Evandro Chagas Ribeiro da Rosa

QSystem: simulador quântico para Python

Florianópolis - SC

2019/1

Evandro Chagas Ribeiro da Rosa

QSystem: simulador quântico para Python

Trabalho de conclusão de curso apresentado
como parte dos requisitos para obtenção do
grau de Bacharel em Ciência da Computação

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Orientador: Bruno Gouvêa Taketani
Responsável: Jerusa Marchi

Florianópolis - SC
2019/1

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Rosa, Evandro Chagas Ribeiro da
QSystem: simulador quântico para Python / Evandro Chagas Ribeiro da Rosa ; orientador, Bruno Gouvêa Taketani, 2019.
209 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2019.

Inclui referências.

1. Ciências da Computação. 2. computação quântica. 3. simulação. 4. circuito quântico. 5. Python. I. Taketani, Bruno Gouvêa . II. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. III. Título.

Evandro Chagas Ribeiro da Rosa

QSystem: simulador quântico para Python

Este Trabalho Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo Curso de Ciência da Computação.

Florianópolis - SC, 26 de junho de 2019.

_____

Prof. Dr. Renato Cislaghi

Coordenador de Projetos

Banca Examinadora:

_____

Prof. Dr. Bruno Gouvêa Taketani

Orientador

Universidade Federal de Santa Catarina

_____

Profª. Drª. Jerusa Marchi

Responsável

Universidade Federal de Santa Catarina

_____

Prof. Dr. Eduardo Inácio Duzzioni

Universidade Federal de Santa Catarina

Resumo

Esse trabalho documenta a implementação de um simulador de computação quântica baseado no modelo de circuitos quânticos onde é possível executar uma computação quântica tanto em vetor de estado quanto em matriz densidade, possibilitando, assim, a simulação de erros quânticos. O simulador foi desenvolvido majoritariamente em C++ e entregue como um módulo de Python, denominado QSystem, desta forma, obtém uma boa performance ao mesmo tempo que é dinâmico para o uso. Toda a base teórica referente a computação quântica necessária para a implementação do simulador é apresentada nos primeiros capítulos.

Palavras-chave: computação quântica, simulação, circuito quântico, Python, C++

Sumário

Introdução

Como apresentado por Hennessy e Patterson(1), dos anos 90 até o início dos anos 2000, diversos fatores eram favoráveis para que a taxa de crescimento do poder computacional fosse de 52% ao ano (dobrando, aproximadamente, a cada dois anos). Porém, em 2003, devido a desaceleração da lei de Moore(2), ao fim da escalabilidade de Dennard(3) e, ao limite dos processadores multinúcleo, causado pela lei de Amdahl(4), a taxa de crescimento do poder computacional caiu para 23% ao ano (dobrando, aproximadamente, a cada quatro anos). Contudo, essa taxa continuou caindo, sendo que a prevista para 2018 (ano seguinte a publicação do livro *Computer Architecture: A Quantitative Approach*(1)) foi de 3,5% (dobrando, aproximadamente, a cada vinte e oito anos e meio). Todavia, com o advento dos primeiros computadores quânticos comerciais, a indústria e a academia veem a computação quântica como a porta de entrada para uma nova era de crescimento exponencial do poder computacional.

A possibilidade de utilizar um computador quântico foi inicialmente apresentada por Feynman(5) em 1982, quando ele propôs que um computador quântico poderia simular eficientemente um sistema quântico, uma tarefa difícil para os computadores clássicos (até mesmo para os atuais). Esse, foi um dos primeiros indícios de que um computador quântico pode ter ganho sobre um computador clássico para resolução de alguns problemas. Contudo, a primeira demonstração disto foi feita por Shor(6) em 1994, quando ele apresentou dois algoritmos quânticos para resolver dois problemas $\mathcal{NP}$ (o logaritmo discreto e a fatoração) que funcionam em tempo polinomial em um computador quântico. Atualmente, já foram propostos vários algoritmos quânticos mais eficientes que seus respectivos algoritmos clássicos, porém, ainda há muito a ser desenvolvido nessa área.

O simulador QSystem foi desenvolvido para auxiliar no estudo de algoritmos, protocolos e códigos quânticos. Por ser baseado no modelo de computação circuital, um circuito quântico é facilmente transcrito para o simulador. Essa característica faz com que o simulador abstraia para o usuário grande parte da álgebra linear envolvida na computação quântica. Construído para operar tanto em vetor de estado quanto em matriz densidade, o simulador possibilita a simulação de erros quânticos, fazendo que o mesmo possa ser usado no estudo de códigos de identificação e correção de erros quânticos.

Para obter um bom desempenho, o simulador, foi desenvolvido em C++ utilizando a biblioteca de álgebra linear Armadillo(7) e a ferramenta SWIG(8), utilizada para gerar a interface entre o código escrito em C++17 e o interpretador do Python 3. Assim, o simulador QSystem é distribuído como um módulo de Python.

Este trabalho é dividido em duas partes, onde inicialmente, na Parte I, é apre-

sentado uma revisão teórica sobre os temas da computação quântica pertinentes para o compreendimento das funcionalidades e da implementação do simulador quântico, posteriormente, na Parte II, as funcionalidades do simulador são apresentadas juntamente a exemplos de uso e uma documentação sobre suas implementações.

A Parte I é composta por cinco capítulos, onde são apresentados os seguintes temas: representação e características de um bit quântico ou qubit (Capítulo 1); os quatro postulados da mecânica quântica segundo o livro *Quantum computation and quantum information*(9) (Capítulo 2); o modelo de circuitos quânticos que inspira o simulador (Capítulo 3); uma representação alternativa do estado quântico, denominada matriz densidade, usada para representar erros quânticos (Capítulo 4); e, como representar um erro quântico e os erros quânticos clássicos (Capítulo 5).

A Parte II é composta por oito capítulos, onde são abordados os seguintes aspectos do simulador: uma visão geral sobre o simulador QSystem (Capítulo 6); como é feita a evolução do estado quântico e criação de portas lógicas quânticas (Capítulo 7); como os qubits são medidos (Capítulo 8); a implementação dos canais de erros quânticos no simulador (Capítulo 9); as operações de adicionar e remover qubits ancilares (Capítulo 10); outras funcionalidades do simulador, como mudança de representação do estado quântico, manipulação da matriz/vetor que representa o estado e, salvar e carregar estados e portas lógicas quânticas (Capítulo 11); como é criada a interface entre o código C++ e o interpretador do Python, além de como é montado o pacote Python para instalação do módulo QSystem (Capítulo 12); e, por fim, são apresentados dois exemplos de implementação utilizando o simulador (Capítulo 13).

Ao final do trabalho é apresentado a conclusão e os possíveis trabalhos futuros, além de alguns apêndices que complementam o texto principal.

Parte I

Revisão teórica

1 Bit quântico

Na computação quântica temos como unidade básica de computação o bit quântico ou qubit. Assim, como o bit clássico, o qubit pode estar nos estados 0 ou 1, mas, diferente do caso clássico, também pode estar nos dois estados ao mesmo tempo.

Este capítulo tem o intuito de dar uma primeira resposta para a pergunta: o que é um qubit? Para isso, primeiramente, é apresentado como podemos representar um qubit (Secção 1.1), passando, então, para como obter informação de um qubit (Secção 1.2) e, por fim, é apresentado uma abstração geométrica de um qubit (Secção 1.3).

1.1 Espaço de Hilbert

Podemos representar um qubit no estado 0 como $|0\rangle$, no estado 1 como $|1\rangle$ ou no que chamamos de uma superposição entre os estados $|0\rangle$ e $|1\rangle$, como sendo $\alpha|0\rangle + \beta|1\rangle$, com a restrição de $|\alpha|^2 + |\beta|^2 = 1$.

Um qubit pode ser visto como um vetor pertencente a um espaço de vetorial, sendo assim, $\{|0\rangle, |1\rangle\}$ uma base para esse espaço, com isso podemos definir um qubit $|\psi\rangle$ como sendo a uma combinação linear de $|0\rangle$ e $|1\rangle$. Matematicamente um qubit pode ser representado por um vetor unitário pertencente a um espaço de Hilbert de duas dimensões sobre os números complexos ($\mathbb{C}^2$).

Na computação quântica usamos a notação de Dirac, ou notação *braket*, para representar um vetor, que no nosso caso é um qubit, assim, denotamos um vetor coluna com um $|ket\rangle$ e seu transposto conjugado, um vetor linha, com um $\langle bra|$. Ter uma base de álgebra linear é necessário para trabalhar com computação quântica, por isso o apêndice A traz uma rápida revisão de álgebra linear usando a notação de Dirac.

Um espaço de Hilbert é simplesmente um espaço vetorial com produto interno definido, onde o produto interno é uma operação entre dois vetores que retorna um escalar. Na notação de Dirac denotamos a operação de produto interno com um $\langle bra|ket\rangle$, assim podemos intuitivamente definir o produto interno entre dois vetores $|\psi\rangle = (\alpha_0, \ldots, \alpha_n)$ e $|\varphi\rangle = (\beta_0, \ldots, \beta_n)$ como sendo

$$\langle\varphi|\psi\rangle = \begin{bmatrix}\beta_0^* & \ldots & \beta_n^*\end{bmatrix} \begin{bmatrix}\alpha_0 \\ \vdots \\ \alpha_n\end{bmatrix} = \beta_0^*\alpha_0 + \cdots + \beta_n^* + \alpha_n, \tag{1.1}$$

como estamos trabalhando com espaços vetoriais complexos $\alpha_0, \ldots, \alpha_n$ e $\beta_0, \ldots, \beta_n$ são números complexos e o símbolo * denota o complexo conjugado do número, como por exemplo, o complexo conjugado de $z = 3 + 2i$ é $z^* = 3 - 2i$.

1.1.1 Base computacional

Podemos representar um qubit pela combinação linear dos vetores base de $\mathbb{C}^2$, porém, como $\mathbb{C}^2$ possui infinitas bases, usamos por padrão a base computacional composta pelos dois vetores

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ e } \qquad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{1.2}$$

Um qubit $|\psi\rangle$ pode ser representado como $\alpha\,|0\rangle + \beta\,|1\rangle$, sendo α e β números complexos e a norma de $|\psi\rangle$ igual a 1. Sendo um qubit composto por $\alpha \neq 0$ e $\beta \neq 0$, não podemos representa-lo apenas com $|0\rangle$ ou $|1\rangle$. Isso significa que o qubit não está mais apenas no estado $|0\rangle$ ou $|1\rangle$, mas em ambos ao mesmo tempo. Assim, dizemos que o qubit está em uma superposição dos estados $|0\rangle$ e $|1\rangle$.

1.2 Medida

Quando queremos obter informação de um qubit fazemos uma operação de medida que nos retorna 0 ou 1 dependendo do estado do qubit, porém, quando o qubit está no estado $\alpha\,|0\rangle + \beta\,|1\rangle$, em uma superposição, a medida pode retornar 0 com uma probabilidade de $|\alpha|^2$ e 1 com uma probabilidade de $|\beta|^2$. Logo, após a medida, o qubit colapsa no estado referente ao valor retornado pela medida.

Como por exemplo, o qubit $\cos\frac{\pi}{6}\,|0\rangle + \sin\frac{\pi}{6}\,|1\rangle$ tem $\cos^2\frac{\pi}{6} = 0.75$ de chance de retornar 0, e colapsar no estado $|0\rangle$, e $\sin^2\frac{\pi}{6} = 0.25$ de chace de retornar 1, e colapsar no estado $|1\rangle$, quando medido.

A soma da probabilidade de medir 0 com a da probabilidade de medir 1 tem que ser igual a 1, por isso, o qubit é um vetor unitário, ou seja com norma igual a 1.

1.3 Esfera de Bloch

Uma maneira conveniente de visualizar um qubit é como sendo um ponto na chamada esfera de Bloch. Com essa abstração podemos descrever qualquer operação de um qubit como sendo rotações na esfera. O qubit $|\psi\rangle$ da Figura 1 é igual a $\cos\frac{\theta}{2}\,|0\rangle + e^{i\phi}\sin\frac{\theta}{2}\,|1\rangle$. As informações contidas na Figura 1 ficarão mais claras no decorrer dos próximos capítulos.

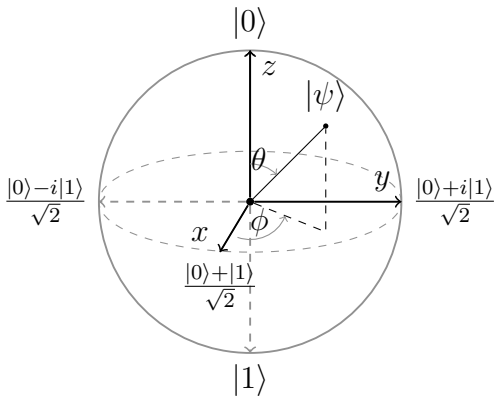


Figura 1 – Esfera de Bloch.

2 Postulados da mecânica quântica

A computação quântica toma proveito do funcionamento da mecânica quântica para fazer computação, portanto, é fundamental conhecer alguns dos seus conceitos.

O livro *Quantum computation and quantum information*(9, Secção 2.2) enumera quatro postulados, que dizem respeito ao espaço do sistema[1] (Secção 2.1), evolução do sistema (Secção 2.2), medida (Secção 2.3) e sistemas composto (Secção 2.4). Os postulados tem o intuito representar o mundo (quântico) físico através de um formalismo matemático, sendo eles aqui apresentados tendo a computação quântica como foco.

2.1 Postulado 1: Espaço do sistema

O primeiro postulado diz respeito a representação de um sistema quântico.

> **Postulado 1:** Associado a cada sistema quântico fechado[2] há um espaço de Hilbert, e o estado do sistema é totalmente representado por um vetor unitário pertencente a esse espaço.

Um computador quântico é um sistema quântico fechado, portanto, podemos definir seu estado como sendo um vetor pertencente a um espaço de Hilbert.

2.2 Postulado 2: Evolução do sistema

O segundo postulado diz respeito a evolução do sistema quântico no tempo, ou seja, como a computação é realizada.

> **Postulado 2:** A evolução de um sistema quântico fechado é descrita pela aplicação de um *operador unitário*, ou seja, a transição de um estado $|\psi\rangle^0$ no tempo t_0 para o estado $|\psi\rangle^1$ no tempo t_1 pode ser totalmente descrita por um operador unitário U, sendo $U|\psi\rangle^0 = |\psi\rangle^1$.

Na computação quântica podemos discretizar o tempo e considerar que cada instante de tempo representa um passo de computação e que as operações executadas a cada passo podem ser representadas por um operador unitário.

---

[1] A palavra sistema se refere a um sistema quântico, que no contexto desse trabalho e por simplificação pode ser considerado como sendo um computador quântico

[2] Consideramos um computador quântico como sendo um sistema fechado.

2.2.1 Operador linear

Uma transformação linear é uma função que leva um vetor de um espaço vetorial em outro, caso o domínio e contradomínio coincidam, essa função é chamada de operador linear e possui as seguintes propriedades:

1. Seja A um operador linear:

$$A(|\varphi\rangle + |\psi\rangle) = A|\varphi\rangle + A|\psi\rangle \tag{2.1}$$

2. Seja A um operador linear e α um escalar:

$$\alpha A|\psi\rangle = A(\alpha|\psi\rangle) \tag{2.2}$$

2.2.1.1 Representação matricial

Todo operador linear que atua em um espaço vetorial de dimensão n pode ser representado por uma matriz $n \times n$. Como por exemplo, o operador linear X que mapeia $\{|0\rangle \to |1\rangle, |1\rangle \to |0\rangle\}$ pode ser representado pela seguinte matriz:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}. \tag{2.3}$$

Como pode ser visto nos exemplos abaixo:

$$X|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle; \tag{2.4}$$

$$X|1\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle. \tag{2.5}$$

2.2.1.2 Operador unitário

Quando um operador linear atua sobre um espaço de Hilbert e sua inversa é igual ao seu Hermitiano[3] ($U^{-1} = U^\dagger$) o operador é dito ser um operador unitário. De outra forma, um operador é unitário se o produto do operador com seu Hermitiano resulta na identidade ($UU^\dagger = I = U^\dagger U$).

O operador X da equação 2.3 é um exemplo de operador unitário, pois

$$XX^\dagger = X^\dagger X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I. \tag{2.6}$$

[3] Hermitiano: transposto conjugado.

2.2.1.3 Composição de função

Considere o caso de uma computação onde a transição do estado $|\psi\rangle^{n-1}$ para o estado $|\psi\rangle^n$ é dada pelo operador unitário U^n.

$$U^n |\psi\rangle^{n-1} = |\psi\rangle^n . \tag{2.7}$$

Assim, podemos relacionar o estado $|\psi\rangle^0$, inicio da computação, como o estado $|\psi\rangle^F$, final da computação, pela seguinte equação:

$$U^F \dots U^2 U^1 |\psi\rangle^0 = |\psi\rangle^F . \tag{2.8}$$

2.2.2 Matrizes de Pauli

Um grupo importante de operadores unitários são as matrizes de Pauli, conforme veremos a seguir.

2.2.2.1 Matriz identidade

A matriz identidade 2×2 pode ser considerada uma matriz de Pauli, a aplicação desse operador não altera o qubit, sendo $I |\psi\rangle = |\psi\rangle$.

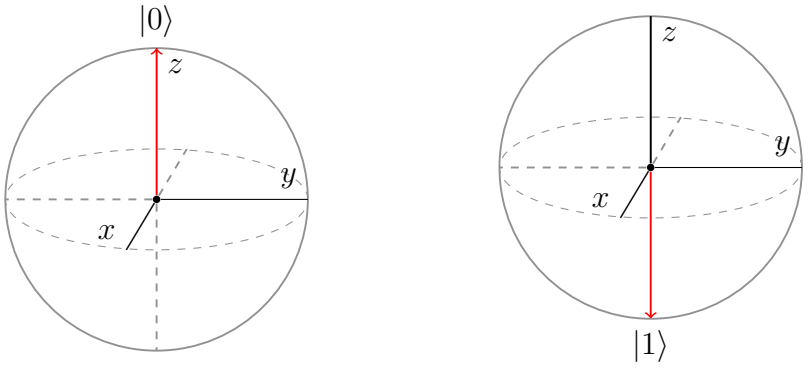
$$I = \sigma_0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = |0\rangle\langle 0| + |1\rangle\langle 1| \tag{2.9}$$

2.2.2.2 Sigma X

A matriz sigma X efetua um *bit flip*, levando o qubit do estado $|0\rangle \to |1\rangle$ e $|1\rangle \to |0\rangle$.

$$X = \sigma_1 = \sigma_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = |0\rangle\langle 1| + |1\rangle\langle 0| \tag{2.10}$$

Outra maneira de visualizar a aplicação desta matriz seria como uma rotação de π em torno do eixo x da esfera de Bloch.



(a) Antes da atuação de X. (b) Após a atuação de X.

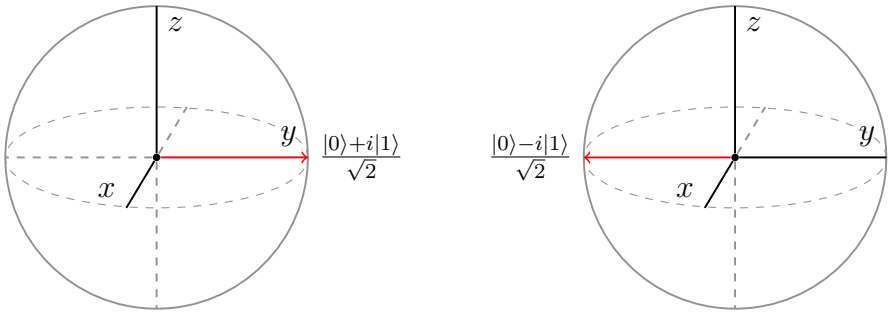
Figura 2 – Efeito da aplicação de X.

2.2.2.3 Sigma Z

A matriz sigma Z adiciona uma fase[4] negativa ao estado $|1\rangle$ e não altera o estado $|0\rangle$, portanto $Z|0\rangle = |0\rangle$ e $Z|1\rangle = -|1\rangle$.

$$Z = \sigma_3 = \sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = |0\rangle\langle 0| - |1\rangle\langle 1| \tag{2.11}$$

Outra maneira de visualizar a aplicação desta matriz seria como uma rotação de π em torno do eixo z da esfera de Bloch.



(a) Antes da atuação de Z. (b) Após a atuação de Z.

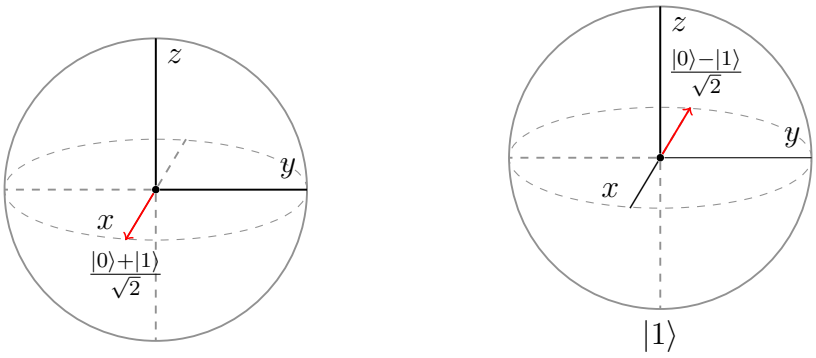
Figura 3 – Efeito da aplicação de Z.

2.2.2.4 Sigma Y

A matriz sigma Y efetua um *bit flip* adicionando uma fase negativa ao estado $|1\rangle$ e uma fase complexa em ambos $|0\rangle$ e $|1\rangle$, sendo $Y|0\rangle = i|1\rangle$ e $Y|1\rangle = -i|0\rangle$.

$$Y = \sigma_2 = \sigma_y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} = i|1\rangle\langle 0| - i|0\rangle\langle 1| \tag{2.12}$$

Outra maneira de visualizar a aplicação desta matriz seria como uma rotação de π em torno do eixo y da esfera de Bloch.



(a) Antes da atuação de Y. (b) Após a atuação de Y.

Figura 4 – Efeito da aplicação de Y.

[4] Fase é um escalar complexo do tipo $e^{i\theta}$ que multiplica um qubit $|\psi\rangle$.

2.3 Postulado 3: Medida

Como visto anteriormente, um qubit pode estar em uma superposição, como $\alpha|0\rangle + \beta|1\rangle$, e, para obter informação do sistema, é necessário medi-lo, colapisando-o no estado $|0\rangle$ ou $|1\rangle$.

Para realizar uma medida precisamos definir uma base onde normalmente é usada a base computacional, porém, em alguns casos, pode ser interessante utilizar outra base para realizar a medida. O terceiro postulado descreve o comportamento de uma medida em uma base qualquer.

Postulado 3: Uma medida quântica é descrita por um conjunto de *operadores de medida* $\{M_m\}$, onde o índice m indica o possível resultado da medida. Sendo $|\psi\rangle$ o estado logo antes da medida, a probabilidade de medir m é

$$p(m) = \langle\psi| M_m^\dagger M_m |\psi\rangle . \tag{2.13}$$

E o estado logo após a medida igual a

$$\frac{M_m|\psi\rangle}{\sqrt{\langle\psi| M_m^\dagger M_m |\psi\rangle}}. \tag{2.14}$$

Os operadores de medida precisam satisfazer a seguinte *equação de completude*

$$\sum_m M_m^\dagger M_m = I, \tag{2.15}$$

ou de maneira equivalente

$$\sum_m p(m) = \sum_m \langle\psi| M_m^\dagger M_m |\psi\rangle = 1. \tag{2.16}$$

Os operadores de medida da base computacional são $M_0 = |0\rangle\langle 0|$ e $M_1 = |1\rangle\langle 1|$ (A operação $|ket\rangle\langle bra|$ tem como resultado uma matriz), logo, a probabilidade de medir 0 no qubit $\alpha|0\rangle + \beta|1\rangle$ é

$$\begin{aligned} p(0) &= (\alpha^* \langle 0| + \beta^* \langle 1|) M_0^\dagger M_0 (\alpha|0\rangle + \beta|1\rangle) && (2.17)\\ &= (\alpha^* \langle 0| + \beta^* \langle 1|) |0\rangle\langle 0|0\rangle\langle 0| (\alpha|0\rangle + \beta|1\rangle) && (2.18)\\ &= (\alpha^* \langle 0|0\rangle + \beta^* \langle 1|0\rangle) \langle 0|0\rangle (\alpha \langle 0|0\rangle + \beta \langle 0|1\rangle) && (2.19)\\ &= |\alpha|^2. && (2.20) \end{aligned}$$

E o estado logo após medir 0 é

$$\frac{|0\rangle\langle 0| (\alpha|0\rangle + \beta|1\rangle)}{\sqrt{|\alpha|^2}} = \frac{|0\rangle (\alpha \langle 0|0\rangle + \beta \langle 0|1\rangle)}{\sqrt{|\alpha|^2}} = \frac{\alpha}{|\alpha|}|0\rangle . \tag{2.21}$$

Da mesma forma, a $p(1) = |\beta|^2$ e o estado após medir 1 é $\frac{\beta}{|\beta|}|1\rangle$.

Quando um qubit é medido, ele colapsa em algum estado da base na qual foi medido. Portanto, a base influencia no resultado e no estado logo após a medida. Como por exemplo, se medirmos o qubit $|0\rangle$ na base computacional sempre vamos colapsar sobre o estado $|0\rangle$, porém, se usarmos a base

$$\left\{|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix}, |-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\-1\end{bmatrix}\right\} \tag{2.22}$$

há 50% de chance de colapsar no estado $|+\rangle$ e 50% de chance de colapsar no estado $|-\rangle$ e o estado após a medida será $\frac{|0\rangle \pm |1\rangle}{\sqrt{2}}$.

2.4 Postulado 4: Sistemas composto

Tudo que vimos até agora diz respeito a um qubit, porém para fazer uma computação relevante precisamos juntar vários qubits. O quarto postulado diz respeito a composição de sistemas quânticos, ou seja, como funciona a dinâmica com vários qubits.

> **Postulado 4:** O estado de um sistema composto é dado pelo produto tensorial dos seus componentes, ou seja, um sistema composto por n sistemas quânticos, nos estados $|\psi_0\rangle$, $|\psi_1\rangle$, ..., $|\psi_{n-1}\rangle$, tem seu estado total representado por $|\psi_0\rangle \otimes |\psi_1\rangle \otimes \cdots \otimes |\psi_{n-1}\rangle$.

Antes de entrar em detalhes de como é feita a evolução e a medida de um sistema composto por vários qubits precisamos entender como funciona a operação de produto tensorial.

2.4.1 Produto tensorial

O produto tensorial é uma operação que leva dois vetores, um de dimensão n e outro de dimensão m, para um espaço de dimensão nm da seguinte maneira:

$$\begin{bmatrix}\alpha_1\\\alpha_2\\\vdots\\\alpha_n\end{bmatrix} \otimes \begin{bmatrix}\beta_1\\\beta_2\\\vdots\\\beta_m\end{bmatrix} = \begin{bmatrix}\alpha_1\beta_1\\\vdots\\\alpha_1\beta_m\\\vdots\\\alpha_n\beta_1\\\vdots\\\alpha_n\beta_m\end{bmatrix}. \tag{2.23}$$

Como por exemplo:

$$\begin{bmatrix}1\\2\end{bmatrix} \otimes \begin{bmatrix}3\\4\end{bmatrix} = \begin{bmatrix}1\times 3\\1\times 4\\2\times 3\\2\times 4\end{bmatrix}. \tag{2.24}$$

Esta operação também pode ser aplicada entre matrizes, sendo, o resultado do produto tensorial de uma matriz $n \times m$ com outra matriz $p \times q$, uma matriz $np \times mq$. O produto tensorial dentre vetores pode ser considerado um caso especial do produto entre matrizes, que é definido da seguinte forma:

$$A \otimes B = \begin{bmatrix} A_{11}B & A_{12}B & \dots & A_{1n}B \\ A_{21}B & A_{22}B & \dots & A_{2n}B \\ \vdots & \vdots & \ddots & \vdots \\ A_{m1}B & A_{m2}B & \dots & A_{mn}B \end{bmatrix}. \tag{2.25}$$

Como por exemplo, o produto tensorial entre X e Z:

$$X \otimes Z = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} 0 \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} & 1 \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \\ 1 \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} & 0 \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{bmatrix}. \tag{2.26}$$

Um sistema composto pelos qubits $|\psi\rangle$ e $|\varphi\rangle$ pode ser denotado das seguintes formas: explicitando o produto tensorial, $|\psi\rangle \otimes |\varphi\rangle$; apenas concatenando os *ket*, $|\psi\rangle\,|\varphi\rangle$; representado ambos dentro do mesmo *ket*, $|\psi\varphi\rangle$; também é possível adicionar uma etiqueta para identificar cada qubit, $|\psi_0\varphi_1\rangle = |\varphi_1\psi_0\rangle$.

A notação $|\psi\rangle^{\otimes n}$ representa o produto tensorial do qubit $|\psi\rangle$ com ele mesmo n vezes.

$$|\psi\rangle^{\otimes n} = \overbrace{|\psi\rangle \otimes \cdots \otimes |\psi\rangle}^{n \text{ vezes}} \tag{2.27}$$

2.4.2 Base do sistema composto

O espaço de Hilbert formado pela composição dos subespaços de um qubit pode ter como base o produto tensorial das bases destes subespaços. Como por exemplo, utilizando a base computacional, o espaço composto por dois qubits pode ter como base os estados:

$$|00\rangle = |0\rangle \otimes |0\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}; \qquad |01\rangle = |0\rangle \otimes |1\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}; \tag{2.28}$$

$$|10\rangle = |1\rangle \otimes |0\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}; \qquad |11\rangle = |1\rangle \otimes |1\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}. \tag{2.29}$$

Em alguns casos é interessante representar os vetores com números decimais, como por exemplo: $|00\rangle = |0\rangle$; $|01\rangle = |1\rangle$; $|10\rangle = |2\rangle$; $|11\rangle = |3\rangle$. Assim podemos extrapolar a base computacional de um sistema de n qubits como sendo: $|0\rangle, |1\rangle, |2\rangle, \dots, |2^n{-}1\rangle$.

2.4.3 Evolução do sistema composto

A evolução de um sistema composto de n qubits é da mesma forma vista na secção 2.2, através de um operador unitário de dimensão 2^n. Podemos construir um operador unitário a partir do produto tensorial dos operadores que atuam em cada subespaço, como por exemplo, a evolução do estado $|00\rangle$ para $|11\rangle$ pode ser descrita pelo operador $X \otimes X$. Em alguns contextos podemos denotar $X \otimes X$ como XX.

$$(X \otimes X) |00\rangle = |11\rangle \tag{2.30}$$

Para representar a evolução de apenas alguns qubits podemos denotar os operadores com um sobrescrito indicando qual qubit queremos evoluir, sendo os qubits indexados da esquerda para direita a partir do 0. Por exemplo, se queremos que X atue apenas no primeiro qubit denotamos como X_0, e se queremos que Y também atue no terceiro qubit denotamos como X_0Y_2. Assim, de maneira genérica se queremos que um operador qualquer atue no i-ésimo qubit podemos denotá-lo como

$$U_i = I_0 \otimes \cdots \otimes U_i \otimes \cdots \otimes I_{n-1}. \tag{2.31}$$

Por exemplo:

$$X_0 |00\rangle = |10\rangle ; \tag{2.32}$$

$$X_0Y_2 |000\rangle = i |101\rangle . \tag{2.33}$$

2.4.4 Medida no sistema composto

Os operadores de medida podem ser construídos utilizando a mesma lógica dos operadores unitários, por exemplo, se queremos medir, na base computacional, o primeiro qubit no estado

$$\alpha_0 |00\rangle + \alpha_1 |01\rangle + \alpha_2 |10\rangle + \alpha_3 |11\rangle , \tag{2.34}$$

podemos usar os operadores

$$\{|0\rangle\langle 0| \otimes I, |1\rangle\langle 1| \otimes I\}. \tag{2.35}$$

Sendo a probabilidade de medir 0 igual a $|\alpha_0|^2 + |\alpha_1|^2$ e de medir 1 igual a $|\alpha_2|^2 + |\alpha_3|^2$, e o estado após a medida: no caso de medir 0

$$\frac{\alpha_0 |00\rangle + \alpha_1 |01\rangle}{\sqrt{|\alpha_0|^2 + |\alpha_1|^2}}, \tag{2.36}$$

e no caso de medir 1

$$\frac{\alpha_2 |10\rangle + \alpha_3 |11\rangle}{\sqrt{|\alpha_2|^2 + |\alpha_3|^2}}. \tag{2.37}$$

Da mesma forma, para um qubit, o estado de um sistema composto tem norma 1. No caso do exemplo acima $|\alpha_0|^2 + |\alpha_1|^2 + |\alpha_2|^2 + |\alpha_3|^2 = 1$.

2.4.5 Emaranhamento quântico

Nem todo estado composto de dois ou mais qubits pode ser decomposto no produto tensorial de cada qubit, como por exemplo, não conseguimos definir α_0, β_0, α_1 e β_1 que satisfaça a equação

$$\frac{|00\rangle + |11\rangle}{\sqrt{2}} = (\alpha_0 |0\rangle + \beta_0 |1\rangle) \otimes (\alpha_1 |0\rangle + \beta_1 |1\rangle). \tag{2.38}$$

Nessas condições dizemos que os qubits estão emaranhados. A principal implicação disso é que a medida de um qubit influencia no resultado da medida de outro e essa característica se mantém independente da distância e do intervalo de tempo entre a medida de cada qubit.

Para ilustrar esse efeito, imagine que Alice preparou dois qubits no estado $\frac{|00\rangle+|11\rangle}{\sqrt{2}}$ e enviou um deles para Bob, que, logo em seguida, viajou para Marte. Ambos combinaram de medir seus qubits ao mesmo tempo, Alice na Terra e Bob em Marte. Desconsiderando qualquer erro durante o processo, os únicos possíveis resultados são onde Alice mediu 0 e Bob mediu 0 ou Alice mediu 1 e Bob mediu 1, pois os dois qubits só podem colapsar nos estados $|00\rangle$ ou $|11\rangle$.

3 Circuitos quântico

Circuitos quântico é um modelo computacional que abstrai o formalismo matemático da mecânica quântica para facilitar a construção e visualização de algoritmos quânticos através de um diagrama de circuitos.

Neste capítulo são apresentados os conceitos básicos para compreensão e construção de um circuito quânticos. Primeiramente, é apresentado o conceito de portas lógicas quânticas (Secção 3.1) e porta de medida (Secção 3.2) e, por fim, são mostrados alguns exemplos de circuitos quânticos (Secção 3.3).

3.1 Porta lógica quântica

A evolução do sistema quântico é representada por portas lógicas quânticas, onde cada porta lógica possui um operador unitário associado. Uma porta lógica quântica é representada por um retângulo com um rótulo. As portas lógicas são concatenadas em uma linha onde, da esquerda para direita, cada uma é aplicada. Por exemplo, a evolução $CBA\,|\psi_{inicial}\rangle = |\psi_{final}\rangle$, sendo os estado $|\psi_{inicial}\rangle$ e $|\psi_{final}\rangle$ composto de n qubits, pode ser representada pelo circuito quânticos da Figura 5.

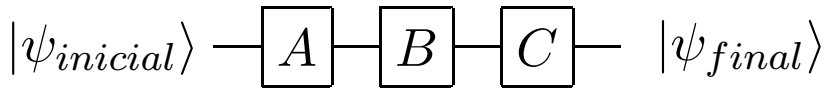


Figura 5 – Representação em circuito de $CBA\,|\psi_{inicial}\rangle = |\psi_{final}\rangle$.

3.1.1 Portas quânticas de um qubit

Em seguida são listadas as principais portas lógicas quânticas de um qubit.

3.1.1.1 Porta X, Z e Y

Dentre as principais portas lógicas quânticas estão as matrizes de Pauli, vistas na Subsecção 2.2.2, que são representadas na forma de circuito como nas Figuras 6, 7 e 8.

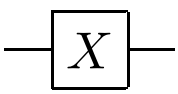


Figura 6 – Porta X, ou NOT, associada a matriz Sigma X vista na Subsecção 2.2.2.2.

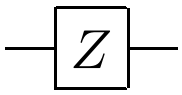


Figura 7 – Porta Z associada a matriz Sigma Z vista na Subsecção 2.2.2.3.

.

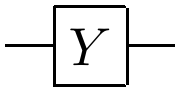


Figura 8 – Porta Y associada a matriz Sigma Y vista na Subsecção 2.2.2.4.

3.1.1.2 Porta de Hadamard

Outra importante porta lógica quântica é a porta de Hadamard, associada ao operador unitário representado pela matriz

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{3.1}$$

A Porta de Hadamard leva um qubit do estado $|0\rangle \to \frac{|0\rangle+|1\rangle}{\sqrt{2}} = |+\rangle$ e $|1\rangle \to \frac{|0\rangle-|1\rangle}{\sqrt{2}} = |-\rangle$, gerando uma superposição com 50% de chance de medir 0 e 50% de chance de medir 1. Pode ser considerada uma porta de mudança de base, entre a base computacional e a base $\{|+\rangle, |-\rangle\}$, onde $H|0\rangle = |+\rangle$, $H|1\rangle = |-\rangle$, $H|+\rangle = |0\rangle$ e $H|-\rangle = |1\rangle$.

Figura 9 – Porta de Hadamard.

3.1.1.3 Porta de Fase

A Porta de Fase adiciona uma fase complexa ao estado $|1\rangle$. É associada ao operador unitário representado pela matriz

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}, \tag{3.2}$$

sendo $S|0\rangle = |0\rangle$ e $S|1\rangle = i|1\rangle$.

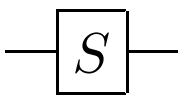


Figura 10 – Porta de fase.

3.1.1.4 Porta T

A Porta T adiciona uma fase de $\frac{\pi}{4}$ ao estado $|1\rangle$. É associada ao operador unitário representado pela matriz

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{bmatrix}. \tag{3.3}$$

O número $e^{i\frac{\pi}{4}}$ pode ser expandido na forma de Euler, onde $e^{ix} = \cos x + i \sin x$, sendo assim $e^{i\frac{\pi}{4}} = \frac{1+i}{\sqrt{2}}$.

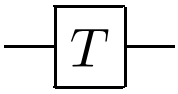


Figura 11 – Porta T.

3.1.2 Porta quântica controlada

Uma porta lógica quântica controlada é uma porta de no mínimo dois qubits, composta por qubits de controle e qubits alvo, onde a porta só é aplicada nos qubits alvo se os qubits de controle estiverem em um estado específico, por exemplo $|1\rangle$.

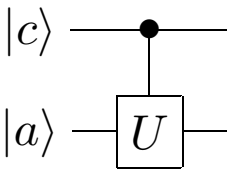
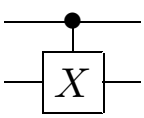
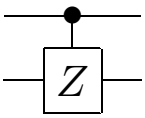


Figura 12 – A operação $U_{c,a}\,|c\rangle\,|a\rangle$ só será aplicada se $|c\rangle = |1\rangle$.

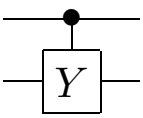
Podemos construir uma porta controlada a partir de qualquer outra porta lógica quântica, como por exemplo, as Portas de Pauli controladas da Figura 13.



(a) Porta X controlada.



(b) Porta Z controlada.



(c) Porta Y controlada.

Figura 13 – Portas de Pauli controladas.

A Porta X controlada é também chamada de `CNOT`. O nome vem da porta `NOT` dos circuitos lógicos, pois ela aplica um *not*, levando o qubit do estado $|0\rangle \to |1\rangle$ e $|1\rangle \to |0\rangle$, tendo um outro qubit como controle. A porta `CNOT` é normalmente representada como na figura 14.

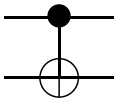


Figura 14 – Porta `CNOT`.

3.1.3 Porta de SWAP

A porta de SWAP, representada em um circuito como na Figura 15, troca dois qubits entre si. Esta porta pode ser construída a partir de três CNOT, como mostra a Figura 16.

Figura 15 – Porta de SWAP.

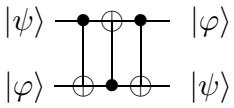


Figura 16 – Circuito que faz uma operação de SWAP entre dois qubits.

Com $[cnot_{c,a}]$ representando uma CNOT entre o qubit de controle $|c\rangle$ e o qubit alvo $|a\rangle$, partindo do estado inicial $|\psi\varphi\rangle$, a execução do circuito da Figura 16 é dada como:

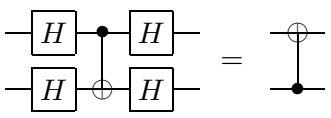
$$|\psi\varphi\rangle = a\,|00\rangle + b\,|01\rangle + c\,|10\rangle + d\,|11\rangle \tag{3.4}$$

$$[cnot_{0,1}]\,|\psi\varphi\rangle = a\,|00\rangle + b\,|01\rangle + c\,|11\rangle + d\,|10\rangle \tag{3.5}$$

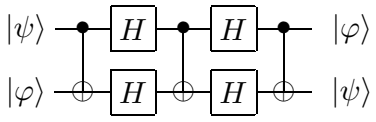
$$[cnot_{1,0}][cnot_{0,1}]\,|\psi\varphi\rangle = a\,|00\rangle + b\,|11\rangle + c\,|01\rangle + d\,|10\rangle \tag{3.6}$$

$$[cnot_{0,1}][cnot_{1,0}][cnot_{0,1}]\,|\psi\varphi\rangle = a\,|00\rangle + b\,|10\rangle + c\,|01\rangle + d\,|11\rangle = |\varphi\psi\rangle \tag{3.7}$$

Devido a limitações arquiteturais, como no computador quântico IBM Q 5(10), podemos aplicar a porta $[cnot_{c,a}]$, porém nem sempre conseguimos aplicar a porta $[cnot_{a,c}]$. Para contrapor essa limitação podemos usar o circuito da Figura 17a para construir um SWAP de maneira alternativa, como mostra na Figura 17b.



(a) Circuitos de CNOT equivalente.



(b) Circuito de SWAP com Hadamard.

Figura 17 – Implementação alternativa do circuito de SWAP.

3.2 Portas de medida

Nos circuitos quântico as medidas, na base computacional, são representadas por portas de medida, onde, logo após a medida, o qubit passa a ser considerado como um bit clássico, sendo representado por duas linhas ao invés de uma, como na Figura 18.

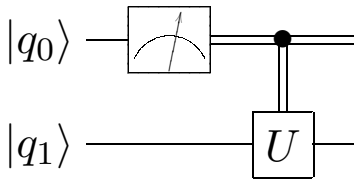


Figura 18 – A medida de $|q_0\rangle$ é usada como controle para porta U.

3.3 Exemplo de circuitos quânticos

3.3.1 Estados de Bell

Os estados de Bell são compostos por dois qubits emaranhados, sendo os quatro estados

$$|\beta_{00}\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}, \qquad |\beta_{01}\rangle = \frac{|01\rangle + |10\rangle}{\sqrt{2}}, \tag{3.8}$$

$$|\beta_{10}\rangle = \frac{|00\rangle - |11\rangle}{\sqrt{2}} \text{ e} \qquad |\beta_{11}\rangle = \frac{|01\rangle - |10\rangle}{\sqrt{2}}, \tag{3.9}$$

que podem ser preparados pelo circuito da Figura 19.

Os estados de Bell são estados extremamente correlacionados, onde a medida de um qubit afeta diretamente na medida do outro. Como por exemplo, se temos dois qubit no estado $|\beta_{10}\rangle$ e medimos 1 no primeiro qubit, colapsamos esse estado em $|11\rangle$, logo, a medida do segundo qubit só pode resultar em 1.

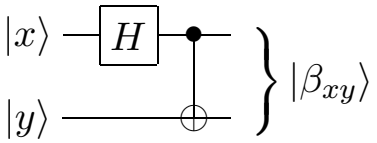


Figura 19 – Circuito de Bell.

3.3.2 Transformada de Fourier Quântica

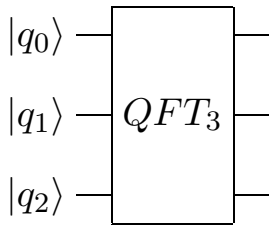
A Transformada de Fourier Quântica é a aplicação da Transformada Discreta de Fourier em uma superposição. Sua aplicação transforma uma superposição quântica de maneira que seja possível extrair informações da mesma. Dentre suas várias aplicações, a Transformada de Fourier Quântica é usada no algoritmo de fatoração de Shor(11).

A matriz que efetua a Transformada de Fourier Quântica em n qubits é

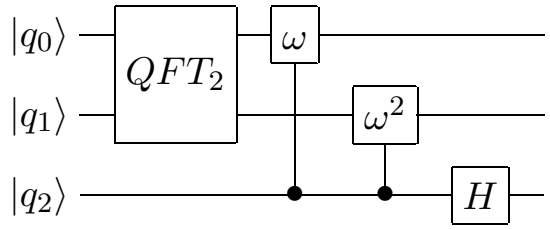
$$QFT_n = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix}, \tag{3.10}$$

onde $N = 2^n$ e $\omega = e^{\frac{2i\pi}{N}}$, podemos dizer também que o elemento da coluna i e linha j de QFT_n é igual a ω^{ij}.

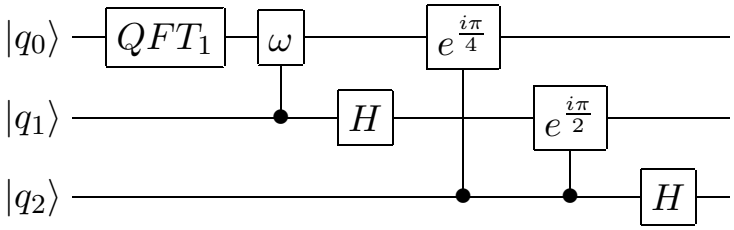
Inspirado na implementação da Transformada Rápida de Fourier, a Transformada de Fourier Quântica pode ser implementada eficientemente utilizando $O(\log^2 N)$ portas lógicas quânticas. Esse processo reduz uma QFT_N em uma QFT_{N-1} com a adição de $n-1$ portas de fase controladas e uma porta de Hadamard, como é exemplificado nos circuitos da Figura 20 que implementam QFT_3. Para mais informações sobre a implementação veja 9[Secção 5.1].



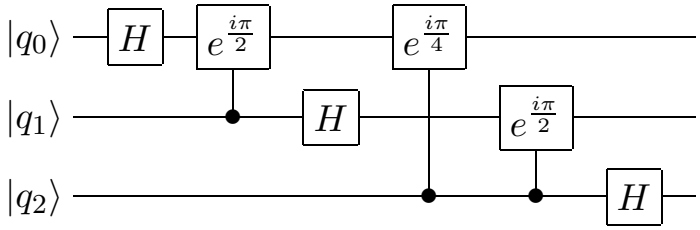
(a) Porta QFT_3.



(b) Implementação do QFT_3 utilizando QFT_2, onde $\omega = e^{\frac{2\pi i}{2^3}}$



(c) Reduzindo QFT_2 para $QFT_1 = H$, $\omega = e^{\frac{2\pi i}{2^2}}$.



(d) Circuito que implementa QFT_3 usando apenas Hadamard e portas de fase controlada.

Figura 20 – Processo recursivo de criação do circuito que implementa QFT_3.

4 Matriz densidade

Matriz densidade ou operador densidade é uma forma alternativa ao vetor de estado para descrever um sistema quântico. Todo formalismo já visto em termos de estado de vetor pode ser visto em termos de matriz densidade, porém a matriz densidade pode trazer informações úteis em alguns casos.

Neste capítulo será apresentado o conceito de estado misto (Secção 4.1) e a reformulação dos quatro postulados da mecânica quântica, apresentados no Capítulo 2, para o formalismo da matriz densidade (Secção 4.2).

4.1 Estados mistos

Um qubit $|\psi\rangle$ é descrito em matriz densidade como $\rho = |\psi\rangle\langle\psi|$. Este estado é o que chamamos de estado puro pois não temos nenhuma incerteza sobre o estado do sistema, apenas sobre resultados de possíveis medidas. Porém, há casos onde não temos total conhecimento do sistema, sendo assim, o sistema pode estar no estado $|\psi_i\rangle$ com uma probabilidade p_i, nesse caso descrevemos o sistema a partir de uma matriz densidade como

$$\rho \equiv \sum_i p_i \, |\psi_i\rangle\langle\psi_i| \, . \tag{4.1}$$

Como por exemplo, se temos um sistema que tem 50% de chance de estar no estado $|0\rangle$ e 50% de chance de estar no estado $|1\rangle$, podemos representa-lo como matriz densidade da seguinte forma:

$$0.5 \, |0\rangle\langle 0| + 0.5 \, |1\rangle\langle 1| = \frac{1}{2} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} . \tag{4.2}$$

Note que essa é uma incerteza clássica, diferente da incerteza quântica, de 50% de chance de medir 0 e 50% de chance de medir 1, dada pelo estado $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$, na qual a representação em matriz densidade é

$$\left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right)\left(\frac{\langle 0| + \langle 1|}{\sqrt{2}}\right) = \frac{1}{2} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} . \tag{4.3}$$

4.2 Postulados da mecânica quântica para matriz densidade

Todos os postulados definidos no Capítulo 2 podem ser redefinidos em termos de matriz densidade, como vemos a seguir.

4.2.1 Postulado 1: Espaço do sistema com matriz densidade

O primeiro postulado diz respeito a representar um sistema quântico.

> **Postulado 1:** Associado a cada sistema quântico fechado há um espaço de Hilbert, e o estado do sistema é totalmente representado por um operador densidade ρ, com traço igual a um, pertencente a esse espaço. Se o sistema estiver o estado ρ_i com probabilidade p_i, então o sistema é descrito como $\sum_i p_i \rho_i$, sendo $\sum_i p_i = 1$.

A operação traço é definida com sendo o somatório dos elementos da diagonal de uma matriz, sendo o traço da matriz A igual a

$$\mathrm{tr}(A) = \sum_i A_{ii}. \tag{4.4}$$

Algumas propriedades da operação traço são definidas no apêndice A.5.

Restringir que o traço da matriz densidade seja igual a um é equivalente a restringir que o vetor que representa o estado seja unitário,

$$\mathrm{tr}(|\psi\rangle\langle\psi|) = 1 \equiv ||\,|\psi\rangle\,|| = 1. \tag{4.5}$$

4.2.2 Postulado 2: Evolução do sistema com matriz densidade

O segundo postulado diz respeito a evolução do sistema quântico no tempo, ou seja, como a computação é realizada.

> **Postulado 2:** A evolução de um sistema quântico fechado é descrito pela aplicação de um *operador unitário*, ou seja, a transição de um estado ρ^0 no tempo t_0 para o estado ρ^1 no tempo t_1 pode ser descrita por um operador unitário U, sendo $\rho^1 = U\rho^0 U^\dagger$.

A evolução do sistema para matriz densidade pode ser facilmente deduzida a partir da evolução do vetor de estados. Sendo ρ formado por $|\psi_i\rangle$ com probabilidade p_i, aplicação do operador unitário U em $|\psi_i\rangle$ resulta em $|\psi_i'\rangle$ com $U\,|\psi_i\rangle = |\psi_i'\rangle$, portanto à aplicação do operador em ρ que resulta em ρ' é dada por

$$\begin{aligned}
\rho' &= \sum_i p_i\,|\psi_i'\rangle\langle\psi_i'| && (4.6)\\
&= \sum_i p_i U\,|\psi_i\rangle\langle\psi_i|\,U^\dagger && (4.7)\\
&= U\left(\sum_i p_i\,|\psi_i\rangle\langle\psi_i|\right)U^\dagger && (4.8)\\
&= U\rho U^\dagger. && (4.9)
\end{aligned}$$

4.2.3 Postulado 3: Medida com matriz densidade

O terceiro postulado descreve o comportamento de uma medida.

> **Postulado 3:** Uma medida quântica é descrita por uma conjunto de *operadores de medida* $\{M_m\}$, onde o índice m indica o possível resultado da medida. Sendo ρ o estado logo antes da medida, a probabilidade m é
>
> $$p(m) = \operatorname{tr}(M_m^\dagger M_m \rho). \tag{4.10}$$
>
> E o estado logo após a medida igual a
>
> $$\frac{M_m \rho M_m^\dagger}{\operatorname{tr}(M_m^\dagger M_m \rho)}. \tag{4.11}$$
>
> Os operadores de medida precisam satisfazer a seguinte a *equação de completude*
>
> $$\sum_m M_m^\dagger M_m = I. \tag{4.12}$$

A medida também pode ser facilmente deduzida a partir da medida do vetor de estado. Sendo ρ formado pelos estados $|\psi_i\rangle$ com a probabilidade p_i, a probabilidade de medir m de ρ é a média ponderada da probabilidade de medir m em $|\psi_i\rangle$.

$$\begin{aligned} p(m) &= \sum_i p_i \langle\psi_i| M_m^\dagger M_m |\psi_i\rangle && (4.13)\\ &= \sum_i p_i \operatorname{tr}(M_m^\dagger M_m |\psi_i\rangle\langle\psi_i|) && (4.14)\\ &= \operatorname{tr}(M_m^\dagger M_m \rho). && (4.15) \end{aligned}$$

Sendo o estado de ρ logo após medir m igual

$$\begin{aligned} \rho_m &= \sum_i p_i \left(\frac{M_m |\psi_i\rangle}{\sqrt{\langle\psi_i| M_m^\dagger M_m |\psi_i\rangle}} \frac{\langle\psi_i| M_m^\dagger}{\sqrt{\langle\psi_i| M_m^\dagger M_m |\psi_i\rangle}} \right) && (4.16)\\ &= \sum_i p_i \left(\frac{M_m |\psi_i\rangle\langle\psi_i| M_m^\dagger}{\langle\psi_i| M_m^\dagger M_m |\psi_i\rangle} \right) && (4.17)\\ &= \frac{M_m \rho M_m^\dagger}{\operatorname{tr}(M_m^\dagger M_m \rho)}. && (4.18) \end{aligned}$$

4.2.4 Postulado 4: Sistemas composto com matriz densidade

O quarto postulado diz respeito a composição de sistemas quânticos, ou seja, como funciona a dinâmica com vários qubits.

> **Postulado 4:** O estado de um sistema composto é dado pelo produto tensorial dos seus componentes, ou seja, um sistema composto por n sistemas quânticos, nos estados ρ_0, ρ_1, ..., ρ_{n-1}, tem seu estado total representado como $\rho_0 \otimes \rho_1 \otimes \cdots \otimes \rho_{n-1}$.

Toda dedução feita para um qubit também vale para n qubits em matriz densidade.

5 Canais de erro

Na teoria, um sistema fechado tem sua evolução dada como na figura 21, sem nenhuma interferência de outros sistemas, porém, na prática, isolar totalmente um sistema quântico parece ser um tarefa muito difícil. Assim, precisamos de mecanismos para modelar a interferência de outros sistemas no nosso sistema de interesse.

Neste capítulo, é apresentado como se caracteriza a influência do meio ambiente em um computador quântico (Secção 5.1) e uma ferramenta para trabalhar com essa influência (Secção 5.2), e, por fim, são apresentados os erros tradicionais que encontramos em um computador quântico (Secção 5.3).

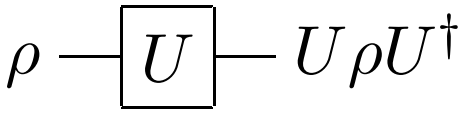


Figura 21 – Evolução unitária.

5.1 Influência do meio ambiente em um sistema fechado

Quando estamos falando da interferência de outros sistemas em um computador quântico, chamamos essa interferência de um erro quântico. Um erro se caracteriza como na Figura 22, onde temos ρ como sendo nosso computador quântico e ρ_{env} como sendo um sistema externo, o qual podemos considerar como sendo o meio ambiente. Assim, podemos definir a influência do meio ambiente em ρ pelo operador $\mathcal{E}(\rho)$, que pode ser descrito como

$$\mathcal{E}(\rho) = \text{tr}_{env}[U(\rho \otimes \rho_{env})U^\dagger]. \tag{5.1}$$

Onde, a operação tr_{env} é o traço parcial do ρ_{env}.

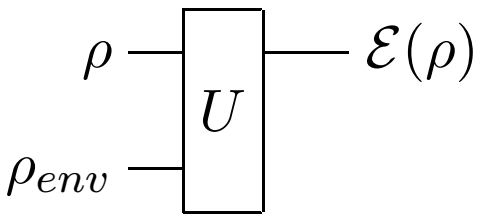


Figura 22 – Evolução com interferência do meio ambiente.

5.1.1 Traço parcial

O traço parcial é uma operação usada para mapear um objeto de um espaço vetorial em outro de dimensão menor que o original, como por exemplo, o traço parcial de B do

sistema $A \otimes B$ é

$$\mathrm{tr}_B(A \otimes B) = \mathrm{tr}(B)A. \tag{5.2}$$

Assim, conseguimos isolar o sistema A levando em consideração a influência do sistema B. De outra forma, sendo $\{|\psi_i\rangle\}$ vetores da base de B podemos descrever o traço parcial como sendo

$$\mathrm{tr}_B(A \otimes B) = \sum_i \langle\psi_i| [A \otimes B] |\psi_i\rangle . \tag{5.3}$$

5.2 Operador soma

Considerando o meio ambiente como um sistema de dimensão finita tendo $\{|e_k\rangle\}$ como base e estando no estado inicial $|e_0\rangle\langle e_0|$, podemos redefinir a equação 5.1 como

$$\begin{aligned}
\mathcal{E}(\rho) &= \mathrm{tr}_{env}[U(\rho \otimes |e_0\rangle\langle e_0|)U^\dagger] && (5.4)\\
&= \sum_k \langle e_k| [U(\rho \otimes |e_0\rangle\langle e_0|)U^\dagger] |e_k\rangle && (5.5)\\
&= \sum_k \langle e_k| U |e_0\rangle \rho \langle e_0| U^\dagger |e_k\rangle && (5.6)\\
&= \sum_k E_k \rho E_k^\dagger. && (5.7)
\end{aligned}$$

Na equação 5.7 temos a representação em operador soma do mapa $\mathcal{E}(\rho)$, sendo os E_k chamados de operadores de Kraus. Com essa representação podemos modelar qualquer interferência apenas pela parte em que influencia ρ.

O operador soma satisfaz a relação de completude mantendo o traço igual a um, sendo

$$\begin{aligned}
1 &= tr\left(\sum_k E_k \rho E^\dagger\right) && (5.8)\\
&= tr\left(\sum_k E^\dagger E_k \rho\right), && (5.9)
\end{aligned}$$

se

$$\sum_k E_k^\dagger E = I. \tag{5.10}$$

5.3 Canais de erro tradicionais

No contexto da computação quântica chamamos a interferência de outros sistemas quânticos de canal de erro. Dentre os erros mais comuns na natureza estão os canais de inversão de bit e de fase, canal de despolarização e canal de decaimento de amplitude.

Todos os possíveis estados de um qubit podem ser representados como estando dentro da esfera de Bloch, com os estados puros, estando na casca da esfera e, os estados mistos, representados por vetores de módulo inferior a um, dentro da esfera. Essa representação ajuda a visualizar a atuação que cada canal de erro tem sobre um estado.

As figuras apresentadas nas próximas subsecções foram geradas utilizando o código que pode ser visto no apêndice C.

5.3.1 Canais de inversão de bit

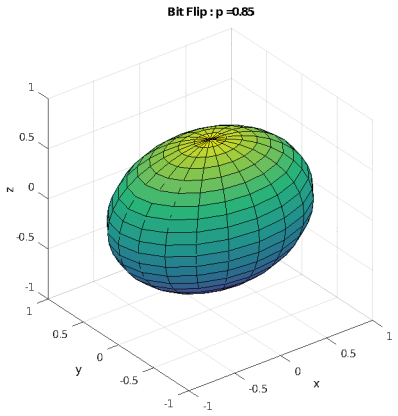
O canal de inversão de bit, ou *bit flip*, inverte o qubit entre os estados $|0\rangle$ e $|1\rangle$ com probabilidade $1-p$ e não altera o sistema com probabilidade p. O canal é representado pelos operadores de Kraus

$$E_0 = \sqrt{p}I, \tag{5.11}$$
$$E_1 = \sqrt{1-p}X. \tag{5.12}$$

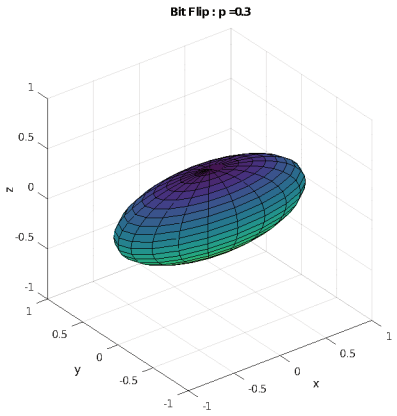
A atuação do canal no estado ρ é dada como

$$\mathcal{E}(\rho) = \sum_k E_k \rho E_k^\dagger \tag{5.13}$$
$$= p\rho + (1-p)X\rho X \tag{5.14}$$

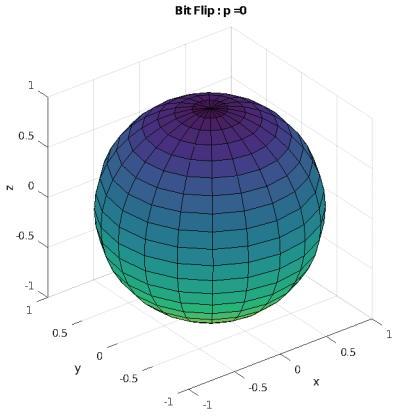
O canal de *bit flip* pode ser observado na esfera de Bloch como visto na Figura 23. Conforme p diminui até 0.5 a esfera encolhe em torno do eixo x, voltando a expandir invertida em torno do eixo x como p menor que 0.5.



(a) p = 0.85.



(b) p = 0.3.



(c) p = 0.

Figura 23 – Efeito do *bit flip* na esfera de Bloch.

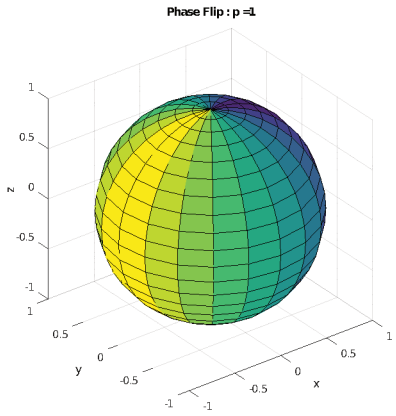
5.3.2 Canal de inversão de fase

O canal de inversão de fase, ou *phase flip*, inverte o qubit em torno do eixo z com probabilidade $1-p$ e tem probabilidade p de não alterar o sistema. O canal é representado pelos operadores de Kraus

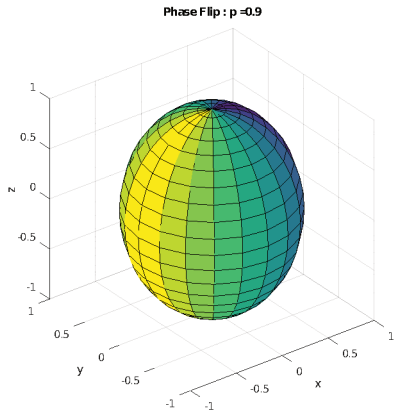
$$E_0 = \sqrt{p}I, \tag{5.15}$$

$$E_1 = \sqrt{1-p}Z. \tag{5.16}$$

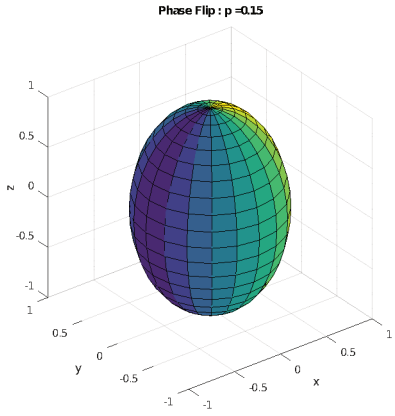
Como visto na figura 24, o canal tem um efeito semelhante ao *bit flip* na esfera de Bloch, porém atua em torno do eixo z.



(a) p = 1.



(b) p = 0.9.



(c) p = 0.15.

Figura 24 – Efeito do *Phase flip* na esfera de Bloch.

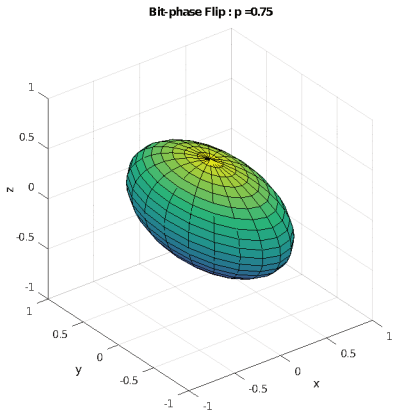
5.3.3 Canal de inversão de bit e fase

O canal de inversão de bit e fase, ou *bit-phase flip*, atua como o canal de inversão de bit e inversão de fase ao mesmo tempo, fazendo que o qubit gire em torno do eixo y com uma probabilidade $1-p$ e não altera o sistema com probabilidade p. O canal é representado pelos operadores de Kraus

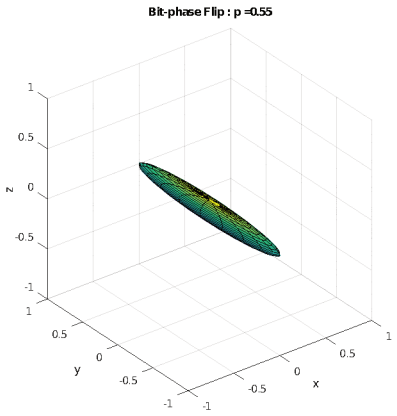
$$E_0 = \sqrt{p}I, \tag{5.17}$$

$$E_1 = \sqrt{1-p}Y. \tag{5.18}$$

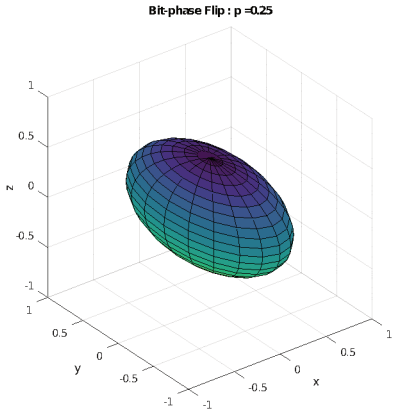
Como visto na Figura 25, o canal faz com que a esfera de Bloch encolha e expanda invertida em torno do eixo y.



(a) p = 0.75.



(b) p = 0.55.



(c) p = 0.25.

Figura 25 – Efeito do *bit-phase flip* na esfera de Bloch.

5.3.4 Canal de despolarização

O canal de despolarização tem o efeito da atuação dos três canais vistos anteriormente, levando o qubit para a identidade com $p = 1$. O canal é representado pelos operadores de Kraus

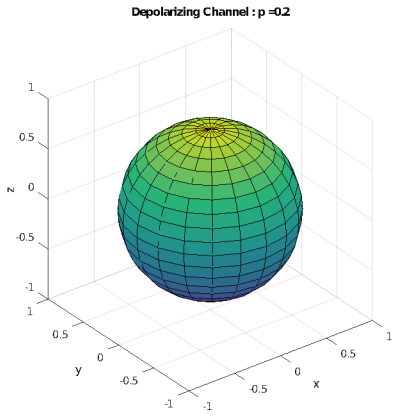
$$E_0 = \sqrt{1-3p/4}I \qquad E_1 = \sqrt{p}X/2 \tag{5.19}$$

$$E_2 = \sqrt{p}Y/2 \qquad E_3 = \sqrt{p}Z/2, \tag{5.20}$$

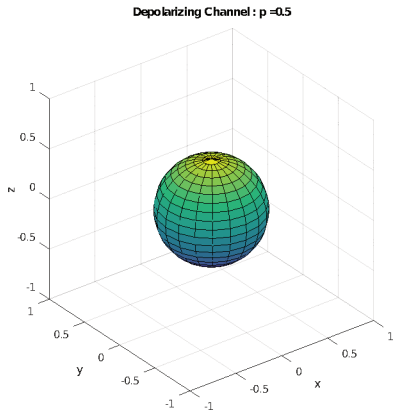
e também pode ser descrito como

$$\mathcal{E}(\rho) = \frac{pI}{2} + (1-p)\rho. \tag{5.21}$$

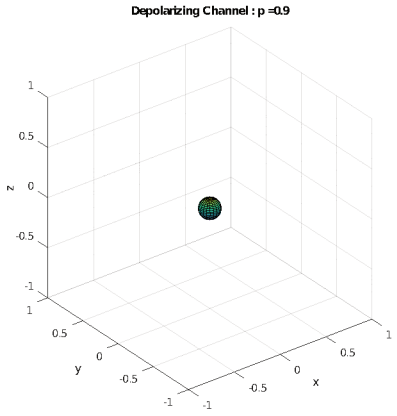
Como observado na Figura 26, conforme p aumenta a esfera diminui levando todos os estados para o centro da esfera.



(a) p = 0.2.



(b) p = 0.5.



(c) p = 0.9.

Figura 26 – Efeito do canal de despolarização na esfera de Bloch.

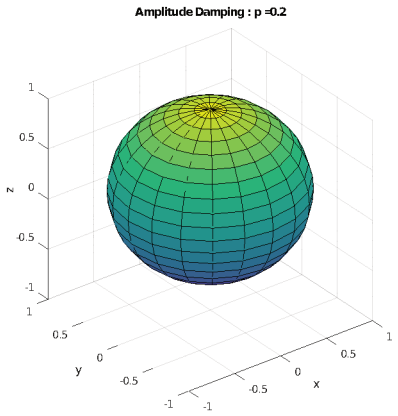
5.3.5 Decaimento de amplitude

O canal de decaimento de amplitude representa a perda de energia do sistema. Considerando que $|0\rangle$ é o estado de menor energia, o canal leva o qubit para o estado $|0\rangle$ com probabilidade p e não altera o sistema com probabilidade $1-p$. O canal é representado pelos operadores de Kraus

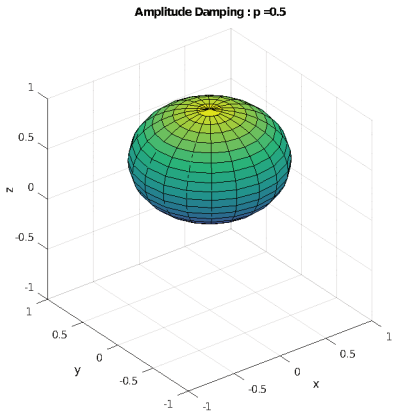
$$E_0 = \begin{bmatrix} 1 & 0 \\ 0 & \sqrt{1-p} \end{bmatrix} \tag{5.22}$$

$$E_1 = \begin{bmatrix} 0 & \sqrt{p} \\ 0 & 0 \end{bmatrix} \tag{5.23}$$

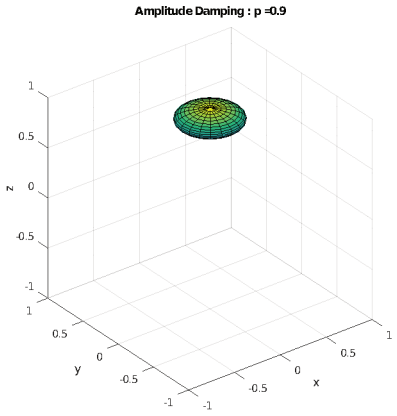
Como visto na Figura 27, conforme p aumenta todos os estado tendem a $|0\rangle$.



(a) p = 0.2.



(b) p = 0.5.



(c) p = 0.9.

Figura 27 – Efeito do canal de decaimento de amplitude na esfera de Bloch.

Parte II

Desenvolvimento do simulador

6 Simulador QSystem

Os capítulos anteriores servem como base teórica para o desenvolvimento do objetivo desse trabalho, ou seja, a implementação de um simulador de computação quântica, baseado no modelo de circuito quântico, para Python. Os próximos capítulos desse trabalho tem o intuito de documentar o que foi desenvolvido para o simulador, apresentando as estratégias usadas na implementação junto de exemplos de uso de cada funcionalidade.

O simulador foi desenvolvido como um módulo para Python 3, denominado QSystem. A escolha para tal linguagem foi feita tanto por ela ser dinâmica e fácil de usar, quanto por ser atualmente a linguagem mais usada pela comunidade de computação quântica, tendo importantes ferramentas, como QuTiP(12) e Qiskit(13), desenvolvidas para ela. Porém, apesar de ter sido construído para Python, por questões de desempenho, o simulador foi implementado em C++17 utilizando a biblioteca de álgebra linear Armadillo(7). Essa biblioteca foi escolhida, pois traz boa performance, além de possuir funcionalidades similares ao Octave(14), facilitando o porte de protótipos escritos para tal ferramenta, e gera um código final mais limpo, aumentando a manutenibilidade do mesmo. Para que as classes desenvolvidas em C++ possam ser instanciadas no Python, foi utilizado a ferramenta SWIG(8), que gera automaticamente a interface entre o que que foi desenvolvido em C++ e o interpretador do Python. A utilização do SWIG possibilita que grande parte do código fonte do simulador não tenha dependência a API[1]do Python.

Todo código fonte do simulador está disponível no Anexo D e no repositório git <https://gitlab.com/evandro-crr/qsystem>. Há também, uma *Wiki* documentando o uso do simulador (em inglês), disponível no site <https://gitlab.com/evandro-crr/qsystem/wikis> e no Anexo E.

O módulo QSystem é composto apenas de duas classes, a classe `QSystem`, que armazena e implementa toda a dinâmica de evolução e medida do estado quântico, e a classe `Gates`, que armazena as portas lógicas quânticas de um qubit e as portas de múltiplos qubits criadas pelo usuário.

6.1 Classe `Gates`

A classe `Gates` (cabeçalho disponível em D.6.1) fornece ao simulador (classe `QSystem`) algumas portas lógicas para a computação, sendo elas: as portas de múltiplos qubits criadas pelo usuário; e, as portas lógicas de um qubit, sejam elas criadas pelo usuário ou já pré carregadas em toda instância da classe.

[1] API: Interface de Programadores de Aplicativos, do inglês *Application Programmers Interface*

As portas lógicas quânticas são representadas por matrizes complexas esparsas (desenvolvidas por Sanderson e Curtin(15) para biblioteca Armadillo) e são armazenadas em dois dicionários, sendo um para as portas de um qubit (tipo de chave = `char`) e outro para portas de múltiplos qubits (tipo de chave = `std::string`). Como o tipo da chave do dicionário que armazena as portas de um qubit é `char`, essas portas devem ser representadas por apenas um caractere, como `'X'`, `'Y'` e `'Z'`, já as portas que afetam múltiplos qubits devem ser representadas por pelo menos dois caracteres, para que haja a diferenciação entre os dois tipos de porta.

Todas as portas quânticas de um qubit listadas na Subsecção 3.1.1, estão pré carregadas em toda instância da classe `Gates`. Além das portas pré carregadas é possível criar outras, como será visto no próximo capítulo.

6.2 Classe `QSystem`

A classe `QSystem` (cabeçalho disponível em D.6.3), que armazena e evolui o estado quântico, possui dois modos de representação, sendo eles: vetor de estado (como visto no Capítulo 2), que possibilita a instância de mais qubits; e, matriz densidade (como visto no Capítulo 4), que possibilita a representação de estados misto e, assim, a simulação de erros quânticos. Independente do modo de representação, o estado quântico é armazenado, da mesma forma que as portas lógicas quânticas, em uma matriz complexa esparsa.

Além de armazenar o estado quântico, a classe `QSystem` armazena os resultados das medidas, o número de qubits do sistema, dentre outras estruturas de controle que serão vistas nos próximos capítulos.

O construtor da classe QSystem recebe quatro parâmetros, sendo eles, em sequência: `nqbits`, número de qubits que será criado; `gates`, instância da classe Gates; `seed`, número que inicia o gerador de números pseudo aleatório, com valor padrão = `42`; `state`, *string* indicando qual será a representação do estado quântico, sendo os valores `'vector'` e `'matrix'` referentes a, respectivamente, estar em vetor e matriz densidade, possui valor padrão = `'vector'`. Todos os qubits são inicializados no estado $|0\rangle$.

Devido a limitações da ferramenta SWIG, não é possível usar *keyword argument* no construtor da classe `QSystem`, porém, é possível invoca-lo de 3 formas distintas, sendo as:

```
q = QSystem(nqbits, gates, seed, state) # Fornecendo os quatro parâmetros
q = QSystem(nqbits, gates, seed)        # Usando representação em vetor
q = QSystem(nqbits, gates)              # Representação em vetor e seed=42
```

7 Evolução e portas lógicas quânticas

O simulador é inspirado no modelo de circuito quântico (Capítulo 3), assim, a evolução do estado quântico é feita através de portas lógicas quânticas. Há vários métodos pertencentes a classe `QSystem` que aplicam portas lógicas quânticas ao sistema, sendo eles: o método `evol`, que aplica uma porta lógica, armazenada na instância da classe `Gates`, ao sistema quântico (Secções 7.2 e 7.3); método `cnot`, que aplica uma porta CNOT no sistema, como visto na Subsecção 3.1.2 (Secção 7.4); método `cphase`, que aplica uma fase relativa controlada no sistema (Secção 7.5); método `swap` que troca dois qubits entre si, como visto na Subsecção 3.1.3 (Subsecção 7.6); e, por último, o método `qft` que aplica uma Transformada de Fourier Quântica no sistema, como visto na Subsecção 3.3.2 (Subsecção 7.7).

A aplicação das portas lógicas quânticas é inspirada na técnica de avaliação preguiçosa(16, Subsecção 14.1.1) que possibilita a evolução atrasada do sistema. Sendo assim, uma porta lógica só é aplicada no sistema caso a aplicação da mesma influencie no resultado de uma nova ação, onde, essa nova ação normalmente é caracterizada por uma média ou uma evolução no mesmo qubit no qual a porta atua. Com essa técnica é possível diminuir o número de multiplicações matriciais, como visto na Secção 7.1.

As implementações dos métodos apresentados neste capítulo estão nos arquivos `src/gates.cpp` (D.7.1), `src/qs_evol.cpp` (D.7.5) e `src/qs_make.cpp` (D.7.6).

7.1 Evolução do estado e avaliação preguiçosa

Antes de apresentar como as portas lógicas são aplicadas no simulador, veremos um pequeno exemplo do efeito da avaliação preguiçosa no número de multiplicações matriciais.

Considerando um sistema de três qubits, iniciados no estado $|000\rangle$, onde é aplicado, em ordem, as portas lógicas quânticas H_0, X_1, $\texttt{SWAP}_1^0$ e X_2, e, logo em seguida, tudo é medido. Sem a avaliação preguiçosa teríamos 4 multiplicações matriz-vetor (ou 8 multiplicações matriz-matriz no caso de matriz densidade) antes da medida, como pode ser visto nas equações 7.1 a 7.4.

$$H_0 |000\rangle = (|000\rangle + |100\rangle)/\sqrt{2} \tag{7.1}$$

$$X_1(|000\rangle + |100\rangle)/\sqrt{2} = (|010\rangle + |110\rangle)/\sqrt{2} \tag{7.2}$$

$$\texttt{SWAP}_1^0(|010\rangle + |110\rangle)/\sqrt{2} = (|100\rangle + |110\rangle)/\sqrt{2} \tag{7.3}$$

$$X_2(|100\rangle + |110\rangle)/\sqrt{2} = (|101\rangle + |111\rangle)/\sqrt{2} \tag{7.4}$$

$$\text{Mede estado } (|101\rangle + |111\rangle)/\sqrt{2} \tag{7.5}$$

Fazendo uso da estrutura de dados *lista de portas*, que será detalhada mais à frente, o simulador QSystem, utilizando a avaliação preguiçosa, consegue diminuir para 2 o número de multiplicações matriz-vetor, como pode ser visto no exemplo abaixo:

`q = QSystem(3, gates)`	`Estado =` $\|000\rangle$`,`
	`lista de portas = [I, I, I].`
`q.evol('H', 0)`	`Adiciona H na posição 0 da lista,`
	`lista de portas = [H, I, I].`
`q.evol('X', 1)`	`Adiciona X na posição 1 da lista,`
	`lista de portas = [H, X, I].`
`q.swap(0, 1)`	`Conflite na lista de portas,`
	`estado =` $HXI\,\|000\rangle = (\|010\rangle + \|110\rangle)/\sqrt{2}$`,`
	`lista de portas = [SWAP, SWAP, I].`
`q.evol('X', 2)`	`Adiciona X na posição 2 da lista,`
	`lista de portas = [SWAP, SWAP, X].`
`q.measure_all()`	`Resultado da medida depende da evolução`
	`estado =` $(\texttt{SWAP}_1^0 \otimes X)(\|010\rangle + \|110\rangle)/\sqrt{2}$`,`
	`mede estado` $(\|101\rangle + \|111\rangle)/\sqrt{2}$

Além de diminuir o número de multiplicações matriciais, adicionar operações não conflitantes à lista de portas toma tempo constatante.

7.1.1 Lista de portas

A estrutura de dados *lista de portas* armazena as informações necessárias para a evolução atrasada do sistema, de tal forma que, mantendo a ordem do programa, seja possível aplicar mais de uma porta lógica quântica em uma única multiplicação matricial.

Cada item da lista de portas é uma instância da classe `QSystem::Gate_aux` correspondente a um qubit, onde o i-ésimo item da lista corresponde ao qubit i. A classe `Gate_aux` armazena os seguintes valores: `tag`, `enum` que indica o tipo de porta que deve ser aplicada no qubit, tendo `GATE_1`, `GATE_N`, `CNOT`, `CPHASE`, `SWAP` e `QFT` como valores possíveis; `data`, estrutura `std::variant`[1] que armazena informações complementares para geração da porta; `size`, número de qubits afetados pela porta; `inver`, booleano que indica se deve ou não ser aplicado a porta inversa, tendo por padrão o valor `False`, que significa não aplicar a porta inversa.

Um item da lista de portas é considerado livre/disponível caso seus atributos sejam `tag = Gate_aux::GATE_1`, `data = 'I'` e `size = 1`, ou seja, se tiver atribuído à ele a porta identidade.

[1] `std::variant` é uma estrutura do tipo `union` proposta por Naumann(17) e aceita como parte do C++17.

Durante a instância de um objeto da classe `QSystem`, a lista de portas, com tamanho igual ao número de qubits, é criada com todos seus itens disponíveis.

7.1.2 Método sync

Quando é necessário que as evoluções presentes na lista de portas sejam aplicadas, o método `QSystem::sync` é chamado. Este método gera uma matriz com todas as operações pendentes e a aplica no sistema. Nesta subsecção é apresentado como isso é feito.

Quando o método `QSystem::sync` é chamado, para não fazer uma operação desnecessária, inicialmente é verificado se a variável booleana `_sync` indica alguma alteração no sistema quântico. Caso tenha valor falso, a lista de portas é iterada juntando todos os itens em uma única matriz usando a operação de produto tensorial (Subsecção 2.4.1). Durante esse processo dois métodos são essenciais, sendo eles, o `QSystem::ops` que retorna um elemento da lista de portas, e o `QSystem::get_gate` que retorna a matriz referente a porta lógica. O processo de criação da matriz usada na evolução é feito pelo código abaixo:

```
sp_cx_mat evolm = get_gate(ops(0));
for (size_t i = ops(0).size; i < size(); i += ops(i).size)
  evolm = kron(evolm, get_gate(ops(i)));
```

Neste código, na primeira linha, é inicializada uma matriz esparsa com a porta que deve ser aplicada no primeiro qubit. Na linha seguinte, um `for` é iterado de `ops(0).size` a `size()`[2]. Esse `for` começa em `ops(0).size` pois a porta presente em `ops(0)` pode afetar vários qubits, porém, apenas o primeiro item da lista é que armazena as informações necessárias para que o método `get_gate` retorne a matriz referente a porta. Este é o mesmo motivo pelo qual a variável `i` é incrementada em `ops(i).size`. Já na terceira linha, a matriz `evolm` é sobrescrita pelo produto tensorial dela com a matriz referente do `for`. Ao final desses passos a matriz `evolm` possui todas as operações presentes na lista de portas.

Tendo a matriz `evolm`, a evolução do estado, em representação vetorial e em matriz densidade, é igual ao apresentado nas Secção 2.2 e Subsecção 4.2.2, respectivamente.

Ao final da execução do método `sync`, a lista de portas é reiniciada com todos os itens livres, e o booleano `_sync` é mudado para verdadeiro.

7.1.2.1 Método `QSystem::get_gate`

O método `get_gate` é implementado usando uma estrutura condicional que chama outros métodos, os quais geram ou retornam as matrizes referentes a porta lógica corres-

[2] O método `QSystem::size()` retorna o número de qubits do sistema.

pondente. Este método recebe apenas um argumento, sendo, uma instância da subclasse `Gate_aux` que possui as informações necessárias para retornar à matriz necessária.

As próximas secções, entre outras coisas, apresentam como são geradas as matrizes retornadas pelo método `get_gate`.

7.2 Porta de um qubit

A aplicação de uma porta lógica de um qubit é feita através do método `QSystem::evol`, que recebe quatro parâmetros, sendo eles, em sequência: `gate`, `char` referente a porta que será aplicada; `qbit`, índice do primeiro qubit no qual a porta será aplicada; `count`, número de qubits afetados, pois a porta `gate` será aplicada nos qubits de índice `qbit` a `qbit+count`, intervalo fechado-aberto, valor padrão = `1`; `inver`, booleano que indica se deve ser aplicada a porta inversa a `gate`, valor padrão = `false`.

Quando o método `evol` é chamado, inicialmente é verificado se as posições `qbit` a `qbit+count`, intervalo fechado-aberto, da lista de portas estão disponíveis, caso negativo o método `sync` é chamado. Logo após, as posições `qbit` a `qbit+count` são marcadas com `tag = Gate_aux::GATE_1`, `data = gate` e `size = 1`.

Quando um item da lista de portas com `tag = Gate_aux::GATE_1` é passado para o método `get_gate`, a matriz que é retornada vem da instância da classe `Gates`. Lembrando que por padrão as seguintes portas estão disponíveis:

$$\texttt{'I'} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}; \qquad \texttt{'X'} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}; \tag{7.6}$$

$$\texttt{'Y'} = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}; \qquad \texttt{'Z'} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}; \tag{7.7}$$

$$\texttt{'H'} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}; \qquad \texttt{'S'} = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}; \text{e,} \tag{7.8}$$

$$\texttt{'T'} = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{bmatrix}. \tag{7.9}$$

7.2.0.1 Exemplo de uso do método `evol`

```
from qsystem import Gates, QSystem
gates = Gates()
q = QSystem(7, gates)
q.evol(gate='X', qbit='2') # Aplica a porta X_2
q.evol('H', 0, count=7)    # Aplica a porta HHHHHHH
q.evol('S', 4, inver=True) # Aplica a porta S_4^†
```

7.2.1 Criação de porta de um qubit

Além das portas lógicas quânticas já existentes por padrão, é possível criar novas portas, para isso é usado o método `Gates::make_gate`. Este método recebe apenas dois argumentos, sendo eles, em sequência: `name`, `char`, ou uma `str` de um caractere no Python, referente ao nome da nova porta lógica quântica; e, `matrix`, um `std::vector<std::complex<double>>`, ou uma `list` de `complex` no Python, com 4 elementos, representando, em ordem, os elementos a, b, c e d, sendo $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ a matriz da nova porta lógica.

7.2.1.1 Exemplo de uso do método `make_gate` no Python

```
from qsystem import Gates, QSystem
from cmath import exp, pi
gates = Gates()
gates.make_gate(name='R', matrix=[1, 0, 0, exp(1j*pi/6)]) # R = [1 0; 0 e^(iπ/6)]
q = QSystem(1, gates)              # +1.0000          |0⟩
q.evol('X', 0)                     # +1.0000          |1⟩
q.evol('R', 0)                     # +0.8660+0.5000i|1⟩
```

7.3 Porta de múltiplos qubits

Utilizando o mesmo método `evol` é possível aplicar portas de múltiplos qubits, a única diferença é que, na chamada do método, o argumento `name` deve possuir mais de um caractere, assim, nesta chamada, os qubits afetados são de `qbit` a `qbit+count`×(tamanho da porta)[3].

Os itens da lista de portas são marcados com `tag = Gate_aux::GATE_N`, `data = gate` e `size =` (tamanho da porta). O método `get_gate` atua da mesma forma vista na secção anterior.

As portas de múltiplos qubits aplicadas pelo método `evol` também são armazenadas na instância da classe `Gates`, porém, não há nenhuma porta carregada por padrão. As próximas subsecções apresentam diferentes maneiras de criar uma porta lógica de múltiplos qubits.

7.3.1 Criação de porta a partir de matriz esparsa

É possível criar uma porta lógica quântica a partir de uma matriz esparsa com o método `Gates::make_mgate`. Este método recebe 5 argumentos, sendo eles, em sequência:

[3] Tamanho da porta: número de qubits afetados

`name`, o nome da nova porta lógica que será criada, deve conter dois ou mais caracteres; `size`, o número de qubits que será afetado; e, os vetores `row`, `col` e `value`, que são usados para a construção da matriz esparsa `m`, de tamanho $2^{\texttt{size}} \times 2^{\texttt{size}}$, da forma `m(row[i], col[i]) = value[i]`.

7.3.1.1 Exemplo de criação de porta usando o método `make_mgate`

```
from qsystem import Gates, QSystem
def func(): # func : |abc⟩|0⟩ → |abc⟩|(a ∨ b) ∧ c⟩
    row, col, value = [], [], []
    for x in range(2**3):
        a, b, c = bool(x & 4), bool(x & 2), bool(x & 1)
        d = (a or b) and c
        row.append((x << 1) | d)
        col.append(x << 1)                          # Estado final =
        value.append(1)                             # +0.354        |0000⟩
    return row, col, value                          # +0.354        |0010⟩
gates = Gates()                                     # +0.354        |0100⟩
row, col, value = func()                            # +0.354        |0111⟩
gates.make_mgate('func', 4, row, col, value)        # +0.354        |1000⟩
q = QSystem(4, gates)                               # +0.354        |1011⟩
q.evol('H', 0, 3)                                   # +0.354        |1100⟩
q.evol('func', 0)                                   # +0.354        |1111⟩
```

7.3.2 Criação de porta controlada XZ

O método `Gates::make_cgate` cria portas controladas de X e Z. Por exemplo, uma porta controlada de $XZZXZI$ que é aplicada apenas quando o qubit 5 estiver no estado $|1\rangle$. Esse tipo de porta é utilizada para a aplicação de estabilizadores em códigos de correção de erro, como pode ser visto no exemplo de código da Secção 13.2.

A porta controlada de Y não é implementada pois pode ser obtida a partir da combinação das portas controladas de X e Z.

Para gerar esse tipo de porta é necessário que seja passado para o método `make_cgate` os seguintes argumentos, em ordem: `name`, nome da porta, deve possuir dois ou mais caracteres; `gates`, `std::string`, ou `str` do Python, com as portas que serão aplicadas quando os qubits de controle estiverem no estado $|1\rangle$; e, `control`, lista com o índice dos qubits de controle.

Para entender como a matriz desta porta é criada é necessário estabelecer a seguinte

propriedade: todo operador unitário

$$U = UI, \tag{7.10}$$

sendo que U opera em n qubits e I tem tamanho $2^n \times 2^n$, podemos reescrever I como

$$I = |0\rangle\langle 0| + |1\rangle\langle 1| + |2\rangle\langle 2| + \cdots + |2^n{-}1\rangle\langle 2^n{-}1|\,. \tag{7.11}$$

Assim, podemos reescrever a Equação 7.10 como

$$U = UI = U\,|0\rangle\langle 0| + U\,|1\rangle\langle 1| + U\,|2\rangle\langle 2| + \cdots + U\,|2^n{-}1\rangle\,\langle 2^n{-}1|\,. \tag{7.12}$$

Portanto, se sabemos o efeito que o operador U tem sobre os vetores da base, sabemos como construir a matriz que o representa.

Antes de apresentar o algoritmo para gerar a matriz U é necessário estabelecer que o produto externo $|\texttt{i}\rangle\langle\texttt{j}|$ gera uma matriz toda em zero, exceto pelo elemento `(i, j) = 1`, idem para $e^{i\theta}\,|\texttt{i}\rangle\langle\texttt{j}|$, exceto que `(i, j) =` $e^{i\theta}$. Assim, para gerar a matriz U, correspondente a porta controlada desejada, é necessário seguir os seguintes passos:

1. Gere a máscara `x` com valor 1 nos bits onde na *string* `gates` estiver o valor `'X'`.

   (*e.g.* `x` para `gates = "XZZXZI"` é `0b100100`[4] `= 36`)

2. Gere a máscara `z` com valor 1 nos bits onde na *string* `gates` estiver o valor `'Z'`.

   (*e.g.* `z` para `gates = "XZZXZI"` é `0b011010 = 26`)

3. `size =` tamanho da *string* `gates`.

4. Crie uma matriz U de dimensão $2^{\texttt{size}} \times 2^{\texttt{size}}$ com todos os elementos em zero.

5. Para `i` entre 0 e $2^{\texttt{size}}$, faça:

   5.1. Se para todo `c` em `control`, o bit `size-c-1` de i for igual a 1, então:[5,6]

   $$U \texttt{ += } (-1)^{\text{paridade}(\texttt{i \& z})}\,|\texttt{i\^{}x}\rangle\langle\texttt{i}| \tag{7.13}$$

   5.2. Se não:

   $$U \texttt{ += } |\texttt{i}\rangle\langle\texttt{i}| \tag{7.14}$$

[4] O prefixo `0b` indica que o número está na base binaria.
[5] A função paridade retorna 0 se o número de bits for par e 1 caso contrário.
[6] Os operadores `&` e `^` representam, respectivamente, as operações *e* bit a bit e *ou exclusivo* bit a bit.

7.3.2.1 Exemplo de criação de porta usando o método `make_cgate`

```
from qsystem import Gates, QSystem
gates = Gates()
gates.make_cgate(name='cixz', gates='IXZ', control=[0])
                                                    # Estado final =
q = QSystem(3, gates)                               # +0.500        |000⟩
q.evol('H', 0)                                      # +0.500        |001⟩
q.evol('H', 2)                                      # +0.500        |110⟩
q.evol('cixz', 0)                                   # -0.500        |111⟩
```

7.3.3 Criação de porta a partir de função

É possível aplicar uma função do Python no sistema quântico. Para isso, é necessário criar uma porta lógica quântica utilizando o método `Gates::make_fgate`. Este método recebe 4 argumentos, sendo eles, em sequência: `name`, nome da porta quântica que será criada, deve conter dois ou mais caracteres; `func`, função do Python que será usada para criar a porta lógica quântica, essa função deve receber e retornar um inteiro positivo, e atuar no intervalo de 0 a $2^{\texttt{size}}-1$; `size`, número de qubits afetados; e, `iterator`, objeto Python iterável que retorna números entre 0 e $2^{\texttt{size}}-1$ na sequência em que a porta deve ser criada, esse é um argumento opcional com valor padrão `= None`, ou seja, iterar na sequência de `range(2**size)`.

Para gerar a matriz `m`, correspondente a porta desejada, é usada a mesma lógica apresentada nas Equações 7.10 a 7.12. Portanto, sendo `m` uma matriz de tamanho $2^{\texttt{size}} \times 2^{\texttt{size}}$, iniciada toda em zero, a matriz final é contraída da seguinte forma:

```
for i in iterator:
    m(func(i), i) = 1
```

7.3.3.1 Exemplo de criação de porta usando o método `make_fgate`

```
from qsystem import Gates, QSystem
def func(x): # |abc⟩|0⟩ → |abc⟩|(a ∨ b) ∧ c⟩
    a, b, c = bool(x & 8), bool(x & 4), bool(x & 2)
    d = (a or b) and c
    return x | d                                    # Estado final =
def iterator():                                     # +0.354        |0000⟩
    for x in range(2**3):                           # +0.354        |0010⟩
        yield x << 1                                # +0.354        |0100⟩
gates = Gates()                                     # +0.354        |0111⟩
gates.make_fgate('func', func, 4, iterator())       # +0.354        |1000⟩
```

```
q = QSystem(4, gates)                    # +0.354        |1011⟩
q.evol('H', 0, 3)                        # +0.354        |1100⟩
q.evol('func', 0)                        # +0.354        |1111⟩
```

7.4 Porta cnot

O método `QSystem::cnot` é utilizado para aplicar uma porta `CNOT` ou uma porta X com vários qubits de controle (com visto na Subsecção 3.1.2). Este método recebe dois argumentos, sendo eles, em ordem: `target`, índice do qubit alvo; e, `control`, lista de índices dos qubits de controle.

Na chamada do método `cnot`, caso algum item entre `minq = min(targer, control)` e `maxq = max(target, control)` da lista de portas não estiver livre, o método `sync` é chamado. Logo após, os item `minq` a `maxq` são marcados com `tag = Gate_aux::CNOT`, e o item `minq`, adicionalmente, `data = (target-minq, [c-minq para c em control])` e `size = maxq-minq+1`.

Quando um item da lista de portas com `tag = Gate_aux::CNOT` é passado para o método `get_gate`, o método `QSystem::make_cnot` é chamado para gerar a matriz referente a porta `CNOT`. A criação da matriz é semelhante a apresentada na Subsecção 7.3.2, com um algoritmo ainda mais simples, apresentado abaixo:

1. Crie uma matriz `m` de dimensão $2^{\texttt{size}} \times 2^{\texttt{size}}$ com todos os elementos em zero.
2. Para `i` entre 0 e $2^{\texttt{size}}$, faça:
   2.1. Se para todo `c` em `control`, o bit `size-c-1` de i for igual a 1, então[7]

$$\texttt{m += |i\^{}(1 << (size-target-1))⟩⟨i|} \tag{7.15}$$

   2.2. Se não:

$$\texttt{m += |i⟩⟨i|} \tag{7.16}$$

7.5 Porta cphase

O método `QSystem::cphase` aplica uma fase relativa controlada, ou seja, se os qubits de controle estiverem todos em $|1\rangle$, o qubit alvo recebe uma fase relativa. Este método recebe três argumentos, sendo eles, em sequência: `phase`, fase que será aplicada; `target`, índice do qubit alvo; e `control`, lista de índices dos qubits de controle.

Este método atua de maneira semelhante ao `cnot`, porém marcando `tag = Gate_aux::CPHASE` e `data = (phase, target-minq, [c-minq para c em control])`.

[7] A operação de deslocamento logico a esquerda `a << b` desloca `a` `b` bits a esquerda. `1 << b` $= 2^{\texttt{b}}$.

Quando um item da lista de portas com `tag = Gate_aux::CPHASE` é passado para o método `get_gate`, o método `QSystem::make_cphase` é chamado para gerar a matriz referente a porta. Da mesma forma que o `make_cnot`, a criação da matriz é semelhante a apresentada na Subsecção 7.3.2, com um algoritmo ainda mais simples, apresentado abaixo:

1. Crie uma matriz `m` de dimensão $2^{\texttt{size}} \times 2^{\texttt{size}}$ com todos os elementos em zero.
2. Para `i` entre 0 e $2^{\texttt{size}}$, faça:
   2.1. Se para todo `c` em `control`, o bit `size-c-1` de i for igual a 1, então:
   $$\texttt{m += phase}^{\text{paridade(i \& (1 << (size-target-1)))}} |\texttt{i}\rangle\langle\texttt{i}| \tag{7.17}$$
   2.2. Se não:
   $$\texttt{m += } |\texttt{i}\rangle\langle\texttt{i}| \tag{7.18}$$

7.6 Porta swap

O método `QSystem::swap` troca o estado de dois qubits entre si (com visto na Subsecção 3.1.3). Este método recebe dois argumentos, `qbit_a` e `qbit_b` referentes aos índices dos qubits que serão trocados. Quando este método é chamado, caso algum item da lista de portas entre os índices `qbit_a` a `qbit_b`, incluindo os mesmos, não estiver livre, o método `sync` é chamado. Logo após, aos mesmo itens, é atribuído a `tag = Gate_aux::SWAP`, e ao primeiro item, o valor `size` igual a $|\texttt{qbit\_b-qbit\_a}| + 1$.

Quando um item com `tag = Gate_aux::SWAP` é passado para o método `get_gate`, quem retorna a matriz é o método `QSystem::make_swap`. Como o item da lista de portas armazena somente a informação `size`, é criado uma porta de tamanho `size`, onde o primeiro e o último qubit são trocados.

Para gerar a matriz `m` correspondente a porta SWAP desejada, é usado um algoritmo que pode ser dividido em duas partes, pois, os vetores da base computacional $|\texttt{i}\rangle$, onde $\texttt{i} < 2^{\texttt{size-1}}$ e `i` é par, não sofrem alteração pela porta `SWAP`, da mesma forma que para $\texttt{i} > 2^{\texttt{size}-1}$ e `i` ímpar. Essa característica pode ser vista no exemplo com três qubits, abaixo:

$$\texttt{SWAP}_2^0 |000\rangle = |000\rangle \qquad \texttt{SWAP}_2^0 |001\rangle = |100\rangle \tag{7.19}$$
$$\texttt{SWAP}_2^0 |010\rangle = |010\rangle \qquad \texttt{SWAP}_2^0 |011\rangle = |110\rangle \tag{7.20}$$

$$\texttt{SWAP}_2^0 |100\rangle = |001\rangle \qquad \texttt{SWAP}_2^0 |101\rangle = |101\rangle \tag{7.21}$$
$$\texttt{SWAP}_2^0 |110\rangle = |011\rangle \qquad \texttt{SWAP}_2^0 |111\rangle = |111\rangle\,. \tag{7.22}$$

Considerando essas características, o algoritmo pode ser descrito como:

1. Crie uma matriz m de dimensão $2^{\texttt{size}} \times 2^{\texttt{size}}$ com todo os elementos em zero.
2. Para i entre 0 e $2^{\texttt{size-1}}$:
   2.1. Se i for ímpar:
   $$\texttt{m += |(i| (1 << (size-1))) ^ 1⟩⟨i|} \tag{7.23}$$
   2.2. Se não:
   $$\texttt{m += |i⟩⟨i|} \tag{7.24}$$
3. Para i entre 0 e $2^{\texttt{size-1}}$:
   3.1. Se i for par:
   $$\texttt{m += |i ^ 1⟩⟨i | (1 << (size-1))|} \tag{7.25}$$
   3.2. Se não:
   $$\texttt{m += |i | (1 << (size-1))⟩⟨i | (1 << (size-1))|} \tag{7.26}$$

7.7 Porta qft

Para aplicar uma Transformada de Fourier Quântica, como visto na Subsecção 3.3.2, é utilizado o método `QSystem::qft`. Este método recebe três parâmetros, sendo eles `qbegin` e `qend`, correspondente ao intervalo, fechado-aberto, de qubits no qual a porta será aplicada; e, `inver`, booleano indicando se deve ou não ser aplicado a Transformada de Fourier Quântica inversa, com valor padrão `= False`, logo, não aplicar.

A chamada do método `qft` altera os itens `qbegin` a `qend`, intervalo fechado-aberto, da lista de portas, logo, os mesmos precisam estar disponíveis, caso contrário, o método `sync` é chamado. Os itens da lista de portas são marcados com `tag = Gate_aux::QFT` e `size = qend-qbegin`.

Quando um item da lista de portas com `tag = Gate_aux::QFT` é passado para o método `get_gate`, é o método `QSystem::make_qft` que é responsável por retornar a matriz. Este método constrói a matriz diretamente a partir da definição $A_{i,j} = \frac{1}{\sqrt{2^{\texttt{size}}}} w^{ij}$, onde $w = e^{\frac{2i\pi}{2^{\texttt{size}}}}$.

7.8 Exemplos de uso dos método cnot, cphase, swap e qft

```
from qsystem import Gates, QSystem
from cmath import exp, pi
```

```
gates = Gates()
q = QSystem(7, gates)
q.cnot(0, [2])                      # Aplica uma CNOT, qubit 0 como
                                    # alvo e qubit 2 controle
q.cphase(-1, 3, [2])                # Aplica uma porta Z controlada
q.cphase(1j, 5, [1, 4])             # Aplica uma porta S controlada
q.cphase(exp(1j*pi/4), 6, [0])      # Aplica uma porta T controlada
q.swap(3, 5)                        # SWAP entre os qubits 3 e 5
q.qft(0, 7)                         # Transformada de Fourier quântica
                                    # em todos as 7 qubits
q.qft(0, 7, True)                   # Aplica uma transformada de Fourier
                                    # quântica inversa
```

8 Operação de medida

O simulador implementa a operação de medida na base computacional (medida em Z). Contudo, a medida em outras bases pode ser alcançada a partir de operações prévias, como apresentado em (18), para as medidas de Pauli.

Há dois métodos que efetuam medidas no sistema quântico: o método `QSystem::measure`, que efetua a média em uma sequência de qubits, e o método `QSystem::measure_all` que efetua a medida de todos os qubits.

O Método `measure` recebe dois argumentos, sendo eles: `qbit`, índice do primeiro qubit que será medido; e, `count`, número de qubits que serão medidos a partir de `qbit`, possui o valor padrão = 1. Já o método `measure_all` não recebe nenhum argumento.

Quando um qubit é medido, seu resultado é armazenado em uma lista acessível pelo método `QSystem::bits`. Cada elemento da lista corresponde a um qubit, sendo o i-ésimo item referente ao qubit i. Os valores possíveis de uma medida são 0 e 1, sendo atribuído o valor `None` aos qubits que não foram medidos. Uma nova medida sobrescreve o resultado da medida anterior.

A implementação dos métodos apresentados neste capítulo estão disponíveis no arquivo `src/qs_measure.cpp` (D.7.7).

8.1 Implementação da operação de medida

O método que faz a operação de medida é o mesmo para vetor de estados e matriz densidade, mas nesta secção eles serão tratados separadamente. O que é comum entre as duas formas de representação é que, por mais que seja possível medir mais de um qubit simultaneamente, na prática, os qubits são medidos separadamente, pois, o número de operações cresce exponencialmente com o número de qubits medidos juntos.

Mesmo os qubits sendo medidos separadamente, o resultado final é o mesmo, como é exemplificado a seguir para vetor de estados. Seja o estado $\frac{1}{\sqrt{4}}(|0\rangle + |1\rangle + |2\rangle + |3\rangle)$, utilizando o operador de medida $|0\rangle\langle 0| \otimes |0\rangle\langle 0|$, a probabilidade de medir 0, ao mesmo tempo, em ambos os qubits é $\frac{1}{4}$ e, o estado logo após a medida é $|0\rangle$. Contudo, se inicialmente medirmos apenas o primeiro qubit com o operador $|0\rangle\langle 0| \otimes I$, temos a probabilidade de $\frac{1}{2}$ de medir 0 e, o estado logo após a medida é $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$; e, em seguida, mediar o segundo qubit como o operador $I \otimes |0\rangle\langle 0|$, temos a probabilidade de $\frac{1}{2}$ de medir 0 e, o estado logo após a medida é $|0\rangle$. Logo, obtemos o mesmo estado final e a mesma probabilidade (probabilidade de medir 0 em ambos os qubits é igual a probabilidade de medir o no primeiro e em seguida no segundo, $\frac{1}{2} \times \frac{1}{2} = \frac{1}{4}$) se medirmos os qubits juntos ou separados.

Este exemplo pode ser estendido para matriz densidade.

Antes de qualquer medida, o método `sync` é chamado para aplicar as portas lógicas quânticas pendentes no sistema.

8.1.1 Medida em vetor de estado

A medida em vetor de estado segue o mesmo princípio apresentado na Secção 2.3. Sendo `qbits` o vetor que representa o estado quântico do sistema e `q` o índice do qubit que será medido. O algoritmo para efetuar a medida pode ser descrito como:

1. $\texttt{p} = 0$
   (Variável que armazena a probabilidade de medir zero)
2. Para todo $\alpha_{\texttt{i}} |\texttt{i}\rangle$ em $(\alpha_0 |0\rangle + \alpha_1 |1\rangle + \alpha_2 |2\rangle + \cdots = \texttt{qbits})$, faça:
   (Como `qbits` é um vetor esparso, apenas onde $\alpha_{\texttt{i}} \neq 0$ são iterados)
   2.1. Se o bit do número `i` na posição `q` for `= 0`, então:
   (Se `~i & (1 << (size-q-1))`, então:)
   $$\texttt{p} \mathrel{+}= |\alpha_{\texttt{i}}|^2 \tag{8.1}$$
3. Sorteia com probabilidade `p` de sair 0 e `1-p` de sair 1:
   3.1 Se sortear 0, então:
   3.1.1 `bits()[q] = 0`
   3.1.2 Para todo $\alpha_{\texttt{i}} |\texttt{i}\rangle$ em $(\alpha_0 |0\rangle + \alpha_1 |1\rangle + \alpha_2 |2\rangle + \cdots = \texttt{qbits})$, faça:
   (Como `qbits` é um vetor esparso, apenas onde $\alpha_{\texttt{i}} \neq 0$ são iterados)
   3.1.2.1 Se o bit do número `i` na posição `q` for `= 0`, então:
   (Se `~i & (1 << (size-q-1))`, então:)
   $$\alpha_{\texttt{i}} = \frac{\alpha_{\texttt{i}}}{\sqrt{\texttt{p}}} \tag{8.2}$$
   3.1.2.2 Se não:
   $$\alpha_{\texttt{i}} = 0 \tag{8.3}$$
   3.2 Se sortear 1, então:
   3.2.1 `bits()[q] = 1`
   3.2.2 Para todo $\alpha_{\texttt{i}} |\texttt{i}\rangle$ em $(\alpha_0 |0\rangle + \alpha_1 |1\rangle + \alpha_2 |2\rangle + \cdots = \texttt{qbits})$, faça:
   (Como `qbits` é um vetor esparso, apenas onde $\alpha_{\texttt{i}} \neq 0$ são iterados)
   3.2.2.2 Se o bit do número `i` na posição `q` for `= 1`, então:
   (Se `i & (1 << (size-q-1))`, então:)
   $$\alpha_{\texttt{i}} = \frac{\alpha_{\texttt{i}}}{\sqrt{\texttt{1-p}}} \tag{8.4}$$

3.2.2.2 Se não:

$$\alpha_{\texttt{i}} = 0 \tag{8.5}$$

8.1.2 Medida em matriz densidade

A medida em matriz densidade segue o mesmo princípio apresentado na Subsecção 4.2.3. Sendo ρ a matriz que representa o estado quântico do sistema e `q` o índice do qubit que será medido. O algoritmo para efetuar a medida pode ser descrito como:

1. $\texttt{p} = 0$
   (Variável que armazena a probabilidade de medir zero)
2. Para todo $\rho_{\texttt{i,i}}$[1], faça:
   2.1. Se o bit do número `i` na posição `q` for `= 0`, então:
   (Se `~i & (1 << (size-q-1))`, então:)

$$\texttt{p} \mathrel{+}= \rho_{\texttt{i,i}} \tag{8.6}$$

3. Sorteia com probabilidade `p` de sair 0 e `1-p` de sair 1:
   3.1 Se sortear 0, então:
   3.1.1 `bits()[q] = 0`
   3.1.2 Para todo $\rho_{\texttt{i,j}}$, faça:
   (Como ρ é uma matriz esparsa, apenas onde $\rho_{\texttt{i,j}} \neq 0$ são iterados)
   3.1.2.1 Se o bit dos números `i` e `j` na posição `q` for `= 0`, então:
   (Se `~i & (1 << (size-q-1))` e `~j & (1 << (size-q-1))`, então:)

$$\rho_{\texttt{i,j}} = \frac{\rho_{\texttt{i,j}}}{\texttt{p}} \tag{8.7}$$

   3.1.2.1 Se não:

$$\rho_{\texttt{i,j}} = 0 \tag{8.8}$$

   3.2 Se sortear 1, então:
   3.2.1 `bits()[q] = 1`
   3.2.2 Para todo $\rho_{\texttt{i,j}}$, faça:
   (Como ρ é uma matriz esparsa, apenas onde $\rho_{\texttt{i,j}} \neq 0$ são iterados)
   3.2.2.1 Se o bit dos números `i` e `j` na posição `q` for `= 0`, então:
   (Se `i & (1 << (size-q-1))` e `j & (1 << (size-q-1))`, então:)

$$\rho_{\texttt{i,j}} = \frac{\rho_{\texttt{i,j}}}{\texttt{p}} \tag{8.9}$$

[1] A notação $A_{i,j}$ indica o elemento da linha i e coluna j da matriz A, da mesma forma que $A(i,j)$.

3.2.2.1 Se não:

$$\rho_{\mathtt{i},\mathtt{j}} = 0 \tag{8.10}$$

8.2 Exemplo de uso dos métodos de medida

```
from qsystem import Gates, QSystem
gates = Gates()
q = QSystem(4, gates, 33)

q.evol('H', 0)                  # (|0000⟩ + |1000⟩)/√2
q.cnot(1, [0])                  # (|0000⟩ + |1100⟩)/√2
q.evol('H', 2)                  # (|0000⟩ + |0010⟩ + |1100⟩ + |1110⟩)/√4
q.cnot(3, [2])                  # (|0000⟩ + |0011⟩ + |1100⟩ + |1111⟩)/√4
q.measure(qbit=1, count=2)      #  |0011⟩
print('measurement =', q.bits())
# measurement = [None, 0, 1, None]
q.measure_all()
print('measurement =', q.bits())
# measurement = [0, 0, 1, 1]
```

9 Implementação dos canais de erro

Os canais de erro tradicionais apresentados na Secção 5.3 são implementados no simulador através dos métodos `QSystem::flip` (Secção 9.1), `QSystem::dpl_channel` (Secção 9.2) e `QSystem::amp_damping` (Secção 9.3). Além desses erros, é possível utilizar o método `QSystem::sum` (Secção 9.4) para aplicar uma operação soma (Secção 5.2) e, assim, simular outros erros.

Para que um canal de erro ou operação soma seja aplicada no sistema quântico, é necessário que o mesmo esteja em representação de matriz densidade.

A implementação dos métodos apresentados neste capítulo estão no arquivo `src/qs_errors.cpp` (D.7.4). Não será explicado a abordagem por trás da implementação de cada método, pois eles são basicamente a implementação de um operador soma.

9.1 Canais de *bit*, *Phase* e *bit-phase flip*

O método `flip` é utilizado para aplicar um *bit flip* (Subsecção 5.3.1), *phase flip* (Subsecção 5.3.2) ou *bit-phase flip* (Subsecção 5.3.3) a um qubit do sistema. Este método recebe três argumentos, sendo eles: `gate`, `char` indicando qual erro deve ser aplicado, sendo `'X'` para *bit-flip*, `'Z'` para *phase-flip* e `'Y'` para *bit-phase flip*; `qbit`, índice do qubit onde será aplicado o erro; `p`, probabilidade do erro acontecer, sendo $0 \leq \mathtt{p} \leq 1$. Os operadores de Kraus aplicados no sistema são:

$$E_0 = \sqrt{1-\mathtt{p}}I \tag{9.1}$$

$$E_1 = \sqrt{\mathtt{p}}\sigma_{\mathtt{gate}_{\mathtt{qbit}}}, \tag{9.2}$$

onde $\sigma_{\mathtt{gate}_{\mathtt{qbit}}}$ é a matriz de Pauli referente ao argumento `gate` atuando no qubit de índice `qbit`.

9.1.1 Exemplo de uso do método `flip`

```
>>> from qsystem import Gates, QSystem, get_matrix
>>> g = Gates()
>>> q = QSystem(2, g, 13, 'matrix')
>>> q.evol('H', 1)              # 1/2(|00⟩ + |01⟩)(⟨00| + ⟨01|)
>>> q.flip('X', 0, 0.14)        # Bit flip no qubit 0 com p=0.14
>>> get_matrix(q).toarray()
array([[0.43+0.j 0.43+0.j 0.  +0.j 0.  +0.j]
       [0.43+0.j 0.43+0.j 0.  +0.j 0.  +0.j]
```

```
       [0.  +0.j 0.  +0.j 0.07+0.j 0.07+0.j]
       [0.  +0.j 0.  +0.j 0.07+0.j 0.07+0.j]])
>>> q.flip('Z', 1, 0.33)        # Phase flip no qubit 0 com p=0.14
>>> get_matrix(q).toarray()
array([[0.43  +0.j 0.1462+0.j 0.    +0.j 0.    +0.j]
       [0.1462+0.j 0.43  +0.j 0.    +0.j 0.    +0.j]
       [0.    +0.j 0.    +0.j 0.07  +0.j 0.0238+0.j]
       [0.    +0.j 0.    +0.j 0.0238+0.j 0.07  +0.j]])
```

9.2 Canal de despolarização

O método `dpl_channel` aplica o canal despolarização (Subsecção 5.3.4) em um qubit, levando o mesmo ao estado maximamente misto com probabilidade `p`. Este método recebe dois argumentos, sendo eles: `qbit`, índice do qubit onde será aplicado o erro; `p`, probabilidade do erro acontecer, sendo $0 \leq \mathtt{p} \leq 1$. Os operadores de Kraus aplicados no sistema são:

$$E_0 = \sqrt{1 - 3\mathtt{p}/4}I \qquad E_1 = \sqrt{\mathtt{p}}X_{\mathtt{qbit}}/2 \tag{9.3}$$

$$E_2 = \sqrt{\mathtt{p}}Y_{\mathtt{qbit}}/2 \qquad E_3 = \sqrt{\mathtt{p}}Z_{\mathtt{qbit}}/2 \tag{9.4}$$

9.2.1 Exemplo de uso do método `dpl_channel`

```
>>> from qsystem import Gates, QSystem, get_matrix
>>> g = Gates()
>>> q = QSystem(2, g, 66, 'matrix')
>>> q.dpl_channel(0, 0.3)    # Canal de despolarização no qubit 0 com p=0.3
>>> q.dpl_channel(1, 0.8)    # Canal de despolarização no qubit 1 com p=0.8
>>> get_matrix(q).toarray()
array([[0.37333333+0.j 0.        +0.j 0.        +0.j 0.        +0.j]
       [0.        +0.j 0.42666667+0.j 0.        +0.j 0.        +0.j]
       [0.        +0.j 0.        +0.j 0.09333333+0.j 0.        +0.j]
       [0.        +0.j 0.        +0.j 0.        +0.j 0.10666667+0.j]])
```

9.3 Canal de decaimento de amplitude

O método `amp_damping` aplica o canal de decaimento de amplitude (Subsecção 5.3.5) em um qubit, levando o mesmo ao estado $|0\rangle$ com probabilidade `p`. Este método recebe dois argumentos, sendo eles: `qbit`, índice do qubit onde será aplicado o erro; `p`, probabilidade do erro acontecer, sendo $0 \leq \mathtt{p} \leq 1$. Os operadores de Kraus aplicados no

sistema são:

$$E_0 = \begin{bmatrix} 1 & 0 \\ 0 & \sqrt{1-\mathtt{p}} \end{bmatrix}_{\mathtt{qbit}} \tag{9.5}$$

$$E_1 = \begin{bmatrix} 1 & \sqrt{\mathtt{p}} \\ 0 & 0 \end{bmatrix}_{\mathtt{qbit}} \tag{9.6}$$

9.3.1 Exemplo de uso do método amp_damping

```
>>> from qsystem import Gates, QSystem, get_matrix
>>> g = Gates()
>>> q = QSystem(2, g, 66, 'matrix')
>>> q.evol('X', 0, 2)          # |11⟩⟨11|
>>> q.amp_damping(0, 0.12)     # Decaimento de amplitude no qubit 0 com p=0.12
>>> get_matrix(q).toarray()
array([[0.  +0.j 0.  +0.j 0.  +0.j 0.  +0.j]
       [0.  +0.j 0.12+0.j 0.  +0.j 0.  +0.j]
       [0.  +0.j 0.  +0.j 0.  +0.j 0.  +0.j]
       [0.  +0.j 0.  +0.j 0.  +0.j 0.88+0.j]])
```

9.4 Operador soma

Usando o método `QSystem::soma`, é possível aplicar operadores de Kraus que seguem o seguinte formato:

$$E_0 = \sqrt{p_0} U_{00} \otimes U_{01} \otimes \cdots \otimes U_{0n} \tag{9.7}$$

$$E_1 = \sqrt{p_1} U_{10} \otimes U_{11} \otimes \cdots \otimes U_{1n} \tag{9.8}$$

$$\vdots \tag{9.9}$$

$$E_m = \sqrt{p_m} U_{m0} \otimes U_{m1} \otimes \cdots \otimes U_{mn}. \tag{9.10}$$

Onde cada U_{ij} é uma porta lógica quântica de um qubit. Para isso, é necessário dividir os operadores de Kraus em duas listas, uma para os valores de p_i e outra para os $U_{i0} \otimes U_{i1} \otimes \cdots \otimes U_{in}$. A lista de probabilidades (p_i) deve ser organizada da seguinte forma:

$$[p_0, p_1, \cdots, p_m]\,. \tag{9.11}$$

Já a segunda lista, deve ser construída de forma que, cada $U_{i0} \otimes U_{i1} \otimes \cdots \otimes U_{ij}$ seja uma `str` onde cada U_{ij} é um caractere.

Para aplicar esse método, é necessário fornecer 3 argumentos, sendo eles, em sequência: `qbit`, índice do qubit afetado por U_{i0}; `kraus`, lista de *string* com os operadores $U_{i0} \otimes U_{i1} \otimes \cdots \otimes U_{in}$; e, `p`, lista com os valores de p_i.

9.4.1 Exemplo de uso do método sum

Aplicação do canal de *bit flip* totalmente correlacionado(19) para 2 qubits composto pelos operadores de Kraus

$$E_0 = \sqrt{p_0} XX \text{ e} \tag{9.12}$$

$$E_1 = \sqrt{p_1} II, \tag{9.13}$$

onde $p_1 = 1 - p_0$.

```
from qsystem import Gates, QSystem
g = Gates()
q = QSystem(4, g, 21, 'matrix')
p0 = 0.25
p1 = 1 - p0
E0 = 'XX'
E1 = 'II'
q.sum(0, [E0, E1], [p0, p1])
print(q)
# (0, 0)    +0.750
# (3, 3)    +0.250
```

9.5 Erros em vetor de estado

Os erros quânticos apresentados geram um estado misto com uma incerteza clássica onde o erro pode ou não ter acontecido. Mesmo não sendo possível representar um estado misto na representação em vetor, é possível alcançar um resultado similar aos canais de erros implementados no simulador com as funções abaixo. Estas funções aplicam um erro com uma probabilidade `p`, porém, o estado não será misto.

Canais de *bit*, *Phase* e *bit-phase flip* :

```
def flip(q, gate,  p):
    from random import choices
    if choices([True, False], weights=[p, 1-p])[0]:
        q.evol(qbit, qbit)
```

Decaimento de amplitude :

```
def amp_damping(q, qbit, p):
    from random import choices
    if choices([True, False], weights=[p, 1-p])[0]:
```

```
            q.measure(qbit)
            if q.bits[qbit] == 1:
                q.evol('X', qbit)
```

Operador soma :

```
    def sum(q, qbit, kraus, p):
        from random import choices
        aux = 0
        for gate in choices(kraus, weights=p)[0]:
            q.evol(gate, qbit+aux)
            aux += 1
```

10 Qubits ancilares

É possível adicionar e remover qubits durante a computação. Esses qubits são chamados de ancilas ou qubits ancilares e são adicionados ao final do sistema quântico, sendo, indexados como uma continuação do sistema, ou seja, para um sistema de n qubits, onde o índice do último é $n-1$, se forem adicionados mais m qubits ancilares, o índice da primeira ancila será n e da última será $n+m-1$.

Qubits ancilares não recebem nenhum tratamento especial, sendo possível utilizá-los em qualquer método. Além disso, o resultado das medidas são armazenadas normalmente na lista acessível pelo método `bits`.

A implementação dos métodos `QSystem::add_ancillas` e `QSystem::rm_ancillas` apresentados neste capítulo estão no arquivo `src/qs_ancillas.cpp` (D.7.3).

10.1 Adicionar e remover ancilas

Para adicionar ancilas ao sistema é utilizado o método `QSystem::add_ancillas`. Este método recebe apenas um argumento, `nqbits`, indicando o número de qubits ancilares que serão adicionados. As ancilas são inicializadas no estado $|0\rangle$.

Quando qubits ancilares são adicionados, a lista de portas e a lista que armazena as medidas são estendidas, e o novo estado quântico é: para estado em vetor, sendo $|\psi\rangle_0$ o estado antes das ancilas serem adicionadas, e $|\alpha\rangle$ o sistema das ancilas, o novo estado é

$$|\psi\rangle = |\psi\rangle_0 \otimes |\alpha\rangle\,; \tag{10.1}$$

e para matriz densidade, sendo ρ_0 o estado antes das ancilas serem adicionadas, e ρ_α o sistema das ancilas, o novo estado é

$$\rho = \rho_0 \otimes \rho_\alpha. \tag{10.2}$$

Para remover as ancilas do sistema é utilizado o método `QSystem::rm_ancilas`. Este método, quando chamado, exclui os itens da lista de portas e da lista que armazena as medidas referentes aos qubits ancilares, além de removê-los. Contudo, a remoção das ancilas do sistema tem diferentes consequências. Dependendo da representação, para estados em vetor, os qubits ancilares são medidos antes de serem removidos, já para matriz densidade, os qubits são removidos com a operação de traço parcial.

Os exemplos a seguir ressaltam os diferentes efeitos do método `rm_ancillas` dependendo da representação.

- vetor de estados, resultado da medida da ancila = 1

```
q = QSystem(1, g, 44, 'vector') # |0⟩
q.add_ancillas(1)               # |00⟩
q.evol('H', 0)                  # (|00⟩ + |10⟩)/√2
q.cnot(1, [0])                  # (|00⟩ + |11⟩)/√2
q.rm_ancillas()                 # |1⟩
q.evol('H', 0)                  # (|0⟩ − |1⟩)/√2
q.measure(0)                    # − |1⟩
print('vector representation, measurement =', q.bits()[0])
# vector representation, measurement = 1
```

- Estado em matriz densidade

```
q = QSystem(1, g, 44, 'matrix') # |0⟩⟨0|
q.add_ancillas(1)               # |00⟩⟨00|
q.evol('H', 0)                  # ½(|00⟩ + |10⟩)(⟨00| + ⟨10|)
q.cnot(1, [0])                  # ½(|00⟩ + |11⟩)(⟨00| + ⟨11|)
q.rm_ancillas()                 # ½ [1 1; 1 1]
q.evol('H', 0)                  # |0⟩⟨0|
q.measure(0)                    # |0⟩⟨0|
print('density matrix representation, measurement =', q.bits()[0])
# density matrix representation, measurement = 0
```

Uma chamada do método `rm_ancillas` remove, uma a uma, todas as ancilas dos sistema. Para vetor de estados, esse processo é feito da seguinte forma:

1. Mede o último qubit.
2. Se o resultado for 0, então o vetor que representa o estado é
$$\begin{bmatrix} |\psi\rangle \\ 0 \end{bmatrix} \tag{10.3}$$
3. Se o resultado for 1, então o vetor que representa o estado é
$$\begin{bmatrix} 0 \\ |\psi\rangle \end{bmatrix} \tag{10.4}$$
4. Sendo $|\psi\rangle$ o vetor que representa o estado sem o último qubit, repita o processo.

E, para matriz densidade, as ancilas são traçadas uma a uma utilizando o algoritmo apresentado na próxima subsecção.

10.1.1 Implementação da operação de traço parcial

A operação de traço parcial (Subsecção 5.1.1) é implementada para um caso bem específico. O traço parcial do último qubit do sistema, ou seja

$$\mathrm{tr}_B(A \otimes B) = A\,\mathrm{tr}(B), \tag{10.5}$$

onde $A \otimes B$ representado o sistema quântico como um todo, e B representa o último qubit. Podemos reescrever a Equação 10.5 como

$$A \sum_i \langle i|\, B\, |i\rangle = \sum_i \langle i|_B\, AB\, |i\rangle_B\,, \tag{10.6}$$

onde $|i\rangle_B = I \otimes |i\rangle$, sendo I da mesma dimensão de A. Como B representa apenas um qubit podemos reescrever a Equação 10.6 como

$$\sum_i \langle i|_B\, AB\, |i\rangle_B = \langle 0|_B\, AB\, |0\rangle_B + \langle 1|_B\, AB\, |1\rangle_B \tag{10.7}$$

Tendo isso em mãos, para exemplificar, podemos aplicar a operação de traço parcial no estado $|\psi\rangle\langle\psi|$ composto por 3 qubits. Sendo

$$\begin{aligned} |\psi\rangle = {} & \sqrt{\alpha_0}\,|000\rangle + \sqrt{\alpha_1}\,|001\rangle + \sqrt{\alpha_2}\,|010\rangle + \sqrt{\alpha_3}\,|011\rangle + \\ & \sqrt{\alpha_4}\,|100\rangle + \sqrt{\alpha_5}\,|101\rangle + \sqrt{\alpha_6}\,|110\rangle + \sqrt{\alpha_7}\,|111\rangle\,, \text{ e} \end{aligned} \tag{10.8}$$

$$|\psi\rangle\langle\psi| = \begin{bmatrix} |\alpha_0| & \sqrt{\alpha_0\alpha_1^*} & \sqrt{\alpha_0\alpha_2^*} & \sqrt{\alpha_0\alpha_3^*} & \sqrt{\alpha_0\alpha_4^*} & \sqrt{\alpha_0\alpha_5^*} & \sqrt{\alpha_0\alpha_6^*} & \sqrt{\alpha_0\alpha_7^*} \\ \sqrt{\alpha_1\alpha_0^*} & |\alpha_1| & \sqrt{\alpha_1\alpha_2^*} & \sqrt{\alpha_1\alpha_3^*} & \sqrt{\alpha_1\alpha_4^*} & \sqrt{\alpha_1\alpha_5^*} & \sqrt{\alpha_1\alpha_6^*} & \sqrt{\alpha_1\alpha_7^*} \\ \sqrt{\alpha_2\alpha_0^*} & \sqrt{\alpha_2\alpha_1^*} & |\alpha_2| & \sqrt{\alpha_2\alpha_3^*} & \sqrt{\alpha_2\alpha_4^*} & \sqrt{\alpha_2\alpha_5^*} & \sqrt{\alpha_2\alpha_6^*} & \sqrt{\alpha_2\alpha_7^*} \\ \sqrt{\alpha_3\alpha_0^*} & \sqrt{\alpha_3\alpha_1^*} & \sqrt{\alpha_3\alpha_2^*} & |\alpha_3| & \sqrt{\alpha_3\alpha_4^*} & \sqrt{\alpha_3\alpha_5^*} & \sqrt{\alpha_3\alpha_6^*} & \sqrt{\alpha_3\alpha_7^*} \\ \sqrt{\alpha_4\alpha_0^*} & \sqrt{\alpha_4\alpha_1^*} & \sqrt{\alpha_4\alpha_2^*} & \sqrt{\alpha_4\alpha_3^*} & |\alpha_4| & \sqrt{\alpha_4\alpha_5^*} & \sqrt{\alpha_4\alpha_6^*} & \sqrt{\alpha_4\alpha_7^*} \\ \sqrt{\alpha_5\alpha_0^*} & \sqrt{\alpha_5\alpha_1^*} & \sqrt{\alpha_5\alpha_2^*} & \sqrt{\alpha_5\alpha_3^*} & \sqrt{\alpha_5\alpha_4^*} & |\alpha_5| & \sqrt{\alpha_5\alpha_6^*} & \sqrt{\alpha_5\alpha_7^*} \\ \sqrt{\alpha_6\alpha_0^*} & \sqrt{\alpha_6\alpha_1^*} & \sqrt{\alpha_6\alpha_2^*} & \sqrt{\alpha_6\alpha_3^*} & \sqrt{\alpha_6\alpha_4^*} & \sqrt{\alpha_6\alpha_5^*} & |\alpha_6| & \sqrt{\alpha_6\alpha_7^*} \\ \sqrt{\alpha_7\alpha_0^*} & \sqrt{\alpha_7\alpha_1^*} & \sqrt{\alpha_7\alpha_2^*} & \sqrt{\alpha_7\alpha_3^*} & \sqrt{\alpha_7\alpha_4^*} & \sqrt{\alpha_7\alpha_5^*} & \sqrt{\alpha_7\alpha_6^*} & |\alpha_7| \end{bmatrix}, \tag{10.9}$$

a operação de traço parcial seria

$$(II\,\langle 0|)\,|\psi\rangle\langle\psi|\,(II\,|0\rangle) + (II\,\langle 1|)\,|\psi\rangle\langle\psi|\,(II\,|1\rangle). \tag{10.10}$$

Inicialmente vamos calcular $(II\,\langle 0|)\,|\psi\rangle =$

$$\begin{aligned} (II\,\langle 0|)(& \sqrt{\alpha_0}\,|000\rangle + \sqrt{\alpha_1}\,|001\rangle + \sqrt{\alpha_2}\,|010\rangle + \sqrt{\alpha_3}\,|011\rangle + \\ & \sqrt{\alpha_4}\,|100\rangle + \sqrt{\alpha_5}\,|101\rangle + \sqrt{\alpha_6}\,|110\rangle + \sqrt{\alpha_7}\,|111\rangle). \end{aligned} \tag{10.11}$$

Podemos reescrever a Equação 10.11, distribuindo o produto interno, como

$$\begin{aligned} & \sqrt{\alpha_0}\,|00\rangle\,\langle 0|0\rangle + \sqrt{\alpha_1}\,|00\rangle\,\langle 0|1\rangle + \sqrt{\alpha_2}\,|01\rangle\,\langle 0|0\rangle + \sqrt{\alpha_3}\,|01\rangle\,\langle 0|1\rangle + \\ & \sqrt{\alpha_4}\,|10\rangle\,\langle 0|0\rangle + \sqrt{\alpha_5}\,|10\rangle\,\langle 0|1\rangle + \sqrt{\alpha_6}\,|11\rangle\,\langle 0|0\rangle + \sqrt{\alpha_7}\,|11\rangle\,\langle 0|1\rangle\,. \end{aligned} \tag{10.12}$$

E, resolvendo o produto interno temos que

$$(II\langle 0|)|\psi\rangle = \sqrt{\alpha_0}|00\rangle + \sqrt{\alpha_2}|01\rangle + \sqrt{\alpha_4}|10\rangle + \sqrt{\alpha_6}|11\rangle. \tag{10.13}$$

Assim, podemos resolver $(II\langle 0|)|\psi\rangle\langle\psi|(II|0\rangle)$ como

$$\begin{bmatrix} |\alpha_0| & \sqrt{\alpha_0\alpha_2^*} & \sqrt{\alpha_0\alpha_4^*} & \sqrt{\alpha_0\alpha_6^*} \\ \sqrt{\alpha_2\alpha_0^*} & |\alpha_2| & \sqrt{\alpha_2\alpha_4^*} & \sqrt{\alpha_2\alpha_6^*} \\ \sqrt{\alpha_4\alpha_0^*} & \sqrt{\alpha_4\alpha_2^*} & |\alpha_4| & \sqrt{\alpha_4\alpha_6^*} \\ \sqrt{\alpha_6\alpha_0^*} & \sqrt{\alpha_6\alpha_2^*} & \sqrt{\alpha_6\alpha_4^*} & |\alpha_6| \end{bmatrix} \tag{10.14}$$

e, da mesma forma, $(II\langle 1|)|\psi\rangle\langle\psi|(II|1\rangle)$ como

$$\begin{bmatrix} |\alpha_1| & \sqrt{\alpha_1\alpha_3^*} & \sqrt{\alpha_1\alpha_5^*} & \sqrt{\alpha_1\alpha_7^*} \\ \sqrt{\alpha_3\alpha_1^*} & |\alpha_3| & \sqrt{\alpha_3\alpha_5^*} & \sqrt{\alpha_3\alpha_7^*} \\ \sqrt{\alpha_5\alpha_1^*} & \sqrt{\alpha_5\alpha_3^*} & |\alpha_5| & \sqrt{\alpha_5\alpha_7^*} \\ \sqrt{\alpha_7\alpha_1^*} & \sqrt{\alpha_7\alpha_3^*} & \sqrt{\alpha_7\alpha_5^*} & |\alpha_7| \end{bmatrix} \tag{10.15}$$

logo $(II\langle 0|)|\psi\rangle\langle\psi|(II|0\rangle) + (II\langle 1|)|\psi\rangle\langle\psi|(II|1\rangle) =$

$$\begin{bmatrix} |\alpha_0| + |\alpha_1| & \sqrt{\alpha_0\alpha_2^*} + \sqrt{\alpha_1\alpha_3^*} & \sqrt{\alpha_0\alpha_4^*} + \sqrt{\alpha_1\alpha_5^*} & \sqrt{\alpha_0\alpha_6^*} + \sqrt{\alpha_1\alpha_7^*} \\ \sqrt{\alpha_2\alpha_0^*} + \sqrt{\alpha_3\alpha_1^*} & |\alpha_2| + |\alpha_3| & \sqrt{\alpha_2\alpha_4^*} + \sqrt{\alpha_3\alpha_5^*} & \sqrt{\alpha_2\alpha_6^*} + \sqrt{\alpha_3\alpha_7^*} \\ \sqrt{\alpha_4\alpha_0^*} + \sqrt{\alpha_5\alpha_1^*} & \sqrt{\alpha_4\alpha_2^*} + \sqrt{\alpha_5\alpha_3^*} & |\alpha_4| + |\alpha_5| & \sqrt{\alpha_4\alpha_6^*} + \sqrt{\alpha_5\alpha_7^*} \\ \sqrt{\alpha_6\alpha_0^*} + \sqrt{\alpha_7\alpha_1^*} & \sqrt{\alpha_6\alpha_2^*} + \sqrt{\alpha_7\alpha_3^*} & \sqrt{\alpha_6\alpha_4^*} + \sqrt{\alpha_7\alpha_5^*} & |\alpha_6| + |\alpha_7| \end{bmatrix} \tag{10.16}$$

Com esse exemplo fica mais fácil visualizar como para o caso específico do traço do último qubit a matriz $C = \mathrm{tr}_B(A \otimes B)$ pode ser descrita como

$$C_{i,j} = (A \otimes B)_{2i,2j} + (A \otimes B)_{2i+1,2j+1}. \tag{10.17}$$

Assim, sendo ρ o estado antes do traço parcial, o algoritmo que efetua a operação de traço parcial é descrito como:

1. `size` = número de qubits em ρ.
2. Crie uma matriz C de tamanho $2^{\texttt{size}-1} \times 2^{\texttt{size}-1}$ com todos os elementos em zero.
3. Para todo $\rho_{\texttt{i},\texttt{j}}$, faça:
   3.1. Se `i` (mod 2) for igual a `j` (mod 2), faça:

$$C(\texttt{i >> 1, j >> 1}) \mathrel{+}= \rho_{\texttt{i},\texttt{j}} \tag{10.18}$$

4. Retorne C

Apesar do exemplo usado para explicar o algoritmo do traço parcial ter sido um estado puro, o mesmo também funciona em estado misto, como é demonstrado a seguir. Considere o seguinte estado misto ρ de dois qubits, onde

$$\rho = \begin{bmatrix} \alpha_{00} & \alpha_{01} & \alpha_{02} & \alpha_{03} \\ \alpha_{10} & \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{20} & \alpha_{21} & \alpha_{22} & \alpha_{23} \\ \alpha_{30} & \alpha_{31} & \alpha_{32} & \alpha_{33} \end{bmatrix} \tag{10.19}$$

e o traço parcial do último qubit é igual a $(I \langle 0|)\rho(I |0\rangle) + (I \langle 1|)\rho(I |1\rangle)$, onde

$$(I \langle 0|)\rho(I |0\rangle) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha_{00} & \alpha_{01} & \alpha_{02} & \alpha_{03} \\ \alpha_{10} & \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{20} & \alpha_{21} & \alpha_{22} & \alpha_{23} \\ \alpha_{30} & \alpha_{31} & \alpha_{32} & \alpha_{33} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} \alpha_{00} & \alpha_{02} \\ \alpha_{20} & \alpha_{22} \end{bmatrix} \tag{10.20}$$

e

$$(I \langle 1|)\rho(I |1\rangle) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha_{00} & \alpha_{01} & \alpha_{02} & \alpha_{03} \\ \alpha_{10} & \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{20} & \alpha_{21} & \alpha_{22} & \alpha_{23} \\ \alpha_{30} & \alpha_{31} & \alpha_{32} & \alpha_{33} \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \alpha_{11} & \alpha_{13} \\ \alpha_{31} & \alpha_{33} \end{bmatrix}, \tag{10.21}$$

logo

$$(I \langle 0|)\rho(I |0\rangle) + (I \langle 1|)\rho(I |1\rangle) = \begin{bmatrix} \alpha_{00} + \alpha_{11} & \alpha_{02} + \alpha_{13} \\ \alpha_{20} + \alpha_{31} & \alpha_{22} + \alpha_{33} \end{bmatrix}. \tag{10.22}$$

O algoritmo proposto retorna a mesma matriz resultante do traço o último de ρ qubit, como poder ser visto a seguir:

1. `size` $= 2$
2. $C = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$
3. $$C = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \xrightarrow{\alpha_{00}} \begin{bmatrix} \alpha_{00} & 0 \\ 0 & 0 \end{bmatrix} \xrightarrow{\alpha_{02}} \begin{bmatrix} \alpha_{00} & \alpha_{02} \\ 0 & 0 \end{bmatrix} \tag{10.23}$$
   $$\xrightarrow{\alpha_{11}} \begin{bmatrix} \alpha_{00} + \alpha_{11} & \alpha_{02} \\ 0 & 0 \end{bmatrix} \xrightarrow{\alpha_{13}} \begin{bmatrix} \alpha_{00} + \alpha_{11} & \alpha_{02} + \alpha_{13} \\ 0 & 0 \end{bmatrix} \tag{10.24}$$
   $$\xrightarrow{\alpha_{20}} \begin{bmatrix} \alpha_{00} + \alpha_{11} & \alpha_{02} + \alpha_{13} \\ \alpha_{20} & 0 \end{bmatrix} \xrightarrow{\alpha_{22}} \begin{bmatrix} \alpha_{00} + \alpha_{11} & \alpha_{02} + \alpha_{13} \\ \alpha_{20} & \alpha_{22} \end{bmatrix} \tag{10.25}$$
   $$\xrightarrow{\alpha_{31}} \begin{bmatrix} \alpha_{00} + \alpha_{11} & \alpha_{02} + \alpha_{13} \\ \alpha_{20} + \alpha_{31} & \alpha_{22} \end{bmatrix} \xrightarrow{\alpha_{33}} \begin{bmatrix} \alpha_{00} + \alpha_{11} & \alpha_{02} + \alpha_{13} \\ \alpha_{20} + \alpha_{31} & \alpha_{22} + \alpha_{33} \end{bmatrix} \tag{10.26}$$
4. Retorne $\begin{bmatrix} \alpha_{00} + \alpha_{11} & \alpha_{02} + \alpha_{13} \\ \alpha_{20} + \alpha_{31} & \alpha_{22} + \alpha_{33} \end{bmatrix}$

10.2 Exemplo de uso de ancilas no QSystem

```
from qsystem import Gates, QSystem
g = Gates()
q = QSystem(3, g)
q.evol('H', 0, 3)
q.add_ancillas(2)
q.cnot(3, [0])
q.cnot(3, [1])
q.cnot(4, [0])
q.cnot(4, [2])              # Estado final =
q.measure(3, 2)             # +0.707       |000⟩ |00⟩
print(q.bits())             # +0.707       |111⟩ |00⟩
# [None, None, None, 0, 0]
```

11 Outras funções

Além das funcionalidades do simulador já apresentadas, há outras usadas para manipular ou extrair informações do sistema quântico, sendo possível: mudar a representação do sistema de vetor de estados para matriz densidade, ou vice versa (Secção 11.1); extrair e trocar a matriz do sistema quântico do simulador (Secção 11.2); salvar/carregar o sistema quântico em/de um arquivo (Secção11.3). O simulador também permite salvar e carregar as portas de múltiplos qubits (Secção 11.4)

A implementação dos métodos e das funções apresentados neste capítulo estão nos arquivos `src/qs_utility.cpp` (D.7.8), `src/gates.cpp` (D.7.1) e `src/qsystem.i` (D.7.9).

11.1 Mudar a representação do sistema

O método `QSystem::change_to` é usado para mudar a representação do sistema quântico. Este método possui apenas um argumento indicando o novo estado do sistema, sendo os valores possíveis: `'matrix'`, para mudar a representação do sistema para matriz densidade; e, `'vector'`, para mudar a representação do sistema para vetor de estados.

A mudança do vetor de estados, $|\psi\rangle$, para matriz densidade, ρ, é dada como $\rho = |\psi\rangle\langle\psi|$. Porém, na mudança de matriz densidade para vetor de estados, a única garantia é que a probabilidade das medidas se mantenham, pois, tanto o sistema pode estar em um estado misto quanto diferentes estados podem resultar na mesma matriz densidade, como por exemplo:

$$|1\rangle \to \begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}0 & 1\end{bmatrix} = \begin{bmatrix}0 & 0\\0 & 1\end{bmatrix} = \rho \tag{11.1}$$

$$-|1\rangle \to \begin{bmatrix}0\\-1\end{bmatrix}\begin{bmatrix}0 & -1\end{bmatrix} = \begin{bmatrix}0 & 0\\0 & 1\end{bmatrix} = \rho \tag{11.2}$$

$$i|1\rangle \to \begin{bmatrix}0\\i\end{bmatrix}\begin{bmatrix}0 & -i\end{bmatrix} = \begin{bmatrix}0 & 0\\0 & 1\end{bmatrix} = \rho \tag{11.3}$$

$$-i|1\rangle \to \begin{bmatrix}0\\-i\end{bmatrix}\begin{bmatrix}0 & i\end{bmatrix} = \begin{bmatrix}0 & 0\\0 & 1\end{bmatrix} = \rho \tag{11.4}$$

onde forma não é possível diferenciar se ρ foi gerado por $|1\rangle$, $-|1\rangle$, $i|1\rangle$ ou $-i|1\rangle$.

O algoritmo usado para mudar de matriz densidade para vetor de estados é o seguinte: sendo ρ_{ij} o elemento da linha i coluna j da matriz ρ e $|0\rangle, \ldots, |2^n-1\rangle$ vetores na

base computacional para n qubits, o vetor de estados resultante da mudança é

$$|\psi\rangle = \sum_{i=0}^{2^n-1} \sqrt{\rho_{ii}}\,|i\rangle\,. \tag{11.5}$$

Seguindo o algoritmo, o estado ρ das Equações 11.1 a 11.4 resultaria no estado $|1\rangle$. Embora este algoritmo possa ser definido, ele não pode ser implementado fisicamente.

11.1.1 Exemplo de uso do método change_to

```
from qsystem import Gates, QSystem       # vetor original:
g = Gates()                              # +0.707       |0>
q = QSystem(2, g)                        # +0.500+0.500i|1>
q.evol('H', 0, 2)                        #
q.evol('S', 0)                           # matriz vinda do vetor original:
q.evol('T', 1)                           # (0, 0)    +0.500
print('vetor original:')                 # (1, 0)    +0.354+0.354i
print(q)                                 # (0, 1)    +0.354-0.354i
q.change_to('matrix')                    # (1, 1)    +0.500
print('matriz vinda do vetor original:') #
print(q)                                 # vetor vinda da matriz:
q.change_to('vector')                    # +0.707       |0>
print('vetor vinda da matriz:')          # +0.707       |1>
print(q)                                 #
```

11.2 Extrair a matriz do sistema

É possível utilizar as funções não membras (*non-member functions*) `get_matrix` e `set_matrix` para extrair e mudar a matriz do sistema quântico.

A função `get_matrix` recebe como argumento uma instância da classe `QSystem`, e retorna uma matriz esparsa `scipy.sparse.csc_matrix`(20). Já a função `set_matrix` recebe dois parâmetros, sendo eles, em sequência: `q`, instância da classe `QSystem`; e `m`, matriz esparsa `scipy.sparse.csc_matrix` com o novo estado do sistema.

11.2.1 Exemplo de uso das funções get_matrix e set_matrix

```
from qsystem import Gates, QSystem, get_matrix, set_matrix
g = Gates()
q = QSystem(1, g)
q.evol('H', 0)
m = get_matrix(q)
```

```
m = m.dot(m.transpose())
set_matrix(q, m)                    # matrix representation
print(q.state(), 'representation')  # [[0.5+0.j 0.5+0.j]
print(get_matrix(q).toarray())      #  [0.5+0.j 0.5+0.j]]
```

11.3 Salvar e carregar o estado quântico

Usando os métodos `QSystem::save` é possível salvar o estado quântico do sistema em um arquivo. Este método recebe apenas um argumento indicando o caminho do arquivo que será criado. Para carregar o estado quântico de um arquivo é utilizado o método `QSystem::load`. Este método também recebe apenas um argumento com o caminho do arquivo que armazena o estado quântico.

O arquivo criado armazena apenas a matriz do sistema em um formato binário dependente de arquitetura, definido pela biblioteca Armadillo. Como o arquivo armazena apenas a matriz do sistema, quando carregado todo os qubits são definidos como não ancilas e as listas de resultado de medidas são reiniciadas.

11.3.1 Exemplo de uso dos métodos `save` e `load`

Salvar estado quântico :

```
from qsystem import Gates, QSystem
g = Gates()
q = QSystem(3, g)
q.evol('H', 0, 2)
q.evol('S', 0)
q.evol('T', 1)
q.save('matrix.qsys')
```

Carregar estado quântico :

```
from qsystem import Gates, QSystem # loaded state
g = Gates()                        # +0.500       |000>
q = QSystem(1, g)                  # +0.354+0.354i|010>
q.load('matrix.qsys')              #       +0.500i|100>
print('loaded state')              # -0.354+0.354i|110>
print(q)                           #
```

11.4 Salvar e carregar portas lógicas

Usando o método `Gates::save` é possível armazenar as portas lógicas quânticas de múltiplos qubits, criadas pelo usuário, em um arquivo. Este método recebe apenas um argumento, `path`, indicando o endereço do arquivo que será criado. Todas as portas lógicas são armazenadas dentro de um arquivo tar, sendo o mesmo criado utilizando a biblioteca microtar[1].

Para carregar as portas lógicas salvas no arquivo tar, é necessário passar o caminho do arquivo para o construtor da classe `Gates`.

Dentro do arquivo tar as portas lógicas são armazenadas em arquivos binários, da mesma forma que o estado quântico, sendo o nome do arquivo o mesmo nome da porta.

11.4.1 Exemplo de como salvar portas lógicas quânticas

Salvar portas lógicas :

```
from qsystem import Gates, QSystem
⋮
gates = Gates()
row, col, value = func_m()
gates.make_mgate('func_m', 4, row, col, value)
gates.make_fgate('func', func, 4, iterator())
gates.make_cgate(name='cixz', gates='IXZ', control=[0])
gates.save('gates.tar')
```

Comando para listar portas lógicas dentro do arquivo tar :

```
-> tar tf gates.tar
cixz
func
func_m
```

Carregar portas lógicas :

```
from qsystem import Gates, QSystem
g = Gates('gates.tar')
print(g)
# cixz - 3 qbits long
# func - 4 qbits long
# func_m - 4 qbits long
```

[1] Biblioteca microtar disponível em <https://github.com/rxi/microtar>.

12 Montagem do módulo QSystem

Neste capítulo é apresentado o processo de montagem do módulo QSystem. Tal processo pode ser dividido em duas partes: sendo a primeira, geração da interface entre o código C++ e o interpretador do Python (Secção 12.1); e, a segunda, criação do pacote que será usado para instalar o módulo (Secção 12.2).

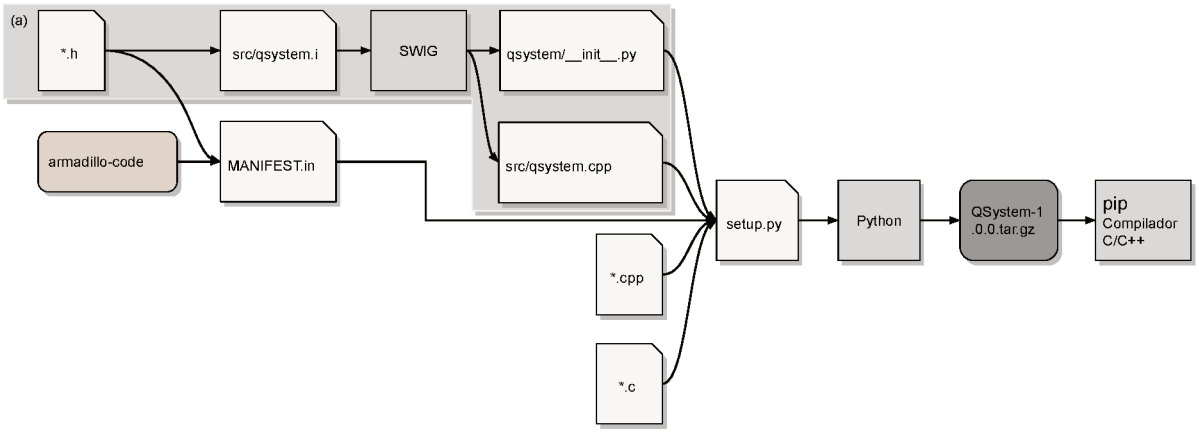


Figura 28 – Processo de criação do módulo QSystem. (a) interface C++/Python. Esse processo pode ser visto no Makefile disponível em D.2.

12.1 Interface C++/Python

A Figura 28a apresenta como a ferramenta SWIG é usada para gerar a interface entre o que foi desenvolvido em C++ para Python.

A ferramenta SWIG é usada para, mas não apenas, conectar programas escritos em C e C++ com linguagens de *script* como Python. Para fazer isso é necessário um arquivo de configuração que especifica como a interface deve ser feita, não sendo necessário nenhuma alteração no código escrito em C ou C++.

O arquivo que descreve para o SWIG como deve ser feita a interface é o `src/qsystem.i` (D.7.9). Nele, apenas algumas configurações bem simples são feitas, como: encaminhar as exceções do C++ para o Python; mapear o `enum QSystem::Bit` com os valores `NONE`, `ZERO` e `ONE` para, respectivamente, `None`, `0` e `1`, no retorno do método `QSystem::bits`; adicionar as funções `get_matrix` e `set_matrix`, escritas em Python, ao módulo que será criado; e, o mais importante, gerar a interface das classe `QSystem` e `Gates` a partir dos cabeçalhos.

O SWIG é capaz de gerar a interface para a maioria dos métodos. Porém, há dois métodos no qual não é possível. O primeiro caso é o `QSystem::get_qbits` que possui como

retorno uma tupla de tuplas (mais específico, um `((list,list,list),(int, int))`), a geração de tuplas não é suportada pela ferramenta, logo é necessário implementar a interface manualmente, fazendo chamadas diretas a API do Python, como pode ser visto no arquivo `src/qs_utility.cpp` (D.7.8). E o segundo caso é o `Gates::make_fgate` que recebe como argumento uma função e um iterável do Python. Nesse caso não há como traduzir esses dados diretamente para C++, logo, também é necessário fazer chamadas para a API do Python. A implementação desse método pode ser vista no arquivo `src/gates.cpp` (D.7.1). Com exceção desses dois método o código escrito em C++ é totalmente independente do Python.

Para gerar os arquivos de interface com o SWIG basta executar o comando:

```
swig -c++ -python -o src/qsystem.cpp src/qsystem.i
```

Logo em seguida, dois arquivos (`src/qsystem.cpp` e `qsystem.py`) serão criados. O arquivo `.cpp` encapsula todas as chamadas de métodos, construtores e destrutores das classes escritas em C++ em funções, que tem como argumento dois `PyObject*`, sendo o primeiro argumento um objeto Python contendo um ponteiro para instância da classe a qual o método, construtor ou destrutor pertence (como o `self` do Python), e o segundo argumento, guarda todos os argumentos do método que será chamado. Todos argumentos e retornos são objetos Python que são devidamente convertidos para tipos em C++ quando necessário. Já o arquivo `qsystem.py` (que é movido para `qsystem/__init__.py`) possui o que o SWIG chama de *proxy class*, classes que encapsulam as chamadas das funções geradas no arquivo `.cpp` em classes do Python.

12.2 Criação e instalação do pacote

Tendo todos os arquivos necessários para geração do pacote, o mesmo é criado utilizando a ferramenta setuptools[1], um módulo para Python desenvolvido para lidar com pacotes.

O arquivo que contém as configurações de como gerar o pacotes é o `setup.py` (D.5). Nele são especificados os arquivos que devem ser compilados, os parâmetros de compilação, os módulos Python que compõem o pacote e os metadados, sendo, estes: versão, autor, descrição, site e classificadores. Outo arquivo importante é `MANIFEST.in` (D.3). Nele são especificados arquivos adicionais necessários para compilação, como, código da biblioteca Armadilho e cabeçalhos.

Tendo o arquivo `setup.py` configurado, para gerar o pacote, basta executar o comando:

[1] setuptools disponível em <https://pypi.org/project/setuptools/>.

```
python setup.py sdist
```

Este comando criará o arquivo `QSystem-1.0.0.tar.gz` com todas informações necessárias para compilar o código em C++ e instalar o módulo QSystem. A instalação do pacote pode ser feita utilizando a ferramenta pip[2] com o seguinte comando:

```
pip install QSystem-1.0.0.tar.gz
```

A fim de que o módulo QSystem seja independente de sistema operacional e arquitetura, o pacote usado na instalação não contém nenhum arquivo binário. Portanto, durante a instalação é necessário ter um compilador de C e C++17 acessível pelo pip.

[2] pip disponível em <https://pypi.org/project/pip/>.

13 Exemplos de código

Este capítulo visa apresentar dois códigos de exemplo usando o simulador QSystem. O primeiro exemplo (Secção 13.1), apresenta a implementação do algoritmo de fatoração apresentado por Shor(11). E, o segundo exemplo (Secção 13.2), consiste na implementação das etapas de preparação e correção do código estabilizador de 7-qubits apresentado por Steane(21).

13.1 Algoritmo de Shor

O algoritmo de Shor é usado para fatoração não trivial de números inteiros, ou seja, dado um inteiro n, o algoritmo retorna dois números p e q, tal que p e q sejam diferentes de 1 e $pq = n$. Este algoritmo pode ser dividido em três passos, sendo que apenas o segundo passo necessita de um computador quântico. Estes passos são:

1. Selecionar aleatoriamente um número a coprimo[1] a o número n que queremos fatorar.

2. Ache o período r da função $f(x) = a^x \mod n$.

3. Se o período for ímpar, volte ao passo 1, senão, compute os dois fatores p e q tal que

$$p = mdc(a^{\frac{r}{2}} - 1, N) \tag{13.1}$$
$$q = mdc(a^{\frac{r}{2}} + 1, N) \tag{13.2}$$

A prova deste algoritmo requer resultados advindos da teoria dos número, que está fora do escopo deste trabalho mas pode ser vista no artigo de Shor(11).

Com esse algoritmo o problema da fatoração é reduzido ao problema de achar o período de uma função periódica, ou seja, dada uma função periódica $f(x)$, achar o menor número r tal que $f(x) = f(r + x)$. Tal problema pode ser resolvido eficientemente por um computador quântico utilizando o circuito da Figura 29, que efetua a computação nos seguintes passos:

2.1. Inicie dois registradores quânticos, com $s = \lceil \log_2(n + 1) \rceil$ qubits cada, no estado $|0\rangle$.

$$|0\rangle\,|0\rangle \tag{13.3}$$

[1] coprimo ou relativamente primo são dois números a e b que possuem 1 como maior divisor comum, ou seja $mdc(a, b) = 1$.

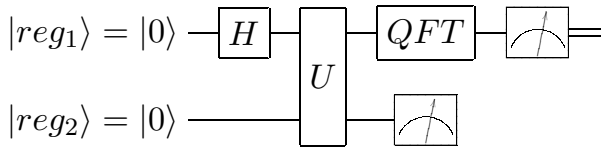


Figura 29 – Circuito quântico utilizado para encontrar o período da função $a^x \mod n$. Os registradores quânticos $|reg_1\rangle$ e $|reg_2\rangle$ devem ser compostos de $\lceil \log_2(n+1) \rceil$ qubits. A porta lógica quântica U mapeá $|x\rangle |0\rangle$ para $|x\rangle |a^x \; mod \, n\rangle$.

2.2. Aplique a porta de Hadamard em todos os qubits do primeiro registrador para gerar uma superposição.

$$|0\rangle |0\rangle \xrightarrow{H^{\otimes s}} \frac{1}{\sqrt{2^s}} \sum_{x=0}^{2^s-1} |x\rangle |0\rangle \tag{13.4}$$

2.3. Aplique o operador $U : |x\rangle |0\rangle \rightarrow |x\rangle |a^x \mod n\rangle$, para gerar uma superposição periódica.

$$\frac{1}{\sqrt{2^s}} \sum_{x=0}^{2^s-1} |x\rangle |0\rangle \xrightarrow{\text{U}} \frac{1}{\sqrt{2^s}} \sum_{x=0}^{2^s-1} |x\rangle |a^x \mod n\rangle \tag{13.5}$$

2.4. Meça e descarte o segundo registrador.

$$\frac{1}{\sqrt{2^s}} \sum_{x=0}^{2^s-1} |x\rangle |a^x \mod n\rangle \rightarrow \sqrt{\frac{r}{2^s}} \sum_{i=0}^{\frac{2^s}{r}-1} |ir + x_0\rangle |a^{x_0} \mod n\rangle \tag{13.6}$$

(Medindo y no segundo registrador, temos, no primeiro registrador, uma superposição com todos os valores de $ir + x_0$, tal que $f(ir + x_0) = y$ e i é um número inteiro positivo)

2.5. Para encontrar o período presente na superposição do primeiro registrador é necessário aplicar uma Transformada de Fourier Quântica (como visto na Subsecção 3.3.2) no mesmo.

$$\sqrt{\frac{r}{2^s}} \sum_{i=0}^{\frac{2^s}{r}-1} |ir + x_0\rangle \xrightarrow{\text{QFT}_n} \frac{1}{\sqrt{r}} \sum_{i=0}^{r-1} \left| i\frac{2^s}{r} \right\rangle e^{\phi_i} \tag{13.7}$$

2.6. Meça o primeiro registrador e repita o processo para encontrar distintos múltiplos de $\frac{2^s}{r}$, então, encontre o maior divisor comum entre todas as medidas.

Utilizando o simulador QSystem, a implementação em Python do algoritmo de Shor é apresentada a seguir:

```
1 from qsystem import QSystem, Gates
2 from random import randint, seed
3 from math import log2, ceil, floor, gcd
```

```
4
5  seed(47) # Inicia o gerador de números pseudo aleatórios
6
7  n = 15 # número que vai ser fatorado
8
9  # 1.
10 a = 0
11 while gcd(n, a) != 1 or a == 1:
12     a = randint(2, n)
13 print('a =', a)
14 # a = 7
15
16 s = ceil(log2(n+1)) # número de qubits por registrador
17 print('s =', s)
18 # s = 4
19
20 # Cria porta logica quântica POWN:|x⟩|0⟩ → |x⟩|a^x mod n⟩
21 gates = Gates()
22 def pown(x):
23     x = x >> s
24     fx = pow(a, x, n)
25     return (x << s) | fx
26 def it():
27     for x in range(2**s):
28         yield x << s
29 gates.make_fgate('POWN', pown, 2*s, it())
30
31 # 2.1.
32 q = QSystem(s, gates, 13)
33 print(q)
34 # +1.000       |0000>
35 #
36
37 # 2.2.
38 q.evol(gate='H', qbit=0, count=s)
39 print(q)
40 # +0.250       |0000>
41 # +0.250       |0001>
42 # +0.250       |0010>
```

```
43 # +0.250       |0011>
44 # +0.250       |0100>
45 # +0.250       |0101>
46 # +0.250       |0110>
47 # +0.250       |0111>
48 # +0.250       |1000>
49 # +0.250       |1001>
50 # +0.250       |1010>
51 # +0.250       |1011>
52 # +0.250       |1100>
53 # +0.250       |1101>
54 # +0.250       |1110>
55 # +0.250       |1111>
56 #
57
58 # 2.3.
59 q.add_ancillas(s)
60 q.evol(gate='POWN', qbit=0)
61 print(q)
62 # +0.250       |0000>|0001>
63 # +0.250       |0001>|0111>
64 # +0.250       |0010>|0100>
65 # +0.250       |0011>|1101>
66 # +0.250       |0100>|0001>
67 # +0.250       |0101>|0111>
68 # +0.250       |0110>|0100>
69 # +0.250       |0111>|1101>
70 # +0.250       |1000>|0001>
71 # +0.250       |1001>|0111>
72 # +0.250       |1010>|0100>
73 # +0.250       |1011>|1101>
74 # +0.250       |1100>|0001>
75 # +0.250       |1101>|0111>
76 # +0.250       |1110>|0100>
77 # +0.250       |1111>|1101>
78 #
79
80 # 2.4.
81 q.measure(qbit=s, count=s)
```

```
82  q.rm_ancillas()
83  print(q)
84  # +0.500        |0011>
85  # +0.500        |0111>
86  # +0.500        |1011>
87  # +0.500        |1111>
88  #
89
90  # 2.5.
91  q.qft(qbegin=0, qend=s)
92  print(q)
93  # +0.500        |0000>
94  #       -0.500i|0100>
95  # -0.500        |1000>
96  #       +0.500i|1100>
97  #
98
99  # 2.6.
100 q.measure_all()
101 c = q.bits()
102 c = sum([m*2**i for m, i in zip(c, reversed(range(len(c))))])
103 measurements = [c]
104 for _ in range(s-1):
105     seed = randint(210,760)
106     q = QSystem(s, gates, seed)
107     q.evol('H', 0, s)
108     q.add_ancillas(s)
109     q.evol('POWN', 0)
110     q.rm_ancillas()
111     q.qft(0, s)
112     q.measure_all()
113     c = q.bits()
114     c = sum([m*2**i for m, i in zip(c, reversed(range(len(c))))])
115     measurements.append(c)
116 print('medidas =', measurements)
117 # medidas = [4, 12, 4, 12]
118 c = measurements[0]
119 for m in measurements:
120     c = gcd(c, m)
```

```
121 if c == 0:
122     print('medidas insuficientes')
123 else:
124     r = 2**s/c
125     print('possível período r =', r)
126 # possível período r = 4.0
127
128 # 3.
129     if r % 2 == 1:
130         print('repita o algoritmo')
131     else:
132         p = gcd(int(a**(r/2)+1), n)
133         q = gcd(int(a**(r/2)-1), n)
134         print(p, '*', q, '=', n)
135 # 5 * 3 = 15
```

13.2 Código estabilizador

Códigos de correção de erros quânticos encapsulam vários qubits físicos em alguns qubits lógicos, onde esses qubits lógicos são capazes de suportar tipos e quantidades específicas de erros quânticos sem que o estado lógico do qubit mude. Por sua vez, códigos estabilizadores são códigos de correção de erros quânticos que podem ser totalmente definidos por seus estabilizadores. Para mais informações sobre códigos de correção de erros e códigos estabilizadores veja o Apêndice B.

O código estabilizador de Steane(21), também conhecido como código de 7-qubits, é definido pelos estabilizadores (operadores)

$$K^1 = IIIXXXX, \qquad K^2 = XIXIXIX, \qquad K^3 = IXXIIXX, \tag{13.8}$$

$$K^4 = IIIZZZZ, \qquad K^5 = ZIZIZIZ \text{ e} \qquad K^6 = IZZIIZZ. \tag{13.9}$$

Para que um conjunto de qubits pertença ao código, o mesmo precisa estar no estado

$$|0\rangle_L = \frac{1}{\sqrt{8}}\left(\begin{array}{l}|0000000\rangle + |0001111\rangle + |0110011\rangle + |0111100\rangle + \\ |1010101\rangle + |1011010\rangle + |1100110\rangle + |1101001\rangle\end{array}\right), \tag{13.10}$$

$$|1\rangle_L = \frac{1}{\sqrt{8}}\left(\begin{array}{l}|1111111\rangle + |1110000\rangle + |1001100\rangle + |1000011\rangle + \\ |0101010\rangle + |0100101\rangle + |0011001\rangle + |0010110\rangle\end{array}\right) \tag{13.11}$$

ou em uma superposição de $|0\rangle_L$ e $|1\rangle_L$, porém, caso aconteça no máximo um erro de *bit flip* e um erro de *phase flip*, é possível utilizar os estabilizadores para efetuar uma medida de síndrome para identificar e posteriormente corrigir o erro.

A seguir, será apresentado a implementação em Python utilizando o módulo QSystem dos circuitos de preparação (Figura 37) e correção de erros (Figura 38) do código de Steane apresentados em 22.

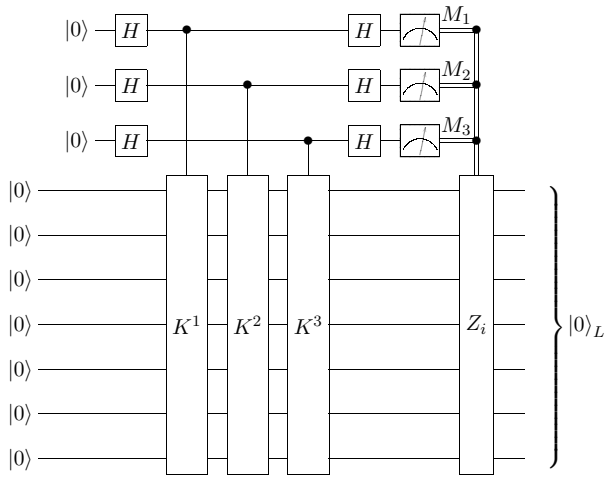


Figura 30 – Circuito para preparar $|0\rangle^{\otimes 7}$ em $|0\rangle_L$ do código de 7-qubits, após a medida das ancilas é aplicado uma porta Z no qubit $i = 4M_1 + M_2 + 2M_3 - 1$.

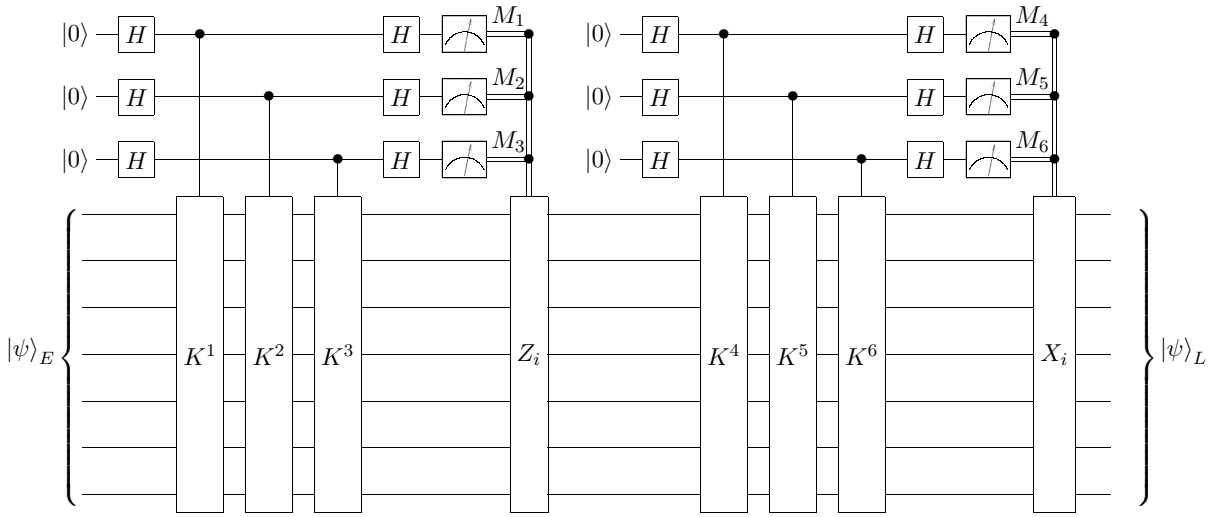


Figura 31 – A medida de síndrome para código de Steane é dividida em duas partes, onde, a primeira, identifica e corrige *phase flip* aplicando uma porta Z no qubit $i = 4M_1 + M_2 + 2M_3 - 1$ e, a segunda, identifica e corrige *bit flip* aplicando uma porta X no qubit $i = 4M_4 + M_5 + 2M_6 - 1$.

```
1 from qsystem import QSystem, Gates
2 from random import randint, random, sample, seed
3
4 gate = Gates()
5 gate.make_cgate('ck1', 'IIIXXXXI', [7]) # Portas dos
6 gate.make_cgate('ck2', 'XIXIXIXI', [7]) # estabilizadores
```

```
gate.make_cgate('ck3', 'IXXIIXXI', [7]) # do código
gate.make_cgate('ck4', 'IIIZZZZI', [7]) # controladas por
gate.make_cgate('ck5', 'ZIZIZIZI', [7]) # um qubit ancilar
gate.make_cgate('ck6', 'IZZIIZZI', [7]) #

def prepare(q, cks, gate): # Circuito genérico para aplicar estabilizadores
    an_bits = []
    for ck in cks:
        q.add_ancillas(1)
        q.evol('H', 7)
        q.evol(ck, 0)
        q.evol('H', 7)
        q.measure(7)
        an_bits.append(q.bits()[7])
        q.rm_ancillas()
    i = sum([x[0]*x[1] for x in zip(an_bits, [4, 1, 2])])-1
    if i != -1:
        q.evol(gate, i)
    return an_bits

def prepare_z(q): # Corrige phase flip ou prepara o estado logico
    return prepare(q, ['ck1', 'ck2', 'ck3'], 'Z')

def prepare_x(q): # Corrige bit fip
    return prepare(q, ['ck4', 'ck5', 'ck6'], 'X')

seed(42)
for i in range(10):
    q = QSystem(7, gate, i, 'matrix')
    pre_bits = prepare_z(q) # Prepara no estado lógico |0>
#             ┌phase flip┐  ┌ bit flip ┐
    qerror = [randint(0,6), randint(0,6)] # qubit onde acontece o erro
    p =      [random(),     random()]     # p do erro
    q.flip('Z', qerror[0], p[0]) # Aplica um erro de phase flip
    q.flip('X', qerror[1], p[1]) # Aplica um erro de bit flip
    z_bits = prepare_z(q) # Corrige phase flip
    x_bits = prepare_x(q) # Corrige bit flip
    q.measure_all()
    print(i, pre_bits, z_bits, x_bits, q.bits(), sum(q.bits())%2)
```

```
46      print('phase flip no qubit', qerror[0] , 'com p =', p[0],
47      '(erro não detectado)' if z_bits == [0, 0, 0] else '(erro detectado)')
48      print('bit flip on qbit', qerror[1] , 'with p =', p[1],
49      '(erro não detectado)' if x_bits == [0, 0, 0] else '(erro detectado)')
50  # 0 [1, 0, 1] [0, 0, 0] [0, 1, 0] [1, 0, 1, 1, 0, 1, 0] 0
51  # phase flip no qubit 5 com p = 0.025010755222666936 (erro não detectado)
52  # bit flip on qbit 0 with p = 0.27502931836911926 (erro detectado)
53  # 1 [1, 0, 1] [0, 0, 0] [0, 0, 0] [1, 0, 1, 1, 0, 1, 0] 0
54  # phase flip no qubit 1 com p = 0.7364712141640124 (erro não detectado)
55  # bit flip on qbit 1 with p = 0.6766994874229113 (erro não detectado)
56  # 2 [1, 1, 0] [0, 0, 0] [0, 0, 0] [1, 1, 0, 0, 1, 1, 0] 0
57  # phase flip no qubit 4 com p = 0.5904925124490397 (erro não detectado)
58  # bit flip on qbit 0 with p = 0.03178267948178359 (erro não detectado)
59  # 3 [1, 0, 0] [0, 0, 0] [0, 0, 1] [0, 0, 0, 0, 0, 0, 0] 0
60  # phase flip no qubit 0 com p = 0.2326608933907396 (erro não detectado)
61  # bit flip on qbit 1 with p = 0.6020187290499803 (erro detectado)
62  # 4 [1, 0, 0] [0, 0, 0] [0, 0, 0] [0, 1, 1, 0, 0, 1, 1] 0
63  # phase flip no qubit 4 com p = 0.7160196129224035 (erro não detectado)
64  # bit flip on qbit 1 with p = 0.7013249735902359 (erro não detectado)
65  # 5 [0, 0, 1] [0, 0, 0] [0, 0, 0] [1, 0, 1, 0, 1, 0, 1] 0
66  # phase flip no qubit 3 com p = 0.4492090462838536 (erro não detectado)
67  # bit flip on qbit 1 with p = 0.2781907082306627 (erro não detectado)
68  # 6 [0, 1, 1] [1, 1, 1] [0, 0, 0] [0, 0, 0, 1, 1, 1, 1] 0
69  # phase flip no qubit 6 com p = 0.7588073671297673 (erro detectado)
70  # bit flip on qbit 0 with p = 0.15965931637689013 (erro não detectado)
71  # 7 [0, 1, 1] [0, 0, 0] [0, 0, 0] [0, 1, 1, 0, 0, 1, 1] 0
72  # phase flip no qubit 3 com p = 0.27787134167164185 (erro não detectado)
73  # bit flip on qbit 2 with p = 0.2153137621075888 (erro não detectado)
74  # 8 [0, 1, 1] [0, 0, 0] [0, 1, 1] [1, 1, 0, 0, 1, 1, 0] 0
75  # phase flip no qubit 6 com p = 0.10221027651984871 (erro não detectado)
76  # bit flip on qbit 2 with p = 0.3799273006373374 (erro detectado)
77  # 9 [0, 1, 0] [0, 1, 1] [0, 0, 0] [1, 1, 0, 1, 0, 0, 1] 0
78  # phase flip no qubit 2 com p = 0.3439557224789711 (erro detectado)
79  # bit flip on qbit 6 with p = 0.26452086722201307 (erro não detectado)
```

Conclusão

Neste trabalho foi desenvolvido um simulador de circuito quântico para Python. O simulador, denominado QSystem, foi implementado como um modulo Python e está disponível no *Python Package Index*, onde pode ser instalado usando a ferramenta pip.

Utilizando a biblioteca Armadillo para C++, o simulador obteve uma boa performance mantendo um código legível e de fácil manutenção. E, usando a ferramenta SWIG, o código escrito em C++ pode ser integrado com o interpretador do Python sem sofrer muitas alterações.

O simulador foi desenvolvido de forma que grande parte da álgebra linear envolvida em uma computação quântica possa ser abstraída pelo usuário. Com ele, é possível instanciar sistemas quânticos com um número arbitrário de qubits, aplicar diversas portas lógicas e simular sistemas tanto em vetor de estado quanto em matriz densidade, que permite a aplicação de erros quânticos.

Como simular um computador quântico toma tempo e espaço exponencial ao número de qubits, estamos limitados a simulação de poucos deles. Em testes, utilizando um computador com as configurações da Tabela 1, foi possível simular até 27 qubits com pouca superposição; e, 14 qubits em superposição total, sendo possível simular um número maior dependendo do circuito. O número total de qubits simulados pode variar dependendo da arquitetura do processador, da versão do compilador e do sistema operacional usado. Esses valores são validos tanto para vetor de estado quanto para matriz densidade. Contudo, a simulação em matriz densidade toma tempo exponencial em relação a simulação em vetor de estado.

Utilizando o QSystem é possível traduzir um circuito quântico para o Python de maneira fácil e direta. Assim, o simulador se torna uma ferramenta dinâmica e de fácil uso para quem quer estudar algoritmos, protocolos e códigos quânticos, dando ênfase na possibilidade de simular códigos identificação e correção de erros quânticos devido ao fato de poder simular erros em estado misto.

CPU	Intel® Core™ i7-4500U
RAM	16 GB
Python	3.7.3
GCC	9.1.0
Linux	Kernel 4.19

Tabela 1 – Configuração do computador usado nos testes.

Trabalhos futuros

***multithreading*:** Multiplicação matricial é a operação com maior custo computacional do simulador e ela tem alto potencial de paralelismo. Porém, o mesmo não é explorado no simulador. A biblioteca Armadillo utiliza *multithreading* apenas para multiplicação de matrizes não esparsas. Porém o desempenho advindo do uso de matrizes esparsas supera o desempenho de matrizes não esparsas, mesmo utilizando *multithreading*.

Matrizes esparsas e não esparsas: Grande parte das matrizes das portas lógicas quânticas são esparsas, o que justifica utilizar matrizes esparsas para representá-las, porém, as matrizes de Hadamard e da Transformada de Fourier Quântica não possuem nenhum elemento não nulo, assim, essas matrizes degradam o desempenho do simulador. Da mesma forma, estados quânticos com uma superposição grande degradam o desempenho do simulador. Logo, é interessante o desenvolvimento de uma classe de matriz que mude dinamicamente entre matriz esparsa e matriz densa, tal implementação aumentaria o desempenho do simulador.

Integração das classes do simulador: A funcionalidade e a necessidade da classe `Gates` pode ser confusa inicialmente, por isso, a integração entre essa classe e a classe `QSystem` pode ser revista. Uma possível melhoria na usabilidade do simulador seria mover a classe `Gates` para dentro da classe `QSystem` e, criar uma nova classe para armazenar uma porta lógica criada pelo usuário. Assim, os objetos desta classe seriam passados como argumento para um método da classe `QSystem` que aplicaria o mesmo no sistema quântico.

Integração com o QuTiP: O QuTip(12) é um pacote Python com diversas ferramentas para a simulação da evolução temporal de sistemas quânticos, que é muito utilizado pela comunidade cientifica . Assim, é interessante estudar a possibilidade de integrar o código desenvolvido com o QuTiP.

Referências

1 HENNESSY, J.; PATTERSON, D. *Computer Architecture: A Quantitative Approach*. Elsevier Science, 2017. (The Morgan Kaufmann Series in Computer Architecture and Design). ISBN 9780128119068. Disponível em: <https://books.google.com.br/books?id=cM8mDwAAQBAJ>. Citado na página 13.

2 MOORE, G. E. et al. *Cramming more components onto integrated circuits*. [S.l.]: McGraw-Hill New York, NY, USA:, 1965. Citado na página 13.

3 DENNARD, R. H. et al. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, IEEE, v. 9, n. 5, p. 256–268, 1974. Citado na página 13.

4 AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: ACM. *Proceedings of the April 18-20, 1967, spring joint computer conference*. [S.l.], 1967. p. 483–485. Citado na página 13.

5 FEYNMAN, R. P. Simulating physics with computers. *International journal of theoretical physics*, Springer, v. 21, n. 6, p. 467–488, 1982. Citado na página 13.

6 SHOR, P. W. Algorithms for quantum computation: Discrete logarithms and factoring. In: IEEE. *Proceedings 35th annual symposium on foundations of computer science*. [S.l.], 1994. p. 124–134. Citado na página 13.

7 SANDERSON, C.; CURTIN, R. Armadillo: a template-based c++ library for linear algebra. *Journal of Open Source Software*, v. 1, n. 2, p. 26–32, 2016. Citado 2 vezes nas páginas 13 e 49.

8 BEAZLEY, D. M. et al. Swig: An easy to use tool for integrating scripting languages with c and c++. In: *Tcl/Tk Workshop*. [S.l.: s.n.], 1996. p. 43. Citado 2 vezes nas páginas 13 e 49.

9 NIELSEN, M. A.; CHUANG, I. *Quantum computation and quantum information*. Cambridge, UK: Cambridge University Press, 2010. 733 p. ISBN 9781107002173. Citado 3 vezes nas páginas 14, 19 e 34.

10 SPIŠIAK, M.; KOLLÁR, J. Quantum programming: A review. In: IEEE. *Informatics, 2017 IEEE 14th International Scientific Conference on*. [S.l.], 2017. p. 353–358. Citado na página 32.

11 SHOR, P. W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, SIAM, v. 41, n. 2, p. 303–332, 1999. Citado 2 vezes nas páginas 33 e 87.

12 JOHANSSON, J. R.; NATION, P.; NORI, F. Qutip: An open-source python framework for the dynamics of open quantum systems. *Computer Physics Communications*, Elsevier, v. 183, n. 8, p. 1760–1772, 2012. Citado 2 vezes nas páginas 49 e 99.

13 ALEKSANDROWICZ, G. et al. *Qiskit: An Open-source Framework for Quantum Computing*. 2019. Citado na página 49.

14 EATON, J. W. et al. *GNU Octave version 5.1.0 manual: a high-level interactive language for numerical computations*. [S.l.], 2019. Disponível em: <https://www.gnu.org/software/octave/doc/v5.1.0/>. Citado na página 49.

15 SANDERSON, C.; CURTIN, R. A user-friendly hybrid sparse matrix class in c++. In: SPRINGER. *International Congress on Mathematical Software*. [S.l.], 2018. p. 422–430. Citado na página 50.

16 WATT, D. *Programming Language Design Concepts*. Wiley, 2006. ISBN 9780470020470. Disponível em: <https://books.google.com.br/books?id=vogP3P2L4tgC>. Citado na página 51.

17 NAUMANN, A. *Variant: a type-safe union for C++17 (v8)*. [S.l.], 2016. Disponível em: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0088r3.html>. Citado na página 52.

18 HESTON, K.; PAZ, A. *Pauli Measurements*. 2017. Disponível em: <https://docs.microsoft.com/en-us/quantum/concepts/pauli-measurements>. Acesso em: 30 mar. 2019. Citado na página 63.

19 HU, M.-L.; FAN, H. Quantum coherence of multiqubit states in correlated noisy channels. *arXiv preprint arXiv:1812.04385*, 2018. Citado na página 70.

20 JONES, E. et al. *SciPy: Open source scientific tools for Python*. [S.l.], 2001–2019. Disponível em: <http://www.scipy.org/>. Citado na página 80.

21 STEANE, A. M. Error correcting codes in quantum theory. *Physical Review Letters*, APS, v. 77, n. 5, p. 793, 1996. Citado 2 vezes nas páginas 87 e 92.

22 DEVITT, S. J.; MUNRO, W. J.; NEMOTO, K. Quantum error correction for beginners. *Reports on Progress in Physics*, IOP Publishing, v. 76, n. 7, p. 076001, 2013. Citado na página 93.

23 TOUCHETTE, D.; ALI, H.; HILKE, M. 5-qubit quantum error correction in a charge qubit quantum computer. *arXiv preprint arXiv:1010.3242*, 2010. Citado na página 114.

Apêndices

APÊNDICE A – Álgebra linear com notação de Dirac

A álgebra linear é uma das principais ferramentas da computação quântica. Portanto, sua compreendimento é fundamental. Neste apêndice é feita uma rápida revisão de álgebra linear utilizando a notação de Dirac, também conhecida como notação *braket*.

A notação de Dirac é notação padrão usada na mecânica quântica e, é muito conveniente quando se trabalha com espaços vetoriais complexos, como é o caso da computação quântica.

Notação	Descrição
z^*	Conjugado de $z \in \mathbb{C}$
A^T	Transposto matriz A
A^*	Conjugado matriz A
$A^\dagger$	Transposto Hermitiano da matriz A, $A^\dagger = (A^T)^*$
$\|\psi\rangle$	Vetor coluna chamado *ket*
$\langle\psi\|$	Vetor dual de $\|\psi\rangle$ chamado *bra*, $\langle\psi\| = \|\psi\rangle^\dagger$
$\langle\varphi\|\psi\rangle$	Produto interno entre $\|\varphi\rangle$ e $\|\psi\rangle$
$\|\varphi\rangle \otimes \|\psi\rangle$	Produto tensorial entra $\|\varphi\rangle$ e $\|\psi\rangle$
$\|\varphi\rangle \|\psi\rangle$	Produto tensorial entra $\|\varphi\rangle$ e $\|\psi\rangle$
$\|\varphi\psi\rangle$	Produto tensorial entra $\|\varphi\rangle$ e $\|\psi\rangle$

Tabela 2 – Resumo da notação de Dirac

A.1 Números complexos

Na computação quântica, representamos um qubit como um vetor pertencente a um espaço complexo, ou seja, um espaço vetorial sobre o corpo $\mathbb{C}$.

Um número $z \in \mathbb{C}$ é dividido em duas partes, $z = a + bi$, a parte real a e a parte imaginária b que é multiplicado pela constante $i = \sqrt{-1}$, onde $i^2 = -1$.

Sendo $z, w \in \mathbb{C}$ e $z = a + bi$, $w = c + di$, define-se:

Identidade

$$z = w \Leftrightarrow a = c \wedge b = d; \tag{A.1}$$

Soma

$$z + w = (a + c) + (b + d)i; \tag{A.2}$$

Produto

$$zw = (a + bi)(c + di) = (ac - bd) + (bc + ad)i; \tag{A.3}$$

Conjugado

$$z^* = a - bi; \tag{A.4}$$

Módulo

$$|z| = \sqrt{a^2 + b^2}. \tag{A.5}$$

Um número complexo pode ser descrito na forma de Euler como

$$re^{i\theta} = r\cos\theta + r\sin\theta i. \tag{A.6}$$

A.2 Base e independência linear

Uma base é um conjunto de vetores linearmente independentes que geram o um espaço vetorial.

Um exemplo de base é a base computacional composto pelos vetores

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ e } |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{A.7}$$

Todo vetor pertencente ao espaço pode ser gerado pela combinação linear dos vetores da base. Assim sendo, $|0\rangle, |1\rangle, \ldots, |n\rangle$ uma base e $\alpha_0, \alpha_1, \ldots, \alpha_n \in \mathbb{C}$, um vetor $|\psi\rangle$ qualquer pode ser descrito como

$$|\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle + \cdots + \alpha_n |n\rangle \tag{A.8}$$

Um conjunto de vetores são linearmente independentes se (sendo 0 o vetor nulo)

$$0 = \alpha_0 |0\rangle + \alpha_1 |1\rangle + \cdots + \alpha_n |n\rangle \Leftrightarrow \sum_i \alpha_i = 0 \tag{A.9}$$

A.3 Produto interno

O produto interno é uma operação entre dois vetores que retorna um escalar, sendo representada na notação de Dirac com um $\langle bra|ket\rangle$

O produto escalar entre os vetores $|\psi\rangle$ e $|\varphi\rangle$ possui as seguintes propriedades:

Associatividade

$$\alpha \langle\varphi|\psi\rangle = (\alpha |\varphi\rangle, |\psi\rangle), \quad \alpha \in \mathbb{C}; \tag{A.10}$$

Simetria hermitiana

$$\langle\varphi|\psi\rangle = \langle\psi|\varphi\rangle^* ; \tag{A.11}$$

Positividade

$$\langle\psi|\psi\rangle > 0, \quad \forall\, |\psi\rangle \neq 0 \tag{A.12}$$

$$\langle\psi|\psi\rangle = 0 \Leftrightarrow |\psi\rangle = 0; \tag{A.13}$$

Distributividade

$$(\langle\upsilon| + \langle\varphi|)\,|\psi\rangle = \langle\upsilon|\psi\rangle + \langle\varphi|\psi\rangle \tag{A.14}$$

Dois vetores $|\psi\rangle$ e $|\varphi\rangle$ são linearmente independentes se e somente se

$$\langle\psi|\varphi\rangle = 0. \tag{A.15}$$

O produto interno pode ser definido de varias formas, porém, será usada a seguinte definição para espaços complexos $\mathbb{C}^n$: seja $|\varphi\rangle = (\nu_1, \nu_2, \ldots, \nu_n)$ e $|\psi\rangle = (\upsilon_1, \upsilon_2, \ldots, \upsilon_n)$

$$\langle\varphi|\psi\rangle = \begin{bmatrix}\nu_1^* & \nu_2^* & \ldots & \nu_n^*\end{bmatrix} \begin{bmatrix}\upsilon_1 \\ \upsilon_2 \\ \vdots \\ \upsilon_n\end{bmatrix} = \sum_i \nu_i^* \upsilon_i. \tag{A.16}$$

A norma de um vetor $|\psi\rangle$ é dada como

$$\|\,|\psi\rangle\,\| = \sqrt{\langle\psi|\psi\rangle}. \tag{A.17}$$

A.4 Produto externo

O produto externo é uma operação entre dois vetores, $|\varphi\rangle = (\upsilon_1, \upsilon_2, \ldots, \upsilon_n)$ e $|\psi\rangle = (\nu_1, \nu_2, \ldots, \nu_n)$, que resulta em uma matriz, definida como

$$|\varphi\rangle\langle\psi| = \begin{bmatrix}\upsilon_1 \\ \upsilon_2 \\ \vdots \\ \upsilon_n\end{bmatrix} \begin{bmatrix}\nu_1^* & \nu_2^* & \ldots & \nu_n^*\end{bmatrix} = \begin{bmatrix}\upsilon_1\nu_1^* & \upsilon_1\nu_2^* & \ldots & \upsilon_1\nu_n^* \\ \upsilon_2\nu_1^* & \upsilon_2\nu_2^* & \ldots & \upsilon_2\nu_n^* \\ \vdots & \vdots & \ddots & \vdots \\ \upsilon_n\nu_1^* & \upsilon_n\nu_2^* & \ldots & \upsilon_n\nu_n^*\end{bmatrix}. \tag{A.18}$$

A representação em produto externo de matizes pode ser conveniente na multiplicação delas, como por exemplo, a multiplicação das matrizes de Pauli X e Z (vista na Subsecção 2.2.2)

$$XZ = (|1\rangle\langle 0| + |0\rangle\langle 1|)(|0\rangle\langle 0| - |1\rangle\langle 1|) \tag{A.19}$$

$$XZ = |1\rangle\langle 0|0\rangle\langle 0| - |1\rangle\langle 0|1\rangle\langle 1| + |0\rangle\langle 1|0\rangle\langle 0| - |0\rangle\langle 1|1\rangle\langle 1| \tag{A.20}$$

$$XZ = |1\rangle\langle 0| - |0\rangle\langle 1|\,. \tag{A.21}$$

A.5 Operação traço

A operação traço é definida com sendo o somatório dos elementos da diagonal de uma matriz, sendo o traço da matriz A igual a

$$\mathrm{tr}(A) = \sum_i A_{ii}. \tag{A.22}$$

E o traço da matriz $|\varphi_0\rangle\langle\psi_1| + \cdots + |\varphi_n\rangle\langle\psi_n|$ igual a

$$\mathrm{tr}(|\varphi_0\rangle\langle\psi_1| + \cdots + |\varphi_n\rangle\langle\psi_n|) = \langle\psi_0|\varphi_0\rangle + \cdots + \langle\psi_n|\varphi_n\rangle\,. \tag{A.23}$$

Sendo A e B matrizes, o traço possua as seguintes propriedades:

Cíclico

$$\mathrm{tr}(AB) = \mathrm{tr}(BA) \tag{A.24}$$

Linearidade

$$\mathrm{tr}(A+B) = \mathrm{tr}(A) + \mathrm{tr}(B) \tag{A.25}$$

Associatividade

$$z\,\mathrm{tr}(A) = \mathrm{tr}(zA), \quad z \in \mathbb{C} \tag{A.26}$$

APÊNDICE B – Correção de erros quânticos

Neste apêndice veremos técnicas para identificar e corrigir erros em um sistema quântico. Em linhas gerais, para construir um qubit tolerante a erros, codificamos k qubits lógicos em n qubits físicos, com uma distância d entre cada estado lógico. Com isso, conseguimos corrigir $\frac{d-1}{2}$ erros. Podemos denotar cada código de correção de erros a partir de suas características como $[[n, k, d]]$.

Primeiramente, o código de repetição $[[3, 1, 3]]$ é apresentado como uma introdução a correção de erros (Secção B.1). Em seguida, é apresentado o código de Shor $[[9, 1, 3]]$ (Secção B.2), que consegue corrigir até dois erros. Por fim, vemos o formalismo do código estabilizador (Secção B.3), exemplificado nos códigos de 7-qubits $[[7, 1, 3]]$.

B.1 Código de repetição

Se queremos proteger um bit clássico contra *bit flip* a técnica mais simples que podemos usar é repeti-lo em três bits, assim, se um bit diferir dos demais sabemos que houve um *bit flip* no mesmo. Desta forma, codificamos o bit 0 em 000 e 1 em 111, e se em algum momento estivermos em um estado diferente de 000 ou 111, como por exemplo, 101 saberemos que aconteceu um *Bit Flip* em algum bit. Podemos extrapolar esse conceito para qubits, codificando o estado $|0\rangle$ em $|000\rangle$ e $|1\rangle$ em $|111\rangle$. Assim, se o qubit não estiver em algum desses estados ou em uma superposição dele sabemos que houve a atuação de um canal de *bit flip*.

O código de *bit flip* é composto pelas palavras

$$|0\rangle_L = |000\rangle \tag{B.1}$$

$$|1\rangle_L = |111\rangle\,, \tag{B.2}$$

chamadas de estados lógicos do qubit, e pela superposição de $|0\rangle_L$ e $|1\rangle_L$, sendo que, qualquer palavra não pertencente ao código, é um erro.

Podemos codificar um qubit $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$ no código de *bit flip* conforme o circuito da Figura 32, pelos seguintes passo:

$$|\psi\rangle\,|00\rangle = \alpha\,|000\rangle + \beta\,|100\rangle \tag{B.3}$$

$$[cnot_{0,1}]\,|\psi\rangle\,|00\rangle = \alpha\,|000\rangle + \beta\,|110\rangle \tag{B.4}$$

$$[cnot_{0,2}][cnot_{0,1}]\,|\psi\rangle\,|00\rangle = \alpha\,|000\rangle + \beta\,|111\rangle = |\psi\rangle_L \tag{B.5}$$

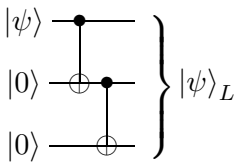


Figura 32 – Preparação do código repetição para *bit flip*.

Para detectar e corrigir um erro adicionamos dois qubits ancilares, como na Figura 33, e os medimos colapsando o sistema conforme a Tabela 3. Chamamos os resultados das medidas das ancilas de síndrome do erro e é com ela que conseguimos identificar o erro para podermos corrigi-lo, a Tabela 4 apresenta as correções necessárias para cada medida de síndrome.

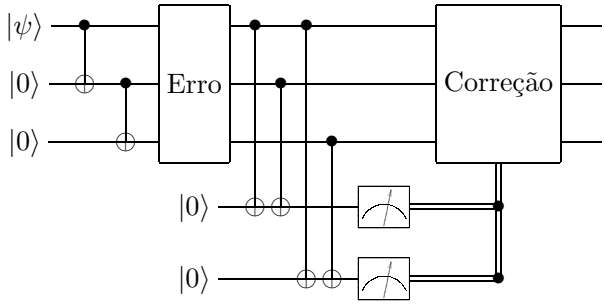


Figura 33 – Código de correção de *bit flip*.

Medida	Estado
00	$\alpha \|000\rangle \|00\rangle + \beta \|111\rangle \|00\rangle$
01	$\alpha \|001\rangle \|01\rangle + \beta \|110\rangle \|01\rangle$
10	$\alpha \|010\rangle \|10\rangle + \beta \|101\rangle \|10\rangle$
11	$\alpha \|100\rangle \|10\rangle + \beta \|011\rangle \|11\rangle$

Tabela 3 – Medida da síndrome para o código de *bit flip*

Erro	Síndrome	Correção
Sem erro	00	
qubit 2	01	X_2
qubit 1	10	X_1
qubit 0	11	X_0

Tabela 4 – Medida da síndrome e correção para o código de *Bit Flip*

Para exemplificar o protocolo de correção, imagine que temos um qubit no estado

$$|\psi\rangle_L = \alpha |0\rangle_L + \beta |1\rangle_L \tag{B.6}$$

que sofreu um *bit flip* no segundo qubit, passando para o estado

$$|\psi\rangle_E = \alpha\,|010\rangle + \beta\,|101\rangle\,. \tag{B.7}$$

Para fazer a identificação e correção adicionamos duas ancilas no estado $|00\rangle$

$$|\psi\rangle_E\,|0_a0_b\rangle = \alpha\,|010\rangle\,|0_a0_b\rangle + \beta\,|101\rangle\,|0_a0_b\rangle \tag{B.8}$$

e aplicamos as *cnot*s conforme a Figura 33

$$[cnot_{0,a}]\,|\psi\rangle_E\,|0_a0_b\rangle = \alpha\,|010\rangle\,|0_a0_b\rangle + \beta\,|101\rangle\,|1_a0_b\rangle \tag{B.9}$$

$$[cnot_{1,a}][cnot_{0,a}]\,|\psi\rangle_E\,|0_a0_b\rangle = \alpha\,|010\rangle\,|1_a0_b\rangle + \beta\,|101\rangle\,|1_a0_b\rangle \tag{B.10}$$

$$[cnot_{0,b}][cnot_{1,a}][cnot_{0,a}]\,|\psi\rangle_E\,|0_a0_b\rangle = \alpha\,|010\rangle\,|1_a0_b\rangle + \beta\,|101\rangle\,|1_a1_b\rangle \tag{B.11}$$

$$[cnot_{2,b}][cnot_{0,b}][cnot_{1,a}][cnot_{0,a}]\,|\psi\rangle_E\,|0_a0_b\rangle = \alpha\,|010\rangle\,|1_a0_b\rangle + \beta\,|101\rangle\,|1_a0_b\rangle\,, \tag{B.12}$$

em seguida, medimos os qubits ancilares, que resultam em 10 e colapsam o sistema em

$$\alpha\,|010\rangle + \beta\,|101\rangle\,, \tag{B.13}$$

e com os resultados das medidas aplicamos a correção conforme a Tabela 4, voltando para o estado sem erro

$$X_1(\alpha\,|010\rangle + \beta\,|101\rangle) = \alpha\,|000\rangle + \beta\,|111\rangle \tag{B.14}$$

$$= \alpha\,|0\rangle_L + \beta\,|1\rangle_L\,. \tag{B.15}$$

O código de *bit flip* é $[[3, 1, d = 3]]$, onde d é a distância entre as palavras $|0\rangle_L$ e $|1\rangle_L$, isso significa que é necessário alterar 3 qubits para levar $|0\rangle_L$ em $|1\rangle_L$ e vice versa. Outra característica é que esse código só corrige $\frac{d-1}{2} = 1$ erro. Mais do que um erro o código é incapaz de corrigir. Se acontecem dois *bit flip* em qubits distintos no estado $|0\rangle_L$, esse estado passa a ficar mais perto do estado $|1\rangle_L$ do que em $|0\rangle_L$. Assim, o código irá corrigir erroneamente para o estado $|1\rangle_L$. No caso de acontecer um *Bit Flip* em cada qubit o código não será capaz de identificar o erro. Exemplos podem ser vistos na Tabela 5.

bit flip nos qubits	Estado com erro	Resultado da correção	Comentário
0	$\alpha\,\|100\rangle + \beta\,\|011\rangle$	$\alpha\,\|0\rangle_L + \beta\,\|1\rangle_L$	Erro identificado e corrigido.
1	$\alpha\,\|010\rangle + \beta\,\|101\rangle$	$\alpha\,\|0\rangle_L + \beta\,\|1\rangle_L$	Erro identificado e corrigido.
2	$\alpha\,\|001\rangle + \beta\,\|110\rangle$	$\alpha\,\|0\rangle_L + \beta\,\|1\rangle_L$	Erro identificado e corrigido.
0 e 1	$\alpha\,\|110\rangle + \beta\,\|001\rangle$	$\alpha\,\|1\rangle_L + \beta\,\|0\rangle_L$	Erro incorretamente identificado.
0 e 2	$\alpha\,\|101\rangle + \beta\,\|010\rangle$	$\alpha\,\|1\rangle_L + \beta\,\|0\rangle_L$	Erro incorretamente identificado.
1 e 2	$\alpha\,\|011\rangle + \beta\,\|100\rangle$	$\alpha\,\|1\rangle_L + \beta\,\|0\rangle_L$	Erro incorretamente identificado.
0,1 e 2	$\alpha\,\|111\rangle + \beta\,\|000\rangle$		Presença de erro não identificado

Tabela 5 – Erros e correções para o estado $\alpha\,|0\rangle_L + \beta\,|1\rangle_L$ codificado no código de *bit flip*.

Podemos utilizar o mesmo conceito para construir um código tolerante a um *phase flip*, sendo as palavras deste código

$$|0\rangle_L = \frac{|000\rangle + |111\rangle}{\sqrt{2}} = |+++\rangle \tag{B.16}$$

$$|1\rangle_L = \frac{|000\rangle - |111\rangle}{\sqrt{2}} = |---\rangle\,. \tag{B.17}$$

Utilizando a base $\{|+\rangle, |-\rangle\}$ visto na equação 2.22, um *phase flip* leva um qubit do estado $|+\rangle$ para o estado $|-\rangle$ e vice versa.

Por exemplo, se acontecer um *phase flip* no primeiro qubit do estado $|+++\rangle$, ele passa a estar no estado $|-++\rangle$. Assim, para corrigir esse erro passamos todos os qubits por uma porta de Hadamard $HHH\,|-++\rangle = |100\rangle$, e utilizamos o mesmo protocolo visto para o código de *Bit Flip* para corrigir o erro. Todo protocolo de codificação e correção de erro do código de *phase flip* pode ser visto no circuito de Figura 34.

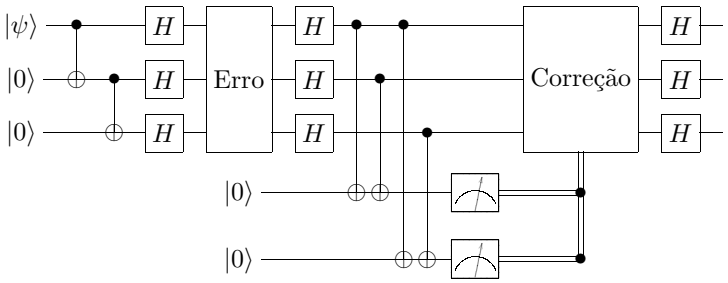


Figura 34 – Código de correção de *phase flip*.

B.2 Código de Shor

O código de Shor concatena 3 códigos de *bit flip* a um código de *phase flip* para construir um código tolerante a dois erros, um *bit flip* e um *phase flip*. Um qubit $|\psi\rangle_L$ pode ser preparado no código de Shor pelo circuito da Figura 35.

O código de Shor possui as seguintes palavras:

$$|0\rangle_L = \frac{(|000\rangle + |111\rangle)(|000\rangle + |111\rangle)(|000\rangle + |111\rangle)}{2\sqrt{2}} \tag{B.18}$$

$$|1\rangle_L = \frac{(|000\rangle - |111\rangle)(|000\rangle - |111\rangle)(|000\rangle - |111\rangle)}{2\sqrt{2}} \tag{B.19}$$

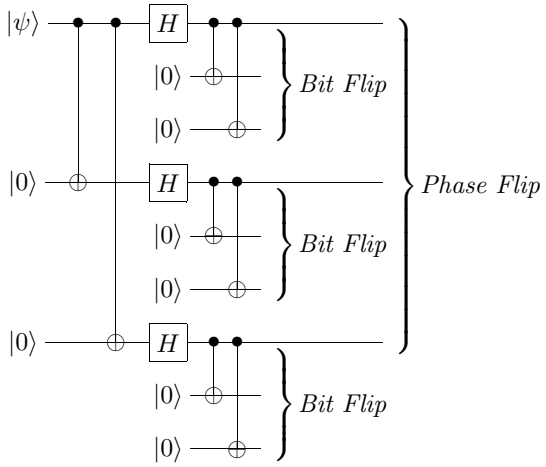


Figura 35 – Preparação do código de Shor.

Para preparar o qubit $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$ no código de Shor primeiramente preparamos ele no código de *phase flip*

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \rightarrow \alpha\left(\frac{|000\rangle + |111\rangle}{\sqrt{2}}\right) + \beta\left(\frac{|000\rangle - |111\rangle}{\sqrt{2}}\right) \tag{B.20}$$

Em seguida preparamos cada qubit com o código de *bit flip*

$$\begin{aligned} |\psi\rangle_L =& \alpha\left[\frac{(|000\rangle + |111\rangle)(|000\rangle + |111\rangle)(|000\rangle + |111\rangle)}{2\sqrt{2}}\right] \\ &+ \beta\left[\frac{(|000\rangle - |111\rangle)(|000\rangle - |111\rangle)(|000\rangle - |111\rangle)}{2\sqrt{2}}\right] \end{aligned} \tag{B.21}$$

Para identificar e corrigir os erros, inicialmente corrigimos a parte do código referente ao *bit flip*, como visto na Figura 33, e, por fim, corrigimos *phase flip*, como visto na Figura 34.

B.3 Códigos estabilizadores

Os códigos estabilizadores são uma importante classe de códigos de correção de erro, pois eles trazem um formalismo que ajuda na generalização da correção de erros e na construção de circuitos. O código de Steane $[[7, 1, 3]]$ será usado como exemplo para apresentar a classe dos códigos estabilizadores, sendo os exemplos apresentados facilmente aplicados para outros códigos estabilizadores.

Os códigos estabilizadores são descritos por um conjunto gerador composto de operadores que formam um grupo fechado na multiplicação[1] que estabiliza todas as palavras do código. Um estado $|\psi\rangle$ é estabilizado por um operador K se e somente se $|\psi\rangle$ for autovetor de K com autovalor $+1$, ou seja

$$K\,|\psi\rangle = |\psi\rangle\,. \tag{B.22}$$

[1] Um conjunto $\mathcal{G}$ é fechado sobre uma operação $*$ se e somente se $a * b = c\ \forall a, b, c \in \mathcal{G}$.

Como exemplo, o código de *bit flip* é estabilizado pelo grupo de operadores $\mathcal{G} = \{ZZI, ZIZ, IZZ, III\}$. Pode ser facilmente verificado que o $\mathcal{G}$ é fechado sobre a multiplicação e que toda palavra do código de *bit flip* é autovetor com autovalor $+1$ dos operadores de $\mathcal{G}$. Podemos definir $\mathcal{G}$ por seu conjunto gerador, ou seja, por um subconjunto que consiga gerar todos os elementos de $\mathcal{G}$ a partir da multiplicação. Como por exemplo, o grupo $\mathcal{G}$ pode ser gerado pelo conjunto gerador $\langle IZZ, ZIZ\rangle$, pois

$$ZZI = IZZ \times ZIZ \tag{B.23}$$
$$III = IZZ \times IZZ. \tag{B.24}$$

Um código estabilizador é totalmente descrito pelos seus estabilizadores, assim, os seguintes códigos podem ser descritos como:

- Código de *bit flip* $[[3, 1, 3]]$

$$K^1 = IZZ \qquad K^2 = ZIZ. \tag{B.25}$$

- Código de *phase flip* $[[3, 1, 3]]$

$$K^1 = IXX \qquad K^2 = XIX. \tag{B.26}$$

- Código de Shor $[[9, 1, 3]]$

$$K^1 = Z_0Z_1 \qquad K^2 = Z_0Z_2 \qquad K^3 = Z_3Z_4 \tag{B.27}$$
$$K^4 = Z_3Z_5 \qquad K^5 = Z_6Z_7 \qquad K^6 = Z_6Z_9 \tag{B.28}$$
$$K^7 = X_0X_1X_2X_3X_4X_5 \qquad K^8 = X_0X_1X_2X_6X_7X_8 \tag{B.29}$$

- Código de Steane (7-qubits) $[[7, 1, 3]]$

$$K^1 = IIIXXXX \qquad K^2 = XIXIXIX \qquad K^3 = IXXIIXX \tag{B.30}$$
$$K^4 = IIIZZZZ \qquad K^5 = ZIZIZIZ \qquad K^6 = IZZIIZZ \tag{B.31}$$

- Código de 5-qubits $[[5, 1, 3]]$

$$K^1 = IZXXZI \qquad K^2 = ZIZXXI \tag{B.32}$$
$$K^3 = XZIZXI \qquad K^4 = XXZIZI \tag{B.33}$$

O código de 5-qubits não será detalhado aqui. Para mais informações veja a referência 23.

B.3.1 Preparação do estado

Para preparar um conjunto de qubits em um qubit lógico, precisamos projetá-los como autovetor com autovalor +1 de todos os estabilizadores do código.

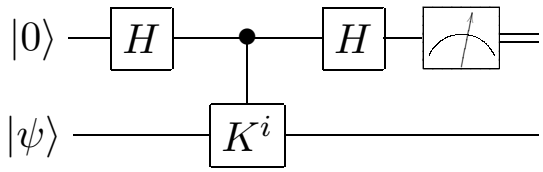


Figura 36 – Prepara $|\psi\rangle$ como autovetor ± 1 de K.

Para preparar o qubit $|\psi\rangle$, primeiramente, é necessário projetá-lo como autovalor ± 1 de todos estabilizadores K^i do código. Como na Figura 36, adicionamos uma ancila no estado $|0\rangle$ e passamos ela por uma porta Hadamard, em seguida, é aplicada a porta controlada K^i deixando o sistema no estado

$$\frac{|\psi\rangle|0\rangle + K^i|\psi\rangle|1\rangle}{\sqrt{2}.} \tag{B.34}$$

Como $|\psi\rangle$ só pode ser autovalor ± 1 de K^i, após o segundo Hadamard temos

$$|\psi\rangle|0\rangle + K^i|\psi\rangle|0\rangle \tag{B.35}$$

se $|\psi\rangle$ for autovalor $+1$ de K^i, e

$$|\psi\rangle|1\rangle - K^i|\psi\rangle|1\rangle \tag{B.36}$$

se $|\psi\rangle$ for autovalor -1 de K^i. É feita a medida da ancila e, após passar por todos os K^i estabilizadores, usamos os resultados das medidas para transformar o estado de autovalor ± 1 para autovalor $+1$ de todos K^i.

Como por exemplo, para preparar um qubit no estado $|0\rangle_L$ do código Steane é necessário, inicialmente que os qubits estejam no estado $|0\rangle^{\otimes 7}$, então são adicionadas 3 ancilas no estado $|000\rangle$, que passam cada uma por uma porta de Hadamard, para, logo em seguida, serem aplicadas às portas K^1,K^2 e K^3 (equação B.30) controladas. Logo após, novamente cada ancila passa por uma porta de Hadamard, e, ao final, elas medidas, e, o resultado da medida é usado para aplicar uma porta Z_i, dependente do resultado da medida, que projeta o estado em um autovetor com autovalor $+1$ de todos os estabilizadores do código de Steane. Esse processo pode ser visto na Figura 37.

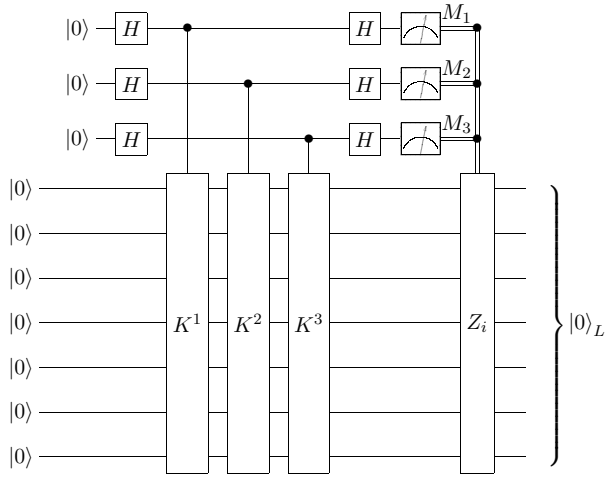


Figura 37 – Circuito para preparar $|0\rangle^{\otimes 7}$ em $|0\rangle_L$ do código de 7-qubits, após a medida das ancilas é aplicado uma porta Z no qubit $i = 4M_1 + M_2 + 2M_3 - 1$.

B.3.2 Medida de síndrome

Os estabilizadores comutam com as operações lógicas e anti comutam com os erros, ou seja, sendo U uma operação lógica que mapeia um estado do código a outro

$$K^i U = U K^i, \quad \forall K^i, \tag{B.37}$$

e, sendo E um erro que leva um estado pertencente ao código a um estado fora do código

$$K^i E = -E K^i. \tag{B.38}$$

Com isso, podemos usar o mesmo circuito da Figura 37 para fazer a medida da síndrome. Para isso, considere $|\psi\rangle_L$ como sendo uma palavra do código e E como sendo um possível erro. Assim, para fazer a medida da síndrome é adicionada uma ancila no estado $|0\rangle$

$$E\,|\psi\rangle_L\,|0\rangle\,, \tag{B.39}$$

em seguida, a ancila é passada por uma porta de Hadamard

$$\frac{E\,|\psi\rangle\,|0\rangle + E\,|\psi\rangle\,|1\rangle}{\sqrt{2}}, \tag{B.40}$$

logo em seguida, é aplicada a operação controlada com o estabilizador K^i tendo a ancila como qubit de controle

$$\frac{E\,|\psi\rangle\,|0\rangle + K^i E\,|\psi\rangle\,|1\rangle}{\sqrt{2}}, \tag{B.41}$$

podemos reescrever o estado como

$$\frac{E\,|\psi\rangle\,|0\rangle + (-1)^m E K^i\,|\psi\rangle\,|1\rangle}{\sqrt{2}} = \frac{E\,|\psi\rangle\,|0\rangle + (-1)^m E\,|\psi\rangle\,|1\rangle}{\sqrt{2}}, \tag{B.42}$$

dado que o erro anti comuta com o operador K^i, m é igual a 0 se E não for um erro e 1 se for um erro, assim, após passar a ancila novamente pela porta de Hadamard temos

$$E\left|\psi\right\rangle\left|0\right\rangle \text{ se } E \text{ não for um erro, e} \tag{B.43}$$

$$E\left|\psi\right\rangle\left|1\right\rangle \text{ se } E \text{ for um erro.} \tag{B.44}$$

Assim, com a medida das ancilas o estado colapsa em um estado onde aconteceu ou não aconteceu o erro. Esse processo é feito para todos os estabilizadores a fim de identificar e corrigir os erros.

A correção do erro para o código de Steane é dividida em duas partes, a primeira, usa o mesmo circuito da Figura 37 para corrigir *phase flip* e, a segunda, usa um circuito semelhante a primeira parte, apenas trocando os estabilizadores que são usados para corrigir *Bit Flip*. A Figura 38 traz todo o circuito para correção de erro do código de 7-qubits.

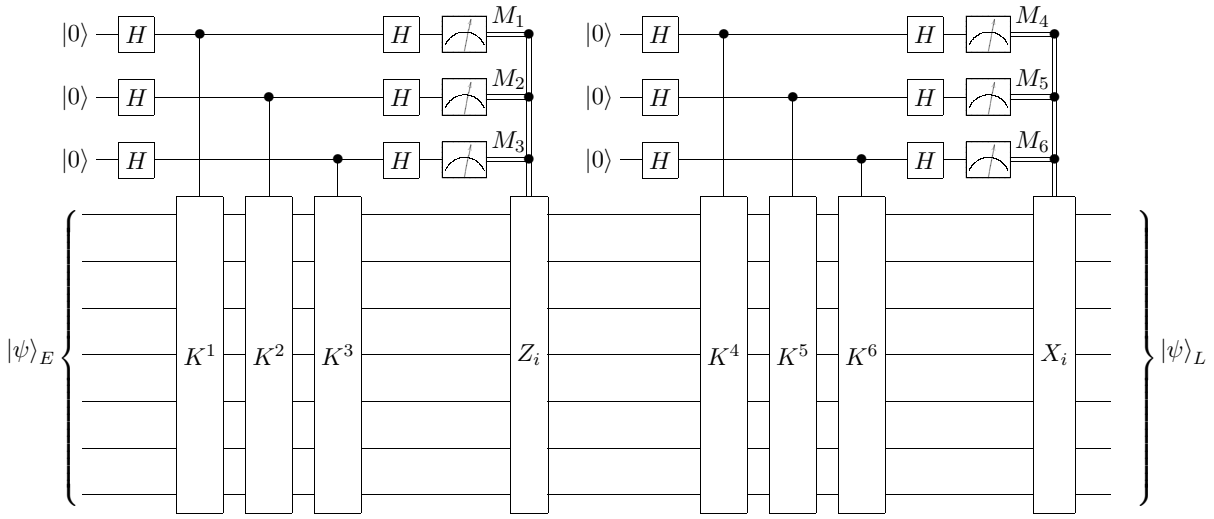


Figura 38 – A medida de síndrome para código de Steane é dividida em duas partes, onde, a primeira, identifica e corrige *phase flip* aplicando uma porta Z no qubit $i = 4M_1 + M_2 + 2M_3 - 1$ e, a segunda, identifica e corrige *bit flip* aplicando uma porta X no quibit $i = 4M_4 + M_5 + 2M_6 - 1$.

APÊNDICE C – Implementação dos canais de erro

As Figuras 23, 24, 25, 26 e 27 foram geradas com o código abaixo usando GNU Octave version 4.4.1.

C.1 main.m

```
[x y z] = sphere(20);
rho = bloch_sphere(x, y, z);

[X,Y] = meshgrid(1:size(x));
C = mod(X,11)(:,1:10);
C = [C fliplr(C) zeros(21, 1)];
X = C;
Z = Y;

#plot_bloch(@bit_flip, rho, "Bit Flip", 0.85, Z)
#plot_bloch(@phase_flip, rho, "Phase Flip", 1, X)
#plot_bloch(@bit_phase_flip, rho, "Bit-phase Flip", 0.75, Z)
#plot_bloch(@depolarizing_channel, rho, "Depolarizing Channel", 0.9, Z)
plot_bloch(@amplitude_damping, rho, "Amplitude Damping", 0.9, Z)
```

C.2 bloch_sphere.m

```
function rho = bloch_sphere(x, y, z)
  pauli_matrices
  i = 1;
  rho = {};
  for r = transpose([x(:) y(:) z(:)])
    rho{i++} = (eye(2)+r(1)*X+r(2)*Y+r(3)*Z)/2;
  end
end
```

C.3 pauli_matrices.m

```
X = [0 1;
     1 0];
Y = [0 -i;
     i 0];
Z = [1 0;
     0 -1];
```

C.4 expected_values.m

```
function [x y z] = expected_values(rho)
  pauli_matrices
  x = [];
  y = [];
  z = [];
  for i = 1 : length(rho)
    x = [x trace(X*rho{i})];
    y = [y trace(Y*rho{i})];
    z = [z trace(Z*rho{i})];
  end
end
```

C.5 plot_bloch.m

```
function plot_bloch(E, rho, name, p, color)
  rho = E(rho, p);
  [ex ey ez] = expected_values(rho);
  n = sqrt(size(ex))(2);
  for i = 1:n
    for j = 1:n
    exi(i,j) = ex(i+(j-1)*n);
    eyi(i,j) = ey(i+(j-1)*n);
    ezi(i,j) = ez(i+(j-1)*n);
    end
  end
  surf(exi, eyi, ezi, color);
  title(strcat(name, " : p = ", mat2str(p)));
  axis equal
```

```
15    xlabel("x")
16    ylabel("y")
17    zlabel("z")
18    axis([-1 1 -1 1 -1 1])
19  end
```

C.6 bit_flip.m

```
1   function rho = bit_flip(rho, p)
2     X = [0 1;
3          1 0];
4
5     E0 = sqrt(p)*eye(2);
6     E1 = sqrt(1-p)*X;
7
8     for i = 1 : length(rho)
9       rho{i} = E0*rho{i}*E0' + E1*rho{i}*E1';
10    end
11  end
```

C.7 phase_flip.m

```
1   function rho = phase_flip(rho, p)
2     Z = [1 0;
3          0 -1];
4
5     E0 = sqrt(p)*eye(2);
6     E1 = sqrt(1-p)*Z;
7
8     for i = 1 : length(rho)
9       rho{i} = E0*rho{i}*E0' + E1*rho{i}*E1';
10    end
11  end
```

C.8 bit_phase_flip.m

```
1   function rho = bit_phase_flip(rho, p)
2     Y = [0 -i;
3          i 0];
```

```
4
5     E0 = sqrt(p)*eye(2);
6     E1 = sqrt(1-p)*Y;
7
8     for i = 1 : length(rho)
9       rho{i} = E0*rho{i}*E0' + E1*rho{i}*E1';
10    end
11  end
```

C.9 depolarizing_channel.m

```
1  function rho = depolarizing_channel(rho, p)
2    for i = 1 : length(rho)
3      rho{i} = p*eye(2)/2 + (1-p)*rho{i};
4    end
5  end
```

C.10 amplitude_damping.m

```
1  function rho = amplitude_damping(rho, gamma)
2    E0 = [1             0;
3          0 sqrt(1-gamma)];
4    E1 = [0 sqrt(gamma);
5          0           0];
6
7    for i = 1 : length(rho)
8      rho{i} = E0*rho{i}*E0' + E1*rho{i}*E1';
9    end
10 end
```

APÊNDICE D – Código fonte do simulador QSystem

Código fonte disponível em <https://gitlab.com/evandro-crr/qsystem>.

D.1 LICENSE

```
MIT License

Copyright (c) 2019 Evandro Chagas Ribeiro da Rosa <ev.crr97@gmail.com>
Copyright (c) 2019 Bruno Gouvêa Taketani <b.taketani@ufsc.br>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

D.2 makefile

```
OBJ = src/gates.o src/microtar.o src/qs_ancillas.o src/qs_errors.o
OBJ += src/qs_make.o src/qs_evol.o src/qs_measure.o
OBJ += src/qs_utility.o src/qsystem.o
HEADER = $(wildcard header/*.h)

OUT = _qsystem.so

PYTHON = /usr/include/python3.7m/

```

```
10 CFLAGS = -Wall -O2 -fPIC
11 CXXFLAGS = $(CFLAGS) -std=c++17 -I$(PYTHON)
12 CLINK = -shared -Xlinker -export-dynamic
13
14 all: $(OBJ) qsystem.py
15         $(CXX) $(OBJ) -o $(OUT) $(CXXFLAGS) $(CLINK)
16
17 %.o: %.cpp $(HEADER)
18         $(CXX) -c $< -o $@ $(CXXFLAGS)
19
20 %.o: %.c
21         $(CC) -c $< -o $@ $(CFLAGS)
22
23 %.cpp: %.i $(HEADER)
24         swig -c++ -python -o $@ $<
25
26 qsystem.py:
27         ln -s src/qsystem.py $@
28
29 dist: src/qsystem.cpp qsystem/__init__.py armadillo-code
30         python setup.py sdist
31
32 qsystem/__init__.py:
33         mkdir -p qsystem
34         ln -s ../src/qsystem.py $@
35
36 armadillo-code:
37         git clone https://gitlab.com/conradsnicta/armadillo-code.git --branch
          ↪  9.300.x
38
39 install: dist
40         pip install dist/*
41
42 clean:
43         rm -rf $(OUT) __pycache__ qsystem.py
44         rm -rf src/{qsystem.cpp,qsystem.py,*.o}
45         rm -rf build dist qsystem QSystem.egg-info armadillo-code
```

D.3 MANIFEST.in

```
1 include header/*.h
2 include armadillo-code/include/*
3 include armadillo-code/include/armadillo_bits/*
```

D.4 README.md

````
# QSystem
[![PyPI](https://img.shields.io/pypi/v/qsystem.svg)](https://pypi.org/project/QSystem/)
[![PyPI - License](https://img.shields.io/pypi/l/qsystem.svg?color=brightgree)](https://gitlab.com
[![PyPI - Python Version](https://img.shields.io/pypi/pyversions/qsystem.svg?color=red)](https://www.pyth
[![Wiki](https://img.shields.io/badge/wiki-available-sucess.svg)](https://gitlab.com/evandro

A quantum computing simulator for Python.

-------------------------
The QSystem simulator is inspired in the quantum circuit model, so it's easy to
convert any quantum circuit to  Python.

Like the follow example:

![circ](https://gitlab.com/evandro-crr/qsystem/raw/1.0.0/circ.svg?inline=false)

```python
from qsystem import Gates, QSystem
from cmath import exp, pi
gates = Gates()

q = QSystem(3, gates, 24)                # init q0, q1, q2

q.evol(gate='H', qbit=0, count=3)        # H q0; H q1; H q2
q.add_ancillas(4)                        # init a0, a1, a2, a3

q.evol(gate='X', qbit=6)                 # X a3
q.cnot(target=4, control=[2])            # CNOT a1, q2
q.cnot(5, [2])                           # CNOT a2, q2
q.cnot(5, [3])                           # CNOT a2, a0
q.cnot(3, [1, 5])                        # Toffoli a1, q1, a2
q.cnot(5, [3])                           # CNOT a2, a0
q.cnot(4, [6])                           # CNOT a1, a3
q.cnot(6, [1, 4])                        # Toffoli a3, q1, a1
q.cnot(4, [6])                           # CNOT a1, a3

q.measure(qbit=3, count=4)               # measure a0, a1, a2, a3
print('ancillas measurement =', q.bits()[3:])
# ancillas measurement = [0, 1, 0, 0]
q.rm_ancillas()                          # rm a0, a1, a2, a3

q.evol('H', 0)                           # H q0
q.cphase(phase=1j, target=1, control=[0]) # Controlled S q1, q0
````
````

```
q.evol('H', 1)                          # H q1
q.cphase(exp(pi*1j/4), 2, [0])          # Controlled T q2, q0   = q.qft(0, 3)
q.cphase(1j, 2, [1])                    # Controlled S q2, q1
q.evol('H', 2)                          # H q1
q.swap(0, 2)                            # SWAP q0, q2

q.measure(0, 3)                         # measure q0, q1, q2
print('final measurement =', q.bits())
# final measurement = [1, 0, 0]
```

Seed the [wiki](https://gitlab.com/evandro-crr/qsystem/wikis/home) for documentation.

This software is suported by
<img src="http://www.fapesc.sc.gov.br/wp-content/uploads/2014/09/logo-Fapesc-fundo-transparente.pn
alt="FEPESC" width="54">

D.5 setup.py

```
from setuptools import setup
from setuptools.extension import Extension

with open('README.md', 'r') as fh:
    long_description = fh.read()

ext_module = Extension('_qsystem',
        sources=['src/qsystem.cpp',
                 'src/gates.cpp',
                 'src/microtar.c',
                 'src/qs_ancillas.cpp',
                 'src/qs_errors.cpp',
                 'src/qs_evol.cpp',
                 'src/qs_make.cpp',
                 'src/qs_measure.cpp',
                 'src/qs_utility.cpp'],
        include_dirs=['armadillo-code/include'],
        extra_compile_args=['-std=c++17']
        )

setup (name = 'QSystem',
       version='1.0.0',
       author='Evandro Chagas Ribeiro da Rosa, Bruno Gouvêa Taketani',
       author_email='ev.crr97@gmail.com',
```

```
25          description='A quantum computing simulator for Python',
26          long_description=long_description,
27          long_description_content_type='text/markdown',
28          url='https://gitlab.com/evandro-crr/qsystem',
29          ext_modules = [ext_module],
30          packages=['qsystem'],
31          classifiers=[
32               'Programming Language :: Python :: 3',
33               'Intended Audience :: Science/Research',
34               'License :: OSI Approved :: MIT License',
35           ]
36          )
```

D.6 header/

D.6.1 gates.h

```
1  /* MIT License
2   *
3   * Copyright (c) 2019 Evandro Chagas Ribeiro da Rosa <ev.crr97@gmail.com>
4   * Copyright (c) 2019 Bruno Gouvêa Taketani <b.taketani@ufsc.br>
5   *
6   * Permission is hereby granted, free of charge, to any person obtaining a copy
7   * of this software and associated documentation files (the "Software"), to deal
8   * in the Software without restriction, including without limitation the rights
9   * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
10  * copies of the Software, and to permit persons to whom the Software is
11  * furnished to do so, subject to the following conditions:
12  *
13  * The above copyright notice and this permission notice shall be included in all
14  * copies or substantial portions of the Software.
15  *
16  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
17  * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
18  * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
19  * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
20  * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
21  * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
22  * SOFTWARE.
23  */
24
25 #pragma once
26 #include "using.h"
27 #include <map>
28 #include <armadillo>
29 #include <Python.h>
```

```
30
31 //! Class that holds some quantum gates
32 /*!
33  * This class store the quantum gates of one qubit and the quantum gates
34  * created by the user.
35  */
36 class Gates {
37   public:
38     //! Constructor
39     /*!
40      * The constructor initialize the follow quantum gates:
41      * * ```'I'``` = \f$\begin{bmatrix}
42      *                   1 & 0 \\
43      *                   0 & 1
44      *                  \end{bmatrix}\f$
45      * * ```'Y'``` = \f$\begin{bmatrix}
46      *                   0 & 1 \\
47      *                   1 & 0
48      *                  \end{bmatrix}\f$
49      * * ```'X'``` = \f$\begin{bmatrix}
50      *                   0 & -i \\
51      *                   i & 0
52      *                  \end{bmatrix}\f$
53      * * ```'Z'``` = \f$\begin{bmatrix}
54      *                   1 & 0 \\
55      *                   0 & -1
56      *                  \end{bmatrix}\f$
57      * * ```'H'``` = \f${1\over\sqrt{2}}\begin{bmatrix}
58      *                                    1 & 1 \\
59      *                                    1 & -1
60      *                                   \end{bmatrix}\f$
61      * * ```'S'``` = \f$\begin{bmatrix}
62      *                   1 & 0 \\
63      *                   0 & i
64      *                  \end{bmatrix}\f$
65      * * ```'T'``` = \f$\begin{bmatrix}
66      *                   1 & 0 \\
67      *                   0 & e^{i\pi\over4}
68      *                  \end{bmatrix}\f$.
69      */
70     Gates();
71
72     //! Load constructor
73     /*!
74      * Used to load gates from a file created by the method Gates::save.
75      *
76      * \param path to the file that store the quantum gates.
```

```
77        *
78        * \sa Gates::save
79        */
80       Gates(std::string path);
81
82       //! Create an one qubit gate
83       /*!
84        * The param `matrix` must have 4 elements organized like: `[a00, a01, a10,
85        * a11]` = \f$\begin{bmatrix}a00&a01\\a10&a11\end{bmatrix}\f$.
86        *
87        * \param name of the new quantum gate.
88        * \param matrix list of complex.
89        *
90        * \sa Gates::make_mgate Gates::make_cgate Gates::make_fgate
91        */
92       void make_gate(char name, vec_complex matrix);
93
94       //! Create a quantum gate from a sparse matrix
95       /*!
96        * The matrix of the new quantum gate is created like: `U(row[i], col[i]) =
97        * value[i]`.
98        *
99        * \param name of the new quantum gate.
100       * \param size number of qubits affected by the new gate
101       * \param row list of row indices.
102       * \param col list of column indices
103       * \param value list of non-zero elements.
104       *
105       * \sa Gates::make_gate Gates::make_cgate Gates::make_fgate
106       */
107      void make_mgate(std::string name,
108                      size_t size,
109                      vec_size_t row,
110                      vec_size_t col,
111                      vec_complex value);
112
113      //! Create a controlled gate of X and Z
114      /*!
115       * Apply a sequence of gates `X`, `Z` and `I` if the `control` gates are in
116       * the state \f$\left|1\right>\f$.
117       *
118       * \param name of the new quantum gate.
119       * \param gates string with all the one qubit quantum gates, *e.g.*
      ↪  `"XZZXI"`.
120       * \param control list of control gates.
121       *
122       * \sa Gates::make_gate Gates::make_mgate Gates::make_fgate
```

```
123         */
124         void make_cgate(std::string name,
125                         std::string gates,
126                          vec_size_t control);
127
128         //! Create a quantum gate from a Python function
129         /*!
130          * The Python function `func` must take as argumente and return an `int`.
131          * This function must be defined in the range 0 to \f$2^\text{size}\f$-1.
132          *
133          * It's passible to pass an iterator that tells the order of creation of the
    ↪  matrix.
134          *
135          * \param name of the new quantum gate.
136          * \param func Python function.
137          * \param size number of qubits affected by the gate.
138          * \param iterator with the matrix order of creation.
139          *
140          * \sa Gates::make_gate Gates::make_mgate Gates::make_cgate
141          */
142         void make_fgate(std::string name,
143                           PyObject* func,
144                              size_t size,
145                           PyObject* iterator=Py_None);
146
147         //! Get a string with information
148         /*!
149          *  This method is used in Python to cast a instance to `str`.
150          *
151          *  \return String with information of the mutliple qubits gates.
152          */
153         std::string __str__();
154
155         //! Save the multiple qubits quantum gates in a file
156         /*!
157          * The file created is a tar with all the multiple qubits quantum gates.
158          *
159          * \param path to the file that will be created.
160          * \sa Gates::Gates
161          */
162         void save(std::string path);
163
164         //! Return a quantum gate of one qubit
165         /*!
166          * This method is used by the QSystem class.
167          *
168          * \param gate name of the gate.
```

```
169       * \return Sparse matrix of the gate.
170       * \sa Gates::mget
171       */
172      arma::sp_cx_mat& get(char gate);
173
174      //! Retorn a quantum gate of multiple qubits
175      /*!
176       * This method is used by the QSystem class.
177       *
178       * \param gate name of the gate.
179       * \return Sparse matrix of the gate.
180       * \sa Gates::get
181       */
182      arma::sp_cx_mat& mget(std::string gate);
183
184    private:
185    std::map<std::string, arma::sp_cx_mat> mmap;
186
187    std::map<char, arma::sp_cx_mat> map{
188      {'I', arma::sp_cx_mat{arma::cx_mat{{{1,0}, {0,0}},
189                                        {{0,0}, {1,0}}}}},
190      {'X', arma::sp_cx_mat{arma::cx_mat{{{0,0}, {1,0}},
191                                        {{1,0}, {0,0}}}}},
192      {'Y', arma::sp_cx_mat{arma::cx_mat{{{0,0}, {0,-1}},
193                                        {{0,1}, {0,0}}}}},
194      {'Z', arma::sp_cx_mat{arma::cx_mat{{{1,0}, {0,0}},
195                                        {{0,0}, {-1,0}}}}},
196      {'H', arma::sp_cx_mat{(1/sqrt(2))*arma::cx_mat{{{1,0}, {1,0}},
197                                                    {{1,0}, {-1,0}}}}},
198      {'S', arma::sp_cx_mat{arma::cx_mat{{{1,0}, {0,0}},
199                                        {{0,0}, {0,1}}}}},
200      {'T', arma::sp_cx_mat{arma::cx_mat{{{1,0}, {0,0}},
201                                        {{0,0}, {1/sqrt(2),1/sqrt(2)}}}}},
202    };
203  };
```

D.6.2 microtar.h

```
1  /**
2   * Copyright (c) 2017 rxi
3   *
4   * This library is free software; you can redistribute it and/or modify it
5   * under the terms of the MIT license. See `microtar.c` for details.
6   */
7
8  #ifndef MICROTAR_H
9  #define MICROTAR_H
```

```
10
11 #ifdef __cplusplus
12 extern "C"
13 {
14 #endif
15
16 #include <stdio.h>
17 #include <stdlib.h>
18
19 #define MTAR_VERSION "0.1.0"
20
21 enum {
22   MTAR_ESUCCESS     =  0,
23   MTAR_EFAILURE     = -1,
24   MTAR_EOPENFAIL    = -2,
25   MTAR_EREADFAIL    = -3,
26   MTAR_EWRITEFAIL   = -4,
27   MTAR_ESEEKFAIL    = -5,
28   MTAR_EBADCHKSUM   = -6,
29   MTAR_ENULLRECORD  = -7,
30   MTAR_ENOTFOUND    = -8
31 };
32
33 enum {
34   MTAR_TREG   = '0',
35   MTAR_TLNK   = '1',
36   MTAR_TSYM   = '2',
37   MTAR_TCHR   = '3',
38   MTAR_TBLK   = '4',
39   MTAR_TDIR   = '5',
40   MTAR_TFIFO  = '6'
41 };
42
43 typedef struct {
44   unsigned mode;
45   unsigned owner;
46   unsigned size;
47   unsigned mtime;
48   unsigned type;
49   char name[100];
50   char linkname[100];
51 } mtar_header_t;
52
53
54 typedef struct mtar_t mtar_t;
55
56 struct mtar_t {
```

```
57    int (*read)(mtar_t *tar, void *data, unsigned size);
58    int (*write)(mtar_t *tar, const void *data, unsigned size);
59    int (*seek)(mtar_t *tar, unsigned pos);
60    int (*close)(mtar_t *tar);
61    void *stream;
62    unsigned pos;
63    unsigned remaining_data;
64    unsigned last_header;
65  };
66
67
68  const char* mtar_strerror(int err);
69
70  int mtar_open(mtar_t *tar, const char *filename, const char *mode);
71  int mtar_close(mtar_t *tar);
72
73  int mtar_seek(mtar_t *tar, unsigned pos);
74  int mtar_rewind(mtar_t *tar);
75  int mtar_next(mtar_t *tar);
76  int mtar_find(mtar_t *tar, const char *name, mtar_header_t *h);
77  int mtar_read_header(mtar_t *tar, mtar_header_t *h);
78  int mtar_read_data(mtar_t *tar, void *ptr, unsigned size);
79
80  int mtar_write_header(mtar_t *tar, const mtar_header_t *h);
81  int mtar_write_file_header(mtar_t *tar, const char *name, unsigned size);
82  int mtar_write_dir_header(mtar_t *tar, const char *name);
83  int mtar_write_data(mtar_t *tar, const void *data, unsigned size);
84  int mtar_finalize(mtar_t *tar);
85
86  #ifdef __cplusplus
87  }
88  #endif
89
90  #endif
```

D.6.3 qsystem.h

```
1   /* MIT License
2    *
3    * Copyright (c) 2019 Evandro Chagas Ribeiro da Rosa <ev.crr97@gmail.com>
4    * Copyright (c) 2019 Bruno Gouvêa Taketani <b.taketani@ufsc.br>
5    *
6    * Permission is hereby granted, free of charge, to any person obtaining a copy
7    * of this software and associated documentation files (the "Software"), to deal
8    * in the Software without restriction, including without limitation the rights
9    * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
10   * copies of the Software, and to permit persons to whom the Software is
```

```
11  * furnished to do so, subject to the following conditions:
12  *
13  * The above copyright notice and this permission notice shall be included in all
14  * copies or substantial portions of the Software.
15  *
16  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
17  * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
18  * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
19  * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
20  * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
21  * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
22  * SOFTWARE.
23  */
24
25 #pragma once
26 #include "gates.h"
27 #include <Python.h>
28 #include <variant>
29
30 //! Quantum circuit simulator class.
31 class QSystem {
32
33   struct Gate_aux {
34     Gate_aux();
35     ~Gate_aux();
36
37     bool busy();
38
39     enum Tag {GATE_1, GATE_N,
40               CNOT, CPHASE,
41               SWAP, QFT} tag;
42
43     std::variant<char,
44                  std::string,
45                  cnot_pair,
46                  cph_tuple> data;
47
48     size_t size;
49     bool   inver;
50   };
51
52   enum Bit {NONE, ZERO, ONE};
53
54   public:
55     //! Constructor
56     /*!
57      * All qubits are initialized in the state \f$\left|0\right>\f$.
```

```
58        *
59        * \param nqbits number of qubits in the system.
60        * \param gates instance of class Gates that holds the gates used in the
61        * method QSystem::evol.
62        * \param seed for the pseudorandom number generator.
63        * \param state representation of the system, use `"vector"` for vector.
64        * state and `"matrix"` for density matrix
65        */
66       QSystem(size_t nqbits,
67                Gates& gates,
68               size_t seed=42,
69          std::string state="vector");
70
71       ~QSystem();
72
73       //! Apply a quantum gate
74       /*!
75        * The gate will be applied in from qubits `qbit` to `qbit*cout*(size of
76        * the gate)`. If `gate` parameter is just one character long, the size of
77        * the gate is necessarily one.
78        *
79        * \param gate name of the gate that will be user.
80        * \param qbit qubit affected by the gate.
81        * \param count number of successive repetitions of the gate.
82        * \param inver if true, apply the inverse quantum gate.
83        * \sa QSystem::cnot QSystem::cphase QSystem::qft QSystem::swap
84        */
85       void evol(std::string gate,
86                      size_t qbit,
87                      size_t count=1,
88                        bool inver=false);
89       //! Apply a controlled not
90       /*!
91        * Apply a not in the `target` qubit if all the `control` qubits are in the
92        * state \f$\left|1\right>\f$.
93        *
94        * \param target target qubit.
95        * \param control list of control qubits.
96        * \sa QSystem::evol QSystem::cphase QSystem::qft QSystem::swap
97        */
98       void cnot(size_t target, vec_size_t control);
99
100      //! Apply a controlled phase
101      /*!
102       * Apply \f$\begin{bmatrix}1&0\\0&e^\phi\end{bmatrix}\f$, whare
103       * \f$e^\phi\f$ = `phase`, in the `target` qubit if all the `control`
104       * qubits are in the state \f$\left|1\right>\f$.
```

```
105         *
106         * \param phase \f$e^\phi\f$ value.
107         * \param target target qubit.
108         * \param control list of control qubits.
109         * \sa QSystem::evol QSystem::cnot QSystem::qft QSystem::swap
110         */
111        void cphase(complex phase, size_t target, vec_size_t control);
112
113        //! Apply a quantum Fourier transformation
114        /*!
115         * Apply the QFT in the range of qubits (`qbegin`, `qend`].
116         *
117         * \param qbegin first qubit affected.
118         * \param qend last qubit affected +1.
119         * \sa QSystem::evol QSystem::cnot QSystem::cphase QSystem::swap
120         */
121        void qft(size_t qbegin, size_t qend, bool inver=false);
122
123        //! Swap two qubit
124        /*!
125         * \param qbit_a qubit that gonna be swapped with `qbit_b`.
126         * \param qbit_a qubit that gonna be swapped with `qbit_a`.
127         * \sa QSystem::evol QSystem::cnot QSystem::cphase QSystem::qft
128         */
129        void swap(size_t qbit_a, size_t qbit_b);
130
131        //! Measure qubits in the computational base
132        /*!
133         * Measure qubits from `qbit` to `qbit*cout`. All the measurements results
134         * are assessable throw the QSystem::bits method.
135         *
136         * \param qbit qubit affected by the measurement.
137         * \param count number qubits measured from `qbit`.
138         * \sa QSystem::measure_all QSystem::bits
139         */
140        void measure(size_t qbit, size_t count=1);
141
142        //! Measure all qubits in the computational base
143        /*!
144         * The measurements results are assessable throw the QSystem::bits method.
145         * \sa QSystem::measure QSystem::bits
146         */
147        void measure_all();
148
149        //! Get the measurements results
150        /*!
151         * The measurements results are stored in a list. Whare the n-th item is
```

```
152         * the measurement result of the qubit n. If the n-th qubits has never been
153         * measured it's value is `None`.
154         *
155         * \return List of the measurement result.
156         * \sa QSystem::measure QSystem::measure_all
157         */
158        vec_int bits();
159
160        //! Apply a bit, phase or bit-phase flip error
161        /*!
162         * Apply the Kraus operator
163         * \f[
164         *    E_0 = \sqrt{p}\sigma\\
165         *    E_1 = \sqrt{1-p}I,
166         * \f]
167         * whare \f$\sigma\f$ is \f$\begin{bmatrix}1&0\\0&1\end{bmatrix}\f$ if
168         * `gate` is ```'X'```,  \f$\begin{bmatrix}1&0\\0&1\end{bmatrix}\f$ if
169         * `gate` is ```'Z'``` or \f$\begin{bmatrix}1&0\\0&-1\end{bmatrix}\f$ if
170         * `gate` is ```'Y'```.
171         *
172         * \param gate use ```'X'``` for bit flip, ```'Z'``` for phase flip
173         * or ```'Y'``` for bit-phase flip.
174         * \param qbit qubit effected by the error.
175         * \param p probability of the error occur.
176         * \sa QSystem::amp_damping QSystem::dpl_channel QSystem::sum
177         */
178        void flip(char gate, size_t qbit, double p);
179
180        //! Apply an amplitude damping channel error
181        /*!
182         * Apply the Kraus operator
183         * \f[
184         *    E_0 = \begin{bmatrix}1&0\\0&\sqrt{1-p}\end{bmatrix}\\
185         *    E_1 = \begin{bmatrix}0&0\\ \sqrt{p}&0\end{bmatrix},
186         * \f]
187         *
188         * The system must be in density matrix representation to use this method,
189         * otherwise you can use the flow code to achieve a similar result:
190         * ```python
191         * def amp_damping(q, qbit, p):
192         *    from random import choices
193         *    if choices([True, False], weights=[p, 1-p])[0]:
194         *        q.measure(qbit)
195         *        if q.bits[qbit] == 1:
196         *            q.evol('X', qbit)
197         * ```
198         *
```

```
199         * \sa QSystem::flip QSystem::dpl_channel QSystem::sum
200         */
201        void amp_damping(size_t qbit, double p);
202
203        //! Apply a depolarization channel error
204        /*!
205         * Apply the operator
206         * \f[
207         *    \mathcal{E}(\rho) = \left(1-{3p\over4}\right)\rho
208         *    +{p\over4}(X\rho X+Y\rho Y+Z\rho Z),
209         * \f]
210         * that takes the qubit to the maximally mixed state with probability `p`.
211         *
212         * The system must be in density matrix representation to use this method.
213         *
214         * \param qbit qubit effected by the error.
215         * \param p probability of the error occur.
216         * \sa QSystem::flip QSystem::amp_damping QSystem::sum
217         */
218        void dpl_channel(size_t qbit, double p);
219
220        //! Apply a sum operator
221        /*!
222         * To apply some Kraus operator like
223         * \f[
224         *    E_1 = {\sqrt{p_1}} (U_{11}\otimes\dots\otimes U_{1n})\\
225         *    E_2 = {\sqrt{p_2}} (U_{21}\otimes\dots\otimes U_{2n})\\
226         *    \vdots\\
227         *    E_m = {\sqrt{p_m}} (U_{m1}\otimes\dots\otimes U_{mn}),
228         * \f]
229         * pass the follow parameters
230         * * `kraus` = [\f$U_{11}\otimes\dots\otimes U_{1n},\,
231         * U_{21}\otimes\dots\otimes U_{2n},\, \dots,\, U_{m1}\otimes\dots\otimes
232         * U_{mn}\f$] and
233         * * `p` = [\f$p_1,\,p_2,\,\dots,\,p_m\f$]
234         *
235         * The system must be in density matrix representation to use this method,
236         * otherwise you can use the flow code to achieve a similar result:
237         * ```python
238         * def sum(q, qbit, kraus, p):
239         *     from random import choices
240         *     aux = 0
241         *     for gate in choices(kraus, weights=p)[0]:
242         *         q.evol(gate, qbit+aux)
243         *         aux += 1
244         * ```
245         *
```

```
246         * \param qbit first qubit effected by the error.
247         * \param kraus Kraus operators list.
248         * \param p probability list.
249         * \sa QSystem::flip QSystem::amp_damping QSystem::dpl_channel
250         */
251        void sum(size_t qbit, vec_str kraus, vec_float p);
252
253        //! Get system state in a string
254        /*!
255         *  This method is used in Python to cast a instance to `str`.
256         *
257         * \return String with the system state.
258         * \sa QSystem::size QSystem::state
259         */
260        std::string __str__();
261
262        //! Get the number of qubits
263        /*!
264         * The ancillary qubits are include in the count.
265         *
266         * \return Number of qubits in the system.
267         * \sa QSystem::state
268         */
269        size_t size();
270
271        //! Get the system representation
272        /*!
273         * \return `"vector"` for vector representation and `"matrix"` for density
274         * matrix.
275         *
276         * \sa QSystem::size QSystem::change_to
277         */
278        std::string state();
279
280        //! Save the quantum state in a file
281        /*!
282         * The file is in a machine dependent binary format defined by the library
283         * Armadillo.
284         *
285         * \param path to the file that will be created.
286         * \sa QSystem::load
287         */
288        void save(std::string path);
289
290        //! Load the quantum state from a file
291        /*!
292         * When load, all qubits are set to non-ancillary.
```

```
293         *
294         * \param path to the file that will be loaded.
295         * \sa QSystem::save
296         */
297        void load(std::string path);
298
299        //! Change the system representation
300        /*!
301         * The change from vector representation to density matrix is done by
302         * \f$\left|\psi\right> \rightarrow
303         * \left|\psi\right>\mkern-7mu\left<\psi\right|\f$. But, in the change from
304         * density matrix to vector representation, just the measurement
305         * probability is maintained.
306         *
307         * \param new_state use "vector" to change vector representation a
308         * \sa QSystem::state
309         */
310        void change_to(std::string new_state);
311
312        //! Get the matrix of the quantum system
313        /*!
314         * This method is used in Python by the non-member function `get_matrix`.
315         * ```python
316         * def get_matrix(q):
317         *    from scipy import sparse
318         *    return sparse.csc_matrix(q.get_qbits()[0], q.get_qbits()[1])
319         * ```
320         *
321         * \return Tuple used to initialize a scipy space matrix.
322         * \sa QSystem::set_qbits
323         */
324        PyObject* get_qbits();
325
326        //! Change the matrix of the quantum system
327        /*!
328         * This method is used in Python by the non-member function `set_matrix`.
329         * ```python
330         * def set_matrix(q, m):
331         *     from scipy import sparse
332         *     from math import log2
333         *     m = sparse.csc_matrix(m)
334         *     if m.shape[0] == m.shape[1]:
335         *         state = 'matrix'
336         *     else:
337         *         state = 'vector'
338         *     size = int(log2(m.shape[0]))
```

```
339       *     q.set_qbits(m.indices.tolist(), m.indptr.tolist(), m.data.tolist(),
      ↪  size, state)
340       * ```
341       *
342       * \param row_ind row indices.
343       * \param col_ptr column pointers.
344       * \param value non-zero values.
345       * \param nqbits number of qubits.
346       * \param state representation.
347       * \sa QSystem::get_qbits
348       */
349      void set_qbits(vec_size_t row_ind,
350                     vec_size_t col_ptr,
351                     vec_complex values,
352                          size_t nqbits,
353                     std::string state);
354
355      //! Add ancillary qubits
356      /*!
357       * The ancillaries qubits are added to the end of the system and can be used
      ↪  in
358       * any method.
359       *
360       * \param nqbits number of ancillas added.
361       * \sa QSystem::rm_ancillas
362       */
363      void add_ancillas(size_t nqbits);
364
365      //! Remove all ancillary qubits
366      /*!
367       * If the state is in vector representation the ancillas are measured
368       * before been removed. If the state is in density matrix representation,
369       * the ancillas are removed by a partial trace operation, without been
370       * measured.
371       *
372       * \param nqbits number of ancillas added.
373       * \sa QSystem::rm_ancillas
374       */
375      void rm_ancillas();
376
377    private:
378      /* src/qs_evol.cpp */
379      void            sync();
380      void            sync(size_t qbegin, size_t qend);
381      Gate_aux&       ops(size_t index);
382      arma::sp_cx_mat get_gate(Gate_aux &op);
383      cut_pair        cut(size_t &target, vec_size_t &control);
```

```
384     void            fill(Gate_aux::Tag tag, size_t qbit, size_t size_n);
385
386     /* src/qs_make.cpp */
387     arma::sp_cx_mat make_gate(arma::sp_cx_mat gate, size_t qbit);
388     arma::sp_cx_mat make_cnot(size_t target,
389                               vec_size_t control,
390                                   size_t size_n);
391     arma::sp_cx_mat make_cphase(complex phase,
392                                    size_t target,
393                                  vec_size_t control,
394                                    size_t size_n);
395     arma::sp_cx_mat make_swap(size_t size_n);
396     arma::sp_cx_mat make_qft(size_t size_n);
397
398     /* src/qs_utility.cpp */
399     void            clear();
400
401     /*--------------------*/
402     Gates&           gates;
403     size_t          _size;
404     std::string     _state;
405     Gate_aux*       _ops;
406     bool            _sync;
407     arma::sp_cx_mat qbits;
408     Bit*            _bits;
409
410     size_t          an_size;
411     Gate_aux*       an_ops;
412     Bit*            an_bits;
413
414     inline void     valid_qbit(std::string name, size_t qbit);
415     inline void     valid_count(size_t qbit, size_t count, size_t size_n=1);
416     inline void     valid_control(vec_size_t &control);
417     inline void     valid_phase(complex phase);
418     inline void     valid_swap(size_t qbit_a, size_t qbit_b);
419     inline void     valid_range(size_t qbegin, size_t qend);
420     inline void     valid_gate(char gate);
421     inline void     valid_p(double p);
422     inline void     valid_state();
423     inline void     valid_krau(vec_str &kraus);
424
425 };
426
427 /*************************************************/
428 inline void QSystem::valid_qbit(std::string name, size_t qbit) {
429   if (qbit >= size()) {
430       sstr err;
```

```
431         err << "\'" << name << "\' argument should be in the range of 0 to "
432             << (size()-1);
433         throw std::invalid_argument{err.str()};
434     }
435 }
436 
437 /*******************************************************/
438 inline void QSystem::valid_count(size_t qbit, size_t count, size_t size_n) {
439     if (count == 0 and qbit+count*size_n <= size()) {
440         sstr err;
441         err << "\'cout\' argument should be greater than 0 "
442             << "and \'qbit+count\' suld be in the range of 0 to "
443             << size();
444         throw std::invalid_argument{err.str()};
445     }
446 }
447 
448 /*******************************************************/
449 inline void QSystem::valid_control(vec_size_t &control) {
450     if (control.size() == 0) {
451       sstr err;
452       err << "\'control\' argument must have at least one item";
453       throw std::invalid_argument{err.str()};
454     }
455     for (auto& i : control) {
456       if (i >= size()) {
457         sstr err;
458         err << "Items in \'control\' should be in the range of 0 to "
459             << (size()-1);
460         throw std::invalid_argument{err.str()};
461       }
462     }
463 }
464 
465 /*******************************************************/
466 inline void QSystem::valid_phase(complex phase) {
467     if (std::abs(std::abs(phase) - 1.0) > 1e-14) {
468       sstr err;
469       err << "abs(phase) must be equal to 1";
470       throw std::invalid_argument{err.str()};
471     }
472 }
473 
474 /*******************************************************/
475 inline void QSystem::valid_swap(size_t qbit_a, size_t qbit_b) {
476     if (qbit_a >= size() or qbit_b >= size()) {
477         sstr err;
```

```
478         err << "Arguments \'qbit_a\' and \'qbit_b\' should be in the "
479             << "range of 0 to " << (size()-1);
480         throw std::invalid_argument{err.str()};
481     }
482 }
483
484 /*****************************************************/
485 inline void QSystem::valid_range(size_t qbegin, size_t qend) {
486     if (qbegin >= size() or qend > size() or qbegin >= qend) {
487         sstr err;
488         err << "\'qbegin\' argument should be in the "
489             << "range of 0 to " << (size()-1)
490             << " and argument \'qend\' should be greater than \'qbegin\' "
491             << "and in the range of 1 to " << size();
492         throw std::invalid_argument{err.str()};
493     }
494 }
495
496 /*****************************************************/
497 inline void QSystem::valid_gate(char gate) {
498     if (not (gate == 'X' or gate == 'Y' or gate == 'Z')) {
499       sstr err;
500       err << "\'gate\' argument must be equal to \'X\', \'Y\' or \'X\'";
501       throw std::invalid_argument{err.str()};
502     }
503 }
504
505 /*****************************************************/
506 inline void QSystem::valid_p(double p) {
507     if (p < 0 or p > 1) {
508       sstr err;
509       err << "\'p\' argument should be in the range of 0.0 to 1.0";
510       throw std::invalid_argument{err.str()};
511     }
512 }
513
514 inline void QSystem::valid_state() {
515     if (_state == "vector") {
516       sstr err;
517       err << "\'state\' must be in \"matrix\" to apply this channel";
518       throw std::runtime_error{err.str()};
519     }
520 }
521
522 inline void QSystem::valid_krau(vec_str &kraus) {
523     size_t ksize = kraus[0].size();
524     for (auto& k : kraus) {
```

```
525       if (k.size() != ksize) {
526         sstr err;
527         err << "All \'kraus\' operators must have the same size";
528         throw std::runtime_error{err.str()};
529       }
530     }
531   }
```

D.6.4 using.h

```
1   /* MIT License
2    *
3    * Copyright (c) 2019 Evandro Chagas Ribeiro da Rosa <ev.crr97@gmail.com>
4    * Copyright (c) 2019 Bruno Gouvêa Taketani <b.taketani@ufsc.br>
5    *
6    * Permission is hereby granted, free of charge, to any person obtaining a copy
7    * of this software and associated documentation files (the "Software"), to deal
8    * in the Software without restriction, including without limitation the rights
9    * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
10   * copies of the Software, and to permit persons to whom the Software is
11   * furnished to do so, subject to the following conditions:
12   *
13   * The above copyright notice and this permission notice shall be included in all
14   * copies or substantial portions of the Software.
15   *
16   * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
17   * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
18   * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
19   * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
20   * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
21   * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
22   * SOFTWARE.
23   */
24
25  #pragma once
26  #include <vector>
27  #include <string>
28  #include <complex>
29  #include <utility>
30  #include <sstream>
31  using size_t = long unsigned;
32  using complex = std::complex<double>;
33  using vec_complex = std::vector<std::complex<double>>;
34  using vec_size_t = std::vector<size_t>;
35  using vec_str = std::vector<std::string>;
36  using vec_int = std::vector<int>;
37  using vec_float = std::vector<double>;
```

```
38 using cnot_pair = std::pair<size_t, vec_size_t>;
39 using cph_tuple = std::tuple<complex, size_t, vec_size_t>;
40 using cut_pair = std::pair<size_t, size_t>;
41 using sstr = std::stringstream;
```

D.7 src/

D.7.1 gates.cpp

```
1  /* MIT License
2   *
3   * Copyright (c) 2019 Evandro Chagas Ribeiro da Rosa <ev.crr97@gmail.com>
4   * Copyright (c) 2019 Bruno Gouvêa Taketani <b.taketani@ufsc.br>
5   *
6   * Permission is hereby granted, free of charge, to any person obtaining a copy
7   * of this software and associated documentation files (the "Software"), to deal
8   * in the Software without restriction, including without limitation the rights
9   * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
10  * copies of the Software, and to permit persons to whom the Software is
11  * furnished to do so, subject to the following conditions:
12  *
13  * The above copyright notice and this permission notice shall be included in all
14  * copies or substantial portions of the Software.
15  *
16  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
17  * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
18  * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
19  * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
20  * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
21  * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
22  * SOFTWARE.
23  */
24
25 #include "../header/gates.h"
26 #include "../header/microtar.h"
27
28 using namespace arma;
29
30 /*********************************************************/
31 Gates::Gates() {}
32
33 /*********************************************************/
34 Gates::Gates(std::string path) {
35   mtar_t tar;
36   mtar_header_t h;
37   char *m;
```

```
38
39    int err = mtar_open(&tar, path.c_str(), "r");
40    if (err < 0)
41      throw std::runtime_error{mtar_strerror(err)};
42
43    for (; mtar_read_header(&tar, &h) != MTAR_ENULLRECORD; mtar_next(&tar)) {
44      m = (char*) calloc(1, h.size);
45      mtar_read_data(&tar, m, h.size);
46      std::stringstream ss;
47      ss.write(m, h.size);
48      free(m);
49      sp_cx_mat matrix;
50      matrix.load(ss, arma_binary);
51      mmap[std::string(h.name)] = matrix;
52    }
53
54    mtar_close(&tar);
55  }
56
57  /*************************************************************/
58  sp_cx_mat& Gates::get(char gate) {
59    return map.at(gate);
60  }
61
62  /*************************************************************/
63  sp_cx_mat& Gates::mget(std::string gate) {
64    return mmap.at(gate);
65  }
66
67  /*************************************************************/
68  void Gates::make_gate(char name, vec_complex matrix) {
69    if (matrix.size() != 4) {
70      sstr err;
71      err << "\'matrix\' argument must have exactly 4 elements: "
72          << "[u00, u01, u10, u11]";
73      throw std::invalid_argument{err.str()};
74    }
75    map[name] = sp_cx_mat{cx_mat{{{matrix[0], matrix[1]},
76                                  {matrix[2], matrix[3]}}}};
77  }
78
79  /*************************************************************/
80  void Gates::make_mgate(std::string name,
81                         size_t size,
82                       vec_size_t row,
83                       vec_size_t col,
84                      vec_complex value) {
```

```
85    if (row.size() != col.size()
86        or row.size() != value.size()
87        or col.size() != value.size()) {
88      sstr err;
89      err << "Arguments \'row\', \'col\' and \'value\' must have the same size";
90      throw std::invalid_argument{err.str()};
91    }
92
93    auto sizem = 1ul << size;
94    sp_cx_mat m{sizem, sizem};
95
96    for (size_t i = 0; i < row.size(); i++) {
97      m(row[i], col[i]) = value[i];
98    }
99
100   mmap[name] = m;
101 }
102
103 /*********************************************************/
104 void Gates::make_cgate(std::string name,
105                        std::string gates,
106                         vec_size_t control) {
107   if (control.size() == 0) {
108     sstr err;
109     err << "\'control\' argument must have at least one item";
110     throw std::invalid_argument{err.str()};
111   }
112
113   size_t size = gates.size();
114
115   for (auto& i : control) {
116     if (i >= size) {
117       sstr err;
118       err << "Items in \'control\' should be in the range of 0 to "
119           << (size-1);
120       throw std::invalid_argument{err.str()};
121     }
122   }
123
124   size_t x = 0;
125   size_t z = 0;
126   for (size_t i = 0; i < size; i++) {
127     if (gates[i] == 'X') {
128       x |= 1ul << (size-i-1);
129     } else if (gates[i] == 'Z') {
130       z |= 1ul << (size-i-1);
131     } else if (gates[i] == 'I') {
```

```
132         continue;
133       } else {
134         sstr err;
135         err << "Argument \'gates\' must have only \'X\', \'Z\' and \'I\'";
136         throw std::invalid_argument{err.str()};
137       }
138     }
139
140     auto parity = [&](size_t x) {
141       for (int i = 32; i > 0; i /= 2)
142         x ^= x >> i;
143       return x & 1;
144     };
145
146     sp_cx_mat cm{1ul << size, 1ul << size};
147
148     for (size_t i = 0; i < (1ul << size); i++) {
149       bool cond = true;
150       for (size_t k = 0; k < control.size(); k++)
151         cond = cond and ((1ul << (size-control[k]-1)) & i);
152
153       if (cond) {
154         size_t row = (i ^ x);
155         cm(row, i) = pow(-1, parity(i & z));
156       } else {
157         cm(i,i) = 1;
158       }
159     }
160
161     mmap[name] = cm;
162   }
163
164   /*********************************************************/
165   void Gates::make_fgate(std::string name,
166                          PyObject* func,
167                             size_t size,
168                          PyObject* iterator) {
169
170     sp_cx_mat m{1ul << size, 1ul << size};
171
172     if (iterator == Py_None) {
173       PyObject *builtins = PyEval_GetBuiltins();
174       PyObject *range = PyDict_GetItemString(builtins , "range");
175       iterator = PyEval_CallFunction(range, "i", 1ul << size);
176     }
177
178     auto* it = PyObject_GetIter(iterator);
```

```
179
180   PyObject* pyj;
181   while ((pyj = PyIter_Next(it))) {
182     auto* arg = PyTuple_Pack(1, pyj);
183     auto* pyi = PyObject_CallObject(func,   arg);
184
185     auto i = PyLong_AsSize_t(pyi);
186     auto j = PyLong_AsSize_t(pyj);
187
188     m(i, j) = 1;
189
190     Py_DECREF(pyi);
191     Py_DECREF(pyj);
192   }
193
194   Py_DECREF(it);
195
196   mmap[name] = m;
197 }
198
199 /*********************************************************/
200 std::string Gates::__str__() {
201   std::stringstream out;
202   for (auto& gate: mmap) {
203     out << gate.first << " - "
204         << log2(gate.second.n_rows)  << " qbits long"<< std::endl;
205   }
206   return out.str();
207 }
208
209 /*********************************************************/
210 void Gates::save(std::string path){
211   mtar_t tar;
212   mtar_open(&tar, path.c_str(), "w");
213
214   for (auto &m : mmap) {
215     std::stringstream file;
216     m.second.save(file, arma_binary);
217     file.seekg(0, ios::end);
218     size_t size = file.tellg();
219     file.seekg(0, ios::beg);
220     mtar_write_file_header(&tar, m.first.c_str(), size);
221     mtar_write_data(&tar, file.str().c_str(), size);
222   }
223
224   mtar_finalize(&tar);
225   mtar_close(&tar);
```

```
226 }
```

D.7.2 microtar.c

```
1  /*
2   * Copyright (c) 2017 rxi
3   *
4   * Permission is hereby granted, free of charge, to any person obtaining a copy
5   * of this software and associated documentation files (the "Software"), to
6   * deal in the Software without restriction, including without limitation the
7   * rights to use, copy, modify, merge, publish, distribute, sublicense, and/or
8   * sell copies of the Software, and to permit persons to whom the Software is
9   * furnished to do so, subject to the following conditions:
10  *
11  * The above copyright notice and this permission notice shall be included in
12  * all copies or substantial portions of the Software.
13  *
14  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
15  * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
16  * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
17  * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
18  * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
19  * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
20  * IN THE SOFTWARE.
21  */
22
23 #include <stdio.h>
24 #include <stdlib.h>
25 #include <stddef.h>
26 #include <string.h>
27
28 #include "../header/microtar.h"
29
30 typedef struct {
31   char name[100];
32   char mode[8];
33   char owner[8];
34   char group[8];
35   char size[12];
36   char mtime[12];
37   char checksum[8];
38   char type;
39   char linkname[100];
40   char _padding[255];
41 } mtar_raw_header_t;
42
43
```

```
44 static unsigned round_up(unsigned n, unsigned incr) {
45   return n + (incr - n % incr) % incr;
46 }
47
48
49 static unsigned checksum(const mtar_raw_header_t* rh) {
50   unsigned i;
51   unsigned char *p = (unsigned char*) rh;
52   unsigned res = 256;
53   for (i = 0; i < offsetof(mtar_raw_header_t, checksum); i++) {
54     res += p[i];
55   }
56   for (i = offsetof(mtar_raw_header_t, type); i < sizeof(*rh); i++) {
57     res += p[i];
58   }
59   return res;
60 }
61
62
63 static int tread(mtar_t *tar, void *data, unsigned size) {
64   int err = tar->read(tar, data, size);
65   tar->pos += size;
66   return err;
67 }
68
69
70 static int twrite(mtar_t *tar, const void *data, unsigned size) {
71   int err = tar->write(tar, data, size);
72   tar->pos += size;
73   return err;
74 }
75
76
77 static int write_null_bytes(mtar_t *tar, int n) {
78   int i, err;
79   char nul = '\0';
80   for (i = 0; i < n; i++) {
81     err = twrite(tar, &nul, 1);
82     if (err) {
83       return err;
84     }
85   }
86   return MTAR_ESUCCESS;
87 }
88
89
90 static int raw_to_header(mtar_header_t *h, const mtar_raw_header_t *rh) {
```

```c
  unsigned chksum1, chksum2;

  /* If the checksum starts with a null byte we assume the record is NULL */
  if (*rh->checksum == '\0') {
    return MTAR_ENULLRECORD;
  }

  /* Build and compare checksum */
  chksum1 = checksum(rh);
  sscanf(rh->checksum, "%o", &chksum2);
  if (chksum1 != chksum2) {
    return MTAR_EBADCHKSUM;
  }

  /* Load raw header into header */
  sscanf(rh->mode, "%o", &h->mode);
  sscanf(rh->owner, "%o", &h->owner);
  sscanf(rh->size, "%o", &h->size);
  sscanf(rh->mtime, "%o", &h->mtime);
  h->type = rh->type;
  strcpy(h->name, rh->name);
  strcpy(h->linkname, rh->linkname);

  return MTAR_ESUCCESS;
}


static int header_to_raw(mtar_raw_header_t *rh, const mtar_header_t *h) {
  unsigned chksum;

  /* Load header into raw header */
  memset(rh, 0, sizeof(*rh));
  sprintf(rh->mode, "%o", h->mode);
  sprintf(rh->owner, "%o", h->owner);
  sprintf(rh->size, "%o", h->size);
  sprintf(rh->mtime, "%o", h->mtime);
  rh->type = h->type ? h->type : MTAR_TREG;
  strcpy(rh->name, h->name);
  strcpy(rh->linkname, h->linkname);

  /* Calculate and write checksum */
  chksum = checksum(rh);
  sprintf(rh->checksum, "%06o", chksum);
  rh->checksum[7] = ' ';

  return MTAR_ESUCCESS;
}
```

```
138
139
140 const char* mtar_strerror(int err) {
141   switch (err) {
142     case MTAR_ESUCCESS     : return "success";
143     case MTAR_EFAILURE     : return "failure";
144     case MTAR_EOPENFAIL    : return "could not open";
145     case MTAR_EREADFAIL    : return "could not read";
146     case MTAR_EWRITEFAIL   : return "could not write";
147     case MTAR_ESEEKFAIL    : return "could not seek";
148     case MTAR_EBADCHKSUM   : return "bad checksum";
149     case MTAR_ENULLRECORD  : return "null record";
150     case MTAR_ENOTFOUND    : return "file not found";
151   }
152   return "unknown error";
153 }
154
155
156 static int file_write(mtar_t *tar, const void *data, unsigned size) {
157   unsigned res = fwrite(data, 1, size, tar->stream);
158   return (res == size) ? MTAR_ESUCCESS : MTAR_EWRITEFAIL;
159 }
160
161 static int file_read(mtar_t *tar, void *data, unsigned size) {
162   unsigned res = fread(data, 1, size, tar->stream);
163   return (res == size) ? MTAR_ESUCCESS : MTAR_EREADFAIL;
164 }
165
166 static int file_seek(mtar_t *tar, unsigned offset) {
167   int res = fseek(tar->stream, offset, SEEK_SET);
168   return (res == 0) ? MTAR_ESUCCESS : MTAR_ESEEKFAIL;
169 }
170
171 static int file_close(mtar_t *tar) {
172   fclose(tar->stream);
173   return MTAR_ESUCCESS;
174 }
175
176
177 int mtar_open(mtar_t *tar, const char *filename, const char *mode) {
178   int err;
179   mtar_header_t h;
180
181   /* Init tar struct and functions */
182   memset(tar, 0, sizeof(*tar));
183   tar->write = file_write;
184   tar->read = file_read;
```

```
185   tar->seek = file_seek;
186   tar->close = file_close;
187
188   /* Assure mode is always binary */
189   if ( strchr(mode, 'r') ) mode = "rb";
190   if ( strchr(mode, 'w') ) mode = "wb";
191   if ( strchr(mode, 'a') ) mode = "ab";
192   /* Open file */
193   tar->stream = fopen(filename, mode);
194   if (!tar->stream) {
195     return MTAR_EOPENFAIL;
196   }
197   /* Read first header to check it is valid if mode is `r` */
198   if (*mode == 'r') {
199     err = mtar_read_header(tar, &h);
200     if (err != MTAR_ESUCCESS) {
201       mtar_close(tar);
202       return err;
203     }
204   }
205
206   /* Return ok */
207   return MTAR_ESUCCESS;
208 }
209
210
211 int mtar_close(mtar_t *tar) {
212   return tar->close(tar);
213 }
214
215
216 int mtar_seek(mtar_t *tar, unsigned pos) {
217   int err = tar->seek(tar, pos);
218   tar->pos = pos;
219   return err;
220 }
221
222
223 int mtar_rewind(mtar_t *tar) {
224   tar->remaining_data = 0;
225   tar->last_header = 0;
226   return mtar_seek(tar, 0);
227 }
228
229
230 int mtar_next(mtar_t *tar) {
231   int err, n;
```

```
232   mtar_header_t h;
233   /* Load header */
234   err = mtar_read_header(tar, &h);
235   if (err) {
236     return err;
237   }
238   /* Seek to next record */
239   n = round_up(h.size, 512) + sizeof(mtar_raw_header_t);
240   return mtar_seek(tar, tar->pos + n);
241 }
242
243
244 int mtar_find(mtar_t *tar, const char *name, mtar_header_t *h) {
245   int err;
246   mtar_header_t header;
247   /* Start at beginning */
248   err = mtar_rewind(tar);
249   if (err) {
250     return err;
251   }
252   /* Iterate all files until we hit an error or find the file */
253   while ( (err = mtar_read_header(tar, &header)) == MTAR_ESUCCESS ) {
254     if ( !strcmp(header.name, name) ) {
255       if (h) {
256         *h = header;
257       }
258       return MTAR_ESUCCESS;
259     }
260     mtar_next(tar);
261   }
262   /* Return error */
263   if (err == MTAR_ENULLRECORD) {
264     err = MTAR_ENOTFOUND;
265   }
266   return err;
267 }
268
269
270 int mtar_read_header(mtar_t *tar, mtar_header_t *h) {
271   int err;
272   mtar_raw_header_t rh;
273   /* Save header position */
274   tar->last_header = tar->pos;
275   /* Read raw header */
276   err = tread(tar, &rh, sizeof(rh));
277   if (err) {
278     return err;
```

```
279   }
280   /* Seek back to start of header */
281   err = mtar_seek(tar, tar->last_header);
282   if (err) {
283     return err;
284   }
285   /* Load raw header into header struct and return */
286   return raw_to_header(h, &rh);
287 }
288
289
290 int mtar_read_data(mtar_t *tar, void *ptr, unsigned size) {
291   int err;
292   /* If we have no remaining data then this is the first read, we get the size,
293    * set the remaining data and seek to the beginning of the data */
294   if (tar->remaining_data == 0) {
295     mtar_header_t h;
296     /* Read header */
297     err = mtar_read_header(tar, &h);
298     if (err) {
299       return err;
300     }
301     /* Seek past header and init remaining data */
302     err = mtar_seek(tar, tar->pos + sizeof(mtar_raw_header_t));
303     if (err) {
304       return err;
305     }
306     tar->remaining_data = h.size;
307   }
308   /* Read data */
309   err = tread(tar, ptr, size);
310   if (err) {
311     return err;
312   }
313   tar->remaining_data -= size;
314   /* If there is no remaining data we've finished reading and seek back to the
315    * header */
316   if (tar->remaining_data == 0) {
317     return mtar_seek(tar, tar->last_header);
318   }
319   return MTAR_ESUCCESS;
320 }
321
322
323 int mtar_write_header(mtar_t *tar, const mtar_header_t *h) {
324   mtar_raw_header_t rh;
325   /* Build raw header and write */
```

```
326   header_to_raw(&rh, h);
327   tar->remaining_data = h->size;
328   return twrite(tar, &rh, sizeof(rh));
329 }
330
331
332 int mtar_write_file_header(mtar_t *tar, const char *name, unsigned size) {
333   mtar_header_t h;
334   /* Build header */
335   memset(&h, 0, sizeof(h));
336   strcpy(h.name, name);
337   h.size = size;
338   h.type = MTAR_TREG;
339   h.mode = 0664;
340   /* Write header */
341   return mtar_write_header(tar, &h);
342 }
343
344
345 int mtar_write_dir_header(mtar_t *tar, const char *name) {
346   mtar_header_t h;
347   /* Build header */
348   memset(&h, 0, sizeof(h));
349   strcpy(h.name, name);
350   h.type = MTAR_TDIR;
351   h.mode = 0775;
352   /* Write header */
353   return mtar_write_header(tar, &h);
354 }
355
356
357 int mtar_write_data(mtar_t *tar, const void *data, unsigned size) {
358   int err;
359   /* Write data */
360   err = twrite(tar, data, size);
361   if (err) {
362     return err;
363   }
364   tar->remaining_data -= size;
365   /* Write padding if we've written all the data for this file */
366   if (tar->remaining_data == 0) {
367     return write_null_bytes(tar, round_up(tar->pos, 512) - tar->pos);
368   }
369   return MTAR_ESUCCESS;
370 }
371
372
```

```
373 int mtar_finalize(mtar_t *tar) {
374   /* Write two NULL records */
375   return write_null_bytes(tar, sizeof(mtar_raw_header_t) * 2);
376 }
```

D.7.3 qs_ancillas.cpp

```
1  /* MIT License
2   *
3   * Copyright (c) 2019 Evandro Chagas Ribeiro da Rosa <ev.crr97@gmail.com>
4   * Copyright (c) 2019 Bruno Gouvêa Taketani <b.taketani@ufsc.br>
5   *
6   * Permission is hereby granted, free of charge, to any person obtaining a copy
7   * of this software and associated documentation files (the "Software"), to deal
8   * in the Software without restriction, including without limitation the rights
9   * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
10  * copies of the Software, and to permit persons to whom the Software is
11  * furnished to do so, subject to the following conditions:
12  *
13  * The above copyright notice and this permission notice shall be included in all
14  * copies or substantial portions of the Software.
15  *
16  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
17  * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
18  * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
19  * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
20  * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
21  * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
22  * SOFTWARE.
23  */
24
25 #include "../header/qsystem.h"
26
27 using namespace arma;
28
29 /*******************************************************/
30 void QSystem::add_ancillas(size_t nqbits) {
31   if (nqbits == 0) {
32     throw std::invalid_argument{"\'an_num\' argument must be greater than 0"};
33   } else if (an_size != 0) {
34     sstr err;
35     err << "There are already ancillas in the system, you can not add more";
36     throw std::invalid_argument{err.str()};
37   }
38
39   sync();
40
```

```
41    an_size = nqbits;
42    an_ops = new Gate_aux[an_size]();
43    an_bits = new Bit[an_size]();
44
45    sp_cx_mat an_qbits{1ul << an_size, _state == "matrix" ? 1lu << an_size : 1};
46    an_qbits(0,0) = 1;
47    qbits = kron(qbits, an_qbits);
48  }
49
50  /*****************************************************/
51  void QSystem::rm_ancillas() {
52    if (an_size == 0)
53      throw std::logic_error{"There are no ancillas on the system"};
54    sync();
55
56    auto tr_pure = [&]() {
57      auto sizet = 1ul << (size()-1);
58      sp_cx_mat qbitst{sizet, 1};
59
60      if (an_bits[an_size-1] == NONE)
61        measure(_size+an_size-1);
62
63      for (auto i = qbits.begin(); i != qbits.end(); ++i)
64        qbitst(i.row() >> 1, 0) += *i;
65
66      return qbitst;
67    };
68
69    auto tr_mix = [&]() {
70      auto sizet = 1ul << (size()-1);
71      sp_cx_mat qbitst{sizet, sizet};
72
73      for (auto i = qbits.begin(); i != qbits.end(); ++i) {
74        if ((i.row() % 2) == (i.col() % 2))
75          qbitst(i.row() >> 1, i.col() >> 1) += *i;
76      }
77
78      return qbitst;
79    };
80
81    while (an_size) {
82      if (_state == "vector")
83        qbits = tr_pure();
84      else if (_state == "matrix")
85        qbits = tr_mix();
86      an_size--;
87    }
```

```
88
89    delete[] an_ops;
90    an_ops = nullptr;
91    delete[] an_bits;
92    an_bits = nullptr;
93  }
```

D.7.4 qs_errors.cpp

```
1   /* MIT License
2    *
3    * Copyright (c) 2019 Evandro Chagas Ribeiro da Rosa <ev.crr97@gmail.com>
4    * Copyright (c) 2019 Bruno Gouvêa Taketani <b.taketani@ufsc.br>
5    *
6    * Permission is hereby granted, free of charge, to any person obtaining a copy
7    * of this software and associated documentation files (the "Software"), to deal
8    * in the Software without restriction, including without limitation the rights
9    * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
10   * copies of the Software, and to permit persons to whom the Software is
11   * furnished to do so, subject to the following conditions:
12   *
13   * The above copyright notice and this permission notice shall be included in all
14   * copies or substantial portions of the Software.
15   *
16   * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
17   * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
18   * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
19   * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
20   * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
21   * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
22   * SOFTWARE.
23   */
24
25  #include "../header/qsystem.h"
26
27  using namespace arma;
28
29  /*****************************************************/
30  void QSystem::flip(char gate, size_t qbit, double p) {
31    valid_gate(gate);
32    valid_qbit("qbit", qbit);
33    valid_p(p);
34
35    if (_state == "vector") {
36      if (auto pr = double(std::rand())/double(RAND_MAX); p != 0 and pr <= p)
37        evol(std::string{gate}, qbit);
38
```

```
39    } else if (_state == "matrix") {
40      sync();
41      sp_cx_mat E0 = make_gate(gates.get(gate), qbit)*sqrt(p);
42
43      size_t eyesize = 1ul << size();
44      sp_cx_mat E1 = eye<sp_cx_mat>(eyesize, eyesize)*sqrt(1.f-p);
45
46      qbits = E0*qbits*E0.t() + E1*qbits*E1;
47    }
48  }
49
50  /*****************************************************/
51  void QSystem::amp_damping(size_t qbit, double p) {
52    valid_state();
53    valid_qbit("qbit", qbit);
54    valid_p(p);
55
56    sync();
57
58    sp_cx_mat E0 = make_gate(sp_cx_mat{cx_mat{{{{1, 0}, {0, 0}},
59                                              {{0, 0}, {sqrt(1-p), 0}}}}}, qbit);
60    sp_cx_mat E1 = make_gate(sp_cx_mat{cx_mat{{{{0, 0}, {sqrt(p), 0}},
61                                              {{0, 0}, {0, 0}}}}}, qbit);
62    qbits = E0*qbits*E0 + E1*qbits*E1.t();
63  }
64
65  /*****************************************************/
66  void QSystem::dpl_channel(size_t qbit, double p) {
67    valid_state();
68    valid_qbit("qbit", qbit);
69    valid_p(p);
70
71    sync();
72
73    sp_cx_mat X = make_gate(gates.get('X'), qbit);
74    sp_cx_mat Y = make_gate(gates.get('Y'), qbit);
75    sp_cx_mat Z = make_gate(gates.get('Z'), qbit);
76
77    qbits = (1-p)*qbits+(p/3)*(X*qbits*X+Y*qbits*Y.t()+Z*qbits*Z);
78  }
79
80  /*****************************************************/
81  void QSystem::sum(size_t qbit, vec_str kraus, vec_float p) {
82    valid_state();
83    valid_qbit("qbit", qbit);
84    valid_krau(kraus);
85
```

```
86    sync();
87
88    sp_cx_mat qbits_tmp{qbits.n_rows, qbits.n_cols};
89
90    for (size_t i = 0; i < kraus.size(); ++i) {
91      sp_cx_mat E = gates.get(kraus[i][0]);
92      for (size_t j = 1; j < kraus[i].size(); ++j)
93        E = kron(E, gates.get(kraus[i][j]));
94      E = make_gate(E, qbit);
95
96      qbits_tmp +=  p[i]*(E*qbits*E.t());
97    }
98
99    qbits = qbits_tmp;
100 }
```

D.7.5 qs_evol.cpp

```
1   /* MIT License
2    *
3    * Copyright (c) 2019 Bruno Gouvêa Taketani <b.taketani@ufsc.br>
4    * Copyright (c) 2019 Evandro Chagas Ribeiro da Rosa <ev.crr97@gmail.com>
5    *
6    * Permission is hereby granted, free of charge, to any person obtaining a copy
7    * of this software and associated documentation files (the "Software"), to deal
8    * in the Software without restriction, including without limitation the rights
9    * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
10   * copies of the Software, and to permit persons to whom the Software is
11   * furnished to do so, subject to the following conditions:
12   *
13   * The above copyright notice and this permission notice shall be included in all
14   * copies or substantial portions of the Software.
15   *
16   * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
17   * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
18   * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
19   * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
20   * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
21   * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
22   * SOFTWARE.
23   */
24
25  #include "../header/qsystem.h"
26  #include <algorithm>
27
28  using namespace arma;
29
```

```
30 /*****************************************************/
31 void QSystem::evol(std::string gate,
32                          size_t qbit,
33                          size_t count,
34                            bool inver) {
35   valid_qbit("qbit", qbit);
36
37   if (gate.size() > 1) {
38     auto size_n = log2(gates.mget(gate).n_rows);
39     valid_count(qbit, count, size_n);
40     sync(qbit, qbit+count*size_n);
41     for (size_t i = 0; i < count; i++) {
42       size_t index = qbit+i*gate.size();
43       fill(Gate_aux::GATE_N, index, size_n);
44       ops(index).data = gate;
45       ops(index).inver = inver;
46     }
47   } else {
48     valid_count(qbit, count);
49     sync(qbit, qbit+count);
50     for (size_t i = 0; i < count; i++) {
51       ops(qbit+i).tag = Gate_aux::GATE_1;
52       ops(qbit+i).data = gate[0];
53       ops(qbit+i).inver = inver;
54     }
55   }
56
57   _sync = false;
58 }
59
60 /*****************************************************/
61 void QSystem::cnot(size_t target, vec_size_t control) {
62   valid_qbit("target", target);
63   valid_control(control);
64
65   auto [size_n, minq] = cut(target, control);
66   fill(Gate_aux::CNOT, minq, size_n);
67   ops(minq).data = cnot_pair{target, control};
68 }
69
70 /*****************************************************/
71 void QSystem::cphase(complex phase, size_t target, vec_size_t control) {
72   valid_qbit("target", target);
73   valid_phase(phase);
74   valid_control(control);
75
76   auto [size_n, minq] = cut(target, control);
```

```
77    fill(Gate_aux::CPHASE, minq, size_n);
78    ops(minq).data = cph_tuple{phase, target, control};
79  }
80
81  /*******************************************************/
82  void QSystem::swap(size_t qbit_a, size_t qbit_b) {
83    valid_swap(qbit_a, qbit_b);
84
85    if (qbit_a == qbit_b) return;
86    size_t a = qbit_a < qbit_b? qbit_a :  qbit_b;
87    size_t b = qbit_a > qbit_b? qbit_a :  qbit_b;
88    fill(Gate_aux::SWAP, a, b-a+1);
89  }
90
91  /*******************************************************/
92  void QSystem::qft(size_t qbegin, size_t qend, bool inver) {
93    valid_range(qbegin, qend);
94
95    fill(Gate_aux::QFT, qbegin, qend-qbegin);
96    ops(qbegin).inver = inver;
97  }
98
99  /*******************************************************/
100 void QSystem::sync() {
101   if (_sync) return;
102
103   sp_cx_mat evolm;
104
105   evolm = get_gate(ops(0));
106
107   for (size_t i = ops(0).size; i < size(); i += ops(i).size) {
108     evolm = kron(evolm, get_gate(ops(i)));
109   }
110
111   if (_state == "vector")
112     qbits = evolm*qbits;
113   else if (_state == "matrix")
114     qbits = evolm*qbits*evolm.t();
115
116   delete[] _ops;
117   _ops = new Gate_aux[_size]();
118   if (an_ops) {
119     delete[] an_ops;
120     an_ops = new Gate_aux[an_size]();
121   }
122
123   _sync = true;
```

```
124 }
125
126 /*****************************************************/
127 void QSystem::sync(size_t qbegin, size_t qend) {
128   for (size_t i = qbegin; i < qend; i++) {
129     if (ops(i).busy()) {
130       sync();
131       break;
132     }
133   }
134 }
135
136 /*****************************************************/
137 QSystem::Gate_aux& QSystem::ops(size_t index) {
138   return index < _size? _ops[index] : an_ops[index-_size];
139 }
140
141
142 /*****************************************************/
143 sp_cx_mat QSystem::get_gate(Gate_aux &op) {
144   auto get = [&]() {
145       switch (op.tag) {
146       case Gate_aux::GATE_1:
147         return gates.get(std::get<char>(op.data));
148       case Gate_aux::GATE_N:
149         return gates.mget(std::get<std::string>(op.data));
150       case Gate_aux::CNOT:
151         return make_cnot(std::get<cnot_pair>(op.data).first,
152                          std::get<cnot_pair>(op.data).second,
153                          op.size);
154       case Gate_aux::CPHASE:
155         return make_cphase(std::get<0>(std::get<cph_tuple>(op.data)),
156                            std::get<1>(std::get<cph_tuple>(op.data)),
157                            std::get<2>(std::get<cph_tuple>(op.data)),
158                            op.size);
159       case Gate_aux::SWAP:
160         return make_swap(op.size);
161       default:
162         return make_qft(op.size);
163       }
164   };
165
166   if (op.inver)
167     return get().t();
168   else
169     return get();
170 }
```

```
171
172 /*****************************************************/
173 cut_pair QSystem::cut(size_t &target, vec_size_t &control) {
174   size_t maxq = std::max(target,
175                          *std::max_element(control.begin(),
176                                            control.end()));
177   size_t minq = std::min(target,
178                          *std::min_element(control.begin(),
179                                            control.end()));
180   size_t size_n = maxq-minq+1;
181   for (auto &i : control)
182     i -= minq;
183   target -= minq;
184   return std::make_pair(size_n, minq);
185 }
186
187 /*****************************************************/
188 void QSystem::fill(Gate_aux::Tag tag, size_t qbit, size_t size_n) {
189   sync(qbit, qbit+size_n);
190
191   ops(qbit).tag = tag;
192   ops(qbit).size = size_n;
193
194   for (size_t i = qbit+1; i < qbit+size_n; i++)
195     ops(i).tag = tag;
196
197   _sync = false;
198 }
```

D.7.6 qs_make.cpp

```
1  /* MIT License
2   *
3   * Copyright (c) 2019 Evandro Chagas Ribeiro da Rosa <ev.crr97@gmail.com>
4   * Copyright (c) 2019 Bruno Gouvêa Taketani <b.taketani@ufsc.br>
5   *
6   * Permission is hereby granted, free of charge, to any person obtaining a copy
7   * of this software and associated documentation files (the "Software"), to deal
8   * in the Software without restriction, including without limitation the rights
9   * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
10  * copies of the Software, and to permit persons to whom the Software is
11  * furnished to do so, subject to the following conditions:
12  *
13  * The above copyright notice and this permission notice shall be included in all
14  * copies or substantial portions of the Software.
15  *
16  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
```

```
17   * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
18   * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
19   * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
20   * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
21   * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
22   * SOFTWARE.
23   */
24
25  #include "../header/qsystem.h"
26
27  using namespace arma;
28  using namespace std::complex_literals;
29
30  /******************************************************/
31  sp_cx_mat QSystem::make_gate(sp_cx_mat gate, size_t qbit) {
32    sp_cx_mat m;
33    size_t gate_size = log2(gate.n_rows);
34    if (qbit == 0) {
35      size_t eyesize = 1ul << (size()-gate_size);
36      m = kron(gate, eye<sp_cx_mat>(eyesize, eyesize));
37    } else if (qbit == size()-gate_size) {
38      size_t eyesize = 1ul << (size()-gate_size);
39      m = kron(eye<sp_cx_mat>(eyesize, eyesize), gate);
40    } else {
41      size_t eyesize = 1ul << qbit;
42      m = kron(eye<sp_cx_mat>(eyesize, eyesize), gate);
43      eyesize = 1ul << (size()-qbit-gate_size);
44      m = kron(m, eye<sp_cx_mat>(eyesize, eyesize));
45    }
46    return m;
47  }
48
49  /******************************************************/
50  sp_cx_mat QSystem::make_cnot(size_t target,
51                               vec_size_t control,
52                                   size_t size_n) {
53    sp_cx_mat cnotm{1ul << size_n, 1ul << size_n};
54
55    for (size_t i = 0; i < (1lu << size_n); i++) {
56      bool cond = true;
57
58      for (size_t k = 0; (k < control.size()) and cond; k++)
59        cond = cond and ((1ul << (size_n-control[k]-1)) & i);
60
61      if (cond)
62        cnotm(i, i ^ (1ul  << (size_n-target-1))) = 1;
63      else
```

```
64        cnotm(i, i) = 1;
65    }
66
67    return cnotm;
68  }
69
70  /*******************************************************/
71  sp_cx_mat QSystem::make_cphase(complex phase,
72                                 size_t target,
73                               vec_size_t control,
74                                 size_t size_n) {
75    sp_cx_mat cphasem{1ul << size_n, 1ul << size_n};
76
77    for (size_t i = 0; i < (1lu << size_n); i++) {
78      bool cond = true;
79
80      for (size_t k = 0; (k < control.size()) and cond; k++)
81        cond = cond and ((1ul << (size_n-control[k]-1)) & i);
82
83      cphasem(i, i) = cond and (i & (1ul << (size_n-target-1)))? phase : 1;
84    }
85
86    return cphasem;
87  }
88
89  /*******************************************************/
90  sp_cx_mat QSystem::make_swap(size_t size_n) {
91    sp_cx_mat swapm{1ul << size_n, 1ul << size_n};
92
93    for (size_t i = 0; i < (1ul << (size_n-1)); i++) {
94      if (i%2 == 1)
95        swapm((i | (1ul << (size_n-1))) ^ 1ul, i) = 1;
96      else
97        swapm(i, i) = 1;
98    }
99
100   for (size_t i = 0; i < (1ul << (size_n-1)); i++) {
101     if (i%2 == 0)
102       swapm(i ^ 1ul, i | (1ul << (size_n-1))) = 1;
103     else
104       swapm(i | (1ul << (size_n-1)), i | (1ul << (size_n-1))) = 1;
105
106   }
107
108   return swapm;
109 }
110
```

```
111 /*****************************************************/
112 sp_cx_mat QSystem::make_qft(size_t size_n) {
113   double pi = acos(-1);
114   complex w = std::exp((2*pi*1i)/(pow(2, size_n)));
115   sp_cx_mat qftm{1ul << size_n, 1ul << size_n};
116   for (size_t i = 0; i < (1ul << size_n); i++)
117     for (size_t j = 0; j < (1ul << size_n); j++)
118       qftm(i, j) = (1/sqrt(1 << size_n))*pow(w, i*j);
119   return qftm;
120 }
```

D.7.7 qs_measure.cpp

```
1  /* MIT License
2   *
3   * Copyright (c) 2019 Bruno Gouvêa Taketani <b.taketani@ufsc.br>
4   * Copyright (c) 2019 Evandro Chagas Ribeiro da Rosa <ev.crr97@gmail.com>
5   *
6   * Permission is hereby granted, free of charge, to any person obtaining a copy
7   * of this software and associated documentation files (the "Software"), to deal
8   * in the Software without restriction, including without limitation the rights
9   * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
10  * copies of the Software, and to permit persons to whom the Software is
11  * furnished to do so, subject to the following conditions:
12  *
13  * The above copyright notice and this permission notice shall be included in all
14  * copies or substantial portions of the Software.
15  *
16  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
17  * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
18  * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
19  * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
20  * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
21  * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
22  * SOFTWARE.
23  */
24
25 #include "../header/qsystem.h"
26
27 using namespace arma;
28
29 /*****************************************************/
30 void QSystem::measure(size_t qbit, size_t count) {
31   valid_qbit("qibt", qbit);
32   valid_count(qbit, count);
33
34   sync();
```

```
35    count += qbit;
36    for (; qbit < count; qbit++) {
37      double pm = 0;
38      if (_state == "vector") {
39        for (auto i = qbits.begin(); i != qbits.end(); ++i) {
40          if (~i.row() & 1ul << (size()-qbit-1)) {
41            auto valor = abs((cx_double) *i);
42            pm += valor*valor;
43          }
44        }
45      } else if (_state == "matrix") {
46        sp_cx_mat m_aux = qbits.diag();
47        for (auto i = m_aux.begin(); i != m_aux.end(); ++i)
48          if (~i.row() & 1ul << (size()-qbit-1))
49            pm += (*i).real();
50      }
51
52      auto result = [&](Bit mea, double pm) {
53        auto lnot = mea == ZERO? [](size_t i) { return ~i; }
54                               : [](size_t i) { return i; };
55        if (qbit < _size) _bits[qbit] = mea;
56          else an_bits[qbit-_size] = mea;
57
58        sp_cx_mat qbitsm{1ul << size(),
59                         _state == "vector"? 1 : 1ul << size()};
60
61        if (_state == "vector") {
62          for (auto i = qbits.begin(); i != qbits.end(); ++i)
63            if (lnot(i.row()) & 1ul << (size()-qbit-1))
64              qbitsm(i.row(), 0) = (complex)(*i)/sqrt(pm);
65        } else if (_state == "matrix") {
66          for (auto i = qbits.begin(); i != qbits.end(); ++i)
67            if (lnot(i.row()) & 1ul << (size()-qbit-1)
68                and lnot(i.col()) & 1ul << (size()-qbit-1))
69              qbitsm(i.row(), i.col()) = (complex)(*i)/pm;
70        }
71
72        return qbitsm;
73      };
74
75      if (pm != 0 and double(std::rand()) / double(RAND_MAX) <= pm)
76        qbits = result(ZERO, pm);
77      else
78        qbits = result(ONE, 1.0 - pm);
79    }
80  }
81
```

```
82  /*****************************************************/
83  void QSystem::measure_all() {
84    measure(0, size());
85  }
86
87  /*****************************************************/
88  std::vector<int> QSystem::bits() {
89    std::vector<int> vec;
90    for (size_t i = 0; i < _size; i++)
91      vec.push_back(_bits[i]);
92    for (size_t i = 0; i < an_size; i++)
93      vec.push_back(an_bits[i]);
94    return vec;
95  }
```

D.7.8 qs_utility.cpp

```
1   /* MIT License
2    *
3    * Copyright (c) 2019 Bruno Gouvêa Taketani <b.taketani@ufsc.br>
4    * Copyright (c) 2019 Evandro Chagas Ribeiro da Rosa <ev.crr97@gmail.com>
5    *
6    * Permission is hereby granted, free of charge, to any person obtaining a copy
7    * of this software and associated documentation files (the "Software"), to deal
8    * in the Software without restriction, including without limitation the rights
9    * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
10   * copies of the Software, and to permit persons to whom the Software is
11   * furnished to do so, subject to the following conditions:
12   *
13   * The above copyright notice and this permission notice shall be included in all
14   * copies or substantial portions of the Software.
15   *
16   * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
17   * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
18   * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
19   * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
20   * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
21   * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
22   * SOFTWARE.
23   */
24
25  #include "../header/qsystem.h"
26  #include <iomanip>
27  #include <sstream>
28
29  using namespace arma;
30
```

```
31  /*****************************************************/
32  QSystem::QSystem(size_t nqbits,
33                   Gates& gates,
34                   size_t seed,
35              std::string state) :
36    gates{gates},
37    _size{nqbits},
38    _state{state},
39    _ops{new Gate_aux[nqbits]()},
40    _sync{true},
41    qbits{1lu << nqbits, state == "matrix" ? 1lu << nqbits : 1},
42    _bits{new Bit[nqbits]()},
43    an_size{0},
44    an_ops{nullptr},
45    an_bits{nullptr}
46  {
47    if (state != "matrix" and state != "vector") {
48      sstr err;
49      err << "\'state\' argument must have value "
50          <<  "\"vector\" or \"matrix\", not \""
51          << state << "\"";
52      throw std::invalid_argument{err.str()};
53    }
54    qbits(0,0) = 1;
55    std::srand(seed);
56  }
57
58
59  /*****************************************************/
60  QSystem::~QSystem() {
61    delete[] _ops;
62    delete[] _bits;
63    if (an_ops) delete[] an_ops;
64    if (an_bits) delete[] an_bits;
65  }
66
67  /*****************************************************/
68  QSystem::Gate_aux::Gate_aux() : tag{GATE_1}, data{'I'}, size{1}, inver{false} {}
69
70  /*****************************************************/
71  QSystem::Gate_aux::~Gate_aux() {}
72
73  /*****************************************************/
74  bool QSystem::Gate_aux::busy() {
75    return not(tag == GATE_1 and std::get<char>(data) == 'I');
76  }
77
```

```
78  /*****************************************************/
79  std::string QSystem::__str__() {
80    auto to_bits = [&](size_t i) {
81      std::string sbits{'|'};
82      for (size_t j = 0; j < _size; j++)
83        sbits += i & 1ul << (size()-j-1)? '1' : '0';
84      sbits += an_size == 0? ">" : ">|";
85      for (size_t j = _size; j < size(); j++)
86        sbits += i & 1ul << (size()-j-1)? '1' : '0';
87      sbits += an_size == 0? "" : ">";
88      return sbits;
89    };
90
91    auto cx_to_str = [&](std::complex<double> i) {
92      std::stringstream ss;
93      if (fabs(i.imag()) < 1e-14) {
94        ss << std::showpos << std::fixed
95           << std::setprecision(3)  << i.real() << "       ";
96      } else if (fabs(i.real()) < 1e-14) {
97        ss << std::showpos << std::fixed
98           << std::setprecision(3) << std::setw(12) << i.imag() << 'i';
99      } else {
100       ss << std::showpos << std::fixed << std::setprecision(3) << i.real()
101          << i.imag() << 'i';
102     }
103     return ss.str();
104   };
105
106   sync();
107   std::stringstream out;
108   if (state() == "vector") {
109     for (auto i = qbits.begin(); i != qbits.end(); ++i) {
110       if (abs((cx_double)*i) < 1e-14) continue;
111       out << cx_to_str(*i) << to_bits(i.row()) << '\n';
112     }
113   } else if (state() == "matrix") {
114     for (auto i = qbits.begin(); i != qbits.end(); ++i) {
115       auto aux = cx_to_str(*i);
116       out << "(" << i.row() << ", " << i.col() << ")    " <<
117         (aux == ""? "1" : aux)  << std::endl;
118     }
119   }
120   return out.str();
121 }
122
123 /*****************************************************/
124 size_t QSystem::size() {
```

```
125   return _size+an_size;
126 }
127
128 /*******************************************************/
129 PyObject* QSystem::get_qbits() {
130   sync();
131   qbits.sync();
132
133   PyObject* csc_tuple = PyTuple_New(3);
134   PyObject* val = PyList_New(qbits.n_nonzero);
135   PyObject* row_ind = PyList_New(qbits.n_nonzero);
136   for (size_t i = 0; i < qbits.n_nonzero; i++) {
137     PyList_SetItem(val, i, PyComplex_FromDoubles(qbits.values[i].real(),
138                                                  qbits.values[i].imag()));
139     PyList_SetItem(row_ind, i, PyLong_FromLong(qbits.row_indices[i]));
140   }
141   PyTuple_SetItem(csc_tuple, 0, val);
142   PyTuple_SetItem(csc_tuple, 1, row_ind);
143
144   PyObject* col_ptr = PyList_New(qbits.n_cols+1);
145   for (size_t i = 0; i < qbits.n_cols+1; i++)
146     PyList_SetItem(col_ptr, i, PyLong_FromLong(qbits.col_ptrs[i]));
147   PyTuple_SetItem(csc_tuple, 2, col_ptr);
148
149   PyObject* size_tuple = PyTuple_New(2);
150   PyTuple_SetItem(size_tuple, 0, PyLong_FromLong(qbits.n_rows));
151   PyTuple_SetItem(size_tuple, 1, PyLong_FromLong(qbits.n_cols));
152
153   PyObject* result = PyTuple_New(2);
154
155   PyTuple_SetItem(result, 0, csc_tuple);
156   PyTuple_SetItem(result, 1, size_tuple);
157
158   return result;
159 }
160
161 /*******************************************************/
162 void QSystem::set_qbits(vec_size_t row_ind,
163                         vec_size_t col_ptr,
164                        vec_complex values,
165                             size_t nqbits,
166                        std::string state) {
167   qbits = sp_cx_mat(conv_to<uvec>::from(row_ind),
168                     conv_to<uvec>::from(col_ptr),
169                     cx_vec(values),
170                     1ul << nqbits,
171                     state == "vector"? 1ul : 1ul << nqbits);
```

```
172
173   this->_state = state;
174   _size = nqbits;
175   clear();
176 }
177
178 /*****************************************************/
179 void QSystem::change_to(std::string new_state) {
180   if (new_state != "matrix" and new_state != "vector") {
181     sstr err;
182     err << "\'state\' argument must have value "
183         <<  "\"vector\" or \"matrix\", not \""
184         << new_state << "\"";
185     throw std::invalid_argument{err.str()};
186   }
187
188   if (new_state == _state)
189     return;
190
191   if (new_state == "matrix") {
192     qbits = qbits*qbits.t();
193   } else if (new_state == "vector") {
194     sp_cx_mat nqbits{1ul << size(), 1};
195     for (size_t i = 0; i < 1ul << size(); i++)
196       nqbits(i,0) = sqrt(qbits(i,i).real());
197     qbits = nqbits;
198   }
199
200   _state = new_state;
201 }
202
203 /*****************************************************/
204 std::string QSystem::state() {
205   return _state;
206 }
207
208 /*****************************************************/
209 void QSystem::save(std::string path) {
210   sync();
211   qbits.save(path, arma_binary);
212 }
213
214 void QSystem::load(std::string path) {
215   qbits.load(path, arma_binary);
216   _size = log2(qbits.n_rows);
217   _state = qbits.n_cols > 1 ? "matrix" : "vector";
218   clear();
```

```
219 }
220
221 /*****************************************************/
222 void QSystem::clear() {
223   _sync = true;
224   an_size = 0;
225   if (an_ops) {
226     delete[] an_ops;
227     delete[] an_bits;
228   }
229   an_ops = nullptr;
230   an_bits = nullptr;
231
232   delete[] _ops;
233   delete[] _bits;
234   _ops = new Gate_aux[_size]();
235   _bits = new Bit[_size]();
236 }
```

D.7.9 qsystem.i

```
1  /* MIT License
2   *
3   * Copyright (c) 2019 Evandro Chagas Ribeiro da Rosa <ev.crr97@gmail.com>
4   * Copyright (c) 2019 Bruno Gouvêa Taketani <b.taketani@ufsc.br>
5   *
6   * Permission is hereby granted, free of charge, to any person obtaining a copy
7   * of this software and associated documentation files (the "Software"), to deal
8   * in the Software without restriction, including without limitation the rights
9   * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
10  * copies of the Software, and to permit persons to whom the Software is
11  * furnished to do so, subject to the following conditions:
12  *
13  * The above copyright notice and this permission notice shall be included in all
14  * copies or substantial portions of the Software.
15  *
16  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
17  * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
18  * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
19  * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
20  * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
21  * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
22  * SOFTWARE.
23  */
24
25 %include <std_string.i>
26 %include <std_vector.i>
```

```
27 %include <std_complex.i>
28
29 %inline %{
30 using size_t = long unsigned;
31 %}
32
33 %template(vec_size_t) std::vector<size_t>;
34 %template(vec_complex) std::vector<std::complex<double>>;
35 %template(vec_int) std::vector<int>;
36 %template(vec_float) std::vector<double>;
37 %template(vec_str) std::vector<std::string>;
38
39 %module qsystem
40 %{
41   #include "../header/qsystem.h"
42 %}
43
44 %exception {
45   try {
46     $action
47   } catch(std::exception &e) {
48     SWIG_exception(SWIG_RuntimeError, e.what());
49   }
50 }
51
52 %typemap(out) std::vector<int> QSystem::bits %{
53   $result = PyList_New($1.size());
54   for (size_t i = 0; i < $1.size(); i++) {
55     if ($1[i] == 0) {
56       Py_INCREF(Py_None);
57       PyList_SetItem($result, i, Py_None);
58     } else {
59       PyList_SetItem($result, i, PyLong_FromLong($1[i]-1));
60     }
61   }
62 %}
63
64 %include "../header/qsystem.h"
65 %include "../header/gates.h"
66 %include "../header/using.h"
67
68 %pythoncode %{
69 def get_matrix(q):
70     from scipy import sparse
71     return sparse.csc_matrix(q.get_qbits()[0], q.get_qbits()[1])
72
73 def set_matrix(q, m):
```

```
74      from scipy import sparse
75      from math import log2
76      m = sparse.csc_matrix(m)
77      if m.shape[0] == m.shape[1]:
78          state = 'matrix'
79      else:
80          state = 'vector'
81      size = int(log2(m.shape[0]))
82      q.set_qbits(m.indices.tolist(), m.indptr.tolist(), m.data.tolist(), size,
        ↪  state)
83  %}
```

APÊNDICE E – QSystem Wiki

Wiki disponível em <https://gitlab.com/evandro-crr/qsystem/wikis/home>.

E.1 Home

QSystem

A quantum computing simulator for Python

- Installation
- Basic usage
- Creating gates
- Ancillary qubits
- Quantum error
- Save and load
- Class methods
- Code examples
  - Shor's factoring algorithm

E.2 Instalation

Installation

[Recommended, OS independent] pip

To install QSystem using pip just use the command:

```
pip install QSystem
```

Observation: pip installation requires a C/C++ compiler.

[Linux] Make

Dependencies:

- Armadillo
- SWIG

In order to make the module without installation just clone the repository

```
git clone https://gitlab.com/evandro-crr/qsystem
```

`cd` to `qsystem` and use the command:

```
make PYTHON=/usr/include/python3.Xm
```

then it will be created 2 files `qsystem.py` and `_qsystem.so`, just put this 2 files in the directory of your Python project to use the module QSystem.

E.3 Basic usage

Basic usage

Table of Contents

Classes

The package QSystem has two class:

- the class `Gates` that store all the one qubit gates and the gates created by the user; and
- the class `QSystem` holds the quantum state and controls the computation.

Constructors

The class `Gates` has one optional argument the will be discussed later.

> Class `QSystem` constructor:
>
> ```
> def __init__(self, nqbits, gates, seed=42, state='vector'):
> ```

The class `QSystem` has a constructor with 4 positional arguments, in order, `nqbits`, `gates`, `seed` and `state`. That create a quantum state with `nqbits` qubits, initialized at $|0\rangle$, in the `state` representation (`state` can be `'vector'` for vector representation and `'matrix'` for density matrix representation). The `gates` argument is a instance of the class `Gates`, that holds some quantum gates used in the computation, and the `seed` argument is used to initialize the pseudo-random number generator.

The arfument `seed` and `gates` has the default value, respectively, `42` and `vector`.

Example

```
from qsystem import Gates, QSystem
gates = Gates()
q1 = QSystem(7, gates, 11, 'matrix') # 7 qubits system in dense matrix, seed = 11
q2 = Qsystem(15, gates, 3)           # 15 qubits system in vector state, seed = 3
q3 = Qsystem(9, gates)               # 9 qubits system in vector state, seed = 42
```

Evolution: one qubit gate

The class `Gates` hold all the one quantum gates, by default it has the following gates:

- `'I'` = $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$;
- `'X'` = $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$;
- `'X'` = $\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$;
- `'Z'` = $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$;
- `'H'` = $\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$;

- `'S'` = $\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$; and
- `'T'` = $\begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{bmatrix}$.

To create new gates see the section Creating gates.

> Method `evol` from `QSystem`:
>
> ```
> def evol(self, gate, qbit, count=1, inver=False):
> ```

To apply one of these gates, or any one that you create, in the quantum system, the class `QSystem` has the method `evol` that takes four arguments, `gate`, `qbit`, `count` and `inver`. This method apply the quantum gate `gate` (string referring the gate from the instance of the class 'Gates' used in the constructor), or the inverse quantum gate if `inver` is `True`, from the qubits `qbit` to `qbits+count`, close-open interval.

> The qubits are indexed from 0 to number of qubits-1.

Example

```
from qsystem import Gates, QSystem
gates = Gates()
q = QSystem(7, gates)

q.evol(gate='X', qbit='2') # Apply X on the qubit 2
q.evol('H', 0, count=7)    # Apply H on the 7 qubits
q.evol('S', 4, inver=True) # Apply S inverse on the qubit 4
```

Evolution: mult-qubits gates

The class `QSystem` has four other methods to apply a quantum gate, those methods are: `cnot`, cphase , swap `and` qft`.

> Method `cnot` from `QSystem`:
>
> ```
> def cnot(self, target, control):
> ```
>
> - `target` : qubit index of the target qubit
> - `control` : list of control qubits index

> Method `cphase` from `QSystem`, apply a controlled relative phase gate:
>
> ```
> def cphase(self, phase, target, control):
> ```
>
> - `phase` : phase that will applied
> - `target` : qubit index of the target qubit
> - `control` : list of control qubits index

> Method `swap` from `QSystem`:
>
> ```
> def swap(self, qbit_a, qbit_b):
> ```

Method `qft` from `QSystem`:

```
def qft(self, qbegin, qend, inver):
```

Apply a quantum Fourier transformation, or the inverse QFT if `inver` is `True`, from the qubits `qbegin` to `qend`, close-open interval.

Examples

```
from qsystem import Gates, QSystem
from cmath import exp, pi
gates = Gates()
q = QSystem(7, gates)

q.cnot(0, [2]) # Apply a CNOT, target qubit 0 and control qubit 2
q.cphase(-1, 3, [2]) # Apply a Controlled Z
q.cphase(1j, 5, [1, 4]) # Apply a Controlled S
q.cphase(exp(1j*pi/4), 6, [0]) # Apply a Controlled T
q.swap(3, 5) # Swap the qubits 3 and 7
q.qft(0, 7) # Quantum Fourier transform all the 7 qubits
q.qft(0, 7, True) # Inverse quantum Fourier transform all the 7 qubits
```

Measurement

Methods from `QSystem` to measure qubits in the computational basis:

```
def measure(self, qbit, count=1):
```

Measure from the qubits `qbit` to `qbits+count`, close-open interval.

```
def measure_all(self):
```

Measure all qubits.

The measures from the methods `measure` and `measure_all` are stored in a list accessible from the method `bits`.

Methods from `QSystem`:

```
def bits(self):
```

Return a list of measurements, the n^{th} position of the list correspond to the measurement result of the qubit n. For the qubits that has never been measured it has the value `None`.

Examples

```
from qsystem import Gates, QSystem
gates = Gates()
q = QSystem(3, gates)

q.evol('X', 0)
q.measure(qbit=0, count=2) # Measure the fist two qubits
print('measurement =', q.bits()) # Print the measurement
```

```
measurement = [1, 0, None]
```

Print and others

It's possible to cast a instance of `QSystem` to a `str`, the result is the quantum state.

You can change the system representation during the execution using the method `change_to`.

Method from `QSystem`:

```
def change_to(self, new_state):
```

If `new_state` equal to 'matrix', change from vector representation to density matrix, $|\psi\rangle \rightarrow |\psi\rangle\langle\psi^{\dagger}|$.

If `new_state` equal to `vector`, change from density matrix to vector representation, keeping the measurement probability.

Method from `QSystem`:

```
def size(self):
```

Return the number of qubits in the system.

Example

```
from qsystem import Gates, QSystem
gates = Gates()
q = QSystem(3, gates)

print(q.size(), 'qubits created')

q.evol('H', 0, 2)
q.evol('S', 0)
q.evol('T', 1)

print('Vector state')
print(q)

q.change_to('matrix')

print('Density matrix')
print(q)
```

```
3 qubits created
Vector state
+0.500       |000>
+0.354+0.354i|010>
      +0.500i|100>
-0.354+0.354i|110>

Density matrix
(0, 0)    +0.250
(2, 0)    +0.177+0.177i
(4, 0)          +0.250i
(6, 0)    -0.177+0.177i
(0, 2)    +0.177-0.177i
(2, 2)    +0.250
(4, 2)    +0.177+0.177i
(6, 2)          +0.250i
(0, 4)          -0.250i
(2, 4)    +0.177-0.177i
(4, 4)    +0.250
(6, 4)    +0.177+0.177i
(0, 6)    -0.177-0.177i
(2, 6)          -0.250i
(4, 6)    +0.177-0.177i
(6, 6)    +0.250
```

E.4 Creating gates

Creating gates

Table of content

One qubit gate

Method

```
def make_gate(self, name, matrix):
```

- `name` : name of the gate, must be one character long
- `matrix` : list with the elements of the matrix that represent the gate, in the following order: `[a, b, c, d]` = $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$.

Example

```
from qsystem import Gates, QSystem
from cmath import exp, pi
gates = Gates()

gates.make_gate(name='R', matrix=[1,           0,
                                  0, exp(1j*pi/6)])
q = QSystem(1, gates)
q.evol('X', 0)
q.evol('R', 0)

print(q)
```

```
+0.8660+0.5000i|1>
```

Multiple qubit gate from a sparse matrix

Method

```
def make_mgate(self, name, size, row, col, value):
```

- `name` : name of the gate, must be two or more characters long
- `size` : number of qubits affected
- The matrix of the new gate is `(row[i], col[i]) = value[1]` for `i` in range of `len(row)`. `row`, `col` and `value` must have the same size.

Example

Gate `func` : $|abc\rangle\,|0\rangle \rightarrow |abc\rangle\,|(a \vee b) \wedge c\rangle$

```
from qsystem import Gates, QSystem

def func():
    row, col, value = [], [], []
    for x in range(2**4):
        a = bool(x & 8)
        b = bool(x & 4)
```

```
        c = bool(x & 2)
        d = (a or b) and c
        row.append((x ^ 0) | d)
        col.append(x)
        value.append(1)
    return row, col, value

gates = Gates()
row, col, value = func()
gates.make_mgate('func', 4, row, col, value)

q = QSystem(4, gates)
q.evol('H', 0, 3)
q.evol('func', 0)

print(q)
```

```
+0.354       |0000>
+0.354       |0010>
+0.354       |0100>
+0.354       |0111>
+0.354       |1000>
+0.354       |1011>
+0.354       |1100>
+0.354       |1111>
```

Gate from a python function

Method

```
def make_fgate(self, name, func, size, iterator=None):
```

- `name` : name of the gate, must be two or more characters long
- `func` : Python function that take an `int` as argument and return an `int` , argument and return must be in the range of 0 to $2^{\text{size}} - 1$
- `size` : number of qubits affected by the gate
- `iterator` : iterable with the order of creation of the gate, if `None` iterate on `range(2**size)`

Example

Gate `func` : $|abc\rangle\,|0\rangle \rightarrow |abc\rangle\,|(a \vee b) \wedge c\rangle$

```
from qsystem import Gates, QSystem

def func(x):
    a = bool(x & 8)
    b = bool(x & 4)
    c = bool(x & 2)
    d = (a or b) and c
    return (x ^ 0) | d

def iterator(): # Consider qubit 3 equal to |0>
    for x in range(2**3):
        yield x << 1

gates = Gates()
gates.make_fgate('func', func, 4, iterator())

q = QSystem(4, gates)
q.evol('H', 0, 3)
q.evol('func', 0)

print(q)
```

```
+0.354       |0000>
+0.354       |0010>
+0.354       |0100>
+0.354       |0111>
+0.354       |1000>
+0.354       |1011>
+0.354       |1100>
+0.354       |1111>
```

Controlled gate of `X` and `Z`

Method

```
def make_cgate(self, name, gates, control):
```

- `name` : name of the gate, must be two or more characters long
- `gates` : string with the gates (`'X'` and `'Z'` only) tha will be applied if the control qubits are all equal to $|1\rangle$
- `control` : list of control qubit index

Example

```
from qsystem import Gates, QSystem

gates = Gates()
gates.make_cgate(name='cixz', gates='IXZ', control=[0])

q = QSystem(3, gates)
q.evol('H', 0)
q.evol('H', 2)
q.evol('cixz', 0)

print(q)
```

```
+0.500       |000>
+0.500       |001>
+0.500       |110>
-0.500       |111>
```

E.5 Ancillary qubits

Ancillary qubits

It's possible to add and remove additional (ancillary) qubits during the execution.

Method from `QSystem` :

```
def  add_ancillas(self, nqbits):
```

- `nqbits` : number of ancillarys qubits that will be added

Method from `QSystem` :

```
def rm_ancillas(self):
```

Remove all the ancillas of the system.

The new qubits will be put in the end of the system. Indexes from, the number of qubits, to, the number of qubits + number of ancillary qubits -1.

Removing ancillary qubits from vector state and density matrix has different results. If you remove the ancillas from a density matrix representation, the result will be the partial trace of the ancillas, but in a vector state representation the ancillas will be measured before being removed.

Example

Evolution and measurement

```
from qsystem import Gates, QSystem
g = Gates()
q = QSystem(3, g)
q.evol('H', 0, 3)

q.add_ancillas(2)
q.cnot(3, [0])
q.cnot(3, [1])
q.cnot(4, [0])
q.cnot(4, [2])
q.measure(3, 2)

print(q.bits())
print(q)
```

```
[None, None, None, 0, 0]
+0.707       |000>|00>
+0.707       |111>|00>
```

Different results in vector and density matrix representation

```
from qsystem import Gates, QSystem
g = Gates()
for state in ['vector', 'matrix']:
    q = QSystem(1, g, 44, state)

    q.add_ancillas(1)
    q.evol('H', 0)
    q.cnot(1, [0])

    q.rm_ancillas()
```

```
    q.evol('H', 0)
    q.measure(0)

    print(state, 'representation, measurement =', q.bits()[0])
```

```
vector representation, measurement = 1
matrix representation, measurement = 0
```

E.6 Quantum error

Quantum errors

Table of content:

It's possible to simulate quantum errors during the execution, there are 5 build-in errors, bit flip, phase flip, bit-phase flip, amplitude dumping and depolarization chanel, and you can use a sum operator to simulate another one.

To simulate an error the system needs to be in a density matrix representation.

Bit/Phase/Bit-phase flip

Method from `QSystem`

```
def flip(self, gate, qbit, p):
```

- `gate` : value `'X'` for bit flip, `'Z'` for phase flip and `'Y'` for bit-phase flip
- `qbit` : qubit index of the error
- `p` : probability of the error occur

The method `flip` do a rotation of π, around the `gate` axis, with probability `p` , in the `qbit` .

The effect of those errors in the Bloch sphere can be seen in the following videos:

Amplitude dumping

Method from `QSystem`

```
def amp_damping(self, qbit, p):
```

- `qbit` : qubit index of the error
- `p` : probability of the error occur

The amplitude dumping error takes the `qbit` to the state $|0\rangle$ with probability `p` .

The effect of this error in the Bloch sphere can be seen in the following video:

Depolarization chanel

Method from `QSystem`

```
def dpl_channel(self, qbit, p):
```

- `qbit` : qubit index of the error
- `p` : probability of the error occur

The depolarization channel takes the `qbit` to the maximally mixed state, with probability `p`.

The effect of this error in the Bloch sphere can be seen in the following video:

Sum operator

Method from `QSystem`

```
def sum(qbit, kraus, p):
```

To apply some Kraus operator like

- $E_1 = \sqrt{p_1}(U_{11} \otimes \cdots \otimes U_{1n})$,
- $E_2 = \sqrt{p_2}(U_{21} \otimes \cdots \otimes U_{2n})$
- $\vdots$
- $E_m = \sqrt{p_m}(U_{m1} \otimes \cdots \otimes U_{mn})$,

using the method `sum`, you need to create two list:

- `kraus` = $[U_{11} \otimes \cdots \otimes U_{1n}, U_{21} \otimes \cdots \otimes U_{2n}, \ldots, U_{m1} \otimes \cdots \otimes U_{mn}]$
- `p` = $[p_1, p_2, \ldots, p_m]$

and inform the first `qbit` affected by the operators.

Example

A correlated bit flip in the qubit 0 and 2.

```
from qsystem import Gates, QSystem
g = Gates()
q = QSystem(4, g, 21, 'matrix')

q.sum(0, ['III', 'XIX'], [0.75, 0.25])

print(q)
```

```
(0, 0)    +0.750
(10, 10)    +0.250
```

E.7 Save and load

Save and load

Save and load a quantum state

Method from `QSystem`

```
def save(path):
```

- `path` : path to de file that will be crated

Method from `QSystem`

```
def load( path):
```

- `path` : path to the file that will be loaded

It's possible to save and load the quantum state of a `QSystem` instance in a binary file. The file store the matrix of the system. When the file is loaded all qubits are set as non-ancillary and all measurements are lost.

Save and load gates

Method from `Gates`

```
def save(path):
```

- `path` : path to the file that will be crated

Method from `Gates`

```
def __init__(path):
```

- `path` : optional argument with the path to load a saved `Gates` instance

It's possible to save all mult qubits gate in a file (tar file), and load the gates passing the path of the file to the `Gates` constructor.

E.8 Class methods

QSystem methods

```
def __init__(nqbits, gates, seed=42, state='vector'):
```

- `nqbits` : number of qubits in the system
- `gates` : instance of `Gates`
- `seed` : seed to initialize the pseudo-random number generator
- `state` : string with the value `'vector'` for state vector representation and `'matrix'` for dense matrix representation

```
def evol(gate, qbit, count=1, inver=False):
```

- `gate` : string referring to the gate that will be applied
- `qbit` : index of the qubit that the gate will be applied, the qubits are indexes from 0 to the size of the system -1
- `count` : how many qubits will be effect by the gate starting at the index `qbit`
- `inver` : if `True` apply the inverse gata

```
def cnot(target, control):
```

- `target` : qubit index of the target qubit
- `control` : list of control qubits index

```
def cphase(phase, target, control):
```

- `phase` : Phase that will applied
- `target` : qubit index of the target qubit

```
def swap(qbit_a, qbit_b):
```

- `qbit_a` : index of the qubit that will be swapped
- `qbit_b` : index of the qubit that will be swapped

```
def qft(qbegin, qend, inver=False):
```

- `qbegin` : begin of the interval, closed
- `qend` : and of the interval, open
- `inver` : if `True` apply the inverse QFT

```
def measure(qbit, count=1):
```

- `qbit` : index of the qubit that will be measured, the qubits are indexes from 0 to the size of the system -1
- `count` : how many qubits will be measured, starting at the index `qbit`

```
def measure_all():
```

```
def flip(gate, qbit, p):
```

- `gate` : possible values:
  - `X` : Bit flip
  - `Z` : Phase flip
  - `Y` : Bit-phase flip
- `qbit` : qubit index where the error happen
- `p` : probability of the error happen, between 0 and 1

```
def amp_damping(qbit, p):
```

- `qbit` : qubit index where the error happen
- `p` : probability of the error happen, between 0 and 1

```
def dpl_channel(qbit, p):
```

- `qbit` : qubit index where the error happen
- `p` : probability of the error happen, between 0 and 1

```
def sum(qbit, kraus, p):
```

- `qbit` : First qubit affect by the sum operator
- `kraus` : List of kraus operators, each item is a string with the gates (one qubit gate), all items must have the same size
- `p` : list of probability of each Kraus operator.

```
def size():
```

- Return the number of qubits in the system

```
def state():
```

- Return the system representation

```
def save(path):
```

- `path` : path to de file that will be crated

```
def load( path):
```

- `path` : path to the file that will be loaded

```
def change_to(state):
```

- `state` : value `'vector'` to change to vector representation, keep measurement probability, and value `'matrix'` to change to density matrix representation.

```
def bits():
```

- Return the measurement result

```
def add_ancillas(nqbits):
```

- `nqbits` : number of ancillary qubits to be add

```
def rm_ancillas():
```

Non-member functions

```
def get_matrix(q):
```

- `q` : instance of `QSystem`
- Return a `scipy.sparse.csc_matrix` with the matix of the system

```
def set_matrix(q, m, size, state):
```

- `q` : instance of `QSystem`

- `m` : matrix of the system
- `size` : number of qubits in the system
- `state` : value `vector` for vector representation and `matrix` for density matrix.

Gates methods

```
def __init__(path):
```

- `path` : optional argument with the path to load a saved `Gates` instance

```
def make_gate(name, matrix):
```

- `name` : name of the new quantum gate, only one character
- `matrix` : list of four `complex` with the matrix of the new gate, `[a, b, c, d]` = $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$

```
def make_mgate(name, size, row, col, value):
```

- `name` : name of the new quantum gate, tow of more characters
- `size` : number of qubits affected by the gate
- `row` : list of row indices of the matrix
- `col` : list of column indices of the matrix
- `value` : list of values of the matrix

```
def make_cgate(name, gates, control):
```

- `name` : name of the new quantum gate, tow of more characters
- `gates` : string of gates (one qubit gate) applied when the control qubits are $|1\rangle$
- `control` : list if control qubits index

```
def make_fgate(name, func, size, iterator=None):
```

- `name` : name of the new quantum gate, tow of more characters
- `func` : function that take an `int` and return an `int`
- `size` : number of qubits affected by the gate
- `iterator` : iterable of `int` with the ordem that the gate will be create, if `None` iterate over `range(2**size)`

```
def save(path):
```

- `path` : path to the file that will be crated

APÊNDICE F – Artigo
QSystem: simulador quântico para Python

Evandro Chagas Ribeiro da Rosa[1], **Bruno Gouvêa Taketani**[2]

[1]Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC) – Florianopolis, SC – Brasil

[2] Departamento de Física
Universidade Federal de Santa Catarina (UFSC) – Florianopolis, SC – Brasil

ev.crr97@gmail.com, b.taketani@ufsc.br

Resumo. *Esse trabalho documenta a implementação de um simulador de computação quântica baseado no modelo de circuitos quânticos onde é possível executar uma computação quântica tanto em vetor de estado quanto em matriz densidade, possibilitando, assim, a simulação de erros quânticos. O simulador foi desenvolvido majoritariamente em C++ e entregue como um módulo de Python, denominado QSystem, desta forma, obtém uma boa performance ao mesmo tempo que é dinâmico para o uso.*

1. Introdução

Como apresentado por [Hennessy and Patterson 2017], dos anos 90 até o início dos anos 2000, diversos fatores eram favoráveis para que a taxa de crescimento do poder computacional fosse de 52% ao ano (dobrando, aproximadamente, a cada dois anos). Porém, em 2003, devido a desaceleração da lei de Moore[Moore et al. 1965], ao fim da escalabilidade de Dennard[Dennard et al. 1974] e, ao limite dos processadores multinúcleo, causado pela lei de Amdahl[Amdahl 1967], a taxa de crescimento do poder computacional caiu para 23% ao ano (dobrando, aproximadamente, a cada quatro anos). Contudo, essa taxa continuou caindo, sendo que a prevista para 2018 (ano seguinte a publicação do livro *Computer Architecture: A Quantitative Approach*[Hennessy and Patterson 2017]) foi de 3,5% (dobrando, aproximadamente, a cada vinte e oito anos e meio). Todavia, com o advento dos primeiros computadores quânticos comerciais, a indústria e a academia veem a computação quântica como a porta de entrada para uma nova era de crescimento exponencial do poder computacional.

A possibilidade de utilizar um computador quântico foi inicialmente apresentada por [Feynman 1982], quando ele propôs que um computador quântico poderia simular eficientemente um sistema quântico, uma tarefa difícil para os computadores clássicos (até mesmo para os atuais). Esse, foi um dos primeiros indícios de que um computador quântico pode ter ganho sobre um computador clássico para resolução de alguns problemas. Contudo, a primeira demonstração disto foi feita por citeshor94, quando ele apresentou dois algoritmos quânticos para resolver dois problemas $\mathcal{NP}$ (o logaritmo discreto e a fatoração) que funcionam em tempo polinomial em um computador quântico. Atualmente, já foram propostos vários algoritmos quânticos mais eficientes que seus respectivos algoritmos clássicos, porém, ainda há muito a ser desenvolvido nessa área.

O simulador QSystem foi desenvolvido para auxiliar no estudo de algoritmos, protocolos e códigos quânticos. Por ser baseado no modelo de computação circuital,

um circuito quântico é facilmente transcrito para o simulador. Essa característica faz com que o simulador abstraia para o usuário grande parte da álgebra linear envolvida na computação quântica. Construído para operar tanto em vetor de estado quanto em matriz densidade, o simulador possibilita a simulação de erros quânticos, fazendo que o mesmo possa ser usado no estudo de códigos de identificação e correção de erros quânticos.

Para obter um bom desempenho, o simulador, foi desenvolvido em C++ utilizando a biblioteca de álgebra linear Armadillo[Sanderson and Curtin 2016] e a ferramenta SWIG[Beazley et al. 1996], utilizada para gerar a interface entre o código escrito em C++17 e o interpretador do Python 3. Assim, o simulador QSystem é distribuído como um módulo de Python.

Este artigo é organizado da seguinte forma: os quatro postulados da mecânica quântica são apresentados na Secção 2; o modelo de circuitos quântico, no qual o simulador QSystem é inspirado, é introduzido na Secção 3; em seguida, o simulador e algumas das suas funcionalidades são descritas na Secção 4; um exemplo do algoritmo de Shor utilizando o simulador é mostrado na Secção 5; e, por fim, a conclusão é apresentada na Secção 6.

2. Postulados da mecânica quântica

A computação quântica toma proveito do funcionamento da mecânica quântica para fazer computação, portanto é fundamental conhecer alguns dos seus conceitos.

O livro *Quantum computation and quantum information*[Nielsen and Chuang 2010, Secção 2.2] enumera quatro postulados que descrevem matematicamente o funcionamento da mecânica quântica.

Postulado 1: Associado a cada sistema quântico fechado[1] há um espaço de Hilbert, e o estado do sistema é totalmente representado por um vetor unitário pertencente a esse espaço.

Postulado 2: A evolução de um sistema quântico fechado é descrita pela aplicação de um *operador unitário*, ou seja, a transição de um estado $|\psi\rangle^0$ no tempo t_0 para o estado $|\psi\rangle^1$ no tempo t_1 pode ser totalmente descrita por um operador unitário U, sendo $U|\psi\rangle^0 = |\psi\rangle^1$.

Postulado 3: Uma medida quântica é descrita por um conjunto de *operadores de medida* $\{M_m\}$, onde o índice m indica o possível resultado da medida. Sendo $|\psi\rangle$ o estado logo antes da medida, a probabilidade de medir m é

$$p(m) = \langle\psi| M_m^\dagger M_m |\psi\rangle . \tag{1}$$

E o estado logo após a medida igual a

$$\frac{M_m|\psi\rangle}{\sqrt{\langle\psi| M_m^\dagger M_m |\psi\rangle}}. \tag{2}$$

Os operadores de medida precisam satisfazer a seguinte *equação de completude*

$$\sum_m M_m^\dagger M_m = I, \tag{3}$$

[1]Consideramos um computador quântico como sendo um sistema fechado.

ou de maneira equivalente

$$\sum_m p(m) = \sum_m \langle\psi| M_m^\dagger M_m |\psi\rangle = 1. \tag{4}$$

Postulado 4: O estado de um sistema composto é dado pelo produto tensorial dos seus componentes, ou seja, um sistema composto por n sistemas quânticos, nos estados $|\psi_0\rangle$, $|\psi_1\rangle$, ..., $|\psi_{n-1}\rangle$, tem seu estado total representado por $|\psi_0\rangle \otimes |\psi_1\rangle \otimes \cdots \otimes |\psi_{n-1}\rangle$.

3. Circuitos quântico

Circuitos quântico é um modelo computacional que abstrai o formalismo matemático da mecânica quântica para facilitar a construção e visualização de algoritmos quânticos através de um diagrama de circuitos. Nele, os qubits são representados por linha e, a evolução é feita da esquerda para direita, através da aplicação de portas lógicas quânticas e portas de medida. Logo após uma medida, o qubit passa a ser considerado um bit clássico, sendo representado por duas linhas.

As principais portas lógicas quânticas são apresentadas na Tabela 1 e, a porta de medida na base computacional é representada como na Figura 1. A Figura 2 apresenta um exemplo de circuito quântico.

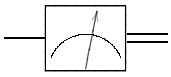


Figura 1. Porta de medida na base computacional.

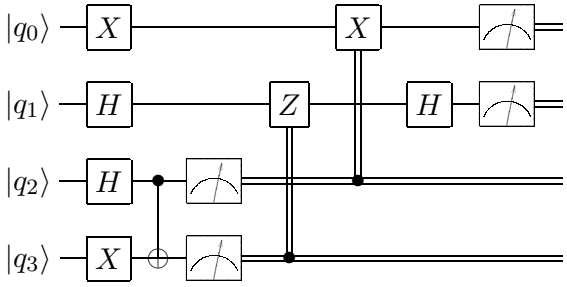


Figura 2. Exemplo de circuito quântico.

4. Simulador QSystem

O simulador foi desenvolvido como um módulo para Python 3, denominado QSystem. A escolha para tal linguagem foi feita tanto por ela ser dinâmica e fácil de usar, quanto por ser atualmente a linguagem mais usada pela comunidade de computação quântica, tendo importantes ferramentas, como QuTiP[Johansson et al. 2012] e Qiskit[Aleksandrowicz et al. 2019], desenvolvidas para ela. Porém, apesar de ter sido construído para Python, por questões de desempenho, o simulador foi implementado em C++17 utilizando a biblioteca de álgebra linear Armadillo[Sanderson and Curtin 2016]. Essa biblioteca foi escolhida, pois traz boa performance, além de possuir funcionalidades similares ao Octave[Eaton et al. 2019], facilitando o porte de protótipos escritos para

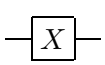
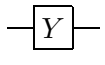
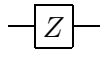
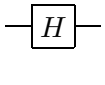
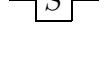
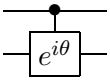
Nome	Circuito	Matriz
X		$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Y		$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Z		$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Hadamard		$\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Fase		$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
T		$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{bmatrix}$
`CNOT`		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
`SWAP`		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
`CPhase`		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\theta} \end{bmatrix}$

Tabela 1. Principais portas lógicas quânticas.

tal ferramenta, e gera um código final mais limpo, aumentando a manutenibilidade do mesmo. Para que as classes desenvolvidas em C++ possam ser instanciadas no Python, foi utilizado a ferramenta SWIG[Beazley et al. 1996], que gera automaticamente a interface entre o que que foi desenvolvido em C++ e o interpretador do Python. A utilização do SWIG possibilita que grande parte do código fonte do simulador não tenha dependência a API[2]do Python.

O módulo QSystem é composto apenas de duas classes, a classe `QSystem`, que armazena e implementa toda a dinâmica de evolução e medida do estado quântico, e a classe `Gates`, que armazena as portas lógicas quânticas de um qubit e as portas de múltiplos qubits criadas pelo usuário.

[2]API: Interface de Programadores de Aplicativos, do inglês *Application Programmers Interface*

O construtor da classe QSystem recebe quatro parâmetros, sendo eles, em sequência: `nqbits`, número de qubits que será criado; `gates`, instância da classe Gates; `seed`, número que inicia o gerador de números pseudo aleatório, com valor padrão = `42`; `state`, *string* indicando qual será a representação do estado quântico, sendo os valores `'vector'` e `'matrix'` referentes a, respectivamente, estar em vetor e matriz densidade, possui valor padrão = `'vector'`. Todos os qubits são inicializados no estado $|0\rangle$.

4.1. Métodos que aplicam portas lógicas quânticas

A aplicação de uma porta lógica de um qubit é feita através do método `QSystem::evol`, que recebe quatro parâmetros, sendo eles, em sequência: `gate`, `char` referente a porta que será aplicada; `qbit`, índice do primeiro qubit no qual a porta será aplicada; `count`, número de qubits afetados, pois a porta `gate` será aplicada nos qubits de índice `qbit` a `qbit+count`, intervalo fechado-aberto, valor padrão = `1`; `inver`, booleano que indica se deve ser aplicada a porta inversa a `gate`, valor padrão = `false`. As portas de um qubits são fornecidas pela instancia da classe `Gates` passada na inicialização, e por padrão, todas as portas de um qubit da Tabela 1 estão pre carregadas.

Utilizando o mesmo método `evol` é possível aplicar portas de múltiplos qubits, a única diferença é que, na chamada do método, o argumento `name` deve possuir mais de um caractere, assim, nesta chamada, os qubits afetados são de `qbit` a `qbit+count`$\times$(tamanho da porta)[3].

O método `QSystem::cnot` é utilizado para aplicar uma porta `CNOT` ou uma porta X com vários qubits de controle. Este método recebe dois argumentos, sendo eles, em ordem: `target`, índice do qubit alvo; e, `control`, lista de índices dos qubits de controle.

O método `QSystem::cphase` aplica uma fase relativa controlada, ou seja, se os qubits de controle estiverem todos em $|1\rangle$, o qubit alvo recebe uma fase relativa. Este método recebe três argumentos, sendo eles, em sequência: `phase`, fase que será aplicada; `target`, índice do qubit alvo; e `control`, lista de índices dos qubits de controle.

O método `QSystem::swap` troca o estado de dois qubits entre si. Este método recebe dois argumentos, `qbit_a` e `qbit_b` referentes aos índices dos qubits que serão trocados. Quando este método é chamado, caso algum item da lista de portas entre os índices `qbit_a` a `qbit_b`, incluindo os mesmos, não estiver livre, o método `sync` é chamado. Logo após, aos mesmo itens, é atribuído a `tag = Gate_aux::SWAP`, e ao primeiro item, o valor `size` igual a $|$`qbit_b-qbit_a`$| + 1$.

Para aplicar uma Transformada de Fourier Quântica é utilizado o método `QSystem::qft`. Este método recebe três parâmetros, sendo eles `qbegin` e `qend`, correspondente ao intervalo, fechado-aberto, de qubits no qual a porta será aplicada; e, `inver`, booleano indicando se deve ou não ser aplicado a Transformada de Fourier Quântica inversa, com valor padrão = `False`, logo, não aplicar.

4.2. Métodos de medida na base computacional

Há dois métodos que efetuam medidas no sistema quântico: o método `QSystem::measure`, que efetua a média em uma sequência de qubits, e o método

[3]Tamanho da porta: número de qubits afetados

`QSystem::measure_all` que efetua a medida de todos os qubits.

O Método `measure` recebe dois argumentos, sendo eles: `qbit`, índice do primeiro qubit que será medido; e, `count`, número de qubits que serão medidos a partir de `qbit`, possui o valor padrão = 1. Já o método `measure_all` não recebe nenhum argumento.

Quando um qubit é medido, seu resultado é armazenado em uma lista acessível pelo método `QSystem::bits`. Cada elemento da lista corresponde a um qubit, sendo o i-ésimo item referente ao qubit i. Os valores possíveis de uma medida são 0 e 1, sendo atribuído o valor `None` aos qubits que não foram medidos. Uma nova medida sobrescreve o resultado da medida anterior.

4.3. Criação de portas lógicas quântica

Além das portas lógicas quânticas já existentes por padrão, é possível criar novas portas, para isso é usado o método `Gates::make_gate`. Este método recebe apenas dois argumentos, sendo eles, em sequência: `name`, `char`, ou uma `str` de um caractere no Python, referente ao nome da nova porta lógica quântica; e, `matrix`, um `std::vector<std::complex<double>>`, ou uma `list` de `complex` no Python, com 4 elementos, representando, em ordem, os elementos a, b, c e d, sendo $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ a matriz da nova porta lógica.

É possível criar uma porta lógica quântica a partir de uma matriz esparsa com o método `Gates::make_mgate`. Este método recebe 5 argumentos, sendo eles, em sequência: `name`, o nome da nova porta lógica que será criada, deve conter dois ou mais caracteres; `size`, o número de qubits que será afetado; e, os vetores `row`, `col` e `value`, que são usados para a construção da matriz esparsa `m`, de tamanho $2^{\texttt{size}} \times 2^{\texttt{size}}$, da forma `m(row[i], col[i]) = value[i]`.

O método `Gates::make_cgate` cria portas controladas de X e Z. Por exemplo, uma porta controlada de $XZZXZI$ que é aplicada apenas quando o qubit 5 estiver no estado $|1\rangle$. Esse tipo de porta é utilizada para a aplicação de estabilizadores em códigos de correção de erro.

É possível aplicar uma função do Python no sistema quântico. Para isso, é necessário criar uma porta lógica quântica utilizando o método `Gates::make_fgate`. Este método recebe 4 argumentos, sendo eles, em sequência: `name`, nome da porta quântica que será criada, deve conter dois ou mais caracteres; `func`, função do Python que será usada para criar a porta lógica quântica, essa função deve receber e retornar um inteiro positivo, e atuar no intervalo de 0 a $2^{\texttt{size}}-1$; `size`, número de qubits afetados; e, `iterator`, objeto Python iterável que retorna números entre 0 e $2^{\texttt{size}}-1$ na sequência em que a porta deve ser criada, esse é um argumento opcional com valor padrão = `None`, ou seja, iterar na sequência de `range(2**size)`.

5. Exemplo de código

O algoritmo de Shor é usado para fatoração não trivial de números inteiros, ou seja, dado um inteiro n, o algoritmo retorna dois números p e q, tal que p e q sejam diferentes de 1 e $pq = n$. Este algoritmo pode ser dividido em três passos, sendo que apenas o segundo passo necessita de um computador quântico. Estes passos são:

1. Selecionar aleatoriamente um número a coprimo[4] a o número n que queremos fatorar.
2. Ache o período r da função $f(x) = a^x \mod n$.
3. Se o período for ímpar, volte ao passo 1, senão, compute os dois fatores p e q tal que

$$p = mdc(a^{\frac{r}{2}} - 1, N) \tag{5}$$

$$q = mdc(a^{\frac{r}{2}} + 1, N) \tag{6}$$

A prova deste algoritmo requer resultados advindos da teoria dos número, que está fora do escopo deste trabalho mas pode ser vista no artigo [Shor 1999].

Com esse algoritmo o problema da fatoração é reduzido ao problema de achar o período de uma função periódica, ou seja, dada uma função periódica $f(x)$, achar o menor número r tal que $f(x) = f(r + x)$. Tal problema pode ser resolvido eficientemente por um computador quântico utilizando o circuito da Figura 3, que efetua a computação nos seguintes passos:

2.1. Inicie dois registradores quânticos, com $s = \lceil \log_2(n + 1) \rceil$ qubits cada, no estado $|0\rangle$.

$$|0\rangle |0\rangle \tag{7}$$

2.2. Aplique a porta de Hadamard em todos os qubits do primeiro registrador para gerar uma superposição.

$$|0\rangle |0\rangle \xrightarrow{H^{\otimes s}} \frac{1}{\sqrt{2^s}} \sum_{x=0}^{2^s-1} |x\rangle |0\rangle \tag{8}$$

2.3. Aplique o operador $U : |x\rangle |0\rangle \rightarrow |x\rangle |a^x \mod n\rangle$, para gerar uma superposição periódica.

$$\frac{1}{\sqrt{2^s}} \sum_{x=0}^{2^s-1} |x\rangle |0\rangle \xrightarrow{\text{U}} \frac{1}{\sqrt{2^s}} \sum_{x=0}^{2^s-1} |x\rangle |a^x \mod n\rangle \tag{9}$$

2.4. Meça e descarte o segundo registrador.

$$\frac{1}{\sqrt{2^s}} \sum_{x=0}^{2^s-1} |x\rangle |a^x \mod n\rangle \rightarrow \sqrt{\frac{r}{2^s}} \sum_{i=0}^{\frac{2^s}{r}-1} |ir + x_0\rangle |a^{x_0} \mod n\rangle \tag{10}$$

(Medindo y no segundo registrador, temos, no primeiro registrador, uma superposição com todos os valores de $ir + x_0$, tal que $f(ir + x_0) = y$ e i é um número inteiro positivo)

2.5. Para encontrar o período presente na superposição do primeiro registrador é necessário aplicar uma Transformada de Fourier Quântica no mesmo.

$$\sqrt{\frac{r}{2^s}} \sum_{i=0}^{\frac{2^s}{r}-1} |ir + x_0\rangle \xrightarrow{\text{QFT}_n} \frac{1}{\sqrt{r}} \sum_{i=0}^{r-1} \left| i\frac{2^s}{r} \right\rangle e^{\phi_i} \tag{11}$$

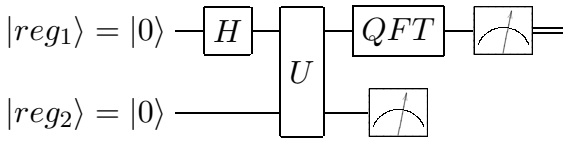


Figura 3. Circuito quântico utilizado para encontrar o período da função $a^x \bmod n$. Os registradores quânticos $|reg_1\rangle$ e $|reg_2\rangle$ devem ser compostos de $\lceil \log_2(n+1) \rceil$ qubits. A porta lógica quântica U mapeá $|x\rangle|0\rangle$ para $|x\rangle|a^x \ mod \, n\rangle$.

2.6. Meça o primeiro registrador e repita o processo para encontrar distintos múltiplos de $\frac{2^s}{r}$, então, encontre o maior divisor comum entre todas as medidas.

Utilizando o simulador QSystem, a implementação em Python do segundo passo do algoritmo de Shor é apresentada a seguir:

```
from qsystem import QSystem, Gates
s = ceil(log2(n+1)) # número de qubits por registrador
# Cria porta logica quântica POWN:|x⟩|0⟩ → |x⟩|a^x mod n⟩
gates = Gates()
def pown(x):
    x = x >> s
    fx = pow(a, x, n)
    return (x << s) | fx
def it():
    for x in range(2**s):
        yield x << s
gates.make_fgate('POWN', pown, 2*s, it())

measurements = []
for _ in range(s):
    seed = randint(210,760)
    # 2.1.
    q = QSystem(s, gates, seed)
    # 2.2.
    q.evol(gate='H', qbit=0, count=s)
    # 2.3.
    q.add_ancillas(s)
    q.evol(gate='POWN', qbit=0)
    # 2.4.
    q.measure(qbit=s, count=s)
    q.rm_ancillas()
    # 2.5.
    q.qft(qbegin=0, qend=s)
    # 2.6.
    q.measure_all()
    c = q.bits()
```

[4] coprimo ou relativamente primo são dois números a e b que possuem 1 como maior divisor comum, ou seja $mdc(a, b) = 1$.

```
32      c = sum([m*2**i for m, i in zip(c,
33                reversed(range(len(c))))])
34      measurements.append(c)
35  c = measurements[0]
36  for m in measurements:
37  c = gcd(c, m)
```

6. Conclusão

Neste trabalho foi desenvolvido um simulador de circuito quântico para Python. O simulador, denominado QSystem, foi implementado como um modulo Python e está disponível no *Python Package Index*, onde pode ser instalado usando a ferramenta pip.

Utilizando a biblioteca Armadillo para C++, o simulador obteve uma boa performance mantendo um código legível e de fácil manutenção. E, usando a ferramenta SWIG, o código escrito em C++ pode ser integrado com o interpretador do Python sem sofrer muitas alterações.

O simulador foi desenvolvido de forma que grande parte da álgebra linear envolvida em uma computação quântica possa ser abstraída pelo usuário. Com ele, é possível instanciar sistemas quânticos com um número arbitrário de qubits, aplicar diversas portas lógicas e simular sistemas tanto em vetor de estado quanto em matriz densidade, que permite a aplicação de erros quânticos.

Como simular um computador quântico toma tempo e espaço exponencial ao número de qubits, estamos limitados a simulação de poucos deles. Em testes, utilizando um computador com as configurações da Tabela 2, foi possível simular até 27 qubits com pouca superposição; e, 14 qubits em superposição total, sendo possível simular um número maior dependendo do circuito. O número total de qubits simulados pode variar dependendo da arquitetura do processador, da versão do compilador e do sistema operacional usado. Esses valores são validos tanto para vetor de estado quanto para matriz densidade. Contudo, a simulação em matriz densidade toma tempo exponencial em relação a simulação em vetor de estado.

Utilizando o QSystem é possível traduzir um circuito quântico para o Python de maneira fácil e direta. Assim, o simulador se torna uma ferramenta dinâmica e de fácil uso para quem quer estudar algoritmos, protocolos e códigos quânticos, dando ênfase na possibilidade de simular códigos identificação e correção de erros quânticos devido ao fato de poder simular erros em estado misto.

CPU	Intel® Core™ i7-4500U
RAM	16 GB
Python	3.7.3
GCC	9.1.0
Linux	Kernel 4.19

Tabela 2. Configuração do computador usado nos testes.

Referências

Aleksandrowicz, G., Alexander, T., Barkoutsos, P., Bello, L., et al. (2019). Qiskit: An open-source framework for quantum computing.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM.

Beazley, D. M. et al. (1996). Swig: An easy to use tool for integrating scripting languages with c and c++. In *Tcl/Tk Workshop*, page 43.

Dennard, R. H., Gaensslen, F. H., Rideout, V. L., Bassous, E., and LeBlanc, A. R. (1974). Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268.

Eaton, J. W., Bateman, D., Hauberg, S., and Wehbring, R. (2019). *GNU Octave version 5.1.0 manual: a high-level interactive language for numerical computations*.

Feynman, R. P. (1982). Simulating physics with computers. *International journal of theoretical physics*, 21(6):467–488.

Hennessy, J. and Patterson, D. (2017). *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science.

Johansson, J. R., Nation, P., and Nori, F. (2012). Qutip: An open-source python framework for the dynamics of open quantum systems. *Computer Physics Communications*, 183(8):1760–1772.

Moore, G. E. et al. (1965). Cramming more components onto integrated circuits.

Nielsen, M. A. and Chuang, I. (2010). *Quantum computation and quantum information*. Cambridge University Press, Cambridge, UK.

Sanderson, C. and Curtin, R. (2016). Armadillo: a template-based c++ library for linear algebra. *Journal of Open Source Software*, 1(2):26–32.

Shor, P. W. (1999). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332.