



UNIVERSIDADE DA CORUÑA

FACULTAD DE INFORMÁTICA
TRABAJO FIN DE GRADO
GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

SIMULACIÓN DE LA TRANSFORMADA CUÁNTICA DE FOURIER

Autor/a: Nicolás Echevarrieta Catalán

Director/a: Vicente Moret Bonillo

OKI

Colaborador Oficial en
Soluciones de Impresión

A Coruña, a 9 de septiembre de 2016.

Resumen

Este proyecto trata la simulación de la Transformada cuántica de Fourier, que es la operación análoga a la Transformada de Fourier para un estado cuántico.

Para ello, se ha elegido una librería de simulación cuántica para *python* bastante utilizada llamada *Qutip* y se ha realizado una implementación de la Transformada cuántica de Fourier diferente a la que ofrece, obteniendo mejores resultados en número de qubits operables, consumo de memoria y tiempo de ejecución.

Palabras clave:

- ✓ Computación Cuántica
- ✓ Transformada de Fourier
- ✓ Transformada cuántica de Fourier
- ✓ Algoritmo cuántico
- ✓ Qubit
- ✓ Operador cuántico
- ✓ Puerta cuántica
- ✓ Optimización
- ✓ Paralelismo
- ✓ Python
- ✓ Qutip

Índice general

	Página
1. Introducción	1
1.1. Motivación	1
1.2. Objetivos generales	2
1.3. Estructura de este documento	3
1.4. Metodología	4
2. Contexto	7
2.1. Mecánica cuántica	8
2.1.1. Principios cuánticos	11
2.2. Computación cuántica	13
2.2.1. Unidad de información: El qubit	14
2.2.2. Esfera de Bloch	17
2.2.3. Colapsar a un estado (medición)	18
2.2.4. Entrelazamiento	18
2.2.5. Superposición de estados	18
2.2.6. Operadores clásicos (puertas lógicas)	19
2.2.6.1. Combinación de operadores	21
2.2.7. Operadores reversibles	22
2.2.8. Ordenador reversible	23
2.2.9. Operadores cuánticos (puertas cuánticas)	25
2.2.10. Puertas cuánticas universales	27
2.2.11. Procesador cuántico de uso general	28
2.2.12. Algoritmos cuánticos	28
2.3. Software utilizado	29
2.3.1. VMware	29

2.3.2. Qutip	30
2.3.3. Python	30
2.3.4. SciPy	31
2.3.5. Numpi	31
3. Planificación	33
3.1. Fases	33
3.2. Infraestructura	34
3.3. Estimación	35
3.4. Estimación del coste	36
4. Transformada de Fourier	43
4.1. Series de Fourier	44
4.2. Transformada de Fourier continua (FT)	45
4.3. Transformada de Fourier discreta (DFT)	45
4.4. Transformada cuántica de Fourier (QFT)	47
4.5. Circuito de la transformada cuántica de Fourier	50
5. Etapa inicial	53
5.1. Diseño de las mediciones	54
5.2. Validación	56
5.3. Diseño de clases	56
5.4. Implementación de <i>qutip</i>	59
6. Implementación base	61
6.1. <i>base_impl</i> : Implementación base	61
6.1.1. Diseño	62
6.1.2. Estudio de coste espacial - Teórico	62
6.1.3. Estudio de coste espacial - Práctico	66
6.1.4. Estudio de coste temporal - Teórico	67
6.1.5. Estudio de coste temporal - Práctico	68
6.1.6. Conclusiones	68
7. Optimización del consumo de memoria	71
7.1. <i>mem_opt_1</i> : Operador parcial	72
7.1.1. Diseño	73
7.1.2. Estudio de coste espacial - Teórico	73
7.1.3. Estudio de coste espacial - Práctico	75

7.1.4.	Estudio de coste temporal - Teórico	77
7.1.5.	Estudio de coste temporal - Práctico	77
7.1.6.	Conclusiones	78
7.2.	<i>mem_opt_2</i> : No convertir el operador	79
7.2.1.	Diseño	80
7.2.2.	Estudio de coste espacial - Teórico	80
7.2.3.	Estudio de coste espacial - Práctico	81
7.2.4.	Estudio de coste temporal - Teórico	82
7.2.5.	Estudio de coste temporal - Práctico	82
7.2.6.	Conclusiones	84
7.3.	<i>mem_opt_3</i> : Operador parcial y no convertir el operador	85
7.3.1.	Diseño	86
7.3.2.	Estudio de coste espacial - Teórico	86
7.3.3.	Estudio de coste espacial - Práctico	87
7.3.4.	Estudio de coste temporal - Teórico	87
7.3.5.	Estudio de coste temporal - Práctico	89
7.3.6.	Conclusiones	89
7.4.	Resultados obtenidos	90
8.	Optimización del consumo de tiempo de ejecución	97
8.1.	<i>time_opt_1</i> : Reducción de cálculos	99
8.1.1.	Diseño	99
8.1.2.	Estudio de coste espacial - Teórico	99
8.1.3.	Estudio de coste espacial - Práctico	100
8.1.4.	Estudio de coste temporal - Teórico	102
8.1.5.	Estudio de coste temporal - Práctico	102
8.1.6.	Conclusiones	103
8.2.	<i>time_opt_2</i> : Ejecución multithread	103
8.2.1.	Diseño	104
8.2.2.	Estudio de coste espacial - Teórico	106
8.2.3.	Estudio de coste espacial - Práctico	107
8.2.4.	Estudio de coste temporal - Teórico	109
8.2.5.	Estudio de coste temporal - Práctico	112
8.2.6.	Conclusiones	112
8.3.	<i>time_opt_3</i> : Ejecución multithread con cálculos reducidos	113
8.3.1.	Diseño	113
8.3.2.	Estudio de coste espacial - Teórico	114

8.3.3. Estudio de coste espacial - Práctico	115
8.3.4. Estudio de coste temporal - Teórico	117
8.3.5. Estudio de coste temporal - Práctico	118
8.3.6. Conclusiones	119
8.4. Resultados obtenidos	120
9. Seguimiento	123
9.1. Coste final	124
10.Resultados finales	131
11.Conclusiones	135
11.1. Valoración	135
11.2. Consecución de objetivos	136
11.3. Problemas surgidos	137
11.4. Lecciones aprendidas	137
11.5. Trabajo futuro	138
A. Manual de usuario	139
Bibliografía	141

Índice de figuras

Figura	Página
2.1. Catástrofe ultravioleta	9
2.2. Formulación moderna del experimento de Young	10
2.3. Resultados del experimento de Young con electrones	11
2.4. Representación de un qubit $ \psi\rangle$ en una esfera de Bloch	17
3.1. Diagrama de Gantt: General	37
3.2. Diagrama de Gantt: Etapas previas	38
3.3. Diagrama de Gantt: Etapa de refinamiento del consumo de memoria . .	39
3.4. Diagrama de Gantt: Etapa de refinamiento del consumo de tiempo . . .	40
3.5. Diagrama de Gantt: Redacción de memoria	41
4.1. Aproximación por serie de Fourier	44
4.2. Circuito cuántico de la QFT para 3 qubits	52
4.3. Circuito cuántico de la QFT para n qubits	52
5.1. Diagrama de clases: Implementaciones	55
5.2. <i>qutip_impl</i> - MemoryError al ejecución para 13 qubits	60
6.1. <i>base_impl</i> - Diagrama de flujo	62
6.2. <i>base_impl</i> - Consumo de recursos solo con el SO en ejecución	65
6.3. <i>base_impl</i> - Ejecución con 12 qubits	67
6.4. <i>base_impl</i> - Ejecución con 13 qubits	67
7.1. <i>mem_opt_1</i> - Diagrama de flujo - Cambios	74
7.2. <i>mem_opt_2</i> - Memoria en el monitor de recursos (12 qubits)	80
7.3. <i>mem_opt_2</i> - Diagrama de flujo - Cambios	81
7.4. <i>mem_opt_2</i> - Memoria en el monitor de recursos (12 qubits)	82

7.5. <i>mem_opt_3</i> - Diagrama de flujo - Cambios	86
7.6. Implementación base - Ejecución para 13 qubits	94
7.7. <i>mem_opt_1</i> - Ejecución para 25 qubits	94
7.8. <i>mem_opt_2</i> - Ejecución para 13 qubits	95
8.1. <i>time_opt_1</i> - Diagrama de flujo - Cambios	100
8.2. <i>time_opt_2</i> - Master - Diagrama de flujo - Cambios	106
8.3. <i>time_opt_2</i> - Slave - Diagrama de flujo	107
8.4. <i>time_opt_2</i> - Medición de memoria con el <i>core</i> de <i>python</i>	111
8.5. <i>time_opt_3</i> - Master - Diagrama de flujo - Cambios	115
8.6. <i>time_opt_3</i> - Slave - Diagramas de flujo	117
9.1. Diagrama de Gantt - Seguimiento: General	125
9.2. Diagrama de Gantt - Seguimiento: Etapas previas	126
9.3. Diagrama de Gantt - Seguimiento: Etapa de refinamiento del consumo de memoria	127
9.4. Diagrama de Gantt - Seguimiento: Etapa de refinamiento del consumo de tiempo	128
9.5. Diagrama de Gantt - Seguimiento: Redacción de memoria	129
9.6. Diagrama de Gantt - Seguimiento - Comparación: General	130

Índice de tablas

Tabla	Página
5.1. Implementación de <i>gutip</i> - Resultados de las mediciones	60
6.1. <i>base_impl</i> - Incremento mínimo teórico en el uso de memoria	64
6.2. <i>base_impl</i> - Resultados de las mediciones	69
7.1. <i>mem_opt_1</i> - Comparativa del uso de memoria (MB)	76
7.2. <i>mem_opt_1</i> - Comparativa del consumo de tiempo (s)	78
7.3. <i>mem_opt_1</i> - Resultados de las mediciones	79
7.4. <i>mem_opt_2</i> - Comparativa del uso de memoria (MB)	83
7.5. <i>mem_opt_2</i> - Comparativa del consumo de tiempo (s)	84
7.6. <i>mem_opt_2</i> - Resultados de las mediciones	85
7.7. <i>mem_opt_3</i> - Comparativa del uso de memoria (MB)	88
7.8. <i>mem_opt_3</i> - Comparativa del consumo de tiempo (s)	90
7.9. <i>mem_opt_3</i> - Resultados de las mediciones	91
7.10. <i>mem_opt</i> - Comparativa del uso de memoria (MB)	92
7.11. <i>mem_opt</i> - Comparativa del consumo de tiempo (s)	93
8.1. <i>time_opt_1</i> - Comparativa del uso de memoria (MB)	101
8.2. <i>time_opt_1</i> - Comparativa del consumo de tiempo (s)	103
8.3. <i>time_opt_1</i> - Resultados de las mediciones	104
8.4. <i>time_opt_2</i> - Comparativa del uso de memoria (MB)	110
8.5. <i>time_opt_2</i> - Comparativa del consumo de tiempo (s)	113
8.6. <i>time_opt_2</i> - Resultados de las mediciones	114
8.7. <i>time_opt_3</i> - Comparativa del uso de memoria (MB)	116
8.8. <i>time_opt_3</i> - Comparativa del consumo de tiempo (s)	118
8.9. <i>time_opt_3</i> - Resultados de las mediciones	119

8.10. time_opt - Comparativa del uso de memoria (MB)	120
8.11. time_opt - Comparativa del consumo de tiempo (s)	121
10.1. Comparativa del uso de memoria (MB)	133
10.2. Comparativa del consumo de tiempo (s)	134

Capítulo 1

Introducción

Contenidos

1.1. Motivación	1
1.2. Objetivos generales	2
1.3. Estructura de este documento	3
1.4. Metodología	4

En este capítulo se describen los rasgos generales del proyecto en su conjunto, así como su estructura.

1.1. Motivación

La computación cuántica supone un área de investigación muy prometedora debido al cambio de complejidad que presentan algunas tareas, pero de momento los procesadores cuánticos disponibles son muy limitados y con un coste desorbitado, por lo que es necesario simular su comportamiento a fin de poder estudiar nuevos algoritmos [11].

Dentro de los algoritmos existentes, la transformada cuántica de Fourier supone un paso básico dentro de otros algoritmos y se espera lo sea también en

nuevos algoritmos que surjan, siendo un punto crítico en la complejidad que suponga simular dichos algoritmos.

Existen librerías de simulación cuántica que permiten realizar la transformada cuántica de Fourier, como por ejemplo la librería *qutip* para *python* (una de las librerías más usadas en simulación cuántica) que ofrece una función cuya salida es un objeto con dicho operador.

La operación de la transformada cuántica de Fourier en dicha librería tiene una limitación en cuanto al número máximo de qubits, pudiendo generarlo para 12 qubits, pero no para 13. Esta limitación no es dependiente de la memoria de la máquina pues se ha comprobado que da en máquinas con diferentes tamaños de memoria.

Debido a esta limitación, no es posible simular con esta librería algoritmos que hagan uso de la transformada de Fourier, para estados de más de 12 qubits, cosa que podría ser interesante.

1.2. Objetivos generales

Con la motivación presentada se pretende crear una implementación de la transformada cuántica de Fourier que permita trabajar con un número de qubits mayor. Además, para agilizar los algoritmos que deban utilizarla, es importante reducir su tiempo de ejecución y consumo de memoria en la medida de lo posible.

Para llevar a cabo la implementación se han diferenciado varios objetivos que engloban todo el proyecto:

Objetivo-1 Comprender las bases de la computación cuántica.

Objetivo-2 Comprender el algoritmo concreto de la transformada cuántica de Fourier.

Objetivo-3 Construir una simulación básica de dicho algoritmo.

Objetivo-4 Optimizar la simulación para poder usarlo con mas cantidad de qubits y que requiera menos memoria.

Objetivo-5 Optimizar la simulación para reducir su tiempo de ejecución.

1.3. Estructura de este documento

Este documento se estructura en: introducción (la presente parte), contexto, planificación, explicación de la transformada de Fourier, una etapa inicial, una implementación inicial un proceso de optimización del consumo espacial, un proceso de optimización de consumo temporal, seguimiento, resultados finales y conclusiones.

En el capítulo de contexto se describirán todos aquellos componentes y conocimientos necesarios para entender el presente trabajo.

La planificación contiene el desglose en tareas de las distintas fases, así como su estimación inicial y secuencia de ejecución.

El capítulo de explicación de la transformada de Fourier contiene una explicación de dicha operación y como se llega hasta la transformada cuántica de Fourier, explicando las características e implementación de esta última.

El proceso de desarrollo del proyecto se ha dividido en cuatro fases cada una con un capítulo propio:

- Una etapa inicial donde se realiza un diseño general de como realizar la implementación, pruebas y mediciones, seguido de un estudio de la implementación ofrecida por la librería de *qutip* que se pretende mejorar.
 - La implementación inicial (base) es una implementación sencilla de la operación que sirve como base de las sucesivas iteraciones de mejora.
 - Esa implementación base es refinada en varias iteraciones tratando de aumentar el número de qubits operables a la vez que reduciendo la memoria necesaria.
-

- Con la mejor implementación de la fase anterior (más número de qubits, con el menor consumo de memoria y a poder ser, el menor consumo temporal) se va a proceder a otro proceso iterativo, esta vez buscando mejorar su consumo temporal para hacer que se ejecute más rápido.

El capítulo de seguimiento detalla como ha ido el proyecto en contraste a la planificación inicial.

Los resultados finales muestran y analizan una comparativa con los datos de las mediciones de todas las implementaciones.

En las conclusiones se presenta una valoración de los resultados, un análisis de los problemas surgidos durante el desarrollo del trabajo y las lecciones aprendidas de dichos problemas, seguido del trabajo que ha quedado más allá del alcance de este trabajo así como las vías de desarrollo futuro que se ven para el sistema.

1.4. Metodología

El proyecto se han estructurado en cuatro etapas:

Estudio previo: Serie de objetivos secuenciales de aprendizaje (números complejos, álgebra vectorial y teoría cuántica), así como una medición de referencia de la función ya existente para realizar la transformada, la cual será usada para contrastar los resultados obtenidos.

Implementación base: Implementación de partida para el presente trabajo. No se realizara optimización alguna, resultando en la simulación cruda del algoritmo.

Refinamiento del coste espacial: Iteraciones sucesivas con el objetivo de reducir la cantidad de memoria utilizada por la implementación base y con ello tratar de aumentar el número de qubits operables.

Refinamiento del coste temporal: Partiendo de la implementación con el mayor número de qubits operables y menor consumo espacial, se le aplicarán

mejoras con el fin de reducir su tiempo de ejecución sin reducir el número de qubits operables.

Capítulo 2

Contexto

Contenidos

2.1. Mecánica cuántica	8
2.1.1. Principios cuánticos	11
2.2. Computación cuántica	13
2.2.1. Unidad de información: El qubit	14
2.2.2. Esfera de Bloch	17
2.2.3. Colapsar a un estado (medición)	18
2.2.4. Entrelazamiento	18
2.2.5. Superposición de estados	18
2.2.6. Operadores clásicos (puertas lógicas)	19
2.2.7. Operadores reversibles	22
2.2.8. Ordenador reversible	23
2.2.9. Operadores cuánticos (puertas cuánticas)	25
2.2.10. Puertas cuánticas universales	27
2.2.11. Procesador cuántico de uso general	28
2.2.12. Algoritmos cuánticos	28
2.3. Software utilizado	29
2.3.1. VMware	29
2.3.2. Qutip	30

2.3.3. Python	30
2.3.4. SciPy	31
2.3.5. Numpy	31

Este proyecto trata la simulación de una operación de computación cuántica y por ello en este capítulo se va a proceder a explicar aquellos conceptos que se consideran necesarios para entender la presente memoria.

Antes de empezar se ha de comentar que, al no ser este un proyecto centrado en el apartado físico de la mecánica cuántica sino en su aplicación a la algoritmia y por tanto a la computación, se va a tratar lo menos posible desde el punto de vista físico, intentando de explicar los conceptos físicos prescindiendo en lo posible de las matemáticas asociadas.

2.1. Mecánica cuántica

La teoría de la mecánica cuántica surgió durante la primera mitad del siglo XX, en la cual se logra explicar ciertas contradicciones entre los hechos descritos por la mecánica clásica o la electrodinámica con lo observado experimentalmente. Dos de estas contradicciones son la llamada “*catástrofe ultravioleta*”, la cual que llevo a Max Planck a desarrollar un modelo heurístico del que surgiría la mecánica cuántica, y la dualidad onda-partícula que explica el experimento de Young.

La “*catástrofe ultravioleta*” es un fenómeno no explicable con la teoría clásica del electromagnetismo que ocurre en la emisión electromagnética de un cuerpo en equilibrio térmico con el ambiente. De acuerdo con las predicciones del electromagnetismo clásico, un cuerpo negro ideal (objeto teórico o ideal que absorbe toda la luz y toda la energía radiante que incide sobre él) en equilibrio térmico debía emitir energía en todos los rangos de frecuencia (aumentando la energía proporcionalmente al cuadrado de la frecuencia), lo cual se comprobó con bajas frecuencias (infrarrojos). No obstante, cuando las técnicas de medición evolucionaron se comprobó que la predicción teórica no se cumplía para el espectro visible y el ultravioleta, pues al aumentar la frecuencia la energía tiende a cero. En la fi-

gura 2.1 se observan los resultados de la medición de energía radiada como función de la longitud de onda para varios cuerpos negros a diferentes temperaturas.

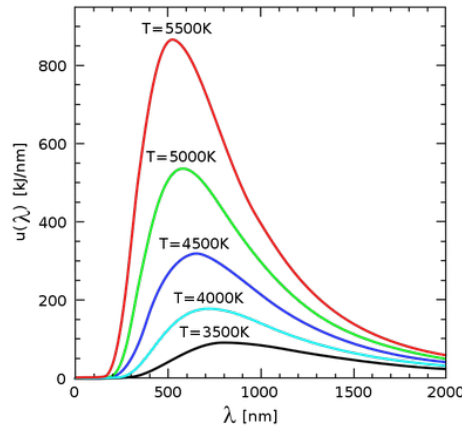


Figura 2.1: Catástrofe ultravioleta

La explicación a este fenómeno fue planteada por Max Planck en 1900, con lo que se conoce ahora como ley de Planck (la cual mide la intensidad de la radiación emitida por un cuerpo negro con una cierta temperatura y frecuencia). Ese momento se considera como el principio de la Mecánica cuántica. Incluso el propio nombre de la mecánica cuántica proviene de la unidad de energía descrita en la ley de Planck, el “cuanto” (quantum en inglés).

El otro fenómeno mencionado, la dualidad onda-partícula, se observa en el experimento de Young (también denominado experimento de la doble rendija) que fue realizado en 1801 por Thomas Young en un intento de discernir sobre la naturaleza corpuscular u ondulatoria de la luz, contribuyendo a la teoría de la naturaleza ondulatoria de la luz, fundamental a la hora de demostrar la dualidad onda corpúsculo, una característica de la mecánica cuántica. En el experimento original un estrecho haz de luz, procedente de un pequeño agujero en la entrada de una cámara, es dividido en dos haces por una tarjeta de una anchura de 0.2 mm, cada uno pasando por un lado distinto de la tarjeta divisoria. Hay no obstante una formulación moderna del experimento que permite mostrar tanto la naturaleza ondulatoria de la luz como la dualidad onda-corpúsculo de la materia. Esta formulación moderna del experimento también tiene lugar en una cámara oscura donde se deja entrar un haz de luz por una rendija estrecha. La luz llega a una pared intermedia con dos rendijas al otro lado de la cual hay una pantalla de

proyección o una placa fotográfica. Cuando una de las rejillas se cubre aparece un único pico correspondiente a la luz que proviene de la rendija abierta. Sin embargo cuando ambas están abiertas, en lugar de formarse una imagen superposición de las obtenidas con las rendijas abiertas individualmente (tal y como ocurriría si la luz estuviera hecha de partículas), se obtiene una figura de interferencias con rayas oscuras y otras brillantes como se muestra en la figura 2.2.

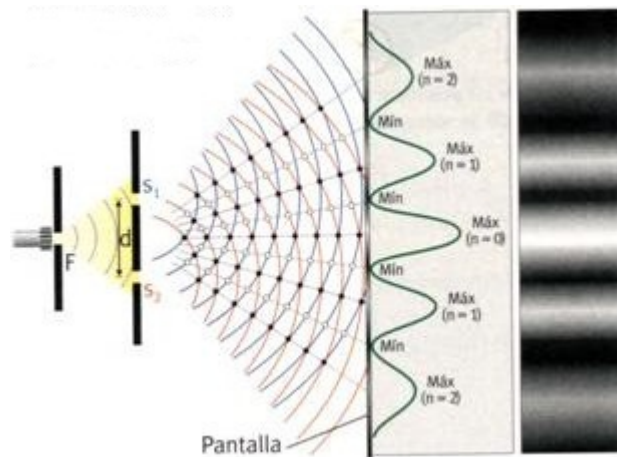


Figura 2.2: Formulación moderna del experimento de Young

Este patrón de interferencias se explica fácilmente a partir de la interferencia de las ondas de luz al combinarse la luz que procede de dos rendijas, de manera muy similar a como las ondas en la superficie del agua se combinan para crear picos y regiones más planas. En las líneas brillantes la interferencia es de tipo “constructiva” (el mayor brillo se debe a la superposición de ondas de luz cuya fase coincide al llegar a la superficie de proyección) y en las líneas oscuras la interferencia es de tipo “destructiva” (prácticamente ausencia de luz a consecuencia de la llegada de ondas de luz de fase opuesta, gráficamente la cresta de una onda se superpone con el valle de otra).

Para la década de 1920, numerosos experimentos habían demostrado que la luz interacciona con la materia únicamente en cantidades discretas, paquetes “cuánticos” denominados fotones. Si la fuente de luz pudiera reemplazarse por una fuente capaz de producir fotones individualmente y la pantalla fuera suficientemente sensible para detectar un único fotón, el experimento de Young podría, en principio, producirse con fotones individuales con idéntico resultado.

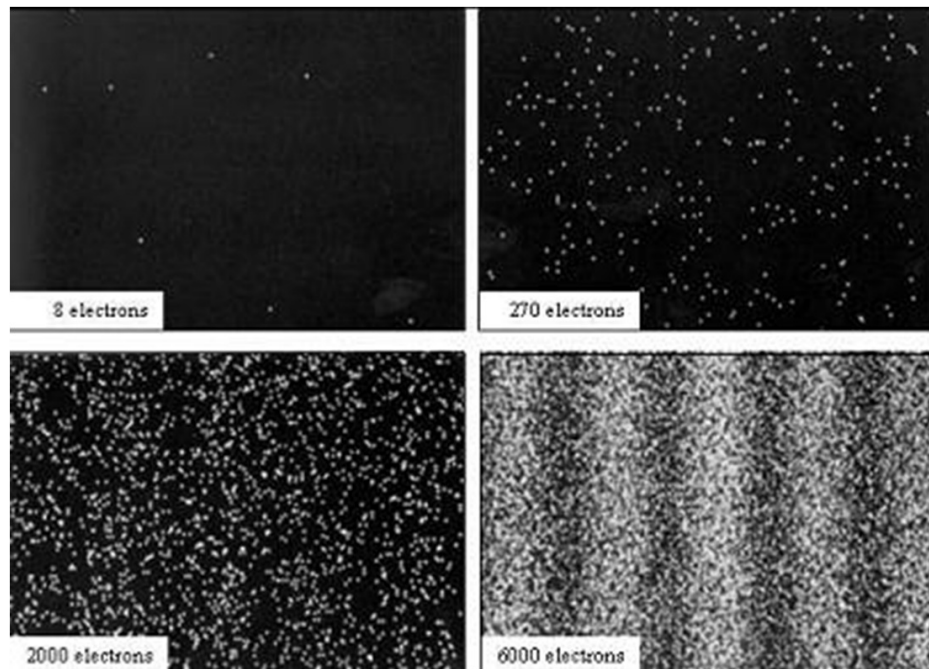


Figura 2.3: Resultados del experimento de Young con electrones

En 1961 se realizó el experimento de Young con electrones demostrando la dualidad onda-corpúsculo de las partículas subatómicas. Los resultados se muestran en la figura 2.3.

Mediante estos y otros experimentos, desde el siglo XX se han ido descubriendo y demostrando los principios (o postulados) de la mecánica cuántica.

2.1.1. Principios cuánticos

La mecánica cuántica se construye sobre unos principios fundamentales, los cuales son

Entrelazamiento: Dos partículas cuánticas pueden tener estados fuertemente correlacionados, debido a que se generaron al mismo tiempo o a que interactuaron (por ejemplo durante un choque). Cuando esto ocurre se dice que sus estados están entrelazados, lo que provoca que la modificación (por manipulación o medición) de una de ellas influye de igual manera en la otra,

sin importar la distancia que las separe. Este fenómeno se explica aplicando las leyes de conservación del momento y de la energía.

Incertidumbre: En la teoría cuántica, algunos pares de propiedades físicas son complementarias (por ejemplo la posición y el momento lineal de una partícula) en el sentido de que es imposible saber a la vez el valor exacto de ambas. Si se mide una propiedad, necesariamente se altera la complementaria, perdiéndose cualquier noción de su valor exacto. Cuanto más precisa sea la medición sobre una propiedad, mayor será la incertidumbre de la otra propiedad. Si por ejemplo pensamos en el proceso de medir la posición de un electrón, es necesario que un fotón de luz choque con el electrón, modificando con esto su momento lineal, por lo que la siguiente medición para conocer su momento lineal no ofrecerá el mismo resultado que si se hubiera realizado antes de medir su posición.

Por ello no es posible conocer exactamente el valor de todas las magnitudes físicas que describen el estado de una partícula cuántica en ningún momento, sino sólo una distribución estadística. Por lo tanto no es posible asignar una trayectoria a una partícula, pero sí se puede decir que hay una determinada probabilidad de que la partícula se encuentre en una determinada región del espacio en un momento determinado.

Superposición: Sostiene que un sistema físico tal como un electrón existe simultáneamente en todos sus teóricamente posibles estados (o la configuración de sus propiedades) de forma simultánea, pero cuando se mide da un resultado que corresponde a sólo una de las posibles configuraciones.

Colapso (Lectura destructiva): Como indica el principio de superposición, una partícula se puede encontrar en todos sus estados posibles simultáneamente, lo cual significa que “potencialmente” contiene información de cada uno de los posibles estados. No obstante, en el momento de ser medida, dicha partícula se “fija” en uno, y solo uno, de sus posibles estados, siendo la información medida perteneciente al estado al cual se ha fijado. Además lo hace de forma que, siempre en cuando no se modifique el estado de la partícula, sucesivas mediciones darán el mismo resultado.

A este efecto de, en el momento de medir una partícula que está en un estado de superposición, fijarse a un estado no superpuesto de entre los posibles estados que se encontraban en superposición, y que además lo haga de forma que la información “potencial” del resto de estados posibles de la superposición se pierda, se le conoce como “lectura destructiva”.

Esta propiedad lleva al teorema de no clonación, el cual demuestra que no es posible la existencia de una operación cuántica que, dado un estado cuántico cualquiera, produzca dos estados exactamente iguales al estado de entrada

$$\nexists U_{CLON} \quad \backslash \quad U_{CLON}|b\rangle|0\rangle = |b\rangle|b\rangle$$

siendo $|b\rangle$ un estado cualquiera.

Llegado a este punto, es natural preguntarse el por qué los objetos a nivel macroscópicos y newtonianos, así como los acontecimientos reales, no parecen exhibir propiedades mecánico cuánticas tales como la superposición. La respuesta es sencilla: para que funcionen principios de la mecánica cuántica como el de superposición se necesita el aislamiento completo del sistema, siendo esto actualmente no posible para objetos macroscópicos ya que están formados por millones de partículas que habría que aislar a su vez, y una vez aisladas, dejarían de formar parte el todo que es el objeto a estudiar, y por lo tanto éste dejaría de existir como tal.

2.2. Computación cuántica

La computación cuántica es una nueva rama de la computación que aprovecha las propiedades de la mecánica cuántica con el fin de resolver ciertos problemas matemáticos.

Aprovechando los principios de la mecánica cuántica, como el de superposición o entrelazamiento, se pueden desarrollar y ejecutar nuevos algoritmos para resolver algunos problemas con una gran mejora respecto al mejor algoritmo de computación clásica conocido para el mismo problema, muchas veces cambiando drásticamente la complejidad del problema.

Por ejemplo, el problema de factorizar un número en números primos. Mediante computación clásica con los mejores algoritmos de uso general conocidos (generalmente basados en congruencia de cuadrados) tiene complejidad temporal exponencial mientras que el mejor algoritmo conocido mediante computación cuántica (el algoritmo de Shor) tiene complejidad temporal polinómica, siendo este un ejemplo de un problema que, al cambiar de paradigma computacional, cambia de complejidad.

Para poder simular el comportamiento de un circuito cuántico en un ordenador clásico se utiliza la representación matemática, tanto de los estados como de los operadores, utilizando álgebra matricial y números complejos. Por ello a partir de este punto, las explicaciones se centran en la representación matemática (de la forma más sencilla posible) orientada a la computación cuántica en lugar de a los procesos físicos que representan.

2.2.1. Unidad de información: El qubit

El qubit es la unidad de información básica en computación cuántica (de la misma forma que el bit lo es en computación clásica).

Para representar un qubit se utiliza la llamada notación de Dirac (también conocida como notación bra-ket) en la cual un estado se representa mediante una matriz columna o ket ($| \rangle$). Así, tenemos dos estados básicos

$$|0\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad |1\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

(correspondientes a los estados binarios de un bit) y la superposición cuántica de ambos estados se representa por su combinación lineal

$$c_0 \cdot |0\rangle + c_1 \cdot |1\rangle = c_0 \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} + c_1 \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \end{bmatrix}$$

tal que c_0 y c_1 son números complejos y $|c_0|^2 + |c_1|^2 = 1$ (probabilidad total), siendo $|c_0|^2$ la probabilidad de que, al medir el qubit, este se encuentre en el

$$C^2 = \begin{bmatrix} \mathbf{0} & c_0 \\ \mathbf{1} & c_1 \end{bmatrix}$$
$$\begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = \frac{|1\rangle + |0\rangle}{\sqrt{2}}$$
$$\begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} = \frac{|0\rangle - |1\rangle}{\sqrt{2}} \neq \frac{|1\rangle - |0\rangle}{\sqrt{2}} = \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$
$$11010010 = |1\rangle|1\rangle|0\rangle|1\rangle|0\rangle|0\rangle|1\rangle|0\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$
[illegible]

$$\begin{array}{cc}
00000000 & c_0 \\
00000001 & c_1 \\
\vdots & \vdots \\
11010011 & c_{211} \\
11010100 & c_{212} \\
\vdots & \vdots \\
11111110 & c_{254} \\
11111111 & c_{255}
\end{array}$$

tal que $\sum_{i=0}^{255} |c_i|^2 = 1$.

Para representar un byte solo es necesario representar 8 dígitos binarios, pero para representar un qubyte (8 qubits) es necesario representar $2^8 = 256$ números complejos. Este incremento exponencial de la capacidad necesaria es uno de los motivos por los que es inviable hacer experimentos con sistemas de muchos qubits (un estado en un sistema de 64 qubits necesita representar $2^{64} = 18,446,744,073,709,551,616$ números complejos).

Así como un mismo estado en un sistema de un qubit tiene varias representaciones, lo mismo pasa con un estado en un sistema de más de un qubit

$$\begin{array}{cc}
00 & c_0 \\
01 & c_1 \\
10 & c_2 \\
11 & c_3
\end{array}
\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}
\rightarrow \frac{1}{\sqrt{3}}
\begin{bmatrix} 1 \\ 0 \\ -1 \\ 1 \end{bmatrix}
= \frac{1}{\sqrt{3}}|00\rangle - \frac{1}{\sqrt{3}}|10\rangle + \frac{1}{\sqrt{3}}|11\rangle = \frac{|00\rangle - |10\rangle + |11\rangle}{\sqrt{3}}$$

y en general, un estado cuántico para un sistema de dos qubits se escribe

$$|\psi\rangle = c_{0,0}|00\rangle + c_{0,1}|01\rangle + c_{1,0}|10\rangle + c_{1,1}|11\rangle$$

Un detalle a tener en cuenta es que la combinación de dos estados cuánticos, al igual que con los bits, no es conmutativa

$$|0\rangle \otimes |1\rangle = |0 \otimes 1\rangle = |01\rangle \neq |10\rangle = |1 \otimes 0\rangle = |1\rangle \otimes |0\rangle$$

2.2.2. Esfera de Bloch

La esfera de Bloch es una representación geométrica del espacio de estados de un sistema cuántico de un qubit. Dicha representación puede observarse en la figura 2.4.

Cualquier punto de la esfera de Bloch representa un estado cuántico o qubit, el cual puede expresarse como

$$|\psi\rangle = \cos(\theta/2)|0\rangle + e^{i\phi} \sin(\theta/2)|1\rangle$$

donde θ y ϕ son números reales tales que $0 \leq \theta \leq \pi$ y $0 \leq \phi \leq 2\pi$ y se corresponden con el ángulo que forma el radio del punto que representa el estado cuántico con el eje Z (θ) y con el eje X (ϕ).

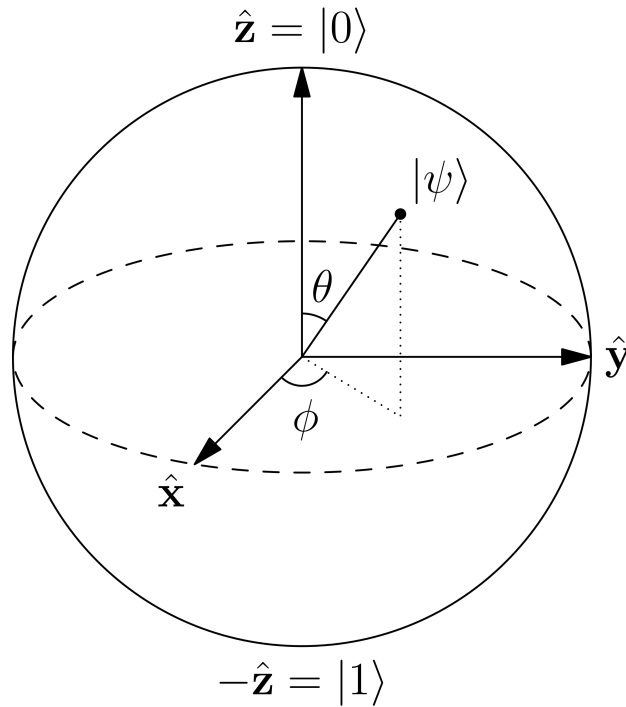


Figura 2.4: Representación de un qubit $|\psi\rangle$ en una esfera de Bloch

2.2.3. Colapsar a un estado (medición)

Mientras se esta trabajando con los qubits, estos se encuentran en un estado de superposición. Lo que se varía durante la ejecución del algoritmo es la probabilidad de que, al realizar la medición del sistema, el resultado sea uno u otro estado concreto de entre todos los posibles, perdiéndose la información referente al resto de estados.

Esto es así pues, debido al principio de lectura destructiva, una vez se realice la medición los qubits medidos dejarán de estar en un estado de superposición $c_0|0\rangle + c_1|1\rangle$ para “colapsar” a $|0\rangle$ o $|1\rangle$ con una probabilidad $|c_0|^2$ de ser $|0\rangle$ y $|c_1|^2$ de ser $|1\rangle$ (en general, $|c_i|^2$ para que colapse en el estado $|i\rangle$).

2.2.4. Entrelazamiento

El entrelazamiento es la propiedad cuántica por la cual cuando dos estados están entrelazados, lo que le ocurra a uno afectará al otro.

Por ejemplo, en un sistema con dos qubits entrelazados se tiene con el estado

$$\frac{|00\rangle + |11\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$$

en el cual si se colapsase el sistema completo se obtendrá $|00\rangle$ o $|11\rangle$, pero nunca se obtendrá $|01\rangle$ ni $|10\rangle$. Por lo tanto, si en ese sistema se colapsa sólo uno de los qubits y este resulta en un estado $|1\rangle$, automáticamente se sabe que el otro qubit también se encuentra en $|1\rangle$ sin necesidad de colapsarlo, debido a que el sistema en su conjunto no permite otras opciones.

2.2.5. Superposición de estados

La propiedad de superposición es la propiedad que permite a un elemento cuántico encontrarse, mientras que no sea medido, en más de un estado simultáneamente.

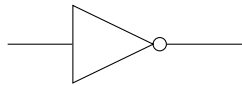
En computación cuántica se parte de un qubit en un estado puro ($|0\rangle$ o $|1\rangle$) y se le aplica un operador de Hadamard, el cual coloca el qubit de entrada en un estado de superposición equidistante de sus dos posibles estados (respectivamente $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ ó $\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$). Si esto se realiza sobre un conjunto de qubits, el resultado es una superposición equidistante de cada uno de los posibles estados del conjunto.

2.2.6. Operadores clásicos (puertas lógicas)

En computación las operaciones se llevan a cabo aplicando operadores a los estados con los que se trabaja. En computación clásica estos operadores se implementan como puertas lógicas y su combinación.

Matemáticamente los operadores se representan como matrices y su aplicación se corresponde a la multiplicación por la izquierda de dicha matriz operador sobre la representación matricial del estado.

Un ejemplo de puerta lógica es la operación NOT, que actúa sobre bits individuales y conmuta el estado $|0\rangle$ por $|1\rangle$ y viceversa se representa mediante la matriz operador:



$$NOT = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

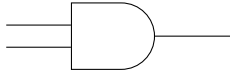
y se puede comprobar que

$$NOT|0\rangle = |1\rangle \quad NOT|1\rangle = |0\rangle$$

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Otro ejemplo puede ser la puerta lógica AND que tiene por entrada un estado compuesto por dos bits y por salida un estado de un solo bit. La salida se

corresponde con el $|1\rangle$ unicamente cuando en la entrada el estado es $|11\rangle$, siendo $|0\rangle$ en cualquiera de los tres casos restantes. Matricialmente se representa como le operador



$$AND = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

y se puede comprobar que, por ejemplo

$$AND|11\rangle = |1\rangle$$

$$AND|01\rangle = |0\rangle$$

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Generalizando, un operador que actúe sobre un estado de n bits y devuelva un estado de m bits sera una matriz con 2^n columnas y 2^m filas donde cada columna se corresponde a un posible estado de entrada, cada fila a un estado de salida, un valor de 1 indica que esa entrada esta relacionada con esa salida y uno de 0 lo contrario.

Así por ejemplo, el operador AND tiene por entrada un estado de 2 elementos y por salida un estado de 1 elemento, por lo que es una matriz $2^2 \times 2^1 = 4 \times 2$. Para las columnas de los estados de entrada $|00\rangle$, $|01\rangle$ y $|10\rangle$ tenemos un 1 en la fila correspondiente a la salida $|0\rangle$ y un 0 en la correspondiente a $|1\rangle$, siendo lo contrario para la entrada correspondiente al estado $|11\rangle$.

$$\begin{array}{c} \mathbf{00} \quad \mathbf{01} \quad \mathbf{10} \quad \mathbf{11} \\ \mathbf{0} \quad \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ \mathbf{1} \end{array}$$

2.2.6.1. Combinación de operadores

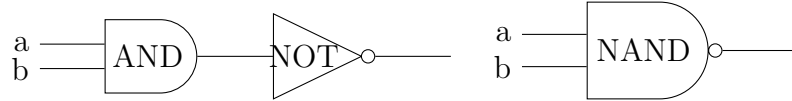
Los operadores pueden ser combinados en serie para una aplicación secuencial sobre el mismo estado o en paralelo para aplicar diferentes operadores a diferentes bits del estado de manera simultanea.

Para combinarlos en serie se aplica una operación después de otra por orden, así si queremos realizar un AND para luego negar su resultado con un NOT, cogemos sus operadores y los multiplicamos en orden por la izquierda.

$$NOT * (AND * |ab\rangle)$$

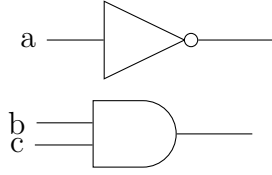
Otra forma es aprovechar las propiedades del álgebra vectorial y crear una operación que efectúe ambas operaciones simultáneamente, en este caso llamada NAND, aprovechando que

$$NOT * (AND * |ab\rangle) = (NOT * AND) * |ab\rangle = NAND * |ab\rangle$$



$$NOT * AND = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} = NAND$$

Para combinar operadores en paralelo, por ejemplo partiendo de un estado de 3 bits ($|abc\rangle$) se quiere negar (NOT) el primer bit (a) y hacer un AND sobre los otros dos (b y c), se utiliza el producto tensorial de los operadores, poniendo más a la izquierda aquellos que afecten a los primeros bits. Esto es, puesto que el estado compuesto se obtendría mediante $a \otimes b \otimes c$, y puesto que a es el primer elemento, NOT será también el primer elemento al combinar los operadores. Por ello, nuestro operador se construiría mediante $NOT \otimes AND$.



$$a \otimes b \otimes c \Rightarrow NOT \otimes AND = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

2.2.7. Operadores reversibles

Ahora bien, no todas las operaciones (puertas) lógicas son aplicables en computación cuántica pues, con excepción de las operaciones de medición, para que una operación sea aplicable en computación cuántica esta debe ser reversible y poder representarse por una matriz hermítica. Por ejemplo la operación AND no es reversible, pues si el resultado es $|1\rangle$ sabemos que la entrada era $|11\rangle$ pero si el resultado es $|0\rangle$ no hay forma de saber, solo con esa información, si la entrada era $|00\rangle$, $|01\rangle$ o $|10\rangle$. En contraste, la operación NOT si es que reversible, pues si el resultado es $|1\rangle$ sabemos que la entrada fue necesariamente $|0\rangle$ y viceversa. En concreto, la operación NOT es su propia inversa, siendo

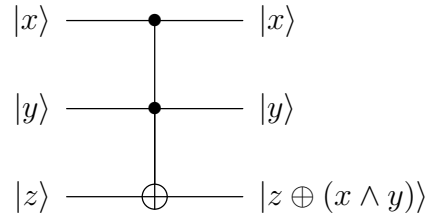
$$NOT * NOT = I_2$$

En 1961 Rolf Landauer[12], tras analizar los procesos computacionales, postuló que era el proceso de pérdida de información (y no su escritura) lo que provocaba la pérdida de energía y por tanto generación de calor. Esto ha sido comprobado recientemente[8], confirmando dicha teoría.

Siguiendo esa linea de pensamiento, en 1972 Charles H. Bennett llego a la conclusión de que, siendo el borrado de información es lo que generaba la pérdida de energía, si un ordenador fuera reversible y no eliminase información, no de-

biera tener consumo energético ni disipar calor. Partiendo de esa idea, empezó a trabajar en circuitos y programas reversibles.

Una de las puertas reversibles más importantes es la puerta de Toffoli



Esta puerta tiene dos líneas de control ($|x\rangle$ y $|y\rangle$) y una línea de información ($|z\rangle$). En caso de que las dos líneas de control se encuentren activadas ($|1\rangle$), la línea de información en negada.

Esta operación es un mapeado $|x, y, z\rangle \mapsto |x, y, z \oplus (x \wedge y)\rangle$ y está representada por la matriz

$$\begin{array}{c}
 \begin{matrix} & 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \end{matrix} \\
 \begin{matrix} 000 \\ 001 \\ 010 \\ 011 \\ 100 \\ 101 \\ 110 \\ 111 \end{matrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}
 \end{array}$$

2.2.8. Ordenador reversible

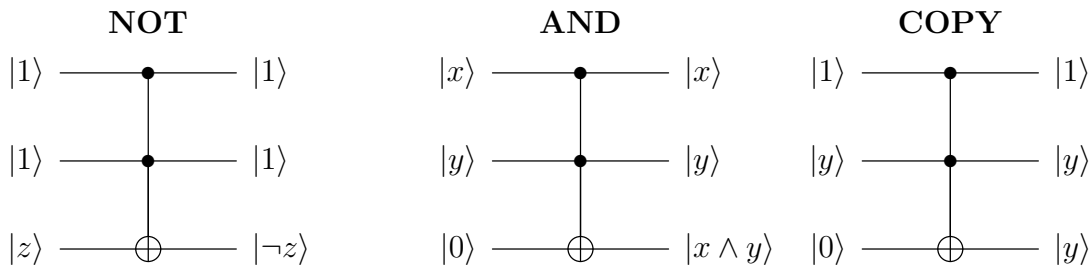
La principal importancia de la puerta de Toffoli es que es universal. Esto significa que cualquier otra puerta lógica puede construirse a base de puertas de Toffoli. Concretamente, solo usando puertas de Toffoli se podría construir un ordenador reversible que, en teoría, no tendría consumo energético ni dispersaría calor.

Para demostrar dicha condición de puerta universal, una forma sencilla es reproducir el comportamiento de otro conjunto de puertas universales ya demostradas. En este caso, las puertas AND y NOT.

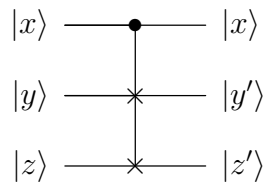
Partiendo de una puerta de Toffoli, si ponemos las entradas $|x\rangle$ y $|y\rangle$ constantes con $|1\rangle$, obtendremos una operación que siempre negará el contenido la entrada $|z\rangle$, lo que es lo mismo que hacer $NOT * |z\rangle$.

Por otro lado, si lo que ponemos constante es la entrada $|z\rangle$ con el valor $|0\rangle$, obtenemos en dicha línea el resultado de $|x \wedge y\rangle$.

Finalmente, para poder construir todas las puertas es importante también poder duplicar un valor concreto. Mediante una puerta de Toffoli con $|x\rangle = |1\rangle$ y $|z\rangle = |0\rangle$, a la salida de la línea de $|z\rangle$ obtenemos una copia de $|y\rangle$.



Otra puerta reversible universal es la puerta de Fredkin, también con 3 entradas $|x, y, z\rangle$ y tres salidas $|x, y', z'\rangle$.

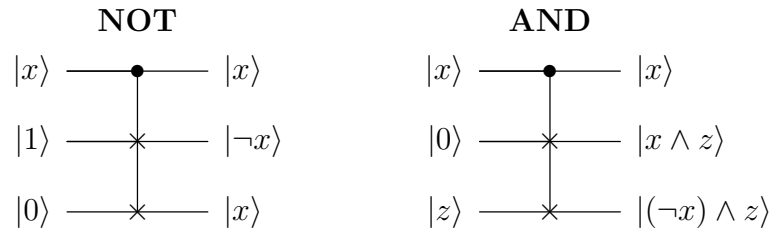


La salida $|x\rangle$ siempre es igual a su entrada, mientras que las salidas $|y', z'\rangle$ dependen del valor de $|x\rangle$. Si es $|0\rangle$ se mantiene la salida de la entrada original, y si es $|1\rangle$ se intercambian las entradas. Así, hay dos opciones $|0, y, z\rangle \mapsto |0, y, z\rangle$ y $|1, y, z\rangle \mapsto |1, z, y\rangle$.

La matriz correspondiente de este operador es

$$\begin{array}{c}
 \begin{array}{cccccccc}
 & 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \\
 000 & \left[\begin{array}{cccccccc}
 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
 \end{array} \right]
 \end{array}
 \end{array}$$

La puerta AND se realizar poniendo $|y\rangle = |0\rangle$ y la puerta NOT poniendo $|y\rangle = |1\rangle$ y $|z\rangle = |0\rangle$.



2.2.9. Operadores cuánticos (puertas cuánticas)

La importancia de los operadores reversibles para la computación cuántica radica en que, para que una operación sea aplicable en computación cuántica (con excepción de la operación de medición), esta ha de ser reversible y poderse representar mediante una matriz unitaria (su inversa es el conjugado de su transpuesta $UU^\dagger = I = U^\dagger U$), y por lo tanto todo operador cuántico se representa con una matriz cuadrada.

De la misma manera que con las puertas lógicas, una puerta cuántica que actué sobre n qubits se representara mediante una matriz de dimensiones $2^n \times 2^n$ y se aplica a un estado cuántico multiplicandole por su izquierda.

Muchos de los operadores cuánticos básicos se definen para un número pequeño de qubits, y luego se combinan para aplicarse a estados de más qubits.

Algunos de estos operadores (o puertas) son:

Puerta de Hadamard (H) es una puerta que actúa sobre un solo qubit y su función es modificar un estado básico ($|0\rangle$ o $|1\rangle$) a un estado de superposición equiprobable.

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$H|0\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}} \quad H|1\rangle = \frac{|0\rangle-|1\rangle}{\sqrt{2}}$$

Puertas de Pauli (X , Y y Z) es un conjunto de 3 puertas cuánticas sobre un solo qubit cada una, las cuales representan una rotación de π radianes en uno de los ejes (X , Y y Z) de la esfera de Bloch.

- La puerta de Pauli- X representa la rotación en el eje X y equivale a una puerta NOT
- La puerta de Pauli- Y representa la rotación en el eje Y
- La puerta de Pauli- Z representa la rotación en el eje Z y supone el caso concreto de la puerta de cambio de fase en la cual $\phi = \pi$

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Puerta de cambio de fase (R_ϕ) es una operación sobre un solo qubit que realiza una rotación de ϕ radianes en el eje Z de la esfera de Bloch, o lo que es lo mismo, mapea el estado $(c_0|0\rangle + c_1|1\rangle)$ al estado $(c_0|0\rangle + e^{i\phi}c_1|1\rangle)$

$$R_\phi = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{bmatrix}$$

Puertas cuánticas controladas actúan sobre 2 o mas qubits y siendo el primer qubit control sobre alguna otra operación sobre el segundo qubit, realizando dicha operación solo si la línea de control contiene un $|1\rangle$.

Toda puerta cuántica de un qubit tiene su análoga puerta cuántica controlada sobre dos qubits. Supongamos un operador cuántico cualquiera U que actúa sobre un qubit

$$U = \begin{bmatrix} x_{00} & x_{01} \\ x_{10} & x_{11} \end{bmatrix}$$

entonces su correspondiente puerta controlada sería

$$C(U) = CU = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & x_{00} & x_{01} \\ 0 & 0 & x_{10} & x_{11} \end{bmatrix}$$

De esta forma, se construyen por ejemplo las puertas $CNOT$, CX , CY , etc.

Puerta de Toffoli es exactamente igual que la puerta de Toffoli descrita para bits, salvo por el hecho de que ahora actúa sobre qubits y no es universal por si misma en el mundo cuántico. Si los dos primeros qubits son $|1\rangle$, entonces aplica un NOT (puerta de Pauli-X) sobre el tercer qubit.

2.2.10. Puertas cuánticas universales

Un conjunto de puertas cuánticas universales, al igual que con las puertas lógicas, es aquel con el cual se pueden construir todo el resto de puertas cuánticas.

Existen varios conjuntos universales de puertas cuánticas, siendo uno por ejemplo el formado por la puerta de Hadamard (H), las puertas de cambio de fase de $\pi/8$ y $\pi/4$ ($R_{\pi/8}$ y $R_{\pi/4}$) y la puerta CNOT.

Ademas, al ser la puerta de Toffoli universal para los operadores lógicos y también existir como puerta cuántica, se demuestra que todo operador lógico puede llevarse a cabo con un conjunto de puertas cuánticas universales.

2.2.11. Procesador cuántico de uso general

Hasta le momento, la mayoría de las operaciones cuánticas implementadas en un circuito precisan de circuitos cuánticos explícitos para la propia operación.

No obstante, se están haciendo esfuerzos en conseguir lo que se denomina un procesador (o computador) cuántico de uso general, lo cual significa que es un sistema hardware que permite programar la aplicación secuencial de un conjunto de operadores sobre un estado cuántico. El punto importante es el de “programar”, pues con ello se permite el realizar múltiples algoritmos con la misma implementación física (hardware) en lugar de tener que tener hardware concreto para cada circuito.

En el caso de que el conjunto de operadores cuya aplicación secuencial permite programar sea un conjunto universal, entonces supone un procesador cuántico universal de uso general en el que se podría implementar cualquier algoritmo cuántico siempre que el número de qubits operables lo permita.

Y ese es el otro punto complicado, siendo los sistemas actuales para tener un estado de qubits estables y manipulables, de momento, muy limitados. Ya no digamos factible económicamente hablando.

Como referencia, durante los últimos meses IBM puso a disposición de la comunidad científica un ordenador cuántico de uso general con una memoria total de 5 qubits.

2.2.12. Algoritmos cuánticos

Un algoritmo cuántico es la transformación controlada de un estado cuántico en otro mediante la aplicación secuencial sobre un estado cuántico de un conjunto de operaciones cuánticas, para finalmente realizar una operación de lectura sobre el estado y obtener así un resultado.

En general, los algoritmos cuánticos tienen un proceso de computación clásica que los engloba, aunque solo sea para recibir los datos de salida.

La parte cuántica de un algoritmo cuántico suele iniciarse superponiendo el estado inicial con una puerta de Hadamard por qubit de forma que quedan en un estado de superposición equidistante, para luego modificarlos de la manera que corresponda y finalmente realizar una medición sobre el estado.

Aquí es donde entra en juego la transformada cuántica de Fourier (QFT). Por ejemplo la parte cuántica del algoritmo de Shor, de forma muy generalizada, tiene por objetivo obtener el periodo de la función $F_a(x) = a^x \bmod P$ siendo P el número a factorizar y a un número tal que $a < P$ y $\text{mcd}(a, P) \neq 1$. Para ello se implementa esa función F_a como un circuito cuántico, se aplica a un estado de entrada superpuesto por operadores de Hadamard y posteriormente se le aplica la transformada de Fourier, con lo que el estado pasa de contener la información de evaluar F_a como amplitudes a tenerla como frecuencias, con lo que es sencillo hallar su periodo.

2.3. Software utilizado

Para el desarrollo del proyecto se han usado varios paquetes de software como el interprete de *python* o VMware.

2.3.1. VMware

VMware[15] es un software de gestión de maquinas virtuales para la ejecución de sistemas operativos.

Aunque la versión completa es de pago, dispone de una versión limitada de uso gratuito. Esta versión limitada es mas que suficiente para los propósitos del proyecto, permitiendo virtualizar y configurar hardware variado de forma muy sencilla.

2.3.2. Qutip

Qutip[5] es una librería gratuita y de código abierto que permite simular el comportamiento de circuitos cuánticos en python apoyándose en las librerías externas de *numpy* y *scipy* y en la librería propia de python de Cython.

Ampliamente utilizada y distribuida bajo una licencia “*BSD[13]*”

Se eligió trabajar con esta librería (cosa que condiciona el resto del proyecto) por lo extendido de su uso, la sencillez de sus interfaces, la documentación que ofrece y su comunidad.

2.3.3. Python

Python[10] (nombre dado en referencia a los Monty Python) es un lenguaje de programación de alto nivel, de uso general, interpretado, multiplataforma, multiparadigma y de tipado dinámico, el cual hace especial hincapié en la facilidad de generar código y en la legibilidad del mismo.

Existen varias implementaciones de este lenguaje, pero la principal implementación se llama *CPython* y esta programada en C. Compila el código *python* a una representación intermedia (*bytecode*), el cual luego es ejecutado en la máquina virtual. Esta implementación es gratuita y de código abierto bajo “Python Software Foundation License”, la cual es muy similar a GPL con la principal diferencia de que no es *copyleft* (permite modificar el código fuente sin necesidad de que el producto modificado se distribuya con el código fuente).

Las librerías estándar son amplias y están en continuo crecimiento al contar con una activa comunidad y poderse implementar ampliaciones o personalizar los módulos ya existentes tanto en el propio lenguaje como en C.

Una de las principales características distintivas de este lenguaje es que utiliza la indentación por espacios en blanco para identificar los bloques de código, obligando con ello a dar una estructura más fácil de ver a simple vista.

El 13 de febrero de 2009 se publicó la versión 3.0 del lenguaje, siendo la implementación más reciente la 3.5.2 (publicada el 27 de Junio de 2016) y la usada en el proyecto (para asegurar compatibilidad con otras librerías) la 3.4.

2.3.4. SciPy

SciPy[6] es una librería de código abierto para *python* que busca simplificar el realizar cálculos complejos mediante módulos para (entre otros) álgebra lineal, integración, interpolación, procesamiento de señales, etc. . .

Esta librería es ampliamente utilizada y contiene otras librerías como por ejemplo *numpi*.

La licencia bajo la que se distribuye esta librería es “*BSD-new license*[13]”.

2.3.5. Numpy

Numpy[9] es una librería para *Python* que agrega un mayor soporte para arrays y matrices multi-dimensionales de gran tamaño, junto con una gran librería de alto nivel para operar con dichos arrays y matrices.

Esta distribuido bajo una licencia “*BSD-new license*[13]”.

Planificación

Contenidos

3.1. Fases	33
3.2. Infraestructura	34
3.3. Estimación	35
3.4. Estimación del coste	36

En este apartado se explica las fases en las que se ha estructurado el presente proyecto, así como su estimación y planificación.

3.1. Fases

El trabajo a realizar se ha dividido en varias fases:

1. Estudio de la base matemática necesaria (álgebra vectorial y números complejos)
2. Estudio de las bases de la computación cuántica
3. Análisis y comprensión del algoritmo de la transformada cuántica de Fourier
4. Estudio de la implementación existente en la librería escogida

5. Diseño, implementación y prueba de una simulación básica
6. Medición del consumo de memoria y tiempo de ejecución de la simulación
7. Estudio, implementación y medición de optimizaciones en el consumo de memoria
8. Estudio, implementación y medición de optimizaciones en tiempo de ejecución

Estas fases se han estructurado en cuatro etapas: estudio previo, etapa inicial, refinamiento de memoria y refinamiento de tiempo de ejecución.

Durante la etapa de estudio previo (fases 1, 2 y 3), se ha invertido esfuerzo en adquirir los conocimientos necesarios para el proyecto. Dichos conocimientos son una base matemática en el uso de números complejos y álgebra vectorial de números complejos, así como los conocimientos necesarios sobre computación cuántica.

La etapa inicial (fases 4 y 5) supone una primera implementación sencilla del algoritmo y un estudio del consumo de recursos que conlleva.

Las etapas de refinamiento (fases 6 y 7) se basan en varias iteraciones de mejora, en cada una de las cuales se plantea una posible mejora, se implementa, se mide el consumo de recursos y se valora con el fin de conseguir una versión final lo más óptima posible fruto de una o varias de las mejoras probadas.

3.2. Infraestructura

El código de este proyecto se ha ejecutado mediante el uso de maquinas virtuales VMware [15] con Ubuntu como sistema operativo sobre un ordenador portátil de uso personal.

El ordenador base es un ACER N56V:

- Intel Core i7 - 3610QM a 2'3GHz con 4 núcleos físicos y 8 lógicos.
-

- 8 GB de memoria RAM.
- Nvidia GeForce 630M de 2GB
- Windows 10

El hardware virtualizado es:

- Procesador de 4 núcleos.
- 2 GB de memoria RAM.
- Ubuntu 14.14

3.3. Estimación

La estimación temporal del proyecto se ha hecho centrada en el esfuerzo estimado y realizado, en lugar de en las fechas de inicio y fin. Por ello las fechas mostradas en los diagramas, salvo al de inicio, no son indicativas.

El esfuerzo total estimado para el proyecto ha sido de 291 horas. un diagrama general puede verse en la figura 3.1.

Las fases de estudio previo y la etapa inicial (la implementación base se ha separado en el diagrama de Gantt de la etapa inicial) se pueden ver en la figura 3.2

Puesto que se ha seguido un desarrollo iterativo, y cada iteración esta bien definida y acotada, se ha realizado una estimación de esfuerzo general para las iteraciones que comprende

- 3 horas para el diseño de la iteración sobre el diseño general
 - 7 horas para la implementación y pruebas
 - 5 horas para mediciones y preparación de los datos
-

- 5 horas para analizar los resultados obtenidos

Las estimaciones de las diferentes iteraciones pueden verse en los diagramas de las figuras 3.3 y 3.4

La redacción de la memoria se ha planificado para cada etapa por separado, estimándose inicialmente en 108 horas de esfuerzo en total. Un desglose mas concreto de las estimaciones para la memoria se puede ver en el diagrama de la figura 3.5

Queda fuera de la planificación el tiempo dedicado a preparar la defensa y la propia defensa del proyecto.

- 3 horas para el diseño de la iteración sobre el diseño general
- 7 horas para la implementación y pruebas
- 5 horas para mediciones y preparación de los datos
- 5 horas para analizar los resultados obtenidos

3.4. Estimación del coste

En lo referente al coste del proyecto, se le va a aplicar un coste total por hora de analista programador junior de 30€/hora, por lo tanto, el coste estimado inicialmente para el proyecto es de

$$291horas \times 30\frac{\text{€}}{\text{hora}} = 8730\text{€}$$

donde se incluye seguridad social, mantenimiento del equipo, etc...

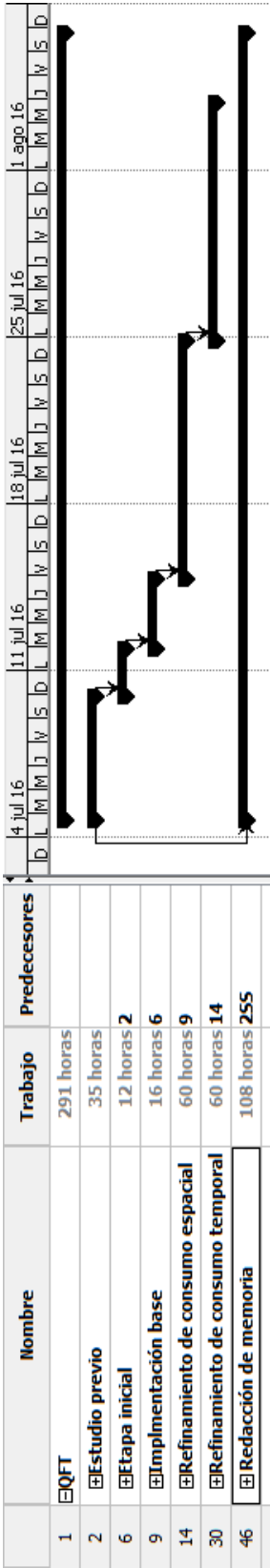


Figura 3.1: Diagrama de Gantt: General

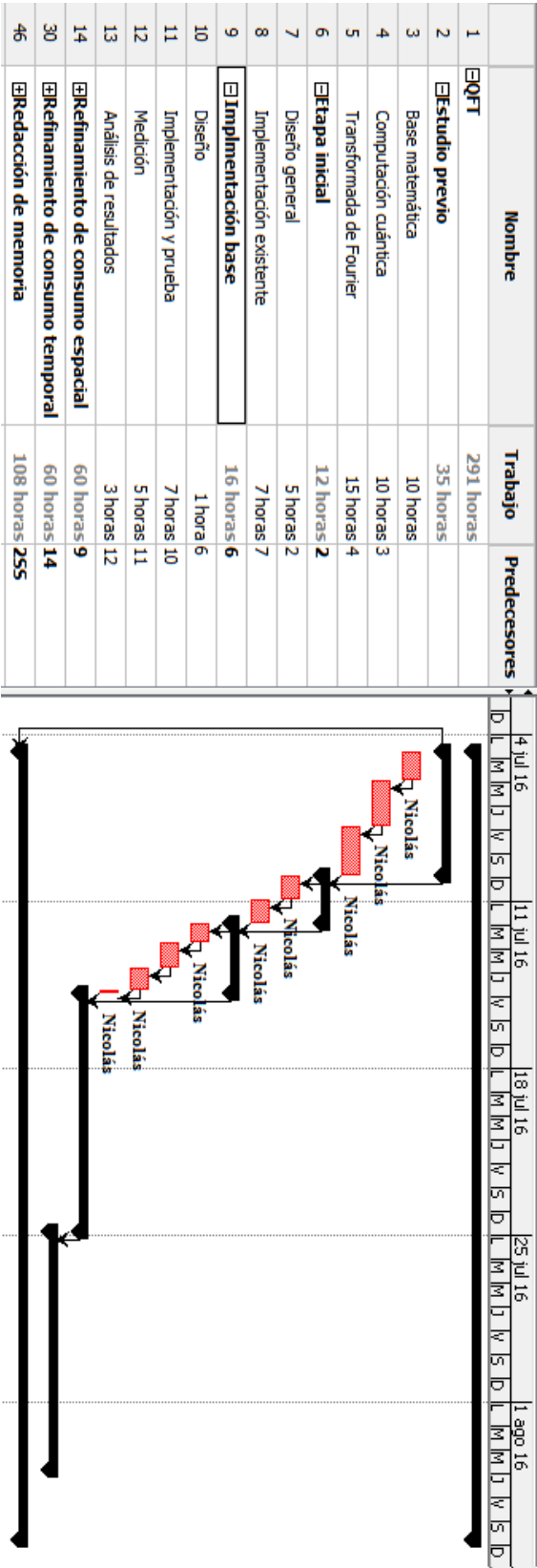


Figura 3.2: Diagrama de Gantt: Etapas previas

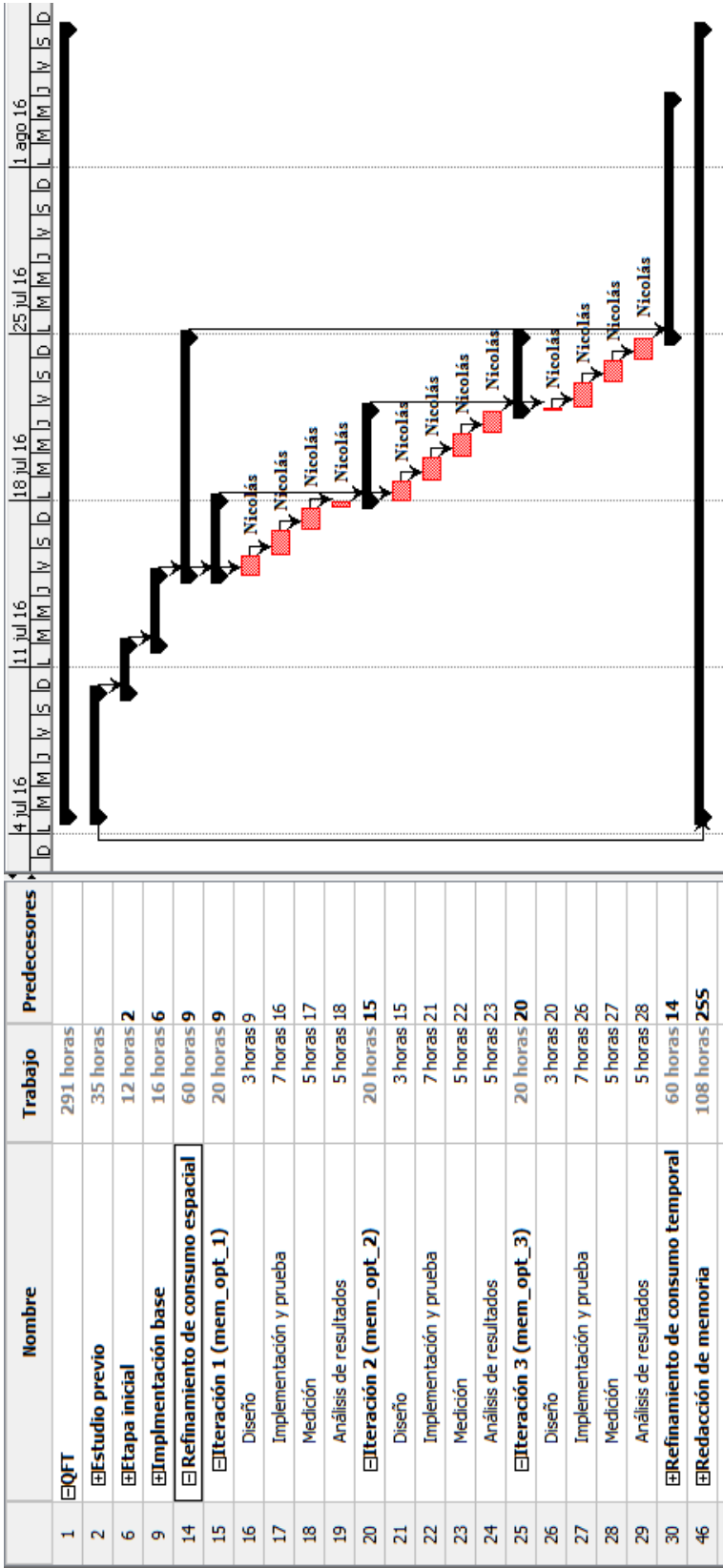


Figura 3.3: Diagrama de Gantt: Etapa de refinamiento del consumo de memoria



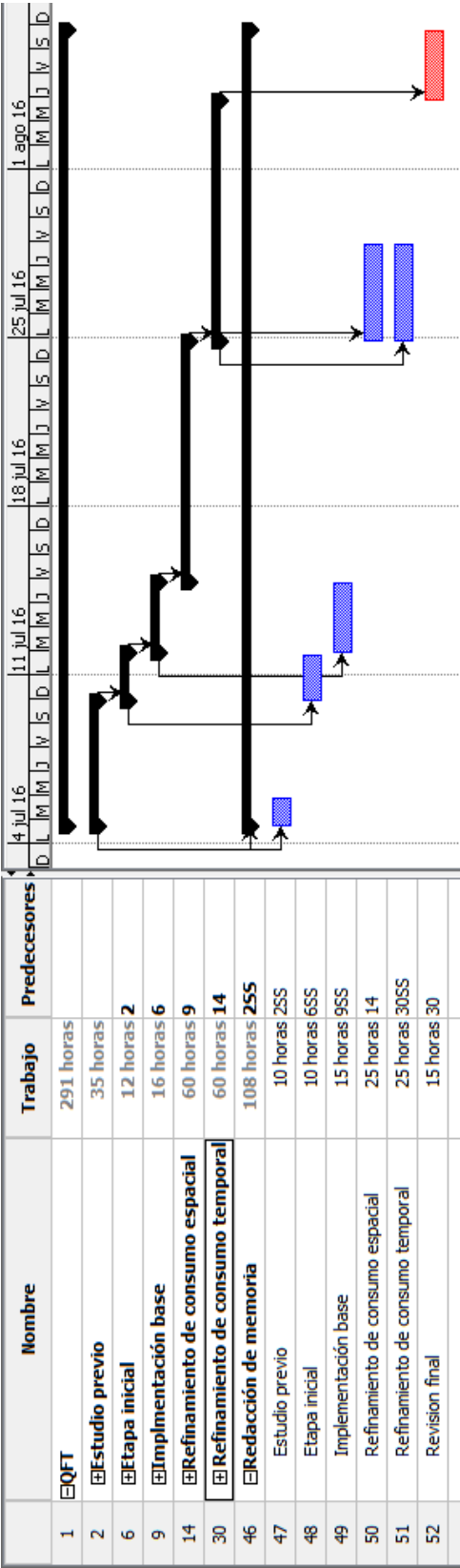


Figura 3.5: Diagrama de Gantt: Redacción de memoria

Transformada de Fourier

Contenidos

4.1. Series de Fourier	44
4.2. Transformada de Fourier continua (FT)	45
4.3. Transformada de Fourier discreta (DFT)	45
4.4. Transformada cuántica de Fourier (QFT)	47
4.5. Circuito de la transformada cuántica de Fourier	50

La transformada de Fourier, llamada así por Jean-Baptiste Joseph Fourier, es una transformación matemática que descompone una función (la cual para unificar nomenclatura se dice que esta en el dominio del tiempo) al conjunto de frecuencias de las que se compone. La transformada de Fourier de una función se refiere tanto a la representación en el dominio de la frecuencia de la función como a la operación matemática que da lugar a dicha representación.

Una de sus propiedad más importantes que es una transformación matemática reversible, pudiendo recuperarse la función original de antes de la transformación.

Esta transformación es útil para el tratamiento de datos pues las operaciones realizables en un dominio tiene su correspondiente operación equivalente en el dominio de la frecuencia, siendo a veces una mucho mas sencilla que la otra (por ejemplo, la convolución en el dominio del tiempo se corresponde con multiplicar en el de la frecuencia). Por ejemplo, si sabemos que un canal introduce un ligero

ruido en una señal y se quiere eliminar, para hacerlo en el dominio del tiempo habría que suavizar la función a lo largo de todo el dominio, pero en el dominio de la frecuencia es tan sencillo como eliminar los valores asociados a las frecuencias que estén por encima de un umbral.

4.1. Series de Fourier

La transformada de Fourier surge de las series de Fourier, llamadas así también debido a Jean-Baptiste Joseph Fourier, el cual fue el matemático que las desarrolló y publicó [3].

Una serie de Fourier es una forma de aproximar una función periódica como la suma de funciones oscilatorias simples (en la figura 4.1 se muestra la aproximación de un pulso cuadrado).

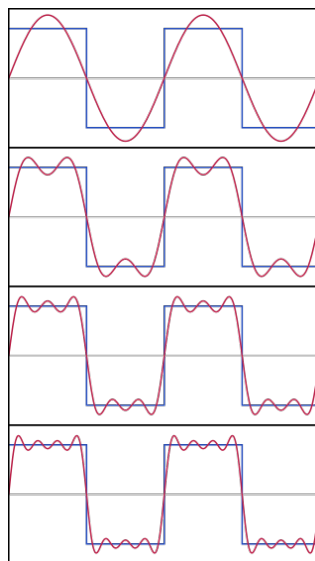


Figura 4.1: Aproximación por serie de Fourier

Más formalmente, una serie de Fourier es una serie infinita que converge puntualmente a una función periódica y continua a trozos (o por partes) con la forma

$$f(t) \sim \frac{a_0}{2} + \sum_{n=1}^{\infty} [a_n \cos(2n\pi f_0 t) + b_n \sin(2n\pi f_0 t)]$$

donde $f_0 = \frac{1}{T}$ es la frecuencia de la función f y a_n y b_n son los llamados coeficientes de Fourier

$$a_n = \frac{2}{T} \int_{-T/2}^{T/2} f(t) \cos\left(\frac{2n\pi}{T}t\right) dt \quad b_n = \frac{2}{T} \int_{-T/2}^{T/2} f(t) \sin\left(\frac{2n\pi}{T}t\right) dt$$

Aplicando la identidad de Euler ($e^{i\pi} = \cos \pi + i \sin \pi$) se puede simplificar de forma que

$$f(t) \sim \sum_{n=-\infty}^{\infty} c_n e^{2n\pi i f_0 t}$$

$$c_n = f_0 \int_{-T/2}^{T/2} f(t) e^{-2n\pi i f_0 t} dt$$

4.2. Transformada de Fourier continua (FT)

A partir de la serie de Fourier se define la transformada de Fourier continua como una integral de exponenciales complejas tal que

$$F(\xi) = \int_{-\infty}^{+\infty} f(x) e^{-2\pi i x \xi} dx, \forall \xi \in \mathbb{R}$$

donde x es la variable independiente en el dominio del tiempo, ξ la variable independiente en el dominio de la frecuencia, f es la función en el dominio del tiempo y F la función en el dominio de la frecuencia.

Esta transformación es invertible, siendo la transformada inversa

$$f(x) = \int_{-\infty}^{+\infty} F(\xi) e^{2\pi i x \xi} d\xi, \forall x \in \mathbb{R}$$

4.3. Transformada de Fourier discreta (DFT)

Puesto que el cálculo de transformada de Fourier continua es relativamente costoso, para aquellos casos en que la función de entrada sea una secuencia finita

de números reales o complejos se usa una generalización llamada transformada de Fourier discreta, la cual se define como una transformada de Fourier para análisis de señales de tiempo discreto y dominio finito, como suele ser la información almacenada en soportes digitales.

Esta transformación (F) convierte una secuencia finita X de N números complejos en otra secuencia finita Y de números complejos de igual longitud

$$X = [x_0, x_1, \dots, x_{N-1}] \mapsto Y = [y_0, y_1, \dots, y_{N-1}] \quad \backslash \quad Y = F(X)$$

mediante la fórmula

$$y_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \quad k = 0, 1, \dots, N-1$$

y su inversa ($Y \mapsto X$) mediante

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} y_k e^{\frac{2\pi i}{N} kn} \quad n = 0, 1, \dots, N-1$$

Otra forma de interpretarla es considerando los conjuntos X e Y como dos vectores columna y expresar la DFT como una matriz de Vandermonde (matriz que presenta una progresión geométrica en cada fila)

$$F[j, k] = e^{(\frac{2\pi i}{N})jk} = w_N^{jk} \quad j, k = 0, 1, \dots, N-1 \quad \backslash \quad Y = FX$$

y su inversa como

$$F^{-1} = \frac{1}{N} F^* \quad \backslash \quad X = F^{-1}Y$$

Avanzando un paso más es posible, mediante normalización, hacer que sea una transformación unitaria $U = \frac{1}{\sqrt{N}}F$ y $U^{-1} = U^*$ tal que $|\det(U)| = 1$

4.4. Transformada cuántica de Fourier (QFT)

Puesto que la transformada discreta actúa sobre un vector discreto y finito de números reales o complejos, y ya que un estado cuántico se representa mediante un vector discreto y finito de números complejos, esta transformación debiera poderse aplicar a un estado cuántico siempre que se pueda implementar en un circuito cuántico.

Así, tenemos una transformación desde un estado de n qubits $|\psi\rangle$ representado por un vector de $N = 2^n$ números complejos, a otro $F|\psi\rangle$ que se corresponde con el primero, pero en el dominio de la frecuencia

$$|\psi\rangle = \sum_{j=0}^{N-1} a_j |j\rangle \mapsto F|\psi\rangle = \sum_{k=0}^{N-1} b_k |k\rangle$$

siendo (de forma análoga a la transformada discreta de Fourier)

$$b_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} a_j w^{jk} \quad \wedge \quad w = e^{\frac{2\pi i}{N}}$$

por simplicidad se va a utilizar $w = w_N$ dejando implícito el término N para cada caso, con el fin de no recargar tanto las ecuaciones.

El término $\frac{1}{\sqrt{N}}$ sirve para normalizar el vector, pues un estado cuántico ha de ser un vector normalizado.

Por ejemplo, para el caso de un estado de 2 qubits ($N = 4$)

$$|\psi\rangle = [a_0, a_1, a_2, a_3]^T = a_0|00\rangle + a_1|01\rangle + a_2|10\rangle + a_3|11\rangle$$

$$F|\psi\rangle = [b_0, b_1, b_2, b_3]^T = b_0|00\rangle + b_1|01\rangle + b_2|10\rangle + b_3|11\rangle$$

calculando los valores del estado $F|\psi\rangle$ por separado

$$b_0 = \frac{1}{\sqrt{4}} \sum_{j=0}^{4-1} a_j w^{0j} = \frac{1}{2} \sum_{j=0}^3 a_j w^0 = \frac{1}{2} [a_0 w^0 + a_1 w^0 + a_2 w^0 + a_3 w^0]$$

$$b_1 = \frac{1}{\sqrt{4}} \sum_{j=0}^{4-1} a_j w^{1j} = \frac{1}{2} \sum_{j=0}^3 a_j w^j = \frac{1}{2} [a_0 w^0 + a_1 w^1 + a_2 w^2 + a_3 w^3]$$

$$b_2 = \frac{1}{\sqrt{4}} \sum_{j=0}^{4-1} a_j w^{2j} = \frac{1}{2} \sum_{j=0}^3 a_j w^{2j} = \frac{1}{2} [a_0 w^0 + a_1 w^2 + a_2 w^4 + a_3 w^6]$$

$$b_3 = \frac{1}{\sqrt{4}} \sum_{j=0}^{4-1} a_j w^{3j} = \frac{1}{2} \sum_{j=0}^3 a_j w^{3j} = \frac{1}{2} [a_0 w^0 + a_1 w^3 + a_2 w^6 + a_3 w^9]$$

es posible representar esta operación mediante matrices, así tenemos una matriz operador F_4 que se aplica al vector que representa el estado $|\psi\rangle$.

$$F_4 = \frac{1}{\sqrt{4}} \begin{bmatrix} w^0 & w^0 & w^0 & w^0 \\ w^0 & w^1 & w^2 & w^3 \\ w^0 & w^2 & w^4 & w^6 \\ w^0 & w^3 & w^6 & w^9 \end{bmatrix} = \frac{1}{\sqrt{4}} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & w^1 & w^2 & w^3 \\ 1 & w^2 & w^4 & w^6 \\ 1 & w^3 & w^6 & w^9 \end{bmatrix}$$

y gracias a que $w = e^{\frac{2\pi i}{4}} = e^{\frac{\pi i}{2}}$, tenemos que para el caso de 2 qubits $w^4 = e^{2\pi} = 1$ y en consecuencia, por ejemplo, $w^6 = w^4 w^2 = 1 w^2 = w^2$, lo que nos permite simplificar la matriz operador a

$$F_4 = \frac{1}{\sqrt{4}} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & w^1 & w^2 & w^3 \\ 1 & w^2 & 1 & w^2 \\ 1 & w^3 & w^2 & w^1 \end{bmatrix}$$

Por lo tanto

$$F_4|\psi\rangle = \frac{1}{\sqrt{4}} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & w^1 & w^2 & w^3 \\ 1 & w^2 & 1 & w^2 \\ 1 & w^3 & w^2 & w^1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Esta matriz operador es generalizable para cualquier N

$$F_N = \frac{1}{\sqrt{N}} \begin{bmatrix} w^0 & w^0 & w^0 & w^0 & \dots & w^0 \\ w^0 & w^1 & w^2 & w^3 & \dots & w^{N-1} \\ w^0 & w^2 & w^4 & w^6 & \dots & w^{2(N-1)} \\ w^0 & w^3 & w^6 & w^9 & \dots & w^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w^0 & w^{N-1} & w^{2(N-1)} & w^{3(N-1)} & \dots & w^{(N-1)(N-1)} \end{bmatrix}$$

y ademas se puede comprobar que es unitaria y por lo tanto reversible (siendo su adjunta su propia inversa) pues

$$FF^\dagger = I = F^\dagger F$$

con lo que cumple dicha condición para ser aplicable en computación cuántica.

Un caso que puede llevar a confusión es el caso de la transformada cuántica de Fourier para 1 qubit ($w = w_2 = e^{\pi i}$), pues su operador es igual al de una puerta de Hadamard

$$F_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} w^0 & w^0 \\ w^0 & w^1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = H_1$$

lo cual podría llevar a pensar que para obtener el operador para la QFT sobre 2 qubits, bastaría con hacer el producto tensorial del operador para un qubit consigo mismo como se haría para obtener una puerta de Hadamard para dos qubits, no obstante esto no sería correcto pues

$$H_2 = H_1 \otimes H_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \frac{1}{\sqrt{4}} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

pero en cambio el operador de la QFT para 2 qubits (F_4) siendo

$$w = w_4 = e^{\frac{2\pi i}{4}} = e^{\frac{\pi i}{2}}$$

y por lo tanto

$$w^0 = 1 \qquad w^1 = i \qquad w^2 = -1 \qquad w^3 = -i$$

no es igual a la puerta de Hadamard para 2 qubits

$$F_4 = \frac{1}{\sqrt{4}} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & w^1 & w^2 & w^3 \\ 1 & w^2 & 1 & w^2 \\ 1 & w^3 & w^2 & w^1 \end{bmatrix} = \frac{1}{\sqrt{4}} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \neq \frac{1}{\sqrt{4}} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} = H_2$$

pues la transformada cuántica de Fourier no puede construirse mediante el producto tensorial de transformadas parciales. Como se explica en la sección siguiente, el circuito de la transformada cuántica de Fourier para n qubits se construye a partir del circuito para $n - 1$ qubits añadiendo n puertas lógicas extra (1 puerta de Hadamard de 1 qubit y $n - 1$ puertas de desplazamiento de fase).

No obstante, en el caso de aplicarse a 1 solo qubit, la transformada cuántica de Fourier y la puerta de Hadamard son equivalente.

4.5. Circuito de la transformada cuántica de Fourier

Teniendo un operador representable mediante una matriz unitaria, falta ver si esta operación es implementable en un circuito cuántico.

Para implementar una operación sobre n qubits en un circuito cuántico de forma sencilla, es importante conseguir una representación de la operación como el producto tensor de n qubits-únicos, de forma que pueda descomponerse en la aplicación secuencial de un conjunto de puertas cuánticas sobre cada qubit individual.

Por comodidad en esta sección se va a usar la notación

$$[0.x_1\dots x_n] = \sum_{k=1}^n x_k 2^{-k} = \frac{x_1}{2} + \frac{x_2}{2^2} + \dots + \frac{x_n}{2^n}$$

En el caso de la transformada cuántica de Fourier, esta puede expresarse

$$F : |x_1, \dots, x_n\rangle \mapsto \frac{1}{\sqrt{2^n}}(|0\rangle + e^{2\pi i[0.x_n]}|1\rangle) \otimes \dots \otimes (|0\rangle + e^{2\pi i[0.x_1x_2\dots x_n]}|1\rangle)$$

siendo por lo tanto el caso de dos qubits

$$F|x_1x_2\rangle = \frac{1}{\sqrt{4}}(|0\rangle + e^{2\pi i[0.x_2]}|1\rangle) \otimes (|0\rangle + e^{2\pi i[0.x_1x_2]}|1\rangle)$$

y el caso de tres qubits

$$F|x_1x_2x_3\rangle = \frac{1}{\sqrt{8}}(|0\rangle + e^{2\pi i[0.x_3]}|1\rangle) \otimes (|0\rangle + e^{2\pi i[0.x_2x_3]}|1\rangle) \otimes (|0\rangle + e^{2\pi i[0.x_1x_2x_3]}|1\rangle)$$

Por lo tanto

$$F|x_1, \dots, x_n\rangle = |y_1, \dots, y_n\rangle$$

tal que

$$\begin{aligned} y_1 &= \frac{1}{\sqrt{2^n}}(|0\rangle + e^{2\pi i[0.x_n]}|1\rangle) \\ y_2 &= (|0\rangle + e^{2\pi i[0.x_{n-1}x_n]}|1\rangle) \\ &\vdots \\ y_n &= (|0\rangle + e^{2\pi i[0.x_1x_2\dots x_n]}|1\rangle) \end{aligned}$$

($\frac{1}{\sqrt{2^n}}$ solo multiplica a uno de los qubits pues $\alpha(B \otimes C) = (\alpha B) \otimes C = B \otimes (\alpha C)$ siendo α un escalar y B y C matrices).

Puesto que puede factorizarse como el producto tensor de n qubits, ahora queda ver si las operaciones sobre cada qubit son implementables.

En este caso cada una de las operaciones de los qubits puede ser implementada eficientemente usando una puerta Hadamard (H) y puertas controladas de desplazamiento de fase (R_ϕ). El primer término (x_n) requiere una puerta Hadamard, el segundo término requiere una puerta Hadamard y una puerta controlada

de desplazamiento de fase, el tercer término requiere una puerta de Hadamard y dos de desplazamiento de fase, y así consecutivamente añadiendo una puerta adicional de desplazamiento de fase para cada término respecto al anterior.

Sumando las puertas para cada término

$$1 + 2 + \dots + n = n(n+1)/2 \Rightarrow O(n^2)$$

puertas, siendo n el número de qubits. Esto es un incremento cuadrático de puertas cuánticas necesarias respecto al número de qubits

En la figura 4.2 se muestra el circuito de la QFT para 3 qubits ($n = 3 \Rightarrow 3(3+1)/2 = 6$ puertas cuánticas). En el puede observarse una estructura recursiva en la cual la línea de $|y_1\rangle$ (correspondiente a $|x_n\rangle$) se construye a partir de la salida de aplicar la QFT a los qubits $|x_1x_2\rangle$.

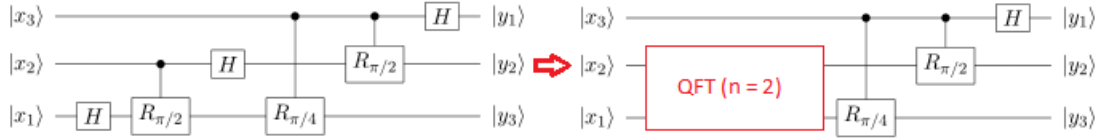


Figura 4.2: Circuito cuántico de la QFT para 3 qubits

En la figura 4.3 se muestra el circuito de la QFT general para un número n de qubits.

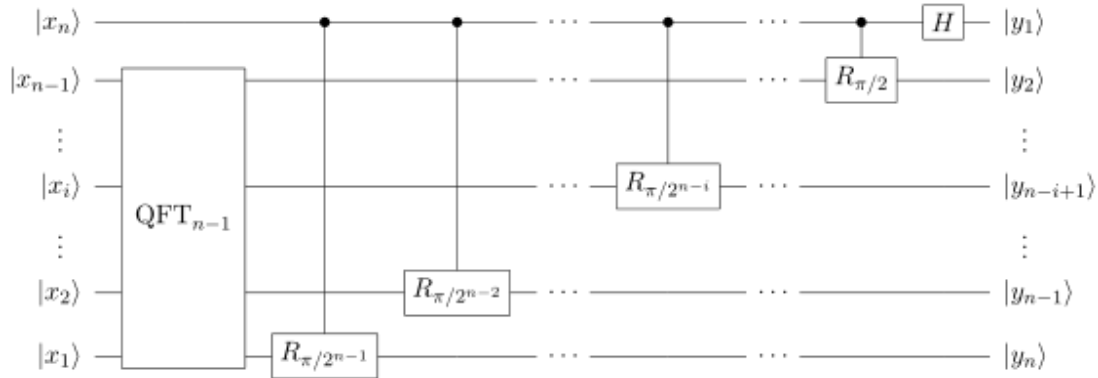


Figura 4.3: Circuito cuántico de la QFT para n qubits

Capítulo 5

Etapa inicial

Contenidos

5.1. Diseño de las mediciones	54
5.2. Validación	56
5.3. Diseño de clases	56
5.4. Implementación de <i>qutip</i>	59

La librería de *qutip* ofrece una función que devuelve el operador para simular la transformada cuántica de Fourier, el cual se construye a partir de las puertas individuales del circuito cuántico y tiene como limitación un máximo de 12 qubits.

El primer paso del desarrollo consiste en un estudio del consumo de recursos por parte de esta implementación con el fin de poder comparar luego los resultados de las implementaciones realizadas.

Con ello, este proyecto tiene por objetivo final la creación de una función que ofrezca la operación de la transformada cuántica de Fourier para estados de mayor tamaño, generando el operador mediante la aplicación directa de la formula matemática en lugar de mediante la secuencia de puertas lógicas de su implementación física. Para ello en un inicio se propone una implementación base sencilla, la cual se va refinando para optimizar su consumo de memoria y tiempo de ejecución, siendo el objetivo último el conseguir poder operar el máximo número de qubits en el menor tiempo posible.

Las diferentes implementaciones realizadas (mostradas en la figura 5.1) son:

base_impl: Implementación de partida del algoritmo sin ninguna optimización.

mem_opt_1: Implementación que parte de *base_impl* y trata de optimizar el uso de memoria generando y aplicando el operador fila a fila.

mem_opt_2: Implementación que parte de *base_impl* y elimina la conversión a objeto de *qutip* del operador antes de operarlo, evitando una tener una matriz de grandes dimensiones como variable.

mem_opt_3: Implementación conjunta que une las optimizaciones de *mem_opt_1* y *mem_opt_2*.

time_opt_1: Implementación que partiendo de *mem_opt_3* pre-calcula los posibles términos del operador.

time_opt_2: Implementación que, con *mem_opt_3* como base, paraleliza la generación y aplicación del operador.

time_opt_3: Implementación que unifica *time_opt_1* y *time_opt_2*.

5.1. Diseño de las mediciones

Para cada implementación, excepto la de *qutip*, se ha llevado a cabo un estudio, tanto teórico como práctico, para el coste espacial y temporal. En concreto el coste espacial se ha estudiado desde el punto de vista teórico para calcular un coste mínimo y estimar una complejidad espacial y desde el punto de vista práctico para calcular los límites reales del algoritmo. En lo tocante al coste temporal, en el estudio teórico se ha mirado la complejidad ciclomática y de forma empírica tomando el tiempo interno del sistema antes y después de ejecutar el algoritmo para calcular su diferencia.

Las mediciones empíricas se han realizado con 10 ejecuciones del algoritmo en las cuales se ha medido el consumo de memoria y el tiempo utilizado. Sobre el

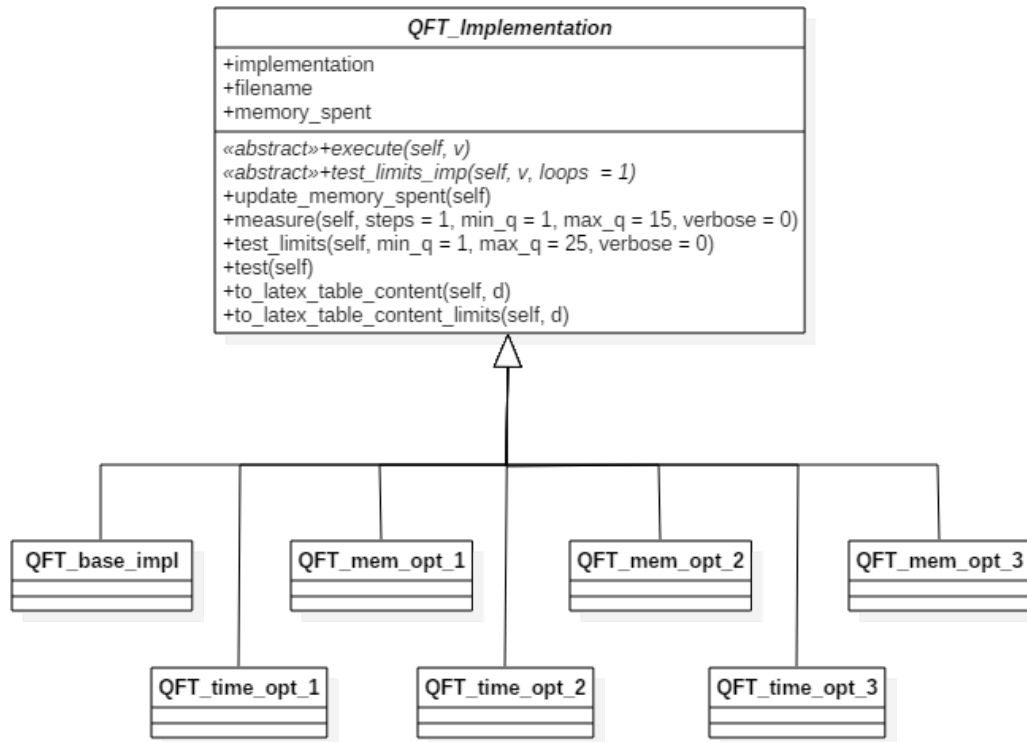


Figura 5.1: Diagrama de clases: Implementaciones

consumo de memoria se ha guardado el valor máximo (pues a la hora de saber si es ejecutable en un sistema lo relevante es el máximo consumo) y para el tiempo se ha calculado la media y varianza (truncadas a 5 decimales). Estas mediciones se han llevado a cabo para estados de diferentes tamaños con el número de qubits (variando entre 1 y 15 pues para más de 15 el tiempo de ejecución se dispara) y se han recogido en tablas comparativas. En caso de que la memoria del sistema no fuera suficiente para ejecutar el algoritmo se muestra mediante “x” en las casillas de memoria y tiempo correspondientes.

Para comprobar el consumo real de memoria por parte únicamente del proceso se ha usado la llamada al sistema proporcionada por la propia librería de python3.4 (`resource.getrusage(resource.RUSAGE_SELF).ru_maxrss`).

En cada implementación se incluye una tabla con los resultados de sus mediciones junto con una tabla comparativa de consumo de memoria y otra de tiempo de ejecución con la implementación que se usó como base en esa iteración.

Ademas, al final de la etapa de optimización de memoria y de la de tiempo, se incluyen tablas comparativas de toda la etapa.

Como final, en el capitulo de resultados (10) se exponen todos los resultados obtenidos en dos tablas comparativas (una para memoria y otra para tiempo).

5.2. Validación

Para la validación de las implementaciones, se realiza una prueba funcional de caja negra comparando los resultados de aplicar el operador de la transformada ofrecida por *qutip* con los obtenidos durante la ejecución del algoritmo para un mismo estado de entrada aleatorio. Puesto que de una implementación a otra solo se esperan cambios no funcionales, esta prueba es válida para todas las implementaciones. Como inciso, al realizar la prueba se vio que los resultados no coincidían. Después de investigarlo, se descubrió que el problema radicaba en el redondeo, pues por ejemplo si en una posición del operador tiene que haber el valor $0 + 5j$ el valor que aparecía en los operadores obtenidos (tanto el del algoritmo como el de *qutip*) era del estilo de $3,4 * 10^{-17} + 0,5j$, que aunque muy cercano al valor esperado, no es el mismo. Por ello, se decidió redondear al 5 decimal los valores del operador, para evitar esos problemas de redondeo.

5.3. Diseño de clases

En lo referente al diseño de clases, se han tomado ciertas decisiones comunes a todas las implementaciones.

Durante todo el proyecto se ha limitado las entradas a estados cuánticos con un número de filas potencia de 2 asumiendo que esta operación se aplicara a

estados de n qubits, los cuales se representan con vectores de 2^n filas. Con esto, se puede utilizar la versión simplificada de la transformada.

Se ha hecho que todas las implementaciones estudiadas cumplan una interfaz común con un método `execute(self, v)` donde `v` es el estado de entrada y el valor de retorno es el resultado de aplicar la transformada cuántica de Fourier a dicho estado.

Para simplificar el estudio de las implementaciones se ha implementado una superclase con métodos destinados a este fin:

- `measure(self, steps=1, min_q=1, max_q=15, verbose=0)`

El objetivo de este método es sacar datos sobre la ejecución del algoritmo a fin de comparar con otras implementaciones. Para ello lo ejecuta `steps` número de veces para estados de entrada aleatorios con tamaño entre `min_q` y `max_q`, tomando mediciones de consumo de tiempo y memoria para luego exportarlas a un fichero bajo una estructura de tabla de LaTeX. En caso de que `steps` sea mayor que 1, se calcula la media y varianza dle coste temporal y se guarda el máximo consumo de memoria.

- `test_limits (self, min_q=1, max_q=25, verbose=0)`

Este método trata de encontrar empíricamente el mayor estado operable por la implementación concreta ejecutando el algoritmo para estados de tamaño entre `min_q` y `max_q` y mide su consumo de memoria hasta que se produzca un error de memoria, para luego volcar los consumos de memoria obtenidos en un fichero con estructura de tabla de LaTeX.

En caso de que la implementación concreta alcance el máximo consumo de memoria sin necesidad de ejecutar el algoritmo hasta le final, a fin de hacer factible la medición en un tiempo razonable y puesto que aquí solo interesa el consumo de memoria, este método llama a una implementación alternativa que ejecuta el mismo algoritmo hasta que alcanza el máximo de memoria que ha de usar, reduciendo el tiempo necesario para realizar las mediciones drásticamente.

- `test (self)`

Con este método se realizan las pruebas funcionales de las implementaciones para su verificación. Compara el resultado de aplicar al mismo estado aleatorio la implementación de *qutip* (redondeando al 5º decimal) y la implementación a probar. Esto lo hace para estados de tamaño entre 1 y 12 (inclusivos) pues la transformación de *qutip* tiene 12 como límite superior.

- `update_memory_spent(self)`

Realiza una medición de la memoria consumida por el proceso, y si es superior al máximo registrado, actualiza el máximo.

5.4. Implementación de *qutip*

Con el fin de poder comparar futuros resultados, se llevó a cabo un estudio empírico del consumo espacial y temporal de la implementación ofrecida por *qutip* para el operador de la transformada cuántica de Fourier.

Para ello, se implementó un pequeño código que llama a la función ofrecida por la librería para obtener el operador y lo aplica a un estado cuántico. Se realizaron mediciones de tiempo y de memoria según lo establecido para obtener el consumo de esta implementación. Puesto que es parte de la librería de referencia que es ampliamente utilizada y la función es un paso básico de otros muchos algoritmos, se ha considerado que ya ha sido probada de forma extensiva por parte de los desarrolladores y la comunidad, no habiéndose encontrado en Internet indicios de que su funcionamiento no sea correcto.

A esta implementación se le ha asignado el identificador *qutip_impl*.

En la tabla 5.1 se observan los resultados obtenidos de las mediciones al ejecutar el algoritmo con hasta 15 qubits y en las tablas 10.1 y 10.2 dos comparativas de memoria y tiempo con otras implementaciones.

En cualquiera de esas tablas puede observarse que el máximo de qubits con los que puede trabajar son 12. La función da una excepción cuando se llama a la función de *qutip* a partir de 13 con el mensaje de error que se muestra en la figura 5.2.

Qubits	Memoria (MB)	Tiempo (s)	Varianza (s)
1	42.34375	0.00108	0.00013
2	42.34375	0.00104	0.00012
3	42.34375	0.00121	0.00022
4	42.34375	0.00127	0.00016
5	42.34375	0.00163	0.00011
6	42.67578	0.00333	0.00030
7	43.70703	0.01037	0.00140
8	47.84375	0.03920	0.00172
9	64.57031	0.14529	0.00248
10	130.57812	0.52456	0.01863
11	394.74219	2.01386	0.03994
12	1450.74219	7.97829	0.14193
13	x	x	x
14	x	x	x
15	x	x	x

Tabla 5.1: Implementación de *qutip* - Resultados de las mediciones

```

gildarth@ubuntu: ~/Desktop/QFT
python3.4 QFT_qutip_impl.py
qubits: 13
Traceback (most recent call last):
  File "QFT_qutip_impl.py", line 53, in <module>
    QFT_qutip_impl().measure (c_repetitions, c_min_q, c_max_q, c_verbose)
  File "/home/gildarth/Desktop/QFT/common_module.py", line 110, in measure
    fv = self.execute(v) # Execute the function and retrieves the result
  File "QFT_qutip_impl.py", line 25, in execute
    F = qft(n)
  File "/usr/lib/python3/dist-packages/qutip/qip/algorithms/qft.py", line 63, in qft
    L = np.exp(L)
MemoryError
make: *** [qutip_impl] Error 1
gildarth@ubuntu:~/Desktop/QFT$

```

Figura 5.2: *qutip_impl* - MemoryError al ejecución para 13 qubits

Implementación base

Contenidos

6.1. <i>base_impl</i>: Implementación base	61
6.1.1. Diseño	62
6.1.2. Estudio de coste espacial - Teórico	62
6.1.3. Estudio de coste espacial - Práctico	66
6.1.4. Estudio de coste temporal - Teórico	67
6.1.5. Estudio de coste temporal - Práctico	68
6.1.6. Conclusiones	68

Como punto inicial, se decidió hacer un programa lo más sencillo posible que coja un estado cuántico, genere la matriz operador de la QFT completa y realice la multiplicación, devolviendo el estado cuántico resultante.

6.1. *base_impl*: Implementación base

A esta implementación se le ha asignado el identificador *base_impl*.

En la tabla 6.2 se observan los resultados obtenidos de las mediciones al ejecutar el algoritmo con hasta 15 qubits y en las tablas 7.10 y 7.11 dos comparativas de memoria y tiempo con otras implementaciones.

6.1.1. Diseño

Para la implementación se decidió extraer como constante parte del exponente $(2\pi i/N)$, calculándolo de esta manera solo una vez en lugar de una vez por término del operador.

Un diagrama de flujo del algoritmo puede observarse en la figura 6.1

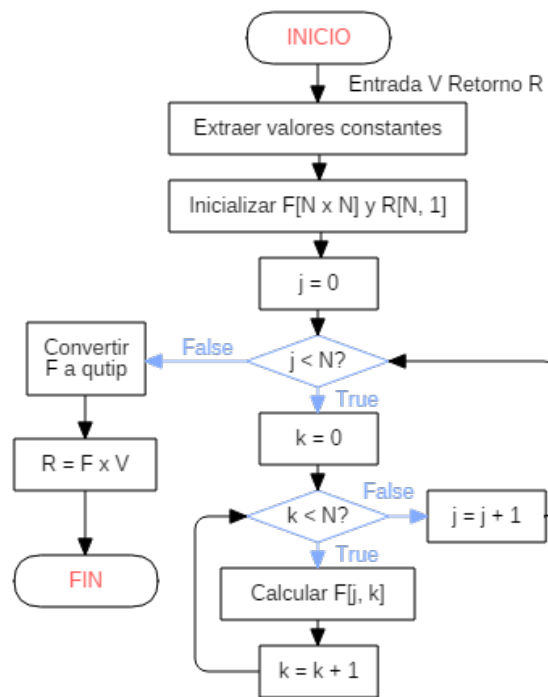


Figura 6.1: *base_impl* - Diagrama de flujo

6.1.2. Estudio de coste espacial - Teórico

El coste en memoria de este algoritmo es directamente dependiente del número de qubits utilizados en el estado. Esta dependencia es exponencial con base 2, pues tanto los estados como los operadores son matrices de números complejos, necesitando 2^n números complejos para los estados y $(2^n)^2$ para los operadores.

De aquí en adelante para simplificar se usará $N = 2^n$, siendo n el número de qubits.

En python los números complejos internamente se representan como dos valores en coma flotante, los cuales son implementados como un *double* de C cada uno (habitualmente codificado en binario usando la codificación IEEE 754 de 64 bits).

Los elementos principales donde se localiza el consumo de memoria son:

- La entrada, que es un estado cuántico de la librería de qutip. Internamente es una matriz columna de números complejos con N filas.
- La salida, al igual que la entrada, es un estado cuántico de la librería de qutip, siendo internamente una matriz columna de números complejos con N filas.
- El operador F es una matriz cuadrada de rango N de números complejos.

Tanto la entrada como la salida necesitan N números complejos lo que supone un incremento lineal de memoria respecto a N , pero el operador en cambio, al ser una matriz cuadrada, son N^2 números complejos y esto es un incremento cuadrático, por lo tanto la complejidad espacial teórica para este algoritmo es

$$2N + N^2 = 2 \times 2^n + (2^n)^2 = 2^{n+1} + 2^{2n}$$

lo que supone una complejidad espacial exponencial respecto al número de qubits que se incrementa el doble de rápido para el operador que para el estado.

Como puede comprobarse en la tabla la tabla 6.1, mientras que el incremento del uso de memoria de los estados es asumible, no lo es así el del operador.

En el supuesto de que se contasen con los 2GB íntegros para ser usados por el programa para el contenido de las variables, tendríamos $2GB = 2 \times 2^{40} \times 8 = 2^{44} \text{bits}$, y puesto que cada número complejo utiliza 2 números en coma flotante y estos suelen necesitar 64 bits, cada número complejo consideramos que precisa

Qubits	Estado (<i>complex</i>)	Operador (<i>complex</i>)
1	2	4
2	4	16
3	8	32
4	16	64
5	32	128
\vdots	\vdots	\vdots
10	$2^{10} =$ 1024	$2^{20} =$ 1.048.576
\vdots	\vdots	\vdots
20	$2^{20} =$ 1.048.576	$2^{40} =$ 1.099.511.627.776

Tabla 6.1: *base_impl* - Incremento mínimo teórico en el uso de memoria

de $1complex = 2float = 2^9bits$, y por lo tanto disponemos de espacio para $2^{44}/2^9 = 2^{35}$ números complejos. Esto hace un máximo almacenable de 17 qubits

$$2 \times 2^{17} + (2^{17})^2 = 2 \times 2^{17} + 2^{34} = 17,180,131,328 < 2^{35} = 34,359,738,368$$

mientras que con 18 qubits ya se pasaría del espacio disponible solo con el operador

$$(2^{18})^2 = 2^{36} > 2^{35}$$

Los cálculos anteriores están realizados en el supuesto de disponer de 2GB enteros de memoria dedicados solo para el contenido de las variables, pero en la vida real la memoria es compartida entre el sistema operativo y los programas en ejecución. Además, aunque el proceso cuente con 2GB enteros, parte de estos se usarán para alojar el código a ser ejecutado, los índices con las direcciones de memoria dinámica, la pila de llamadas (stack) y el espacio para el contenido de las variables dinámicas (heap).

El sistema operativo en reposo mantiene un uso aproximado de memoria RAM de medio GB (como puede observarse en la figura 6.2), lo cual reduce la memoria disponible aproximadamente a 1536 MB ($2^{43} + 2^{42}$ bits, que son $2^{34} + 2^{33}$ números complejos).

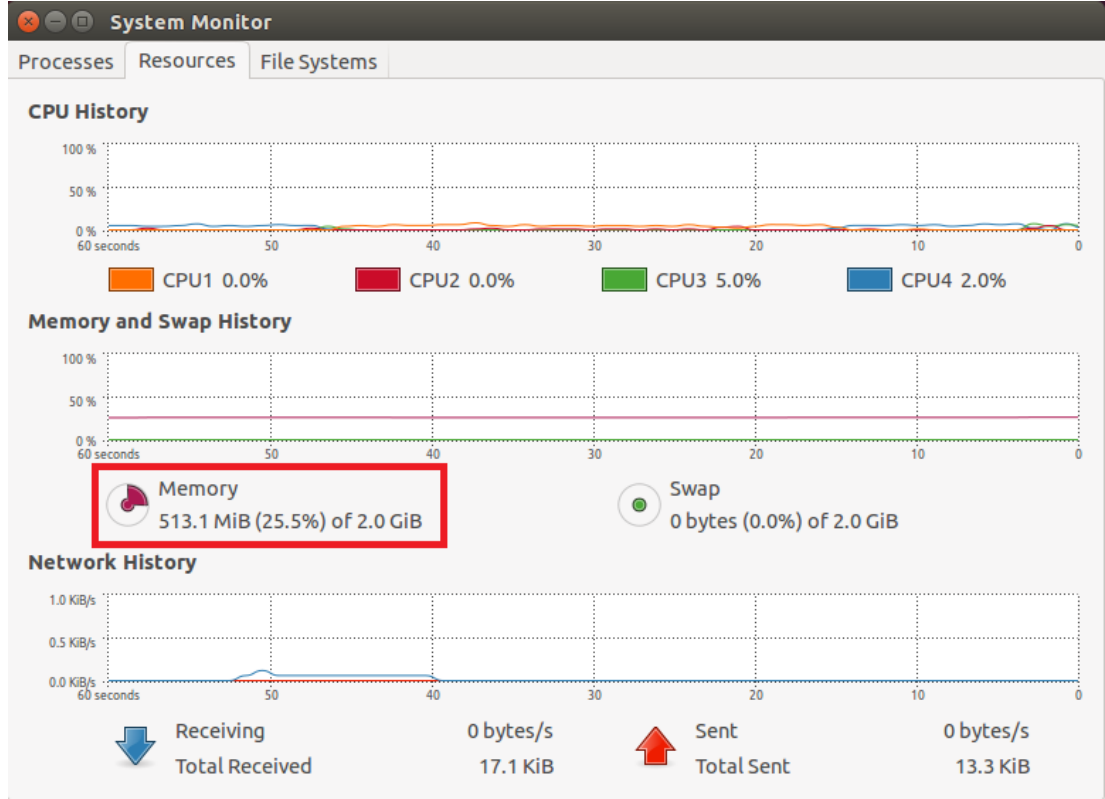


Figura 6.2: *base_impl* - Consumo de recursos solo con el SO en ejecución

Otro detalle a tener en cuenta es que en el algoritmo, el operador primero se construye como una matriz de la librería *numpy*, para luego transformarse en un objeto de la librería cuántica de *qutip*, por lo que en ese momento el operador se encuentra duplicado en memoria, aumentando el consumo real frente al consumo teórico mínimo calculado previamente. Así, en el momento de máximo uso de la memoria, se precisa de espacio para

$$2N + 2N^2$$

números complejos (frente a los $2N + N^2$ calculados previamente).

Con estas nuevas restricciones, el número máximo teórico de qubits antes de que haya problemas de memoria es de 16 qubits

$$2 \times 2^{16} + 2(2^{16})^2 = 2 \times 2^{16} + 2 \times 2^{32} = 8,590,065,664 < 2^{34} + 2^{33} = 25,769,803,776$$

mientras que con 17 qubits ya se pasaría con el tamaño del operador en el momento que esta duplicado

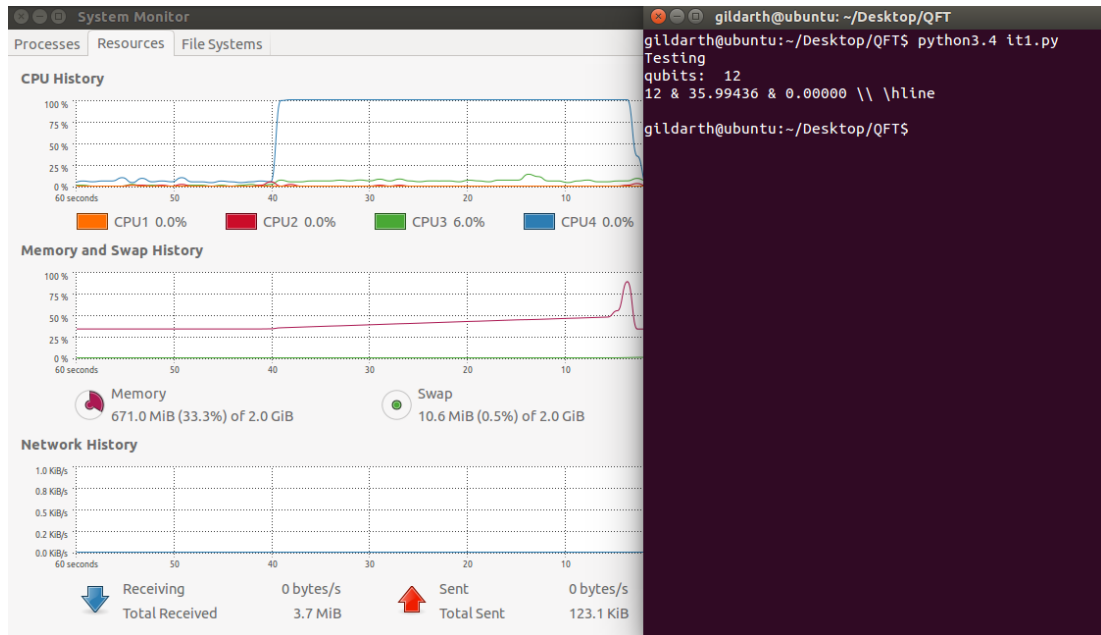
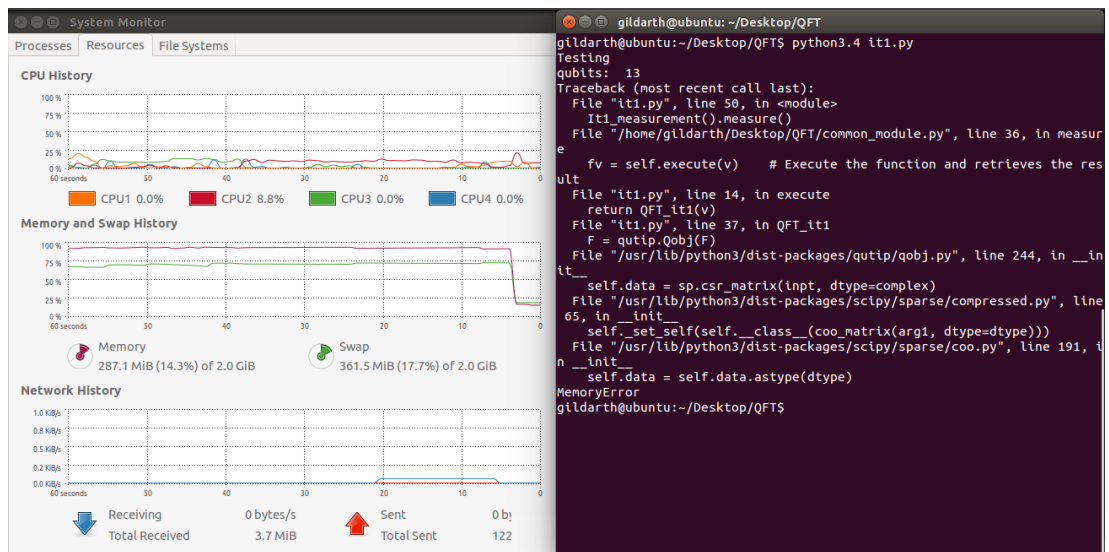
$$2(2^{17})^2 = 2 \times 2^{34} = 2^{35} > 2^{34} + 2^{33}$$

6.1.3. Estudio de coste espacial - Práctico

Pese a los cálculos teóricos, durante la ejecución del sistema hay más factores a tener en cuenta que consumen memoria, como son el propio shell desde el que se ejecuta el sistema, el interprete de python usado para la ejecución, el código propio y de librerías que ha de cargarse, el uso de espacio adicional de las estructuras de memoria utilizadas (por ejemplo un objeto de *qutip* no es solo la matriz interna que lo representa) y el uso de memoria extra necesario para las operaciones. Todo esto conlleva que el número máximo de qubits con los que se puede trabajar antes de tener problemas de memoria es inferior al esperado.

En concreto como puede verse en las figuras 6.3 y 6.4, ejecutando el sistema con 12 qubits este se ejecuta satisfactoriamente, estando siendo usados un total de 1.9 GB de memoria RAM, pero al tratar de ejecutar el programa para 13 qubits, la librería de *scipy* utilizada lanza una excepción debido a falta de memoria. Un análisis instrucción por instrucción del código demuestra que el punto donde salta esa excepción es la instrucción “ $F = \text{qutip.Qobj}(F)$ ” que es cuando se transforma la matriz de *numpy* F con los valores del operador en un objeto operador de *qutip*. Ese punto es también donde la ejecución con 12 qubits alcanzó el máximo uso de memoria.

Esto puede observarse en la tabla 6.2. Como se puede observar, hay un mínimo consumo por parte del proceso de algo más de 42.41 MB, lo cual es suficiente hasta que se opera con 6 qubits, momento a partir del cual se incrementa muy rápido.

Figura 6.3: *base_impl* - Ejecución con 12 qubitsFigura 6.4: *base_impl* - Ejecución con 13 qubits

6.1.4. Estudio de coste temporal - Teórico

Así como el incremento en el uso de memoria crece de forma exponencial al número de qubits, también así lo hace el incremento en tiempo de ejecución, pues

no solo ha de generarse el operador recorriendo una matriz cuadrada con la consiguiente complejidad cuadrática, sino que luego ha de realizarse una multiplicación de matrices al aplicar el operador al estado.

Por ello y puesto que el tamaño de las matrices es exponencial respecto al número de qubits, la complejidad se dispara, siendo la de construir el operador

$$O(N^2) = O((2^n)^2) = O(2^{2n})$$

y la de aplicar el operador al estado

$$O((2N)^2) = O((2(2^n))^2) = O((2^{n+1})^2) = O(2^{2n+2})$$

lo que supone una complejidad total de ambas operaciones respecto al número de qubits (n) de

$$O(2^{2n} + 2^{2n+2}) \approx O(2^{2n} + 2^{2n}) \approx O(2 \times 2^{2n}) \approx O(2^{2n})$$

esto es una complejidad cuadrática respecto al tamaño del vector (N) y por lo tanto exponencial respecto al número de qubits (n).

6.1.5. Estudio de coste temporal - Práctico

Los tiempos de ejecución para distintos tamaños del estado pueden observarse en la tabla 6.2. Observando esos resultados se ve un incremento exponencial en el coste temporal del algoritmo, lo cual coincide con lo calculado teóricamente.

6.1.6. Conclusiones

El algoritmo ha sido implementado de forma efectiva, aunque dista mucho de ser eficiente. A nivel de memoria se ha conseguido una mejora significativa respecto a la implementación de *qutip* como se ve en la tabla 7.10, pero no se ha aumentado el límite superior al tamaño del estado (número de qubits) debido a una limitación por parte de la librería *scipy*.

Qubits	Memoria (MB)	Tiempo (s)	Varianza (s)
1	42.41797	0.00090	0.00009
2	42.41797	0.00095	0.00008
3	42.41797	0.00097	0.00008
4	42.41797	0.00115	0.00008
5	42.41797	0.00196	0.00030
6	42.96484	0.00464	0.00030
7	43.84766	0.01541	0.00047
8	46.98438	0.05860	0.00131
9	59.82812	0.23258	0.01176
10	110.97656	0.88197	0.02050
11	315.49219	3.78109	0.10530
12	1131.83984	14.74178	0.18408
13	x	x	x
14	x	x	x
15	x	x	x

Tabla 6.2: *base_impl* - Resultados de las mediciones

El consumo de memoria se concentra en el operador siendo este una matriz cuadrada y teniendo que estar duplicada en el momento de convertirla a un objeto de *qutip* para operar con el estado de entrada. Se probó a construir el operador directamente sobre un objeto de *qutip*, pero resultó en un gran aumento del tiempo de ejecución, por ejemplo al operar con un estado de 8 qubits pasa de tardar 0.05 segundos a casi 1 segundo, por lo que al ser una implementación inicial que será refinada en un futuro, se decidió seguir construyendo contenido del operador como una matriz aparte para luego transformarlo a un objeto de *qutip*.

En el aspecto temporal el consumo es considerable, y gran parte del tiempo se emplea en crear el operador, siendo necesario para cada posición de este una operación exponencial cuyo exponente es un número complejo.

En comparación con los resultados de *qutip*, no se ha conseguido aumentar el número de qubits operables pero si se ha reducido el consumo de memoria (por ejemplo de 1450 MB a 1131 MB para 12 qubits), aunque el coste temporal es más elevado siendo en el caso de 12 qubits cercano a 3/4 partes más que con la implementación de *qutip* (pasa de casi 8 segundos con *qutip* a casi 14).

Optimización del consumo de memoria

Contenidos

7.1. <i>mem_opt_1</i>: Operador parcial	72
7.1.1. Diseño	73
7.1.2. Estudio de coste espacial - Teórico	73
7.1.3. Estudio de coste espacial - Práctico	75
7.1.4. Estudio de coste temporal - Teórico	77
7.1.5. Estudio de coste temporal - Práctico	77
7.1.6. Conclusiones	78
7.2. <i>mem_opt_2</i>: No convertir el operador	79
7.2.1. Diseño	80
7.2.2. Estudio de coste espacial - Teórico	80
7.2.3. Estudio de coste espacial - Práctico	81
7.2.4. Estudio de coste temporal - Teórico	82
7.2.5. Estudio de coste temporal - Práctico	82
7.2.6. Conclusiones	84
7.3. <i>mem_opt_3</i>: Operador parcial y no convertir el operador . . .	85
7.3.1. Diseño	86

7.3.2. Estudio de coste espacial - Teórico	86
7.3.3. Estudio de coste espacial - Práctico	87
7.3.4. Estudio de coste temporal - Teórico	87
7.3.5. Estudio de coste temporal - Práctico	89
7.3.6. Conclusiones	89
7.4. Resultados obtenidos	90

Para la optimización en el consumo de memoria se han llevado a cabo tres variaciones de la implementación base (*mem_opt_1*, *mem_opt_2* y *mem_opt_3*) con el fin de reducir el consumo espacial del algoritmo y con ello tratar de aumentar el número de qubits operables, siendo por el momento un máximo de 12 qubits. El punto donde más se puede optimizar dicho consumo es el operador, y por lo tanto es ahí donde se centran los esfuerzos.

Cada implementación realizada se ha ejecutado y medido según lo establecido en la sección 5.1. En la tabla 7.10 y en 7.11 puede observarse respectivamente una comparativa de los resultados obtenidos sobre el coste de memoria y el coste temporal de estas iteraciones con *qutip_impl* y *base_impl*.

A cada implementación se le han aplicado la misma validación que se le aplicó a la implementación base, pues solo se espera cambiar características no funcionales del algoritmo y por lo tanto debieran ser capaces de pasar las mismas pruebas funcionales.

7.1. *mem_opt_1*: Operador parcial

Al analizar el operador, por sus propias características, no se ha encontrado ningún método de compresión de matrices adecuado para reducir su consumo de memoria (por ejemplo, no se pueden usar técnicas para matrices huecas pues el operador no contiene ceros). Por ello se decidió generar el operador fila por fila (en lugar de generarlo completamente) y operar cada fila por separado, con lo que el uso de memoria del operador se reduce de N^2 números complejos (matriz cuadrada) en todo momento, a N números complejos (matriz fila) en cada momento

concreto. Esto supone un aumento en el coste temporal del algoritmo, pero con una sustancial mejora espacial.

A esta implementación se le ha asignado el identificador *mem_opt_1*.

En la tabla 7.3 se observan los resultados obtenidos de las mediciones al ejecutar el algoritmo con hasta 15 qubits y en las tablas 7.1 y 7.2 dos comparativas de memoria y tiempo con otras implementaciones.

7.1.1. Diseño

Partiendo de la implementación de *base_impl*, se han de modificar los bucles donde se genera el operador de forma que, mientras antes se generaba el operador completo para luego convertirlo aun objeto de *qutip* y aplicarlo obteniendo el resultado completo, ahora se convierte a un objeto de *qutip* y se aplica cada fila generada por separado, obteniendo el resultado fila a fila. Para aprovechar y reducir al máximo el consumo de memoria, las filas del operador se construyen sobre el mismo array de forma que el consumo de memoria sea constante durante la ejecución.

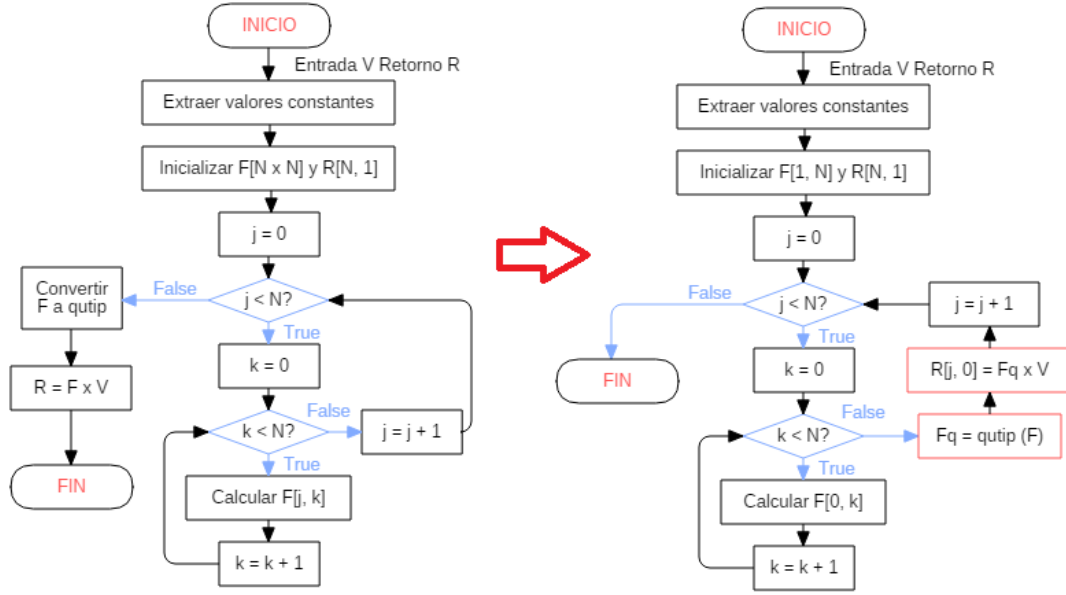
Un diagrama de flujo del algoritmo con los cambios principales respecto a *base_impl* en cajas rojas puede observarse en la figura 7.1

7.1.2. Estudio de coste espacial - Teórico

Al igual que en *base_impl*, el coste en memoria es dependiente del número de qubits, pero a diferencia de antes, el operador no es una matriz cuadrada, sino una matriz fila creada para cada valor del estado resultante. Dicha matriz fila contiene la misma cantidad de números complejos que cada uno de los estados (el de entrada y el resultante)

Esto supone un consumo espacial teórico mínimo para este algoritmo de

$$4N = 4 \times 2^n = 2^{n+2}$$

Figura 7.1: *mem_opt_1* - Diagrama de flujo - Cambios

($2N$ de los estados de entrada y salida y $2N$ del operador duplicado al convertirlo a objeto de *qutip*) frente al consumo mínimo de la implementación anterior que es

$$2N + 2N^2 = 2(2^n) + 2 \times (2^n)^2 = 2^{n+1} + 2^{2n+1}$$

para un número n de qubits.

La mejora teórica es sustancial ($4N \ll 2N + 2N^2$) al haberse eliminado el termino cuadrático y quedando solo términos lineales respecto a N .

Volviendo al supuesto de que se contasen con los 2GB íntegros para ser usados por el programa en el contenido de las variables, tendríamos $2GB = 2 \times 2^{40} \times 8 = 2^{44}bits$, y puesto que cada número complejo utiliza 2 números en coma flotante y estos suelen necesitar 64 bits, cada número complejo consideramos que precisa de 2^9bits (2 float de C), y por lo tanto disponemos de espacio para $2^{44}/2^9 = 2^{35}$ números complejos. Esto hace un máximo teórico ideal de 33 qubits exactos

$$4 \times 2^{33} = 2^{33+2} = 2^{35} = 34,359,738,368$$

mientras que con 34 qubits obviamente se pasaría

$$4 \times 2^{34} = 2^{34+2} = 2^{36} > 2^{35}$$

7.1.3. Estudio de coste espacial - Práctico

No obstante, y al igual que en *base_impl*, la situación real no es ideal, teniendo que tener en cuenta el consumo del propio SO, del programa, etc. . .

Puesto que los tiempo de ejecución del algoritmo completo para estados de más de 15 qubits se dispara, se ha ejecutado para estados de hasta 15 qubits de la misma forma que en la primera iteración y se ha medido el consumo de memoria del proceso.

En la tabla 7.3 puede observarse los resultados completos de la ejecución, y en la tabla 7.1 una comparativa con la implementación de partida. Se ha conseguido una enorme mejoría con respecto a los datos de *base_impl*. Ya no solo se ejecuta para estados de 13 qubits y más, sino que el consumo es inferior al mínimo del proceso hasta 11 qubits, incrementándose luego de forma mucho más lenta. Como ejemplo *base_impl* precisa para 11 qubits 315.5 MB y para 12 qubits 1131.8 MB, mientras que esta implementación consume respectivamente solo 43.6 MB y 43.8 MB.

En cuanto a los límites del algoritmo con la memoria disponible, se ha realizado una segunda implementación del algoritmo que permite calcular solo unos cuantos valores del vector resultado, pudiendo así comprobar como se comporta el consumo de memoria para estados de más de 15 qubits en un tiempo razonable. El consumo de memoria es igual para calcular el valor de una posición del estado resultado que para el resultado completo, pues se reutiliza el espacio reservado para el operador sobrescribiendo su contenido, y por lo tanto el consumo de memoria no crece una vez se ha calculado el primer valor.

Para comprobar que la medición del consumo es correcta, se han comparado que los valores resultantes son acordes con los obtenidos ejecutando el algoritmo completo para estados de 15 qubits y menos (teniendo en cuenta que diferen-

n	base_impl	mem_opt_1
1	42.81250	42.36719
2	42.81250	42.36719
3	42.81250	42.36719
4	42.81250	42.36719
5	42.81250	42.36719
6	42.81250	42.36719
7	43.70312	42.36719
8	47.01172	42.36719
9	59.87109	42.36719
10	110.89844	42.36719
11	315.03906	42.36719
12	1128.67188	42.93359
13	x	43.44141
14	x	44.04688
15	x	45.37500
16	x	47.97656
17	x	53.75000
18	x	64.54688
19	x	86.23438
20	x	129.47656
21	x	215.96484
22	x	388.94141
23	x	735.15234
24	x	1427.06250
25	x	x

Tabla 7.1: *mem_opt_1* - Comparativa del uso de memoria (MB)

tes ejecuciones del mismo programa pueden tener ligeras variaciones, como por ejemplo que el consumo mínimo del interprete de *python* no siempre es el mismo).

Como puede observarse en la tabla 7.10, hasta 24 qubits el algoritmo se ejecuta, con un consumo máximo por parte del proceso de algo más de 1400 MB, no obstante cuando se ejecuta para un estado de 25 qubits se lanza una excepción **MemoryError**. Esto se produce a la hora de aplicar el operador parcial al estado de entrada.

7.1.4. Estudio de coste temporal - Teórico

Al estudiar la distribución y contenido de los bucles, se ve para cada fila se genera el operador parcial ($O(N)$), se convierte a un objeto de qutip (como se hace una copia de la matriz $O(N)$) y se multiplican el estado y el operador (al ser una multiplicación del vector fila operador por el vector columna estado $O(N)$) lo cual da lugar a $O(3N)$, y esto se repite N veces

$$O(3N \times N) = O(3N^2) = O(3(2^n)^2) = O(3(2^{2n})) \approx O(2^{2n})$$

siendo una complejidad cuadrática respecto al tamaño del vector (N) y por lo tanto exponencial respecto al número de qubits (n).

A grandes rasgos no hay cambio con *base_impl* cuya complejidad teórica es también cuadrática respecto a N y exponencial respecto a n ($O(2^{2n} + 2^{2n+2}) \approx O(2^{2n})$). Esto se explica pues el número de bucles es el mismo solo que ordenados de diferente manera: mientras que en *base_impl* se generan todas las filas del operador, se convierten todas a la vez a un objeto qutip y se opera con todo a la vez, en esta implementación se genera, convierte y aplica cada fila por separado, pero al final en ambos se han generado, convertido y operado N filas del operador.

Aún así, el tiempo de ejecución de esta implementación se espera sea mayor que el de *base_impl* pues su término tiene un multiplicador aproximado de 2, mientras que el de esta implementación es 3

$$O(2^{2n} + 2^{2n+2}) \approx O(2(2^{2n})) < O(3(2^{2n}))$$

7.1.5. Estudio de coste temporal - Práctico

Como se esperaba y se observa en la tabla 7.2, se obtuvo peores resultados que en *base_impl*, aumentando el tiempo de ejecución. Por ejemplo, para 11 qubits se incrementó el tiempo de 3.8 segundos en *base_impl* a 5.7 segundos y para 12 qubits de 14.7 segundos a 19 segundos.

n	base_impl	mem_opt_1
1	0.00090	0.02397
2	0.00095	0.02114
3	0.00097	0.02437
4	0.00115	0.03104
5	0.00196	0.04386
6	0.00464	0.07197
7	0.01541	0.13336
8	0.05860	0.28425
9	0.23258	0.70011
10	0.88197	1.82345
11	3.78109	5.78695
12	14.74178	19.07792
13	x	68.59218
14	x	260.01283
15	x	1083.12267

Tabla 7.2: *mem_opt_1* - Comparativa del consumo de tiempo (s)

También se observa un aumento del consumo de memoria para estados de pocos qubits junto a un incremento menos acusado: el incremento entre 1 qubit y 9 de *base_impl* es de 0.00090 hasta 0.23258 (mas de 200 veces más para 9 qubits que para 1) mientras que en esta implementación aunque empieza ya en 0.02397 solo aumenta hasta 0.70011 (alrededor de 30 veces más)

7.1.6. Conclusiones

Como se observa en la tabla 7.1, el consumo de memoria se ha reducido drásticamente consiguiendo poder operar con hasta 24 qubits con un consumo máximo de 1427 MB, frente al límite de 12 qubits con un consumo de 1131 MB obtenidos con *base_impl*.

En lo referente al coste temporal, el aumento respecto a *base_impl* que se observa en la tabla 7.2 es asumible, sobretodo teniendo en cuenta la reducción del consumo de memoria. Dicho coste temporal sigue centrado principalmente en la construcción del operador, aunque ahora sea un proceso fragmentado, debido a la gran cantidad de operaciones necesarias y a complejidad de estas.

Qubits	Memoria (MB)	Tiempo (s)	Varianza (s)
1	42.58203	0.02397	0.00998
2	42.58203	0.02114	0.00090
3	42.58203	0.02437	0.00095
4	42.58203	0.03104	0.00172
5	42.58203	0.04386	0.00098
6	42.58203	0.07197	0.00108
7	42.58203	0.13336	0.00126
8	42.58203	0.28425	0.01635
9	42.58203	0.70011	0.02804
10	43.32812	1.82345	0.03105
11	43.58594	5.78695	0.28245
12	43.80078	19.07792	0.13347
13	44.24609	68.59218	0.52252
14	45.40234	260.01283	1.68897
15	47.80078	1083.12267	7.55778

Tabla 7.3: *mem_opt_1* - Resultados de las mediciones

7.2. *mem_opt_2*: No convertir el operador

Al observar el monitor del sistema durante la ejecución del algoritmo se ha detectado un pico de memoria considerablemente grande (el cual puede verse en la figura 7.2). Este pico de memoria se corresponde con el momento en el cual se convierte el operador generado a un objeto de *qutip* para posteriormente aplicarlo al estado.

Por ello, se ha decidido realizar una variación de *base_impl* la cual omite esa parte de la conversión del operador a la que se le ha asignado el identificador *mem_opt_2*.

En la tabla 7.6 se observan los resultados obtenidos de las mediciones al ejecutar el algoritmo con hasta 15 qubits y en las tablas comparativas 7.10 y 7.11 dos comparativas de memoria y tiempo con otras implementaciones.

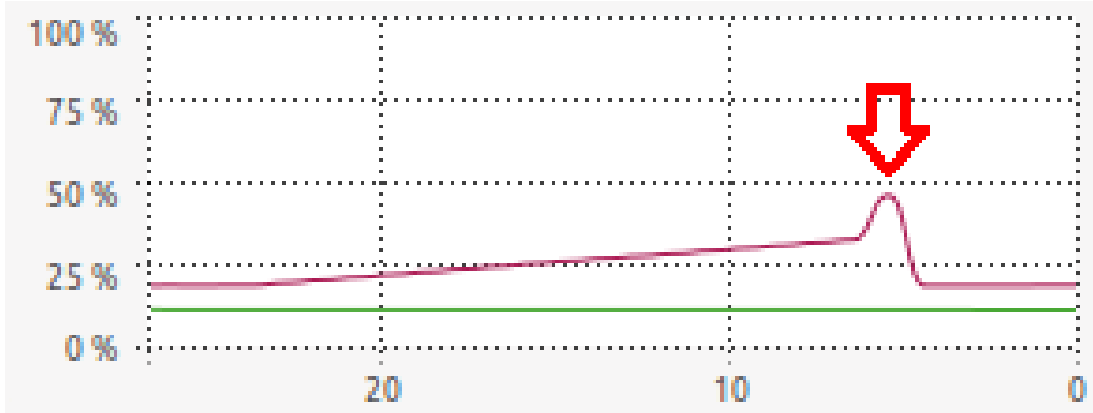


Figura 7.2: *mem_opt_2* - Memoria en el monitor de recursos (12 qubits)

7.2.1. Diseño

Para poder llevar a cabo estos cambios, se precisa obtener la representación matricial del estado de entrada para poder operar con las matrices directamente y luego convertir a objeto de *qutip* solo la matriz resultado.

Un diagrama de flujo del algoritmo con los cambios principales respecto a *base_impl* en cajas rojas puede observarse en la figura 7.3

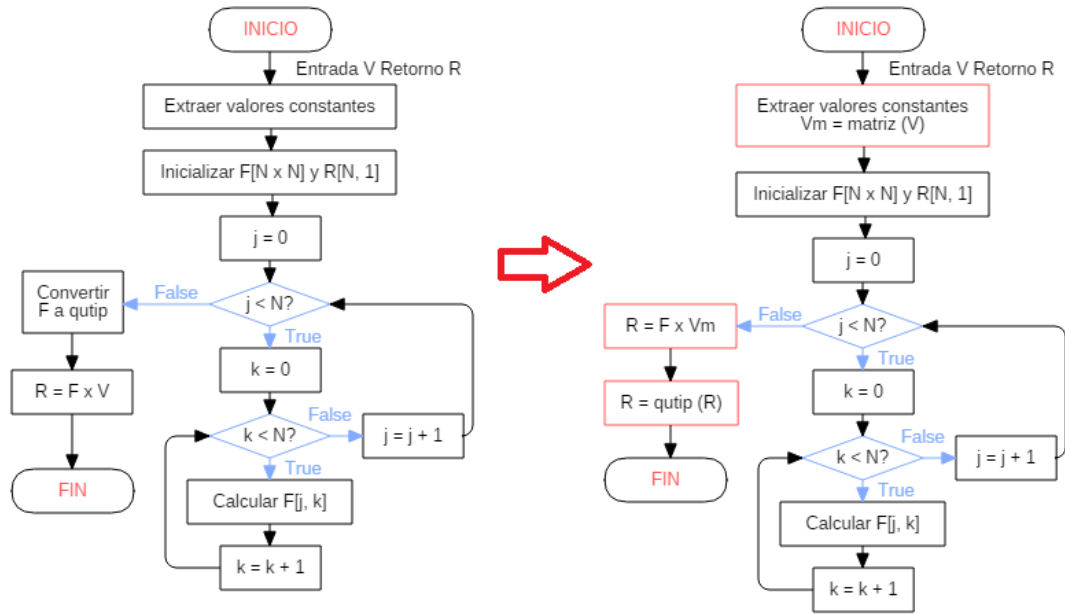
7.2.2. Estudio de coste espacial - Teórico

La principal diferencia respecto a *base_impl* radica en que, al no convertir el operador a un objeto de *qutip* antes de aplicarlo, este ya no se encuentra duplicado en memoria. A cambio es la matriz resultado la que, antes de terminar, se encuentra duplicada en memoria. No obstante, la matriz resultado es de tamaño N mientras que el operador es de tamaño N^2 y por lo tanto el consumo es muy inferior duplicando el resultado que duplicando el operador.

Más concretamente, la complejidad espacial teórica es de

$$3N + N^2 = 3(2^n) + (2^n)^2 = 3(2^n) + 2^{2n} \approx 2^n + 2^{2n}$$

mientras que la de *base_impl* era de $2N + 2N^2$.

Figura 7.3: *mem_opt_2* - Diagrama de flujo - Cambios

Por ello, se espera una reducción del consumo de memoria cercana al 50 %, pero no un aumento en el número de qubits operables al no eliminarse el término cuadrático y por lo tanto la restricción de *scipy* para operar con matrices de ese tamaño que surgió en *base_impl* para más de 12 qubits sigue presente.

7.2.3. Estudio de coste espacial - Práctico

Los resultados sobre el consumo de memoria para diferentes tamaños del estado de entrada se pueden observar en la tabla 7.6 y en las tablas comparativa 7.4 y 7.10.

Como se esperaba, a la hora de ejecutar el algoritmo para 13 qubits la librería *scipy* lanza una excepción **MemoryError** a la hora de realizar la multiplicación de matrices.

También conforme a lo esperado, la memoria requerida se reduce de forma más visible a cuantos más qubits se operen, siendo de aproximadamente el 50 %

para 12 qubits (de 1128 MB en *base_impl* a 554 MB). Además, como se observa en la figura 7.4 el pico de memoria al final de la ejecución se ha eliminado.

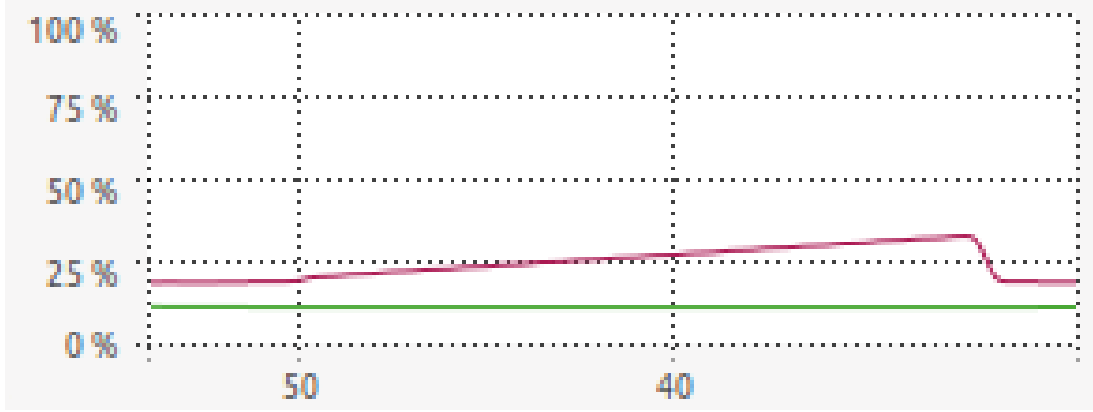


Figura 7.4: *mem_opt_2* - Memoria en el monitor de recursos (12 qubits)

7.2.4. Estudio de coste temporal - Teórico

Puesto que el único cambio respecto a *base_impl* es la conversión a objeto de *qutip*, la complejidad temporal sigue centrándose en la generación del operador y la aplicación de este al estado de entrada, implicando ambas operaciones el recorrido de una y dos matrices cuadradas respectivamente y por lo tanto la complejidad temporal teórica es exponencial respecto a n

$$O(2^{2n} + 2^{2n+2}) \approx O(2^{2n} + 2^{2n}) \approx O(2 \times 2^{2n}) \approx O(2^{2n})$$

lo que resulta en una complejidad cuadrática respecto al tamaño del vector (N) y por lo tanto exponencial respecto al número de qubits (n).

7.2.5. Estudio de coste temporal - Práctico

Como puede observarse en la tabla 7.6 y en las tablas comparativas 7.5 y 7.11, pese a la eliminación del paso de conversión de la matriz operador a un objeto de *qutip*, se ha aumentado ligeramente el tiempo de ejecución a partir de 8

n	base_impl	mem_opt_2
1	42.81250	42.31250
2	42.81250	42.31250
3	42.81250	42.31250
4	42.81250	42.31250
5	42.81250	42.31250
6	42.81250	42.31250
7	43.70312	42.64844
8	47.01172	44.71094
9	59.87109	50.73047
10	110.89844	74.69922
11	315.03906	170.80078
12	1128.67188	554.88672
13	x	x
14	x	x
15	x	x

Tabla 7.4: mem_opt_2 - Comparativa del uso de memoria (MB)

qubits (por ejemplo para 12 qubits de una media de 14.7 a una de 15.7 segundos aproximadamente).

Puesto que las únicas diferencias son la generación de una representación matricial del estado de entrada, el operar con matrices directamente en lugar de objetos de *qutip* y la conversión a objeto de *qutip* solo del estado resultante, se deduce que la diferencia del tiempo de ejecución ha de estar en la extracción de la representación matricial del estado de entrada o en la aplicación del operador directamente con las matrices pues la conversión de matriz a objeto de *qutip* ya se hacía antes y con un mayor volumen de datos. Ambas posibilidades consideradas tiene como origen la implementación interna que haga de sus objetos *qutip*, pudiendo ser que sea muy costoso extraer una representación matricial de ellos o que se realice alguna optimización interna a la hora de aplicar un operador a un estado.

Haciendo pruebas sobre la ejecución del algoritmo y la librería se comprobó que el aumento de tiempo se debe, principalmente y con diferencia, a que durante la aplicación del operador al estado *qutip* ha de hacer alguna optimización interna de la operación.

n	base_impl	mem_opt_2
1	0.00090	0.00044
2	0.00095	0.00048
3	0.00097	0.00058
4	0.00115	0.00067
5	0.00196	0.00145
6	0.00464	0.00441
7	0.01541	0.01491
8	0.05860	0.05886
9	0.23258	0.24231
10	0.88197	0.94307
11	3.78109	3.87133
12	14.74178	15.67649
13	x	x
14	x	x
15	x	x

Tabla 7.5: *mem_opt_2* - Comparativa del consumo de tiempo (s)

7.2.6. Conclusiones

Como se observa en las tablas comparativas 7.4 y 7.5 el no convertir el operador a un objeto de *qutip* en comparación a *base_impl* ha permitido reducir la memoria hasta un 50 % (de 1128 a 554 MB) para 12 qubits con un aumento en el tiempo de ejecución de 1 segundo (de 14.7 a 15.7 segundos). Teniendo en cuenta que lo que se trata de reducir es el consumo de memoria estos resultados se consideran positivos.

Ademas, como se observa en la figura 7.4, el incremento del uso de memoria durante una ejecución es completamente lineal, sin sufrir ningún pico brusco hasta que acaba el algoritmo.

No obstante, el objetivo principal (que es aumentar el número de qubits operables respecto a *base_impl*) no se ha conseguido, obteniendo un `MemoryError` de la librería *scipy* a la hora de operar con más de 12 qubits (que es tambien el límite de partida).

Qubits	Memoria (MB)	Tiempo (s)	Varianza (s)
1	42.55859	0.00044	0.00007
2	42.55859	0.00048	0.00006
3	42.55859	0.00058	0.00010
4	42.55859	0.00067	0.00003
5	42.55859	0.00145	0.00014
6	42.55859	0.00441	0.00020
7	43.33984	0.01491	0.00067
8	45.18750	0.05886	0.00308
9	51.24219	0.24231	0.01846
10	75.19531	0.94307	0.02185
11	171.42969	3.87133	0.10386
12	555.65625	15.67649	0.33205
13	x	x	x
14	x	x	x
15	x	x	x

Tabla 7.6: *mem_opt_2* - Resultados de las mediciones

7.3. *mem_opt_3*: Operador parcial y no convertir el operador

Considerando los resultados obtenidos en *mem_opt_1* y *mem_opt_2*, ambos consiguieron mejorar el consumo de memoria respecto a *base_impl*, y puesto que ambos incluyen modificaciones compatibles entre si, se ha realizado una nueva implementación uniendo las mejoras de ambas.

A esta implementación se le asignó el identificador *mem_opt_3*.

En la tabla 7.9 se observan los resultados obtenidos de las mediciones al ejecutar el algoritmo con hasta 15 qubits y en las tablas comparativas 7.10 y 7.11 dos comparativas de memoria y tiempo con otras implementaciones.

7.3.1. Diseño

Para unir ambas optimizaciones, se ha partido del código de *mem_opt_1* y se le ha añadido las partes de extraer la matriz representativa del estado de entrada, operar con las matriz parcial del operador y la del estado de entrada directamente y convertir el estado resultado al final del algoritmo tal y como se muestra en el diagrama de flujo de la figura 7.5.

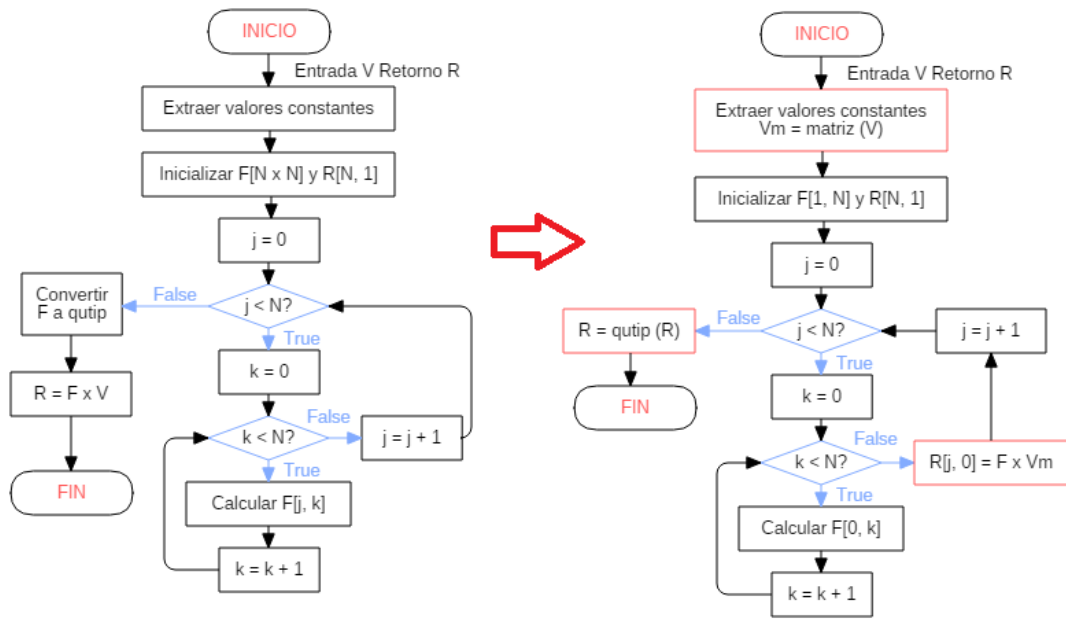


Figura 7.5: *mem_opt_3* - Diagrama de flujo - Cambios

7.3.2. Estudio de coste espacial - Teórico

Puesto que se parte *mem_opt_1* se espera mantener el número máximo de qubits del estado de entrada que son operables en 24.

En cuanto a la complejidad, la principal diferencia con *mem_opt_1* es que no se duplica el operador parcial a la hora de convertirlo a un objeto de *qutip* pero a cambio si se duplica el estado resultado, y puesto que ambos son de tamaño N ,

la complejidad final es la misma que para *mem_opt_1*

$$4N = 4 \times 2^n = 2^{n+2}$$

7.3.3. Estudio de coste espacial - Práctico

Como se esperaba y se puede observar en la tabla 7.7, se ha mantenido el límite superior en el número de qubits operables de 24 que hay en *mem_opt_1*, teniendo el mismo problema a la hora de aplicar el operador al estado de entrada para 25 qubits, donde la librería de *scipy* lanza un **MemoryError**.

También, en consecuencia a que la complejidad espacial no haya cambiado respecto a *mem_opt_1*, el consumo de memoria es prácticamente igual (como se ve en la tabla 7.10), siendo por lo general el consumo de esta implementación superior, pero siempre por menos de 1 MB.

7.3.4. Estudio de coste temporal - Teórico

Partiendo de lo explicado para *mem_opt_1*, la diferencia es que se mueve la conversión a un objeto de *quint* del operador que esta dentro del bucle, al resultado que esta fuera y se le añade el obtener la matriz del estado de entrada, por lo tanto en el bucle principal hay dos operaciones de de N pasos y otras dos fuera de este (una antes y otra después)

$$O(N + N \times N + N) = O(2N^2 + 2N) = O(2(2^n)^2 + 2(2^n)) = O(2(2^{2n}) + 2(2^n))$$

siendo de nuevo una complejidad cuadrática respecto al tamaño del vector (N) y por lo tanto exponencial respecto al número de qubits (n).

En comparación con *mem_opt_1* ($O(3(2^{2n}))$) se puede esperar una ligera mejora temporal pues

$$O(3(2^{2n})) > O(2(2^{2n}) + 2(2^n))$$

n	base_impl	mem_opt_3
1	42.81250	42.91797
2	42.81250	42.91797
3	42.81250	42.91797
4	42.81250	42.91797
5	42.81250	42.91797
6	42.81250	42.91797
7	43.70312	42.91797
8	47.01172	42.91797
9	59.87109	42.91797
10	110.89844	42.91797
11	315.03906	42.91797
12	1128.67188	43.19531
13	x	43.45312
14	x	44.20703
15	x	45.75391
16	x	48.33594
17	x	54.20312
18	x	64.99609
19	x	86.68750
20	x	129.92969
21	x	216.41797
22	x	389.39062
23	x	735.60547
24	x	1427.51172
25	x	x

Tabla 7.7: *mem_opt_3* - Comparativa del uso de memoria (MB)

y respecto a *mem_opt_2* ($O(2^{2n} + 2^{2n+2})$) también hay una mejora teórica

$$O(2^{2n} + 2^{2n+2}) \approx O(2 \times 2^{2n}) \approx O(4^{2n}) > O(2(2^{2n}) + 2(2^n))$$

pero esto último depende del balance entre lo que se gane por no convertir cada operador parcial a un objeto de *qutip* y lo que se pierda al no contar con las optimizaciones de *qutip* para aplicar un operador a un estado frente a una multiplicación de matrices normal.

7.3.5. Estudio de coste temporal - Práctico

En las tablas 7.8, 7.9 y 7.11 puede observarse que respecto a *mem_opt_1* se ha reducido ligeramente el tiempo de ejecución (de 1083 a 1003 segundos para 15 qubits).

También se observa que al igual que *mem_opt_1*, para estados de pocos qubits el consumo temporal (0.01873 segundos) es más alto que con *base_impl* (0.00090 segundos) o *mem_opt_2* (0.00044 segundos), pero luego tiene un incremento mucho menos acusado, llegando al punto que para 12 qubits el tiempo de ejecución de *mem_opt_2* solo está por debajo en 0.3 segundos (15.67 segundos de *mem_opt_2* frente a 15.96 de esta implementación).

7.3.6. Conclusiones

Los resultados comparativos con las implementaciones anteriores están en las tablas 7.7 y 7.8.

Aunque tanto *mem_opt_1* como *mem_opt_2* suponen grandes mejoras en el consumo de memoria frente a *base_impl*, unirlos no ha proporcionado ningún beneficio a este respecto en comparación con *mem_opt_1*, e incluso es peor aunque por muy poco (menos de 1 MB para 15 qubits), y no obstante se ha conseguido reducir el tiempo de ejecución de *mem_opt_1* pese a no ser el objetivo (del orden de 3 segundos para 12 qubits y 80 segundos para 15 qubits) quedando solo ligeramente por encima del de *base_impl* (para 12 qubits tarda 15.96 segundos frente

n	base_impl	mem_opt_3
1	0.00090	0.01873
2	0.00095	0.01814
3	0.00097	0.01922
4	0.00115	0.01910
5	0.00196	0.02111
6	0.00464	0.02585
7	0.01541	0.04662
8	0.05860	0.09352
9	0.23258	0.29347
10	0.88197	1.06126
11	3.78109	4.09253
12	14.74178	15.96296
13	x	62.77112
14	x	249.24228
15	x	1003.94503

Tabla 7.8: mem_opt_3 - Comparativa del consumo de tiempo (s)

a los 14.74 segundos de *base_impl* lo cual es menos de un 10 % más de tiempo). Por ello, en general, se considera que esta implementación es una mejora frente a *mem_opt_1*.

7.4. Resultados obtenidos

Se ha conseguido aumentar el número de qubits operables hasta 24 con solo un ligero aumento en el tiempo de ejecución (inferior al 10 % para 12 qubits). En las tablas 7.10 y 7.11 puede verse una comparativa del consumo de memoria y tiempo (respectivamente) de las diferentes implementaciones.

A continuación se va a realizar una valoración de los resultados obtenidos con el fin de poder seleccionar una implementación de partida para las optimizaciones del tiempo de ejecución que se vayan a realizar.

Analizando los datos obtenidos, se observó que el máximo número de qubits operables son 24 con *mem_opt_1* y *mem_opt_3* (siendo esto el doble exacto que en la implementación base) y de 12 con *mem_opt_2*. Esto coincide con que el operador

Qubits	Memoria (MB)	Tiempo (s)	Varianza (s)
1	42.33594	0.01873	0.00093
2	42.33594	0.01814	0.00062
3	42.33594	0.01922	0.00162
4	42.33594	0.01910	0.00067
5	42.33594	0.02111	0.00066
6	42.33594	0.02585	0.00090
7	42.33594	0.04662	0.01152
8	42.33594	0.09352	0.00143
9	42.33594	0.29347	0.00387
10	42.85938	1.06126	0.02876
11	43.37500	4.09253	0.01879
12	43.38281	15.96296	0.06718
13	44.08594	62.77112	0.18768
14	45.69922	249.24228	1.08615
15	47.21875	1003.94503	4.57757

Tabla 7.9: *mem_opt_3* - Resultados de las mediciones

de *base_impl* y *mem_opt_2* cuenta con $N^2 = 2^{2n}$ celdas y el de *mem_opt_1* y *mem_opt_3* son $N = 2^n$, siendo la misma cantidad de celdas para *mem_opt_1* y *mem_opt_3* con 24 qubits que para los otros dos con 12.

Pensando sobre ello se llegó a la conclusión que el problema es, no tanto la falta de espacio de almacenamiento, sino la cantidad de celdas de los arrays donde se almacena el operador, no pudiendo la librería de cálculo matemático operar con tantas celdas.

Para probar esta teoría, se aumento temporalmente la memoria RAM de la máquina virtual a 4GB y se ejecuto el cálculo del límite de las diferentes implementaciones, siendo los posibles resultados esperados que el número máximo de qubits operable se mantenga (por lo tanto no dependa tanto de la memoria del sistema) o que aumente el número de qubits operables (por lo tanto se posible escalar el número de qubits operables mediante un aumento de memoria).

Los resultados obtenidos fueron prácticamente idénticos con 4GB de RAM que con 2GB, siendo la diferencia el que no se llegara a utilizar la memoria de SWAP en ningún momento (saltando la excepción antes de que se consumieran los 4GB de RAM). Por lo demas, tanto los errores de *base_impl* y *mem_opt_2*

n	qutip_impl	base_impl	mem_opt_1	mem_opt_2	mem_opt_3
1	42.34375	42.81250	42.36719	42.31250	42.91797
2	42.34375	42.81250	42.36719	42.31250	42.91797
3	42.34375	42.81250	42.36719	42.31250	42.91797
4	42.34375	42.81250	42.36719	42.31250	42.91797
5	42.34375	42.81250	42.36719	42.31250	42.91797
6	42.67578	42.81250	42.36719	42.31250	42.91797
7	43.70703	43.70312	42.36719	42.64844	42.91797
8	47.84375	47.01172	42.36719	44.71094	42.91797
9	64.57031	59.87109	42.36719	50.73047	42.91797
10	130.57812	110.89844	42.36719	74.69922	42.91797
11	394.74219	315.03906	42.36719	170.80078	42.91797
12	1450.74219	1128.67188	42.93359	554.88672	43.19531
13	x	x	43.44141	x	43.45312
14	x	x	44.04688	x	44.20703
15	x	x	45.37500	x	45.75391
16	x	x	47.97656	x	48.33594
17	x	x	53.75000	x	54.20312
18	x	x	64.54688	x	64.99609
19	x	x	86.23438	x	86.68750
20	x	x	129.47656	x	129.92969
21	x	x	215.96484	x	216.41797
22	x	x	388.94141	x	389.39062
23	x	x	735.15234	x	735.60547
24	x	x	1427.06250	x	1427.51172
25	x	x	x	x	x

Tabla 7.10: mem_opt - Comparativa del uso de memoria (MB)

n	qutip_impl	base_impl	mem_opt_1	mem_opt_2	mem_opt_3
1	0.00108	0.00090	0.02397	0.00044	0.01873
2	0.00104	0.00095	0.02114	0.00048	0.01814
3	0.00121	0.00097	0.02437	0.00058	0.01922
4	0.00127	0.00115	0.03104	0.00067	0.01910
5	0.00163	0.00196	0.04386	0.00145	0.02111
6	0.00333	0.00464	0.07197	0.00441	0.02585
7	0.01037	0.01541	0.13336	0.01491	0.04662
8	0.03920	0.05860	0.28425	0.05886	0.09352
9	0.14529	0.23258	0.70011	0.24231	0.29347
10	0.52456	0.88197	1.82345	0.94307	1.06126
11	2.01386	3.78109	5.78695	3.87133	4.09253
12	7.97829	14.74178	19.07792	15.67649	15.96296
13	x	x	68.59218	x	62.77112
14	x	x	260.01283	x	249.24228
15	x	x	1083.12267	x	1003.94503

Tabla 7.11: mem_opt - Comparativa del consumo de tiempo (s)

para 13 qubits y los errores de *mem_opt_1* y *mem_opt_3* para 25 qubits fueron los mismos con 2GB y con 4GB de memoria RAM.

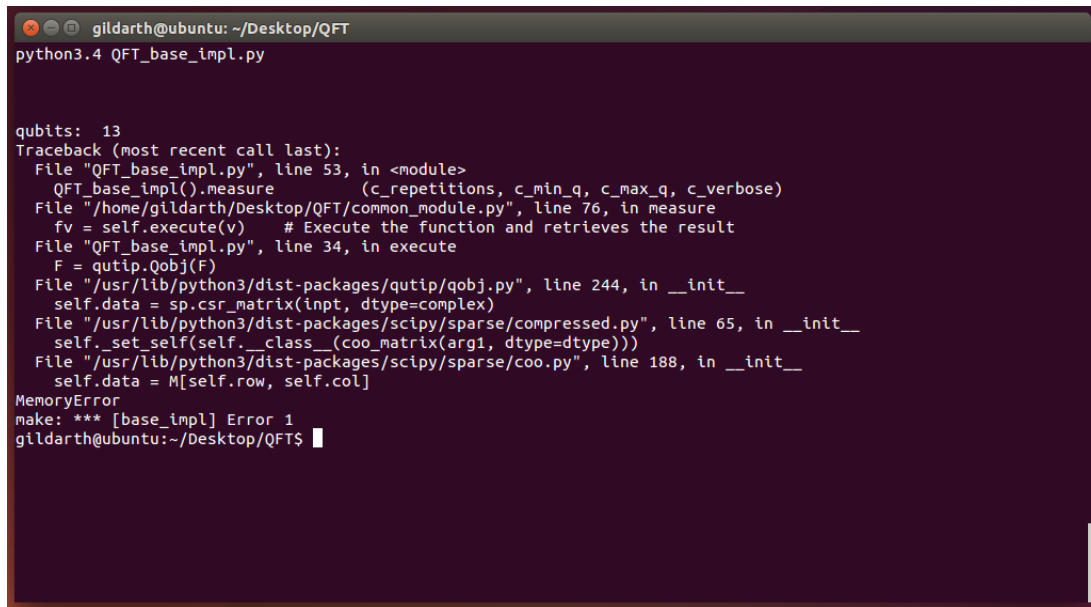
En cada implementación el error obtenido es diferente, siendo en general el punto donde se origina la excepción una clase de la librería *scipy*:

base_impl El error producido es al convertir la matriz cuadrada a un objeto operador de la librería *qutip* para poder aplicarlo al estado y se genera desde el fichero `/usr/lib/python3/dist-packages/scipy/sparse/coo.py` (Imagen 7.6).

mem_opt_1 El error en este caso se produce igual que para *base_impl* (Imagen 7.7), pero esta vez lo hace a partir de 25 qubits en lugar de 13.

mem_opt_2 El error se genera al operar la matriz operador por la matriz estado, ambos siendo matrices de *numpy* y no objetos de *qutip*. En este caso el interprete no ofrece más información sobre la fuente, solo informa de que se produjo un **MemoryError** en el momento de multiplicar (Imagen 7.7).

mem_opt_3 El error que se produce es el mismo que con *mem_opt_2*, supuestamente por que el tamaño máximo de una matriz para ser operable ha de tener un número de celdas entre 2^{24} y 2^{25} como máximo.



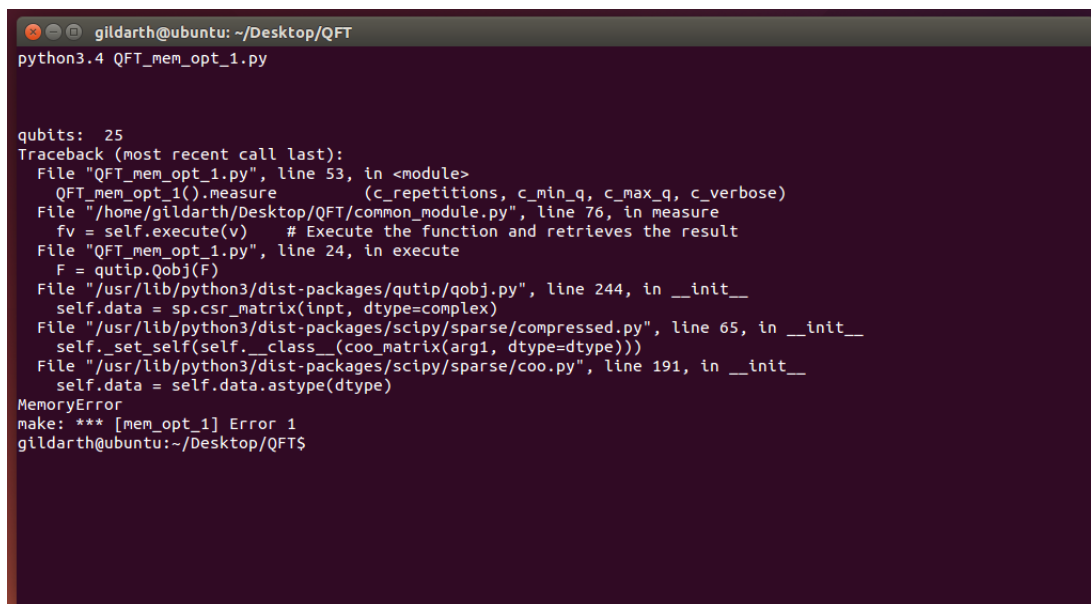
```

gildarth@ubuntu: ~/Desktop/QFT
python3.4 QFT_base_impl.py

qubits: 13
Traceback (most recent call last):
  File "QFT_base_impl.py", line 53, in <module>
    QFT_base_impl().measure (c_repetitions, c_min_q, c_max_q, c_verbose)
  File "/home/gildarth/Desktop/QFT/common_module.py", line 76, in measure
    fv = self.execute(v) # Execute the function and retrieves the result
  File "QFT_base_impl.py", line 34, in execute
    F = qutip.Qobj(F)
  File "/usr/lib/python3/dist-packages/qutip/qobj.py", line 244, in __init__
    self.data = sp.csr_matrix(inpt, dtype=complex)
  File "/usr/lib/python3/dist-packages/scipy/sparse/compressed.py", line 65, in __init__
    self._set(self.__class__(coo_matrix(arg1, dtype=dtype)))
  File "/usr/lib/python3/dist-packages/scipy/sparse/coo.py", line 188, in __init__
    self.data = M[self.row, self.col]
MemoryError
make: *** [base_impl] Error 1
gildarth@ubuntu:~/Desktop/QFT$

```

Figura 7.6: Implementación base - Ejecución para 13 qubits



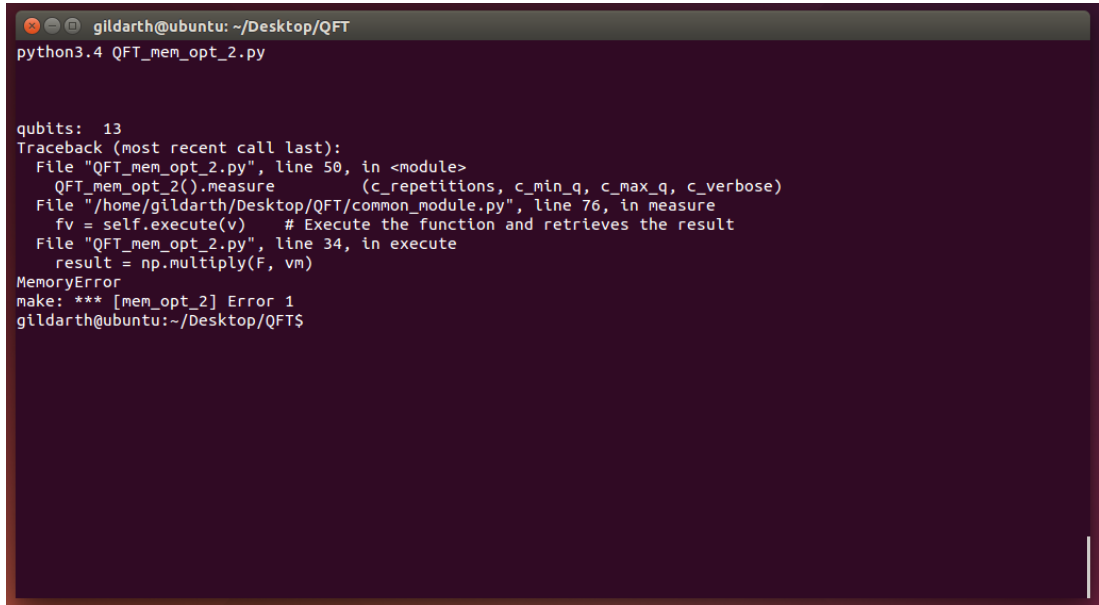
```

gildarth@ubuntu: ~/Desktop/QFT
python3.4 QFT_mem_opt_1.py

qubits: 25
Traceback (most recent call last):
  File "QFT_mem_opt_1.py", line 53, in <module>
    QFT_mem_opt_1().measure (c_repetitions, c_min_q, c_max_q, c_verbose)
  File "/home/gildarth/Desktop/QFT/common_module.py", line 76, in measure
    fv = self.execute(v) # Execute the function and retrieves the result
  File "QFT_mem_opt_1.py", line 24, in execute
    F = qutip.Qobj(F)
  File "/usr/lib/python3/dist-packages/qutip/qobj.py", line 244, in __init__
    self.data = sp.csr_matrix(inpt, dtype=complex)
  File "/usr/lib/python3/dist-packages/scipy/sparse/compressed.py", line 65, in __init__
    self._set(self.__class__(coo_matrix(arg1, dtype=dtype)))
  File "/usr/lib/python3/dist-packages/scipy/sparse/coo.py", line 191, in __init__
    self.data = self.data.astype(dtype)
MemoryError
make: *** [mem_opt_1] Error 1
gildarth@ubuntu:~/Desktop/QFT$

```

Figura 7.7: *mem_opt_1* - Ejecución para 25 qubits



```
gildarth@ubuntu: ~/Desktop/QFT
python3.4 QFT_mem_opt_2.py

qubits: 13
Traceback (most recent call last):
  File "QFT_mem_opt_2.py", line 50, in <module>
    QFT_mem_opt_2().measure(c_repetitions, c_min_q, c_max_q, c_verbose)
  File "/home/gildarth/Desktop/QFT/common_module.py", line 76, in measure
    fv = self.execute(v) # Execute the function and retrieves the result
  File "QFT_mem_opt_2.py", line 34, in execute
    result = np.multiply(F, vm)
MemoryError
make: *** [mem_opt_2] Error 1
gildarth@ubuntu:~/Desktop/QFT$
```

Figura 7.8: *mem_opt_2* - Ejecución para 13 qubits

Con estos datos, no se ve una forma factible de mejorar los resultados obtenidos en *mem_opt_3* sin entrar en la implementación de las librerías utilizadas (*numpy* y *scipy*), por lo que se dio por concluida etapa de refinamiento en el uso de memoria y se empezó el refinamiento del tiempo de ejecución con *mem_opt_3* como implementación de partida.

Optimización del consumo de tiempo de ejecución

Contenidos

8.1. <i>time_opt_1</i>: Reducción de cálculos	99
8.1.1. Diseño	99
8.1.2. Estudio de coste espacial - Teórico	99
8.1.3. Estudio de coste espacial - Práctico	100
8.1.4. Estudio de coste temporal - Teórico	102
8.1.5. Estudio de coste temporal - Práctico	102
8.1.6. Conclusiones	103
8.2. <i>time_opt_2</i>: Ejecución multithread	103
8.2.1. Diseño	104
8.2.2. Estudio de coste espacial - Teórico	106
8.2.3. Estudio de coste espacial - Práctico	107
8.2.4. Estudio de coste temporal - Teórico	109
8.2.5. Estudio de coste temporal - Práctico	112
8.2.6. Conclusiones	112
8.3. <i>time_opt_3</i>: Ejecución multithread con cálculos reducidos . .	113
8.3.1. Diseño	113

8.3.2. Estudio de coste espacial - Teórico	114
8.3.3. Estudio de coste espacial - Práctico	115
8.3.4. Estudio de coste temporal - Teórico	117
8.3.5. Estudio de coste temporal - Práctico	118
8.3.6. Conclusiones	119
8.4. Resultados obtenidos	120

Para optimizar el tiempo de ejecución, se pensó en dos formas: reducir el número de operaciones matemáticas necesarias y paralelizar el algoritmo. La primera se ha implementado en *time_opt_1* pre-calculando los valores del operador que se calculaban varias veces y la segunda se ha implementado en *time_opt_2* haciendo el algoritmo concurrente de forma que cada hilo calcule una parte del resultado de forma paralela. Como implementación final, se han juntado *time_opt_1* y *time_opt_2* en una sola implementación llamada *time_opt_3* que paraleliza tanto el pre-cálculo de los valores del operador como la generación de los operadores parciales y su aplicación

Se eligió partir de la implementación de *mem_opt_3* dado que es la que cumple el objetivo inicial de aumentar el número de qubits operables con el menor tiempo de ejecución.

De igual manera que en las optimizaciones de consumo de memoria, cada implementación realizada se ha ejecutado y medido según lo establecido en la sección 5.1. En la tabla 8.10 y en 8.11 puede observarse respectivamente una comparativa de los resultados obtenidos sobre el coste de memoria y el coste temporal de las diferentes implementaciones de este apartado con la implementación de qutip y *mem_opt_3*. También se les ha aplicado la misma validación que se aplicó a las implementaciones anteriores, pues solo se espera cambiar características no funcionales del algoritmo y por lo tanto debieran ser capaces de pasar las mismas pruebas funcionales.

8.1. *time_opt_1*: Reducción de cálculos

La base de esta optimización radica en que el operador tiene solo N posible valores (w^x para $x \in [0..N]$), con los cuales se genera un operador de N^2 celdas. La idea es pre-calcular los N valores y almacenarlos en un array para luego, a la hora de construir el operador, asignar el valor correspondiente en lugar de calcularlo. Con esto, se reduce la cantidad de operaciones a realizar de N^2 a N , siendo una reducción más que considerable de $N^2 - N = N(N - 1)$ operaciones.

En la tabla 8.3 se observan los resultados obtenidos de las mediciones al ejecutar el algoritmo con hasta 15 qubits y en las tablas 8.1 y 8.2 dos comparativas de memoria y tiempo con *mem_opt_3s*.

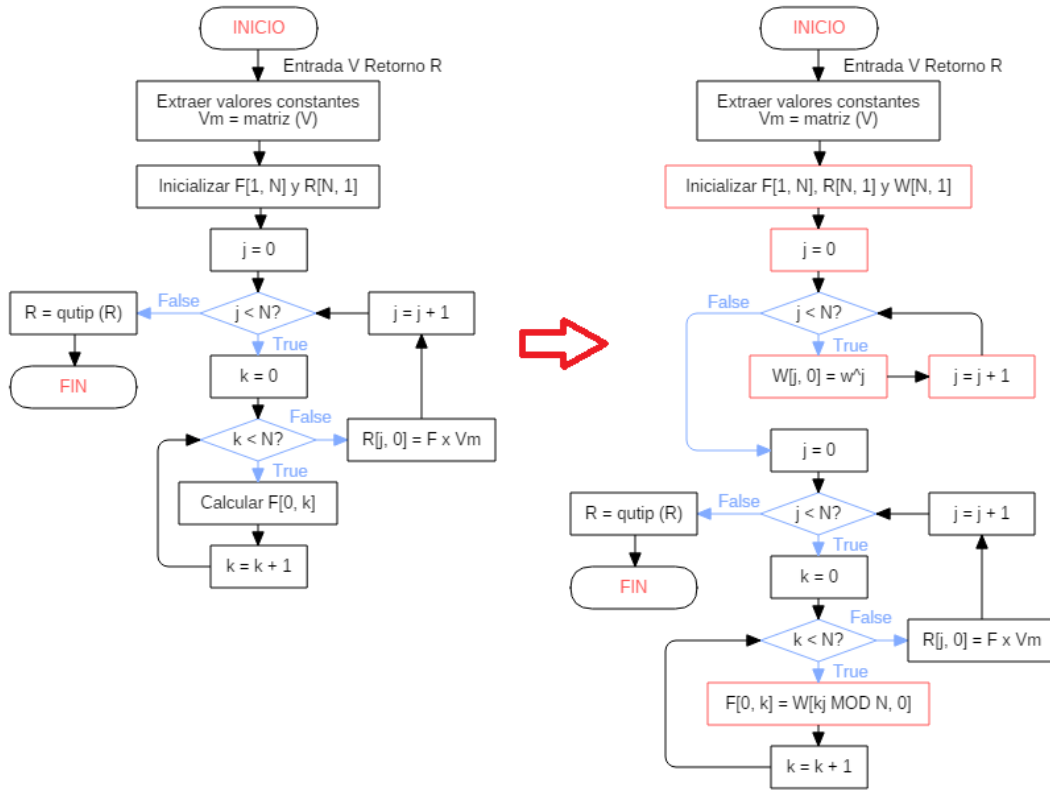
8.1.1. Diseño

Las modificaciones de esta implementación se llevaron a cabo añadiendo un array extra (W) de tamaño N y un bucle que genera el valor asociado de w^x para $x \in [0..N]$ y lo asigna a la celda correspondiente del array W para después en el bucle principal, donde antes se calculaba el valor de cada celda, asignar el que corresponda desde el vector W .

Un diagrama de flujo del algoritmo con los cambios principales respecto a *mem_impl_3* en cajas rojas puede observarse en la figura 8.1

8.1.2. Estudio de coste espacial - Teórico

La complejidad espacial de esta implementación es de $5N = 5 * 2^n$ siendo n el número de qubits, pues la implementación base usada (*mem_opt_3*) tiene complejidad espacial $4N$ y en esta implementación se ha incluido un nuevo vector de tamaño N (para almacenar los valores pre-calculados). Por lo tanto, se ha aumentado en 1 el número de vectores de tamaño N y en consecuencia la complejidad pasa de $4N \Rightarrow 5N$.

Figura 8.1: *time_opt_1* - Diagrama de flujo - Cambios

8.1.3. Estudio de coste espacial - Práctico

En la tabla comparativa 8.1 se puede observar como se siguen pudiendo operar hasta 24 qubits y que, contrariamente a lo esperado, el uso de memoria apenas si ha crecido siendo un incremento de menos de 30MB para 24 qubits (de 1427.51 MB a 1452.88 MB)

Esto supone un resultado muy por debajo del incremento teórico de complejidad espacial que aumentó un 25 % (de $4N$ a $5N$).

Este resultado refuerza la hipótesis sobre la cual se basa *mem_opt_2* de que la conversión a un objeto de qutip (y el propio objeto), consume mucha más memoria que los arrays con los que se construyen.

n	mem_opt_3	time_opt_1
1	42.91797	42.41016
2	42.91797	42.41016
3	42.91797	42.41016
4	42.91797	42.41016
5	42.91797	42.41016
6	42.91797	42.41016
7	42.91797	42.41016
8	42.91797	42.41016
9	42.91797	42.41016
10	42.91797	42.41016
11	42.91797	42.41016
12	43.19531	43.26562
13	43.45312	43.56641
14	44.20703	44.49609
15	45.75391	45.95312
16	48.33594	48.31250
17	54.20312	55.32031
18	64.99609	66.69141
19	86.68750	88.83594
20	129.92969	132.80859
21	216.41797	220.75781
22	389.39062	396.91406
23	735.60547	748.75000
24	1427.51172	1452.88672
25	x	x

Tabla 8.1: time_opt_1 - Comparativa del uso de memoria (MB)

8.1.4. Estudio de coste temporal - Teórico

La complejidad ciclomática de esta implementación es igual que la de *mem_opt_3* con el añadido de recorrer un vector de N elementos para pre-calcular los valores del operador, por lo que se añade un $O(N)$ a la complejidad de *mem_opt_3* ($O(2N^2 + 2N)$) y por lo tanto tenemos

$$O(2N^2 + 2N) + O(N) = O(2N^2 + 2N + N) = O(2N^2 + 3N)$$

que sigue siendo una complejidad cuadrática ($O(N^2)$) respecto a N . Si nos fijamos en la parte no dominante, esta empeora de $2N$ a $3N$, por lo que habrá un aumento del número de vueltas de bucle que implique la ejecución.

No obstante el ligero aumento de complejidad ciclomática, se espera una reducción considerable del tiempo de ejecución por el hecho de que se han de calcular $N(N - 1)$ menos operaciones exponenciales ($e^{\frac{2\pi i}{N}(jk \% N)}$) las cuales conllevan muchos ciclos de cpu, más aun por ser un número complejo el exponente.

8.1.5. Estudio de coste temporal - Práctico

El tiempo de ejecución ha sido considerablemente reducido al tener que hacer solo N operaciones exponenciales frente a las N^2 que se hacían antes. En la tabla 8.2 puede observarse que el tiempo de ejecución para 14 qubits ha pasado de 249 segundos a 129 segundos y para 15 qubits de 1004 segundos a 586 segundos, siendo esto una reducción cercana al 50 %.

El resultado obtenido para 1 qubit no esta acorde al resto de resultados, siendo una media de 0.00345 segundos cuando para 2 qubits la media fue de 0.00065 segundos. Esto se explica con que al inicio de la medición el proceso no debía tener pleno control sobre la cpu (teniendo que turnarse con otros procesos), lo cual se prueba con que la varianza para un qubit sea de casi 0.009 segundos mientras que para 2 qubits fue de 0.00006. segundos

n	mem_opt_3	time_opt_1
1	0.01873	0.00345
2	0.01814	0.00065
3	0.01922	0.00094
4	0.01910	0.00147
5	0.02111	0.00292
6	0.02585	0.00635
7	0.04662	0.01650
8	0.09352	0.05842
9	0.29347	0.16518
10	1.06126	0.62181
11	4.09253	2.21205
12	15.96296	8.39911
13	62.77112	33.14104
14	249.24228	129.69665
15	1003.94503	586.90145

Tabla 8.2: time_opt_1 - Comparativa del consumo de tiempo (s)

8.1.6. Conclusiones

Con los resultados mostrados en las tablas comparativas 8.1 y 8.2 y en la tabla 8.3, se puede asegurar que ciertamente una gran cantidad del tiempo de ejecución se consume repitiendo operaciones ya calculadas previamente, llegando a suponer casi un 50 % del tiempo total de la ejecución dichas operaciones repetidas.

Ademas, no se ha disminuido el número de qubits operables ni ha habido un aumento considerable de memoria utilizada (menos de 30 MB más que en *mem_opt_3* para 24 qubits).

8.2. *time_opt_2*: Ejecución multithread

En esta optimización se va a partir de *mem_opt_3* y se va a paralelizar.

mem_opt_3 es especialmente fácil de paralelizar, pues calcula cada elemento del estado resultante por separado y los agrupa. Por ello, se puede dividir

Qubits	Memoria (MB)	Tiempo (s)	Varianza (s)
1	42.54688	0.00345	0.00896
2	42.54688	0.00065	0.00006
3	42.54688	0.00094	0.00013
4	42.54688	0.00147	0.00004
5	42.54688	0.00292	0.00015
6	42.54688	0.00635	0.00023
7	42.54688	0.01650	0.00072
8	42.54688	0.05842	0.02661
9	42.54688	0.16518	0.00482
10	43.07031	0.62181	0.09528
11	43.32812	2.21205	0.08540
12	44.28125	8.39911	0.05594
13	46.14453	33.14104	0.54002
14	49.95312	129.69665	1.00550
15	56.67969	586.90145	7.49229

Tabla 8.3: *time_opt_1* - Resultados de las mediciones

el procesado de estos elementos entre distintos *threads* (hilos de ejecución) para finalmente juntar todos los resultados. Con ello, se espera acelerar considerablemente el cálculo del resultado, más a cuantos más núcleos tenga el equipo donde se ejecute, sin reducir por ello el número máximo de qubits operables (24 en *mem_opt_1*).

A esta implementación se le ha asignado el identificador *time_opt_2*.

8.2.1. Diseño

La arquitectura utilizada para esta paralelización es un *Master/Slave*.

La idea original era utilizar concurrencia dentro de un mismo proceso, teniendo el estado inicial y el resultado como datos globales compartidos para cada *thread*, pero eso ha resultado imposible por una limitación impuesta por el propio *python*, en el cual aunque existe la clase *Thread* dentro de la librería *Threading* (<https://docs.python.org/3.4/library/threading.html>), esta no permite aprovecharse de los procesadores de varios núcleos pues un proceso del intérprete

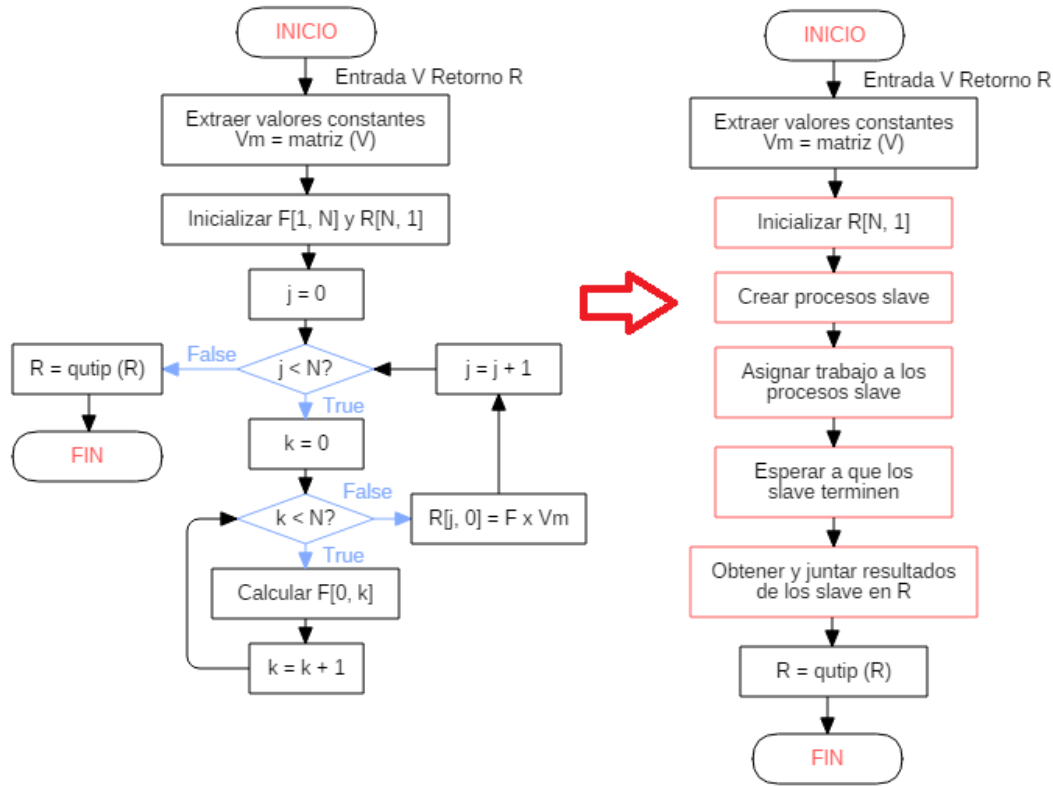
te de *python* solo puede ejecutarse en un núcleo y por lo tanto aunque se creen *threads* todos competirán por el mismo núcleo, nunca ejecutándose realmente en paralelo.

Para solventar este problema se utiliza una aproximación de varios procesos hijos paralelos usando la librería de *multiprocessing*, incluida en el propio lenguaje. Dicha librería permite, a alto nivel, generar un *pool* de procesos, mandarles ejecutar a cada uno una función concreta con unos parámetros y recibir el resultado de la ejecución cuando termine.

Para usar dicha librería se hizo una función que recibe por parámetro el estado de entrada y lo necesario para saber que rango de valores del resultado debe calcular. En el método de ejecución de la transformada (que hace el papel de *Master* representado en el diagrama de la figura 8.2), se generan un *pool* con tantos procesos (*Slaves*) como núcleos tenga el sistema y se manda a cada proceso calcular una parte del resultado mediante la función mencionada (representada en el diagrama de la figura 8.3). Una vez todos los procesos han terminado el *Master* los resultados y se juntan en orden, formando así el estado resultante de aplicar la transformación.

Unos diagramas de flujo del algoritmo con los cambios principales respecto a *mem_impl3* en cajas rojas puede observarse en la figura 8.2, la cual se corresponde al proceso *Master*, y la figura 8.3 la correspondiente al diagrama de flujo de los procesos *Slave*

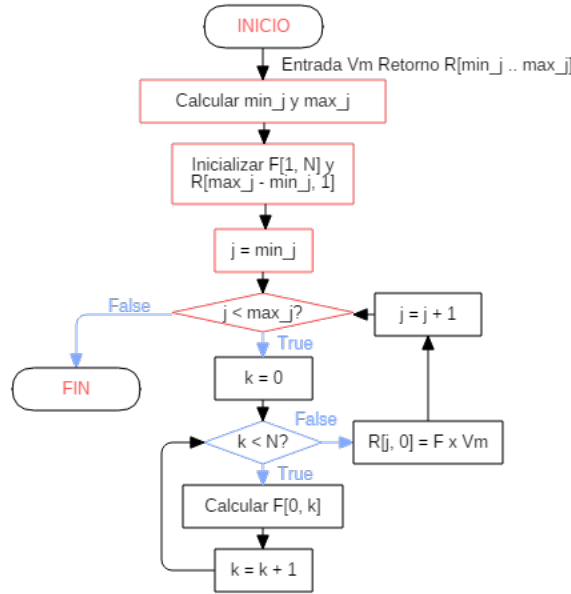
Por ello, se ha de modificar el método de medición de la memoria consumida. Convenientemente se usa una función de la librería ofrecida por *python*, la cual devuelve el máximo de memoria usada por el proceso actual, pero que modificando una constante que se le pasa por parámetro, devuelve la memoria máxima consumida por los procesos hijo del proceso actual. Por lo tanto, se añadió una nueva llamada a la función con la constante para que devuelva la memoria de los hijos y se sumo a la del padre, siendo dicha suma el resultado de la medición.

Figura 8.2: *time_opt_2* - Master - Diagrama de flujo - Cambios

8.2.2. Estudio de coste espacial - Teórico

El coste en memoria se esperaba que fuera superior al de la implementación de partida (sobretudo para pocos qubits), pues cada proceso hijo es un proceso independiente y “adquiere” una cantidad mínima de memoria al iniciarse aunque esta no se use. Aun así, puesto que no se transforma a un objeto de qutip el operador ni el resultado en los procesos *Slave*, estos no debieran tener un consumo muy excesivamente grande en comparación con el total, siendo el consumo de memoria de estos solo el de la matriz del estado de entrada, la matriz operador y la matriz con la parte del resultado correspondiente. Eso es

$$(N + N + N/p) \times p = \left(\frac{pN + pN + N}{p} \right) p = pN + pN + N = N(2p + 1)$$

Figura 8.3: *time_opt_2* - Slave - Diagrama de flujo

siendo p el número de procesos *Slave*. Si esto lo unimos al consumo del padre, el cual contiene los objetos *qutip* de entrada y de salida ($2N$), así como la matriz del estado de entrada que se manda a los procesos y la del resultado donde se recogen los resultados de cada proceso ($2N$), tenemos un total de

$$2N + 2N + N(2p + 1) = 4N + N(2p + 1) = N(2p + 1 + 4) = N(2p + 5)$$

frente a la complejidad de $4N$ que tiene *mem_impl_3*.

8.2.3. Estudio de coste espacial - Práctico

En lo referente al número máximo de qubits operables, el resultado es el mismo que en la implementación de partida (*mem_opt_3*) como se observa en la tabla 8.4, pudiéndose operar hasta el máximo de 24 qubits como se muestra en la tabla comparativa 8.4, donde además se observa que el incremento del uso de memoria respecto a la implementación de partida es casi nulo, siendo de 1400 MB para 24 qubits.

Sorprendentemente al ejecutar el código, aunque cada núcleo trabaja al 100 % de su capacidad, el consumo de memoria apenas aumenta y, al mostrar por terminal la memoria usada por padre e hijos, la de los hijos resulta ser 0.0 MB, siendo la ocupada por el padre íntegramente la usada por el programa. Esto se confirma al observar en el administrador del sistema que el consumo de memoria aumenta una cantidad similar a la ocupada por el padre.

Pese a que los valores de la medición y el administrador del sistema coinciden, se realizó un segundo método para medir la memoria consumida el cual la calcula mediante llamadas al sistema encapsuladas por la librería *psutil*, iterando sobre la información de los procesos *slave* (a la que accede desde la información del proceso *master*) y suma la memoria consumida por cada uno.

En la imagen 8.4 se puede observar el estado del sistema antes, durante y después de la ejecución del algoritmo para 14 qubits, así como la información mostrada por terminal de los dos métodos descritos para medir la memoria consumida. Se puede comprobar que el sistema funciona de forma paralela pues los 4 núcleos trabajan a plena capacidad, y el PID de los procesos *slave* (children) es diferente al del proceso *master* (father). Así mismo, se puede observar que la memoria ocupada antes de la ejecución es de 517 MiB y durante su ejecución es 559 MiB, lo que hace una diferencia de 42 MiB \approx 40 MB. El método de medida proceso por proceso calcula un consumo total de memoria de 180 MB lo cual no se corresponde con los datos del monitor del sistema. Por otro lado, el método que trata los hijos como un todo, considera que estos tienen un consumo de 0.0 MB y el total es el consumo del padre, siendo este 47.87 MB, resultado que está muchísimo más cerca de lo observado en el monitor del sistema.

La conclusión a la que se llegó después de investigar, es que la librería de python, pese a que crea procesos diferentes, debe hacer algo con la memoria de manera que los hijos acceden a la memoria del padre en lugar de a un espacio de memoria propio, y por ende la memoria que ocupan es de 0.0, siendo gran parte de esta compartida entre los 4 (por ejemplo el array con el estado de entrada que se le pasa como parámetro a la función que ejecuta cada proceso *Slave* o el código del propio intérprete de *python*). Esto explicaría que la medir con la librería de *python* desde el proceso *Master* (que es el proceso padre) se considere que los hijos

no tienen consumo de memoria, pero que a la hora de consultar el consumo desde el propio proceso *Slave* (proceso hijo) se obtenga como resultado un consumo de 33 MB por proceso *Slave* (valor mostrado en el terminal de abajo de la figura 8.4), ya que la información de que esa memoria en realidad es compartida debe de ser accesible solo desde el proceso *Master*.

Se observa también, mediante el nuevo método de medición por procesos, un proceso hijo extra con un consumo de 0.0 MB, lo cual no era esperado. Este proceso se ha comprobado que aparece también para aquellas implementaciones no paralelas, por lo que se ha deducido que no tiene que ver con el *pool* de procesos usado ni con estos, y por ello no se le ha dado mayor importancia.

Con esto se ha mantenido el objetivo de paralelizar la ejecución y seguirse pudiendo operar con hasta 24 qubits, sin que por ello se haya aumentado en exceso el consumo de memoria.

8.2.4. Estudio de coste temporal - Teórico

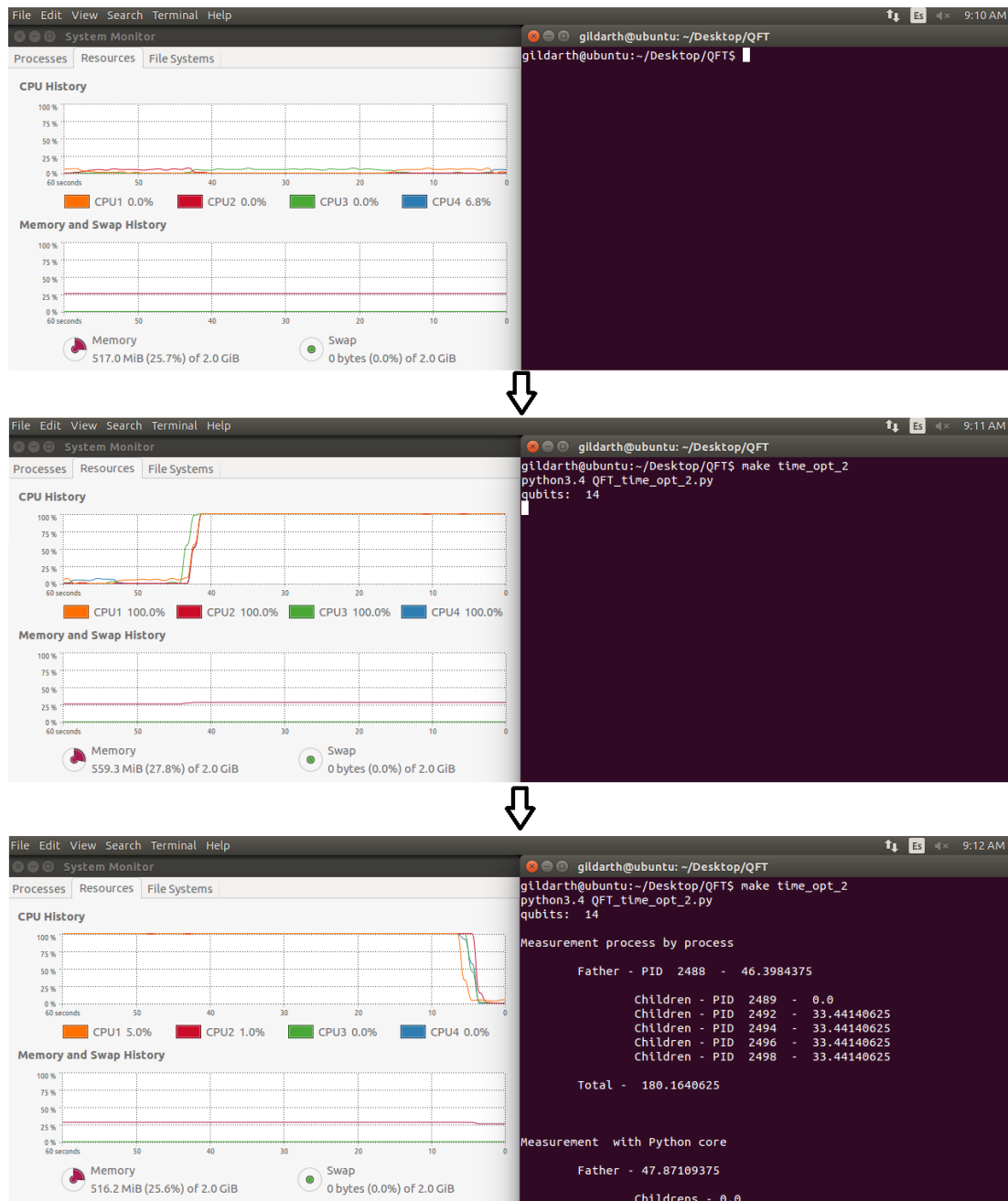
El comportamiento esperado para esta implementación es un aumento en el tiempo de ejecución para estados pequeños (para los cuales es mas costoso el lanzar los procesos que realizar la operación directamente), y una reducción creciente y muy considerable según se aumenta el número de qubits.

Esta reducción debiera ser proporcional al número de procesos y en un mundo ideal el tiempo de ejecución sería $1/P$ del tiempo original, siendo P el número de procesos, pero en realidad se espera que esté un poco por encima pues hay que añadir por lo menos el tiempo de creación de procesos, paso de los datos de entrada, recepción de los datos de salida y conversión a objeto de *qutip*, funciones las cuales realiza el proceso *Master*.

Las modificaciones del algoritmo solo son en lo referente al bucle principal, estando esta ahora dividido entre P procesos. Por este motivo, la complejidad ciclomática se ve modificada con respecto al bucle principal que es el término N^2 de la complejidad ciclomática de *mem_opt_3* ($O(2N^2 + 2N)$).

n	mem_opt_3	time_opt_2
1	42.91797	43.57812
2	42.91797	43.57812
3	42.91797	43.57812
4	42.91797	43.57812
5	42.91797	43.57812
6	42.91797	43.57812
7	42.91797	43.57812
8	42.91797	43.57812
9	42.91797	43.57812
10	42.91797	43.57812
11	42.91797	43.57812
12	43.19531	43.57812
13	43.45312	43.69531
14	44.20703	44.15234
15	45.75391	45.58203
16	48.33594	48.39844
17	54.20312	54.23438
18	64.99609	64.86328
19	86.68750	86.55078
20	129.92969	129.79688
21	216.41797	216.28125
22	389.39062	389.26172
23	735.60547	735.85156
24	1427.51172	1427.64453
25	x	x

Tabla 8.4: *time_opt_2* - Comparativa del uso de memoria (MB)

Figura 8.4: *time_opt_2* - Medición de memoria con el *core* de *python*

Este $O(N^2)$ original se descompone en un primer bucle que cuenta los términos del resultado a calcular y en un segundo bucle anidado para generar el operador parcial para el término del resultado correspondiente. El bucle interno no se ha modificado ($O(N)$), pero el bucle exterior se ha dividido equitativamente entre

los P procesos ($O(\frac{N}{P})$). Todo ello repetido dentro de los P procesos, pero aunque se realiza P veces, se hace de forma simultánea y por lo tanto no se ha contado en la complejidad ciclomática, la cual es

$$O(\frac{N}{P}N) + O(2N) = O(\frac{N^2}{P} + 2N)$$

que coincide con lo teorizado en los párrafos anteriores sobre que el consumo temporal es inversamente proporcional al número de procesos de que se disponga.

8.2.5. Estudio de coste temporal - Práctico

Con todo esto, en la tabla 8.6 se puede observar los resultados de las mediciones con hasta 15 qubits, en la tabla 8.5 una comparativa con *mem_opt_3* y en 8.11 la comparativa con otras implementaciones de tiempo de ejecución, donde se observa la esperada reducción drástica y se comprueba que según va aumentando el número de qubits, el tiempo requerido tiende a estar entre $1/3 \approx 33\%$ y $2/5 = 40\%$ del requerido por *mem_opt_3*, lo cual concuerda con lo que se esperaba pues al ser un sistema con 4 núcleos se estimó que requeriría algo mas de $1/4 = 25\%$ del tiempo de *mem_opt_3*.

También se puede observar, en concordancia con lo esperado, un tiempo mínimo de ejecución alrededor de los 0.05 segundos, lo cual se debe al tiempo necesario para la creación de procesos, distribución del trabajo y recogida de resultados.

8.2.6. Conclusiones

Se ha conseguido paralelizar la implementación de *mem_opt_3* reduciendo considerablemente el tiempo de ejecución, manteniendo el número máximo de qubits operables.

Además, para 24 qubits el consumo de memoria es prácticamente igual al de *mem_opt_3*, lo que refuerza la conclusión de que python realiza memoria compar-

n	mem_opt_3	time_opt_2
1	0.01873	0.05523
2	0.01814	0.05791
3	0.01922	0.04330
4	0.01910	0.04675
5	0.02111	0.04410
6	0.02585	0.04949
7	0.04662	0.05933
8	0.09352	0.08112
9	0.29347	0.16032
10	1.06126	0.47548
11	4.09253	1.46610
12	15.96296	5.49805
13	62.77112	20.82074
14	249.24228	83.20696
15	1003.94503	352.20139

Tabla 8.5: time_opt_2 - Comparativa del consumo de tiempo (s)

tida entre procesos y que el grueso del consumo de memoria esta en los objetos de qutip y no en las matrices individuales.

8.3. *time_opt_3*: Ejecución multithread con cálculos reducidos

Viendo los buenos resultados de *time_opt_1* y *time_opt_2* se ha decidido hacer una implementación con ambas mejoras, a la cual se le ha asignado el identificador *time_opt_3*

8.3.1. Diseño

Como diseño se ha cogido el mismo que en *time_opt_2* y se ha modificado para que pre-calcule los posibles valores del operador en un array que se le pasa a la función que realiza los cálculos. Tras comprobar su correcto funcionamiento, se ha aprovechado la existencia de los procesos para paralelizar también el pre-cálculo

Qubits	Memoria (MB)	Tiempo (s)	Varianza (s)
1	44.16016	0.05523	0.03290
2	44.16016	0.05791	0.01164
3	44.16016	0.04330	0.00888
4	44.16016	0.04675	0.01043
5	44.16016	0.04410	0.00509
6	44.16016	0.04949	0.00367
7	44.16016	0.05933	0.00562
8	44.16016	0.08112	0.00682
9	44.16016	0.16032	0.01387
10	44.31641	0.47548	0.08794
11	44.71484	1.46610	0.07870
12	45.67578	5.49805	0.17024
13	47.82031	20.82074	0.16353
14	51.87109	83.20696	0.76349
15	57.12109	352.20139	1.40222

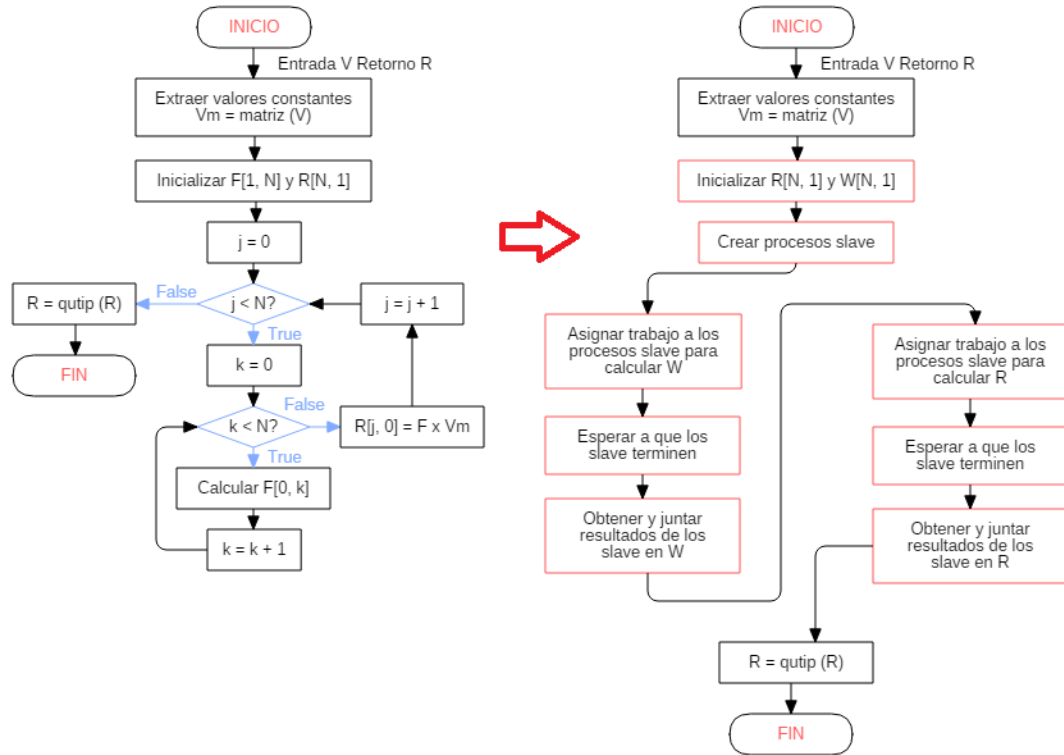
Tabla 8.6: *time_opt_2* - Resultados de las mediciones

de los posibles valores del operador de forma análoga a la usada para calcular el resultado. El diagrama de flujo del proceso *Master* puede observarse en la figura 8.5 y los diagrama de flujo de los procesos *Slave* se pueden observar en la figura 8.6.

8.3.2. Estudio de coste espacial - Teórico

En lo referente al coste de memoria, al igual que en *time_opt_1* se espera un aumento de la memoria utilizada respecto a *mem_opt_3*, pues ahora hay otro dato global de N elementos (la matriz con los valores pre-calculados) el cual se le pasa a mayores del estado de entrada a cada proceso *slave* a fin de poderse utilizar para construir el operador. Con esto, a la complejidad espacial de *time_opt_2* se le ha de añadir el vector de N elementos usado para almacenar los valores pre-calculados, y por lo tanto la complejidad es

$$N(2p + 5) + N = N(2p + 5 + 1) = N(2p + 6)$$

Figura 8.5: *time_opt_3* - Master - Diagrama de flujo - Cambios

la parte de la ejecución donde se pre-calculan estos valores no se ha añadido en el cálculo, pues los procesos usados son los mismos que se usan luego para el cálculo de la transformada, siendo el consumo de memoria de la segunda vez muy superior al que tendrá en la primera y por lo tanto lo engloba.

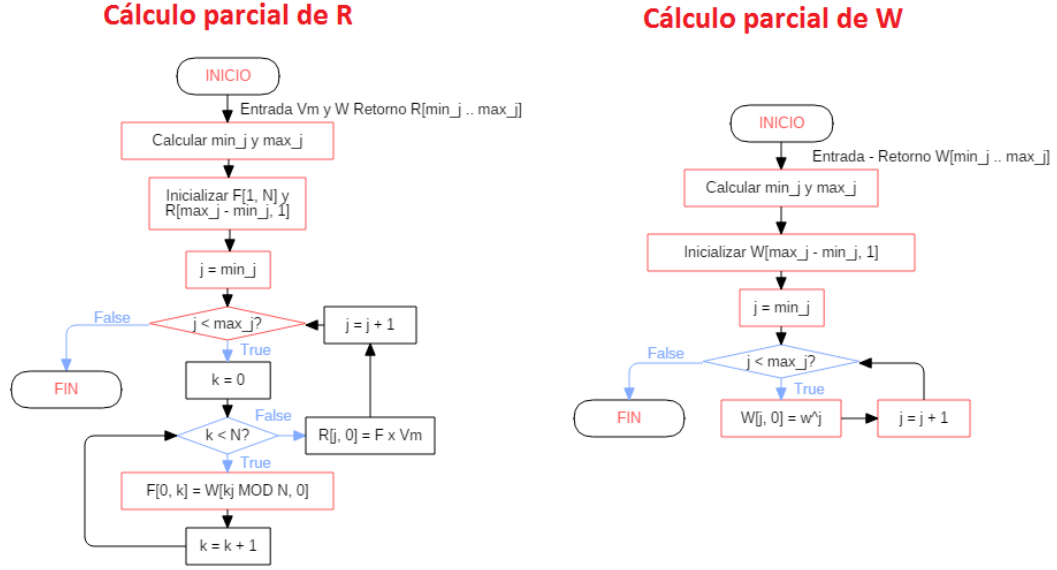
8.3.3. Estudio de coste espacial - Práctico

Los resultados de la ejecución se muestran en la tabla 8.8 y en la tabla 8.7 se compara el consumo de memoria con el de **mem_opt_3**.

Se ha mantenido en 24 el número de qubits esperable y el consumo de memoria ha sido solo ligeramente superior al de **mem_opt_3**, siendo casi idéntico al de **time_opt_2** como se ve en la tabla comparativa 8.10.

n	mem_opt_3	time_opt_3
1	42.91797	43.62109
2	42.91797	43.62109
3	42.91797	43.62109
4	42.91797	43.62109
5	42.91797	43.62109
6	42.91797	43.62109
7	42.91797	43.62109
8	42.91797	43.62109
9	42.91797	43.62109
10	42.91797	43.62109
11	42.91797	43.69141
12	43.19531	43.78906
13	43.45312	44.43359
14	44.20703	45.58203
15	45.75391	47.60938
16	48.33594	51.37500
17	54.20312	55.86719
18	64.99609	67.33984
19	86.68750	90.83203
20	129.92969	138.07031
21	216.41797	232.56250
22	389.39062	421.53516
23	735.60547	800.13281
24	1427.51172	1421.3125
25	x	x

Tabla 8.7: *time_opt_3* - Comparativa del uso de memoria (MB)

Figura 8.6: *time_opt_3* - Slave - Diagramas de flujo

8.3.4. Estudio de coste temporal - Teórico

Para el calculo de la complejidad ciclomática, se va a partir de los resultados obtenidos en *time_opt_2* y se va a añadir el proceso de pre-calcular los valores de forma paralela.

En se llego a la conclusión de que la complejidad ciclomática era de $O(\frac{N^2}{P} + 2N)$, a la cual ahora hay que añadir el cálculo de N valores entre P procesos

$$O(\frac{N^2}{P} + 2N) + O(\frac{N}{P}) = O(\frac{N^2}{P} + 2N + \frac{N}{P}) = O(\frac{N^2 + N}{P} + 2N)$$

no obstante, al ser dos optimizaciones compatibles las que se han juntado, el tiempo de ejecución se espera se reduzca con respecto al resto de implementaciones.

n	mem_opt_3	time_opt_3
1	0.01873	0.07445
2	0.01814	0.08768
3	0.01922	0.08578
4	0.01910	0.08457
5	0.02111	0.08111
6	0.02585	0.08659
7	0.04662	0.10730
8	0.09352	0.09962
9	0.29347	0.15352
10	1.06126	0.35369
11	4.09253	0.99499
12	15.96296	3.37213
13	62.77112	12.96976
14	249.24228	50.79075
15	1003.94503	226.98583

Tabla 8.8: *time_opt_3* - Comparativa del consumo de tiempo (s)

8.3.5. Estudio de coste temporal - Práctico

Como se esperaba, al pre-calcular los valores posibles del operador y paralelizar los cálculos, se ha conseguido reducir aun más el tiempo de ejecución, como se ve en la tabla de resultados 8.9 y en la comparativa de la tabla 8.8.

El tiempo de ejecución para 10 qubits y más se ha mantenido por debajo del 25% respecto a los resultados de *mem_opt_3*. Para menos de 10 qubits el tiempo de ejecución es mayor que en *mem_opt_3* debido al tiempo usado en crear y gestionar los procesos *Slave*.

En la tabla 8.11 se puede observar la comparativa con el resto de implementaciones, donde se ve, para 15 qubits, una reducción de más de 100 segundos respecto a *time_opt_2* (lo que supone una reducción de más de un 30%) y de más de 300 segundos con *time_opt_1* (siendo una reducción de más del 50%).

Qubits	Memoria (MB)	Tiempo (s)	Varianza (s)
1	44.24609	0.07445	0.01457
2	44.24609	0.08768	0.01604
3	44.24609	0.08578	0.02390
4	44.24609	0.08457	0.01701
5	44.24609	0.08111	0.01778
6	44.24609	0.08659	0.01261
7	44.24609	0.10730	0.03136
8	44.24609	0.09962	0.00855
9	44.24609	0.15352	0.02529
10	44.46484	0.35369	0.02176
11	44.98047	0.99499	0.03840
12	45.73828	3.37213	0.10449
13	48.23047	12.96976	0.26093
14	51.62891	50.79075	0.51136
15	57.72656	226.98583	1.91672

Tabla 8.9: *time_opt_3* - Resultados de las mediciones

8.3.6. Conclusiones

Las dos mejoras que se han juntado en esta implementación han resultado bastante compatibles, consiguiendo unos tiempos de ejecución mucho menores que los de la implementación de partida como se ve en la tabla 8.9 (para 15 qubits de 1003 segundos en *mem_opt_3* a 227 segundos, lo que supone una reducción de más de un 75 %), y suponiendo mejoras sustanciales respecto a cada una de las implementaciones por separado.

Por otro lado, no se ha reducido el número de qubits ejecutables ni se ha incrementado demasiado el consumo de memoria respecto a la implementación de partida (*mem_opt_3*).

Todo esto conllevan el considerar como positivos los resultados obtenidos de la implementación conjunta de *time_opt_1* y *time_opt_2*.

n	qutip_impl	mem_opt_3	time_opt_1	time_opt_2	time_opt_3
1	42.34375	42.91797	42.41016	43.57812	43.62109
2	42.34375	42.91797	42.41016	43.57812	43.62109
3	42.34375	42.91797	42.41016	43.57812	43.62109
4	42.34375	42.91797	42.41016	43.57812	43.62109
5	42.34375	42.91797	42.41016	43.57812	43.62109
6	42.67578	42.91797	42.41016	43.57812	43.62109
7	43.70703	42.91797	42.41016	43.57812	43.62109
8	47.84375	42.91797	42.41016	43.57812	43.62109
9	64.57031	42.91797	42.41016	43.57812	43.62109
10	130.57812	42.91797	42.41016	43.57812	43.62109
11	394.74219	42.91797	42.41016	43.57812	43.69141
12	1450.74219	43.19531	43.26562	43.57812	43.78906
13	x	43.45312	43.56641	43.69531	44.43359
14	x	44.20703	44.49609	44.15234	45.58203
15	x	45.75391	45.95312	45.58203	47.60938
16	x	48.33594	48.31250	48.39844	51.37500
17	x	54.20312	55.32031	54.23438	55.86719
18	x	64.99609	66.69141	64.86328	67.33984
19	x	86.68750	88.83594	86.55078	90.83203
20	x	129.92969	132.80859	129.79688	138.07031
21	x	216.41797	220.75781	216.28125	232.56250
22	x	389.39062	396.91406	389.26172	421.53516
23	x	735.60547	748.75000	735.85156	800.13281
24	x	1427.51172	1452.88672	1427.64453	1421.3125
25	x	x	x	x	x

Tabla 8.10: time_opt - Comparativa del uso de memoria (MB)

8.4. Resultados obtenidos

Una vez implementadas y medidas las optimizaciones del consumo de tiempo de ejecución, se han reunido sus resultados en dos tables comparativas, a las cuales se les ha añadido la información de la implementación de *qutip* y la de la implementación de partida utilizada para estas implementaciones (*mem_opt_3*). Estos resultados pueden observarse en las tablas 8.10 y 8.11.

La valoración no obstante, puesto que no se va a continuar el trabajo de momento, se deja para una valoración global de todo el trabajo

n	qutip_impl	mem_opt_3	time_opt_1	time_opt_2	time_opt_3
1	0.00108	0.01873	0.00345	0.05523	0.07445
2	0.00104	0.01814	0.00065	0.05791	0.08768
3	0.00121	0.01922	0.00094	0.04330	0.08578
4	0.00127	0.01910	0.00147	0.04675	0.08457
5	0.00163	0.02111	0.00292	0.04410	0.08111
6	0.00333	0.02585	0.00635	0.04949	0.08659
7	0.01037	0.04662	0.01650	0.05933	0.10730
8	0.03920	0.09352	0.05842	0.08112	0.09962
9	0.14529	0.29347	0.16518	0.16032	0.15352
10	0.52456	1.06126	0.62181	0.47548	0.35369
11	2.01386	4.09253	2.21205	1.46610	0.99499
12	7.97829	15.96296	8.39911	5.49805	3.37213
13	x	62.77112	33.14104	20.82074	12.96976
14	x	249.24228	129.69665	83.20696	50.79075
15	x	1003.94503	586.90145	352.20139	226.98583

Tabla 8.11: time_opt - Comparativa del consumo de tiempo (s)

Capítulo 9

Seguimiento

Contenidos

9.1. Coste final	124
-----------------------------------	------------

En este capítulo se va a exponer como ha ido el desarrollo del proyecto con respecto al esfuerzo realizado en las diferentes tareas. Para ello, se instauro linea base con lo establecido en el apartado de planificación, y se fueron introduciendo los datos reales de esfuerzo realizado.

Al final, el proyecto completo a conllevado un esfuerzo total de 350 horas, suponiendo con ello un desvío respecto a la planificación inicial de 59 horas (más o menos un 20 % sobre la estimación inicial).

Una comparativa de las tareas de las etapas puede verse en la figura 9.6.

Este desvío se ha visto producido principalmente por la redacción de la memoria, que ha tenido un tiempo final de 150 horas sobre las 108 estimadas inicialmente, estando concentradas casi todas esas horas de más en la revisión final, donde se ha modificado la estructura de capítulos y secciones a la vez que se simplificaban las explicaciones matemáticas y físicas del contexto para hacerlo más asequible al lector.

Los diagramas de Gantt del seguimiento se muestran en las siguientes figuras: el diagrama general en al figura 9.1, la etapa previa y la inicial en la figura 9.2, la

etapa de optimización espacial esta en la figura 9.3 y la de optimización temporal en la figura 9.4. Por último, el seguimiento del esfuerzo invertido en la memoria se puede observar en la figura 9.5.

9.1. Coste final

Puesto que el proyecto al final ha tenido un coste en esfuerzo de 350 horas, el coste real, aplicando el coste total por hora de analista programador junior de 30€/hora, ha sido de

$$350horas \times 30\frac{\text{€}}{\text{hora}} = 10500\text{€}$$

frente a los 8730€ estimados inicialmente, suponiendo unas pérdidas frente al coste inicial para el desarrollador/empresario de 1770€.

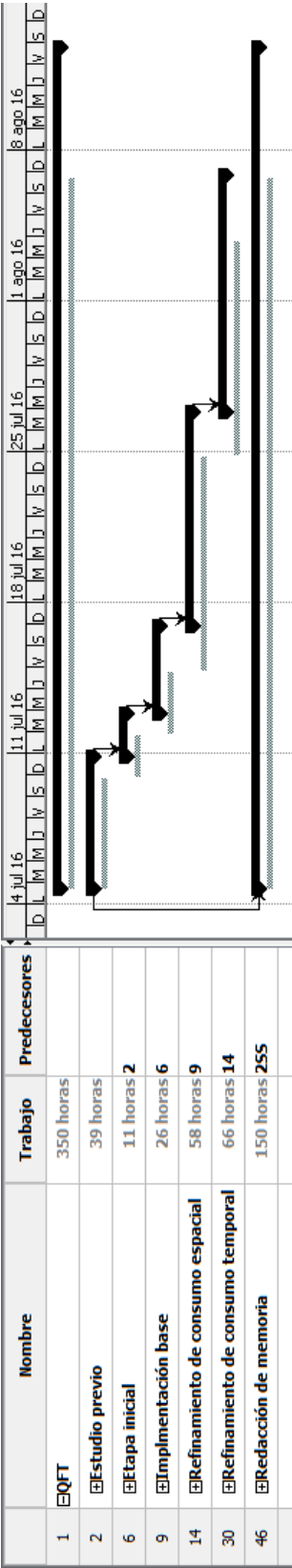


Figura 9.1: Diagrama de Gantt - Seguimiento: General

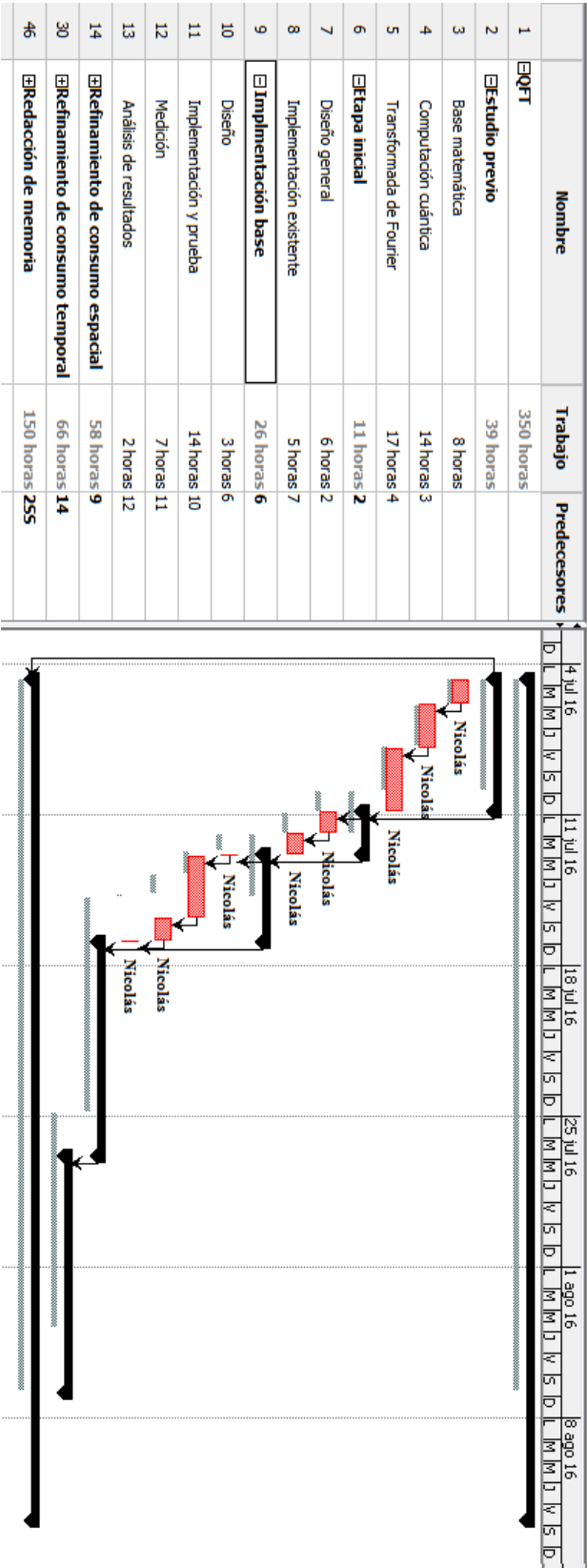


Figura 9.2: Diagrama de Gantt - Seguimiento: Etapas previas

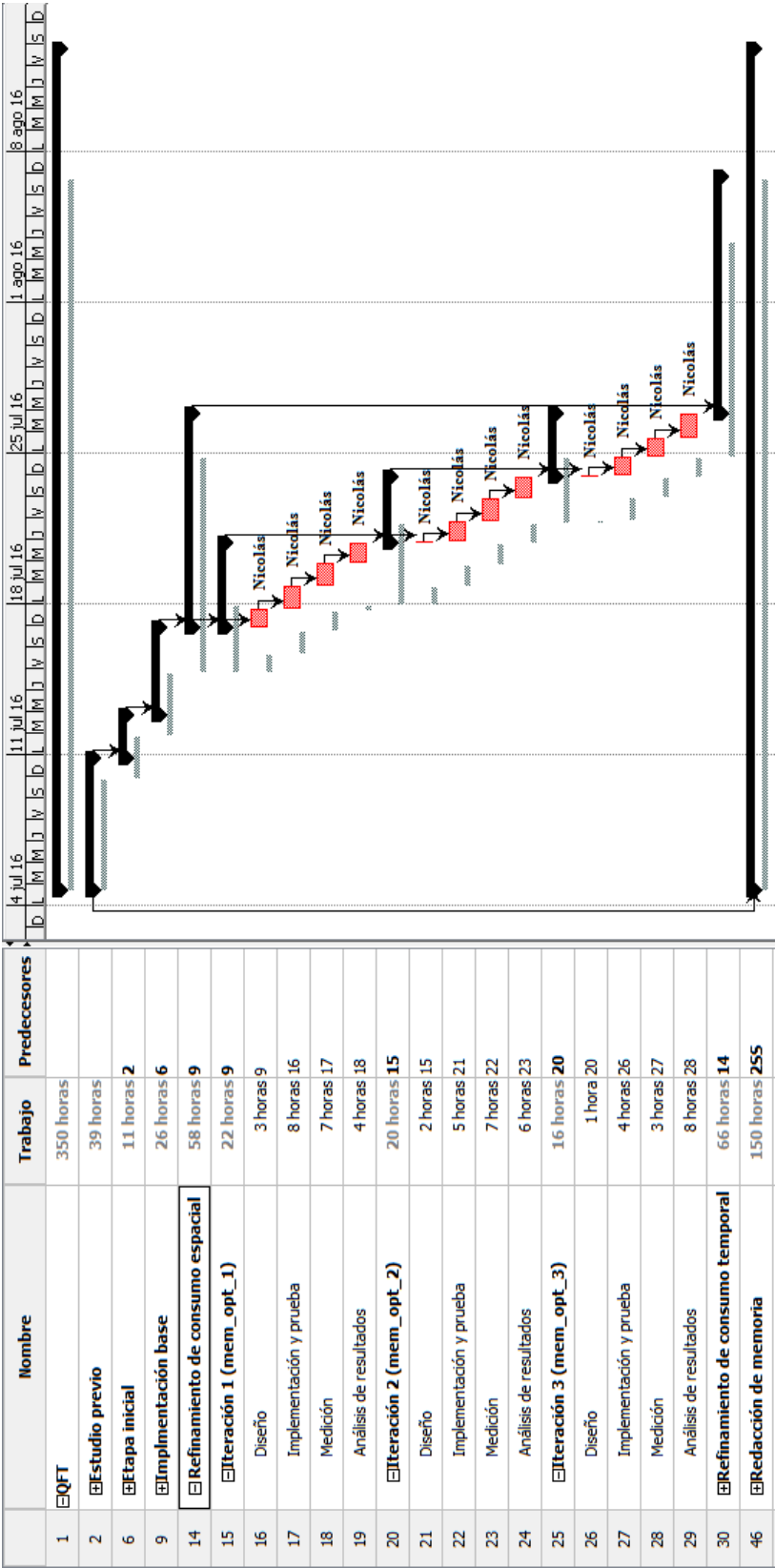


Figura 9.3: Diagrama de Gantt - Seguimiento: Etapa de refinamiento del consumo de memoria

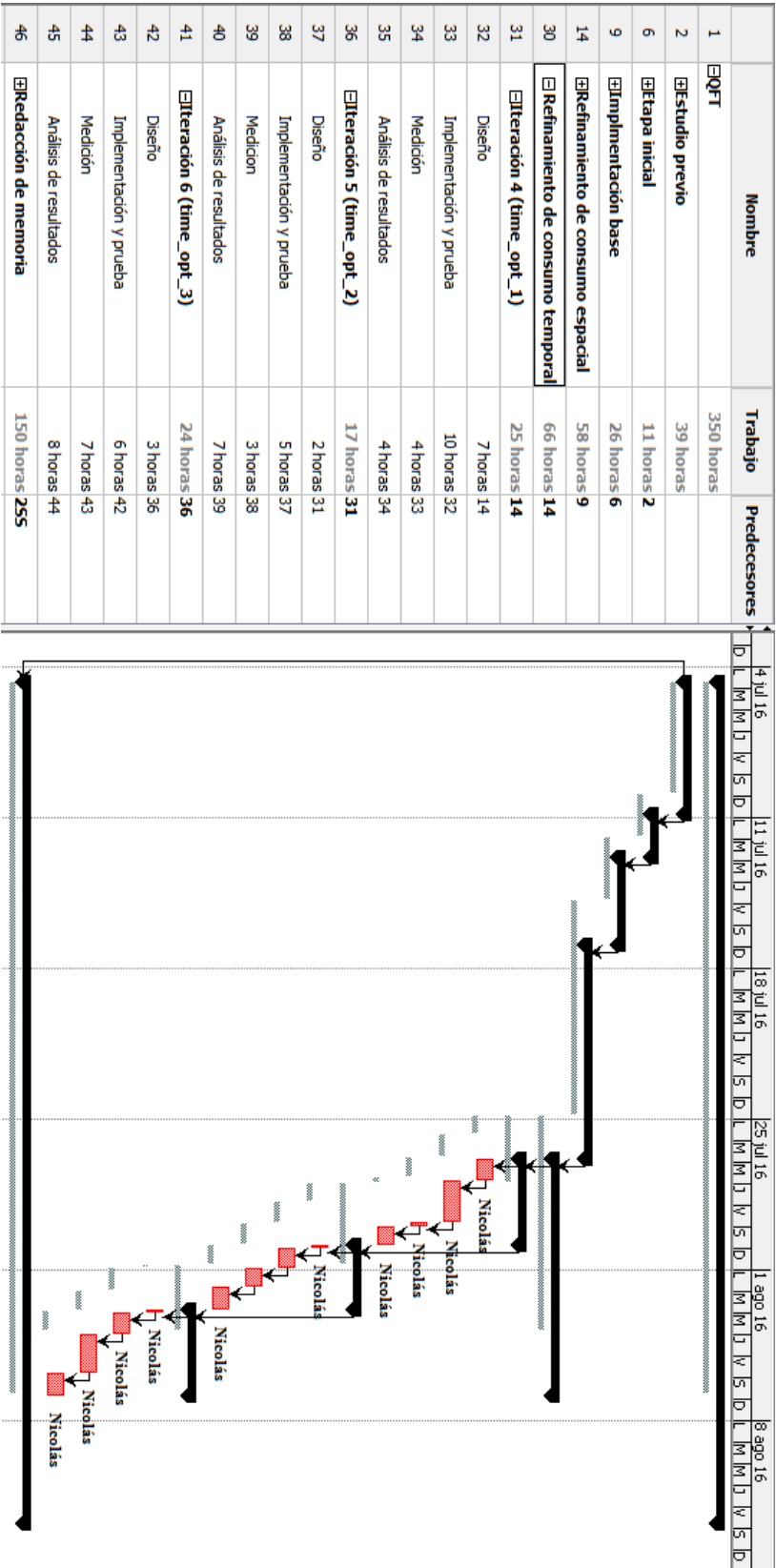


Figura 9.4: Diagrama de Gantt - Seguimiento: Etapa de refinamiento del consumo de tiempo

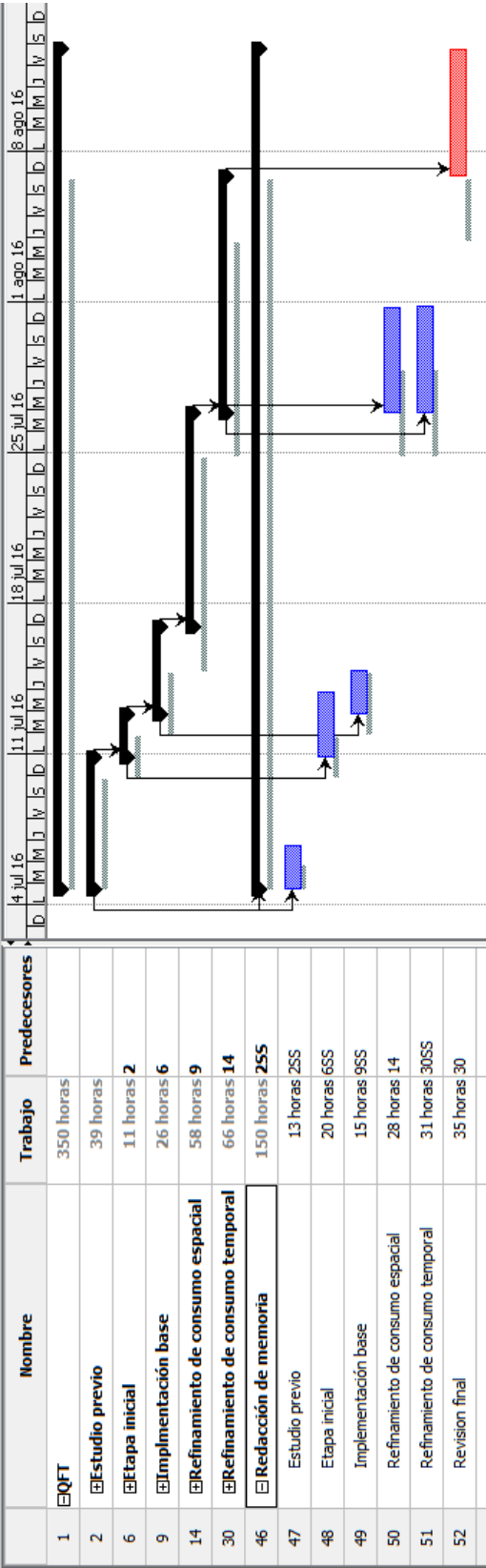


Figura 9.5: Diagrama de Gantt - Seguimiento: Redacción de memoria

	Nombre	Trabajo	Predcesores
1	☒ QFT	291 horas	
2	☒ Estudio previo	35 horas	
6	☒ Etapa inicial	12 horas	2
9	☒ Implementación base	16 horas	6
14	☒ Refinamiento de consumo espacial	60 horas	9
30	☒ Refinamiento de consumo temporal	60 horas	14
46	☒ Redacción de memoria	108 horas	255

	Nombre	Trabajo	Predcesores
1	☒ QFT	350 horas	
2	☒ Estudio previo	39 horas	
6	☒ Etapa inicial	11 horas	2
9	☒ Implementación base	26 horas	6
14	☒ Refinamiento de consumo espacial	58 horas	9
30	☒ Refinamiento de consumo temporal	66 horas	14
46	☒ Redacción de memoria	150 horas	255

Figura 9.6: Diagrama de Gantt - Seguimiento - Comparación: General

Capítulo 10

Resultados finales

En este capítulo se van a comentar los resultados comparativos de todas las implementaciones, salvo lo ya comentado en la sección 7.4.

Los resultados de las mediciones para diferente número de qubits de todas las implementaciones se han puesto juntos en tablas para así compararlos más fácilmente. En la tabla 10.1 se muestra los resultados de consumo de memoria hasta 25 qubits (mostrando “x” en caso de que ejecutar para un estado de ese tamaño no fuera posible) y en la tabla 10.2 se muestra una comparativa de los tiempos de ejecución hasta 15 qubits.

Puesto que el principal objetivo del proyecto es aumentar el número de qubits operables, los resultados se separan principalmente en los que operan hasta 12 qubits como máximo, que son aquellas implementaciones que utilizan el operador completo (*base_impl* y *mem_opt_1*), y los que operan hasta 24 qubits, que corresponden a las implementaciones que usan operadores parciales (*mem_opt_2*, *mem_opt_3*, *time_opt_1*, *time_opt_2* y *time_opt_3*). Aquellas implementaciones que no consiguen superar la barrera de los 12 qubits son considerados fallos pues no cumplen el objetivo principal del proyecto.

Para aquellas implementaciones capaces de ejecutar más de 24 el consumo de memoria se ha mantenido casi constante. En comparación con la implementación ofrecida por *qutip*, el consumo de memoria para los tamaños de estado operables por esta se ha reducido enormemente, no llegando a 45MB en ninguno de los casos

que operan hasta 24 qubits frente a los 1450 MB que precisa la implementación de *qutip_impl* para 12 qubits.

En lo referente a tiempos de ejecución, aunque las implementaciones hasta *time_opt_2* incurren en tiempos superiores a los obtenidos ejecutando el operador de *qutip_impl*, al final se consiguió llegar a tiempos muy inferiores. Dentro de los tamaños de estado comparables, *time_opt_2* y *time_opt_3* solo precisan más tiempo que *qutip_impl* para 9 y menos qubits (lo cual es normal especialmente para *time_opt_3* pues hasta 8 qubits el tiempo utilizado apenas aumenta al deberse principalmente a la gestión de los múltiples procesos de ejecución), y para estados de 10, 11 y 12 qubits, el tiempo requerido por *time_opt_3* es inferior al de *qutip_impl*, legando a ser menos de la mitad (para 12 qubits *qutip_impl* necesita 7.97 segundos y *time_opt_3* solo 3.37 segundos), y pudiéndose reducir más al aumentar el número de núcleos disponibles.

n	qutip_impl	base_impl	mem_opt_1	mem_opt_2	mem_opt_3	time_opt_1	time_opt_2	time_opt_3
1	42.34375	42.81250	42.36719	42.31250	42.91797	42.41016	43.57812	43.62109
2	42.34375	42.81250	42.36719	42.31250	42.91797	42.41016	43.57812	43.62109
3	42.34375	42.81250	42.36719	42.31250	42.91797	42.41016	43.57812	43.62109
4	42.34375	42.81250	42.36719	42.31250	42.91797	42.41016	43.57812	43.62109
5	42.34375	42.81250	42.36719	42.31250	42.91797	42.41016	43.57812	43.62109
6	42.67578	42.81250	42.36719	42.31250	42.91797	42.41016	43.57812	43.62109
7	43.70703	43.70312	42.36719	42.64844	42.91797	42.41016	43.57812	43.62109
8	47.84375	47.01172	42.36719	44.71094	42.91797	42.41016	43.57812	43.62109
9	64.57031	59.87109	42.36719	50.73047	42.91797	42.41016	43.57812	43.62109
10	130.57812	110.89844	42.36719	74.69922	42.91797	42.41016	43.57812	43.62109
11	394.74219	315.03906	42.36719	170.80078	42.91797	42.41016	43.57812	43.69141
12	1450.74219	1128.67188	42.93359	554.88672	43.19531	43.26562	43.57812	43.78906
13	x	x	43.44141	x	43.45312	43.56641	43.69531	44.43359
14	x	x	44.04688	x	44.20703	44.49609	44.15234	45.58203
15	x	x	45.37500	x	45.75391	45.95312	45.58203	47.60938
16	x	x	47.97656	x	48.33594	48.31250	48.39844	51.37500
17	x	x	53.75000	x	54.20312	55.32031	54.23438	55.86719
18	x	x	64.54688	x	64.99609	66.69141	64.86328	67.33984
19	x	x	86.23438	x	86.68750	88.83594	86.55078	90.83203
20	x	x	129.47656	x	129.92969	132.80859	129.79688	138.07031
21	x	x	215.96484	x	216.41797	220.75781	216.28125	232.56250
22	x	x	388.94141	x	389.39062	396.91406	389.26172	421.53516
23	x	x	735.15234	x	735.60547	748.75000	735.85156	800.13281
24	x	x	1427.06250	x	1427.51172	1452.88672	1427.64453	1421.3125
25	x	x	x	x	x	x	x	x

Tabla 10.1: Comparativa del uso de memoria (MB)

n	qutip_impl	base_impl	mem_opt_1	mem_opt_2	mem_opt_3	time_opt_1	time_opt_2	time_opt_3
1	0.00108	0.00090	0.02397	0.00044	0.01873	0.00345	0.05523	0.07445
2	0.00104	0.00095	0.02114	0.00048	0.01814	0.00065	0.05791	0.08768
3	0.00121	0.00097	0.02437	0.00058	0.01922	0.00094	0.04330	0.08578
4	0.00127	0.00115	0.03104	0.00067	0.01910	0.00147	0.04675	0.08457
5	0.00163	0.00196	0.04386	0.00145	0.02111	0.00292	0.04410	0.08111
6	0.00333	0.00464	0.07197	0.00441	0.02585	0.00635	0.04949	0.08659
7	0.01037	0.01541	0.13336	0.01491	0.04662	0.01650	0.05933	0.10730
8	0.03920	0.05860	0.28425	0.05886	0.09352	0.05842	0.08112	0.09962
9	0.14529	0.23258	0.70011	0.24231	0.29347	0.16518	0.16032	0.15352
10	0.52456	0.88197	1.82345	0.94307	1.06126	0.62181	0.47548	0.35369
11	2.01386	3.78109	5.78695	3.87133	4.09253	2.21205	1.46610	0.99499
12	7.97829	14.74178	19.07792	15.67649	15.96296	8.39911	5.49805	3.37213
13	x	x	68.59218	x	62.77112	33.14104	20.82074	12.96976
14	x	x	260.01283	x	249.24228	129.69665	83.20696	50.79075
15	x	x	1083.12267	x	1003.94503	586.90145	352.20139	226.98583

Tabla 10.2: Comparativa del consumo de tiempo (s)

Capítulo 11

Conclusiones

Contenidos

11.1. Valoración	135
11.2. Consecución de objetivos	136
11.3. Problemas surgidos	137
11.4. Lecciones aprendidas	137
11.5. Trabajo futuro	138

Con los datos mostrados en el capítulo de resultados (capítulo 10), se da por concluido el presente proyecto.

11.1. Valoración

Durante el desarrollo de las sucesivas iteraciones (implementaciones) se ha refinado la implementación inicial basada por completo en una simulación pura del operador de la transformada cuántica de Fourier que surge de aplicar la transformada discreta de Fourier a un estado cuántico. Esto se ha hecho de forma que sea utilizable en cualquier programa que use la librería de simulación cuántica para python de *qutip*, la cual es cada vez más usada y esta en continuo crecimiento. Aunque esta librería ya ofrezca una función para poder aplicar al transformada

cuántica de Fourier, tiene una fuerte limitación en el número de qubits operables, siendo como máximo solo 12 qubits y por lo tanto mejorable.

Pese a la complejidad espacial y temporal que presenta esta simulación, siendo en todo momento cuadrática respecto al tamaño de los vectores N y exponencial respecto al número de qubits n ($N = 2^n$), se ha conseguido poder aplicarla a estados de hasta 24 qubits aprovechando los procesadores de múltiples núcleos que hoy en día tienen hasta los teléfonos móviles. Los tiempos de ejecución se han refinado hasta conseguir que sean inferiores a los de generar y aplicar el operador ofrecido por *qutip*, reduciéndose más a cuantos más núcleos disponibles se tengan y siendo ya inferior a *qutip* aun con un solo núcleo (*time_opt_2* también tiene un tiempo de ejecución inferior a *qutip_impl* y la única diferencia con *time_opt_3* con 1 solo núcleo disponible es que los cálculos se hacen en otro proceso).

11.2. Consecución de objetivos

En el apartado de introducción se exponen los objetivos generales del proyecto, los cuales se consideran todos cumplidos, sobretodo el de aumentar el número de qubits operables y reducir el consumo de memoria.

- ✓**Objetivo-1** Comprender las bases de la computación cuántica.
 - ✓**Objetivo-2** Comprender el algoritmo concreto de la transformada cuántica de Fourier.
 - ✓**Objetivo-3** Construir una simulación básica de dicho algoritmo.
 - ✓**Objetivo-4** Optimizar la simulación para poder usarlo con mas cantidad de qubits y que requiera menos memoria.
 - ✓**Objetivo-5** Optimizar la simulación para reducir su tiempo de ejecución.
-

11.3. Problemas surgidos

Durante la realización del proyecto han surgido diversos problemas, los más problemáticos de los cuales han sido la diferencia del resultado entre la implementación de *qutip* y las propias por problemas de redondeo, los altos tiempos de ejecución para realizar las mediciones hasta 10 veces y sacar datos más robustos que con una ejecución y comprender y explicar bien los conceptos básicos de computación cuántica.

Las diferencias de los resultados con la implementación de *qutip* se debían a diferencias en las operaciones o en su orden que producían que los resultados fueran diferentes por cantidades ínfimas (diferencias como por ejemplo 7 y 7,000000000000001, llegando a ser diferencias del orden de 10^{-17}). Esto se resolvió redondeando ambos resultados a 5 decimales (configurable mediante una constante) antes de compararlos, con lo que el problema quedo resuelto.

Los altos tiempos de ejecución tenían tres posibles soluciones: no ejecutar el algoritmo completo para estados de más de 12 o 13 qubits, reducir el número de mediciones o dejarlo ejecutar el tiempo que necesitase. Al final se decidió realizar la última opción, lo que conlleva mas de una vez tener que posponer hasta el día siguiente el análisis de los resultados para poder dejar toda la noche el ordenador ejecutar a fin de que tuviera las menos interferencias de otros procesos posibles durante toda la ejecución.

En lo referente a entender los conceptos y explicarlos, se invirtió más tiempo a buscar una forma adecuada de redactarlos, lo que supuso, con diferencia, la mayor parte del desvío en esfuerzo del proyecto.

11.4. Lecciones aprendidas

A la hora de estudiar la transformada cuántica de Fourier, se asumió que las fuentes de información explicarían las optimizaciones posibles de haberlas, no obstante más tarde revisando documentación se vio que existe una implementación

de la transformada discreta de Fourier, llamada transformada rápida de Fourier (FFT), que mejora su complejidad de $O(N^2)$ a $O(N \log N)$ mediante un cambio en el orden de los elementos del vector de entrada y la separación del operador en dos operadores. Por ello, aunque probablemente no se hubiese usado ese algoritmo debido a la modificación del estado de entrada (alejando un poco más la simulación del comportamiento real), si que se podría haber sopesado y tenido en cuenta a la hora de decidir que optimizaciones probar y en otros casos esa falta de información inicial puede acarrear una gran cantidad de esfuerzo hecho en vano.

11.5. Trabajo futuro

Como trabajo futuro queda la posibilidad de paralelizar aún más el proceso de *time_opt_3*, distribuyendo el trabajo entre varios equipos a través de una red, los cuales a su vez aprovechen los núcleos que tengan con el fin de acelerar el proceso y hacer factible temporalmente operar con 20 qubits.

Otra mejora posible es la de tratar de realizar la aplicación del operador parcial explícitamente en lugar de usar la librería de *scipy* con el fin de tratar de aumentar aun más el número de qubits operables pues uno de los errores obtenidos tiene que ver con la imposibilidad de *scipy* de operar dos matrices de 2^{25} elementos cada una.

También queda como trabajo futuro ponerse en contacto con los desarrolladores de *qutip* con el fin de ofrecer el que incluyan la función de *time_opt_3* en su librería y que así pueda ser usada y tenga un cometido no solo didáctico.

Apéndice A

Manual de usuario

La forma de utilizar el software creado es sencilla:

1. Importar el fichero correspondiente a la implementación que se quiera ejecutar
2. Generar el estado cuántico de la librería de *qutip* al que se le quiera aplicar la transformada cuántica de Fourier
3. Utilizar el método `execute(self, v)` pasándole por parámetro el estado a transformar y recuperando como valor de retorno el estado transformado.

Por ejemplo con un estado de entrada aleatorio de 5 qubits con *time_opt_3*

```
from QFT_time_opt_3 import *  
  
input = qutip.rand_ket (2**5)  
output = QFT_time_opt_3().execute(v)
```


Bibliografía

- [1] Carlos Ruiz Jiménez (N.D.).
fisicafundamental.net - computación cuántica.
<http://www.fisicafundamental.net/misterios/computacion.html>.
- [2] Dave Bacon (N.D.).
Cse 599d - quantum computing the quantum fourier transform and jordan's algorithm.
<https://courses.cs.washington.edu/courses/cse599d/06wi/lecturenotes9.pdf>.
- [3] Fourier, J.B.J. (1822).
Théorie analytique de la chaleur.
- [4] Free Software Foundation (N.D.).
Proyecto gnu.
<https://www.gnu.org/>.
- [5] Johansson, J. R., Nation, P. D., and Nori, F. (2013).
QuTiP 2: A Python framework for the dynamics of open quantum systems.
Computer Physics Communications, 184(4):1234–1240.
- [6] Jones, E., Oliphant, T., Peterson, P., et al. (2001–).
SciPy: Open source scientific tools for Python.
<http://www.scipy.org/>.
- [7] Jose Castro (2004).
Introducción a la computación cuántica.
<http://www.ic-itcr.ac.cr/~jcastro/libro/libro/node40.html>.

- [8] Jun, Y., Gavrilov, M. c. v., and Bechhoefer, J. (2014).
High-precision test of landauer’s principle in a feedback trap.
Phys. Rev. Lett., 113:190601.
 - [9] Oliphant, T. (2005–).
NumPy: Open source scientific tools for Python.
<http://www.numpy.org/>.
 - [10] Python Software Foundation (1991–).
Python: Lenguaje de programación.
<https://www.python.org/>.
 - [11] Richard Feynman and Peter W. Shor (1982).
Simulating physics with computers.
SIAM Journal on Computing, 26:1484–1509.
 - [12] Rolf Landauer (1961).
Irreversibility and heat generation in the computing process.
 - [13] Universidad de California (N.D.).
Bsd-new license.
 - [14] Vicente Moret Bonillo (2013).
La evolución del bit y un paseo por la energía.
 - [15] VMware, Inc (N.D.).
Vmware.
<http://www.vmware.com/es.html>.
 - [16] Wikipedia (N.D.).
Wikipedia - transformada cuántica de fourier.
 - [17] Yanofsky, N. S. and Mannucci, M. A. (2008).
Quantum Computing for Computer Scientists.
Cambridge University Press, New York, NY, USA, 1 edition.
-