Universidad Europea VALENCIA

GRADO EN FÍSICA

Estudio de la eficiencia y comportamiento de QAOA frente a algoritmos clásicos en problemas de optimización.

Presentado por:

Alberto Martínez Gallardo

Dirigido por:

Ezequiel Valero Lafuente

Curso académico 2024-2025

This Page Intentionally Left Blank

Agradecimientos

A mi tutor Ezequiel Valero por guiarme y aconsejarme a lo largo de este trabajo. También a todos los profesores que durante la carrera me enseñaron tanto académicamente como personalmente.

A mis compañeros de clase. A los que hoy siguen e incluso a los que se fueron. Muchas gracias por hacer que estos 4 años se pasaran como si nada. No podría haber deseado mejores compañeros.

A mis padres, por confiar en mi y por darme todas las oportunidades. Muchas gracias por ayudarme todos estos años y por creer siempre en mi. No estaría aquí de no ser por vosotros.

A Javier López, mi compañero de piso, de clase y un gran amigo. Gracias por aguantarme 4 años y por arreglarme el gradient descent cuando me estaba volviendo loco.

A Claudia Carrillo, por animarme y nunca dejar de confiar en mi. Por estos años y lo que nos queda.

Muchas gracias.

Abstract

Este trabajo se centra en el estudio de algoritmos de computación cuántica, con una introducción general al marco teórico de la computación cuántica, su naturaleza, estado actual y potencial en la resolución de problemas complejos. En particular, se aborda el uso de algoritmos cuánticos en tareas de optimización, un área de gran relevancia.

Se analizarán problemas de optimización de funciones, enfocándonos en la optimización de diferentes tipos de potenciales. Para ello, se implementará el algoritmo QAOA *(Quantum Approximate Optimization Algorithm)*, uno de los algoritmos cuánticos más representativos en este ámbito. Con el objetivo de contextualizar su rendimiento, se compararán sus resultados con los obtenidos mediante algoritmos clásicos de optimización como el método del descenso del gradiente y el método de Newton.

El estudio incluirá tanto una comparación cuantitativa en términos de precisión, tiempo de ejecución y calidad de la solución, como un análisis cualitativo de las diferencias entre enfoques clásicos y cuánticos. Asimismo, se examinará la influencia de los hiperparámetros de cada algoritmo sobre su rendimiento, con el propósito de identificar patrones o relaciones que permitan optimizar su configuración para distintos tipos de problemas.

Palabras clave: QAOA, computación cuántica, algoritmos de optimización

This work focuses on the study of quantum computing algorithms, beginning with a general introduction to the theoretical foundations of quantum computing, its nature, current state, and potential for solving complex problems. Specifically, we address the use of quantum algorithms in function optimization, a key area.

We analyze optimization problems involving various types of potential functions. To tackle these problems, we implement the Quantum Approximate Optimization Algorithm (QAOA), one of the most widely used quantum optimization algorithms. To contextualize its performance, we compare it with classical optimization algorithms such as the Gradient Descent, Newton's Method.

The study includes a quantitative comparison in terms of accuracy, execution time, and solution quality, as well as a qualitative analysis of the differences between classical and quantum approaches. Additionally, we examine the influence of algorithm hyperparameters on performance, aiming to identify patterns or relationships that help optimize algorithm configurations for different types of problems.

Key words: QAOA, quantum computing, optimization algorithms

Índice

1. Introducción

En 1965, Gordon E. Moore, cofundador de Intel, formuló una observación empírica según la cual el número de transistores en un microprocesador se duplicaría aproximadamente cada dos años. Este principio, conocido como la Ley de Moore, ha guiado la evolución de la informática durante décadas, impulsando avances exponenciales en la capacidad de cómputo. Sin embargo, en la última década, el ritmo de crecimiento de la potencia de los procesadores ha comenzado a ralentizarse, lo que ha llevado a cuestionarse la vigencia de esta ley en la actualidad. Este estancamiento ha generado un creciente interés en la computación cuántica, una disciplina emergente que podría representar una alternativa para mantener el ritmo de progreso en el ámbito computacional.

Dentro del campo de la computación cuántica, uno de los enfoques más prometedores es la optimización mediante algoritmos cuánticos. Desde su concepción, esta tecnología ha despertado un gran interés debido a su potencial para abordar problemas que son intratables para los ordenadores clásicos, aprovechando fenómenos propios de la mecánica cuántica, como la superposición y el entrelazamiento cuántico.

El concepto de computación cuántica fue introducido por primera vez en 1982 por el físico Richard Feynman en su trabajo *Simulating Physics with Computers* (Feynman, 1982). En este artículo, Feynman argumentaba que para simular eficientemente sistemas cuánticos, sería necesario utilizar ordenadores cuánticos, ya que los ordenadores clásicos no pueden manejar de manera eficiente la complejidad inherente a estos sistemas.

La principal ventaja de la computación cuántica radica en el uso de qubits en lugar de los bits tradicionales. En los ordenadores clásicos, la unidad mínima de información es el bit, que puede representar un valor de 0 o 1. En cambio, los qubits pueden existir en una superposición de ambos estados, lo que significa que pueden representar simultáneamente múltiples combinaciones de 0 y 1 en diferentes proporciones. Esta capacidad permite el desarrollo de algoritmos significativamente más eficientes para ciertos problemas específicos. Además, los qubits pueden formar estados de entrelazamiento, lo que permite correlaciones cuánticas que no tienen equivalentes en la computación clásica. Estas propiedades se pueden visualizar en la esfera de Bloch, una representación matemática que ilustra el estado de un qubit en términos de coordenadas tridimensionales.

Este trabajo se centrará en el estudio de los algoritmos de optimización cuántica, analizando sus principios fundamentales y su aplicabilidad en diferentes dominios. A lo largo del documento, se explorarán las ventajas y desafíos de esta tecnología emergente, así como su impacto potencial en la computación moderna.

2. Objetivos

El objetivo de este trabajo es ilustrar el estado y potencial de la computación cuántica, así como dar a conocer sus posibles aplicaciones. A lo largo del trabajo se presentan los conceptos fundamentales de la computación cuántica, junto con una descripción de los problemas que serán abordados mediante algoritmos tanto clásicos como cuánticos.

Una vez introducidos los fundamentos teóricos necesarios, se procederá a comparar distintos algoritmos de optimización. Por el lado clásico, se estudiarán métodos ampliamente utilizados como el método de Newton y el descenso del gradiente (*gradient descent*), empleados comúnmente para la búsqueda de mínimos de funciones. En cuanto al enfoque cuántico, se implementará el algoritmo QAOA (*Quantum Approximate Optimization Algorithm*), uno de los algoritmos más representativos en este contexto. Los algoritmos serán enfrentados a diferentes potenciales y sobre estos contextos se compararán en tiempo de cómputo, precisión de los resultados, consistencia en sus medidas así como sus relaciones entre los diferentes parámetros de cada uno y los resultados.

Durante el desarrollo del trabajo se detallará el funcionamiento de cada algoritmo, tanto desde el punto de vista teórico como computacional. Este enfoque tiene como finalidad facilitar la comprensión profunda de las herramientas utilizadas, así como permitir futuras modificaciones o ampliaciones del trabajo.

Durante el trabajo, se detallarán los pasos de cada algoritmo, tanto teóricamente como computacionalmente. Esto, con el fin de poder tener un mayor control y conocimiento de los algoritmos que son utilizados. Además, de hacer más sencillo su lectura y posibles modificaciones en el futuro.

Los algoritmos serán evaluados mediante su aplicación a potenciales representativos del campo de la física y la química, con el objetivo de enfrentar cada técnica a problemas lo más cercanos posible a casos reales. Esto permitirá valorar de forma más precisa la aplicabilidad de los enfoques cuánticos frente a los clásicos en estos contextos.

3. Marco teórico

3.1. Fundamentos de mecánica cuántica

A lo largo de la historia, la física ha logrado describir con gran precisión una enorme variedad de fenómenos naturales, desde el movimiento de los cuerpos celestes hasta la dinámica de fluidos o el comportamiento de sistemas electromagnéticos. Sin embargo, al adentrarse en las escalas más pequeñas del universo —correspondientes a átomos, electrones o fotones—, emerge una realidad profundamente distinta, la del mundo cuántico. En contraste con otras ramas de la física clásica, en las que es posible determinar de manera simultánea y con certeza

la posición y la velocidad de una partícula, en mecánica cuántica dicha certeza se ve limitada por principios fundamentales, tales como el principio de incertidumbre y conceptos como la superposición, el entrelazamiento o la cuantización.

Esta rama de la física, desarrollada a comienzos del siglo XX como respuesta a fenómenos que no podían ser explicados por la teoría clásica (como el efecto fotoeléctrico, la estabilidad de los átomos o el espectro del cuerpo negro), se distingue por una aproximación probabilística y ondulatoria a la descripción de la realidad. Es en ese momento en el que surgen diversas interpretaciones sobre la naturaleza de la mecánica cuántica. De entre todas ellas se va a considerar la interpretación de Copenhage, por ser la más extendida y la más desarrollada. En este contexto, las partículas no son tratadas simplemente como cuerpos con trayectorias definidas, sino como entidades cuya naturaleza está descrita por una función de onda, ψ, que contiene toda la información accesible sobre el sistema físico y cuya interpretación está vinculada a la probabilidad de encontrar una partícula en cierto lugar o con ciertas propiedades.

No obstante, la mecánica cuántica comparte una serie de postulados o principios fundamentales que, si bien pueden variar en número y formulación según la fuente consultada, coinciden en esencia con lo expuesto por Dirac (Dirac, 1928) en *The Fundamental Equations of Quantum Mechanics*. Estos pueden escribirse como (Carcassi et al., 2021), (Griffiths y Schroeter, 2018), ("Principios o postulados de la mecánica cuántica", s.f.):

Postulado I

El estado físico de un sistema cuántico está completamente descrito por un vector (*ket*) $|\psi\rangle$ perteneciente a un espacio de Hilbert. Este vector contiene toda la información accesible sobre el sistema. Donde $\psi(x)$ es una función de onda.

Postulado II

A cada magnitud física observable se le asocia un operador lineal Hermítico $\hat{A}$ que actúa sobre los elementos del espacio de Hilbert. Cuando se realiza una medición de un observable $\hat{A}$, el único resultado posible es uno de sus autovalores a, y el sistema colapsa al autoestado correspondiente $|a\rangle$.

Postulado III

Si el sistema se encuentra en un estado definido por una función de onda, $|\psi\rangle$, que no es autofunción de un operador, $\hat{A}$, asociado a un observable, a, una medida del observable *a* dará como resultado un autovalor de $\hat{A}$, pero no se puede predecir cuál de todos los posibles será. En este postulado se deja ver la naturaleza no determinista de la cuántica, como se verá más adelante, el modelo de Born describe la probabilidad de obtener cada uno de los estados cuánticos.

Postulado IV

En ausencia de medición, la evolución temporal del estado cuántico está gobernada por

la ecuación de Schrödinger dependiente del tiempo:

$$i\hbar\frac{d}{dt}\left|\psi(t)\right\rangle = \hat{H}\left|\psi(t)\right\rangle \tag{1}$$

Donde $\hat{H}$ es el operador Hamiltoniano del sistema.

Es por estos motivos que la mecánica cuántica es una rama tan peculiar y que aborda problemas de una manera particular. Por ejemplo, supongamos una partícula de masa m que se puede desplazar en el eje x, si queremos determinar su posición y velocidad no tendríamos problema en conocer ciertas magnitudes como su velocidad $v = \frac{dx}{dt}$, momento $p = mv$, energía cinética $E = \frac{1}{2}mv^2$ o alguna otra variable del movimiento. Y para conocer su posición haríamos uso de la ecuación de Newton. Si nos encontramos en mecánica cuántica y quisiéramos resolver esta misma situación, no buscaríamos las mismas magnitudes que antes, sino que buscaríamos la función de onda de la partícula, que viene regida por la que es probablemente la expresión más importante en la mecánica cuántica, la ecuación de Schrödinger (Griffiths y Schroeter, 2018).

$$i\hbar\frac{\partial\Psi}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\Psi}{\partial x^2} + V\Psi \tag{2}$$

A través de este tipo de situaciones, nos encontramos con otro principio fundamental de la mecánica cuántica. El principio de incertidumbre de Heisenberg. Este principio establece que no es posible conocer simultáneamente con precisión la posición y el momento lineal de una partícula. Es decir, cuanto mayor sea la precisión en la medición de una de estas magnitudes, mayor será la incertidumbre asociada a la otra. Esta relación está cuantificada por la desigualdad:

$$\Delta x \Delta p \geq \frac{\hbar}{2} \tag{3}$$

Donde ΔxyΔp es la incertidumbre en la medida de la posición y del momento respectivamente.

Se puede construir un estado tal que las mediciones de posición estén muy próximas entre sí (creando un "pico" localizado), pero a costa de aumentar la dispersión de las mediciones de momento en ese estado. O, por otro lado, se puede construir un estado con un momento definido (creando una onda sinusoidal larga), pero en ese caso las mediciones de posición estarán ampliamente dispersas. Naturalmente, al tratarse de una desigualdad, se puede construir el estado en el cual ni la posición ni el momento pueda definirse con precisión.

Esta limitación no es consecuencia de errores instrumentales, sino una manifestación directa de la naturaleza ondulatoria del sistema cuántico, la cual se describe a través de la función de onda. Ahora bien, ¿Qué es exactamente esta función de onda y qué información te da una

vez obtenida? Habitualmente, una partícula se encuentra localizada en un punto, sin embargo, una función de onda se encuentra distribuida en el espacio. Es a través de la interpretación estadística de Born que obtenemos la información de la función de onda. Esta interpretación define $|\Psi(x,t)|^2$ como la probabilidad de encontrar la partícula en la posición x en un momento t. O, de manera más precisa:

$$|\Psi(x,t)|^2\,dx = \begin{array}{c}\text{Probabilidad de encontrar la partícula entre } x \text{ y} \\ x+dx \text{ en un tiempo } t\end{array} \tag{4}$$

Esta interpretación de la función de onda de la partícula saca a la luz una de las mayores características de la mecánica cuántica. Es un modelo probabilístico. La función de onda nos informa de las posibilidades de encontrar una partícula, pero no da certeza sobre cómo se comporta. Esta naturaleza del comportamiento cuántico da lugar a una indeterminación en la mecánica cuántica. Incluso aún conociendo toda la información sobre la partícula (su función de onda), no podemos predecir con total seguridad el resultado que se obtendría al medir una partícula.

Esta imposibilidad de predecir con certeza el resultado de una medición concreta introduce uno de los fenómenos más característicos y contraintuitivos de la mecánica cuántica: la superposición de estados. A diferencia de los sistemas clásicos, donde un objeto se encuentra en un único estado definido en cada instante, en mecánica cuántica un sistema puede existir simultáneamente en una combinación lineal de varios estados posibles. Matemáticamente, si un sistema puede estar en los estados Ψ_1 y Ψ_2, entonces también puede encontrarse en el estado $\Psi = \alpha\Psi_1 + \beta\Psi_2$, donde α y β son coeficientes complejos que satisfacen la condición de normalización $|\alpha|^2 + |\beta|^2 = 1$.

$$|\Psi\rangle = c_0\,|0\rangle + c_1\,|1\rangle \tag{5}$$

Ejemplo de un estado en superposición

Esta propiedad implica que, antes de una medición, la partícula no posee un valor definido de la magnitud observada, sino que se encuentra en un estado superpuesto en el que coexisten múltiples posibilidades. Es precisamente el acto de medir el que "colapsa" la función de onda a uno de los posibles estados, seleccionando una de las opciones contenidas en la superposición.

Otro fenómeno característico de la naturaleza cuántica es el entrelazamiento. Se dice que dos partículas están entrelazadas (o que un estado esta entrelazado) cuando el estado cuántico del sistema total no puede expresarse como el producto de los estados individuales de cada partícula. Es decir, la función de onda conjunta no se puede factorizar como una combinación separada de funciones para cada una de ellas. No es hasta que no se realiza una medición que el estado no se determina (colapsa) y ambas partículas determinan su estado.

Este fenómeno trae consigo grandes consecuencias. En un estado entrelazado, una me-

dición sobre una de las partículas entrelazadas afecta instantáneamente el estado de la otra, sin importar la distancia que las separe. Este comportamiento va en contra del principio de localidad, lo que llevo a Einstein, Podolsky y Rosen a escribir un paper en 1935 (Einstein et al., 1935) en el que discutían este aspecto de la mecánica cuántica. Durante este documento se formuló un experimento mental que hoy se conoce como la paradoja EPR, aunque más que una paradoja fue una crítica a la completitud de la mecánica cuántica.

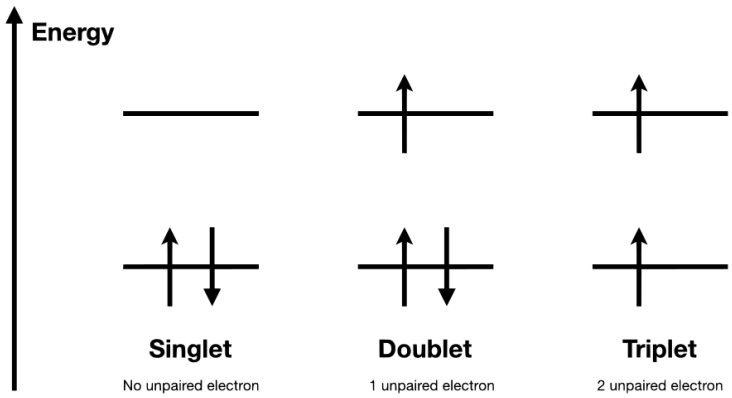


Figura 1: Diagrama de los estados singlete, doblete y triplete

No obstante, experimentos posteriores confirmaron empíricamente el entrelazamiento cuántico. Un ejemplo típico de estado entrelazado es el llamado estado singlete (*singlet state*), un estado el cual la suma total de su spin es 0. Se escribe como:

$$\frac{1}{2}(|\uparrow\downarrow\rangle - |\downarrow\uparrow\rangle) \tag{6}$$

La superposición y el entrelazamiento no solo representan los principios fundamentales que distinguen a la mecánica cuántica de la física clásica, sino que también constituyen la base teórica sobre la cual se sustentan los avances actuales en computación cuántica y otras tecnologías emergentes en este campo.

3.2. Fundamentos de la computación cuántica

Uno de los problemas fundamentales de la física cuántica, y por extensión, de la computación cuántica, es el problema de la medida. En los sistemas clásicos, medir no implica alterar el estado del sistema, en cambio, en cuántica, el acto de medir colapsa el estado cuántico, eliminando sus propiedades de superposición y entrelazamiento. Esto representa un obstáculo significativo para la computación cuántica ya que muchas de sus ventajas se sustentan precisamente en estos fenómenos no clásicos.

En un computador cuántico, la información se codifica en qubits, que pueden encontrarse en una superposición de los estados base $|0\rangle$ y $|1\rangle$. Como se ha visto antes, la expresión para una partícula cuántica viene definido por Ψ, y puede entenderse como la suma de dos estados con coeficientes α y β, por lo que se puede escribir un estado general del tipo:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{7}$$

Sin embargo, al realizar una medición, este estado se proyecta de manera aleatoria a uno de los estados base (es decir colapsa), $|0\rangle$ y $|1\rangle$, con probabilidades $|\alpha|^2$ y $|\beta|^2$, respectivamente. Como consecuencia, toda la información contenida en el entrelazamiento se pierde. Es por eso que los algoritmos cuánticos deben de construirse teniendo en cuenta este fenómeno. Conociendo esta limitación, los algoritmos cuánticos se diseñan de tal manera que la información deseada se obtenga a partir del resultado de las medidas y no del entrelazamiento. No obstante, esto presenta una gran desafío. Transformar los estados del sistema a través de puertas cuánticas en unos tales que, al realizar la medida, obtengamos con mayor probabilidad el estado que representa la solución correcta es una tarea muy compleja.

Existen varios algoritmos cuánticos que ilustran cómo este problema puede ser sorteado eficientemente. Un ejemplo destacado es el algoritmo de Grover, que permite buscar un elemento en una lista no ordenada con complejidad cuadráticamente menor que el mejor algoritmo clásico. En este caso, la amplitud del estado deseado se amplifica antes de la medición, aumentando la probabilidad de que el colapso conduzca al resultado correcto.

Al igual que en un circuito clásico, un circuito cuántico esta formado por cables y por puertas. A través de los cables se transporta la información del circuito y mediante las puertas lógicas se manipula esta información. Para un circuito cuántico, los cables son los diferentes qubits y la información contenida en ellos sería la información del estado. Y las puertas cuánticas serían las operaciones que se realizan sobre los qubits para construir el programa.

Un ejemplo de puerta clásica es la puerta *NOT* la cual invierte el valor del bit de $0 \rightarrow 1$ y $1 \rightarrow 0$. Si se quisiera realizar una transformación equivalente en un circuito cuántico, se necesitaría de una puerta cuántica que modificara los estados de esta manera. Es decir, se busca una puerta cuántica que pase del estado $|0\rangle \rightarrow |1\rangle$ y que $|1\rangle \rightarrow |0\rangle$. Más concretamente se quiere obtener:

$$\alpha\,|0\rangle + \beta\,|1\rangle \quad \rightarrow \quad \alpha\,|1\rangle + \beta\,|0\rangle \tag{8}$$

De esta manera se contempla el caso de la superposición y se elabora una puerta completa y

funcional. Así, la matriz que defina la puerta cuántica *NOT* se define como:

$$X \equiv \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{9}$$

Los estados $|0\rangle$ y $|1\rangle$ se escriben en notación vectorial.

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{10}$$

Entonces el estado en superposición queda escrito como:

$$\alpha\,|0\rangle + \beta\,|1\rangle = \alpha \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \beta \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \tag{11}$$

Por lo que, como queríamos, la puerta transforma adecuadamente.

$$X \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \beta \\ \alpha \end{pmatrix} \tag{12}$$

La puerta X también es conocida como la puerta *Pauli-X*, ya que coincide con la matriz de Pauli σ_x. De la misma manera, existen las puertas cuánticas correspondientes a las matrices de Pauli σ_y y σ_z, estas se conocen como las puertas *Y* y *Z*, y se pueden entender como rotaciones en cada uno de los ejes en una esfera de Bloch. La puerta *Y* transforma de $|0\rangle$ a $i\,|1\rangle$ y $|1\rangle$ a $-i\,|0\rangle$. La puerta Z solo transforma el estado $|1\rangle$ en $-\,|1\rangle$.

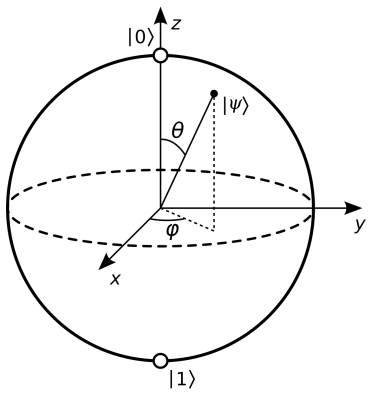


Figura 2: Esfera de Bloch

Estas puertas son ampliamente utilizadas, aunque se suelen modificar su fase para que

no realizan una rotación completa (π radianes). En su forma matricial, estas se escriben como:

$$Y \equiv \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \qquad Z \equiv \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{13}$$

Otra puerta cuántica muy importante es la puerta Hadamard. Esta puerta es la encargada de poner en superposición un qubit para poder aprovechar así su naturaleza cuántica. Los algoritmos cuánticos suelen iniciar con esta puerta para poder así aprovechar las características del entrelazamiento (Nielsen y Chuang, 2010).

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{14}$$

Esta puerta transforma los estados de la siguiente manera:

$$|0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \qquad |1\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} \tag{15}$$

Existen tambien puertas que actúan sobre más de un qubit como es el caso de la puerta *CNOT* (Controled Not), tambien conocida como CX.

$$CNOT \equiv \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{16}$$

Las puertas de control actúan sobre al menos dos qubits, sobre los cuales se distinguen dos tipos. Un qubit de control y uno o más como targets. Esta, tipo de puertas realizan o no la acción dependiendo de si el qubit de control tiene valor $|1\rangle$.

Las puertas de control son un concepto muy útil para las construcción de algoritmos cuánticos y pueden ser escritas también para otras puertas. De la misma manera que existe la Controlled Not, también existen la Controlled-X, Controlled-Y y Controlled-Z, donde el target realiza una rotación en el eje indicado.

A través de estas puertas lógicas cuánticas se construyen los circuitos cuánticos. Estos suelen representarse a través de líneas horizontales que indican el número de qubits, y sobre estas se plasman las puertas que actúan sobre cada uno de ellos. Al final de este diagrama se suele indicar que los qubits se miden para obtener la información deseada, y que esta queda guardada en canales clásicos.

Un ejemplo de este tipo de representación de circuitos podría darse para de algoritmo de Deutsch-Joza. Este algoritmo es muy usado para familiarizarse con el tipo de estructura de

los circuitos cuánticos. El objetivo de este algoritmo es determinar si una función esta balanceada o no. Se considera que una función esta balanceada si esta función devuelve con la misma frecuencia dos valores, por ejemplo 0 y 1. Por el contrario, se considera que está desbalanceada, si solo devuelve uno de estos dos valores. Para todos los casos intermedios, se considera parcialmente desbalanceada.

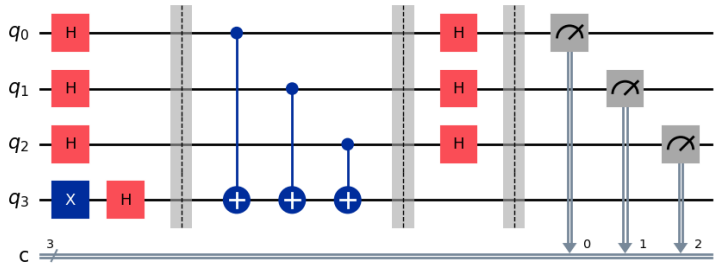


Figura 3: Circuito de Deutsch-Joza para 4 qubits para una función balanceada.

Como todos los algoritmos cuánticos, los qubits comienzan en el estado $|0\rangle$.Al iniciar se aplica una puerta *NOT* al último qubit para cambiar su valor a 1 y después se aplican puertas Hadamard sobre todos ellos. Esto con el objetivo de poner todos los estados en superposición. Tras esto, se aplican una serie de puertas *CNOT* sobre los qubits. Por último, se deshace la superposición volviendo a introducir puertas Hadamard y se mide.

3.3. Fundamentos de los algoritmos

El principal algoritmo de este trbajo, el QAOA (Quantum Aproximate Optimization Algorithm), es un algoritmo híbrido cuántico-clásico que resuelve problemas QUBO.

QUBO (Quadratic Unconstrained Binary Optimization) son problemas que cumplen las características nombradas. Son problemas de optimización, por lo que buscan un mínino o máximo. Son problemas binarios, donde las variables tienen dos posibles valores, son problemas sin restricciones y la función objetivo tiene una forma cuadrática. (Alonso-Linaje, 2024)

El algoritmo tiene dos operadores principalmente, el operador $U(C, \gamma)$,que es equivalente a la exponencial $e^{i\gamma H}$, que se encarga de cambiar la fase de los valores del Hamiltoniano, esta amplitud dependerá del γ. No puede ser muy pequeño porque no se separaría lo suficiente, ni muy grande, a riesgo de que dé la vuelta. Esto no altera la probabilidad de los estados (Farhi et al., 2014)

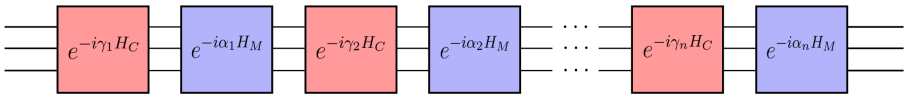


Figura 4: Esquema del circuito del QAOA

El segundo operador, $U(B, \beta)$, va a transformar la fase en amplitud a través de unas rotaciones. De manera similar al γ, en este caso el parámetro β va a medir la rotación de la fase. Estos operadores se ejecutan repetidamente para encontrar la solución. El número de repeticiones de estas puertas hasta que se mide es el parámetro profundidad o p.

En cuanto a los algoritmos clásicos se van a estudiar dos, el descenso del gradiente (*gradient descent*) y el metodo de Newton. El Gradient Descent es un algoritmo de optimización capaz de encontrar los mínimos de una función. Es ampliamente utilizado, reconocido por su eficacia y simplicidad. Imagina que estas en lo alto de una montaña con mucha niebla, tanto que solo puedes ver un poco alrededor de tus pies. Una buena manera de bajar la montaña seria caminar hacia donde vieras que el suelo desciende más. Así, si siguieras caminando hacia donde ves que el suelo desciende, acabarías bajando la montaña. Este es el ejemplo más habitual que se suele dar para explicar la idea detrás de este algoritmo (Géron, 2019). Como vemos conceptualmente es muy sencillo, lo que hace que sea muy fácil de entender y de implementar.

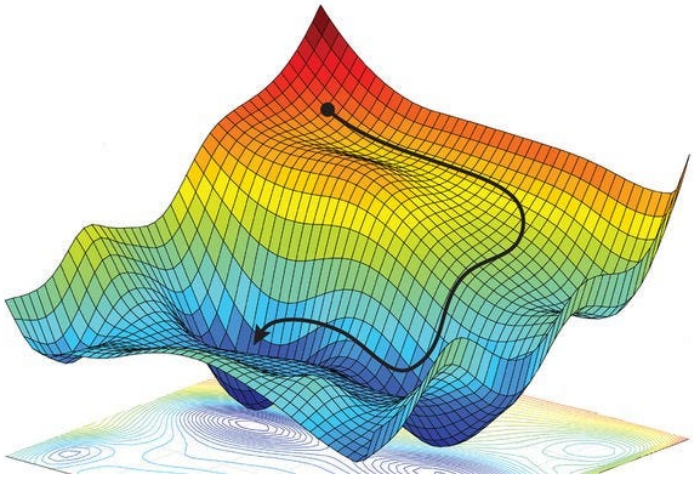


Figura 5: Recorrido de un punto a través del descenso del gradiente

De manera más matemática, el gradiente de una función escalar nos proporciona un vector que indica la dirección en la que la función f crece más rápidamente, es decir, donde la tasa de cambio de f es máxima. De esta forma, podemos entender el comportamiento de la función en torno a un punto dado.

Con esto en mente, el Gradient Descent comienza seleccionando un punto aleatorio dentro del dominio de la función. A partir de este punto, se calcula el gradiente, y luego se realiza un desplazamiento en la dirección opuesta a la indicada por el gradiente. De esta manera, evitamos los máximos locales y nos aproximamos a los mínimos de la función.

De este concepto se extraen otras muchas variaciones del Descenso del Gradiente. Aunque, en general, todas tengan como base la misma idea. El más común y básico, el Batch Gradient Descent, que es el que se ha explicado anteriormente. Aunque también existen otros como el *Stochastic Gradient Descent* o el Mini-Batch Gradient Descent que pueden ser mas completos o específicos. Sin embargo, para este trabajo se usará unicamente el Batch Gradient Descent, ya que el interés real es la comparativa con un algoritmo cuántico, no es conveniente profundizar más de lo necesario en los algoritmos clásicos.

El algoritmo de Newton puede parecer muy similar al gradient descent, sin embargo, las diferencias entre estos son más que sutilezas. Tanto el mínimo como el máximo de una función son puntos que cumplen que $f'(x) = 0$, por lo que normalmente se suele emplear esta primera derivada para poder encontrar el punto mínimo, como se hacia en el Gradient Descent.

Sin embargo, la segunda derivada también aporta mucha información sobre la forma de una función. Si la primera derivada habla de cuánto y en qué dirección cambia una función (es decir la pendiente), la segunda derivada dice cuánto está cambiando esta pendiente.

Si $f''(x) > 0$ es que la pendiente está creciendo, si $f''(x) < 0$ la pendiente decrece. Haciendo uso de este concepto, el algoritmo de Newton es capaz de encontrar el mínimo de una función.

4. Metodología

En este trabajo se estudia el algoritmo QAOA, así como sus parámetros y relaciones con los resultados. Para poder comparar con otros algoritmos tradicionales se toman dos algoritmos de optimización clásicos, el método de Newton y el Gradient Descent. Para poder desarrollar los algoritmos planteados en este trabajo, se va a usar Python. De esta manera, se podrá modificar el código a voluntad para poder alterar parámetros, entender mejor el funcionamiento de cada uno de los algoritmos así como extraer los resultados. Todos estos algoritmos serán programados usando JupyterNotebook ya que permite no solo ejecutar celdas de código de manera independiente, sino que también permite añadir celdas de Markdown donde escribir anotaciones y explicaciones. Aunque para las versiones definitivas se usarán archivos de Python *.py* por comodidad para ejecutarlas.

Más concretamente para la elaboración del QAOA se usara la librería de Python *Qiskit* de IBM. Ya que al tratarse de un algoritmo cuántico requiere de una librería que pueda tratar

este problema como tal.

Qiskit es una librería de Python desarrollada por IBM, se trata de la librería de computación cuántica más extendida, cuenta con una amplia documentación y permite generar circuitos, usar funciones ya definidas de Qiskit así como poder ejecutar en circuito en un ordenador cuántico real o a través de un simulador. Otra opción a considerar para implementar el QAOA es la biblioteca PennyLane, una herramienta también muy utilizada en el campo de la computación cuántica. Sin embargo, PennyLane está más enfocada en el aprendizaje automático cuántico (QML), y suele presentar más dificultades al momento de ejecutarse en un ordenador cuántico real, como se tiene previsto en este trabajo. De todos modos, esta posibilidad dependerá de si finalmente se cuenta con los recursos necesarios para llevarla a cabo.

Durante el desarrollo del trabajo, se hará uso de los simuladores que ofrece Qiskit para ejecutar el codigo, ya que hacerlo constantemente en un ordenador cuántico real seria muy costoso. Si fuera posible se ejecutara en un ordenador cuántico real en ocasiones concretas, donde quedara indicado y se tendrá en cuenta que el algoritmo esta siendo ejecutado en un QPU.

A pesar de que Qiskit cuenta ya con funciones que permiten implementar el QAOA de manera directa en tu circuito, en este trabajo se ha optado por realizar los pasos del QAOA de manera explícita. La finalidad de esta decisión es poder modificarlo con mayor libertad y más específicamente. Asimismo, formular el QAOA de esta manera proporciona mucho más control y entendimiento sobre el funcionamiento de este.

4.1. QAOA

El QAOA cuenta con dos partes principales. El primer operador se encarga de modificar las fases de los estados, sin modificar las probabilidades. Mientras que el segundo operador se encarga de modificar las probabilidades según su fase, dejando así estados con diferentes probabilidades. Es la iteración de estas puertas, así como el ajuste de los parámetros para cada uno de los operadores de estas puertas los que permiten obtener una solución correcta (Alonso-Linaje, 2024) (IBM, 2025).

Sin embargo, para poder ejecutar el algoritmo se necesita una función de coste, en este caso esa función sera el Hamiltoniano. Esta función se construye al expresar la función que se pretende minimizar de tal manera que podamos tratarla como un problema del tipo QUBO (Quadratic Unconstrained Binary Optization). Estos problemas cumplen las siguientes características: son problemas de optimización, por lo que buscan un mínino o máximo; son problemas binarios, donde las variables tienen dos posibles valores; son problemas sin restricciones y la función objetivo tiene una forma cuadrática. Y es que, para que una función pueda ser expresada y entendida en un circuito cuántico es necesario realizar ciertas transformaciones.

En primer lugar, se debe de discretizar el dominio de la función. No hay que olvidar que

al tratarse de un algoritmo cuántico, al final se medirá un estado como el estado más probable, y este será el que se considerará como solución. Por ello, es necesario asignar un valor a cada uno de los estados con los que se trabaja.

Si se supone una función $f(x)$ y n qubits, entonces, la función podrá que ser discretizada siguiendo:

Estado	x	$f(x)$
$\|0\rangle^{\otimes n}$	x_0	$f(x_0)$
$\|1\rangle^{\otimes n}$	x_1	$f(x_1)$
$\|2\rangle^{\otimes n}$	x_2	$f(x_2)$
$\|3\rangle^{\otimes n}$	x_3	$f(x_3)$
$\dots$	$\dots$	$\dots$
$\|2^n\rangle^{\otimes n}$	x_n	$f(x_{2^n})$

Donde la notación $|a\rangle^{\otimes n}$ expresa el número a para n qubits y x_a refiere a los valores elegido para la discretización.

Una vez se han elegido los valores a los que corresponden cada estado del qubit, se procede a construir el Hamiltoniano. Para poder construir el Hamiltoniano, es necesario calcular las puertas de Pauli para cada combinación de qubit y para cada estado posible.

Las matrices de Pauli se definen de la siguiente manera.

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \quad \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{17}$$

Y los estados de los qubits en forma matricial se escriben como una matriz columna.

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{18}$$

También puede escribirse la matriz de Pauli Z siguiendo la notación bracket

$$Z = |0\rangle\langle 0| - |1\rangle\langle 1| \tag{19}$$

Como durante este trabajo solo se va a usar la puerta z de Pauli, se toma la siguiente notación.

$$Z_\alpha = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} |\alpha\rangle \tag{20}$$

Donde $|\alpha\rangle$ corresponde al valor en la posición α de cada estado leído de derecha a izquierda.

El último paso para resolver obtener el Hamiltoniano es resolver el sistema de ecuaciones formado por las matrices de Pauli. Se conoce la forma de la función de coste Hamiltoniano,

Estado	Z_0	Z_1	Z_2	Z_0Z_1	Z_0Z_2	Z_1Z_2	$Z_0Z_1Z_2$
$\|000\rangle$	1	1	1	1	1	1	1
$\|001\rangle$	-1	1	1	-1	-1	1	-1
$\|010\rangle$	1	-1	1	-1	1	-1	-1
$\|011\rangle$	-1	-1	1	1	-1	-1	1
$\|100\rangle$	1	1	-1	1	-1	-1	-1
$\|101\rangle$	-1	1	-1	-1	1	-1	1
$\|110\rangle$	1	-1	-1	-1	-1	1	1
$\|111\rangle$	-1	-1	-1	1	1	1	-1

Cuadro 1: Ejemplo matrices de Pauli para 3 qubits

está formado por la suma de todas las combinaciones de matrices de Pauli multiplicadas por un factor.

$$H_c = a_0 I + a_1 Z_0 + a_2 Z_1 + \cdots + a_7 Z_0 Z_1 Z_2 \tag{21}$$

Ejemplo de Hamiltoniano para 3 qubits

Con esta expresión y las matrices de Pauli calculadas se resuelve el sistema de ecuaciones, que, para seguir con el ejemplo de 3 qubits, tendría la siguiente forma:

$$\begin{cases} |000\rangle \rightarrow & \begin{cases} a_0 I + a_1 Z_0 + a_2 Z_1 + \cdots + a_7 Z_0 Z_1 Z_2 & = f(x_0) \\ a_0 I + a_1 Z_0 + a_2 Z_1 + \cdots + a_7 Z_0 Z_1 Z_2 & = f(x_1) \\ a_0 I + a_1 Z_0 + a_2 Z_1 + \cdots + a_7 Z_0 Z_1 Z_2 & = f(x_2) \\ \cdots & = f(\dots) \\ a_0 I + a_1 Z_0 + a_2 Z_1 + \cdots + a_7 Z_0 Z_1 Z_2 & = f(x_7) \end{cases} \\ |001\rangle \rightarrow & \\ |010\rangle \rightarrow & \\ \dots \rightarrow & \\ |011\rangle \rightarrow & \end{cases} \tag{22}$$

Una vez resuelto este sistema, ya se obtendrían los coeficientes que formarán el Hamiltoniano. Estos coeficientes son utilizados dentro del QAOA.

El QAOA comienza poniendo en superposición todos los qubits a través de puertas Hadamard, este paso es bastante habitual en los algoritmos cuánticos. Tras esto, se aplican las puertas de rotación a los qubits. Esto se logra al aplicar una matriz de Pauli en Z con un factor de $2 \cdot \gamma \cdot a_n$ donde γ es un parámetro externo y a_n es el valor obtenido del Hamiltoniano. Tras esta operación la fase de los estados se ha modificado de manera adecuada. Sin embargo, el operador fase no modifica las probabilidades de los estados, es por eso que se necesita de un segundo operador. Este consisten en puertas de rotación en X, las cuales si que alteran la probabilidad. Estas rotaciones son aplicadas con un factor de $2 \cdot \beta$. Donde β es un parámetro externo.

4.2. Gradient Descent

El Gradient Descent inicia su algoritmo con un valor inicial aleatorio θ_0, luego, a través de ejecutar operaciones iterativas consigue obtener una solución. Cada término de la iteración viene dado por la siguiente expresión (Luenberger y Ye, 2015).

$$\theta_n = \theta_{n-1} - \mu \nabla f(\theta_{n-1}) \tag{23}$$

En esta expresión tenemos un valor inicial θ al cual se le resta el gradiente de la función evaluada para ese parámetro θ multiplicado por un parámetro μ. Esta ecuación devuelve una θ algo más cerca del mínimo de la función. Si se toma el nuevo θ como inicial y se repite la operación suficientes veces, se obtiene la solución correcta.

Es por esto que es muy importante en este algoritmo la elección del parámetro μ, que se conoce como tasa de aprendizaje (learning rate). Si la tasa de aprendizaje es demasiado grande, el algoritmo podría oscilar alrededor de la solución, mientras que si es demasiado pequeño el algoritmo podría tardar demasiado en encontrar la solución deseada. Es relevante comentar que algunas variaciones del algoritmo no consideran una tasa de aprendizaje constante, pudiendo hacer que disminuya con cada iteración para poder lograr mejores resultados, este tipo de implementaciones no se van a llevar a cabo en este trabajo.

La tolerancia (ϵ), aunque no es un parámetro observable en la ecuación anterior, también es un parámetro muy importante para la optimización del Gradient Descent. La definición de tolerancia en cualquier algoritmo es el criterio utilizado para detener el mismo. Debido a las limitaciones numéricas es muy difícil que el algoritmo logre alcanzar el mínimo absoluto de una función, es por eso que se establecen criterios de parada. Algunos muy comunes son:

$$|\theta_{n-1} - \theta_n| < \epsilon \tag{24}$$

$$|f(\theta_{n-1}) - f(\theta_n)| < \epsilon \tag{25}$$

Lo habitual es elegir valores de ϵ muy bajos, normalmente relacionados con la precisión que se espera obtener del resultado. Si se espera una precisión del orden de 10^{-3} un valor adecuado de la tolerancia sería 10^{-4} para asegurar la precisión de los decimales.

Esta técnica evita que el algoritmo realice más iteraciones de las necesarias y mejora su velocidad.

4.3. Algoritmo de Newton

El algoritmo de Newton se aprovecha de la información que ofrece la segunda derivada de una función para calcular el mínimo. Además, usa la expansión en serie de Taylor para poder

obtener la expresión que se necesita. La serie de Taylor de una función para segundo orden se puede escribir como

$$f(x) \approx f(x_k) + f'(x_k)(x - x_k) + \frac{1}{2}f''(x_k)(x - x_k)^2 \tag{26}$$

por lo que se puede considerar esta expresión como la nueva expresión de la función. El siguiente paso, es encontrar el valor de x tal que $f'(x) = 0$.

$$f'(x) = f'(x_k) + f''(x_k)(x - x_k) \tag{27}$$

$$0 = f'(x_k) + f''(x_k)(x - x_k) \tag{28}$$

$$0 = \frac{f'(x_k)}{f''(x_k)} + (x - x_k) \tag{29}$$

$$x = x_k - \frac{f'(x_k)}{f''(x_k)} \tag{30}$$

Esta es la ecuación del algoritmo de Newton. De manera similar a cómo ocurría con el gradient descent, esta función se inicia en un valor aleatorio de x_k y, a través, de iterar esta expresión, se calcula el mínimo de una función. Al tratarse de un algoritmo iterativo, también necesita de una tolerancia (ϵ) para definir un criterio de parada. Y así, de la misma manera que para el gradient descent, evitar que el algoritmo itere más de lo necesario.

4.4. Muestreo de Montecarlo

Debido a la naturaleza cuántica del QAOA, este puede no presentar la misma consistencia y reproducibilidad que otros algoritmos clásicos. Aunque el estado que corresponde a la solución más probable es el que tiene más probabilidad de manifestarse al medir (suponiendo que los parámetros estén bien ajustados). Esto no asegura que la medida sea la correcta. Además, tanto el QAOA como ambos algoritmos clásicos son capaces de localizar un mínimo, mientras que los potenciales a los que se enfrentan tienen en algunos casos dos mínimos.

Por estos motivos, se ha decidido incorporar métodos de Montecarlo a los algoritmos utilizados. Una simulación de Montecarlo es un tipo de algoritmo computacional que se basa en realizar múltiples muestreos aleatorios para estimar la probabilidad de distintos resultados. En este caso, se empleará este enfoque para ejecutar varias veces los algoritmos, comenzando cada ejecución con un valor aleatorio. Con este enfoque se logra reducir la incertidumbre que puede generarse en el QAOA por causas propias de la cuántica. Además, de poder localizar varios mínimos en una función si los hubiera.

4.5. Potenciales de interés físico

Los potenciales que se van a estudiar son el potencial de Lennard-Jones, el potencial de Higgs y el potencial de una molécula diatómica, como lo es el hidrógeno. Todos son potenciales de interés físico para los cuales sus mínimos representan un valor clave. En este trabajo no es de interés el valor mínimo real de estos potenciales sino que solo se pretende estudiar la capacidad de los algoritmos de encontrar los valores mínimos de unos potenciales. Es por ello que para plantear los potenciales se van a recurrir a aproximaciones, normalizaciones o algún método que modifique el potencial pero no su forma. Esto con el objetivo de poder ejecutar estos algoritmos dentro del rango deseado sin alterar la forma del potencial.

4.5.1. Potencial de Lennard-Jones

El potencial de Lennard-Jones modeliza la energía potencial intermolecular total. Cuando una molécula se comprime, las fuerzas de repulsión del núcleo y de los electrones, así como de la energía cinética de los electrones, se dispara y domina frente a las fuerzas de atracción. Estas repulsiones aumentan bruscamente conforme disminuye la distancia intermolecular. Es por esto que su cálculo con precisión es una tarea difícil y que involucra modelos y cálculos muy extensos y complicados de estructuras moleculares. Afortunadamente, el potencial de Lennard-Jones ofrece una aproximación numérica bastante fiel.

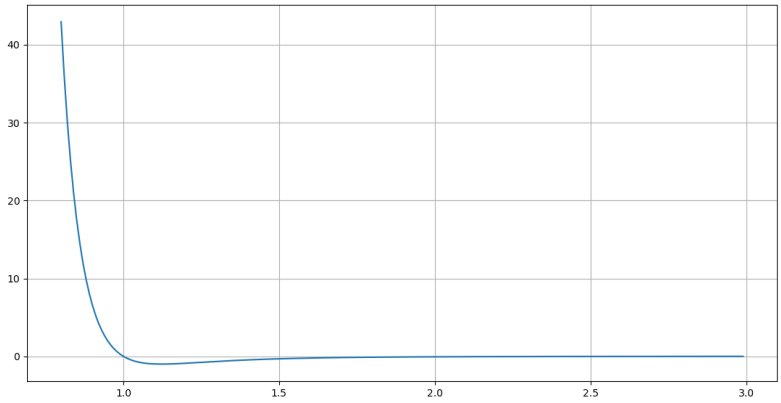


Figura 6: Potencial de Lennard Jones

El potencial cuenta con una parte referente a la fuerza de repulsión r^{-12}y otra a la fuerza de atracción r^{-6} (Atkins y de Paula, 2010).

$$V(r) = 4\epsilon \left[\left(\frac{r_0}{r} \right)^{12} - \left(\frac{r_0}{r} \right)^{6} \right] \tag{31}$$

El mínimo de este potencial, informa de la distancia r a la cual esta energía potencial itermolecular es menor. En este trabajo no es de interés el valor exacto del potencial para una molécula concreta sino que se encuentre el mínimo de la función. Se tomará una versión con los parámetros $\epsilon = r_0 = 1$.

$$V(r) = 4\left[\left(\frac{1}{r}\right)^{12} - \left(\frac{1}{r}\right)^{6}\right] \tag{32}$$

La expresión del potencial de Lennard-Jones que se ha tomado para calcular es, una versión con los parámetros simplificados y cuyo mínimo es $x_{\text{min}} = 2^{1/6} \approx 1{,}1225$

$$f(x) = 4\left[\left(\frac{1}{x}\right)^{12} - \left(\frac{1}{x}\right)^{6}\right] = 4\left(x^{-12} - x^{-6}\right) \tag{33}$$

$$\begin{aligned} f'(x) = 0 &= 4(-12x^{-13} + 6x^{-7}) \\ &= -48x^{-13} + 24x^{-7} \qquad (34) \\ &= -48 + 24x^{6} \end{aligned}$$

$$x^6 = \frac{48}{24} = 2 \tag{35}$$

$$x = \sqrt[6]{2} \approx 1{,}1225 \tag{36}$$

4.5.2. Potencial de Higgs

En el marco del Modelo Estándar de la física de partículas, el mecanismo de ruptura espontánea de simetría es esencial para explicar el origen de la masa de las partículas elementales. Este mecanismo se formaliza a través del llamado potencial de Higgs, una función que describe el comportamiento del campo de Higgs y cuya forma permite comprender cómo las partículas adquieren masa sin violar las leyes de simetría. El potencial de Higgs se expresa típicamente como:

$$V(\phi) = \mu^2|\phi|^2 + \lambda|\phi|^4 \tag{37}$$

La expresión usada para el potencial de Higgs se corresponde a $V(x) = -1x^2 + 0{,}5x^4$.

La cual tiene dos mínimos globales. $x_{\text{min}} = \pm 1$

$$f(x) = -x^2 + 0{,}5x^4 \tag{38}$$

$$\begin{aligned} f'(x) = 0 &= -2x + 2x^3 \\ &= -x + x^3 \\ &= -1 + x^2 \end{aligned} \tag{39}$$

$$x^2 = 1 \tag{40}$$

$$x = \pm 1 \tag{41}$$

Existe una raíz para $x = 0$, sin embargo, no se va a tomar al tratarse esta de un punto máximo.

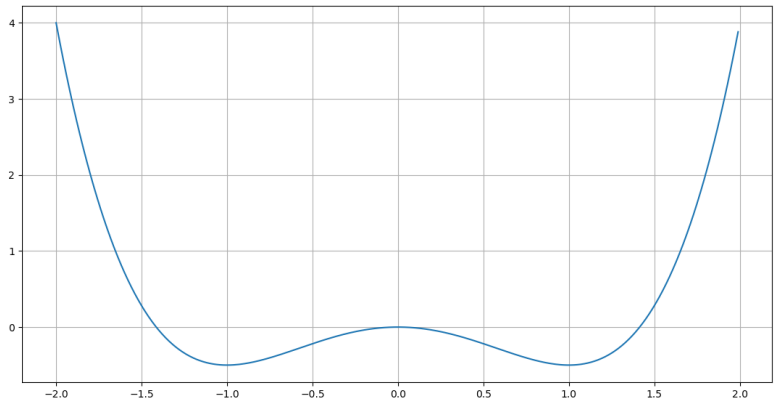


Figura 7: Potencial de Higgs

4.5.3. Potencial hidrógeno

El siguiente potencial que se ha tomado es el potencial de una molécula diatómica, como lo puede ser el hidrógeno. El potencial que se ha tomado no es un potencial fiel a los valores reales, sino que mantiene la forma de este, pero adaptado a un rango de valores más cómodos.

En este caso, para poder trabajar con una expresión que permita mantener el rango de valores y poder trabajar con comodidad, sin sacrificar la forma del potencial, se va a tomar la expresión del potencial:

$$f(x) = 3x^4 - 7x^2 - 0{,}5x + 6 \tag{42}$$

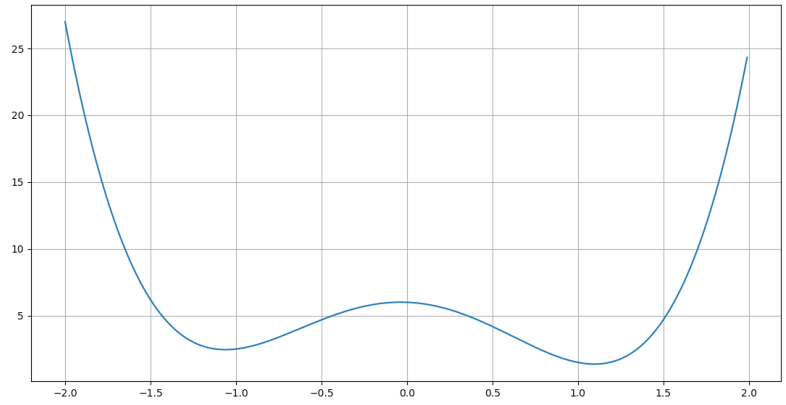


Figura 8: Potencial del Hidrógeno

Esta se trata de una función polinómica. La cual, al no tener resolución analítica, debe tratarse mediante métodos numéricos. La resolución de esta ecuación a través de software arroja los valores: $x_{\text{local min}} = -1{,}06$, $x_{\text{global min}} = 1{,}1$. Y $x_{\text{máximo}} = -0{,}04$.

$$f(x) = 3x^4 - 7x^2 - 0{,}5x + 6 \tag{43}$$

$$\begin{aligned} f'(x) = 0 &= 12x^3 - 14x - 0{,}5 \\ &= 24x^3 - 28x - 1 \end{aligned} \tag{44}$$

$$x = -1{,}06 \qquad x = -0{,}04 \qquad x = 1{,}1 \tag{45}$$

Evidentemente, de estas 3 raíces para el potencial, solo se tendrán en cuenta las 2 que corresponden a los mínimos.

5. Análisis de los resultados

Para la ejecución de los algoritmos en los diferentes potenciales, se ha generado una versión de los algoritmos con una interfaz desde la cual se pueden introducir los parámetros y ver los resultados obtenidos. Asimismo, se cuenta con otra versión la cual recibe combinaciones de parámetros a través de un diccionario previamente definido y guarda los datos en un csv. Estas dos opciones permiten, tanto obtener grandes cantidades de datos para diferentes parámetros, como poder observar los resultados obtenidos para un parámetro concreto.

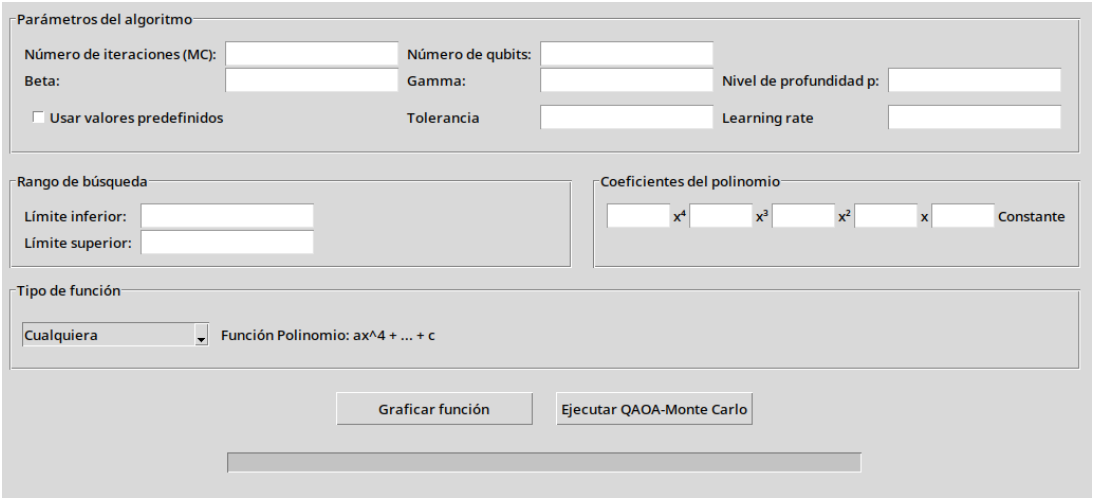


Figura 9: Interfaz de ejecución para el QAOA

Se han seleccionado diferentes parámetros para ejecutar los algoritmos con el objetivo de obtener los mejores resultados para cada potencial, así como poder estudiar su dependencia con cada parámetro. Cabe destacar, que para estudiar la reproducibilidad y consistencia de los algoritmos se han ejecutado 3 veces para cada potencial. Si los algoritmos se ejecutaran más de 3 veces los datos serían más precisos, sin embargo, debido a limitaciones de los recursos con los que se cuenta se van a ejecutar 3 veces para cada combinación. Los parámetros sobre los cuales se va a ejecutar el QAOA son:

- **Número de qubits:** 2, 3, 4
- **Parámetro beta (β):** 0.5, 0.3, 0.1
- **Parámetro gamma (γ):** 0.5, 0.25, 0.1, 0.05, 0.01
- **Profundidad de p:** 2, 3, 4
- **Número de iteraciones del MC:** 50, 100
- **Tolerancia:** 0.05
- **Learning rate:** 0.5

El número de qubits es el número de qubits con el que se va a ejecutar el algoritmo. Tanto el parámetro β como el γ modifican los valores de los dos operadores del QAOA. La profundidad de p, indica el número de veces que los dos operadores del QAOA se aplican antes de medir. El número de iteraciones de Montecarlo es el número de veces que se ejecuta el algoritmo, cada vez con un valor inicial aleatorio. La tolerancia y el learning rate son parámetros peculiares en este algoritmo ya que no tienen exactamente el mismo significado que en los

algoritmos clásicos. La tolerancia, como en la mayoría de algoritmos, es el valor mínimo que debe de mejorar el valor generado con respecto al anterior para que el algoritmo siga buscando la solución. Por otro lado, en este algoritmo el learning rate es la magnitud por la que se desplazan los valores asignados a cada qubit tras cada iteración. Es decir, en cada iteración del algoritmo los valores asignados a cada qubit se recalculan siguiendo una distancia, esta distancia está definida como learning rate. Aunque en este algoritmo este valor se ha definido de tal manera que disminuya con cada iteración, con el objetivo de tener un valor dinámico que pueda aproximarse mejor a la solución.

Con estos parámetros el algoritmo se ejecutará para 270 combinaciones de parámetros. Se puede ver que tanto la tolerancia como la tasa de aprendizaje (*learning rate*) son parámetros que no se varían. Esto es debido a que aumentar el número de valores posibles para los parámetros supone un añadido computacional enorme que no se puede permitir con los recursos que se disponen, una vez considerado eso, los parámetros que menos interesa estudiar son precisamente estos parámetros que no son únicos de este algoritmo cuántico. Además, su repercusión es relativamente predecible, una menor tolerancia dará lugar a un resultado más preciso a costa de más tiempo de cómputo, mientras que un menor learning rate resultara en una mejor convergencia a costa de más iteraciones del algoritmo.

Para el gradient descent se han podido tomar más variedad de valores para cada parámetro al contar este con menos parámetros. Por ello se han seleccionado estos parámetros, los cuales dan lugar a 1280 combinaciones de parámetros.

- **Número de iteraciones:** 100, 500, 1000, 2000
- **Distancia entre mínimos** 0.1, 0.2, 0.25, 0.5, 1
- **Número de iteraciones del MC:** 50, 100, 150, 200
- **Tolerancia:** 0.1, 0.05, 0.01, 0.0001
- **Learning rate:** 0.1, 0.05, 0.01, 0.0001

De la misma manera, se han seleccionado para el método de Newton los siguientes parámetros. Considerando estos parámetros se han tomado 320 combinaciones para este algoritmo.

- **Número de iteraciones:** 100, 500, 1000, 2000
- **Distancia entre mínimos** 0.1, 0.2, 0.25, 0.5, 1
- **Número de iteraciones del MC:** 50, 100, 150, 200
- **Tolerancia:** 0.1, 0.05, 0.01, 0.0001

Aunque pueda parecer que se realiza una búsqueda más intensiva para los parámetros de los algoritmos clásicos, lo cierto es que es para el QAOA para el que se invierten más recursos computacionales en obtener las combinaciones de los parámetros. La búsqueda de parámetros en estos dos últimos algoritmos no es tan exhaustiva como para el QAOA. En primer lugar, porque estos algoritmos cuentan con menor cantidad de parámetros que optimizar. Y en segundo lugar, porque no presentan el mismo interés de estudio que el QAOA, al menos, en este trabajo.

Para almacenar los datos que se generan con cada ejecución de los algoritmos se van a utilizar ficheros cvs. Así, se podrán tratar, modificar y comparar los datos de manera ordenada y visual, para poder obtener mejores resultados.

Para el estudio y comparación de los algoritmos se va a estudiar la precisión de las soluciones. Es decir, cuánta variación hay entre la solución obtenida a través del algoritmo y la solución real. El tiempo de cómputo, es decir, cuánto tiempo tarda en devolver una solución con los parámetros seleccionados. Y, por último, su consistencia, cuán reproducibles son los resultados. Estos datos se obtendrán ejecutando los algoritmos para los mismos parámetros y comparando la variación entre los resultados. En este punto no interesa si el resultado corresponde o no con el mínimo, simplemente si el resultado es consistente con las iteraciones. Esta características es de especial interés en el QAOA, ya que al tratarse de un algoritmo cuántico, tiene un factor no determinista que no es despreciable.

Además, se pretende estudiar, la dependencia y sensibilidad a los hiperparámetros para los diferentes algoritmos. Haciendo especial hincapié en el QAOA, por ser el algoritmo con más parámetros y del que presenta más interés en este trabajo.

5.1. Potencial de Lennard-Jones

A diferencia del resto de potenciales que se evalúan, este potencial cuenta solo con un mínimo en 1.12 por lo que se puede esperar que el tiempo de cómputo sea menor y que tenga más capacidad de convergencia. No obstante, al tener un mínimo muy sutil y localizado y una pendiente muy suave durante casi todo el potencial, puede ocurrir que algoritmos muy dependientes de la pendiente de una función tengan problemas para localizar el mínimo.

5.1.1. QAOA

Tras ejecutar el QAOA para los parámetros indicados, se obtiene un csv. El cual al leerse muestra una tabla con la siguiente forma.

N_qubits	beta	gamma	Prof_p	N_mc	Toler	Learning rate	Tiempo ejecucion	Primer mínimo	True min
2	0.5	0.50	2	50	0.05	0.5	14.1236	2.1	0.9775
2	0.5	0.50	2	100	0.05	0.5	26.5927	2.5	1.3775
2	0.5	0.50	3	50	0.05	0.5	13.0457	2.2	1.0775
2	0.5	0.50	3	100	0.05	0.5	27.2008	2.5	1.3775
2	0.5	0.50	4	50	0.05	0.5	13.9767	2.5	1.3775
...	...	...	...	...	...	...	...	...	...
4	0.1	0.01	2	100	0.05	0.5	156.4824	0.9	0.2225
4	0.1	0.01	3	50	0.05	0.5	118.7553	2.8	1.6775
4	0.1	0.01	3	100	0.05	0.5	234.1638	2.4	1.2775
4	0.1	0.01	4	50	0.05	0.5	94.8446	2.0	0.8775
4	0.1	0.01	4	100	0.05	0.5	251.9345	1.9	0.7775

Cuadro 2: Primeros y últimos valores de un csv generado con el QAOA

Esta tabla cuenta con los valores de los parámetros introducidos para cada iteración, así como tres columnas adicionales con los resultados recogidos. La primera de ellas es el tiempo de ejecución del algoritmo. Las otras dos columnas añadidas son el valor del mínimo obtenido usando esos parámetros y su desviación con respecto al valor real del mínimo respectivamente.

Al juntar los tres ficheros csv generados por el algoritmo, se obtiene finalmente unos datos completos sobre los que extraer las conclusiones. Se busca encontrar los parámetros que corresponden al valor más preciso y consistente y también las relaciones entre los parámetros introducidos y los valores obtenidos.

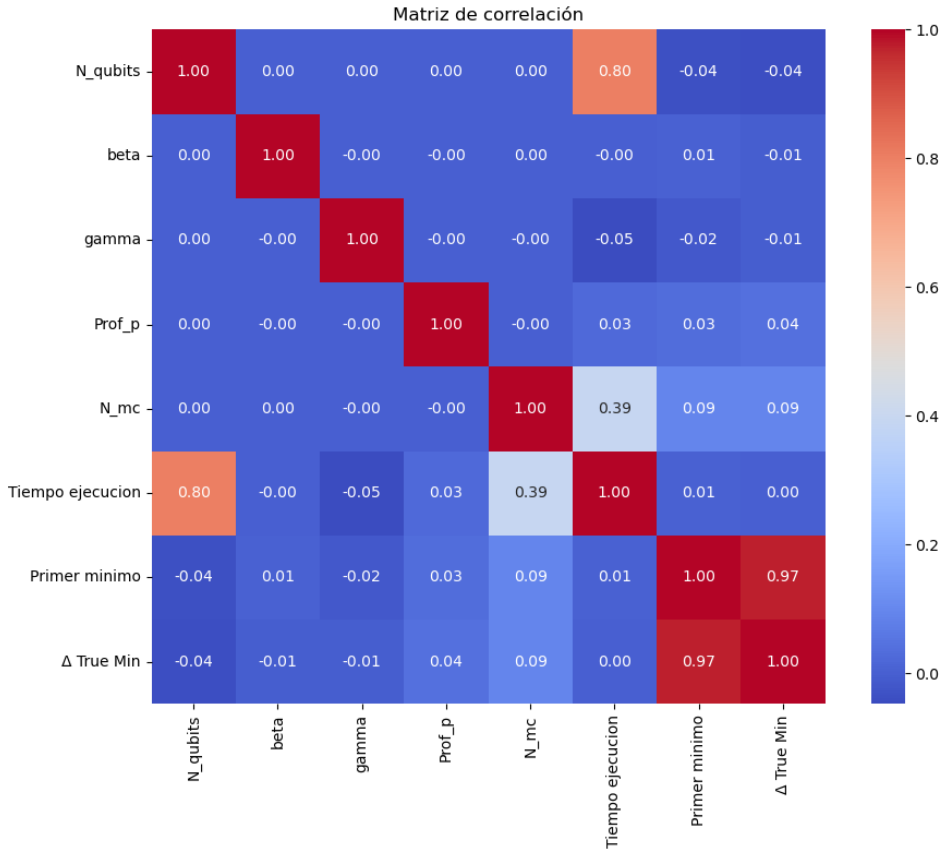


Figura 10: Matriz de correlación del QAOA

A través de una matriz de correlación se puede estudiar la relación entre los parámetros y los resultados obtenidos. En este caso, los parámetros que más destacan son el número de qubits y número de iteraciones de Montecarlo, los cuales influyen de gran manera en el tiempo de ejecución. Para tener una mejor visión de la relación entre los parámetros se pueden observar directamente las gráficas entre estos parámetros.

Esta gráfico de cajas muestra como al aumentar el número de qubits, no solo aumenta el tiempo de cómputo, sino que también su dispersión. Este tipo de gráficos consisten un una primera caja donde se encuentran los datos dentro del rango intercuartílico, es decir, entre el primer y tercer cuartil. Dentro de esta caja se encuentra además indicada la mediana. Y, a cada lado de la caja, unos ”bigotes” que indican los valores extremos dentro de los datos.

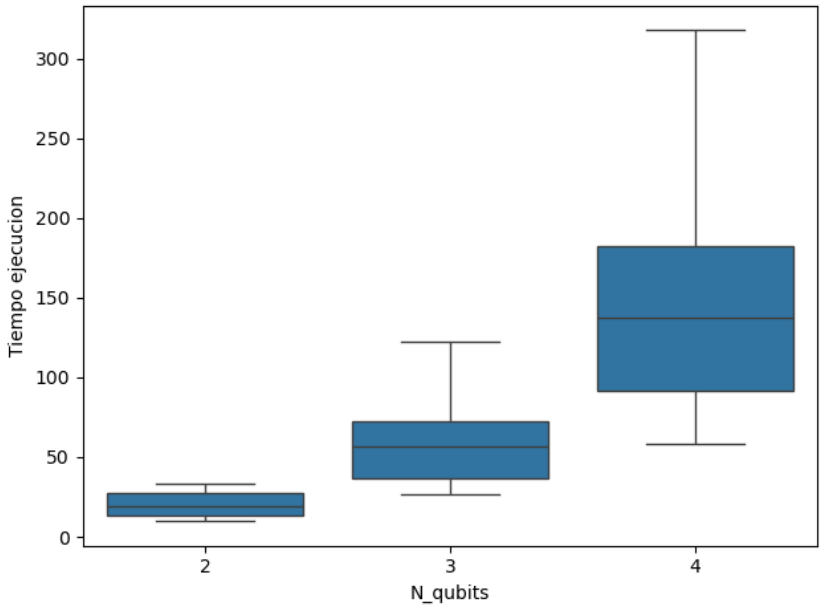


Figura 11: Relación entre el número de qubits y el tiempo de cómputo

Se puede observar en esta gráfica cómo el número de qubits dispara la complejidad de los cálculos, lo que se traduce en un mayor tiempo de cómputo. De una manera similar ocurre con el número de Montecarlo, pues este también aumenta el número de ejecuciones del algoritmo.

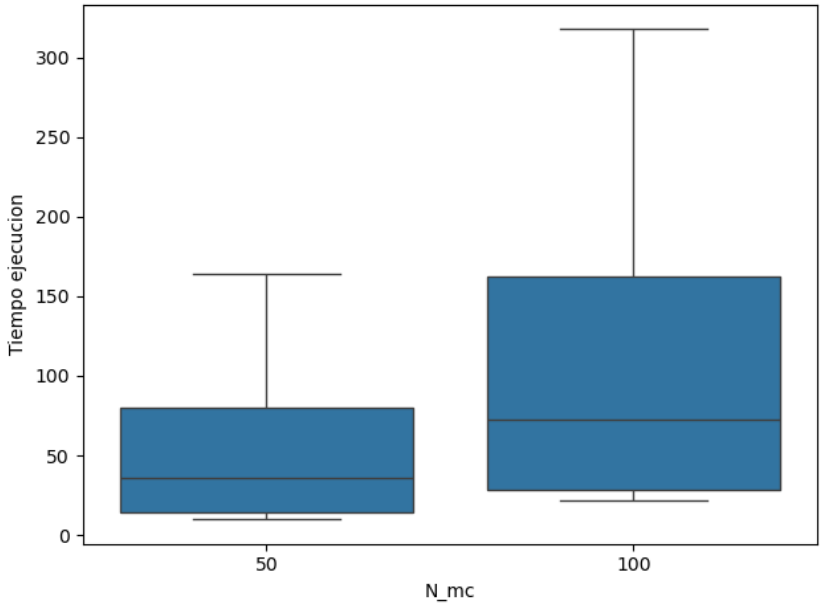


Figura 12: Relación entre el número de Montecarlo y el tiempo de cómputo

A primera vista, desde la matriz de correlación parece que la única influencia en la precisión de los mínimos viene dada por el número de iteraciones de Montecarlo. Para poder

conocer la precisión del algoritmo se visualizan estos parámetros. La gráfica resultante refleja imprecisión a la hora de encontrar una solución, pues sin importar el número de iteraciones de Montecarlo, la precisión de los datos es muy pobre. Además, la dispersión de las soluciones es muy alta.

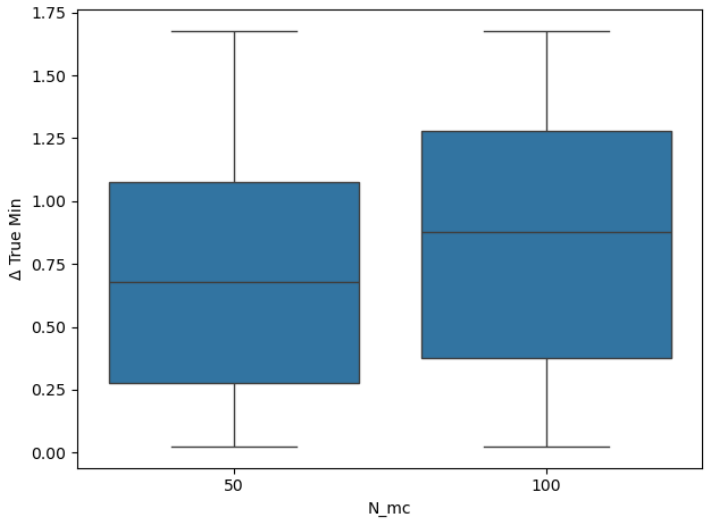


Figura 13: Relación entre el número de iteraciones de Montecarlo y la precisión

Estos datos no solo indican que el algoritmo tiene mejor precisión para menores iteraciones de Montecarlo, sino que además ambos casos tienen distribuciones uniformes. Esto ya es un indicativo de que el algoritmo no es capaz de localizar correctamente el mínimo de este potencial.

No obstante, si se observan los valores obtenidos que más se aproximan a una solución real se encuentra que para 61 combinaciones de parámetros se encuentra el mínimo en 1.1. Muy aproximado al valor real de 1.1225. Sin embargo, en estas soluciones aparecen una gran variedad de parámetros lo que parece indicar que estos resultados no son fruto de unos parámetros adecuados que reproduzcan una solución correcta si no que por la aleatoriedad e imprecisión han resultado cerca del mínimo esperado. Para confirmar esto, se puede tomar el valor que menor error presente e intentar reproducir sus resultados, ya que debe ser el más consistente.

N_qubits	beta	gamma	Prof_p	N_mc	Toler	Learning rate
4	0.3	0.25	3	100	0.05	0.5

Cuadro 3: Hiperparámetros óptimos del QAOA para potencial de Lennard-Jones

Al ejecutar a través de la interfaz, se obtiene mayor información, ya que se cuenta con la precisión total de los datos, así como con la distribución de las soluciones de Montecarlo.

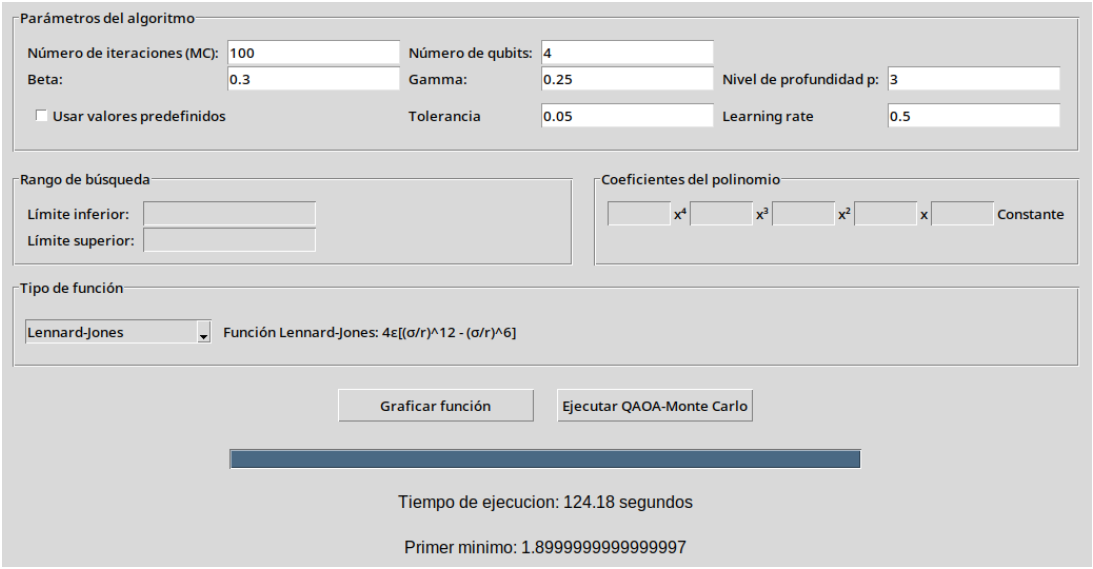


Figura 14: Interfaz para ejecutar el QAOA

La gráfica de la ejecución muestra una gran dispersión de los parámetros para una gran variedad de valores. A demás, el valor que determina como mínimo esta mal localizado.

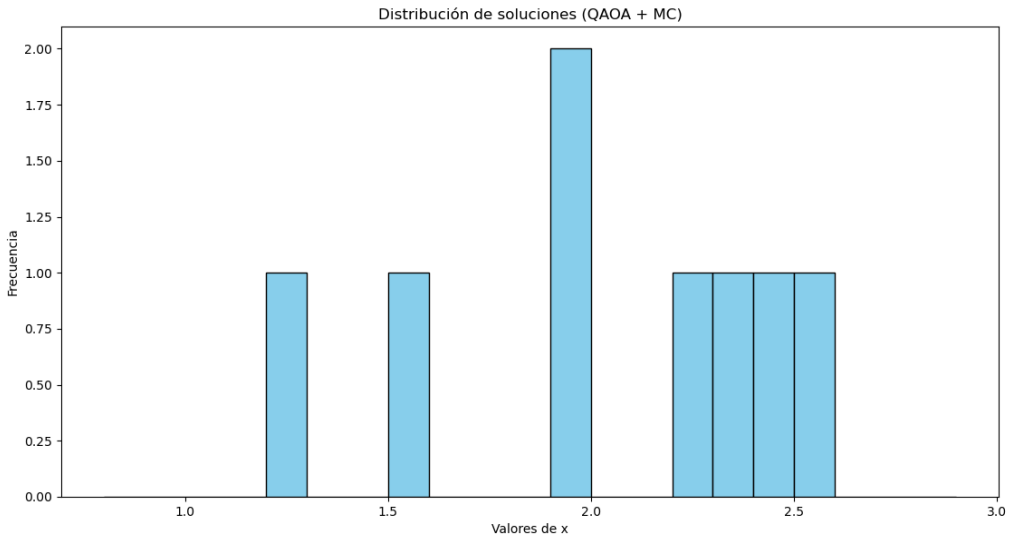


Figura 15: Medidas obtenidas para los mejores parámetros

Con todo lo observado hasta ahora, se puede concluir que este algoritmo, al menos como se ha concebido en este trabajo no es capaz de encontrar la solución para este potencial.

5.1.2. Gradient Descent

N_iter	N_mc	Toler	Learning rate	Tiempo ejecucion	Primer mínimo	True min
100	50	0.1000	0.1000	0.0182	2.7	1.5775
100	50	0.1000	0.0500	0.0159	1.1	0.0225
100	50	0.1000	0.0100	0.0139	2.2	1.0775
100	50	0.1000	0.0001	0.0148	1.7	0.5775
100	50	0.0500	0.1000	0.0152	0.9	0.2225
...	...	...	...	...	...	...
2000	200	0.0100	0.0001	0.0339	2.3	1.1775
2000	200	0.0001	0.1000	2.4819	2.9	1.1775
2000	200	0.0001	0.0500	5.3047	2.7	1.5775
2000	200	0.0001	0.0100	2.4146	2.0	0.8775
2000	200	0.0001	0.0001	0.0352	2.7	1.5775

Cuadro 4: Primeros y últimos valores de un csv generado con el gradient descent

Repitiendo el procedimiento anterior, se generan los archivos scv y la tabla de valores correspondiente y tras esto se visualiza la matriz de correlación.

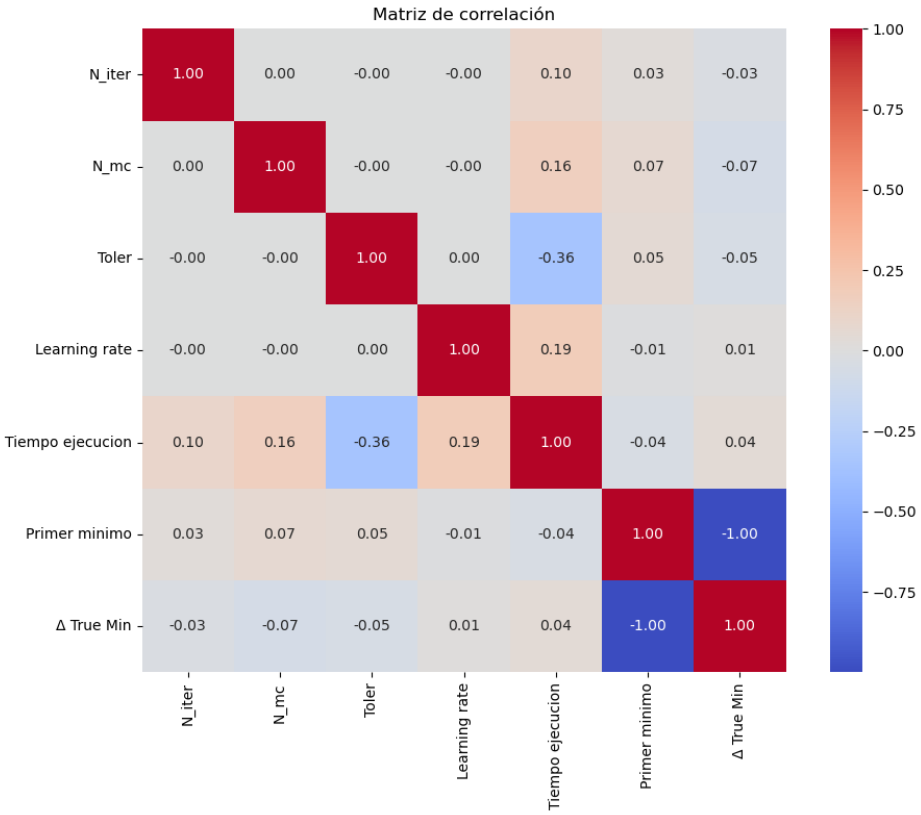


Figura 16: Matriz de correlación del gradient descent

Como se puede apreciar en esta matriz de correlación, salvo la tolerancia, la cual cuenta

con una relación mínima con el tiempo de ejecución, no existe ninguna relación considerable entre los parámetros y los resultados. Esto puede indicar que, al menos para este potencial, tampoco se puede obtener un mínimo con la precisión adecuada aunque todavía es pronto para concluir algo así.

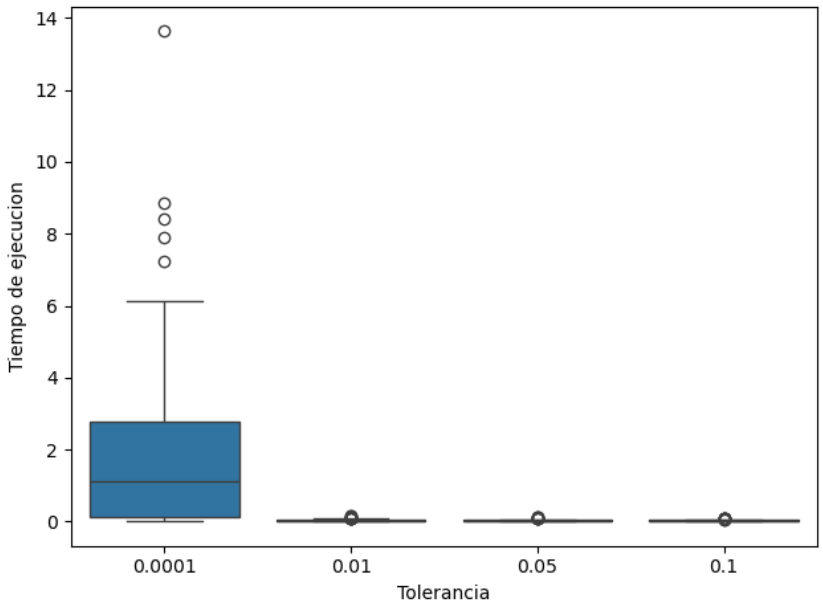


Figura 17: Relación entre la tolerancia y el tiempo de ejecución

Aquí se puede observar, que salvo para el caso extremo en el que la tolerancia es $0{,}0001$, el tiempo de ejecución es mínimo. Antes de tratar de visualizar más relaciones entre los parámetros, es conveniente localizar el mejor resultado, para poder ver qué tipo de parámetros los generan. Para este caso los parámetros que reproducen un mejor resultado son:

N_iter	N_mc	Toler	Learning rate
500	150	0.0001	0.1

Cuadro 5: Hiperparámetros óptimos del QAOA para potencial de Lennard-Jones

Estos parámetros devuelven un valor para el mínimo de 1.1063 segun los datos recogidos. Para asegurar que esta solución es resultado de una buena selección de los algoritmos, se va a ejecutar a través de la interfaz.

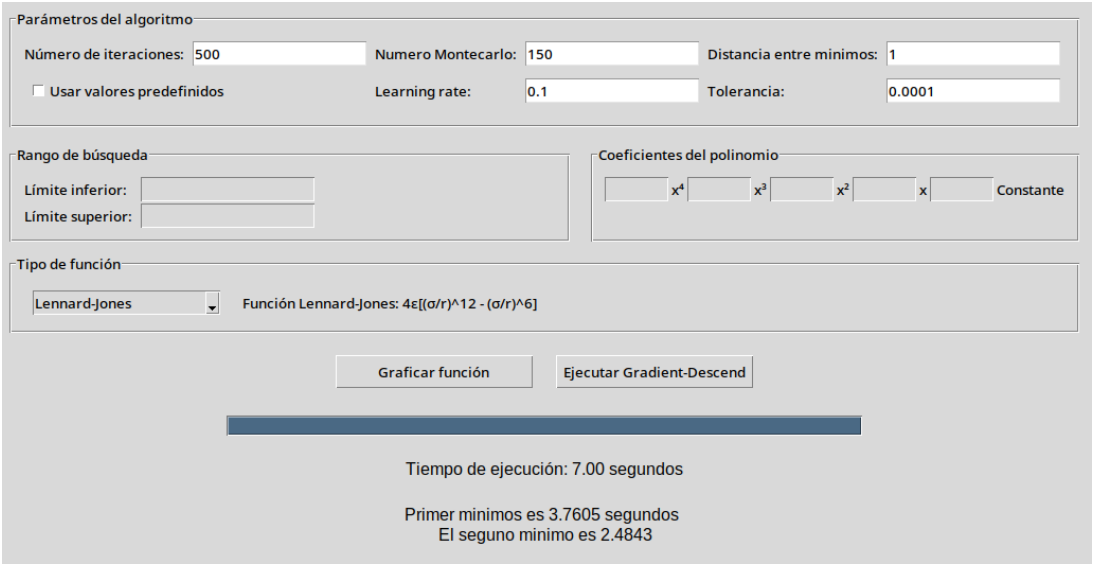


Figura 18: Interfaz para ejecutar el gradient descent

Se puede ver que los resultados obtenidos distan mucho de los resultados esperados y que el mínimo anterior no se debía a una correcta elección de los parámetros. Así, para este algoritmo también se puede afirmar que no es capaz de localizar el mínimo.

5.1.3. Método de Newton

Para el método de Newton, de la misma manera que se ha realizado para el gradient descent, se ejecuta hasta obtener los 3 ficheros cvs. Luego se genera un dataframe con la combinación de estos y, por último, una matriz de correlación donde poder observar cómo los parámetros influyen en los resultados.

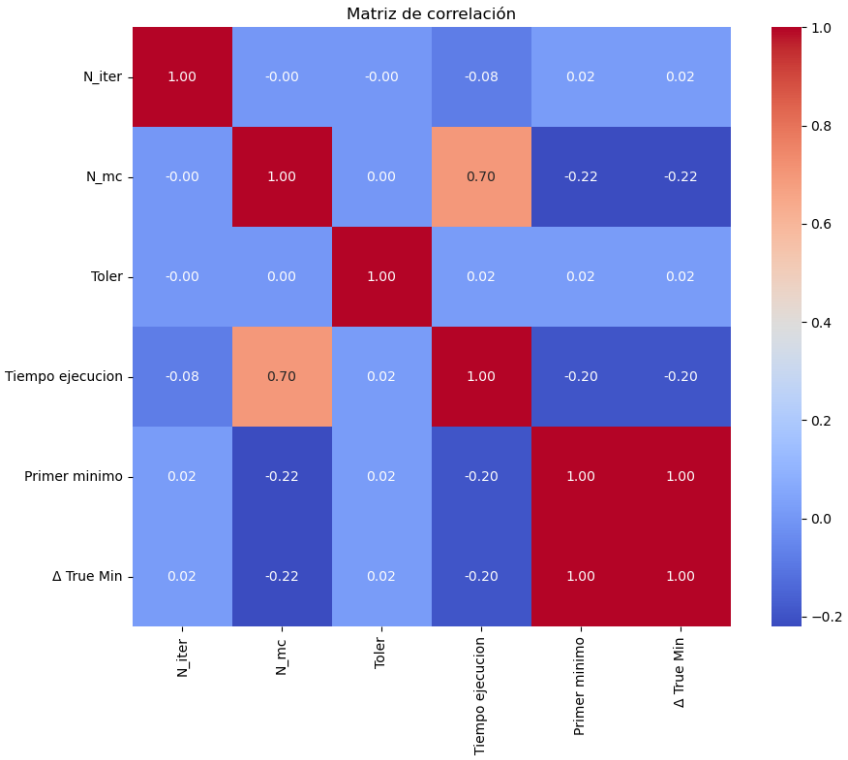


Figura 19: Matriz de correlación del método de Newton

Se puede apreciar que el único parámetro que tienen un impacto significativo es el número de iteraciones de Montecarlo. Este repercute en el tiempo de ejecución y en el primer mínimo. Esta información, junto a los resultados obtenidos con los otros algoritmos, parece indicar que el método de Newton tampoco va a presentar resultados sólidos. Sin embargo, aun hay que estudiar los resultados. Para ver la relación más en profundidad se pueden generar las gráficas necesarias. En este caso, relacionando el número de Montecarlo con el tiempo de ejecución.

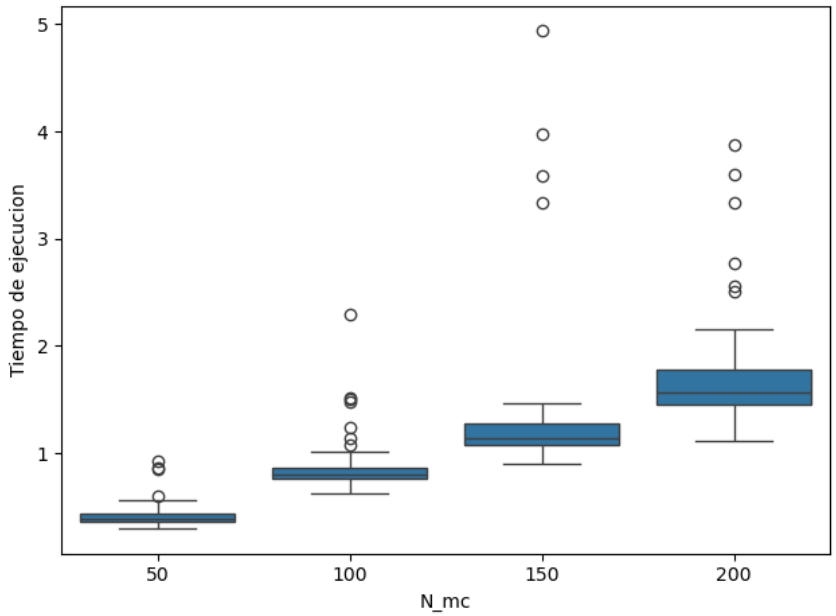


Figura 20: Relación entre el número de Montecarlo y el tiempo de ejecución

Como se puede ver, el número de iteraciones de Montecarlo está relacionado de manera casi lineal con el tiempo de ejecución, como cabría esperar. Sin embargo, la precision de los datos es muy baja para todos los valores de Montecarlo. Además, en todos los casos, la amplitud y dispersión de los valores es enorme.

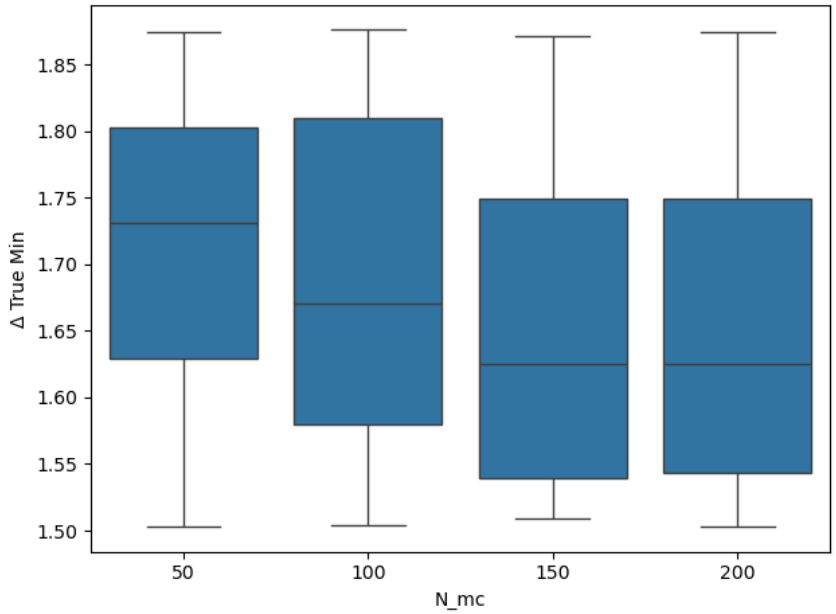


Figura 21: Relación entre el número de Montecarlo y la precisión de la solución

Estos datos no parecen indicar una buena relación entre los parámetros y los resultados.

De hecho, al tomar el mejor resultado esto se confirma. Y es que, de todos los valores generados para este potencial, aquel que más se ha aproximado a la solución lo ha hecho con un valor de $2{,}6253$, muy alejado del valor real. Con todo esto, este algoritmo se considera, al menos concebido como lo está en este trabajo, totalmente inservible para resolver este potencial.

5.2. Potencial de Higgs

El potencial de Higgs presenta dos mínimos globales en -1 y 1. Para ambos valores la función presenta un mínimo. Es por eso que los datos obtenidos para este potencial cuentan con más columnas. De la misma manera ocurrirá con el siguiente mínimo, el cual también cuenta con dos mínimos. En este caso, al ser ambos mínimos globales, se consideran el primer mínimo en 1 y el segundo en -1, o viceversa, soluciones idénticas. Esto no sería así si un mínimo fuera global y otro local. En ese caso, el primer mínimo tendría que ser el global y el segundo el local.

5.2.1. QAOA

Al ejecutar el programa para todos los parámetros indicados, se obtiene un archivo csv con esta forma. Donde las columnas de *True Min* hacen referencia a la distancia del mínimo calculado con respecto al mínimo real.

N_qubits	beta	gamma	Prof_p	N_mc	Toler	Learning rate	Tiempo ejecucion	Primer mínimo	True min	Segundo mínimo	True min 2
2	0.5	0.50	2	50	0.05	0.5	22.4645	-1.0	8.88e-16	0.8	2.00e-01
2	0.5	0.50	2	100	0.05	0.5	47.0457	-1.0	8.88e-16	0.7	3.00e-01
2	0.5	0.50	3	50	0.05	0.5	23.9527	-1.0	8.88e-16	0.7	3.00e-01
2	0.5	0.50	3	100	0.05	0.5	48.2147	-1.1	1.0e-01	0.7	3.00e-01
2	0.5	0.50	4	50	0.05	0.5	11.9297	-1.1	1.00e-01	0.8	2.00e-01
...	...	...	...	...	...	...	...	...	...	...	...
4	0.1	0.01	2	100	0.05	0.5	396.8081	-1.6	6.00e-01	0.3	7.00e-01
4	0.1	0.01	3	50	0.05	0.5	182.7692	-1.5	5.00e-01	-1.2	2.00e-01
4	0.1	0.01	3	100	0.05	0.5	241.3784	1.3	3.00e-01	-1.3	3.00e-01
4	0.1	0.01	4	50	0.05	0.5	120.3056	1.1	1.00e-01	0.9	1.00e-01
4	0.1	0.01	4	100	0.05	0.5	238.5175	1.1	1.00e-01	1.0	2.66e-15

Cuadro 6: Primeros y últimos valores del csv del QAOA para Higgs

Ahora, con los 3 csv's generados, se puede generar un cuarto fichero que considere la consistencia y dispersión de los resultados para las mismas condiciones. Para poder medir adecuadamente la dispersión de los datos se usa el *std* y el *CV* como métricas de dispersión

N_qubits	beta	gamma	Prof_p	N_mc	Prom. tiempo	Tiempo std	CV Tiempo	Min prom.	Min std	Min CV
2	0.5	0.50	2	50	24.25	2.63	10.83	-1.03	0.0577	-5.87
2	0.5	0.50	2	100	48.37	5.36	11.09	-1.07	0.0577	-5.87
2	0.5	0.50	3	50	25.19	3.02	12.00	-1.00	0.0000	-0.00
2	0.5	0.50	3	100	44.21	3.84	8.68	-0.40	1.1269	-281.73
2	0.5	0.50	4	50	12.62	0.59	4.72	-0.40	1.2124	-303.11
...	...	...	...	...	...	...	...	...	...	...
4	0.1	0.01	2	100	305.38	79.47	26.02	-1.7	0.1732	-10.19
4	0.1	0.01	3	50	132.80	44.39	33.43	-0.43	1.5948	-368.03
4	0.1	0.01	3	100	283.54	52.86	18.64	0.20	1.4177	708.87
4	0.1	0.01	4	50	150.87	32.43	21.49	0.37	1.1846	323.08
4	0.1	0.01	4	100	266.66	47.83	17.94	1.07	0.0577	5.41

True Min avg	True Min std	True Min CV	2Min prom.	2Min std	2Min CV	True 2Min avg	True 2Min std	True 2Min CV
0.03	0.06	173.21	0.70	0.10	14.29	0.30	0.10	33.33
0.07	0.06	86.60	0.73	0.06	7.87	0.27	0.06	21.65
0.00	0.00	0.00	0.70	0.00	0.00	0.30	0.000	0.00
0.07	0.06	86.60	0.13	0.98	736.12	0.20	0.17	86.60
0.07	0.06	86.60	0.00	1.47	0.00	0.33	0.32	96.44
...	...	...	...	...	...	...	...	...
0.70	0.17	86.60	-0.97	1.14	-117.65	0.63	0.31	48.24
0.37	0.15	41.66	-1.13	0.31	-26.96	0.63	0.12	43.30
0.33	0.15	17.32	-0.50	1.06	-211.66	0.23	0.12	49.49
0.03	0.06	173.21	0.77	0.23	30.12	0.23	0.23	98.97
0.07	0.06	86.60	0.23	1.24	532.42	0.10	0.10	100.00

Cuadro 7: Resultados del QAOA para el potencia de higgs tras ejecutar 3 veces el algoritmo. Con valores redondeados.

Las columnas que van indicadas como *std* o como *CV* se refieren a la desviación estándar y al coeficiente de variación respectivamente. Estos valores son diferentes maneras de medir la dispersión de los resultados entre medidas, y son útiles para poder comparar entre diferentes medidas.

Con el objetivo de estudiar la relación entre los parámetros y los resultados, se visualiza la relación entre los diferentes parámetros. Esto se puede realizar a través de la matriz de correlación, de la misma manera que para el potencial de Lennard-Jones.

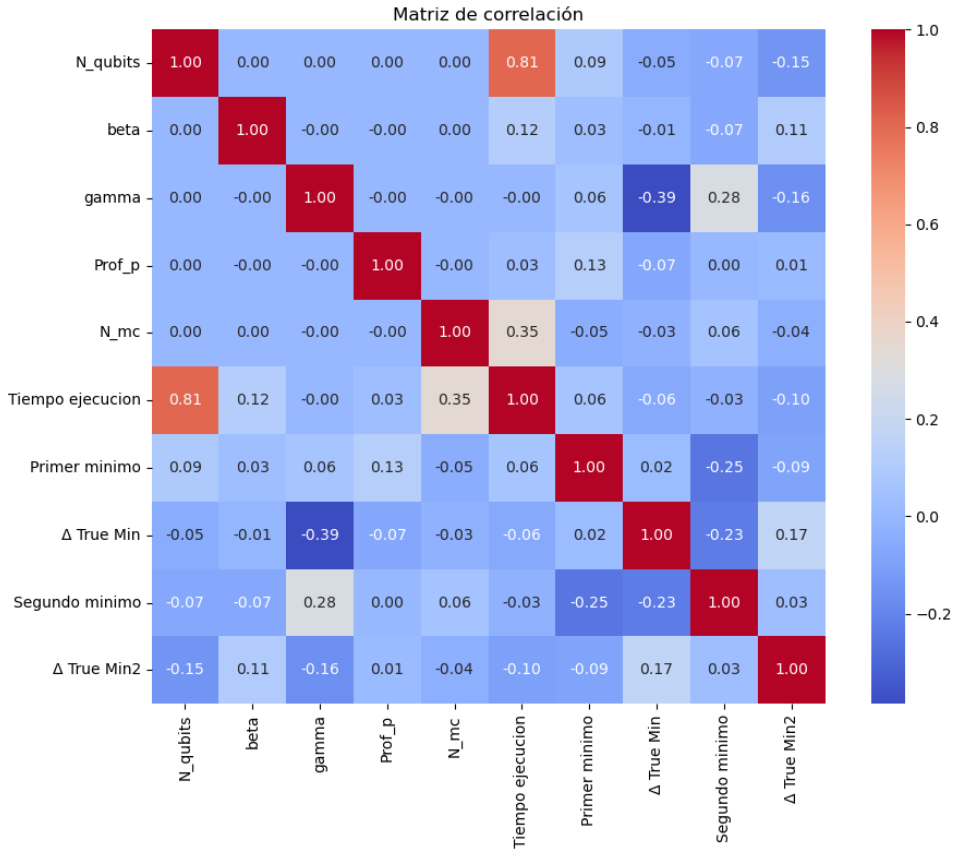


Figura 22: Matriz de correlación para los parámetros de QAOA

A diferencia de las matrices anteriores, en esta sí se pueden observar múltiples relaciones entre los parámetros y los resultados. El parámetro que más destaca es el número de qubits, el cual está altamente relacionado con el tiempo de ejecución.

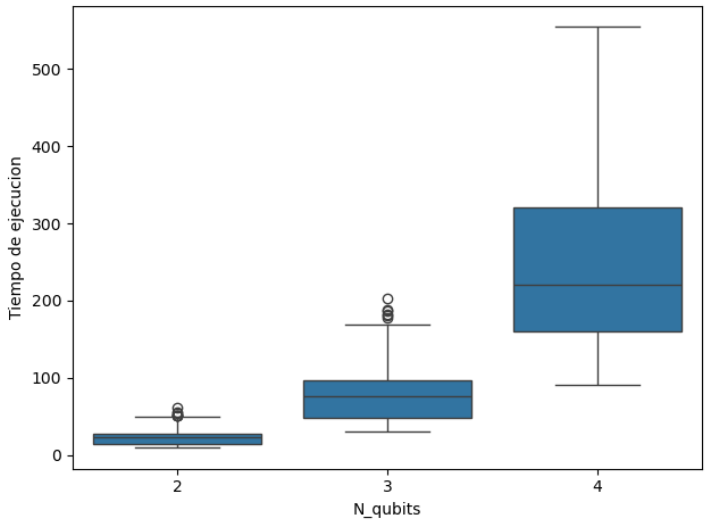


Figura 23: Gráfico de barras entre el número de qubits y el tiempo de ejecución

Como se puede observar, parece que hay una relación casi exponencial entre el número de qubits y el tiempo de ejecución, además, este parámetro, junto con el número de Montecarlo, son los únicos que impactan de forma relevante en el tiempo de ejecución. Como cabría esperar ya que son los que aumentan la complejidad de las operaciones y el número de estas respectivamente.

Aparentemente no existe una gran correlación entre la solución y el número de qubits. Sin embargo, al visualizar los datos se puede apreciar cómo la precisión de los datos se mejora con el número de qubits tanto en la primera solución, como en la segunda solución.

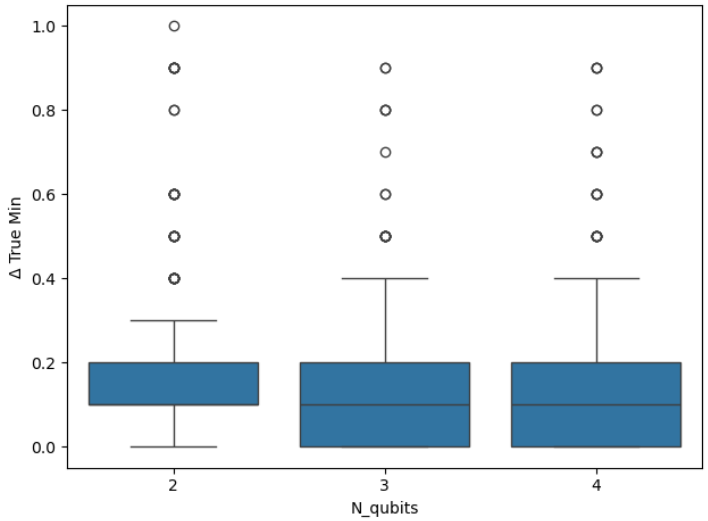


Figura 24: Relación entre el número de qubits y la precision de la primera solución

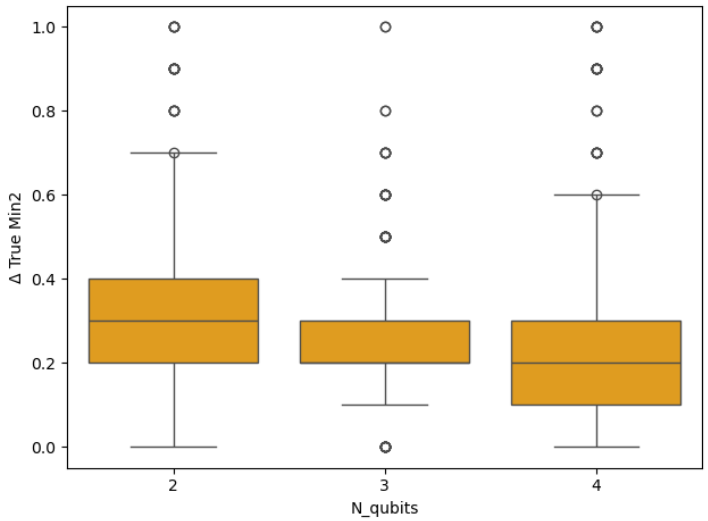
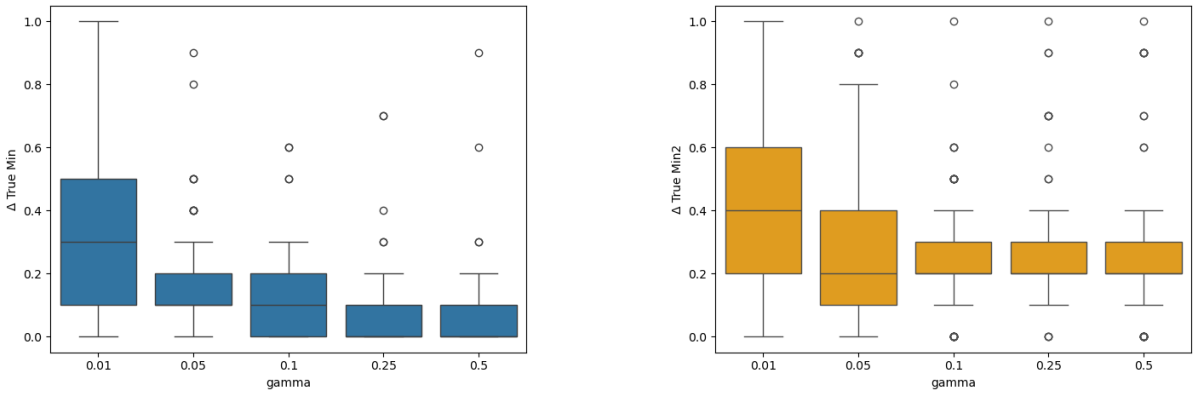


Figura 25: Relación entre el número de qubits y la precisión de la segunda solución

Las ”cajas” de los gráficos se acercan más a 0 conforme los qubits aumentan. Esto se

traduce en resultados más precisos. En la matriz de correlación también salta a la vista que el parámetro que más influye en la precisión de las soluciones, sobre todo en la primera solución, es el parámetro gamma. Al gráficar este parámetro, se puede ver todavía más claro.



(a) Primera solución

(b) Segunda solución

Figura 26: Relación entre el gamma y la dispersión con las soluciones

Se puede ver que el parámetro gamma ejerce una gran influencia sobre los resultados. La dispersión de los datos respecto a la soluciones indica cómo se comporta para cada valor. Siendo el que mejor resultados reproduce para valores de entre gamma $= 0{,}25$ y gamma $= 0{,}5$. Por lo que se puede esperar que este parámetro sea el que ofrezca los mejores resultados.

Con todos los resultados obtenidos para todos los posibles parámetros, es de interés buscar aquellos para los cuales se observa una mejor solución, que en este caso sería encontrar los valores -1 y 1. Si se toman los datos para los cuales se obtienen estas soluciones, los parámetros que reproducen esta solución son los siguientes (donde la tasa de aprendizaje y la tolerancia se han omitido por ser parámetros fijos):

N_qubits	beta	gamma	Prof_p	N_mc	Tiempo ejecucion	Primer mínimo	Segundo mínimo
4	0.1	0.50	3	100	237.0484	-1.0	1.0
4	0.1	0.50	4	50	113.7055	-1.0	1.0
4	0.1	0.05	4	100	413.0613	-1.0	1.0
4	0.1	0.50	4	100	203.9234	-1.0	1.0
4	0.1	0.50	3	100	250.7845	-1.0	1.0

Cuadro 8: Primeros y últimos valores del csv del QAOA para Higgs

De todos estos datos, se puede observar que los parámetros número de qubits y beta no varían mientras que gamma lo hace para un caso particular. Además, los parámetros de

profundidad y el número de iteraciones son prácticamente máximos, lo que confirma que estos parámetros aumentan la precisión de los resultados como se había precedido con anterioridad.

Ahora, de todos estos, para elegir no solo el que es capaz de producir mejores parámetros, sino también el que es capaz de reproducirlos con mayor consistencia se va a tomar aquel que tiene menor variación en su medida.

N_qubits	beta	gamma	Prof_p	N_mc	Tiempo ejecucion	Primer mínimo	Segundo mínimo	True Min CV	True Min2 CV
4	0.1	0.50	3	100	237.0484	-1.0	1.0	173.2	49.48
4	0.1	0.50	4	50	113.7055	-1.0	1.0	26.6	163.2
4	0.1	0.05	4	100	413.0613	-1.0	1.0	86.6	114.5
4	0.1	0.50	4	100	203.9234	-1.0	1.0	0	86.6
4	0.1	0.50	3	100	250.7845	-1.0	1.0	173	49.48

Cuadro 9: Primeros y últimos valores del csv del QAOA para Higgs

De entre todos los valores tomados, se selecciona aquel que presenta una mayor consistencia, el cual según la muestra tomada es el cuarto caso de la tabla anterior. Si se ejecuta el QAOA con los parámetros indicados, se puede obtener un resultado como el mostrado a continuación. Estos resultados pueden no ser reproducibles ya que es un algoritmo con componente aleatorio que no se puede obviar a pesar de tener los parámetros bien ajustados.

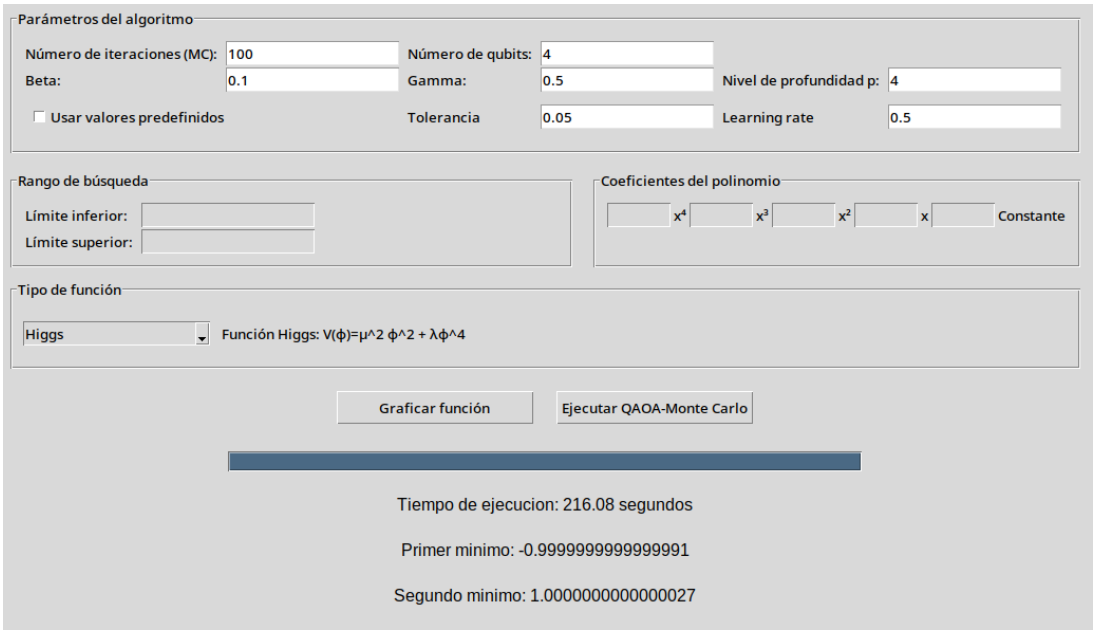


Figura 27: Resultados para los mejores parámetros

Como se podría esperar, los resultados del algoritmo para estos parámetros son muy precisos y encuentran las soluciones esperadas. También se observa la dispersión de las medidas

para cada iteración del Montecarlo. Se hacen evidentes dos picos de medidas sobre cada una de las soluciones las cuales se traducen en la solución final encontrada.

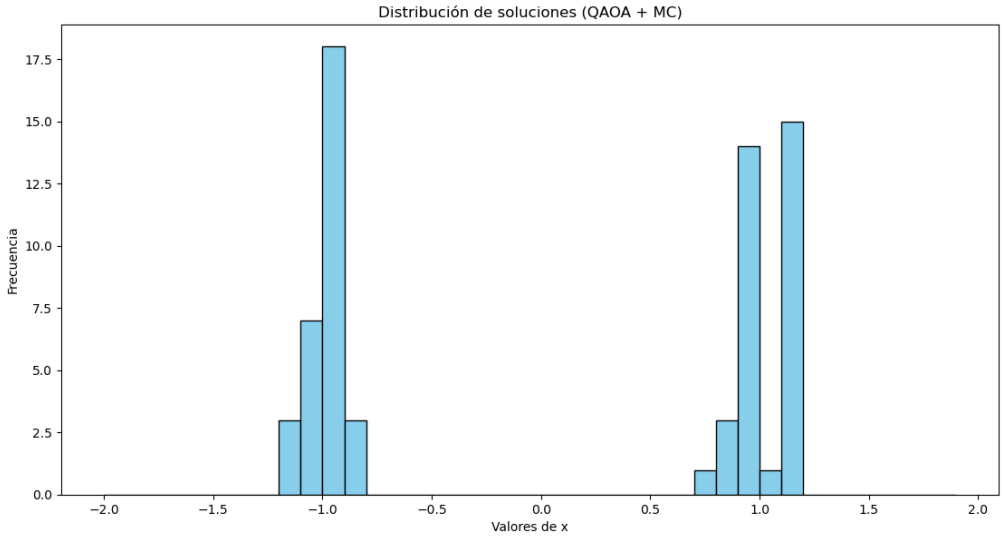


Figura 28: Resultado de cada una de las iteraciones del MC

En contraparte, también es interesante ver qué parámetros reproducen los peores resultados. Cabría esperar que los parámetros contrarios a los considerados para la mejor solución (es decir, los que se indicaban como los menos óptimos) sean los que reproduzcan peores resultados. Si se encontraran parámetros similares a los que se consideraban para la mejor solución en la peor, indicaría un algoritmo muy poco consistente y fiable. Utilizando la misma metodología que para localizar el mejor resultado, se encuentran:

N_qubits	beta	gamma	Prof_p	N_mc	Tiempo ejecucion	Primer mínimo	Segundo mínimo	True Min CV	True Min2 CV
2	0.1	0.01	2	50	12.3047	-2.0	-1.5	1.0	0.5

Cuadro 10: Primeros y últimos valores del csv del QAOA para Higgs

Como era de esperar, el caso en el que se obtienen peores resultados es aquel con menor número de qubits, menor profundidad y menor número de iteraciones de Montecarlo es menor. Parámetros que influyen directamente en el resultado y que cuanto mayores sean mejores resultados se pueden esperar. Así como el parámetro gamma, parámetro muy sensible, se encuentra muy lejos del parámetro óptimo para este algoritmo. Se puede visualizar este resultado de manera más detalla.

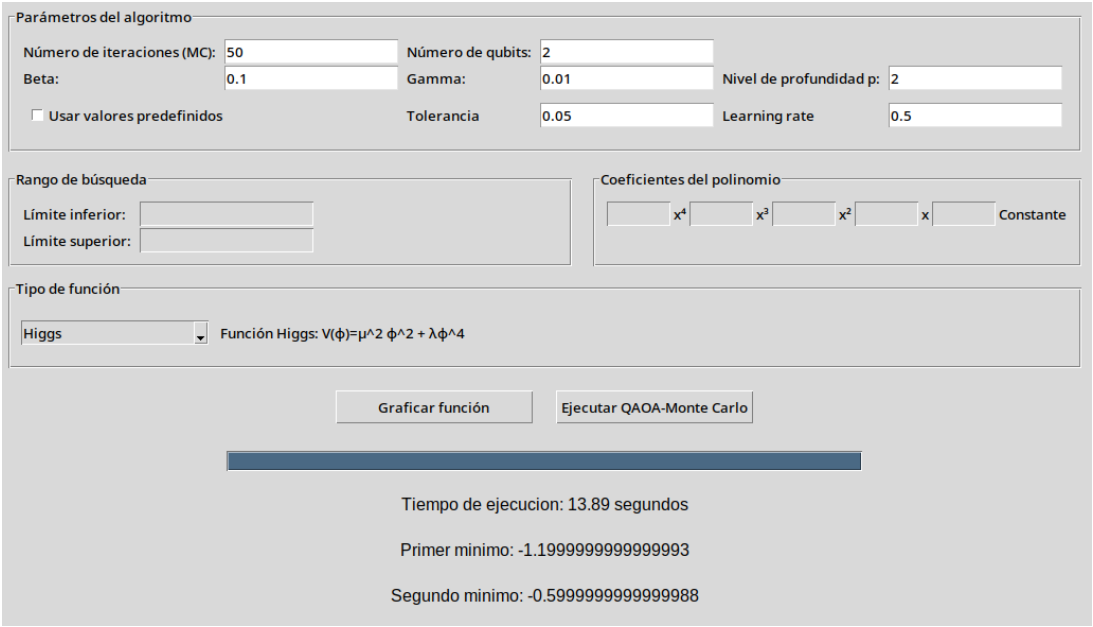


Figura 29: Interfaz para los peores parámetros del QAOA

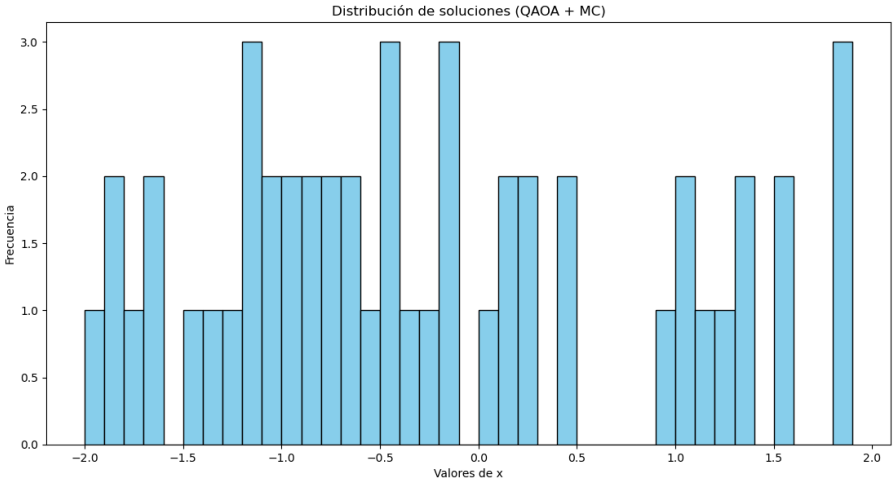


Figura 30: Resultados para los peores parámetros

Se observa cómo las medidas están totalmente dispersas. Claramente es una configuración muy poco consistente, precisa y fiable.

5.2.2. Gradient Descent

Realizando un procedimiento similar para el Gradient descent se obtiene unos datos similares a los generados con el QAOA, salvo que las columnas de los parámetros corresponden a los seleccionados para el gradient descent.

N_iter	Distancia mínima entre mínimos	N_mc	Toler	Learning rate	Tiempo ejecucion	Primer mínimo	True min	Segundo mínimo	True min 2
100	0.1	50	0.1000	0.1000	0.0142	1.1312	0.1312	0.7000	0.3000
100	0.1	50	0.1000	0.0500	0.0106	-1.3125	0.3125	-0.7000	0.3000
100	0.1	50	0.1000	0.0100	0.0089	-1.7000	0.7000	-0.4000	0.6000
100	0.1	50	0.1000	0.0001	0.0081	-0.2000	0.8000	0.5000	0.5000
100	0.1	50	0.0500	0.1000	0.0276	0.8539	0.1461	-0.8806	0.1194
...	...	...	...	...	...	...	...	...	
-0.4000	1.0	200	0.0100	0.0001	0.0286	-1.8000	0.8000	-1.6	0.6000
2000	1.0	200	0.0001	0.1000	0.3726	0.9998	0.0002	-0.9998	0.0002
2000	1.0	200	0.0001	0.0500	0.7279	-0.9995	0.0005	1.0004	0.0004
2000	1.0	200	0.0001	0.0100	2.8147	-1.0025	0.0025	1.0025	0.0025
2000	1.0	200	0.0001	0.0001	15.9800	-1.1915	0.1915	1.1915	0.1915

Cuadro 11: Primeros y últimos valores del csv del gradient descent para Higgs

De la misma manera, se visualizan a través de una matriz de correlación la relación entre los resultados y los parámetros obtenidos.

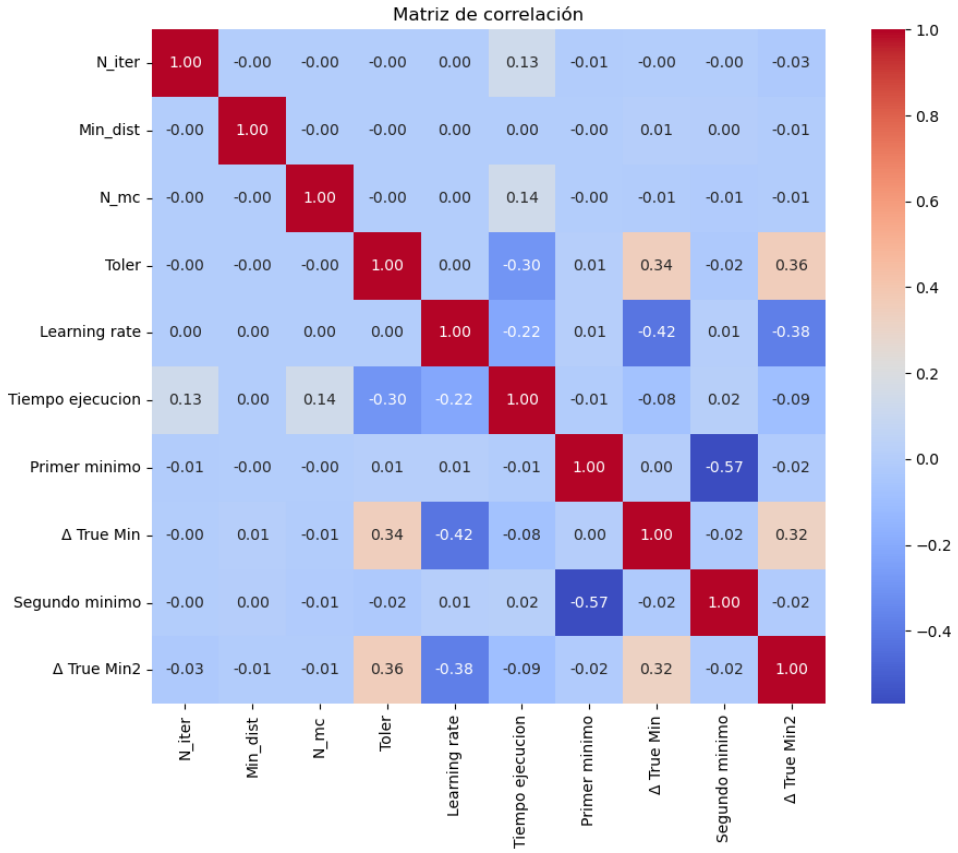


Figura 31: Matriz de correlación de los parámetros del gradient descent

Salta a la vista que los parámetros que más influyen en el tiempo de ejecución son la

tasa de aprendizaje, la tolerancia y en menor medida el número de iteraciones y número de Montecarlo. Como cabría esperar, los parámetros que aumentan la precisión de convergencia aumentan el tiempo de cómputo conforme estos disminuyen.

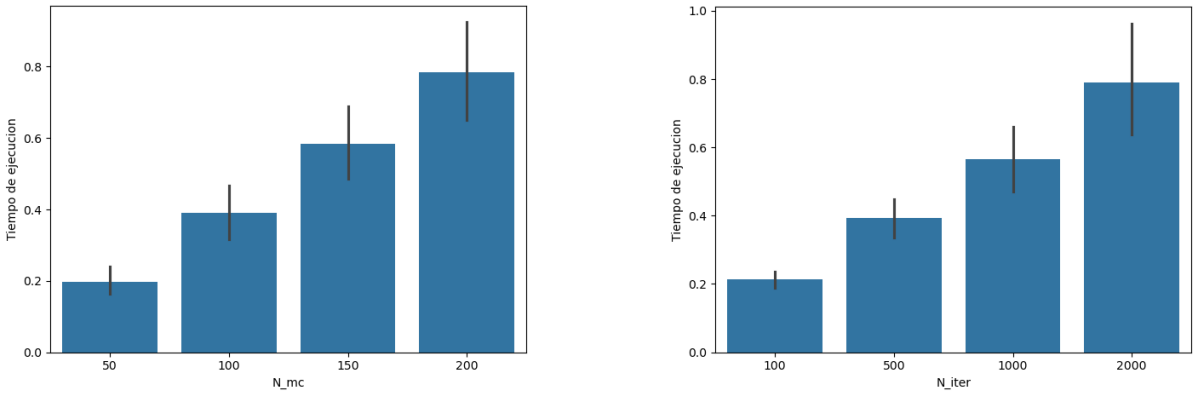


Figura 32: Relación entre el número de Montecarlo y de iteraciones y el tiempo de ejecución

Los parámetros que indican el número de ejecuciones como lo son *n_iter* y *N_mc* aumentan el tiempo de cómputo de manera prácticamente lineal. Por otro lado, los parámetros de tolerancia y de learning rate, también tienen una fuerte relación con el tiempo de ejecución.

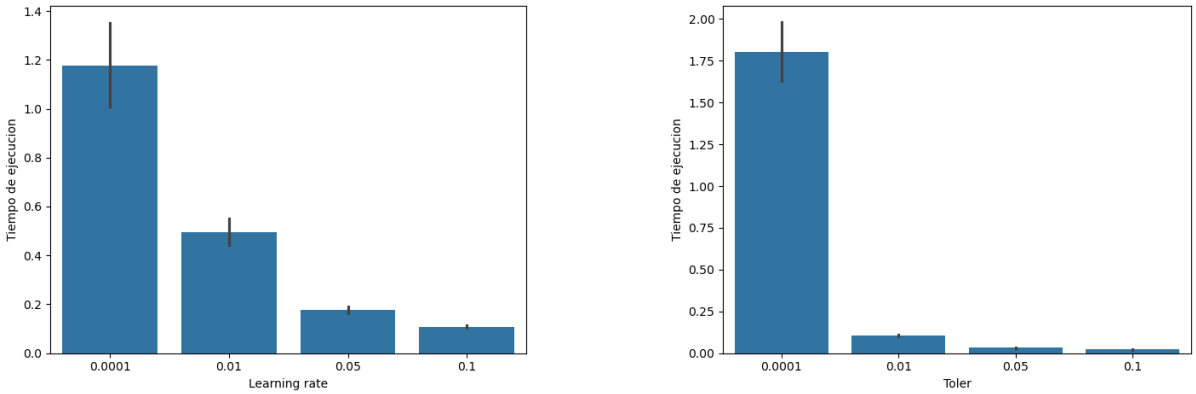


Figura 33: Relación entre la tolerancia y el learning rate y le tiempo de ejecución

También destacan mucho los parámetros de learning rate y tolerancia frente a la precisión de los datos. Como cabría esperar, una de las mayores ventajas de este tipo de algoritmos es su fuerte dependencia ante los pocos parámetros que tiene, lo que hace que estos sean responsables casi en su totalidad de la calidad de los resultados.

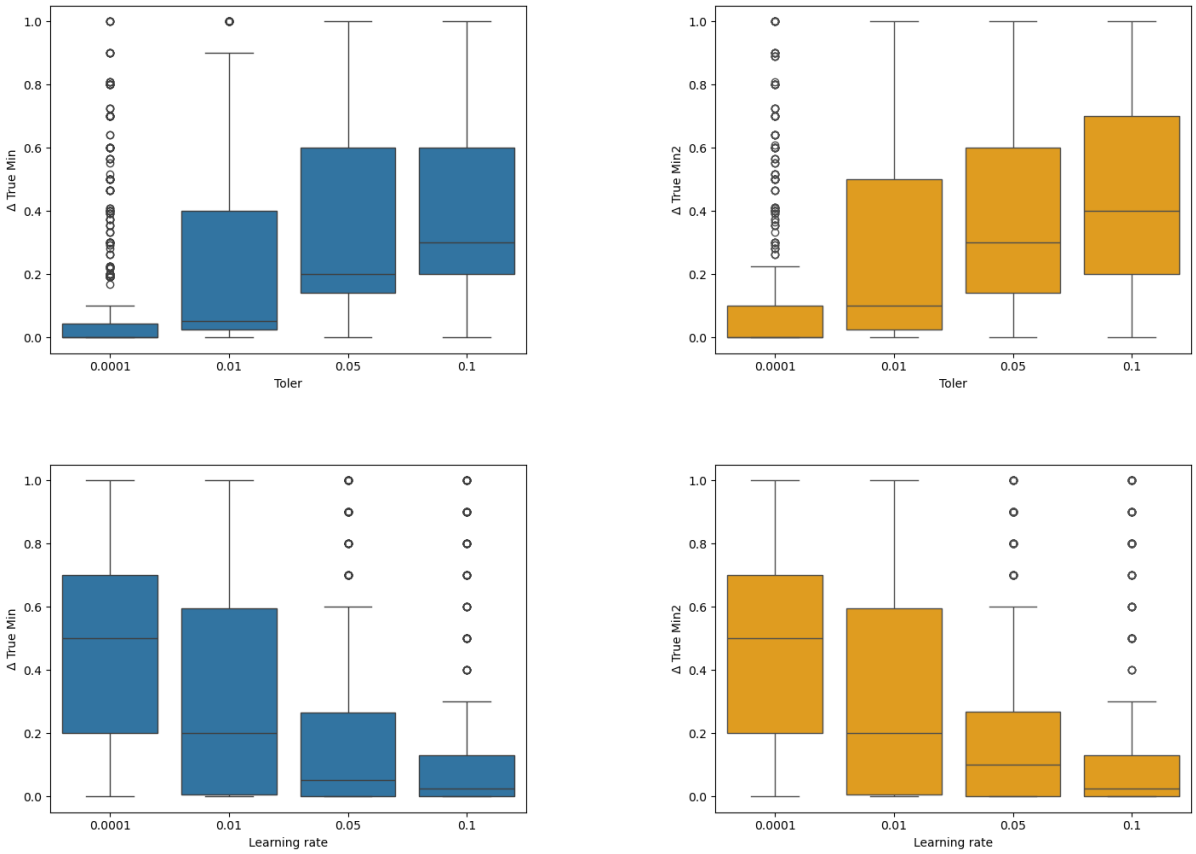


Figura 34: Relación entre los parámetros y la precision de los resultados

A través de los datos obtenidos, se puede tomar el resultado que mejor solución arroja. Este coincide con los parámetros:

N_iter	Distancia Minima entre mínimo	N_mc	Toler	Learning rate	Tiempo ejecucion	Primer mínimo	Segundo mínimo
2000	0.10	150	0.05	0.0001	0.0217	-1.0	1.0
2000	0.25	200	0.01	0.0100	0.1832	1.0	-1.0

Cuadro 12: Mejores parámetros para el gradient descent para Higgs

Sorprendentemente, ambos resultados ofrecen combinaciones de parámetros relativamente dispares. Ambos de ellos localizan los dos mínimos con la misma precisión, salvo que uno le asigna el primer mínimo al -1 mientras que otro lo hace al 1. En este caso, al tratarse de un potencial con dos mínimos locales, ambos casos son indistintos.

5.2.3. Método de Newton

Para el algoritmo de Newton se repite la metodología usada para evaluar el gradient descent. A través de la cual se toman las medidas de los datos, se transforman de distintas maneras los csv's generados y se visualiza la relación entre los parámetros y los resultados.

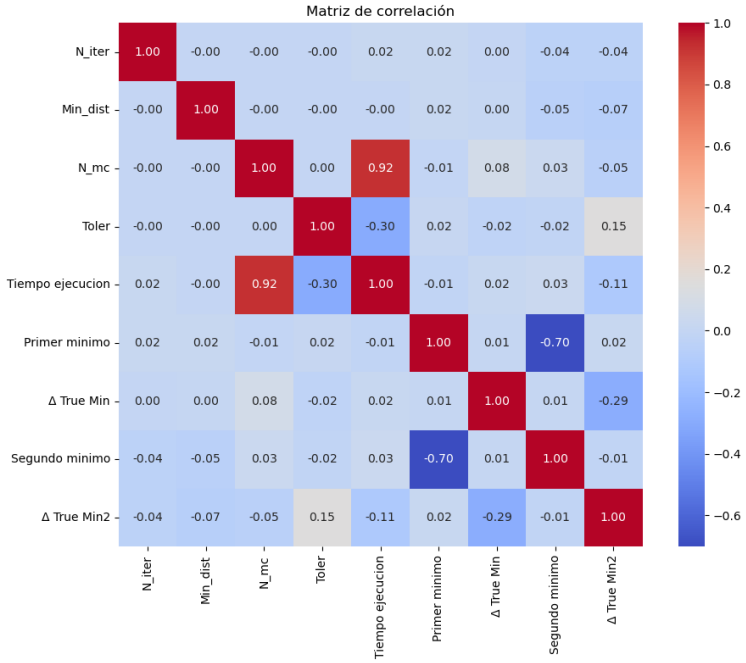


Figura 35: Matriz de correlación de los parámetros del método de Newton

Destaca, de la misma manera que en los otros algoritmos, la relación entre el número de ejecuciones de Montecarlo y el tiempo de ejecución. En este caso se ve perfectamente la relación entre las iteraciones de Montecarlo y el tiempo de cómputo.

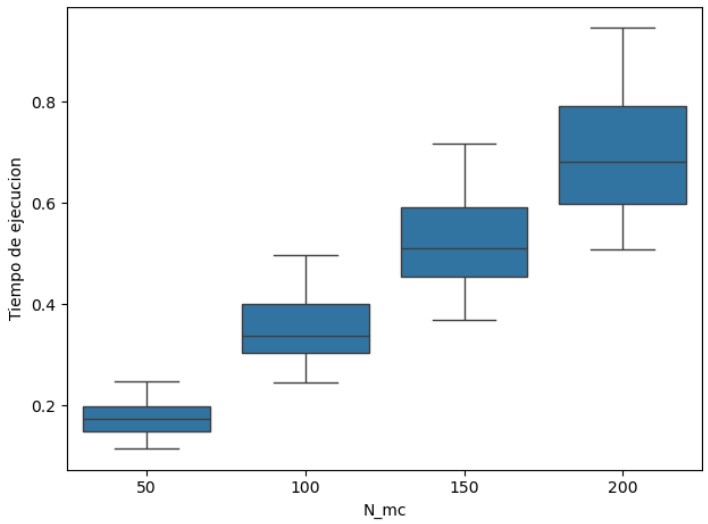


Figura 36: Relación entre el número de ejecuciones del Montecarlo y el tiempo de ejecución

También, como cabría esperar, se puede relacionar la tolerancia con el tiempo de cómputo, ambos parámetros están relacionados con el tiempo de ejecución ya que son los que aumentan el número de iteraciones y exigen más precisión para la convergencia.

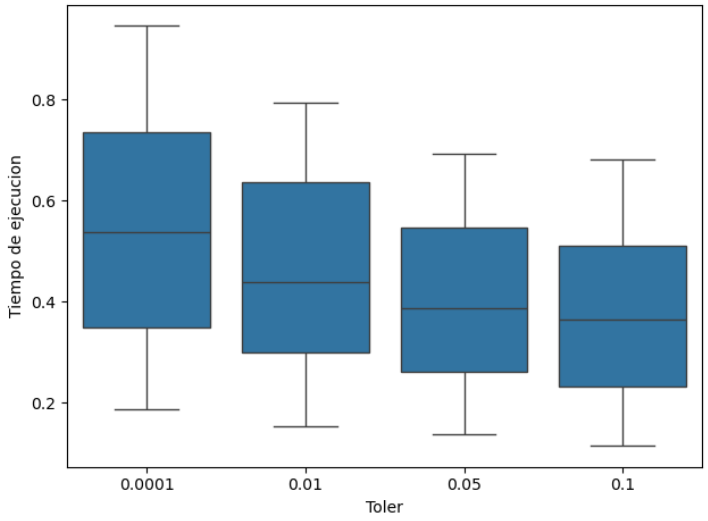


Figura 37: Relación entre el la tolerancia y el tiempo de ejecución

Respecto a la precisión en las medidas, se espera que sea el parámetro de tolerancia el mayor responsable de los valores.

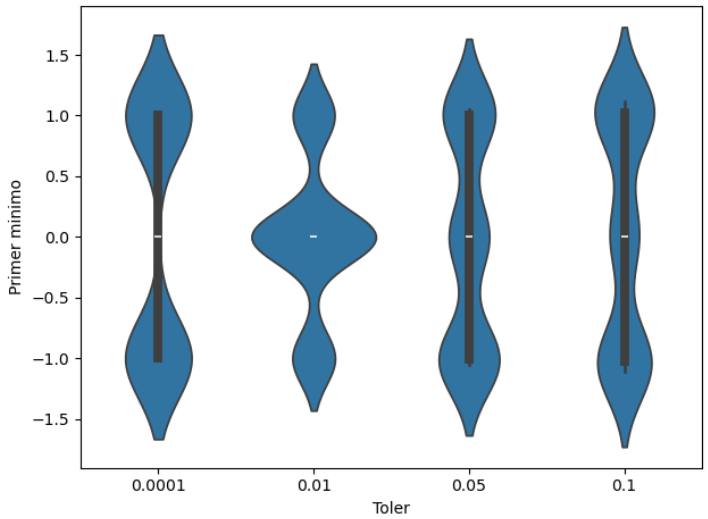


Figura 38: Relación entre la tolerancia y el primer mínimo

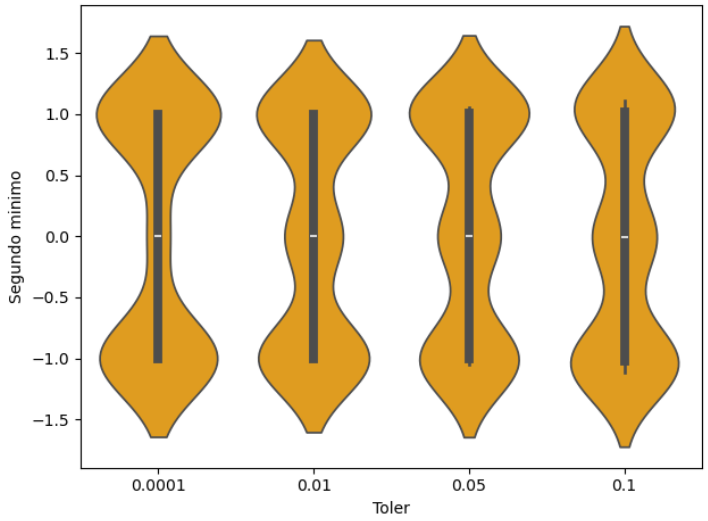


Figura 39: Relación entre la tolerancia y el segundo mínimo

Se puede ver como para menor tolerancia los resultados son mejores, al concentrase los valores entorno a -1 y 1, los valores reales de los mínimos. Sin embargo, en el cálculo del primer mínimo, se puede apreciar cómo existe una concentración de valores en 0. Esto ocurre porque este algoritmo, al depender de la derivada para determinar su movimiento a través del potencial, considera todos los puntos donde la pendiente sea nula como soluciones. Por eso algunos resultados de los mínimos se encuentran en 0, donde hay un máximo. Este fenómeno es altamente dominante para una tolerancia de $0{,}01$.

5.3. Potencial de hidrógeno

Este potencial tienen una peculiaridad. A diferencia del potencial de Lennard-Jones donde solo existía un mínimo, o del potencial de Higgs donde había dos mínimos, pero ambos eran mínimos globales. Aquí se observan dos mínimos, uno global y otro local. Se espera de los algoritmos que puedan localizar ambos mínimos y que puedan distinguir el global del local. Siendo el primer mínimo el global y el segundo mínimo el local.

5.3.1. QAOA

Con cada ejecución del QAOA para este potencial se obtiene un fichero csv con los parámetros introducidos, así como las columnas con los resultados. Estos datos se generan 3 veces para poder tener una muestra de datos menos sensibles a fenómenos aleatorios propios de este algoritmos.

N_qubits	beta	gamma	Prof_p	N_mc	Toler	Learning rate	Tiempo ejecución	Primer mínimo	True min	Segundo mínimo	True min 2
2	0.5	0.50	2	50	0.05	0.5	17.8229	-1.1	4.00e-02	0.9	2.00e-01
2	0.5	0.50	2	100	0.05	0.5	37.7537	-1.1	4.00e-02	0.9	2.00e-01
2	0.5	0.50	3	50	0.05	0.5	16.2167	-1.2	1.40e-01	1.5	4.00e-01
2	0.5	0.50	3	100	0.05	0.5	32.2957	1.1	2.66e-15	1.5	4.00e-01
2	0.5	0.50	4	50	0.05	0.5	15.7118	1.0	1.00e-01	-1.1	4.00e-02
...	...	...	...	...	...	...	...	...	...	...	...
4	0.1	0.01	2	100	0.05	0.5	264.2365	1.5	4.00e-01	-1.1	4.00e-02
4	0.1	0.01	3	50	0.05	0.5	130.4424	-1.3	2.40e-01	1.1	2.66e-15
4	0.1	0.01	3	100	0.05	0.5	242.8516	-1.3	2.40e-01	1.2	1.00e-01
4	0.1	0.01	4	50	0.05	0.5	143.1075	1.0	1.00e-01	-1.2	1.40e-01
4	0.1	0.01	4	100	0.05	0.5	276.4973	-1.1	4.00e-02	0.8	3.00e-15

Cuadro 13: Primeros y últimos valores del csv del QAOA para el Hidrógeno

Como hasta ahora, una vez recogido los datos, se genera una matriz de correlación donde se puedan observar los parámetros y sus relaciones.

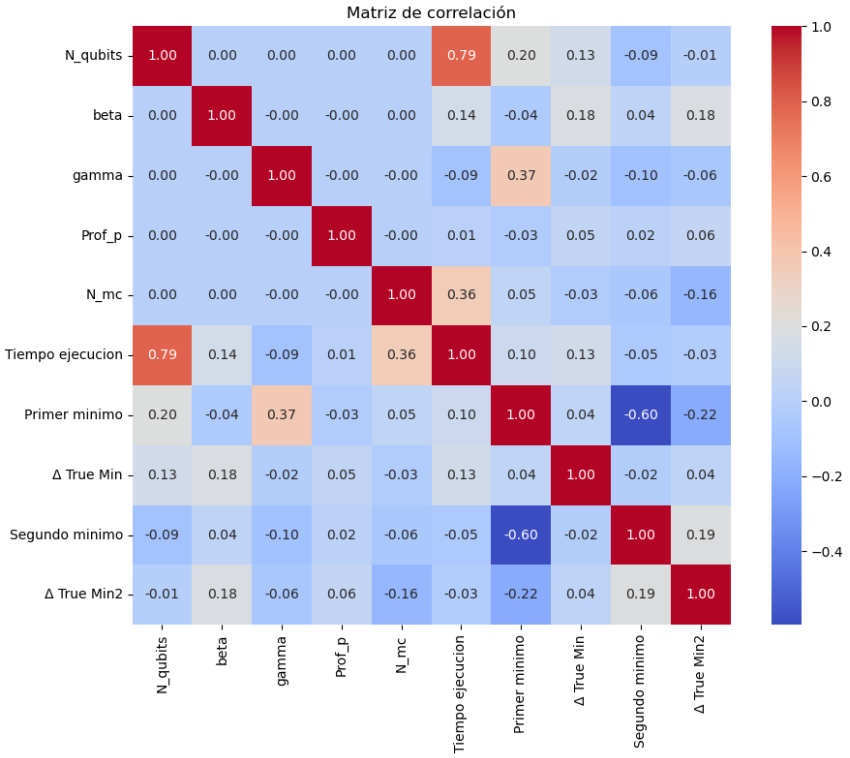


Figura 40: Matriz de correlación de los parámetros del QAOA

Como cabría esperar, la relación entre el número de qubits y el tiempo de ejecución sigue siendo como en los potenciales anteriores. Así como el número de iteraciones de Montecarlo también tiene una influencia similar a la que se observaba en el potencial de Higgs.

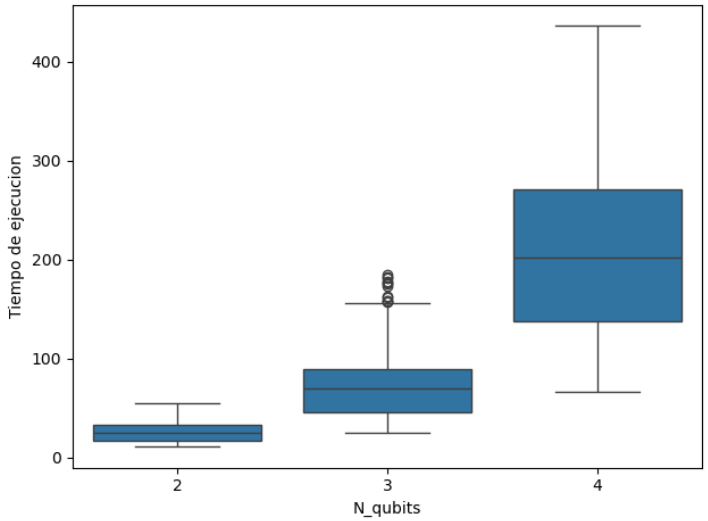


Figura 41: Relación entre el número de qubits y el tiempo de ejecución

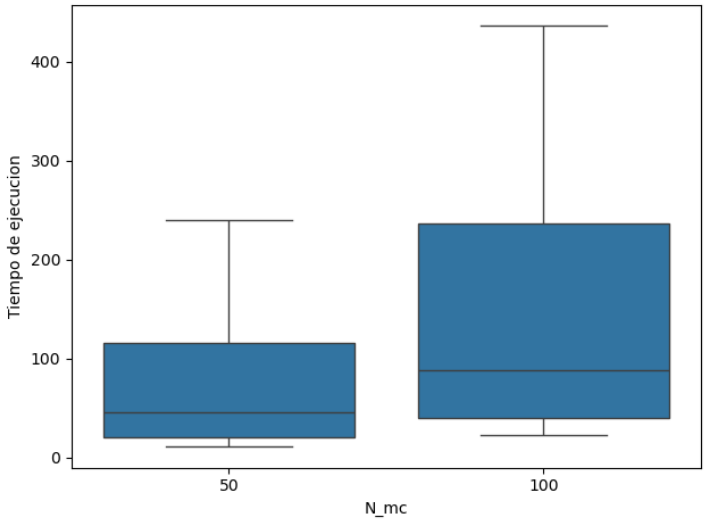


Figura 42: Relación entre el número de qubits y el número de Montecarlo y el tiempo de ejecución.

Como hasta ahora se ha mostrado en las ejecuciones anteriores del QAOA el número de qubits influye de manera aproximadamente exponencial. Por otro lado, el número de iteraciones de Montecarlo, como se ha podido observar no solo en el QAOA, aumenta linealmente el tiempo de cómputo.

Si bien en el potencial anterior la relación que se observaba para el parámetro gamma era con respecto a la precisión de los resultados, en este caso la mayor influencia la tiene con el primer mínimo. Esto puede deberse a que, como ya se ha mencionado, este potencial cuenta con un mínimo local y uno global y del primer mínimo se espera que sea capaz de encontrar el mínimo global y el segundo mínimo encuentre el local. Debido a esto, un correcto ajuste de este

parámetro significará un primer mínimo más cerca del $1{,}1$ y no del $-1{,}06$ como ocurría para el potencial de Higgs, donde era indistinto cuál se definía como primer mínimo.

Para poder visualizar mejor estos resultados se ha optado por el gráfico de violín (*violín plot*), de esta manera la anchura del gráfico refleja la densidad con la que se encuentra el mínimo en ese valor. Así, se puede observar cómo existen valores de gamma que localizan el valor en $1{,}1$ de manera regular, y cómo estos son los mismos que localizan el segundo mínimo entorno a $-1{,}06$ con más frecuencia.

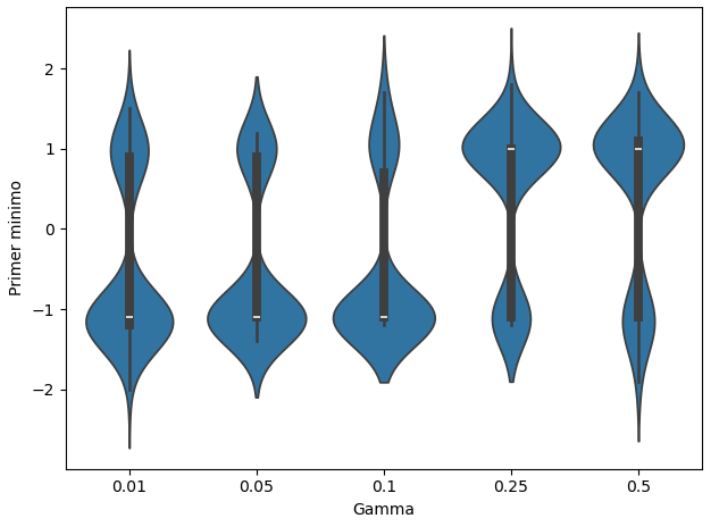


Figura 43: Relación entre gamma y primer mínimo

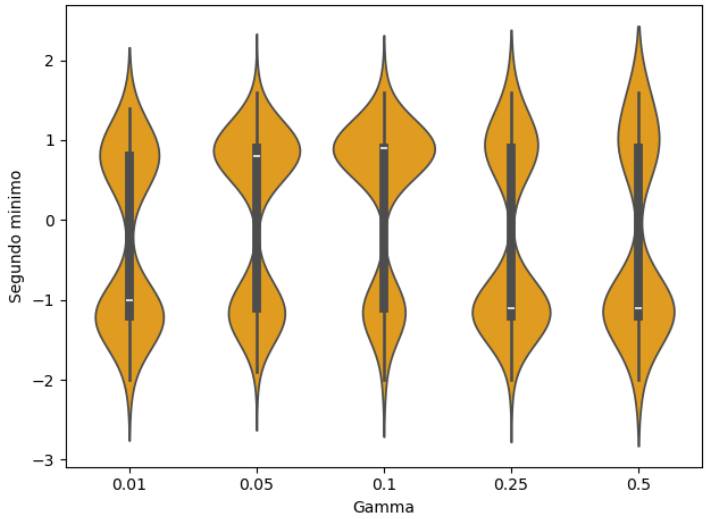


Figura 44: Relación entre el gamma y el segundo mínimo

Tal y como se esperaba, el valor de gamma que mejor y más frecuentemente identifica el primer mínimo también es aquel que identifica con mayor claridad del segundo mínimo.

También se garantiza que los valores reproducidos con esta gamma son los ideales al ver que presentan muy poca dispersión, lo que le hace poder reproducir soluciones de forma muy consistente.

Se toman los valores que generan la mejor solución, que en este caso resulta en $\min_1 = 1{,}1$ y $\min_2 = -1{,}1$. Para estas condiciones existen 28 combinaciones de parámetros capaces de replicar esta solución. Estos resultados son combinaciones de parámetros muy diversos, es por eso, que para poder determinar qué solución presenta el mejor resultado se va a tomar de cada parámetro, aquel que más se repite

	N_qubits	Beta	Gamma	Prof_p	N_mc
Valor	3	0.1	0.5	3	100
Veces que aparece	13	16	13	13	20

Cuadro 14: Mejores parámetros para el QAOA

Se puede ver con más detalle la solución que reproduce el mejor parámetro encontrado si se ejecuta desde la interfaz. A través de la interfaz se obtiene la gráfica de todos los valores del QAOA para cada iteración del Montecarlo, así como los valores de los mínimos con más precisión.

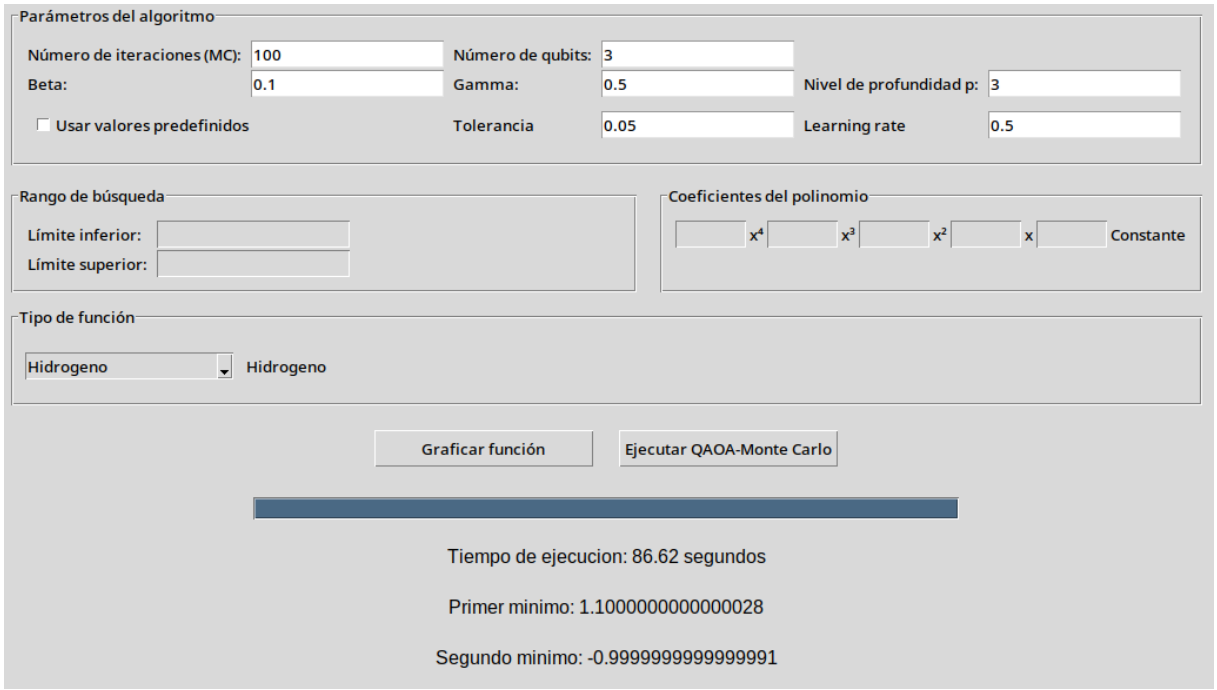


Figura 45: Interfaz para el QAOA

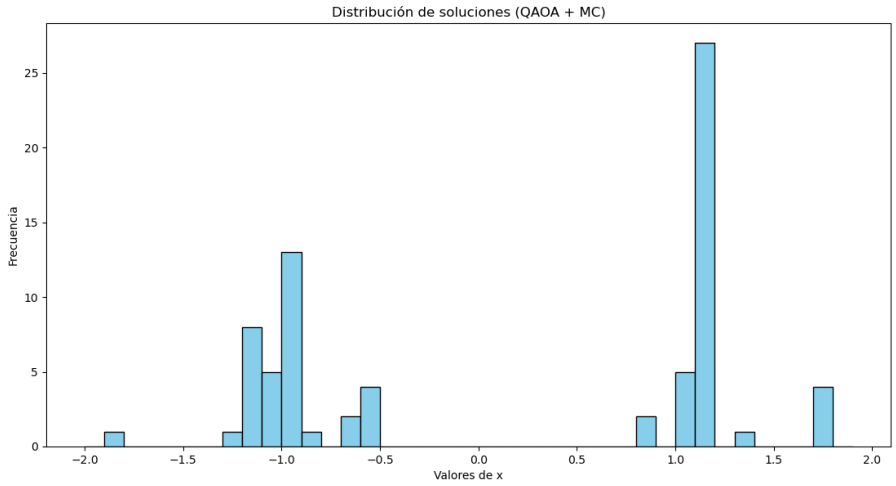


Figura 46: Resultados para los mejores parámetros

Tal y como se esperaba de estos parámetros, reproducen una solución que coincide con una precisión aceptable con lo que se podría esperar de esta configuración. Además, el algoritmo distingue perfectamente entre el mínimo global y el mínimo local, ya que existe una amplia diferencia entre el número de muestras para cada medida. Con todo esto, se puede afirmar que el QAOA es capaz de localizar este mínimo de manera precisa.

5.3.2. Gradient Descent

Al realizar las medidas para el gradient descent y obtener los ficheros csv correspondientes se realiza la matriz de correlación con el objetivo de poder tener una primera impresión de cómo están relacionados los parámetros con los resultados.

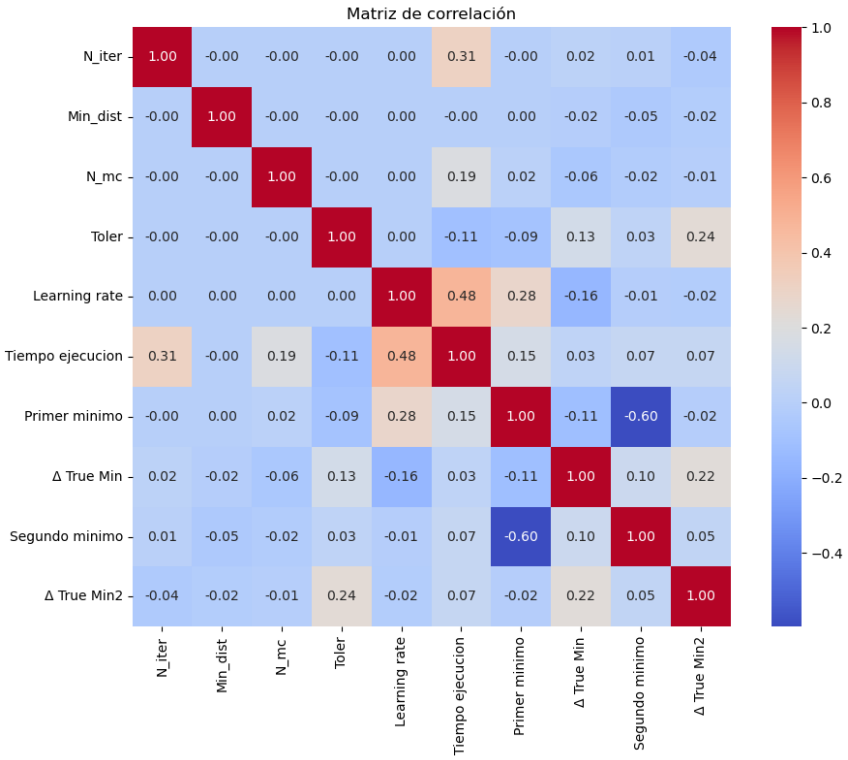


Figura 47: Matriz de correlación del gradient descent

De manera similar al caso del potencial de Higgs los parámetros responsables del tiempo de ejecución son el número de iteraciones, el número de iteraciones de Montecarlo, la tolerancia y el learning rate. Aunque sí que es cierto que en dicho potencial los parámetros que más impacto tenían eran la tolerancia y el learning rate, mientras que en este caso lo son el número de iteraciones y el tiempo de ejecución.

Como se hacía con el QAOA, se deben de gráficar la tolerancia y el learning rate con los valores de los mínimos y no con la precisión de los resultados. Así, se puede ver con mayor claridad cuándo el algoritmo es capaz de ubicar correctamente los dos mínimos.

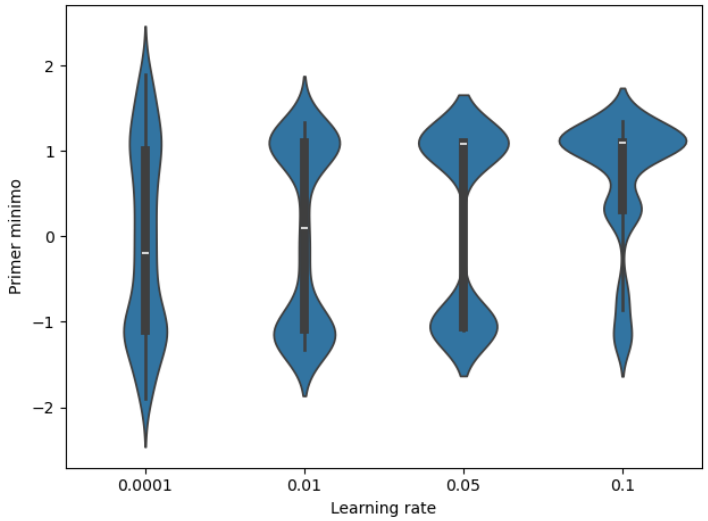


Figura 48: Relación entre el learning rate y el valor del primer mínimo

Observando ambas gráficas se puede deducir que el valor óptimo para el learning rate debe de encontrarse entorno al $0{,}05$ y $0{,}1$. Pues para estos valores el algoritmo localiza ambos mínimos donde debería con más consistencia.

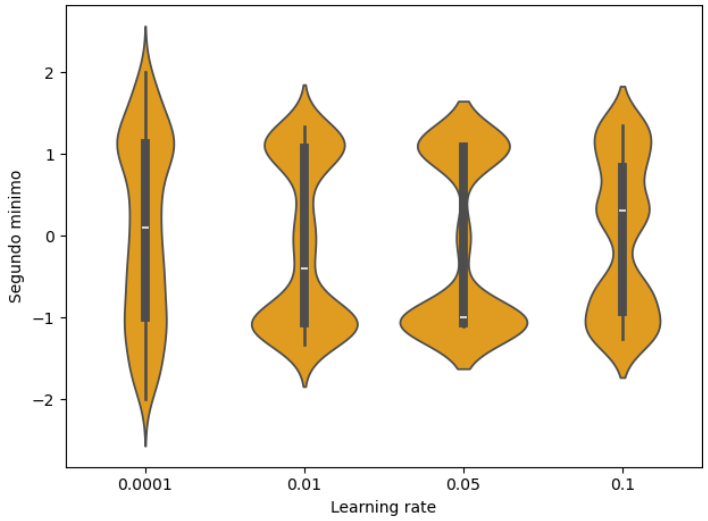


Figura 49: Relación entre el learning rate y el valor del segundo mínimo

Respecto a la tolerancia, como se cabría esperar, los resultados más precisos son para las tolerancias más bajas.

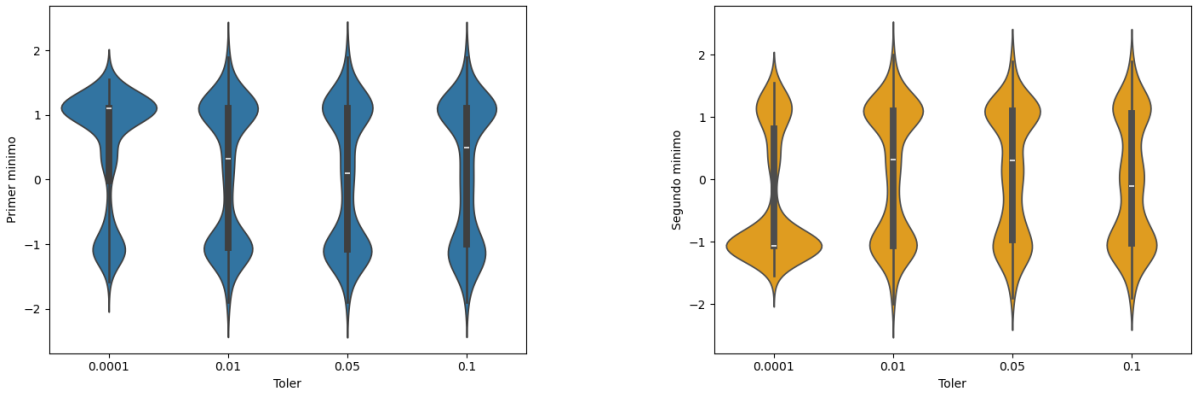


Figura 50: Relación la tolerancia y el valor de los mínimos

5.3.3. Método de Newton

De la misma manera que hasta ahora, se vuelve a reproducir el mismo método, para obtener 3 csv, un cuarto resultado de la combinación de estos. Y, una matriz de correlación donde estudiar el impacto de los parámetros sobre los resultados.

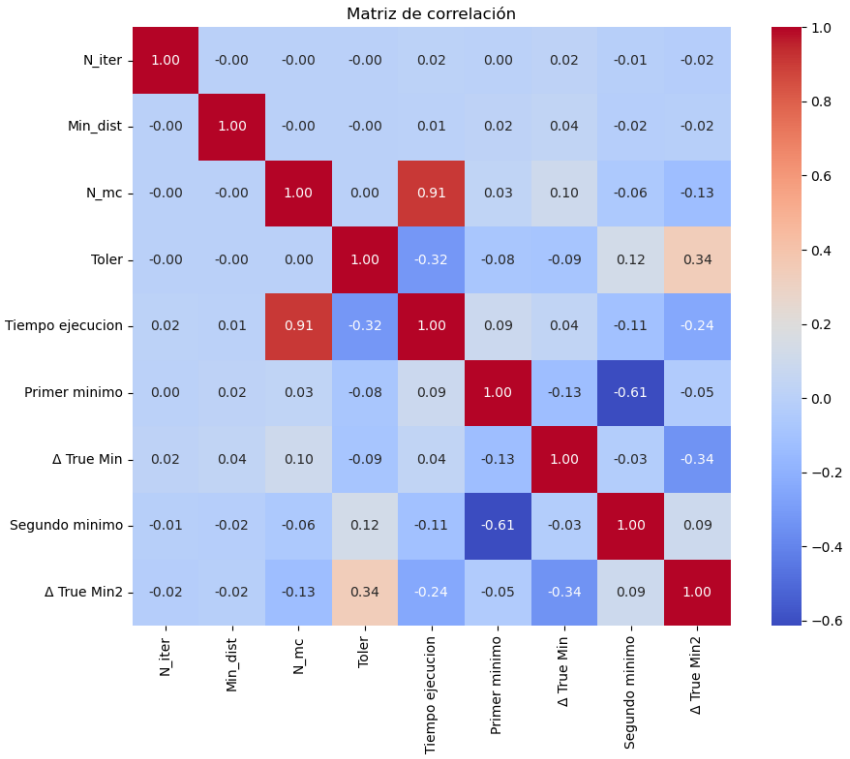


Figura 51: Matriz de correlación para el método de Newton

Como se esperaba, esta matriz de correlación es muy similar a la obtenida para el potencial de Higgs, así que el tratamiento de los datos sera muy similar, así como las conclusiones

que se esperan obtener.

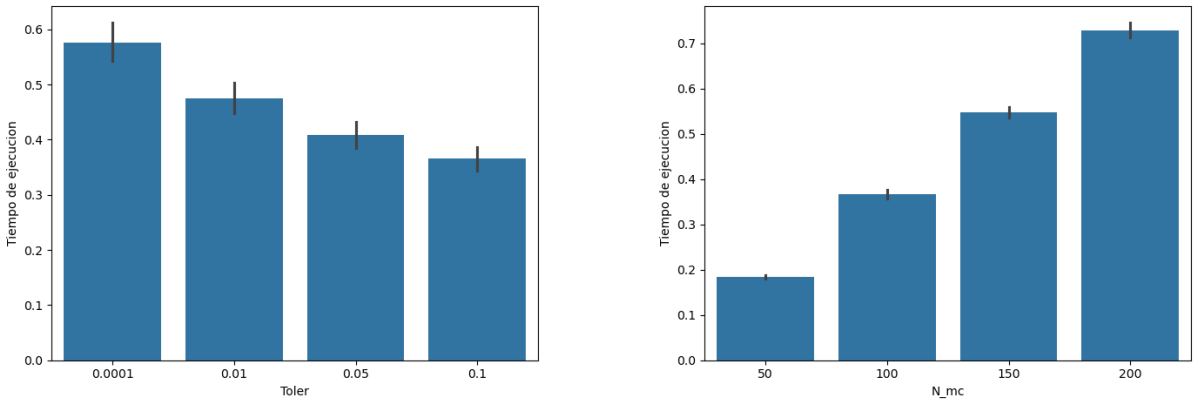


Figura 52: Relación entre la tolerancia y el número de iteraciones de Montecarlo y el tiempo de ejecución.

La tolerancia y el número de iteraciones de Montecarlo continúan siendo los parámetros que más influyen en el tiempo de cómputo. Una diferencia notable es que en este caso la tolerancia es considerablemente más influyente en la precisión del segundo mínimo que para el potencial anterior. Esto puede deberse a que este segundo mínimo es un mínimo local y esto requiera de una tolerancia más específica para poder determinar con precisión resultados. De hecho, ya ocurría en el potencial anterior que en ocasiones este algoritmo ubicaba los mínimos en el máximo que se formaba en $x = 0$ para el potencial de Higgs. Esto puede verificarse en este caso, volviendo a enfrentar en una gráfica la tolerancia con los valores de los mínimos. Con esto se obtiene:

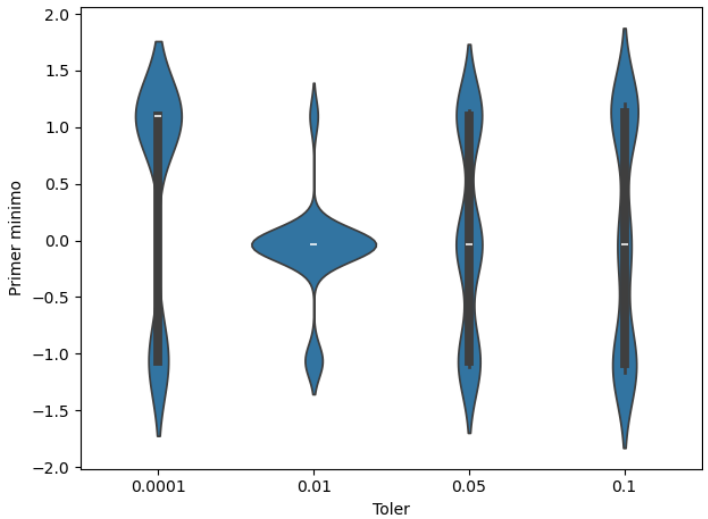


Figura 53: Relación entre la tolerancia y el primer mínimo

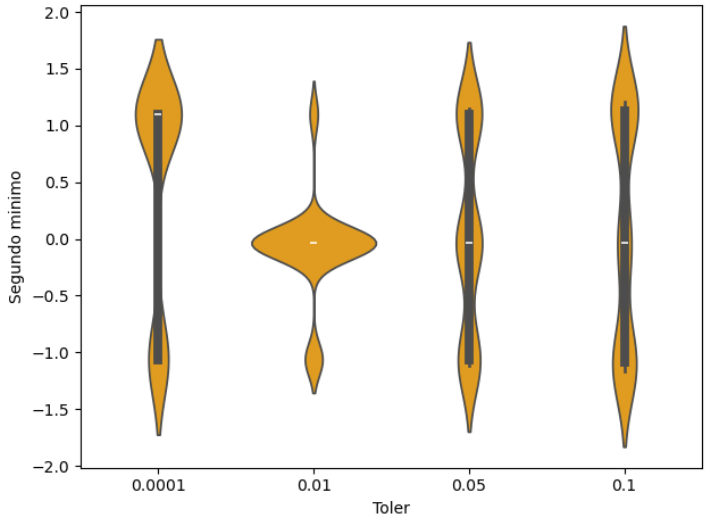


Figura 54: Relación entre la tolerancia y el segundo mínimo

Se puede observar cómo el mismo fenómeno que se producía para el potencial anterior en el método de Newton vuelve a aparecer. Esta vez en el valor de la tolerancia $0{,}01$, para la cual el algoritmo ubica con una alta frecuencia el mínimo donde se encuentra el máximo. De la misma manera que antes, esto puede ser porque el algoritmo busca lugares donde la pendiente sea 0, como lo son los máximos y los mínimos. En cualquier caso, cuando la tolerancia es menor se puede ver cómo el algoritmo localiza con mayor frecuencia el mínimo entorno al 1.

6. Conclusión

Durante el desarrollo de este trabajo se ha analizado el comportamiento del algoritmo QAOA frente a distintos potenciales, evaluando su rendimiento y sensibilidad en comparación con métodos clásicos como el gradient descent y el método de Newton. Analizando los resultados y estudiando las relaciones con los parámetros.

El primer potencial planteó grandes dificultades para los tres algoritmos, siendo el método de Newton el que presentó mayores desviaciones respecto al mínimo real. Este potencial representaba un verdadero desafío, principalmente debido a la sutileza del mínimo en el potencial de Lennard-Jones y la fuerte dependencia de ciertos métodos con la pendiente de la función. En este sentido, el QAOA muestra una clara fortaleza. A diferencia de los métodos clásicos, cuya eficacia depende en gran medida de la primera o segunda derivada, el QAOA puede minimizar la función objetivo sin requerir dicha información. Esta característica, junto con los resultados observados, sugiere que el QAOA —aunque probablemente requiera un ajuste más exhaustivo de parámetros y un mayor número de qubits— podría ofrecer un rendimiento superior, especialmente en potenciales de forma similar pero con mínimos más marcados, con

forma de ”pico”.

En el caso del potencial de Higgs, el QAOA ofrece resultados al menos, tan buenos como el gradient descent y el método de Newton. Se ha mostrado preciso y consistente en la localización de los mínimos, y menos exigente en cuanto al ajuste de parámetros. De hecho, en algunos escenarios ha demostrado mayor robustez que sus contrapartes clásicas, que ocasionalmente convergen hacia máximos en lugar de mínimos, debido a su dependencia con la derivada. Esta independencia representa una ventaja significativa para el QAOA.

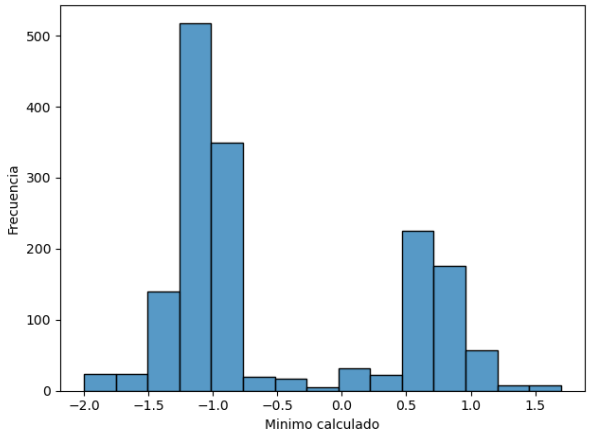
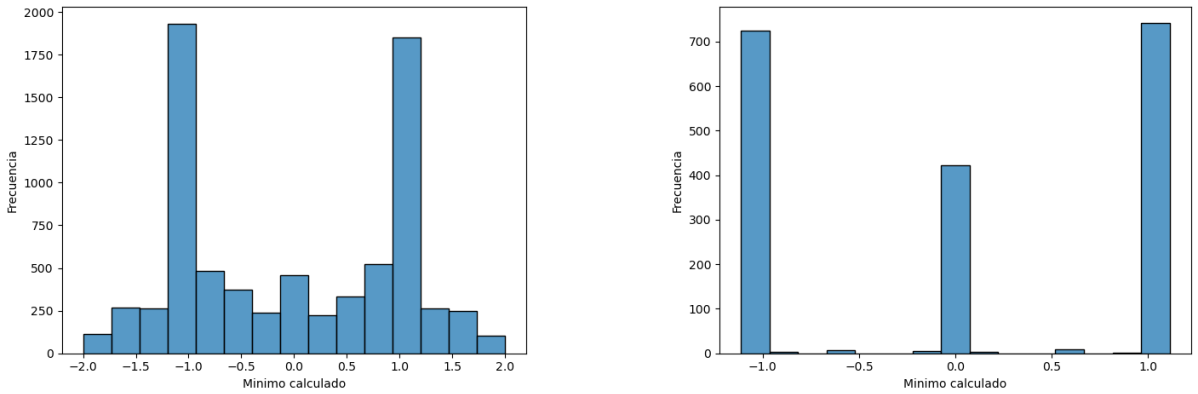


Figura 55: Resultados para el QAOA en el potencial de Higgs



(a) Gradient descent

(b) Metodo de Newton

Figura 56: Resultados clásicos para el potencial de Higgs

Al comparar los resultados obtenidos por los distintos algoritmos se hace evidente este fenómeno. Por un lado, el QAOA presenta dos mínimos bien definidos y localizados. Por otro lado, el gradient descent también identifica estos dos mínimos, aunque de manera algo más dispersa, y en ocasiones converge erróneamente hacia un máximo. De forma similar, aunque con mayor precisión, el método de Newton muestra este comportamiento: ofrece una localiza-

ción más exacta de los mínimos, pero también presenta una mayor tendencia a confundirse y posicionarse en un máximo en lugar del mínimo esperado.

Algo comparable se observa en el caso del potencial del hidrógeno. Este potencial cuenta con dos mínimos: uno local y otro global. Al igual que en el caso anterior, el QAOA no presenta dificultades significativas al localizar estos mínimos. Esta independencia respecto a la pendiente vuelve a demostrar la robustez del QAOA frente a los métodos clásicos, los cuales pueden verse afectados negativamente por la estructura del gradiente.

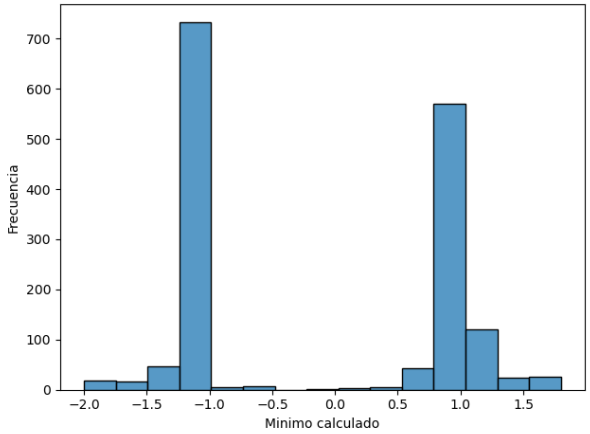
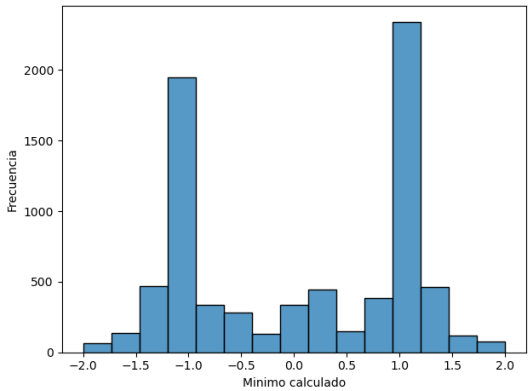
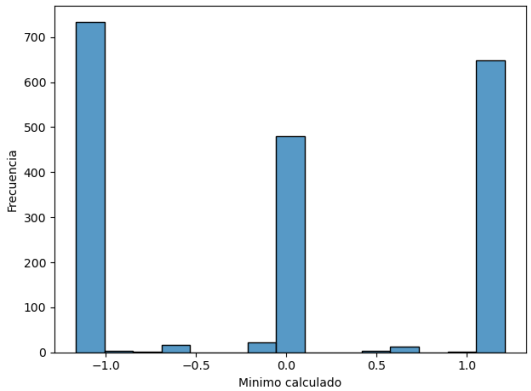


Figura 57: Resultados para el QAOA en el potencial de Hidrógeno



(a) Gradient descent



(b) Metodo de Newton

Figura 58: Resultados clásicos para el potencial de Hidrógeno

Incluso en este caso, el QAOA muestra una menor dispersión en los resultados en comparación con el caso anterior. En contraste, tanto el descenso por gradiente como el método de Newton mantienen un nivel de dispersión relativamente constante en sus soluciones.

En cuanto al tiempo de ejecución, el QAOA se encuentra considerablemente por detrás de los algoritmos clásicos. Tal como se puede observar en las gráficas, ciertas configuraciones del QAOA provocan un aumento significativo en el tiempo de cómputo. En menor medida,

esto también ocurre ocasionalmente con el descenso por gradiente. Por otro lado, el método de Newton presenta un tiempo de ejecución prácticamente inmediato.

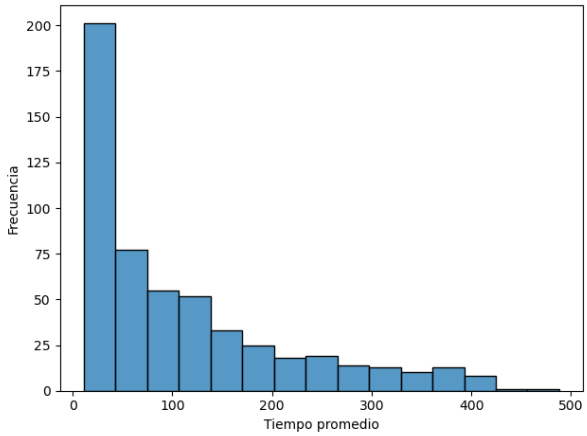
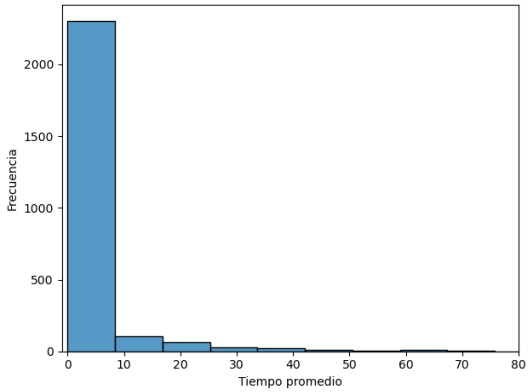
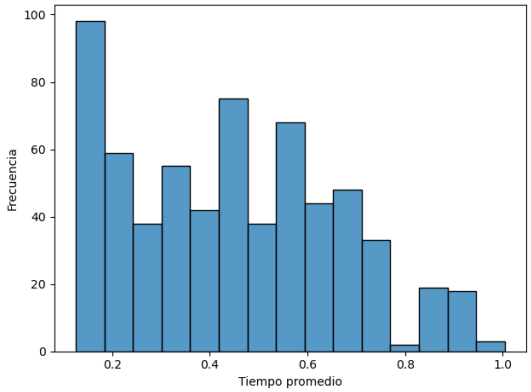


Figura 59: Resultados para el QAOA en el tiempo



(a) Gradient descent



(b) Metodo de Newton

Figura 60: Resultados clásicos para el tiempo

Estos resultados eran esperables, ya que no debe olvidarse que el QAOA está siendo ejecutado en un ordenador clásico mediante un simulador, el cual reproduce las condiciones, el comportamiento y los resultados esperados de un ordenador cuántico.

En relación con los parámetros, se ha podido observar la sensibilidad de cada algoritmo frente a su ajuste, dependiendo del tipo de potencial considerado. En el caso del gradient descent, se obtienen mejores resultados con tolerancias bajas y un valor del learning rate en torno a 0.05, de forma general. De manera similar, el método de Newton también muestra mejor rendimiento con tolerancias estrictas. Al no depender de un parámetro de learning rate, la tolerancia adquiere un papel aún más relevante en este algoritmo.

Por otro lado, el QAOA requiere ajustar un mayor número de parámetros, muchos de los cuales tienen un impacto considerable en la calidad de los resultados. Algunos de estos,

como el número de qubits o el número de iteraciones Montecarlo, afectan de forma positiva a la precisión, aunque incrementan significativamente el tiempo de cómputo, especialmente número de qubits. Además, los parámetros β y γ resultan fundamentales. En general, se ha observado que el algoritmo ofrece mejores resultados con valores de β en torno a $0{,}1$, y valores de γ entre $0{,}25$ y $0{,}5$.

En conclusión, el algoritmo QAOA representa una propuesta prometedora dentro del campo de la computación cuántica, con el potencial de abordar eficientemente ciertos problemas que presentan dificultades para los algoritmos clásicos, como el descenso por gradiente o el método de Newton. Sin embargo, su rendimiento y aplicabilidad están estrechamente vinculados al desarrollo de hardware cuántico lo suficientemente avanzado como para aprovechar al máximo sus ventajas teóricas. A nivel personal, auguro un futuro prometedor para la computación cuántica, especialmente en ámbitos como la supercomputación y la simulación a gran escala, donde podría marcar una diferencia significativa.

7. Bibliografía

Feynman, R. P. (1982). Simulating Physics with Computers. *International Journal of Theoretical Physics*, *21*(6), 467-488.

Dirac, P. A. M. (1928). The Quantum Theory of the Electron. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, *117*(778), 610-624. https://doi.org/10.1098/rspa.1928.0023

Carcassi, G., Maccone, L., & Aidala, C. A. (2021). Four Postulates of Quantum Mechanics Are Three. *Physical Review Letters*, *126*(11). https://doi.org/10.1103/physrevlett.126.110402

Principios o postulados de la mecánica cuántica [Accedido el 13 de mayo de 2025]. (s.f.).

Griffiths, D. J., & Schroeter, D. F. (2018). *Introduction to Quantum Mechanics* (3rd). Cambridge University Press.

Einstein, A., Podolsky, B., & Rosen, N. (1935). Can Quantum-Mechanical Description of Physical Reality Be Considered Complete? *Physical Review*, *47*(10), 777-780.

Nielsen, M. A., & Chuang, I. L. (2010). Quantum Computation and Quantum Information. Cambridge University Press.

Alonso-Linaje, G. (2024). Key.G [Consultado el 3 de febrero de 2025].

Farhi, E., Goldstone, J., & Gutmann, S. (2014). A Quantum Approximate Optimization Algorithm. *arXiv preprint arXiv:1411.4028*. https://arxiv.org/abs/1411.4028

Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems* (2nd). O'Reilly Media.

IBM. (2025). Solve utility-scale quantum optimization problems [Consultado el 20 de abril de 2025].

Luenberger, D. G., & Ye, Y. (2015). *Introduction to Optimization* (4th). Springer.

Atkins, P., & de Paula, J. (2010). Physical Chemistry. W. H. Freeman; Company.

8. Anexos

Anexo A: QAOA

```
import numpy as np
from qiskit import QuantumCircuit
from qiskit_aer import QasmSimulator
from qiskit.circuit import Parameter
from qiskit.visualization import plot_histogram
from qiskit.quantum_info import Statevector
from scipy.optimize import minimize
import matplotlib.pyplot as plt

import pandas as pd
import time

import itertools
from sympy import symbols, Eq, solve
```

Listing 1: Importar librerias

```
def qubit_states_generator(n_qubits):
        qubit_states = []
        if n_qubits < 6:
                for comb in itertools.product('01', repeat=n_qubits):
                        qubit_states.append(''.join(comb))

        return qubit_states
```

Listing 2: Funcion generadora de los estados de los qubits

```
def Z_values_generator(qubit_states, n_qubits):
        Z_values = {}
        for i in range(len(qubit_states)):
                string = qubit_states[i]
                Z = []
                #parte no combinada de los terminos de Z
                for j in range(len(string)):
                        if string[j] == '0':
                                Z.append(+1)
                        else:
                                Z.append(-1)
                Z = list(reversed(Z)) #invierto la lista, recordar que los
                    qubits se leen de derecha a izquierda

        ######### parte combinada de los terminos de mas a menos #########
```

```
15
16                #creacion del indice para los terminos cruzados
17                #Esta parte crea la variable combination, que ira sirviendo
                      de indice para crear los terminados cruzados
18                combination = '01'
19                for j in range(n_qubits-2):
20                        combination += str(j+2)
21                        #Para 3 qubits devolera 012, para 4 0123, para 5
                             01234 etc.
22
23                for i in range(n_qubits-1):
24                #Este bucle es para tener todas las combinaciones  de
                     longitud 2 hasta longitud n_qubits
25
26                        for j in itertools.combinations(combination,i+2):
27                                j = list(j) #pasamos a una lista los
                                     valores de la combinacion generada por
                                     itertools
28                                New_Z = 1 #Se crea el valor que vamos a a\~
                                     nadir
29                                #Ej: si tenemos ('012',2) nos devolvera
                                     ['0','1'], ['0','2'] y ['1','2']
30
31                                for k in range(len(j)):
32                                        #Para determinar el valor de ese
                                             nuevo resultado, se multiplican
                                             los valores que lo forman
33                                        New_Z *= Z[int(j[k])]
34
35                                Z.append(New_Z)
36                                #finalmente se añade el nuevo resultado
37
38                Z_values[string] = Z
39
40        return Z_values
```

Listing 3: Funcion generadora de los valores de Z

```
1  def solver(Z_values, qubit_states, x_vals, f):
2          symbols_list = []
3          equations = []
4
5          for i in range(len(Z_values[qubit_states[0]])+1):
6                  symbols_list.append(symbols(f"a{i}"))
7                  # creo una lista de todos los strings que van a ser las
                       variables
```

```
8
9           for i in range(len(symbols_list)):  # i es igual a la cantidad de
                incognitas
10              Ecuacion = symbols_list[0]  # El primer simbolo a0 va con
                    la identidad
11              for j in range(len(symbols_list)-1):
12                      Ecuacion += symbols_list[j+1]*Z_values[qubit_states
                            [i]][j]
13                      # añado el nuevo termino a la ecuacion
14
15              # eq representa la ecuacion de sympy con su valor f(x)
16              eq = Eq(Ecuacion, f(x_vals[i]))
17              equations.append(eq)  # añado la ecuacion nueva
18
19          # esta parte transforma el diccionario solucion en una lista de
                floats
20          solution_dic = solve(equations, symbols_list)
21          solution = []
22          for k in range(len(solution_dic)):
23                  solution.append(float(solution_dic[symbols_list[k]]))
24
25          return solution
```

Listing 4: Función para construir y resolver el sistema de ecuaciones

```
1   def Z_lengths_iterations(n):
2           Z_lenghts = []
3           Z_lenghts.append(n)
4
5           combination = '01'
6           for j in range(n-2):
7                   combination += str(j+2)
8
9           for i in range(n-1):
10                  Z_lenghts.append(len(list(itertools.combinations(
                        combination,i+2))))
11
12          return Z_lenghts
```

Listing 5: Funcion calculadora del numero de iteraciones de Z

```
1   def QAOA_circuit(n_qubits, solution, gamma, beta, p=1):
2           qc = QuantumCircuit(n_qubits)
3           qc.h(range(n_qubits))
4           #hadammard en todos
5
```

```
6         Z_len = Z_lengths_iterations(n_qubits)
7
8         for _ in range(p):
9                 for i in range(Z_len[0]):
10                        qc.rz(2 * gamma * solution[i+1],i)
11                qc.barrier()
12                #aplico las puertas Z a los qubits correspondientes
13
14                #esta parte es simplemente para tener combination como lo
                     tenia en la parte del Z_values_generator
15                combination = '01'
16                for j in range(n_qubits-2):
17                        combination += str(j+2)
18
19                posicion_aux = n_qubits+1 #esta va a ser una variable
                     auxiliar, simplemente para poder referirnos a los
                     valores de solucion correctamente
20                for i in range(n_qubits-1): # determino las combinaciones
                     que tengo para esos qubits.
21                        lista = list(itertools.combinations(combination,i
                             +2)) #listo las combinaciones de i puertas Z
22
23                        #hacemos la parte de los terminos de varios Z de
                             manera general:
24                        for j in range(len(lista)):
25                                for k in range(len(lista[0])-1):
26                                        qc.cx(int(lista[j][k]), int(lista[j
                                             ][k+1]))
27
28                                qc.rz(2 * gamma * solution[posicion_aux],
                                     int(lista[j][k+1]))
29                                posicion_aux += 1
30                                for k in range(len(lista[0])-1):
31                                        k = len(lista[0])-2 -k # es
                                             necesario este cambio para que
                                             las rotaciones se deshagan en el
                                              orden correcto
32                                        qc.cx(int(lista[j][k]), int(lista[j
                                             ][k+1]))
33
34                                qc.barrier()
35
36                qc.barrier()
37
38                for q in range(n_qubits):
```

```
                        qc.rx(2 * beta, q)

        qc.measure_all()
        return qc
```

Listing 6: Funcion para construir el circuito del QAOA en qiskit

```
def QAOA(qubit_states, Z_values, n_qubits, x_vals, beta, gamma, p, f):
        solution = solver(Z_values, qubit_states, x_vals, f)
        #calculo la solucion del hamiltoniano y la guardo en una lista

        qc_QAOA = QAOA_circuit(n_qubits, solution,gamma, beta, p)

        backend = QasmSimulator()
        job = backend.run(qc_QAOA, shots=2048)
        counts = job.result().get_counts()

        prob_state = max(counts, key=counts.get)
        prob_val_position = qubit_states.index(prob_state)
        prob_val = x_vals[prob_val_position]

        return(prob_val)
```

Listing 7: Funcion que ejecuta el QAOA

```
def QAOA_MC(n_MC, inf_range, sup_range, qubit_states, Z_values, n_qubits,
   beta, gamma, p, tolerance, learning_rate, f):
        soluciones = []

        for m in range(n_MC):
                #genero una soluciona a partir de un valor aleatorio
                rand_val = round(np.random.uniform(inf_range,sup_range), 1)
                x_vals = [rand_val]


                #Creo la lista con los valores iniciales
                for a in range(int((2**n_qubits)/2)):
                        x_vals.insert(0, round(rand_val - (0.05*(a+1)),2))
                        x_vals.append(round(rand_val +(0.05*(a+1)),2))
                x_vals.pop(2**n_qubits)

                QAOA_solution = QAOA(qubit_states, Z_values, n_qubits,
                   x_vals, beta, gamma, p, f)

                #genero otra solucion de entre unos valores, cuyo rango
                   esta centrado en la solucion anterior
```

```
                x_vals = [QAOA_solution]
                for a in range(int((2**n_qubits)/2)):
                        x_vals.insert(0, round(QAOA_solution - (0.05*(a+1))
                            ,2))
                        x_vals.append(round(QAOA_solution +(0.05*(a+1)),2))
                x_vals.pop(2**n_qubits)
                QAOA_solution2 = QAOA(qubit_states, Z_values, n_qubits,
                    x_vals, beta, gamma, p, f)

                n_points = n_qubits**2
                center_index = n_points // 2  # indice donde estara
                    QAOA_solution

                for i in range(30):
                        if abs(QAOA_solution - QAOA_solution2) < tolerance:
                                soluciones.append(QAOA_solution2)
                                break

                        QAOA_solution = QAOA_solution2
                        mu = learning_rate / (1+ (i))
                        x_vals = [
                                round(QAOA_solution + mu*(j-center_index),
                                    2)
                                for j in range(n_points)
                        ]
                        QAOA_solution2 = QAOA(qubit_states, Z_values,
                            n_qubits, x_vals, beta, gamma, p, f)

        return soluciones
```

Listing 8: Funcion que añade MC al QAOA

```
def generar_combinaciones(diccionario):
        claves = diccionario.keys()
        valores = diccionario.values()
        combinaciones = list(itertools.product(*valores))
        return [dict(zip(claves, c)) for c in combinaciones]
```

Listing 9: Funcion para generar las combinaciones de los parametros

```
if __name__ == "__main__":
        T4, T3, T2, T1, T0 = 0.0, 0.0, 0.0, 0.0, 0.0
        input("Comprueba el nombre del archivo csv que se va a guardar")

    QAOA_parameters = {
                "N_qubits": [2, 3, 4],
```

```
                "beta": [0.5, 0.3, 0.1],
                "gamma": [0.5, 0.25, 0.1, 0.05, 0.01],
                "Prof_p": [2, 3, 4],
                "N_mc": [50, 100],
                "Toler": [0.05],
                "Learning rate": [0.5],
        }

        opcion = input('1.Lennard-Jones, 2.Higgs, 3.Hidrogeno: ')

        x = sp.symbols('x')

        if opcion == "1":
                true_min = 1.1225
                def f(x):
                        if x == 0:
                                x = 0.01
                        y = 4 * 10 * ((0.25/x)**12 - (0.25/x)**6)
                        return -y

                function = y = 4 * 10 * ((0.25/x)**12 - (0.25/x)**6)
                inf_range = 0.8
                sup_range = 3

        elif opcion == "2":
                true_min = -1
                true_min2 = 1
                def f(x):
                        y = -1 * x**2 + 0.5 * x**4
                        return -y

                function = -1 * x**2 + 0.5 * x**4
                inf_range = -2
                sup_range = 2

        elif opcion == "3":
                true_min = 1.1
                true_min2 = -1.06
                def f(x):
                        y = 3*(x**4) - 7*(x**2) - 0.5*x + 6
                        return -y

                function = 3*(x**4) - 7*(x**2) - 0.5*x + 6
                inf_range = -2
                sup_range = 2
```

```
52
53          else:
54                  print(f'Selecciona una opcion valida')
55
56
57      combinaciones = generar_combinaciones(QAOA_parameters)
58
59          contador = 0
60          resultados = []
61          for params in combinaciones:
62
63          contador += 1
64                  print(f'===============Iteracion {contador} de {len(
                        combinaciones)}==================')
65
66                  n_qubits = params["N_qubits"]
67                  beta = params["beta"]
68                  gamma = params["gamma"]
69                  p = params["Prof_p"]
70                  n_MC = params["N_mc"]
71                  toler = params["Toler"]
72                  learn = params["Learning rate"]
73
74
75                  print(f"Parametros a introducir: n_MC {n_MC}, n_qubits {
                        n_qubits}, beta {beta}, gamma {gamma}, p {p}, toler {
                        toler}, learn {learn}")
76
77                  qubit_states = qubit_states_generator(n_qubits)
78                  Z_values = Z_values_generator(qubit_states, n_qubits)
79
80
81                  soluciones_lista = []
82                  start_time = time.time()
83
84                  soluciones_lista = QAOA_MC(n_MC, inf_range, sup_range,
                        qubit_states, Z_values, n_qubits, beta, gamma, p, toler,
                         learn, f)
85
86                  elapsed_time = time.time() - start_time
87
88                  x = np.arange(inf_range, sup_range, 0.1)
89
90                  conteos, bordes = np.histogram(soluciones_lista, bins=x)
91                  conteos = list(conteos)
```

```
92
93                  max_conteos = conteos.index(max(conteos))
94                  primer_min = bordes[max_conteos]
95
96                  if opcion == '1':
97                          resultado = params.copy()
98                          resultado["Tiempo ejecucion"] = round(elapsed_time,
                                4)
99                          resultado["Primer minimo"] = round(primer_min, 4)
100                         resultado["Δ True Min"] = abs(true_min - primer_min
                                )
101
102                         print(f'Minimo es {primer_min} y el minimo real {
                                true_min}')
103
104                 elif opcion == '2':
105                         resultado = params.copy()
106                         resultado["Tiempo ejecucion"] = round(elapsed_time,
                                4)
107                         resultado["Primer minimo"] = round(primer_min, 4)
108                         resultado["Δ True Min"] = min(abs(true_min -
                                primer_min), abs(true_min2 - primer_min))
109
110                         print(f'Minimo es {primer_min} y el minimo real ±1'
                                )
111
112                         conteos.pop(max_conteos)
113                         max_conteos2 = conteos.index(max(conteos))
114                         if abs(primer_min - bordes[max_conteos2]) < 0.25:
115                                 conteos.pop(max_conteos2)
116                                 max_conteos2 = conteos.index(max(conteos))
117
118                         segundo_min = bordes[max_conteos2]
119
120                         resultado["Segundo minimo"] = segundo_min
121                         resultado["Δ True Min2"] = min(abs(true_min -
                                segundo_min), abs(true_min2 - segundo_min))
122
123                         print(f'Segundo minimo es {segundo_min} y el minimo
                                real ±1')
124
125                 elif opcion == '3':
126                         resultado = params.copy()
127                         resultado["Tiempo ejecucion"] = round(elapsed_time,
                                4)
```

```
                        resultado["Primer minimo"] = round(primer_min, 4)
                        resultado["Δ True Min"] = min(abs(true_min -
                            primer_min), abs(true_min2 - primer_min))

                        print(f'Minimo es {primer_min} y el minimo real {
                            true_min}')

                        conteos.pop(max_conteos)
                        max_conteos2 = conteos.index(max(conteos))
                        if abs(primer_min - bordes[max_conteos2]) < 0.25:
                                conteos.pop(max_conteos2)
                                max_conteos2 = conteos.index(max(conteos))

                        segundo_min = bordes[max_conteos2]

                        resultado["Segundo minimo"] = segundo_min
                        resultado["Δ True Min2"] = min(abs(true_min -
                            segundo_min), abs(true_min2 - segundo_min))

                        print(f'Segundo minimo es {segundo_min} y el minimo
                             real {true_min2}')


                print(f'El resultado es {resultado}')

                resultados.append(resultado)

        if resultados:
                df = pd.DataFrame(resultados)
                nombre_archivo = "QAOA_NombrePotencial.csv"

                df.to_csv(nombre_archivo, index=False)
                print(f" Resultados guardados en: {nombre_archivo}")
        else:
                print("No se generaron resultados para guardar.")
```

Listing 10: Ejecucion del codigo para multiples variables

Anexo B: Gradient Descent

```
import numpy as np
import autograd as ag
import matplotlib.pyplot as plt
import time
```

```
from collections import Counter

import itertools
import pandas as pd

import sympy as sp
```

Listing 11: Importar librerias

```
def ini_vec(inf_range, sup_range):
        return round(np.random.uniform(inf_range,sup_range), 1)
```

Listing 12: Funcion para generar el vector inicial

```
def gradient(v, function):
        grad_f = ag.grad(function)
        return (grad_f(np.array(v)))
```

Listing 13: Funcion para calcular el gradiente

```
def gradient_descent(inf_range, sup_range, n_iter, learning_rate, tolerance
    , function):
        v = ini_vec(inf_range,sup_range)
        for i in range(n_iter):
                g = gradient(v, function)
                v_new = v - learning_rate*g

                if abs(v_new - v) < tolerance:
                        break

                v = v_new

        return v
```

Listing 14: Funcion del gradient descent

```
def generar_combinaciones(diccionario):
        claves = diccionario.keys()
        valores = diccionario.values()
        combinaciones = list(itertools.product(*valores))
        return [dict(zip(claves, c)) for c in combinaciones]
```

Listing 15: Funcion para generar las combinaciones de los parametros

```
if __name__ == "__main__":
        T4, T3, T2, T1, T0 = 0.0, 0.0, 0.0, 0.0, 0.0
        input("Comprueba el nombre del archivo csv que se va a guardar")
```

```
4
5       Gradient_parameters = {
6               "N_iter": [100, 500, 1000, 2000],
7               "Min_dist": [0.1, 0.2, 0.25, 0.5, 1],
8               "N_mc": [50, 100, 150, 200],
9               "Toler": [0.1, 0.05, 0.01, 0.0001],
10              "Learning rate": [0.1, 0.05, 0.01, 0.0001],
11      }
12
13      opcion = input('1.Lennard-Jones, 2.Higgs, 3.Hidrogeno: ')
14
15
16      if opcion == "1":
17              Gradient_parameters = {
18                      "N_iter": [100, 500, 1000, 2000],
19                      "N_mc": [50, 100, 150, 200],
20                      "Toler": [0.1, 0.05, 0.01, 0.0001],
21                      "Learning rate": [0.1, 0.05, 0.01, 0.0001],
22              }
23              true_min = 1.1225
24              def f(x):
25                      if x == 0:
26                              x = 0.01
27                      y = 4 * 10 * ((0.25/x)**12 - (0.25/x)**6)
28                      return y
29
30              inf_range = 0.8
31              sup_range = 3
32
33      elif opcion == "2":
34              true_min = -1
35              true_min2 = 1
36              def f(x):
37                      y = -1 * x**2 + 0.5 * x**4
38                      return y
39
40              inf_range = -2
41              sup_range = 2
42
43      elif opcion == "3":
44              true_min = 1.1
45              true_min2 = -1.06
46              def f(x):
47                      y = 3*(x**4) - 7*(x**2) - 0.5*x + 6
48                      return y
```

```
49
50              inf_range = -2
51              sup_range = 2
52      else:
53              print(f'Selecciona una opcion valida')
54
55      combinaciones = generar_combinaciones(Gradient_parameters)
56
57      contador = 0
58      resultados = []
59
60
61      for params in combinaciones:
62              contador += 1
63              print(f'==============Iteracion {contador} de {len(
                    combinaciones)}==================')
64
65              n_iter = params['N_iter']
66              num_mc = params['N_mc']
67              tolerance = params['Toler']
68              learning_rate = params['Learning rate']
69
70
71              soluciones = []
72              if opcion == '1':
73                      print(f"Parametros a introducir: n_MC {num_mc},
                            n_iter {n_iter}, toler {tolerance}, learning
                            rate {learning_rate}")
74
75                      start_time = time.time()
76                      for n in range(num_mc):
77                              solucion = gradient_descent(inf_range,
                                    sup_range, n_iter,  learning_rate,
                                    tolerance, f)
78                              soluciones.append(round(solucion,4))
79
80                      minimo = Counter(soluciones).most_common(1)[0][0]
81                      print(f'El valor {minimo} se ha repetido {Counter(
                            soluciones).most_common(1)[0][1]}')
82
83                      elapsed_time = time.time() - start_time
84                      resultado = params.copy()
85                      resultado["Tiempo ejecucion"] = round(elapsed_time,
                             4)
86                      resultado["Primer minimo"] = round(minimo, 4)
```

```
87                          resultado["Δ True Min"] = abs(true_min - minimo)
88
89                  elif opcion == '2':
90                          min_dist = params['Min_dist']
91                          print(f"Parametros a introducir: n_MC {num_mc},
                                n_iter {n_iter}, min_dist {min_dist}, toler {
                                tolerance}")
92
93                          start_time = time.time()
94                          for n in range(num_mc):
95                                  solucion = gradient_descent(inf_range,
                                        sup_range, n_iter,  learning_rate,
                                        tolerance, f)
96                          soluciones.append(round(solucion,4))
97
98                          cont = Counter(soluciones)
99                          mas_comunes = cont.most_common()
100                         primer_min = mas_comunes[0][0]
101
102                         for i in range(len(mas_comunes)-1):
103                                 if (abs(primer_min - mas_comunes[i+1][0]))
                                        > min_dist:
104                                         segundo_min = mas_comunes[i+1][0]
105                                         segundo_min_val = i+1
106                                         break
107                         else:
108                                 segundo_min = None
109
110                         print(f'El primer minimo {primer_min} se ha
                                repetido {Counter(soluciones).most_common(1)
                                [0][1]}')
111                         print(f'El segundo minimo {segundo_min} se ha
                                repetido {Counter(soluciones).most_common()[
                                segundo_min_val][1]}')
112
113                         elapsed_time = time.time() - start_time
114                         resultado = params.copy()
115                         resultado["Tiempo ejecucion"] = round(elapsed_time,
                                4)
116                         resultado["Primer minimo"] = round(primer_min, 4)
117                         resultado["Δ True Min"] = min(abs(true_min -
                                primer_min), abs(true_min2 - primer_min))
118
119                         resultado["Segundo minimo"] = segundo_min
120                         resultado["Δ True Min2"] = min(abs(true_min -
```

```
                        segundo_min), abs(true_min2 - segundo_min))
121
122
123             elif opcion == '3':
124                     min_dist = params['Min_dist']
125                     print(f"Parametros a introducir: n_MC {num_mc},
                            n_iter {n_iter}, min_dist {min_dist}, toler {
                            tolerance}")
126
127                     start_time = time.time()
128                     for n in range(num_mc):
129                             solucion = gradient_descent(inf_range,
                                    sup_range, n_iter,  learning_rate,
                                    tolerance, f)
130                             soluciones.append(round(solucion,4))
131
132                     cont = Counter(soluciones)
133                     mas_comunes = cont.most_common()
134                     primer_min = mas_comunes[0][0]
135
136                     for i in range(len(mas_comunes)-1):
137                             if (abs(primer_min - mas_comunes[i+1][0]))
                                    > min_dist:
138                                     segundo_min = mas_comunes[i+1][0]
139                                     segundo_min_val = i+1
140                                     break
141                     else:
142                             segundo_min = None
143
144                     print(f'El primer minimo {primer_min} se ha
                            repetido {Counter(soluciones).most_common(1)
                            [0][1]}')
145                     print(f'El segundo minimo {segundo_min} se ha
                            repetido {Counter(soluciones).most_common()[
                            segundo_min_val][1]}')
146
147                     elapsed_time = time.time() - start_time
148                     resultado = params.copy()
149                     resultado["Tiempo ejecucion"] = round(elapsed_time,
                             4)
150                     resultado["Primer minimo"] = round(primer_min, 4)
151                     resultado["Δ True Min"] = min(abs(true_min -
                            primer_min), abs(true_min2 - primer_min))
152
153                     resultado["Segundo minimo"] = segundo_min
```

```
                        resultado["Δ True Min2"] = min(abs(true_min -
                            segundo_min), abs(true_min2 - segundo_min))


                print(f'El resultado es {resultado}')

                resultados.append(resultado)

        if resultados:
                df = pd.DataFrame(resultados)
                nombre_archivo = "Gradient_Descend_NombrePotencial.csv"

                df.to_csv(nombre_archivo, index=False)
                print(f" Resultados guardados en: {nombre_archivo}")
        else:
                print("No se generaron resultados para guardar.")
```

Listing 16: Ejecucion del gradient descent para multiples variables

Anexo C: Metodo de Newton

```
import numpy as np
import autograd as ag
import matplotlib.pyplot as plt
import time
from collections import Counter

import itertools
import pandas as pd

import sympy as sp
```

Listing 17: Importar librerias

```
def ini_vec(inf_range, sup_range):
        return np.random.uniform(inf_range, sup_range)
```

Listing 18: Funcion para generar las combinaciones de los parametros

```
def newton_method(inf_range, sup_range, n_iter, tolerance, function, x):
        k = ini_vec(inf_range, sup_range)


        f_prime = sp.diff(function, x)
        f_second = sp.diff(function, x, 2)
```

```
7
8           for i in range(n_iter):
9                   if f_second.subs(x, k) == 0:
10                          print("Error en la segunda derivada, division entre
                                0")
11                          break
12
13                  f_prime_val = f_prime.subs(x, k)
14                  f_second_val = f_second.subs(x, k)
15
16                  frac = float(f_prime_val / f_second_val)
17                  k_new = (k - frac)
18
19                  if k_new < inf_range or k_new > sup_range:#condicion añ
                        adida por que se puede disparar para lennard jones
20                          break
21
22                  if abs(k_new - k) < tolerance:
23                          break
24
25                  k = k_new
26
27          return k
```

Listing 19: Funcion para el metodo de Newton

```
1 def generar_combinaciones(diccionario):
2           claves = diccionario.keys()
3           valores = diccionario.values()
4           combinaciones = list(itertools.product(*valores))
5           return [dict(zip(claves, c)) for c in combinaciones]
```

Listing 20: Funcion para generar las combinaciones de los parametros

```
1 if __name__ == "__main__":
2           T4, T3, T2, T1, T0 = 0.0, 0.0, 0.0, 0.0, 0.0
3           input("Comprueba el nombre del archivo csv que se va a guardar")
4
5           Newton_parameters = {
6                   "N_iter": [100, 500, 1000, 2000],
7                   "Min_dist": [0.1, 0.2, 0.25, 0.5, 1],
8                   "N_mc": [50, 100, 150, 200],
9                   "Toler": [0.1, 0.05, 0.01, 0.0001],
10          }
11
12          opcion = input('1.Lennard-Jones, 2.Higgs, 3.Hidrogeno: ')
```

```
13
14          x = sp.symbols('x')
15
16          if opcion == "1":
17                  Newton_parameters = {
18                          "N_iter": [100, 500, 1000, 2000],
19                          "N_mc": [50, 100, 150, 200],
20                          "Toler": [0.1, 0.05, 0.01, 0.0001],
21                  }
22                  true_min = 1.1225
23                  def f(x):
24                          if x == 0:
25                                  x = 0.01
26                          y = 4 * 10 * ((0.25/x)**12 - (0.25/x)**6)
27                          return y
28
29                  function = y = 4 * 10 * ((0.25/x)**12 - (0.25/x)**6)
30                  inf_range = 0.8
31                  sup_range = 3
32
33          elif opcion == "2":
34                  true_min = -1
35                  true_min2 = 1
36                  def f(x):
37                          y = -1 * x**2 + 0.5 * x**4
38                  return y
39
40                  function = -1 * x**2 + 0.5 * x**4
41                  inf_range = -2
42                  sup_range = 2
43
44          elif opcion == "3":
45                  true_min = 1.1
46                  true_min2 = -1.06
47                  def f(x):
48                          y = 3*(x**4) - 7*(x**2) - 0.5*x + 6
49                          return y
50
51                  function = 3*(x**4) - 7*(x**2) - 0.5*x + 6
52                  inf_range = -2
53                  sup_range = 2
54          else:
55                  print(f'Selecciona una opcion valida')
56
57          combinaciones = generar_combinaciones(Newton_parameters)
```

```
58
59          contador = 0
60          resultados = []
61
62          for params in combinaciones:
63                  contador += 1
64                  print(f'===============Iteracion {contador} de {len(
                        combinaciones)}==================')
65
66                  n_iter = params['N_iter']
67                  num_mc = params['N_mc']
68                  tolerance = params['Toler']
69
70
71                  soluciones = []
72                  if opcion == '1':
73                          print(f"Parametros a introducir: n_MC {num_mc},
                                n_iter {n_iter}, toler {tolerance}")
74
75                          start_time = time.time()
76                          for n in range(num_mc):
77                                  solucion = newton_method(inf_range,
                                        sup_range, n_iter, tolerance, function,
                                        x)
78                                  soluciones.append(round(solucion,4))
79
80                          minimo = Counter(soluciones).most_common(1)[0][0]
81                          print(f'El valor {minimo} se ha repetido {Counter(
                                soluciones).most_common(1)[0][1]}')
82
83                          elapsed_time = time.time() - start_time
84                          resultado = params.copy()
85                          resultado["Tiempo ejecucion"] = round(elapsed_time,
                                4)
86                          resultado["Primer minimo"] = round(minimo, 4)
87                          resultado["Δ True Min"] = abs(true_min - minimo)
88
89                  elif opcion == '2':
90                          min_dist = params['Min_dist']
91                          print(f"Parametros a introducir: n_MC {num_mc},
                                n_iter {n_iter}, min_dist {min_dist}, toler {
                                tolerance}")
92
93                          start_time = time.time()
94                          for n in range(num_mc):
```

```
95                                  solucion = newton_method(inf_range,
                                        sup_range, n_iter, tolerance, function,
                                        x)
96                                  soluciones.append(round(solucion,4))
97
98                          cont = Counter(soluciones)
99                          mas_comunes = cont.most_common()
100                         primer_min = mas_comunes[0][0]
101
102                         for i in range(len(mas_comunes)-1):
103                                 if (abs(primer_min - mas_comunes[i+1][0]))
                                        > min_dist:
104                                         segundo_min = mas_comunes[i+1][0]
105                                         segundo_min_val = i+1
106                                         break
107                         else:
108                                 segundo_min = None
109
110                         print(f'El primer minimo {primer_min} se ha
                            repetido {Counter(soluciones).most_common(1)
                            [0][1]}')
111                         print(f'El segundo minimo {segundo_min} se ha
                            repetido {Counter(soluciones).most_common()[
                            segundo_min_val][1]}')
112
113                         elapsed_time = time.time() - start_time
114                         resultado = params.copy()
115                         resultado["Tiempo ejecucion"] = round(elapsed_time,
                             4)
116                         resultado["Primer minimo"] = round(primer_min, 4)
117                         resultado["Δ True Min"] = min(abs(true_min -
                            primer_min), abs(true_min2 - primer_min))
118
119                         resultado["Segundo minimo"] = segundo_min
120                         resultado["Δ True Min2"] = min(abs(true_min -
                            segundo_min), abs(true_min2 - segundo_min))
121
122
123                 elif opcion == '3':
124                         min_dist = params['Min_dist']
125                         print(f"Parametros a introducir: n_MC {num_mc},
                            n_iter {n_iter}, min_dist {min_dist}, toler {
                            tolerance}")
126
127                         start_time = time.time()
```

```
128                         for n in range(num_mc):
129                                 solucion = newton_method(inf_range,
                                        sup_range, n_iter, tolerance, function,
                                        x)
130                                 soluciones.append(round(solucion,4))
131
132                         cont = Counter(soluciones)
133                         mas_comunes = cont.most_common()
134                         primer_min = mas_comunes[0][0]
135
136                         for i in range(len(mas_comunes)-1):
137                                 if (abs(primer_min - mas_comunes[i+1][0]))
                                        > min_dist:
138                                         segundo_min = mas_comunes[i+1][0]
139                                         segundo_min_val = i+1
140                                         break
141                         else:
142                                 segundo_min = None
143
144                         print(f'El primer minimo {primer_min} se ha
                            repetido {Counter(soluciones).most_common(1)
                            [0][1]}')
145                         print(f'El segundo minimo {segundo_min} se ha
                            repetido {Counter(soluciones).most_common()[
                            segundo_min_val][1]}')
146
147                         elapsed_time = time.time() - start_time
148                         resultado = params.copy()
149                         resultado["Tiempo ejecucion"] = round(elapsed_time,
                             4)
150                         resultado["Primer minimo"] = round(primer_min, 4)
151                         resultado["Δ True Min"] = min(abs(true_min -
                            primer_min), abs(true_min2 - primer_min))
152
153                         resultado["Segundo minimo"] = segundo_min
154                         resultado["Δ True Min2"] = min(abs(true_min -
                            segundo_min), abs(true_min2 - segundo_min))
155
156
157                 print(f'El resultado es {resultado}')
158
159                 resultados.append(resultado)
160
161         if resultados:
162                 df = pd.DataFrame(resultados)
```

```
163                 nombre_archivo = "Newton_Method_NombreArchivo.csv"
164
165                 df.to_csv(nombre_archivo, index=False)
166                 print(f" Resultados guardados en: {nombre_archivo}")
167         else:
168                 print("No se generaron resultados para guardar.")
```

Listing 21: Funcion para ejecutar el metodo de Newton para multiples varibales

Anexo D: Dependencias

Librería	Versión
Python	3.12.5
qiskit	2.0.0
qiskit-aer	0.17.0
matplotlib	3.9.2
autograd	1.7.0
numpy	2.0.2
sympy	1.13.3
seaborn	0.13.2
pandas	2.2.2

Cuadro 15: Librerías y versiones utilizadas en el proyecto