se hizo en resultados previos.

En la Figura 5.1 se pueden observar los resultados obtenidos.

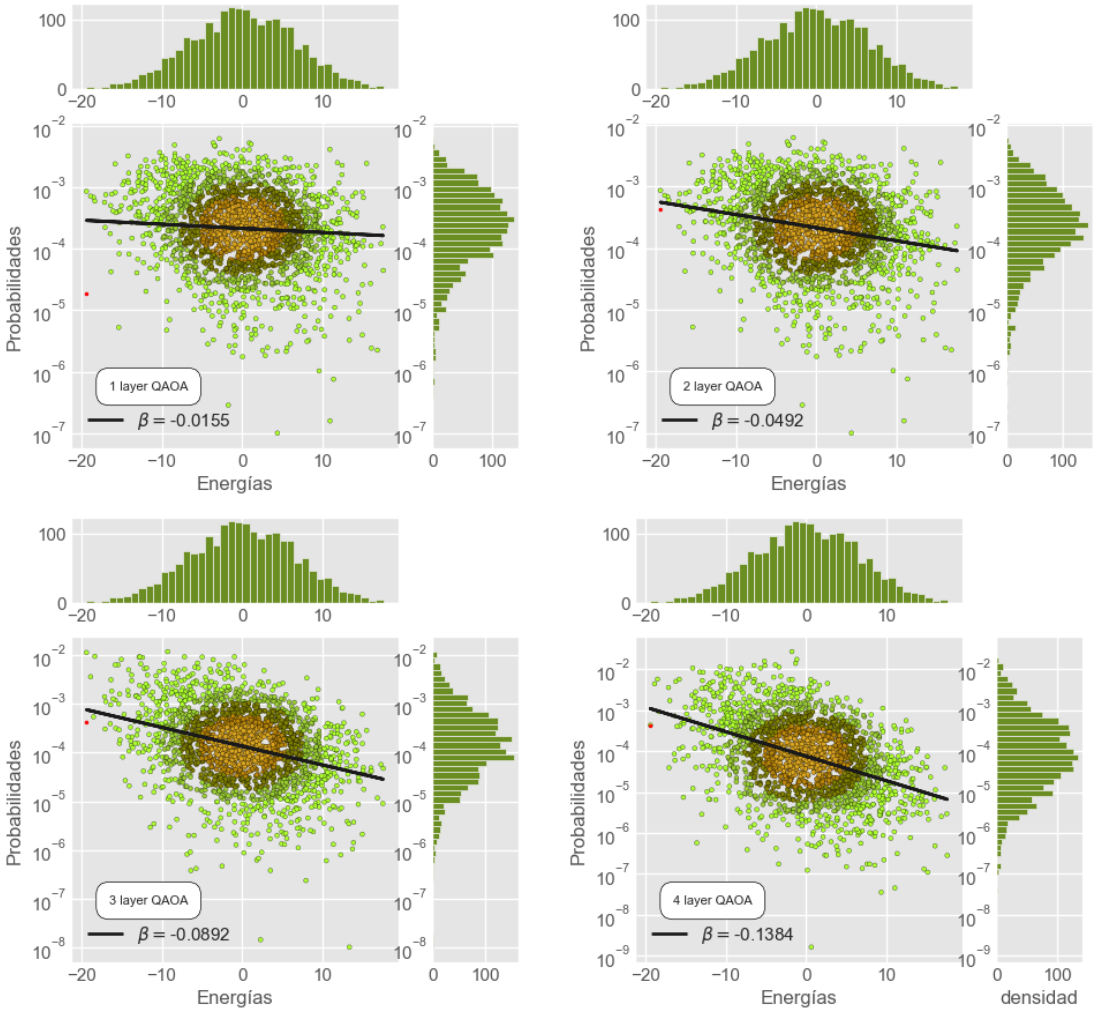



Figura 5.1: Representación de las probabilidades del estado final de QAOA en función de su energías para diferente número de capas. Para una mejor interpretación se ha coloreado los diferentes puntos en función de las elipses de confianza de los propios datos. En rojo se representa el estado de mínima energía buscado y entorno a la representación principal encontramos unos histogramas que permiten visualizar como se distribuyen las energías y las probabilidades de forma independiente.

Analizando esta Figura 5.1 se pueden resaltar una serie de aspectos:

- Lo primero que reparamos es que esta distribución de probabilidad que aparece con varias capas es, como se podría esperar, igual que en el caso de una sola, una distribución de Bolztmann con cierta componente normal así como ese ruido estocástico que no se podía controlar
- Además de eso, se observa que la pendiente de Boltzmann es casi siempre mayor en el caso de varias capas. Esto implica que conforme aumente la energía del autoestado, disminuye su

probabilidad de manera más rápida y por tanto es un indicativo de que para más capas los resultados son más acertados. Esta idea se plasma en la Figura 5.2 donde se han representado las probabilidades acumuladas de forma que conforme aumenta el número de capas, la distribución se desplaza a la izquierda debido a esa mayor probabilidad de los estados de baja energía.

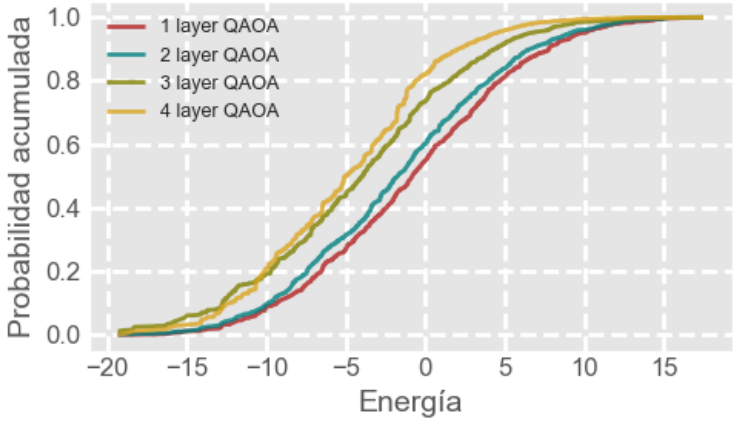



Figura 5.2: Representación de las probabilidades acumuladas del estado final en función de las energías para diferente números de capas y para un mismo problema Tipo Ising.

- Sin embargo el hecho anterior no significa que el estado buscado, el estado de mínima energía, tenga una probabilidad mayor en el caso de más capas. Esto se debe a que como se vio en el resultado teórico, hay una componente estocástica que no se puede controlar y que altera las probabilidades.
- En cuanto a los histogramas, el superior, siempre será el mismo ya que refleja la distribución de energías. El segundo, aunque parezca muy similar, consiste en la distribución normal desplazada ligeramente a las probabilidades alta debido a esa contribución de la distribución de Boltzmann.

Estos resultados nos indican que a pesar de esas mejores pendientes, aumentando el número de capas no siempre se obtiene un estado final próximo al estado de mínima energía (es decir, cuya probabilidad de estar en este mínimo sea la más alta de todas). Por tanto, el siguiente paso será generar un número elevado de problemas, (se han llevado a cabo 2000 problemas tipo Ising para ordenadores cuánticos de 6 qubits), y calcular la probabilidad de que el estado final se encuentre en el ground state para un diferente número de capas y comparar los resultados obtenidos. Esto se puede visualizar en la Figura 5.3.

Es importante también destacar que en todos los histogramas de la Figura 5.3, inicialmente aparecía un pico muy pronunciado en las proximidades del cero. Se ha tomado la decisión de eliminar este pico ya que, al igual que se ha mencionado en apartados previos, se debe a fallos en el algoritmo de optimización.

Por otro lado y con el objetivo de tener un visión comparativa de los resultados, calculamos las

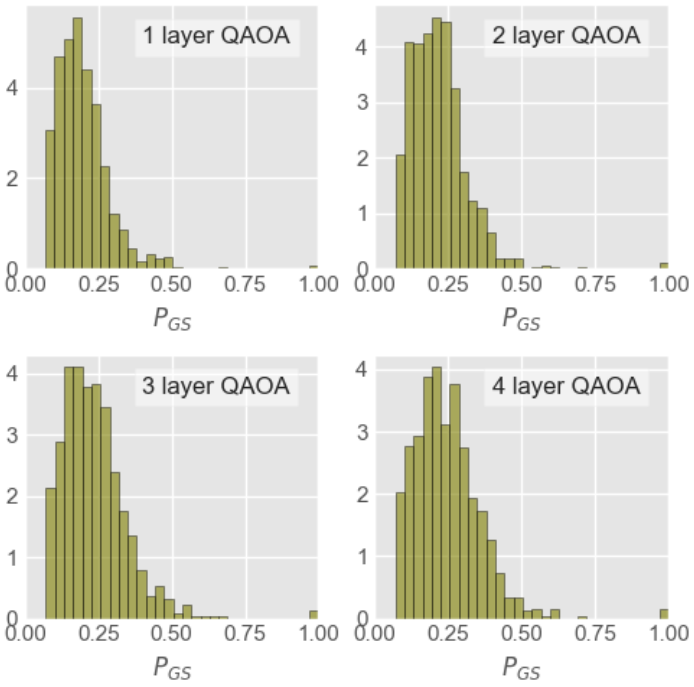



Figura 5.3: Histogramas para diferente número de capas de las probabilidades del estado de mínima energía para 2000 simulaciones de QAOA para hamiltonianos Tipo Ising.

funciones de densidad asociadas a estos datos para poder así superponerlas como se muestra en la Figura 5.4.

Teniendo esto en cuenta, es posible llegar a una serie de conclusiones acerca del impacto que tiene un aumento de número de capas en los resultados finales del QAOA:

- Se vio previamente que para un caso concreto el estado final para un mayor número de capas no tiene porque tener una mayor probabilidad de caer en el estado buscado. Sin embargo, al estudiar estos nuevos resultados, se observa que un aumento en el número de capas produce un desplazamiento hacia la derecha de las densidades asociadas a la probabilidad del estado base, es decir, estas probabilidades aumentan. Este aumento no es excesivamente significativo pero si supone una mayor eficacia.

- Además, fijándonos en los histogramas se observa que en aquellos algoritmos con tres o cuatro capas aparece alguna simulación donde se ha llegado al estado de mínima energía con probabilidad uno o muy cercana de uno. Esto se debe a que existe un mayor número de grados de libertad, es decir, se está intentando optimizar un función dependiente de 8 variables (2 ángulos por cada capa) en lugar de solamente dos variables. Para visualizar esto de manera sencilla, imagínese una función a optimizar que depende de 8 parámetros. Variando estos 8 parámetros es más sencillo encontrar un máximo que si fueran solo 2, hay más posibilidades. Un ejemplo de lo que está ocurriendo se vio en el primer caso de un único cubit en el que hay exactamente dos probabilidades ($2^N = 2^1 = 2$) y dos parámetros variables, por eso es posible llegar a el máximo ideal de probabilidad igual a uno.

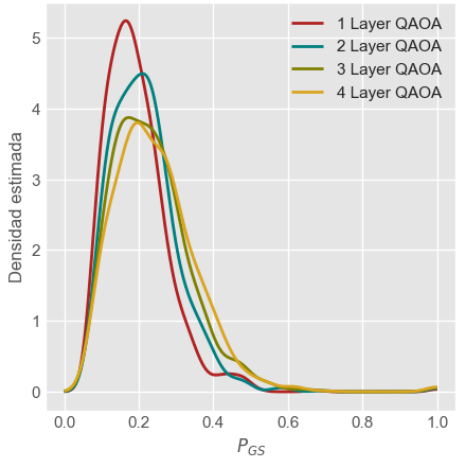



Figura 5.4: Representación comparativa de las densidades asociadas a la probabilidad del estado de mínima energía.

Estas conclusiones nos permiten deducir finalmente que existe esa mejora que el propio fundamento teórico de los algoritmos QAOA predice. Sin embargo, la mejora no es tan significativa como cabía esperar ya que al fin y al cabo, se están poniendo cuatro veces más capas.

Esto nos lleva a preguntarnos la rentabilidad del aumento del número de capas de estos algoritmos aplicados en un ordenador cuántico y no una simulación en un ordenador convencional como se ha trabajado. Esto se debe a muchos factores. Primero, intrínsecos a la naturaleza cuántica, ya que como se menciono inicialmente, conforme se aumenta el número de capas aparece un ruido debido a la decoherencia cuántica. Sería necesario buscar un equilibrio que nos permita un resultado fiable que no se vea afectado por este ruido. Y segundo, el factor económico, los ordenadores cuánticos necesitan muy bajas temperaturas y aumentar el número de capas supondría un mayor coste tanto energético como de infraestructura. Todo ello por un aumento de la probabilidad ligeramente significativo.

# Capítulo 6

# Conclusiones

Tras realizar un estudio tanto teórico como numérico del comportamiento de los **Algoritmos de Optimización Cuántica Aproximada** (QAOA) hemos podido llegar a una serie de conclusiones analizando debidamente los resultados que nosotros mismos hemos sido capaces de generar en los diferentes enfoques que se han hecho.

En primer lugar, pese a que este tipo de algoritmos estuvieran inicialmente pensados para tiempos largos o equivalentemente para un número elevado de capas, al ir paralelo a un optimización paramétrica clásica, es posible reducir estas capas obteniendo resultados totalmente válidos. Es más, se puede reducir hasta tal punto que se trabaje con un sola capa y obtener resultados perfectamente válidos.

Paralelo a esta optimización, el procedimiento que se lleve clásicamente para la obtención de los ángulos óptimos involucrados tiene un peso sorprendentemente importante en los resultados. Sin embargo, al no ser el centro de interés del trabajo, hemos optado por adoptar un razonamiento crítico a la hora de aceptar o rechazar una solución. Consecuentemente, una mejora en este aspecto supondría una gran repercusión en los resultados de este tipo de algoritmos.

Pasando a otro aspecto, la conclusión central que extraemos de estos algoritmos recae en esas distribución que presentan los estados finales: la distribución de Boltzmann. Esta densidad refleja en primer lugar la disminución de la probabilidad conforme aumenta la energía de los distintos estados de la base computacional. Esto provoca que de norma general, los estados con mayor probabilidad sean aquellos que se buscan, los de menor energía. Sin embargo, para una correcta comprensión del problema hay que tener también en cuenta los pesos Gaussianos de las energías y ese ruido estocástico que, aunque no se pueda eliminar, es uniforme independientemente del problema.

Esta distribución que presentan los estados a la salida de nuestro ordenador cuántico, nos permite tras varias mediciones, y quedándonos con el mejor resultado existente, obtener una energía final muy próxima al valor óptimo de la función de coste protagonista. Recordemos que este resultado no será exacto pero si muy cercano al estado objetivo. Es por esta razón por la que el nombre del algoritmo contiene el término *aproximación*.

Finalmente, tras el estudio de los resultados obtenidos para un mayor número de capas, podemos concluir que el algoritmo aplicado a una única capa es también válido frente a los de un número

superior de capas. Y que además, en caso de querer aumentar este número de capas para mejorar la solución de nuestro problema, es necesario buscar un equilibrio que no destroce la coherencia cuántica de este tipo de ordenadores.

De cara a futuras investigaciones sobre este tema, el siguiente paso consiste en aplicar este tipo de algoritmos a problemas más concretos y conocidos como es el *MaxCut*. Esto va a permitir comprender patrones que este tipo de problemas ocultan y que se verán reflejados en las distribuciones de energías y probabilidades de los estados finales tras ser estudiado bajo los algoritmos QAOA.

# Bibliografía


[1] N. S. Yanofsky y M. A. Mannucci, Quantum computing for computer scientists Cambridge University Press, 2008).

[2] E. Farhi y J. Goldenstone, A quantum approximate optimization algorithm, arXiv preprint arXiv:1411.4028 (2014). Enlace.

[3] E. Farhi, J. Goldenstone, S. Gutmann y L. Zhou, The quantum approximate optimization algorithm and the Sherrington-Kirkpatrick model at infinite size, Quantum **6**, 759 (2022). Enlace.

[4] M. Cerezo, A. Arrasmith, et al., Variational quantum algorithms, Nat. Rev. Phys. **3** 625-644 (2021). Enlace.

[5] L. Zhou, S-T. Wang, S. Choi, H. Pichler, and M. D. Lukin, Quantum Approximate Optimization Algorithm: Performance, Mechanism, and Implementation on Near-Term Devices, Phys. Rev. X **10**, 021067 (2020). Enlace.

[6] A. Ozaeta, W. van Dam y P. L. McMahon, Expectation values from the single-layer quantum approximate optimization algorithm on Ising problems, Front. Quantum Sci. Technol. **7**, 045036 (2022) Enlace.

[7] P. Diez-Valle, D. Porras y J.J. Garcia-Ripoll, Connection between single-layer quantum approximate optimization algorithm interferometry and thermal distribution sampling, Front. Quantum Sci. Technol. **7**, 1321264 (2024). Enlace.

[8] P. Diez-Valle, D. Porras y J.J. Garcia-Ripoll, Quantum approximate optimization algorithm pseudo-Boltzmann states, Phys. Rev. Lett. **130**, 050601 (2023). Enlace.

[9] C. Sellero y A. Sánchez, Tema 3: Contraste de la normalidad multivariante (Universidad de Santiago de Compostela, 2010)

[10] Porras y C. Jaime, Comparación de pruebas de normalidad multivariada, Anales Científicos, **77**, 141-146 (2016). Enlace.

[11] J. Han, W. Cai, L. Hu, X. Mu, Y. Ma, Y. Xu, W. Wang,H. Wang, Y.P. Song, C. Zou y C. L. Sun, Experimental Simulation of Open Quantum System Dynamics via Trotterization, Phys. Rev. Lett. **127** (2021). Enlace.

[12] V. M. Bonillo, Principios Fundamentales de Computación Cuántica, (Universidad de A Coruña, 2013). Enlace.

[13] Y. Sun, J.Y. Zhang, M.S. Byrd y L.A. Wu, Lian-Ao, Adiabatic quantum simulation using trotterization, arXiv:1805.11568 (2018). Enlace.

[14] E. Farhi, J. Goldstone, S. Gutmann y M. Sopser, Quantum Computation by Adiabatic Evolution, arXiv[quant-ph/0001106] (2000). Enlace.

[15] D. López-Sandobal y C. Cobos-Lozada, Computación Cuántica Adiabática aplicada, a la solución del Problema de la Mochila Binaria, Revista Ibérica de Sistemas e Tecnologias de Informacao **E38**,214-227 (2020). Enlace.

[16] Z. Bian, F. Chudak, Macready, The Ising model: teaching an old problem new tricks, D-wave systems **2**, 1-32 (2010). Enlace.

[17] D. Layden, G. Mazzola, R. Mishmash, M. Motta, P. Wocjan, J.S. Kim, S. Sheldon, Quantum-enhanced markov chain Monte Carlo, Nat. Phys. **619**, 282-287 (2023). Enlace.

[18] H.S. Zhong, Y.H. Deng, J. Qin, H. Wang, M.C. Chen, L. Peng y Y. Luo, Phase-programmable gaussian boson sampling using stimulated squeezed light, Phys. Rev. Lett. **127**, 180502 (2021). Enlace.

[19] F. Arute, K. Arya, R. Babbush, D. Bacon, J. Bardin, R. Barends, R. Biswas, S. Boixo, F. Brandao, D. Buell y otros, Quantum supremacy using a programmable superconducting processor, Nat. Phys. **574**, 505-510 (2019). Enlace.

[20] F. Glover, G. Kochenberger, R. Hennig, Y. Du, Yu, Quantum bridge analytics I: a tutorial on formulating and using QUBO models, Ann. Oper. Res. **314**, 141-183, (2022). Enlace.

[21] A. M. Steane y D.M. Lucas, Quantum computing with trapped ions, atoms and light, Fortschr. Phys. **48**, 839–858, (2000). Enlace.

[22] M. Markham, D. Twitchen, The diamond quantum revolution, Phys. World **33**, 39, (2020). Enlace.

[23] D. Vajner, L. Rickert, T. Gao, K. Kaymazlar, T. Heindel, Quantum communication using semiconductor quantum dots, Adv. Quantum Technol. **5**, 2100116, (2022). Enlace.

[24] M. Leuenberger y D. Loss, Daniel, Quantum computing in molecular magnets, Nat. Phys. **410**, 789-793 (2001). Enlace.

[25] C. Cohen-Tannoudji, B. Diu y F. Laloe, Quantum Mechanics **Vol. 1** (Hermann, 1977).

[26] C. Cohen-Tannoudji, B. Diu y F. Laloe, Quantum Mechanics **Vol. 2** (Hermann, 1977).

[27] E. Rieffel y W.H. Polak, Quantum computing: A gentle introduction, (MIT pres 2011).

[28] J. J. Sakurai y J. Napolitano, Modern quantum mechanics, (Cambridge University Press 2020).

[29] D. J. Griffiths y D. F. Schroeter, Introduction to quantum mechanics, (Cambridge University Press 2018).

[30] O. Adesina, A Comparative Study of Sorting Algorithms, Afr. J. Comp. **6**, 199-206, (2013). Enlace.

# Apéndice A

# Conceptos estadísticos previos

En este capítulo, exploraremos una serie de conceptos estadísticos fundamentales que son esenciales para comprender y analizar datos en el contexto de distribuciones normales multivariantes y distribuciones empíricas de probabilidad. La comprensión de estos conceptos es crucial para interpretar correctamente los datos, identificar patrones significativos y tomar decisiones que permitan llegar a entender aspectos críticos del modelo.

## A.1. Distribución normal multivariante

En teoría de la probabilidad y estadística, la distribución normal multivariante, o también conocida como, distribución Gaussiana multivariante es una generalización de la distribución normal unidimensional a dimensiones superiores.

La distribución normal en una dimensión es sigue la siguiente forma: $f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2\sigma^2}(x-\mu)^2}$. De está forma, para definirla, cada una de las dimensiones tendrá asociada una media y una varianza $\mu = (\mu_{\mathbf{1}}, ..., \mu_{\mathbf{n}})$ y $\sigma = (\sigma_{\mathbf{1}}, ..., \sigma_{\mathbf{n}})$. Sin embargo, al no existir una única variable, aparece una correlación entre las distintas dimensiones que se refleja en lo que se conoce como **covarianza**.

La covarianza es básicamente un estadístico que se usa para determinar si existe dependencia entre dos variables. Matemáticamente se escribe como:

$$\mathrm{Cov}(X, Y) = \sigma_{XY} = \frac{1}{n} \sum_{i=1}^{n} (x_i - \mu_x) (y_i - \mu_y) \sigma_{YX}. \tag{A.1}$$

De esta forma, cuando ambas variables tienen un comportamiento similar, valores altos con altos y bajos con bajos, está será elevada y positiva, y si ocurre al contrario, altos con bajos, la covarianza será negativa. Y si por otro lado, ambas son totalmente independientes, habrá de todo y la covarianza será nula.

Está información se recoge en lo que se conoce como matriz de covarianzas n-dimensional:

$$\mathbf{\Sigma} = \begin{pmatrix} \sigma^2_{x_i} & \sigma_{x_1,x_2} & \dots & \sigma_{x_1,x_n} \\ \sigma_{x_2,x_1} & \sigma^2_{x_2} & \dots & \sigma_{x_2,x_n} \\ \dots & \dots & \dots & \dots \\ \sigma_{x_n,x_1} & \sigma_{x_2,x_n} & \dots & \sigma^2_{x_n} \\ . & & & \end{pmatrix} \tag{A.2}$$

Y la forma que toma entonces la distribución normal multivariante es

$$f(\mathbf{x}) = \frac{\mathbf{1}}{(\mathbf{2}\pi)^{\mathbf{k/2}}|\mathbf{\Sigma}|^{\mathbf{1/2}}} \exp\left(-\frac{\mathbf{1}}{\mathbf{2}}(\mathbf{x}-\boldsymbol{\mu})^{\mathbf{T}}\mathbf{\Sigma}^{\mathbf{-1}}(\mathbf{x}-\boldsymbol{\mu})\right) \qquad \text{donde } k \text{ es la dimensión.} \tag{A.3}$$

Para ver como se comportara está distribución en función de las covarianzas se han ilustrado una serie de datos generados aleatoriamente para diferentes covarianzas todas en dos dimensiones y centradas en el $(0, 0)$

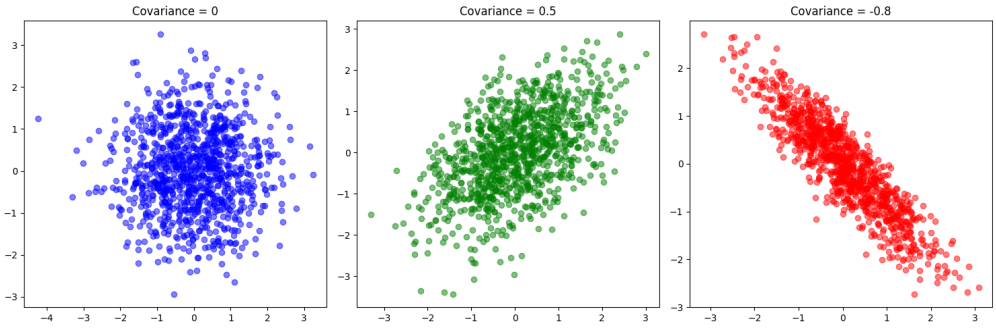



Figura A.1: Visualización de tres distribuciones normales con $k = 2$ centradas en (0,0) y varianzas individuales igual a 1 pero con diferentes covarianzas.

**ESTANDARIZACIÓN**

Dentro de las distribuciones normales multivariantes destaca la distribución normal estándar (primer gráfico de Figura 4.1) que se trata de aquella caracterizada por media igual al vector cero y matriz de covarianzas la identidad ya que la función de densidad se simplifica enormemente: $f(\mathbf{x}) = \frac{\mathbf{1}}{(\mathbf{2}\pi)^{\mathbf{k/2}}}\mathbf{e}^{-\frac{\mathbf{1}}{\mathbf{2}}\mathbf{x}^{\mathbf{T}}\mathbf{x}}$.

Lo útil de esta distribución es a partir de ella se puede obtener cualquier otras y viceversa, siempre podremos a partir de una distribución normal cualquiera volver a la normal estándar.

$$\mathbf{X} \sim \mathcal{N}(\boldsymbol{\mu}, \mathbf{\Sigma}) \iff \mu \in \mathbb{R}^k, \boldsymbol{A} \in \mathbb{R}^{k\times\ell} \text{ tales que } \mathbf{X} = \boldsymbol{A}\mathbf{Z} + \mu \text{ y } \forall n = 1, \dots, \ell : Z_n \sim \mathcal{N}(0, 1) \text{ i.i.d.} \tag{A.4}$$

Donde se cumple que $\mathbf{\Sigma} = \boldsymbol{A}\boldsymbol{A}^{\mathrm{T}}$. Es importante darse cuenta que dado $\mathbf{\Sigma}$ la matriz $\boldsymbol{A}$ no es única.

**DISTANCIA DE MAHALANOBIS**

La distancia de Mahalanobis es una medida de la distancia entre un punto $P$ y una distribución $D$, introducida por P. C. Mahalanobis en 1936. Matemáticamente se define como:

$$D_M(\mathbf{x}, \mu, \mathbf{\Sigma}) = \sqrt{(\mathbf{x}-\mu)^{\mathrm{T}}\mathbf{\Sigma}^{-1}(\mathbf{x}-\mu)}\,. \tag{A.5}$$

La idea proviene básicamente en medir la distancia de un punto al punto donde se encuentra la media lo que equivaldría a:$\sqrt{(\mathbf{x}-\mu)^{\mathrm{T}}(\mathbf{x}-\mu)}$. Sin embargo, con idea de darle un sentido más general se le añade ese término de normalización mencionado anteriormente. $\mathbf{\Sigma}$ ($\mathbf{Z} = \boldsymbol{A}^{-1}(\mathbf{X}-\mu)$):

$$D_M(\mathbf{x}) = \sqrt{\mathbf{z^T z}} = \sqrt{(\mathbf{x}-\mu)^{\mathrm{T}}(\boldsymbol{A}^T)^{-1}\boldsymbol{A}^{-1}(\mathbf{x}-\mu)} = \sqrt{(\mathbf{x}-\mu)^{\mathrm{T}}\mathbf{\Sigma}^{-1}(\mathbf{x}-\mu)}\,. \tag{A.6}$$

Esta distancia, está definida para cualquier distribución. Pero cuando se aplican a una distribución normal de cualquier número de dimensiones, la densidad de probabilidad de una observación $\mathbf{x}$ está determinada de manera única por la distancia de Mahalanobis $D_M(\mathbf{x})$:

$$f(\mathbf{x}) = \frac{\mathbf{1}}{\sqrt{(\mathbf{2}\pi)^{\mathbf{n}}|\mathbf{\Sigma}|}} \exp\left(-\frac{(\mathbf{x}-\mu)^{\mathbf{T}}\mathbf{\Sigma^{-1}}(\mathbf{x}-\mu)}{\mathbf{2}}\right) \mathbf{dx} = \frac{\mathbf{1}}{\sqrt{\det(\mathbf{2}\pi\mathbf{\Sigma})}} \exp\left(-\frac{\mathbf{D_M}(\mathbf{x})^{\mathbf{2}}}{\mathbf{2}}\right). \tag{A.7}$$

Específicamente, $D_M(\mathbf{x})^2$ sigue una distribución **chi-cuadrado con $n$ grados de libertad**, donde $n$ es el número de dimensiones de la distribución normal.

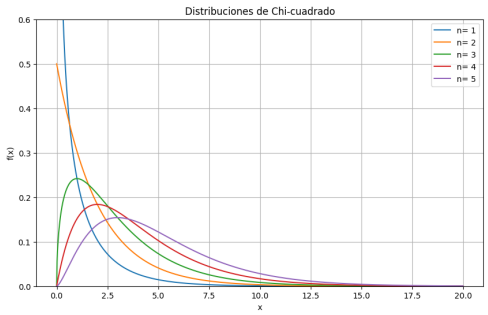



Figura A.2: Visualización diferentes distribuciones chi cuadrado para varios grados de libertad, es decir, para varias dimensiones de distribución multivariante.

La utilidad de este tipo de distribución se refleja en uno de los test de hipótesis que existen para comprobar si, dado una serie de puntos experimentales, pueden ser considerados o no como una distribución normal multivariante en función de si la distancia de Mahalanobis se comporta o no como un tal distribución. Estos test se explican más profundamente en [9, 10].

Nosotros nos basaremos en comprobar como se adaptan los datos experimentales a los cuantiles teóricos de la distribución chi cuadrado. Esto se hace mediante un tipo de gráfico llamado **QQ-plot**. Para ello, creamos los pares $(x_i, y_i)$ para $i = 1, 2, 3, ..., k$ siendo $k$ el número de datos existentes. $y_i$ serán esos datos experimentales ordenados de menor a mayor valor, mientras que $x_i$ será el cuantil $i-esimo$ de la distribución teórica que queremos comparar.Es decir, si $p(x)$ es la función de densidad teórica con la que queremos comparar nuestros datos, el cuantil i-esimo es el valor $x_i$ tal que $\frac{i}{k} = \int_{-\infty}^{x_i} p(x)dx$.

Por tanto, si las distribuciones teórica y experimental coinciden, el gráfico debería seguir la recta $y = x$, al coincidir ambos cuantiles.

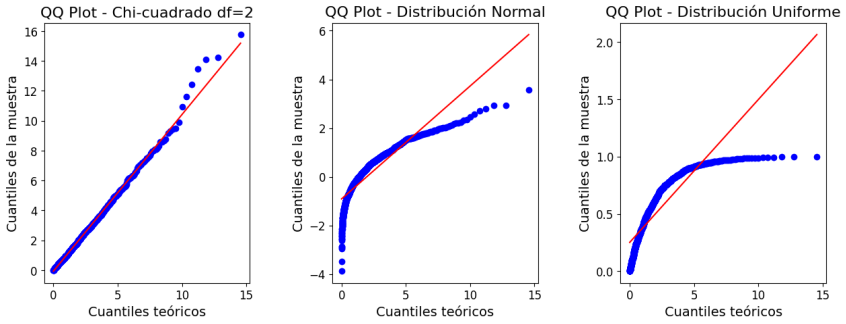



Figura A.3: Comparación de tres distribuciones distintas: chi cuadrado con dos grados de libertad, normal estándar y uniforme $(0, 1)$ con los cuantiles teóricos de la distribución chi cuadrado con dos grados de libertad.

## A.2. Kernel Density Estimation (KDE)

La estimación de una función de distribución con Kernel Density Estimation (KDE) es un método utilizado en estadística para aproximar la función de densidad de probabilidad de una variable aleatoria continua a partir de un conjunto de datos observados. Estos métodos utilizan una técnica no paramétrica que se basa en la colocación de **kernels** suaves alrededor de cada punto de datos de forma que la suma de todas estas funciones forme una estimación suave de la distribución subyacente.

Un kernel en el contexto de KDE es una función comúnmente representada por $K(u)$ que asigna un peso a cada punto del espacio dependiendo de su proximidad a los datos. De esta forma, la contribución de cada punto de los datos a la estimación de la densidad en un punto particular $x$ viene dada por $K\left(\frac{x-x_i}{h}\right)$, donde $x_i$ es el punto de datos, $h$ es el ancho de banda (una medida de suavidad), y $K(\cdot)$ es el kernel. Los kernels típicamente tienen forma de campana y son funciones suaves y simétricas. Los más comunes son el kernel gaussiano $K(u) = \frac{1}{\sqrt{2\pi}} e^{-u^2/2}$ y el kernel uniforme $K(u) = \frac{1}{2}$ para $|u| \leq 1$ y $K(u) = 0$ para $|u| > 1$.

## A.3. Generación de grafos según el Modelo Erdös–Rényi (ER).

El modelo Erdős-Rényi es un modelo clásico de teoría de grafos que fue propuesto por Paul Erdős y Alfréd Rényi en 1959. Este modelo aleatorio se utiliza para generar grafos de manera probabilística. En el modelo Erdős-Rényi, $G(n, p)$,se especifica el número de nodos del grafo y la probabilidad $(p)$ de que exista una arista entre cualquier par de nodos. En un grafo con $n$ nodos, existen un total de $\binom{n}{2} = \frac{n(n-1)}{2}$ y por tanto el número de aristas esperado es $m = p\frac{n(n-1)}{2}$, aunque no tiene porque ser ese exactamente. De esta forma, existe otra forma de generar este tipo de grafos, $G(n, m)$, se crea un grafo con $n$ nodos y $m$ aristas exactamente, elegidas aleatoriamente entre todas las posibles combinaciones de pares de nodos.

# Apéndice B

# Códigos utilizados

A la hora de desarrollar el software para la elaboración de los programas se ha utilizado como lenguaje ***Python*** y todo ello desarrollado en el entorno ***Visual Studio Code***. Además para su realización, se han consultado los siguientes paquetes junto a su documentación:

- Numpy - NumPy Contributors, Enlace.
- Matplotlib - Matplotlib Development Team, Enlace.
- Pandas - Pandas Development Team, Enlace.
- Gaussian_kde - SciPy Developers, Enlace.
- Linear_regression - SciPy Developers, Enlace.
- Minimize - SciPy Developers, Enlace
- Probability Plot (ProbPlot) - SciPy Developers, Enlace
- Graphs (networks) - NetworkX Developers, Enlace

Con el objetivo de hacerlo, hay que diferenciar tres componentes principales:

- Esta primera parte consiste en llevar a cabo la evolución del sistema cuántico que haría el ordenador en caso de disponer de ello. Su función es básicamente obtener ese estado $|\gamma, \theta\rangle$ a través del producto matricial de los respectivos operadores.
- La segunda parte, consiste en la optimización de la función de conste. Aspecto que efectivamente si es trabajo del ordenador clásico. Todo ello con el objetivo de obtener el valor óptimo de los ángulos y pasárselos posteriormente al ordenador cuántico y medir esa energía mínima.
- La última parte consiste en un análisis estadístico y presetación de los resultados.

## B.1. Iniciación en los Algoritmos QAOA: *1 qubit and 1 layer*

### B.1.1. Funciones algortimo QAOA

```
import numpy as np
import math as mt
import scipy.linalg as LA
import pandas as pd
from tqdm import tqdm
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import random
from scipy.optimize import minimize


# Matrices identidad y Pauli
identity = np.eye(2)
sigma_x = np.array([[0, 1], [1, 0]])
sigma_z = np.array([[1, 0], [0, -1]])

# Matriz asociada a la puerta Hadamard
Hadamard = (1/np.sqrt(2)) * (sigma_x + sigma_z)
```

Listing B.1: Importación de los paquetes que se van a usar y definición de las constantes y matrices fundamentales que se van a usar.

```
def Rot_x(theta):
    return mt.cos(theta/2) * identity - 1j * mt.sin(theta/2) * sigma_x
```

Listing B.2: Función que devuelve la matriz de rotación del ángulo theta entorno al eje X.

```
def Hamiltonian(omega, delta):
    return -omega * sigma_z
```

Listing B.3: Función que devuelve el Hamiltoniano deseado dado el valor de la energía máxima para un único qubit

```
def layer_QAOA(gamma, theta, H, ket_inicial):
    ket_final = Rot_x(theta) @ LA.expm(-1j * gamma * H) @ ket_inicial
    return ket_final
```

Listing B.4: Función que devuelve el estado final de un ket tras atravesar una capa de QAOA dado los ángulos caracterisitcos de estas

```
def Cost_Prob_UP(ket_QAOA):
    up_ket = np.array([[1], [0]])
    up_bra = np.conjugate(up_ket).T

    return (np.abs(up_bra @ ket_QAOA)**2).item()
```

Listing B.5: Función que devuelve la probabilidad de encontrar el qubit en el estado \ket{0}

```
def Cost_Prob_DOWN(ket_QAOA):
    up_ket = np.array([[0], [1]])
```

```
    up_bra = np.conjugate(up_ket).T

    return (np.abs(up_bra @ ket_QAOA)**2).item()
```

Listing B.6: Función que devuelve la probabilidad de encontrar el qubit en el estado \ket{1}

```
def Cost_Prob_E(ket_QAOA, Ham):
    return (np.real(np.conjugate(ket_QAOA).T @ Ham @ ket_QAOA)).item()
```

Listing B.7: Funciónque devuelve la energia esperada del estado y hamiltoniano introducidos

```
def fun_minimize(angle):
    """
    Funcion para minimizar la energia media.
    """
    ket_final = layer_QAOA(angle[0], angle[1], Ham, ket_inicial)
    return Cost_Prob_E(ket_final, Ham)

def minimo(gamma, theta):
    """
    Encuentra los angulos optimos para maximizar la probabilidad de los
    estados.
    """
    x0 = np.array([gamma, theta])
    res = minimize(fun_minimize, x0,
                   method='L-BFGS-B',
                   bounds=[(-mt.pi/2, mt.pi/2), (0.0, mt.pi)])
    ket_final = layer_QAOA(res.x[0], res.x[1], Ham, ket_inicial)
    ang = np.array([res.x[0], res.x[1]])
    return ang, res.fun, abs(layer_QAOA(ang[0], ang[1], Ham, ket_inicial)
    [0])**2, abs(layer_QAOA(ang[0], ang[1], Ham, ket_inicial)[1])**2
```

Listing B.8: Funciones que minimizan la energía del Hamiltoniano

### B.1.2. Obtención de datos

```

# Definicion del conjunto de valores para las graficas
iteration = 100
theta_vec = np.linspace(0, 2*np.pi, iteration)
gamma_vec = np.linspace(-np.pi, np.pi, iteration)

# Creamos un DataFrame para almacenar los datos
data = pd.DataFrame(columns=['gamma_vec',
                             'theta_vec',
                             'Ket_QAOA',
                             'Cost_Prob_UP',
                             'Cost_Prob_DOWN',
                             'Cost_Prob_E'])

# Generamos los datos para las graficas y los almacenamos en el DataFrame
for gamma in tqdm(gamma_vec):
    for theta in theta_vec:
```

```
        ket_QAOA = layer_QAOA(gamma, theta, Ham, ket_inicial)
        nueva_fila = pd.DataFrame({'gamma_vec': [gamma],
                                   'theta_vec':[theta],
                                   'Ket_QAOA':[ket_QAOA],
                                   'Cost_Prob_UP':[Cost_Prob_UP(ket_QAOA)
    ],
                                   'Cost_Prob_DOWN':[Cost_Prob_DOWN(
    ket_QAOA)],
                                   'Cost_Prob_E': [Cost_Prob_E(ket_QAOA,
    Ham)]})
        data.loc[len(data)] = nueva_fila.iloc[0]
```

Listing B.9: Creacion de los datos de energía probabilidad UP y DOWN para diferentes valores de ángulos

### B.1.3. Análisis y visualización de resultados

```
# Creamos las matrices para las graficas
z1 = data['Cost_Prob_E'].values.reshape(iteration, iteration)
z2 = data['Cost_Prob_UP'].values.reshape(iteration, iteration)
z3 = data['Cost_Prob_DOWN'].values.reshape(iteration, iteration)
y, x = np.meshgrid(theta_vec, gamma_vec)

# Creamos las figuras y las graficas 3D
fig = plt.figure(figsize=(15, 5))

# Primera grafica 3D
ax1 = fig.add_subplot(131, projection='3d')
ax1.plot_surface(y, x, z1, cmap='inferno')
ax1.set_title('Energia media')
ax1.text(0, -3.75, 4, s = f'a={omega}', color='black', fontsize=12)
ax1.set_ylabel('gamma')
ax1.set_xlabel('thete')

# Segunda grafica 3D
ax2 = fig.add_subplot(132, projection='3d')
ax2.plot_surface(y, x, z2, cmap='inferno')
ax2.set_title('Prob Ground State')
ax2.text(0, -4, 1.5, s = f'a={omega}', color='black', fontsize=12)
ax2.set_ylabel('gamma')
ax2.set_xlabel('theta')

# Tercera grafica 3D
ax3 = fig.add_subplot(133, projection='3d')
ax3.plot_surface(y, x, z3, cmap='inferno')
ax3.set_title('Prob Excitated state')
ax3.set_ylabel('gamma')
ax3.set_xlabel('theta')
ax3.text(0, -4, 1.5, s = f'a={omega}', color='black', fontsize=12)

# Mostramos la grafica
```

```
plt.show()
```

Listing B.10: Creación de las gráficas

```
n = 3  # numero de filas
m = 4  # numero de columnas

numeros = np.array(range(iteration))
index = random.sample(list(numeros), n*m)
index = np.array(index)

# Crear una figura con una matriz de subgraficos
fig, axs = plt.subplots(n, m, figsize=(13, 8))

# Generar y mostrar gr\'aficas en bucle
for i in range(n):
    for j in range(m):
        df = data[data['gamma_vec'] == gamma_vec[index[3*i+j]]][['
    Cost_Prob_E','theta_vec']]
        y = df['Cost_Prob_E'].values
        x = df['theta_vec'].values

        # Mostrar la gr\'afica actual en la posici\'on (i, j)
        axs[i, j].plot(x, y,c='Red')
        axs[i, j].set_ylim(data['Cost_Prob_E'].min()-0.1*(data['
    Cost_Prob_E'].max()-data['Cost_Prob_E'].min()),data['Cost_Prob_E'].max
    ()+0.1*(data['Cost_Prob_E'].max()-data['Cost_Prob_E'].min()))
        axs[i, j].set_title(f'gamma = {gamma_vec[index[3*i+j]]}')

# Ajustar dise\~no y mostrar las graficas
plt.tight_layout()
plt.show()
```

Listing B.11: Graficas para visualizar como se comportan el valor medio de la energía para gamma fijo

```

fig, axs = plt.subplots(n, m, figsize=(13, 8))

# Generar y mostrar graficas en bucle
for i in range(n):
    for j in range(m):
        df = data[data['theta_vec'] == theta_vec[index[3*i+j]]][['
    Cost_Prob_E','gamma_vec']]
        y = df['Cost_Prob_E'].values
        x = df['gamma_vec'].values

        # Mostrar la grafica actual en la posicion (i, j)
        axs[i, j].plot(x, y,c='Red')
        axs[i, j].set_ylim(data['Cost_Prob_E'].min()-0.1*(data['
    Cost_Prob_E'].max()-data['Cost_Prob_E'].min()),data['Cost_Prob_E'].max
    ()+0.1*(data['Cost_Prob_E'].max()-data['Cost_Prob_E'].min()))
        axs[i, j].set_title(f'theta = {theta_vec[index[3*i+j]]}')

```

```
# Ajustar dise\~no y mostrar las graficas
plt.tight_layout()
plt.show()
```

Listing B.12: Graficas para visualizar como se comportan el valor medio de la energía para theta fijo

## B.2. Algoritmos QAOA para un mayor n umero de qubits: N qubits and 1 layer

### B.2.1. Funciones algortimo QAOA

```
import numpy as np
import networkx as nx
import scipy.sparse as sp
import math as mt
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from scipy.linalg import solve_banded
```

Listing B.13: Importación de paquetes

```
def Asp_big(matrix, L, index):
    if index == 0:
        A_big = sp.kron(matrix, sp.eye(2 ** (L - 1)))
    elif index == L - 1:
        A_big = sp.kron(sp.eye(2 ** (L - 1)), matrix)
    else:
        lf = index
        rg = L - 1 - index
        A_big = sp.kron(sp.eye(2 ** (lf)), sp.kron(matrix, sp.eye(2 ** (rg
    ))))
    return A_big
```

Listing B.14: Función que calcula el producto tensorial de la matriz indicada con la identidad para L qubits

```
def H_ZZ_target(L, J_array, Bz, gz, J=-1):
    sz = sp.csr_matrix(np.array([[1, 0], [0, -1]]))
    H_target = 0
    for nn in range(L):
        for mm in range(L):
            Jeff = J * J_array[nn, mm]
            H_target += 0.5 * Jeff * Asp_big(sz, L, nn) @ Asp_big(sz, L,
    mm)
    for nz in range(L):
        H_target += gz * Bz[nz] * Asp_big(sz, L, nz)
    return H_target
```

Listing B.15: Función que genera el Hamiltoniano tipo Ising dado el tamaño del sistema la matriz de los $J_{ij}$ y los $H_i$

```
def U_NRot(L, direction, angle):
    sx = sp.csr_matrix(np.array([[0, 1], [1, 0]]))
    sy = sp.csr_matrix(np.array([[0, -1j], [1j, 0]]))
    sz = sp.csr_matrix(np.array([[1, 0], [0, -1]]))
    iden = sp.csr_matrix(np.array([[1, 0], [0, 1]]))
    if direction == 1:
        matrix = iden * np.cos(angle / 2) - 1j * np.sin(angle / 2) * sx
    elif direction == 2:
        matrix = iden * np.cos(angle / 2) - 1j * np.sin(angle / 2) * sy
    else:
        matrix = iden * np.cos(angle / 2) - 1j * np.sin(angle / 2) * sz
    int_matrix = matrix
    for nz in range(L - 1):
        mat = sp.kron(int_matrix, matrix)
        int_matrix = mat
    UbigRot = mat
    return UbigRot
```

Listing B.16: Función que realiza una rotación individual en todos los qubits en un ángulo dado en la dirección indicada

```
def Adj_Erdos(L, rho):
    m = np.ceil(0.5 * rho * L * (L - 1))
    order_nodes = np.arange(0, L)
    Net = nx.gnm_random_graph(L, m, directed=False)
    Adj_Erd = nx.attr_matrix(Net, rc_order=list(order_nodes))
    return Adj_Erd
```

Listing B.17: Función que genera la matriz de adyacencia para un grafo tipo Erdös–Rényi

```
def H_RandIsingNot(L, graph, sigma, mu=0):

    Jnm_rand = np.random.normal(mu, sigma, size=(L, L))
    Jnm_rand_triang = np.triu(Jnm_rand, k=1)
    Jnm_sym = Jnm_rand_triang + Jnm_rand_triang.T

    J_array = Jnm_sym * graph

    return H_ZZ_target(L, J_array, np.zeros(L), 0)
```

Listing B.18: Función para generar el hamiltoniano tipo Ising sin campo magnético transversal dado el grafo que le representa.

```
def H_RandIsing(L, graph, sigma, mu=0):


    Jnm_rand = np.random.normal(mu, sigma, size=(L, L))
    Jnm_rand_triang = np.triu(Jnm_rand, k=1)
    Jnm_sym = Jnm_rand_triang + Jnm_rand_triang.T

    J_array = Jnm_sym * graph

    Bz = np.random.normal(mu, sigma, size=L)

```

```
    return H_ZZ_target(L, J_array, Bz, 1)
```

Listing B.19: Función para generar el hamiltoniano tipo Ising con campo magnético transversal dado el grafo que le representa.

```
def H_MaxCut(L, graph):
    J_array = graph
    sz = sp.csr_matrix(np.array([[1, 0], [0, -1]]))
    H_target = 0
    for nn in range(L):
        for mm in range(L):
            if nn == mm:
                Jeff = J_array[nn, mm]
                H_target +=  Jeff * Asp_big(sz, L, nn) @ Asp_big(sz, L, mm
    )
            else:
                Jeff = 0
                for nnn in range(L):
                    for mmm in range(L):
                        Jeff = Jeff + J_array[nnn, mmm]+ J_array[mmm, nnn]
                H_target +=  Jeff * Asp_big(sz, L, nn)
    return H_target
```

Listing B.20: Función que genera el Hamiltoniano para el problema MaxCut con diferentes pesos enteros las aristas.

```
def IsingEnergies(H):
    if isinstance(H, np.ndarray):
        return np.diag(H, 0)
    elif isinstance(H, sp.spmatrix):
        return H.diagonal(0)
    else:
        raise Exception(f"Unknown Hamiltonian matrix format in
    IsingEnergies:\n{H}")
```

Listing B.21: Función que calcula las energías del Hamiltoniano

```
def Norm_psi(psi):
    return psi / sum(abs(psi) ** 2)
```

Listing B.22: Función que normaliza un vector

```
def random_QAOA_initial_conditions():
    theta_i = 1.0 + (np.random.rand() - 0.5) * 10 ** (-1)
    gamma_i = 0.5 + (np.random.rand() - 0.5) * 10 ** (-2)
    return [theta_i, gamma_i]
```

Listing B.23: Función que genera condiciones iniciales aleatorias para la optimización de QAOA

```
def QAOA(theta, gamma, Nq, En):
    phi_int = np.ones(2**Nq) / np.sqrt(2**Nq)
    phi_QAOA = U_NRot(Nq, 1, theta) @ np.exp(-1j * En * gamma) * phi_int
    return np.abs(phi_QAOA) ** 2
```

Listing B.24: Función que ejecuta una capa de QAOA devolviendo las porbabilidades individuales.

```python
def QAOA_success(theta, gamma, Nq, En, ground_state_indices):
    phi_QAOA = QAOA(theta, gamma, Nq, En)
    success_probability = np.sum(phi_QAOA[ground_state_indices])
    return success_probability
```

Listing B.25: Función que calcula la probabilidad de que el estado final de QAOA este en el estado base.

```python
def QAOA_energy(theta, gamma, Nq, En):
    phi_QAOA = QAOA(theta, gamma, Nq, En)
    return np.dot(En, phi_QAOA)
```

Listing B.26: Función que calcula la energía media al final de QAOA

```python
def Opt_QAOA_Ener(Nq, En, start=None, **kwargs):
    if start is None:
        start = random_QAOA_initial_conditions()
    res = minimize(
        lambda x: QAOA_energy(x[0], x[1], Nq, En),
        start,
        method="L-BFGS-B",
        **kwargs,
    )
    return res.x, res.fun, QAOA(res.x[0], res.x[1], Nq, En)
```

Listing B.27: Función que minimiza la energía de QAOA

### B.2.2. Obtención de datos

```python
Nq = 10        # Tamano del sistema
rho = 0.5      # Densidad de aristas para el grafo de Erdos-Renyi
sigma = 1      # Varianza de la distribucion normal
itt = 100      # Numero de muestreos

phi_int = np.ones(2**Nq) / np.sqrt(2**Nq)
phi_1L_it = []
opt_ang_1L = []
En_iterations = []
Ex_ene_1L = []

# Bucle de iteraciones
for it in tqdm(range(itt)):
    # Generacion de un grafo de Erdos-Renyi
    graph = Adj_Erdos(Nq, rho)
    # Generacion de la Hamiltoniana para el modelo de Ising aleatorio
    HA = H_RandIsing(Nq, graph, sigma)
    # Obtencion de las energias de la Hamiltoniana
    En = IsingEnergies(HA)
    # Optimizacion de los angulos del QAOA
    angles_1L, E_1L, psi_1L = Opt_QAOA(Nq, En)
    # Almacenamiento de resultados
    opt_ang_1L.append(angles_1L)
```

```
24      En_iterations.append(En)
25      phi_1L_it.append(psi_1L)
26      Ex_ene_1L.append(E_1L)
```

Listing B.28: Generar los datos que se van a utilizar para estudiar este modelo. consiste en 100 muestreos diferentes con hamiltonianos tipo Ising generados aleatoriamente segun una distribución normal estandar.

### B.2.3. Análisis de resultados

```
1  import numpy as np
2  import scipy.sparse as sp
3  import math as mt
4  import networkx as nx
5  import matplotlib.pyplot as plt
6  import matplotlib.transforms as transforms
7  from tqdm import tqdm
8  import pandas as pd
9
10 from matplotlib.patches import Ellipse
11 from scipy.stats import gaussian_kde, linregress
12 from scipy.optimize import minimize, curve_fit
13 from scipy.sparse.linalg import eigs
14
15 from Main_Func import *
16
17 from matplotlib import style
18 from scipy import stats
19 from sklearn.neighbors import KernelDensity
20 from sklearn.model_selection import GridSearchCV
21 from sklearn.model_selection import LeaveOneOut
22
23 from sklearn import mixture
24 from sklearn import datasets
```

Listing B.29: Importación y ajustes iniciales.

```
1  def hamming_distance(num1, num2):
2      # Convertir los numeros a su representacion binaria y eliminar el
       prefijo '0b' y en string type
3      bin_num1 = bin(num1)[2:]
4      bin_num2 = bin(num2)[2:]
5
6      # Rellenar con ceros a la izquierda para que tengan la misma longitud
7      max_len = max(len(bin_num1), len(bin_num2))
8      bin_num1 = bin_num1.zfill(max_len)
9      bin_num2 = bin_num2.zfill(max_len)
10
11     # Calcular la distancia de Hamming contando los bits diferentes
12     hamming_dist = sum(bit1 != bit2 for bit1, bit2 in zip(bin_num1,
       bin_num2))
13
```

```
    return hamming_dist
```

Listing B.30: Funcion que te calcula la Hamming Distance de dos numeros en su representación binaria.

```
def mahalanobis_distance_squared(x, mean, covariance):
    # Calcular la diferencia entre el punto y la media
    diff = x - mean

    # Calcular la inversa de la matriz de covarianza
    cov_inv = np.linalg.inv(covariance)

    # Calcular la distancia de Mahalanobis al cuadrado
    distance_squared = np.dot(np.dot(diff.T, cov_inv), diff)

    return distance_squared
```

Listing B.31: Función para calcular la distancia de Mahalanobis al cuadrado.

```
def confidence_ellipse(x, y, ax, n_std=3.0, facecolor='none', **kwargs):
    """
    Crea un grafico de la elipse de confianza de la covarianza entre *x* e
    *y*.
    """
    if x.size != y.size:
        raise ValueError("x and y must be the same size")

    cov = np.cov(x, y)
    pearson = cov[0, 1]/np.sqrt(cov[0, 0] * cov[1, 1])
    # Usando un caso especial para obtener los autovalores de este
    # conjunto de datos bidimensional.
    ell_radius_x = np.sqrt(1 + pearson)
    ell_radius_y = np.sqrt(1 - pearson)
    ellipse = Ellipse((0, 0), width=ell_radius_x * 2, height=ell_radius_y
   * 2,
                      facecolor=facecolor, alpha = 0.3, **kwargs)

    # Calculando la desviacion estandar de x a partir de
    # la raiz cuadrada de la varianza y multiplicando
    # por el numero dado de desviaciones estandar.
    scale_x = np.sqrt(cov[0, 0]) * n_std
    mean_x = np.mean(x)

    # Calculando la desviacion estandar de y ...
    scale_y = np.sqrt(cov[1, 1]) * n_std
    mean_y = np.mean(y)

    transf = transforms.Affine2D() \
        .rotate_deg(50) \
        .scale(scale_x, scale_y) \
        .translate(mean_x, mean_y)

    ellipse.set_transform(transf + ax.transData)
```

```
    return ax.add_patch(ellipse)
```

Listing B.32: Función para graficar la elipse de confianza de la covarianza entre dos variables normales.

```
Nq2 = 10
rho2 = 0.2
sigma2 = 0.5

graph_rand = Adj_Erdos(Nq2, rho2)
HA_rand = H_RandIsing(Nq2, graph_rand, sigma2)
Energia_rand = IsingEnergies(HA_rand)

i = np.where(Energia_rand == Energia_rand.min())[0][0] + 1
i = 2 ** Nq2 - i

Hamming_dist_min = []

for j in range(2 ** Nq2):
    Hamming_dist_min.append(hamming_distance(i, 2 ** Nq2 - j - 1))

fig, ax1 = plt.subplots(1, 1)
ax1.scatter(Energia_rand, Hamming_dist_min, s=5, c='b')
ax1.set_xlabel('Energia')
ax1.set_ylabel('Hamming distance')
ax1.set_ylim(-1, 13)
```

Listing B.33: Obtención y representación de la distribución discreta de Energias y distancias de Hamming para el estado base.

```
# Ajuste del modelo KDE
# ================================================================
modelo_kde = KernelDensity(kernel='gaussian', bandwidth=1)
modelo_kde.fit(X=datos)

# Mapa de densidad de probabilidad
# ================================================================
fig, ax1 = plt.subplots(nrows=1, ncols=1, figsize=(6,6))

# Grid de valores dentro del rango observado (2 dimensiones)
x = np.linspace(min(data.Energia), max(data.Energia), 200)
y = np.linspace(min(data.HammingDistance), max(data.HammingDistance), 200)
xx, yy = np.meshgrid(x, y)
grid = np.column_stack((xx.flatten(), yy.flatten()))

# Densidad de probabilidad de cada valor del grid
log_densidad_pred = modelo_kde.score_samples(grid)
densidad_pred = np.exp(log_densidad_pred)

ax1.scatter(grid[:,0], grid[:,1], alpha=0.6, c=densidad_pred)
ax1.set_title('Funcion de densidad empirica')
ax1.set_xlabel('Energia')
ax1.set_ylabel('Hamming Distance')
ax1.set_ylim(0,12)
```

```
ax1.set_xlim(Energia_rand.min(),Energia_rand.max())

# Representacion 3D
# ========================================================
from mpl_toolkits.mplot3d import axes3d

plt.style.use('default')
fig = plt.figure(figsize=(5, 5))
ax = plt.axes(projection='3d')
#ax.view_init(60, 35)
ax.plot_surface(xx,yy, densidad_pred.reshape(xx.shape), cmap='viridis')
ax.set_xlabel('Energia')
ax.set_ylabel('Hamming distance')
ax.set_zlabel('Densidad empirica')
plt.show()
plt.style.use('ggplot');
```

Listing B.34: Ajuste de los daots au¡ un modelo Kernel Density Estimation (KDE) es decir densidad estimada y mapa de densidad de probabilidad.

```

from sklearn import mixture
from sklearn import datasets
y_pred_empirico = gm.sample(2**Nq2)

# Mapa de densidad de probabilidad
# ========================================================

fig, ax1 = plt.subplots(nrows=1, ncols=1, figsize=(5,5))

confidence_ellipse(y_pred_empirico[0][:,0], y_pred_empirico[0][:,1], ax1,
    n_std=1.9, facecolor='b')

ax1.scatter(y_pred_empirico[0][:,0], y_pred_empirico[0][:,1], alpha=0.6, c
    ='b', s=1, label='Gaussiana Bivariante')

ax1.scatter(data.Energia, data.HammingDistance, alpha=0.6, c='r', s=1,
    label='Datos reales')
ax1.legend()
ax1.set_xlabel('Energia')
ax1.set_ylabel('Hamming Distance')
```

Listing B.35: Comparacion de la densidad experimental y la normal multivariante estimada.

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(15, 5))

# Graficar cada serie de datos en su respectivo subgrafico
ax1.scatter(y1[:,0], y1[:,1], color='blue', s=5, alpha=0.6)
ax1.set_xlabel('Energia')
ax1.set_ylabel('Hamming Distance')
ax1.set_title('Gaussiana')

ax2.scatter(y2[:,0], y2[:,1], color='green', s=5, alpha=0.6)
ax2.set_xlabel('Energia')
```

```
ax2.set_ylabel('Hamming Distance')
ax2.set_title('Experimental')

ax3.scatter(y3[:,0], y3[:,1], color='red', s=5, alpha=0.6)
ax3.set_xlabel('Energia')
ax3.set_ylabel('Hamming Distance')
ax3.set_title('Uniforme')

# Mostrar la figura
plt.show()
```

Listing B.36: Gráficos de dispersión de los datos en tres subgráficos.

```
cov1 = gm.covariances_[0]
meas1 = gm.means_[0]

cov2 = np.cov(Energia_rand, Hamming_dist_min)
meas2 = np.array([np.mean(Energia_rand), np.mean(Hamming_dist_min)])

y1 = gm.sample(2**Nq2)[0]
y2 = datos
a = Energia_rand.min()
b = Energia_rand.max()

# Generar datos para la primera columna (distribucion uniforme)
c1 = np.random.uniform(a, b, 2**Nq2).reshape(-1, 1)
c2 = np.random.uniform(0, 10, 2**Nq2).reshape(-1, 1)

# Concatenar las dos columnas para formar el array de 2 columnas
y3 = np.concatenate((c1, c2), axis=1)

dist1 = []
dist2 = []
dist3 = []
for i in range(2**Nq2):
    d = mahalanobis_distance_squared(y1[i], meas1, cov1)
    dist1.append(d)
    d = mahalanobis_distance_squared(y2[i], meas2, cov2)
    dist2.append(d)
    d = ((y3[i][0] - meas1[0])**2) / cov1[0][0] + ((y3[i][1] - meas1[1])
   **2) / cov1[1][1]
    dist3.append(d)

fig,(ax1,ax2,ax3) = plt.subplots(nrows=1, ncols=3, figsize=(16,5))

ax1.hist(dist1, bins=20, color='#1E90FF')
ax1.set_title('Gaussiana')
ax1.set_xlabel('Distancia de Mahalanobis')
ax2.set_xlabel('Distancia de Mahalanobis')
ax3.set_xlabel('Distancia de Mahalanobis')
ax1.set_ylabel('Frecuencia')
ax2.hist(dist2, bins=20 , color='#006400', alpha=0.7)
ax2.set_title('Experimental')
ax3.hist(dist3, bins=20, color='#8B0000', alpha=0.8)
```

```
ax3.set_title('Uniforme')
ax1.set_ylim(0,1500)
ax2.set_ylim(0,1500)
ax3.set_ylim(0,1500)
```

Listing B.37: Cálculo de las distancias de Mahalanobis cuadradas y su representación en histogramas.

```
# Importar la funcion chi2 desde scipy.stats
from scipy.stats import chi2

# Calcular los cuantiles de la distribucion chi cuadrado
cuantiles_chi2 = chi2.ppf(np.linspace(0.01, 0.99, 100), df=2)

# Crear las figuras y los subgraficos
fig, axs = plt.subplots(1, 3, figsize=(15, 5))

# Graficar los Q-Q plots para cada conjunto de datos
for ax, datos, titulo in zip(axs, [dist1, dist2, dist3], ['Gaussiana', '
    Experimental', 'Uniforme']):
    (osm, osr), (slope, intercept, r) = probplot(datos, dist="chi2",
    sparams=(2), plot=ax)
    ax.plot(osm, osm * slope + intercept, color='red', label='Recta de
    regresion')
    ax.plot(osm, osm, color='blue', linestyle='--', label='Identidad')
    ax.set_title(titulo)
    ax.legend()

# Ajustar el espaciado entre subgraficos
plt.tight_layout()

# Mostrar la figura
plt.show()
```

Listing B.38: Test de hipotesis mediante gráficos Q-Q plots para cada conjunto de datos con ajuste de recta de regresión.

```
cov = []
En_min = []

for jjj in tqdm(range(len(En_definitivo))):
    Energia_rand = En_definitivo[jjj]
    min_valor = np.min(Energia_rand)
    max_valor = np.max(Energia_rand)
    Energia_rand = (Energia_rand - min_valor) / (max_valor - min_valor)

    for jj in range(2**Nq):
        Ener = Energia_rand[jj]
        i = np.where(Energia_rand == Ener)[0][0] + 1
        i = 2**Nq - i

        Hamming_dist_min = []
        for j in range(2**Nq):
            Hamming_dist_min.append(hamming_distance(i, 2**Nq - j - 1))

```

```
    datos = np.array([Energia_rand, Hamming_dist_min]).transpose()
    gm = mixture.BayesianGaussianMixture(n_components=2, covariance_type='
    full').fit(X=datos)
     #if para seleccionar la varianza que nos interesa
    if gm.covariances_[0][0][1] > 0.001:
        if Ener < 0.5:
            cov.append(abs(gm.covariances_[0][0][1]))
            En_min.append(Ener)
        else:
            cov.append(-abs(gm.covariances_[0][0][1]))
            En_min.append(Ener)
    elif gm.covariances_[1][0][1] < -0.001:
        if Ener < 0.5:
            cov.append(-abs(gm.covariances_[1][0][1]))
            En_min.append(Ener)
        else:
            cov.append(abs(gm.covariances_[0][0][1]))
            En_min.append(Ener)

fig, ax1 = plt.subplots()

# Graficar el scatter plot de los dos vectores en el mismo subgrafico
ax1.scatter(En_min, cov, color='blue')

# Etiquetas y titulo
ax1.set_xlabel('Energia')
ax1.set_ylabel('Eje Y')
ax1.set_title('Grafico de dispersion')

# Mostrar la figura
plt.show()
```

Listing B.39: Cálculo de la relación entre covarianzas de distancias y energías .

```

# Calcular el ajuste de regresion lineal en escala logaritmica
x = En
y1 = psi_1L
y1log = np.log(y1)
slope1, intercept1, r1, p1, se1 = linregress(x, y1log)

# Imprimir la ecuacion de la curva ajustada
print(f'Curve one: p(E) = {slope1}*E + {intercept1}')

# Crear la figura y el subgrafico
fig, ax1 = plt.subplots(1,1)

# Graficar los puntos y la curva ajustada
ax1.scatter(x , y1, s=1)
ax1.plot(x, np.exp(intercept1 + slope1*x), 'k', label=f'beta = {slope1}')

# Configurar la escala del eje y en logaritmica
ax1.set_yscale('log')
ax1.set_ylim(10**-8, 10**0)
```

```
ax1.set_xlabel('Energia')
ax1.set_ylabel('Probabilidad')
ax1.legend()

# Mostrar la figura
plt.show()
```

Listing B.40: Ajuste de regresión lineal en escala logarítmica para la distribución de probabilidad definitiva de energias.

## B.3. Algoritmo QAOA para múltiples capas

### B.3.1. Funciones para 2 dos capas

```
import numpy as np
import networkx as nx
import scipy.sparse as sp
import math as mt

import matplotlib.pyplot as plt

from scipy.optimize import minimize
from scipy.linalg import solve_banded
from Main_Func import *
```

Listing B.41: Importación de paquetes

```

def QAOA_2Layer(theta_1, gamma_1, theta_2, gamma_2, Nq, En):

    psi = np.ones(2**Nq) / np.sqrt(2**Nq)

    # First layer
    psi = U_NRot(Nq, 2, theta_1) @ (np.exp(-1j * En * gamma_1) * psi)

    # Second layer
    psi = U_NRot(Nq, 2, theta_2) @ (np.exp(-1j * En * gamma_2) * psi)

    return np.abs(psi) ** 2
```

Listing B.42: Función que devuelve el estado final tras dos capas de QAOA dados los ángulos caracterísiticos

```

def random_QAOA_2L_initial_conditions():
    theta_i_1 = 1.0 + (np.random.rand() - 0.5)
    gamma_i_1 = 1.0 + (np.random.rand() - 0.5)

    theta_i_2 = 1.0 + (np.random.rand() - 0.45)
    gamma_i_2 = 1.0 + (np.random.rand() - 0.45)

```

```
    return [theta_i_1, gamma_i_1, theta_i_2, gamma_i_2]
```

Listing B.43: Función que devuelve unas condiciones iniciales para los ángulos del algoritmo QAOA.

```

def QAOA_2L_energy(theta_1, gamma_1, theta_2, gamma_2, Nq, En):
    phi_QAOA = QAOA_2Layer(theta_1, gamma_1, theta_2, gamma_2, Nq, En)
    return np.dot(En, phi_QAOA)
```

Listing B.44: Función que devuelve le energía media del estado final

```

def Opt_QAOA_2L_Ener(Nq, En, start=None, **kwargs):
    if start is None:
        start = random_QAOA_2L_initial_conditions()

    res = minimize(
        lambda x: QAOA_2L_energy(x[0], x[1], x[2], x[3], Nq, En),
        start,
        method="L-BFGS-B",
        **kwargs,
    )
    x = res.x
    return x, res.fun, QAOA_2Layer(x[0], x[1], x[2], x[3], Nq, En)
```

Listing B.45: Función que minimiza la energía.

```

def QAOA_success_2L(theta_1, gamma_1, theta_2, gamma_2, Nq, En,
   ground_state_indices):
    phi_QAOA = QAOA_2Layer(theta_1, gamma_1, theta_2, gamma_2, Nq, En)
    success_probability = np.sum(phi_QAOA[ground_state_indices])
    return success_probability
```

Listing B.46: Función que devuelve la probabilidad del ground state.

### B.3.2. Funciones para 3 capas

```
import numpy as np
import networkx as nx
import scipy.sparse as sp
import math as mt

import matplotlib.pyplot as plt

from scipy.optimize import minimize
from scipy.linalg import solve_banded
from Main_Func import *
```

Listing B.47: Importación de paquetes

```
def QAOA_3Layer(theta_1, gamma_1, theta_2, gamma_2, theta_3, gamma_3, Nq,
   En):
```

```
2
3       psi = np.ones(2**Nq) / np.sqrt(2**Nq)
4
5       # First layer
6       psi = U_NRot(Nq, 2, theta_1) @ (np.exp(-1j * En * gamma_1) * psi)
7
8       # Second layer
9       psi = U_NRot(Nq, 2, theta_2) @ (np.exp(-1j * En * gamma_2) * psi)
10
11      # Third layer
12      psi = U_NRot(Nq, 2, theta_3) @ (np.exp(-1j * En * gamma_3) * psi)
13
14      return np.abs(psi) ** 2
```

Listing B.48: Función que devuelve el estado final tras tres capas de QAOA.

```
1 def random_QAOA_3L_initial_conditions():
2     theta_i_1 = 1.0 + (np.random.rand() - 0.5)
3     gamma_i_1 = 1.0 + (np.random.rand() - 0.5)
4
5     theta_i_2 = 1.0 + (np.random.rand() - 0.45)
6     gamma_i_2 = 1.0 + (np.random.rand() - 0.45)
7
8     theta_i_3 = 1.0 + (np.random.rand() - 0.45)
9     gamma_i_3 = 1.0 + (np.random.rand() - 0.45)
10
11    return [theta_i_1, gamma_i_1, theta_i_2, gamma_i_2, theta_i_3,
      gamma_i_3]
```

Listing B.49: Función que devuelve unas condiciones iniciales aleatorios para los ángulos del algoritmo QAOA.

```
1 def QAOA_3L_energy(theta_1, gamma_1, theta_2, gamma_2, theta_3, gamma_3,
      Nq, En):
2     phi_QAOA = QAOA_3Layer(theta_1, gamma_1, theta_2, gamma_2, theta_3,
      gamma_3, Nq, En)
3     return np.dot(En, phi_QAOA)
```

Listing B.50: Función que devuelve la energía media del estado final.

```
1 def Opt_QAOA_3L_Ener(Nq, En, start=None, **kwargs):
2     if start is None:
3         start = random_QAOA_3L_initial_conditions()
4
5     res = minimize(
6         lambda x: QAOA_3L_energy(x[0], x[1], x[2], x[3], x[4], x[5], Nq,
      En),
7         start,
8         method="L-BFGS-B",
9         **kwargs,
10    )
11    x = res.x
12    return x, res.fun, QAOA_3Layer(x[0], x[1], x[2], x[3], x[4], x[5], Nq,
      En)
```

Listing B.51: Función que minimiza la energía ne función de los ángulos.

```
def QAOA_success_3L(theta_1, gamma_1, theta_2, gamma_2, theta_3, gamma_3,
    Nq, En, ground_state_indices):
     phi_QAOA = QAOA_3Layer(theta_1, gamma_1, theta_2, gamma_2, theta_3,
    gamma_3, Nq, En)
     success_probability = np.sum(phi_QAOA[ground_state_indices])
     return success_probability
```

Listing B.52: Función que devuelve la probabilidad del estado base.

### B.3.3. Funciones para 4 capas

```
import numpy as np
import networkx as nx
import scipy.sparse as sp
import math as mt

import matplotlib.pyplot as plt

from scipy.optimize import minimize
from scipy.linalg import solve_banded
from Main_Func import *
```

Listing B.53: Importación de paquetes

```
def QAOA_4Layer(theta_1, gamma_1, theta_2, gamma_2, theta_3, gamma_3,
    theta_4, gamma_4, Nq, En):

     psi = np.ones(2**Nq) / np.sqrt(2**Nq)

     # First layer
     psi = U_NRot(Nq, 2, theta_1) @ (np.exp(-1j * En * gamma_1) * psi)

     # Second layer
     psi = U_NRot(Nq, 2, theta_2) @ (np.exp(-1j * En * gamma_2) * psi)

     # Third layer
     psi = U_NRot(Nq, 2, theta_3) @ (np.exp(-1j * En * gamma_3) * psi)

     # Fourth layer
     psi = U_NRot(Nq, 2, theta_4) @ (np.exp(-1j * En * gamma_4) * psi)

     return np.abs(psi) ** 2
```

Listing B.54: Función que devuelve el estado final tras cuatro capas de QAOA.

```
def random_QAOA_4L_initial_conditions():
     theta_i_1 = 1.0 + (np.random.rand() - 0.5)
     gamma_i_1 = 1.0 + (np.random.rand() - 0.5)

     theta_i_2 = 1.0 + (np.random.rand() - 0.45)
     gamma_i_2 = 1.0 + (np.random.rand() - 0.45)

     theta_i_3 = 1.0 + (np.random.rand() - 0.45)
```

```
    gamma_i_3 = 1.0 + (np.random.rand() - 0.45)

    theta_i_4 = 1.0 + (np.random.rand() - 0.45)
    gamma_i_4 = 1.0 + (np.random.rand() - 0.45)

    return [theta_i_1, gamma_i_1, theta_i_2, gamma_i_2, theta_i_3,
   gamma_i_3, theta_i_4, gamma_i_4]
```

Listing B.55: Función que devuelve unas condiciones iniciales para los ángulos del algoritmo QAOA.

```
def QAOA_4L_energy(theta_1, gamma_1, theta_2, gamma_2, theta_3, gamma_3,
   theta_4, gamma_4, Nq, En):
    phi_QAOA = QAOA_4Layer(theta_1, gamma_1, theta_2, gamma_2, theta_3,
   gamma_3, theta_4, gamma_4, Nq, En)
    return np.dot(En, phi_QAOA)
```

Listing B.56: Función que devuelve la energía media del estado final.

```
def Opt_QAOA_4L_Ener(Nq, En, start=None, **kwargs):
    if start is None:
        start = random_QAOA_4L_initial_conditions()

    res = minimize(
        lambda x: QAOA_4L_energy(x[0], x[1], x[2], x[3], x[4], x[5], x[6],
    x[7], Nq, En),
        start,
        method="L-BFGS-B",
        **kwargs,
    )
    x = res.x
    return x, res.fun, QAOA_4Layer(x[0], x[1], x[2], x[3], x[4], x[5], x
   [6], x[7], Nq, En)
```

Listing B.57: Función que minimiza la energía.

```
def QAOA_success_4L(theta_1, gamma_1, theta_2, gamma_2, theta_3, gamma_3,
   theta_4, gamma_4, Nq, En, ground_state_indices):
    phi_QAOA = QAOA_4Layer(theta_1, gamma_1, theta_2, gamma_2, theta_3,
   gamma_3, theta_4, gamma_4, Nq, En)
    success_probability = np.sum(phi_QAOA[ground_state_indices])
    return success_probability
```

Listing B.58: Función que devuelve la probabilidad del estado base.

## B.4. Obtención de datos para múltiples capas.

```
import numpy as np
import scipy.sparse as sp
import math as mt
import networkx as nx
import matplotlib.pyplot as plt
import matplotlib.transforms as transforms
```

```
from tqdm import tqdm

from matplotlib.patches import Ellipse
from scipy.stats import gaussian_kde, linregress
from scipy.optimize import minimize, curve_fit
from scipy.sparse.linalg import eigs

from Main_Func import *
from Main_Func_2layer import *
from Main_Func_3layer import *
from Main_Func_4layer import *

from matplotlib.patches import Ellipse
import matplotlib.transforms as transforms
```

Listing B.59: Importación de paquetes necesarios.

```
Nq = 11   # System size
rho = 0.5   # Edge density for the Erdos-Renyi Graph
sigma =  1   # variance

graph = Adj_Erdos(Nq, rho)
HA = H_RandIsing(Nq, graph, sigma)
En = IsingEnergies(HA)
ground_state_indices = np.where(En == np.min(En))
x = En
```

Listing B.60: Configuración inicial y generación de un único problema TIpo Ising

```
#1 layer
slope1 = 1000000
while slope1 > -0.01:
    angles_1L, E_1L, psi_1L = Opt_QAOA_Ener(Nq, En)
    y1 = psi_1L
    y1log = np.log(y1)
    slope1, intercept1, r1, p1, se1 = linregress(x, y1log)

print(slope1)
```

Listing B.61: Optimización de QAOA de 1 capa.

```
#2 layer
slope2 = 1000000
while slope2 > -0.01:
    angles_2L, E_2L, psi_2L = Opt_QAOA_2L_Ener(Nq, En)
    y2 = psi_2L
    y2log = np.log(y2)
    slope2, intercept2, r2, p2, se2 = linregress(x, y2log)

print(slope2)
```

Listing B.62: Optimización de QAOA de 2 capas.

```
#3 layer
slope3 = 1000000
```

```
while slope3 > -0.01:
    angles_3L, E_3L, psi_3L = Opt_QAOA_3L_Ener(Nq, En)
    y3 = psi_3L
    y3log = np.log(y3)
    slope3, intercept3, r3, p3, se3 = linregress(x, y3log)

print(slope3)
```

Listing B.63: Optimización de QAOA de 3 capas.

```
#4 layer
slope4 = 100000
while slope4 > -0.01:
    angles_4L, E_4L, psi_4L = Opt_QAOA_4L_Ener(Nq, En)
    y4 = psi_4L
    y4log = np.log(y4)
    slope4, intercept4, r4, p4, se4 = linregress(x, y4log)

print(slope4)
```

Listing B.64: Optimización de QAOA de 4 capas.

```
Nq = 6 # System size
rho = 0.5 # Edge density for the Erdos-Renyi Graph
sigma = 1 # variance

itt = 1000 Numero de muestreos

phi_int = np.ones(2**Nq) / np.sqrt(2**Nq)

prob_1L = 0
phi_1L_it = []
opt_ang_1L = []
Ex_ene_1L = []
probabilidades_1L = []

prob_2L = 0
phi_2L_it = []
opt_ang_2L = []
Ex_ene_2L = []
probabilidades_2L = []

prob_3L = 0
phi_3L_it = []
opt_ang_3L = []
Ex_ene_3L = []
probabilidades_3L = []

prob_4L = 0
phi_4L_it = []
opt_ang_4L = []
Ex_ene_4L = []
probabilidades_4L = []

```

```
En_iterations = []

for it in tqdm(range(itt)):

    graph = Adj_Erdos(Nq, rho)
    HA = H_RandIsing(Nq, graph, sigma)
    En = IsingEnergies(HA)
    En = En / np.sqrt(np.sum(En ** 2))
    En_iterations.append(En) # En es las energias del hamiltoniano
    ground_state_indices = np.where(En == np.min(En))

    prob_1L = 0
    prob_2L = 0
    prob_3L = 0
    prob_4L = 0

    # 1 layer
    angles_1L, E_1L, psi_1L = Opt_QAOA_Ener(Nq, En)
    prob_1L = QAOA_success(angles_1L[0], angles_1L[1], Nq, En,
   ground_state_indices)

    opt_ang_1L.append(angles_1L)
    phi_1L_it.append(psi_1L)
    Ex_ene_1L.append(E_1L)
    probabilidades_1L.append(prob_1L)

    # 2 layer
    angles_2L, E_2L, psi_2L = Opt_QAOA_2L_Ener(Nq, En)
    prob_2L = QAOA_success_2L(angles_2L[0], angles_2L[1], angles_2L[2],
   angles_2L[3], Nq, En, ground_state_indices)

    opt_ang_2L.append(angles_2L)
    phi_2L_it.append(psi_2L)
    Ex_ene_2L.append(E_2L)
    probabilidades_2L.append(prob_2L)

    # 3 layer
    angles_3L, E_3L, psi_3L = Opt_QAOA_3L_Ener(Nq, En)
    prob_3L = QAOA_success_3L(angles_3L[0], angles_3L[1], angles_3L[2],
   angles_3L[3], angles_3L[4], angles_3L[5], Nq, En, ground_state_indices)

    opt_ang_3L.append(angles_3L)
    phi_3L_it.append(psi_3L)
    Ex_ene_3L.append(E_3L)
    probabilidades_3L.append(prob_3L)

    # 4 layer
    angles_4L, E_4L, psi_4L = Opt_QAOA_4L_Ener(Nq, En)
    prob_4L = QAOA_success_4L(angles_4L[0], angles_4L[1], angles_4L[2],
   angles_4L[3], angles_4L[4], angles_4L[5], angles_4L[6], angles_4L[7],
   Nq, En, ground_state_indices)

    opt_ang_4L.append(angles_4L)
```

```
    phi_4L_it.append(psi_4L)
    Ex_ene_4L.append(E_4L)
    probabilidades_4L.append(prob_4L)

umbral = 0.1
# Filtrar los valores mayores o iguales que el umbral
probabilidades_1L_fin = [valor for valor in probabilidades_1L if valor >=
    umbral and valor < 1]
probabilidades_2L_fin = [valor for valor in probabilidades_2L if valor >=
    umbral and valor < 1]
probabilidades_3L_fin = [valor for valor in probabilidades_3L if valor >=
    umbral and valor < 1]
probabilidades_4L_fin = [valor for valor in probabilidades_4L if valor >=
    umbral and valor < 1]
```

Listing B.65: Configuración inicial para el estudio de 2000 problemas tipo Ising y su relación con la probabilidad del grund State.

### B.4.1. Análisis de resultados

```
def plot_ellipse_and_color_points(x, y, ax, n_std, facecolor='none', **
    kwargs):

    if x.size != y.size:
        raise ValueError("x and y must be the same size")

    cov = np.cov(x, y)
    pearson = cov[0, 1]/np.sqrt(cov[0, 0] * cov[1, 1])
    # Using a special case to obtain the eigenvalues of this
    # two-dimensional dataset.
    ell_radius_x = np.sqrt(1 + pearson)
    ell_radius_y = np.sqrt(1 - pearson)
    ellipse = Ellipse((0, 0), width=ell_radius_x * 2, height=ell_radius_y
    * 2,
                      facecolor=facecolor, **kwargs)

    # Calculating the standard deviation of x from
    # the squareroot of the variance and multiplying
    # with the given number of standard deviations.
    scale_x = np.sqrt(cov[0, 0]) * n_std
    mean_x = np.mean(x)

    # calculating the standard deviation of y ...
    scale_y = np.sqrt(cov[1, 1]) * n_std
    mean_y = np.mean(y)

    transf = transforms.Affine2D() \
        .rotate_deg(45) \
        .scale(scale_x, scale_y) \
        .translate(mean_x, mean_y)

    ellipse.set_transform(transf + ax.transData)
```

```
31
32      # Color the points
33      c = []
34      for i in range(len(x)):
35          if (x[i] - mean_x) ** 2 / scale_x ** 2 + (y[i] - mean_y) ** 2 /
    scale_y ** 2 <= 0.25:
36              c.append('goldenrod')
37
38          elif (x[i] - mean_x) ** 2 / scale_x ** 2 + (y[i] - mean_y) ** 2 /
    scale_y ** 2 <= 0.5:
39              c.append('darkgoldenrod')
40
41          elif (x[i] - mean_x) ** 2 / scale_x ** 2 + (y[i] - mean_y) ** 2 /
    scale_y ** 2 <= 1:
42              c.append('olive')
43
44          elif (x[i] - mean_x) ** 2 / scale_x ** 2 + (y[i] - mean_y) ** 2 /
    scale_y ** 2 <= 1.7:
45              c.append('yellowgreen')
46
47          else:
48              c.append('greenyellow')
49
50
51      ax.scatter(x, np.exp(y), color=c, s=8, edgecolor='black', linewidths
    =0.2)
52
53      return ellipse
```

Listing B.66: Función para graficar elipses de confianza y colorear puntos en función de estas.

```
1  import matplotlib.pyplot as plt
2  import seaborn as sns
3  import numpy as np
4
5  y = y1
6
7  # Crear el layout de la figura
8  fig = plt.figure(figsize=(5, 5))
9  grid = plt.GridSpec(4, 4, hspace=0.4, wspace=0.4)
10
11 # Scatter plot
12 main_ax = fig.add_subplot(grid[1:, :-1])
13 plot_ellipse_and_color_points(x, y2log, main_ax ,n_std, edgecolor='red')
14 main_ax.scatter(En.min(), y1[np.where(En == En.min())], color='red', s=4)
15 main_ax.plot(x, np.exp(intercept1 + slope1*x), 'k', label='$\\beta =$'+str
       (round(slope1, 4)))
16 main_ax.set_yscale('log')
17
18 # Histograma en el eje x
19 x_hist = fig.add_subplot(grid[0, :-1], sharex=main_ax)
20 x_hist.hist(x, bins=40, orientation='vertical', color='olivedrab')
21
22 # Histograma en el eje y
```

```
y_hist = fig.add_subplot(grid[1:, -1], sharey=main_ax)
y_bins = np.logspace(np.log10(np.min(y[y > 0])), np.log10(np.max(y)), 50)
y_hist.hist(y, bins=y_bins, orientation='horizontal', color='olivedrab')

# Etiquetas y titulos
main_ax.set_xlabel('Energias')
main_ax.set_ylabel('Probabilidades')
plt.figtext(0.19, 0.21, '1 layer QAOA', fontsize=8, bbox=dict(facecolor='
    white', edgecolor='black', boxstyle='round,pad=1'))

main_ax.legend(loc='lower left')

plt.show()
```

Listing B.67: Generación de graficos con histogramas adyacentes en ambas direcciones axiales.

```
# Ordenar x y cada uno de los vectores y
sorted_indices = np.argsort(x)
x_sorted = x[sorted_indices]
y1_sorted = y1[sorted_indices]
y2_sorted = y2[sorted_indices]
y3_sorted = y3[sorted_indices]
y4_sorted = y4[sorted_indices]

import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

plt.style.use('ggplot')

# Crear la figura y el grafico
plt.figure(figsize=(5, 3))

# Trazar lineas uniendo los puntos de cada conjunto de datos
plt.plot(x_sorted, np.cumsum(y1_sorted), color='firebrick', alpha=0.8,
    label='1 layer QAOA', linewidth=2)
plt.plot(x_sorted, np.cumsum(y2_sorted), color='teal', alpha=0.8, label='2
     layer QAOA', linewidth=2)
plt.plot(x_sorted, np.cumsum(y3_sorted), color='olive', alpha=0.8, label='
    3 layer QAOA', linewidth=2)
plt.plot(x_sorted, np.cumsum(y4_sorted), color='goldenrod', alpha=0.8,
    label='4 layer QAOA', linewidth=2)

# Configuracion del grafico
plt.xlabel('Energia', fontsize=14)
plt.ylabel('Probabilidad acumulada', fontsize=14)
plt.legend(fontsize=9)
plt.grid(True, linestyle='--', linewidth=2, alpha=1)

# Ajustes de los ejes
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)

# Mostrar el grafico
```

```
plt.tight_layout()
plt.show()
```

Listing B.68: Representación acumulativa de las probabilidades en función de la energía.

```
# Datos de ejemplo
y1 = probabilidades_1L_fin
y2 = probabilidades_2L_fin
y3 = probabilidades_3L_fin
y4 = probabilidades_4L_fin

# Crear una figura y ejes con 4 subplots
fig, axs = plt.subplots(1, 4, figsize=(12, 4), sharey=True)

# Histograma para y1
axs[0].hist(y1, bins=20, color='skyblue', alpha=0.7, density=True)
axs[0].set_title('Histograma y1')

# Histograma para y2
axs[1].hist(y2, bins=20, color='salmon', alpha=0.7, density=True)
axs[1].set_title('Histograma y2')

# Histograma para y3
axs[2].hist(y3, bins=20, color='lightgreen', alpha=0.7, density=True)
axs[2].set_title('Histograma y3')

# Histograma para y4
axs[3].hist(y4, bins=20, color='orange', alpha=0.7, density=True)
axs[3].set_title('Histograma y4')

# Ajustar el espacio entre subplots
plt.tight_layout()

# Mostrar el grafico
plt.show()
```

Listing B.69: Creación de un histograma para cada conjunto de datos asociado a la probabilidad del estado de mínima energía para diferente numero de capas.

```
# Calcular densidades de kernel
kde_y1 = gaussian_kde(y1)
kde_y2 = gaussian_kde(y2)
kde_y3 = gaussian_kde(y3)
kde_y4 = gaussian_kde(y4)

# Valores x para la grafica
x = np.linspace(-0.025, 0.25, 1000)

# Trazar las funciones de densidad empirica
plt.plot(x, kde_y1(x), label='y1', color='skyblue')
plt.plot(x, kde_y2(x), label='y2', color='salmon')
plt.plot(x, kde_y3(x), label='y3', color='lightgreen')
plt.plot(x, kde_y4(x), label='y4', color='orange')

```

```
# Configuracion de la grafica
plt.xlabel('Valor')
plt.ylabel('Densidad')
plt.xlim([-0.025, 1])
plt.title('Funciones de Densidad Empirica')
plt.legend()

# Mostrar la grafica
plt.show()
```

Listing B.70: Cálculo de las densidades de kernel y trazado de las funciones de densidad empírica para cada conjunto de datos previos.