Computação Quântica com Qiskit: Algoritmos Básicos

Jonas Maziero
Diego Samuel Starke
Lucas Friedrich

Computação Quântica com Qiskit: Algoritmos Básicos

LF EDITORIAL
2026

1ª Edição

Direção editorial: Victor Pereira Marinho e José Roberto Marinho

Capa: Fabrício Ribeiro

Edição revisada segundo o Novo Acordo Ortográfico da Língua Portuguesa

Dados Internacionais de Catalogação na publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Maziero, Jonas
Computação quântica com Qiskit : algoritmos básicos / Jonas Maziero, Diego Samuel Starke, Lucas Friedrich. – 1. ed. – São Paulo, SP: LF Editorial,2026.

ISBN 978-65-5563-720-5

1. Algoritmos de computadores 2. Computação 3. Ciência da computação 4. Framework (Programa de computador) 5. Software - Desenvolvimento I. Starke, Diego Samuel. II. Friedrich, Lucas. III. Título.

26-345090.0 CDD-005.1

Índices para catálogo sistemático:
1. Computação quântica: Processamento de dados: Ciência da computação 005.1

Maria Alice Ferreira - Bibliotecária - CRB-8/7964

LF Editorial
www.livrariadafisica.com.br
www.lfeditorial.com.br
(11) 2648-6666 | Loja do Instituto de Física da USP
(11) 3936-3413 | Editora

Sumário

Capítulo 1

INTRODUÇÃO

A computação quântica surge da interseção entre a física quântica e a ciência da computação, explorando leis fundamentais da natureza para processar informação de maneiras que não têm equivalente clássico. Ao substituir bits por qubits, sistemas quânticos permitem a exploração de superposição, interferência e emaranhamento, abrindo caminho para novos paradigmas computacionais e para algoritmos que, em certos problemas, superam significativamente os métodos clássicos conhecidos NIELSEN; CHUANG (2010); WATROUS (2025).

Nas últimas décadas, avanços experimentais tornaram possível a construção e o acesso remoto a computadores quânticos reais, ainda que limitados e sujeitos a ruídos, os chamados dispositivos da era *Noisy Intermediate-Scale Quantum* (NISQ) CHEN *et al.* (2007). Mesmo nesse regime, já é possível estudar fenômenos fundamentais da informação quântica e implementar algoritmos emblemáticos, tanto por meio de simulações clássicas quanto por execuções em hardware quântico disponível na nuvem.

Este livro tem como objetivo introduzir a computação quântica de forma prática e progressiva, utilizando o framework Qiskit como ferramenta de programação JAVADI-ABHARI *et al.* (2024). O Qiskit é uma biblioteca Python desenvolvida pela IBM para programação, simulação clássica e execução de circuitos quânticos, em última instância, em chips quânticos reais. Ao longo dos capítulos, conceitos teóricos essenciais serão apresentados em conjunto com implementações explícitas de circuitos e algoritmos quânticos, permitindo ao leitor não apenas compreender os princípios matemáticos envolvidos, mas também programar para reproduzir, testar e explorar ativamente os resultados em simuladores clássicos e em computadores quânticos reais.

Nos próximos capítulos, apresentaremos os códigos que permitirão essas execuções e que serão explicados de forma mais detalhada no início e, conforme o livro avança, descreveremos menos a função de cada linha, partindo do pressuposto de que esses conceitos já foram aprendidos anteriormente. Mostraremos a implementação de diversos algoritmos diferentes utilizando o Qiskit. Para isso, realizaremos as implementações considerando como base os notebooks criados por meio do Google Colab, uma plataforma desenvolvida pelo Google que permite programar em Python ou em outras linguagens compatíveis diretamente na nuvem GOOGLE (2026). No Colab, também é possível criar blocos de texto escritos em Markdown com suporte a LaTeX, além de blocos de código destinados à execução dos programas. Embora adotemos o Google Colab como ambiente padrão, não há impedimento para que os códigos sejam executados em outras plataformas, como o Jupyter Notebook, desde que as devidas adaptações sejam realizadas. Além de ser um ambiente em nuvem, o Google Colab executa cada notebook em uma máquina virtual temporária. Dessa forma, alguns pacotes já vêm pré-instalados, enquanto outros precisam ser instalados manualmente. Assim, sempre que uma nova sessão é iniciada, é necessário reinstalar os pacotes que não fazem parte da instalação padrão. O Código 1.1 ilustra como instalar alguns dos pacotes utilizados nos exemplos e experimentos apresentados neste livro.

Como exemplo, considere o Qiskit: ao criarmos um novo notebook no Colab, esse pacote não

Código 1.1: Instalação dos pacotes essenciais para execução e simulação de circuitos quânticos com o Qiskit e verificação da versão instalada. O primeiro bloco realiza as instalações necessárias, incluindo o núcleo do Qiskit, o simulador local (`qiskit_aer`), as bibliotecas de visualização e renderização em LaTeX, e o módulo de integração com o IBM Cloud. O segundo bloco importa o pacote `qiskit` e exibe a versão utilizada no ambiente do Google Colab. Comentado, há também a forma de verificar a versão dos outros pacotes.

```
!pip install qiskit==2.2.3
!pip install qiskit-ibm-runtime==0.43.1
!pip install qiskit_aer==0.17.2
!pip install matplotlib==3.10.0
!pip install pylatexenc==2.10
```

```
import qiskit
qiskit.__version__
#import qiskit_ibm_runtime
#qiskit_ibm_runtime.__version__
#import qiskit_aer
#qiskit_aer.__version__
#import matplotlib
#matplotlib.__version__
#import pylatexenc
#pylatexenc.__version__
```

está disponível por padrão. Portanto, é preciso instalá-lo manualmente utilizando o comando `!pip install qiskit==2.2.3`. Observe que a especificação `qiskit==2.2.3` indica explicitamente que desejamos instalar a versão 2.2.3 do Qiskit. Essa é a versão que utilizamos ao longo de todo o livro. Portanto, para garantir a compatibilidade nas execuções, recomenda-se sempre utilizar essa mesma versão.

Além do pacote principal, também instalamos módulos complementares essenciais para o trabalho com o framework: `qiskit-ibm-runtime`, que permite a comunicação entre o Colab e à IBM Quantum Platform (IBMQ), tornando possível a execução de circuitos em computadores quânticos reais. `qiskit-aer`, responsável pela simulação clássica local de circuitos quânticos. `matplotlib`, utilizado para a criação de gráficos e visualizações de circuitos quânticos. `pylatexenc`, que possibilita a renderização de expressões matemáticas escritas em LaTeX. Por fim, o segundo bloco do Código 1.1 realiza a importação do pacote `qiskit` e exibe a versão instalada, garantindo que o ambiente esteja configurado corretamente para a reprodução dos experimentos apresentados neste livro. Como comentário nesse bloco (os comentários em Python são iniciados por #), há também a possibilidade de verificar as versões dos outros pacotes removendo #. Em caso de erro na execução de um código, recomendamos, como primeira etapa de diagnóstico, verificar se as versões indicadas neste livro estão corretamente instaladas.

Para cada protocolo ou algoritmo, recomendamos a criação de um novo notebook no Google Colab. Após a instalação dos pacotes necessários, no Código 1.2 apresentamos a importação de algumas das bibliotecas essenciais utilizadas ao longo deste livro. Inicialmente, importamos o pacote numpy por meio do comando `import numpy as np`. O NumPy é amplamente empregado em cálculos numéricos, especialmente na manipulação de vetores e matrizes, tanto reais quanto complexas, que são fundamentais para a representação de estados quânticos. A sigla np funciona como um *alias*, permitindo utilizar o pacote de forma abreviada ao longo do código.

Em seguida, importamos o módulo `math` utilizando o comando `import math`. Esse módulo complementa as funcionalidades do NumPy ao fornecer funções matemáticas adicionais, como funções trigonométricas, exponenciais e constantes importantes.

Também importamos a classe `QuantumCircuit`, proveniente do pacote `qiskit`. Essa classe é o elemento central do framework, pois permite criar, modificar e visualizar circuitos quânticos, servindo de base para o desenvolvimento, teste e simulação dos algoritmos que serão apresentados nos capítulos seguintes. Introduzimos também as classes `QuantumRegister` e `ClassicalRegister` que, respectivamente, permitem incluir nos circuitos quânticos os registros quânticos (qubits) e os registros clássicos para o armazenamento dos resultados das medições feitas nos qubits.

Por fim, a última linha deste bloco de código especifica o número de execuções (`nshots`) realizadas para uma determinada configuração experimental, denominadas *shots*. Em geral, `nshots = 2**12 = 4096 shots` é um valor suficiente para produzir um conjunto de amostras estatisticamente representativo, permitindo a obtenção de estimativas confiáveis dos resultados. Eventualmente, precisaremos aumentar essa quantidade, mas deixaremos indicado quando precisarmos alterar esse valor.

Código 1.2: Importação das bibliotecas necessárias para a construção de circuitos quânticos. O pacote `numpy` é utilizado para operações numéricas e manipulação de vetores e matrizes, o módulo `math` fornece funções matemáticas complementares e o `QuantumCircuit`, do pacote `qiskit`, é empregado na criação e manipulação de circuitos quânticos.

```
import numpy as np
import math
from qiskit import QuantumCircuit,  QuantumRegister, ClassicalRegister
nshots = 2**12 # 4096
```

Agora, vamos configurar o ambiente onde os circuitos quânticos serão executados. Inicialmente, no Código 1.3, o objeto `AerSimulator` é instanciado como *backend*, representando um simulador clássico que reproduz o comportamento de um computador quântico real de forma local, permitindo testar e validar algoritmos sem a necessidade de acesso ao hardware físico. Em seguida, o `SamplerV2`, proveniente do módulo `qiskit-ibm-runtime`, é criado e configurado para utilizar esse *backend* como modo de execução para realizar as operações e coletar as amostras resultantes das medições dos qubits, retornando distribuições de probabilidades associadas aos estados quânticos finais.

Código 1.3: Inicialização do simulador e do `Sampler` (amostrador, em tradução livre) para execução de circuitos quânticos. O objeto `AerSimulator` é utilizado como **backend** para simulações clássicas locais de circuitos quânticos, enquanto o `SamplerV2`, proveniente do módulo `qiskit-ibm-runtime`, é configurado para utilizar esse *backend* como modo de execução e será o responsável por obter as amostragens dos resultados de medições.

```
from qiskit_aer import AerSimulator
backend = AerSimulator()
from qiskit_ibm_runtime import SamplerV2 as Sampler
sampler = Sampler(mode=backend)
```

Embora simuladores clássicos do comportamento de computadores quânticos se tornem rapidamente ineficientes à medida que o número de qubits em um circuito aumenta, mesmo quando executados em máquinas computacionalmente potentes, eles continuam sendo ferramentas fundamentais para o estudo, desenvolvimento e validação de algoritmos quânticos, bem como para o desenvolvimento de novas arquiteturas de circuitos, testes de protocolos de comunicação quântica, avaliação de estratégias de correção de erros e o estudo dos fundamentos da computação quântica. As simulações clássicas permitem testar circuitos, ajustar parâmetros e realizar análises estatísticas de forma controlada, garantindo que os experimentos estejam corretos e otimizados antes da execução em dispositivos quânticos reais, cuja utilização pode envolver restrições de acesso, tempo de fila e custos financeiros significativos.

No Código 1.4 é realizada a configuração e a inicialização do serviço `QiskitRuntimeService`, responsável por autenticar o usuário e estabelecer a conexão entre o ambiente local e a IBMQ. Inicialmente, o método `save_account()` é utilizado para registrar localmente as credenciais do usuário, incluindo o *token* de acesso e o identificador da instância na IBM Cloud, permitindo que o sistema reconheça e autentique automaticamente futuras sessões. Os parâmetros `set_as_default=True` e `overwrite=True` asseguram, respectivamente, que essa conta seja definida como padrão e que eventuais configurações anteriores sejam substituídas. Em seguida, a variável `service` é instanciada por meio de `QiskitRuntimeService()`, possibilitando o acesso aos *backends* quânticos disponíveis e a execução de experimentos no ambiente de computação quântica da IBM.

As credenciais apresentadas neste livro correspondem às que foram utilizadas na execução dos experimentos em hardware quântico real. Entretanto, elas não estarão ativas nem acessíveis ao leitor, de modo que não poderão ser reutilizadas para autenticação. Consequentemente, também não será possível recuperar os dados experimentais associados, por se tratarem de informações de uso pessoal. Sendo assim, é necessário que o leitor crie uma conta pessoal no IBM Cloud para acessar suas próprias credenciais do IBMQ e executar seus próprios experimentos. Para isso, basta acessar https://quantum.cloud.ibm.com/, clicar em "Conectar" e será possível encontrar formas de criar uma conta gratuita. Atualmente, essa conta oferece recursos limitados, correspondentes a 10 minutos de tempo de uso em hardware quântico, os quais são renovados mensalmente. Todavia, destacamos que todos os experimentos realizados neste livro são factíveis de serem executados com esses recursos gratuitos.

A Plataforma é relativamente simples de utilizar e, sendo assim, também será fácil criar a "API key" para substituir no *token* e a criação de uma instância aberta (*open instance*) para a obtenção do "CRN". É por meio da instância aberta que será possível acessar os recursos gratuitos disponibilizados pela IBM. Com essas informações, você já terá acesso a sua própria conta e será capaz de executar e replicar todo o conteúdo desenvolvido aqui.

Código 1.4: Configuração do serviço `QiskitRuntimeService` para autenticação e conexão com a IBMQ. O método `save_account()` armazena as credenciais do usuário e define a conta padrão para acesso aos *backends* quânticos disponíveis por meio da variável `service` para estabelecer a conexão ativa com o serviço.

```
from qiskit_ibm_runtime import QiskitRuntimeService
service = QiskitRuntimeService.save_account(
channel="ibm_cloud",
token="dEcCJbqCJY7J3U2fRkiPmKH-C0j5fYX7iQS5oRUnQRPA",
instance="crn:v1:bluemix:public:quantum-computing:us-east:a/9143f48a0f5
↪  e47cea3728e5a5c4a67c4:71b71129-5c4c-45f6-a990-5af3bdc337df::",
set_as_default=True,overwrite=True)
service = QiskitRuntimeService()
```

No Código 1.5 é apresentada a seleção do *backend* quântico utilizado para a execução experimental dos circuitos. O nome do dispositivo, definido pela variável `backend_name`, é passado como argumento para o método `service.backend()`, o qual instancia o *backend* correspondente por meio do serviço `QiskitRuntimeService`. É importante ressaltar que você deve verificar previamente, na instância configurada no ambiente da IBMQ, quais computadores quânticos estão disponíveis para sua conta, uma vez que a disponibilidade dos *backends* pode variar ao longo do tempo. No momento da elaboração deste livro, o dispositivo `ibm_torino` encontrava-se disponível e foi utilizado nas execuções.

Código 1.5: Seleção do *backend* quântico `ibm_torino` a partir do serviço `QiskitRuntimeService`. O código armazena o nome do *backend* na variável `backend_name` e o instancia por meio do método `service.backend()`, permitindo que o ambiente utilize esse dispositivo específico para execução dos circuitos quânticos como `backend_exp`.

```
backend_name = "ibm_torino"
backend_exp = service.backend(backend_name)
```

No Código 1.6 é apresentada a etapa de transpilação do circuito quântico, uma das fases mais importantes do Qiskit para o aprimoramento dos resultados experimentais. Iniciamos a importação do transpilador do `qiskit`. De forma geral, a transpilação consiste na conversão de um circuito quântico abstrato para uma forma equivalente que seja compatível com o hardware específico selecionado no *backend*. Esse processo adapta o circuito às restrições físicas do dispositivo, tais como a conectividade entre qubits, o conjunto de portas disponíveis e a profundidade máxima permitida. O parâmetro `optimization_level=1`, que assume valores inteiros entre 0 e 3, controla quanto tempo de processamento clássico o transpilador gasta na otimização do circuito.

O Código 1.7 configura opções avançadas de execução para o estimador quântico no ambiente `Qiskit Runtime`. Inicialmente, importa-se a classe `EstimatorOptions` do módulo

Código 1.6: Código para a transpilação do circuito quântico ideal para as restrições impostas pelo chip quântico real.

```
from qiskit import transpile
qc_transpiled = transpile(qc_list, backend=backend_exp,
↪  optimization_level=1)
```

`qiskit_ibm_runtime`, que permite ajustar parâmetros relacionados ao desempenho e à mitigação de ruído durante a execução em dispositivos reais. Em seguida, cria-se uma instância `options = EstimatorOptions()` para armazenar essas configurações. As linhas seguintes ativam e especificam o tipo de desacoplamento dinâmico (*dynamical decoupling*), uma técnica utilizada para reduzir os efeitos da decoerência e do ruído ambiental nos qubits. O comando `options.dynamical_decoupling.enable = True` habilita essa funcionalidade, enquanto `options.dynamical_decoupling.sequence_type = "XY4"` define a sequência de pulsos a ser utilizada, neste caso, a sequência *XY4*, amplamente empregada ao alternar rotações nos eixos X e Y da esfera de Bloch, de modo a cancelar erros acumulativos de fase. Essa configuração melhora a fidelidade das operações quânticas, especialmente em execuções que envolvem tempos de coerência longos. Ademais, o Código 1.8 é semelhante à discussão feita acima. A diferença é que aqui estamos utilizando o *Sampler* no lugar do *Estimator*.

Código 1.7: Código para o desacoplamento dinâmico com o *Estimator*.

```
from qiskit_ibm_runtime import EstimatorOptions
options = EstimatorOptions()
options.dynamical_decoupling.enable = True
options.dynamical_decoupling.sequence_type = "XY4"
options.default_shots = nshots
```

Código 1.8: Código para o desacoplamento dinâmico com o *Sampler*.

```
from qiskit_ibm_runtime import SamplerV2 as Sampler
sampler = Sampler(mode=backend_exp)
sampler.options.dynamical_decoupling.enable = True
sampler.options.dynamical_decoupling.sequence_type = "XY4"
```

No Código 1.9, realiza-se a importação e a inicialização do objeto `EstimatorV2`, proveniente do módulo `qiskit_aer.primitives` para a simulação clássica e do módulo `qiskit_ibm_runtime` para o experimento no chip quântico real. Esse componente faz parte das primitivas do Qiskit e é responsável pelo cálculo dos valores esperados de observáveis quânticos a partir dos estados produzidos por circuitos definidos. O estimador permite avaliar quantidades da forma $\langle\psi|O|\psi\rangle$, em que $|\psi\rangle$ representa o estado final do circuito e O o observável de interesse. Essa ferramenta é essencial para análises de correlação e verificação de efeitos quânticos, como a violação da Desigualdade de Bell.

Nos capítulos seguintes, depois de introduzirmos as principais ferramentas do Qiskit no Capítulo 2, passaremos a apresentar alguns dos algoritmos básicos da computação quântica. No Capítulo 3, discutiremos o algoritmo de Deutsch e sua extensão, o algoritmo de Deutsch-Jozsa. Esse algoritmo foi uma das primeiras indicações de que a computação quântica poderia oferecer vantagens sobre a computação clássica. Na sequência, no Capítulo 4, descreveremos o algoritmo de Grover, que é um algoritmo quântico com ganho quadrático em relação aos algoritmos clássicos para um problema de aplicação prática recorrente, que é a busca em bancos de dados não estruturados. Nos Capítulos seguintes, veremos modificações do problema de Deutsch que foram inventadas com o objetivo de aumentar a separação clássico-quântica em complexidade computacional, que é constante para o algoritmo de Deutsch-Jozsa em relação a algoritmos clássicos probabilísticos. Os algoritmos de Bernstein-Vazirani e de Simon, apresentados nos Capítulos 5 e 6, respectivamente, fornecem ganhos super-polinomial e exponencial em relação a quaisquer algoritmos clássicos. Nos Capítulos 7 e 8, descreveremos, respectivamente, a transformada de Fourier quântica e o algoritmo para estimativa de fase quântica, que são duas sub-rotinas essenciais para o algoritmo de Shor e para outros algoritmos da computação quântica

Código 1.9: Importação e inicialização do objeto `EstimatorV2`. Essa classe implementa o estimador quântico responsável pelo cálculo dos valores esperados de observáveis a partir de circuitos definidos no Qiskit. O bloco de cima é referente à simulação clássica e o bloco inferior é referente ao experimento no chip quântica real.

```
from qiskit_aer.primitives import EstimatorV2 as Estimator
estimator = Estimator()
```

```
from qiskit_ibm_runtime import EstimatorV2 as Estimator
estimator = Estimator(mode=backend_exp, options=options)
```

que não serão tratados neste livro. Por fim, depois de introduzir alguns conceitos e provas de teoria de números no Capítulo 9, apresentaremos o algoritmo de Shor no Capítulo 10. Este algoritmo continua sendo uma das aplicações mais importantes da computação quântica até hoje, pois fornece um ganho exponencial e resolve um problema que tem implicações revolucionárias no sistema de criptografia clássica que usamos atualmente.

Capítulo 2

INTRODUÇÃO AO QISKIT

Neste capítulo, examinaremos aspectos fundamentais do Qiskit por meio de exemplos. Serão tratados conceitos essenciais de informação quântica, como a Desigualdade de Bell-CHSH, a Codificação Superdensa e o Teletransporte Quântico. Esses tópicos servirão para ilustrar a aplicação prática de recursos do Qiskit, como *Sampler*, *Estimator* e Circuitos Dinâmicos, bem como a criação e execução de circuitos quânticos. Nosso propósito é oferecer uma introdução breve e simular esses protocolos em computadores quânticos. Nos capítulos seguintes, retomaremos parte dessas ferramentas ao abordarmos algoritmos quânticos.

2.1 Violação da desigualdade de Bell-CHSH com o *Estimator*

A desigualdade de BELL (1964) foi concebida em um contexto de realismo local, envolvendo a notória discussão levantada em EINSTEIN; PODOLSKY; ROSEN (1935), denominada paradoxo EPR. O que a desigualdade de Bell demonstrou teoricamente foi que as variáveis ocultas não conseguem descrever a correlação envolvendo um par emaranhado, uma vez que a desigualdade apresentada foi violada no cenário quântico. Posteriormente, em CLAUSER; HORNE; SHIMONY; HOLT (1969) foi apresentada uma análise teórica mais simplificada da desigualdade de Bell, conhecida como desigualdade CHSH, que viabilizou verificações experimentais.

Experimentos subsequentes que testaram a desigualdade CHSH, aliados a avanços decisivos na consolidação da ciência da informação quântica, levaram ao reconhecimento das contribuições de John Clauser, Alain Aspect e Anton Zeilinger, agraciados com o Prêmio Nobel de Física de 2022 (https://www.nobelprize.org/prizes/physics/2022/press-release/).

Consideremos o cenário bipartido padrão de Bell-CHSH, no qual dois observadores espacialmente separados, Alice e Bob, escolhem independentemente entre duas configurações de medição cada um. Nesse contexto, vamos definir um observável da seguinte forma

$$O = A_1(B_1 + B_2) + A_2(B_1 - B_2), \tag{2.1}$$

onde cada um dos observáveis envolvidos na definição de O pode assumir apenas dois valores: $A_j, B_k = \pm 1,\ \forall j, k = 1, 2$. Em um modelo determinístico de variáveis ocultas locais, assume-se que os resultados são funções de uma variável oculta λ, de modo que $A_j = a_j(\lambda)$ e $B_k = b_k(\lambda)$ em que $a_j(\lambda), b_k(\lambda) \in \{\pm 1\}$. O valor médio do observável para uma distribuição normalizada $\rho(\lambda)$ sobre o espaço de λ é expresso como:

$$\langle O \rangle_{dl} = \int d\lambda \rho(\lambda) \left\{ a_1(\lambda)[b_1(\lambda) + b_2(\lambda)] + a_2(\lambda)[b_1(\lambda) - b_2(\lambda)] \right\}. \tag{2.2}$$

uma vez que o valor médio de um observável qualquer é obtido por meio de $\langle O\rangle = \int d\lambda\, \rho(\lambda)\, O(\lambda)$ e surge devido à inacessibilidade da variável oculta λ. Por outro lado, a hipótese de localidade impõe que as escolhas de medição de Alice não dependam das escolhas de medição de Bob e vice-versa. Sendo assim, os valores possíveis de O são ± 2. As condições físicas de **determinismo e localidade** (*dl*), também denominadas teoria de variáveis ocultas locais, levam à desigualdade de CHSH: $-2 \leq \langle O\rangle_{dl} \leq 2$.

Considerando os observáveis quânticos definidos como $A_1 = Z,\ A_2 = X,\ B_1 = \frac{1}{\sqrt{2}}(Z + X),\ B_2 = \frac{1}{\sqrt{2}}(Z - X)$, e substituindo na definição de O da Eq. (2.1), obtemos

$$O = \sqrt{2}(Z \otimes Z + X \otimes X). \tag{2.3}$$

A matriz de Pauli $X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ atua na base computacional $\{|0\rangle, |1\rangle\} = \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}$, também denominada base padrão, como segue:

$$X|0\rangle = |1\rangle,\ X|1\rangle = |0\rangle. \tag{2.4}$$

Ou seja, a porta lógica quântica X é equivalente a uma operação de inversão de bit (análogo quântico ao NOT clássico). Já a matriz de Pauli $Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$ atua na base computacional da seguinte forma:

$$Z|0\rangle = |0\rangle,\ Z|1\rangle = -|1\rangle. \tag{2.5}$$

Ou seja, Z introduz uma inversão de fase no estado $|1\rangle$.

Para o estado Bell de dois sistemas quânticos de dois níveis (qubits), dado por $|\Psi_-\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)$ (estado singleto), obtém-se da atuação do primeiro termo do observável da Eq. (2.3) a seguinte relação: $Z \otimes Z|\Psi_-\rangle_{AB} = \frac{1}{\sqrt{2}}\left(|0\rangle_A \otimes (-|1\rangle_B) - (-|1\rangle_A) \otimes |0\rangle_B\right) = -|\Psi_-\rangle_{AB}$. Do segundo termo do observável, temos que $X \otimes X|\Psi_-\rangle_{AB} = -|\Psi_-\rangle_{AB}$. Portanto, o valor médio do observável na Mecânica Quântica é dado por:

$$\begin{aligned} \langle O\rangle_{\text{mq}} &= \langle\Psi_-|\, O\, |\Psi_-\rangle = \sqrt{2}\,\langle\Psi_-|\,(Z \otimes Z + X \otimes X)\,|\Psi_-\rangle \\ &= \sqrt{2}\,\left(\langle\Psi_-|\, Z \otimes Z\, |\Psi_-\rangle + \langle\Psi_-|\, X \otimes X\, |\Psi_-\rangle\right) = \sqrt{2}\,[(-1) + (-1)] = -2\sqrt{2}, \end{aligned} \tag{2.6}$$

em que usamos a propriedade $(A \otimes B)(C \otimes D) = AC \otimes BD$. Consequentemente, a previsão da Mecânica Quântica viola a restrição imposta pela teoria de variáveis ocultas locais.

Cabe destacar que, nas passagens anteriores e ao longo deste livro, adotamos a equivalência entre as seguintes notações: $|\psi\rangle_A \otimes |\phi\rangle_B \equiv |\psi\rangle \otimes |\phi\rangle \equiv |\psi\rangle_A|\phi\rangle_B \equiv |\psi\rangle|\phi\rangle \equiv |\psi_A\phi_B\rangle, \equiv |\psi\phi\rangle$. Com frequência, omitiremos o símbolo $\otimes$ que representa o produto tensorial, bem como os índices nos estados.

2.1.1 Executando no Qiskit

Agora, vamos implementar os observáveis que utilizamos na formulação quântica e verificá-los nos computadores quânticos por meio do Qiskit. Primeiramente, devemos iniciar um novo notebook no Google Colab e adicionar no início os Códigos 1.1 e 1.2. Essa informação não será passada novamente ao longo deste livro, sendo assim, fique atento para incluir esses dois códigos sempre no início de cada notebook.

Seguindo, vamos adicionar o código apresentado abaixo, cuja finalidade é gerar o circuito quântico que produz a base de Bell $|\Psi_-\rangle$ em computadores quânticos:

```
qc = QuantumCircuit(2)
qc.h(0); qc.cx(0,1); qc.x(1); qc.z(0)
qc.draw('mpl')
```

Nesse código, o comando `QuantumCircuit(2)` cria um circuito com dois qubits. Um aspecto relevante nos computadores quânticos da IBM é que todos os qubits são inicializados no estado padrão $|0\rangle$. Portanto, qualquer outro estado precisa ser preparado por meio da aplicação de portas lógicas quânticas com base nesse estado inicial. O comando `qc.h(0)` aplica a porta de Hadamard ao qubit 0 (denotado como q_0 na Figura 2.1). Em seguida, o comando `qc.cx(0,1)` aplica uma porta CNOT, usando o qubit q_0 como controle e o qubit 1 (denotado como q_1 na Figura 2.1) como alvo. O comando `qc.x(1)` aplica a porta de Pauli X ao qubit q_1. Já o comando `qc.z(0)` aplica a porta de Pauli Z ao qubit q_0. Por fim, o comando `qc.draw('mpl')` é utilizado para gerar a representação gráfica do circuito quântico apresentada na Figura 2.1. Esse comando sempre pode ser utilizado para apresentar o último circuito quântico gerado no notebook. Vale notar que o tempo cresce da esquerda para a direita do circuito quântico e as operações unitárias (no caso, H, CNOT, X e Z) são executadas sequencialmente.

O comando `qc.h(0)` aplica a porta de Hadamard, que é definida como $H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$ e atua na base computacional da seguinte forma:

$$H|0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} =: |+\rangle\,, \quad H|1\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} =: |-\rangle\,. \tag{2.7}$$

O comando `qc.cx(0,1)` aplica a porta *CNOT* (ou *controlled-NOT*, negação controlada em tradução livre), que denotamos como $C_X^{A_1\to B}$ e é representada por:

$$C_X^{A_1\to B} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} = |0\rangle_A\langle 0| \otimes \mathbb{I}_B + |1\rangle_A\langle 1| \otimes X_B \;\Rightarrow\; \begin{cases} C_X^{A_1\to B}|00\rangle = |00\rangle, \\ C_X^{A_1\to B}|01\rangle = |01\rangle, \\ C_X^{A_1\to B}|10\rangle = |11\rangle, \\ C_X^{A_1\to B}|11\rangle = |10\rangle. \end{cases} \tag{2.8}$$

A porta CNOT, denotada acima como $C_X^{A_1\to B}$, inverte o segundo qubit alvo (*target*, identificado por B, qubit da direita na notação de braket) somente quando o primeiro qubit de controle (*control*, identificado por A_1, qubit da esquerda) encontra-se no estado $|1\rangle$. O subscrito em A_1 em $C_X^{A_1\to B}$ indica o estado de ativação associado ao qubit de controle na operação controlada, ou seja, a operação X somente ocorre no qubit alvo se o estado do qubit de controle for $|1\rangle$. Para simplificar a notação, escreveremos $C_X^{A_1\to B} = C_X^{A\to B}$, omitindo o subscrito de A, assim, subentende-se que o estado de ativação do qubit de controle é $|1\rangle$. Eventualmente, iremos, inclusive, omitir a notação $A \to B$. Por outro lado, denotaremos de maneira mais explícita quando o estado de ativação for o $|0\rangle$. O símbolo ⊕ no qubit alvo da CNOT refere-se à porta de Pauli X definida na Eq. (2.4), enquanto o símbolo ● refere-se à notação de controle com estado de ativação $|1\rangle$. Depois, veremos que o símbolo ○ refere-se à notação de controle com estado de ativação $|0\rangle$. Em geral, a porta X da CNOT pode ser substituída por qualquer porta unitária U de um qubit e será denotada por $C_U^{A\to B}$, sendo assim, o efeito sobre o qubit alvo passa a corresponder à ação dessa nova unitária U. Acima, temos que $|0\rangle\langle 0| = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$ e $|1\rangle\langle 1| = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$.

No Qiskit, os observáveis são definidos a partir da importação da classe `Pauli` do módulo chamado `qiskit.quantum_info`, que é responsável por representar, de forma simbólica e manipulável, operadores de Pauli individuais ou compostos dentro do ambiente Qiskit. A partir disso, construímos os operadores `ZZ` e `XX` presentes na Eq. (2.3), que correspondem, respectivamente, aos produtos tensoriais $Z_1 \otimes Z_0$ e $X_1 \otimes X_0$, e são implementados conforme o código a seguir:

```
from qiskit.quantum_info import Pauli
ZZ = Pauli('ZZ') # Z_1 Z_0
XX = Pauli('XX') # X_1 X_0
observables = [ZZ,XX]
```

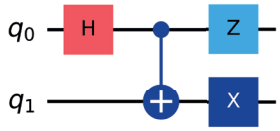

Figura 2.1: Circuito quântico responsável por produzir o estado da base de Bell $|\Psi_-\rangle$.

2.1.2 Simulação clássica

Vamos iniciar os testes por meio da simulação clássica de um computador quântico. Devemos incluir o primeiro bloco do Código 1.9. As linhas de código a seguir apresentam o estimador quântico (*Estimator*) para a obtenção dos valores esperados dos observáveis definidos. Formamos uma lista do par circuito quântico e observável que é passado para o método `estimator.run()`. A extração dos valores esperados é realizada por meio de `job.result()[0].data.evs` e armazenada na lista `values`. O comando `print(values)` exibe a lista com dois objetos, cada um contendo o valor esperado calculado para um observável. Vamos ao código.

```
job = estimator.run([(qc, observables)])
values = job.result()[0].data.evs
print(values) # Output: [array(-1.), array(-1.)]
```

Por fim, calculamos o valor médio do observável quântico apresentado na Eq. (2.3) por meio do seguinte código.

```
O_avg_sim = math.sqrt(2)*(values[0]+values[1])
O_avg_sim # Output: np.float64(-2.8284271247461903)
```

2.1.3 Experimento no chip quântico

Para executar o experimento no chip quântico real, é necessário carregar suas credenciais do IBMQ por meio do Código 1.4. Lembrando que aqui você deve utilizar suas credenciais, conforme a discussão feita sobre esses códigos no capítulo anterior. Devemos também escolher um dos chips quânticos reais por meio do Código 1.5. Aqui deve-se atentar para o nome de um dos *backends* disponíveis na sua instância. Vamos incluir o Código 1.7 para otimizar a execução e, por fim, o segundo bloco do Código 1.9.

Em seguida, importamos a função `generate_preset_pass_manager` que está presente no módulo `qiskit.transpiler.preset_passmanagers`. Essa função cria um gerenciador de passes, isto é, o componente que organiza e executa a sequência de transformações aplicadas ao circuito durante a transpilação, de acordo com o nível de otimização escolhido. No nosso caso, utilizamos o nível máximo de otimização disponível no Qiskit, buscando reduzir a profundidade do circuito e o número total de portas quânticas, o que é fundamental para minimizar erros de decoerência e das portas em execuções reais. O comando `pass_manager.run(qc)` aplica o processo de transpilação ao circuito `qc`, produzindo uma nova versão otimizada denominada `qc_transpiled`. Essa versão considera as conexões físicas e o conjunto de portas nativas do *backend* selecionado, tornando o circuito compatível com o hardware alvo. Por fim, a linha `operators_transpiled_list` ajusta os observáveis originalmente definidos para refletir o novo mapeamento de qubits resultante da transpilação. O método `apply_layout()` aplica a correspondência entre os qubits lógicos do circuito original e os qubits físicos do dispositivo, garantindo que as medições dos operadores ocorram nos qubits corretos após a otimização. O código fica da seguinte forma:

```
from qiskit.transpiler.preset_passmanagers import
↪  generate_preset_pass_manager
pass_manager = generate_preset_pass_manager(optimization_level=3,
↪  backend=backend_exp)
qc_transpiled = pass_manager.run(qc)
operators_transpiled_list = [op.apply_layout(qc_transpiled.layout) for
↪  op in observables]
```

Vamos, agora, utilizar o circuito quântico e os observáveis transpilados para rodar o experimento:

```
estimator = Estimator(mode=backend_exp, options=options)
job = estimator.run([(qc_transpiled, operators_transpiled_list)])
job_id = job.job_id(); print(job_id) # Output: d5n2k09h2mqc739bgd20
```

O *output* do código acima, `d5n2k09h2mqc739bgd20`, é o identificador do experimento (ID). Sendo assim, por meio do ID do experimento executado em sua conta pessoal, que certamente será diferente do apresentado acima, é possível recuperar os resultados desse experimento. Por conta disso, é interessante armazenar esses identificadores para acesso futuro. Um ponto que deve ser considerado é que essas informações podem ser excluídas do armazenamento em nuvem, em geral mediante aviso prévio das empresas responsáveis. Assim, caso esses dados sejam importantes, recomenda-se atenção para baixá-los e mantê-los acessíveis localmente. Por meio do Qiskit, recuperamos os dados deste experimento utilizando o ID obtido por meio do seguinte código:

```
job_id = 'd5n2k09h2mqc739bgd20'; job = service.job(job_id)
```

Armazenamos os resultados na variável `values` e extraímos os resultados para cada um dos observáveis:

```
values = job.result()[0].data.evs
values # Output: array([-0.9960968 , -0.99547229])
```

Por fim, calculamos o valor médio do observável definido inicialmente na Eq. (2.3), utilizando os valores obtidos nos experimentos do código acima:

```
O_avg_exp = math.sqrt(2)*(values[0]+values[1])
O_avg_exp # Output: np.float64(-2.816504012744918)
```

Abaixo, apresentamos um código que pode ser utilizado em qualquer experimento com chip real e serve para cancelar uma determinada execução. Em determinado momento, o chip quântico pode ficar bastante sobrecarregado de requisições, e pode surgir o interesse em mudar de *backend*. Tudo o que precisamos é do ID do experimento para ser incluído no seguinte código:

```
job = service.job(job_id)
job.cancel()
```

A seguir, apresentamos um comparativo da desigualdade CHSH no contexto de teorias de variáveis ocultas e na formulação da mecânica quântica, incluindo o resultado teórico, a simulação clássica e o resultado experimental obtido no computador quântico:

$$|\langle O\rangle_{CHSH}| \leq 2, \quad |\langle O\rangle_{mq}| = 2\sqrt{2} \approx 2.8284, \tag{2.9}$$

$$|\langle O\rangle_{sim}| \approx 2.8284, \quad |\langle O\rangle_{exp}| \approx 2.8165. \tag{2.10}$$

O limite de Tsirelson diz que o valor do observável CHSH não pode ser maior que $2\sqrt{2}$. Entretanto, em computadores quânticos mais antigos e nas versões anteriores do Qiskit, sob condições de otimização menos eficientes, já observamos violações desse limite, o que ocorre por diversas razões. Em primeiro lugar, como realizamos apenas um número finito de *shots* (preparações e medições), surgem flutuações estatísticas significativas. Além disso, efeitos de decoerência e erros de preparação, portas e medição também contribuem para que, em certos casos, os resultados obtidos não sejam físicos.

2.2 Codificação Superdensa com o *Sampler*

O protocolo de codificação superdensa (CSD), proposto em BENNETT; WIESNER (1992), é uma forma de **enviar 2 cbits (bits clássicos) enviando "somente" 1 qubit**, com alguns detalhes a serem discutidos a seguir. Este protocolo é, no mínimo, instigante, pois, em informação quântica, existe o limite de Holevo, que estabelece que podemos extrair no máximo n bits de informação clássica de um sistema de n qubits WILDE (2016).

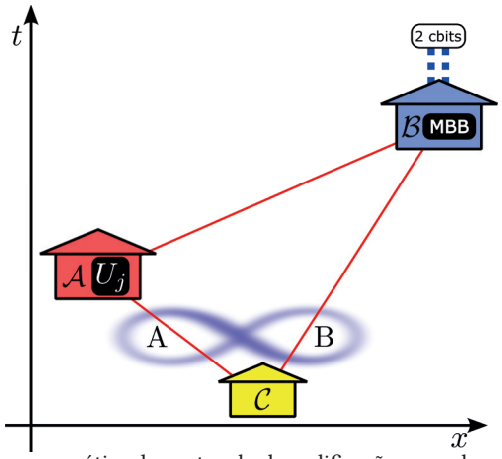

Figura 2.2: Representação esquemática do protocolo de codificação superdensa. Charlie ($\mathcal{C}$) prepara um par de qubits maximamente emaranhado e envia o qubit A para Alice ($\mathcal{A}$) e o qubit B para Bob ($\mathcal{B}$). Alice aplica uma operação unitária local (U_j) em seu qubit, para codificar dois cbits, e envia o qubit a Bob, que então realiza uma medição na base de Bell (MBB) e recupera os 2 cbits codificados por Alice.

O protocolo de codificação superdensa funciona da seguinte forma (ver a Figura 2.2). Charlie ($\mathcal{C}$) prepara um par de qubits maximamente emaranhados e envia um qubit para Alice ($\mathcal{A}$) e outro para Bob ($\mathcal{B}$), que estão em laboratórios diferentes e desejam trocar bits clássicos de informação. O par emaranhado que escolhemos para realizar o protocolo é dado por $|\Phi_+\rangle = \frac{1}{\sqrt{2}}\big(|00\rangle_{AB} + |11\rangle_{AB}\big) = \frac{1}{\sqrt{2}}\big(|00\rangle + |11\rangle\big)$. O qubit da Alice é o da esquerda (subíndice A), e o do Bob é o da direita (subíndice B). Por simplicidade na notação, omitimos os subíndices A e B, conforme apresentado na última igualdade. A ordem dos qubits só é alterada mediante aviso prévio em caso de necessidade ou reidentificados por subíndices.

Dois bits clássicos podem assumir quatro valores distintos. Esses quatro valores formam a base do espaço clássico de duas posições binárias e têm uma correspondência natural com os quatro estados da base computacional de dois qubits $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$. A seguir, Alice percebe o seguinte efeito das **operações locais** no estado compartilhado:

$$(\mathbb{I}\otimes\mathbb{I})|\Phi_+\rangle = |\Phi_+\rangle, \tag{2.11}$$

$$(Z\otimes\mathbb{I})|\Phi_+\rangle = (Z\otimes\mathbb{I})\frac{1}{\sqrt{2}}\big(|00\rangle + |11\rangle\big) = \frac{1}{\sqrt{2}}\big(|00\rangle - |11\rangle\big) = |\Phi_-\rangle, \tag{2.12}$$

$$(X\otimes\mathbb{I})|\Phi_+\rangle = (X\otimes\mathbb{I})\frac{1}{\sqrt{2}}\big(|00\rangle + |11\rangle\big) = \frac{1}{\sqrt{2}}\big(|10\rangle + |01\rangle\big) = |\Psi_+\rangle, \tag{2.13}$$

$$(ZX\otimes\mathbb{I})|\Phi_+\rangle = (ZX\otimes\mathbb{I})\frac{1}{\sqrt{2}}\big(|00\rangle + |11\rangle\big) = \frac{1}{\sqrt{2}}\big(-|10\rangle + |01\rangle\big) = |\Psi_-\rangle, \tag{2.14}$$

em que usamos as operações das portas de Pauli X e Z, apresentadas nas Eqs. (2.4) e (2.5), na base computacional.

Sendo assim, Alice consegue preparar os quatro estados ortogonais (completamente distinguíveis) da base de Bell usando somente operações locais. Então, conforme a sequência de bits de informação que Alice quer enviar para Bob, ela realiza uma das seguintes operações para a **codificação** da informação:

$$00 \;\to\; \mathbb{I} \;\to |\Phi_+\rangle =: |B_{00}\rangle, \quad 01 \;\to\; X \;\to |\Psi_+\rangle =: |B_{01}\rangle, \tag{2.15}$$

$$10 \;\to\; Z \;\to |\Phi_-\rangle =: |B_{10}\rangle, \quad 11 \;\to\; ZX \;\to |\Psi_-\rangle =: |B_{11}\rangle. \tag{2.16}$$

Claro, Alice e Bob devem ter combinado essa codificação de antemão para a implementação do protocolo.

Depois de feita a escolha da sequência de bits clássicos a ser enviada e de aplicar a operação local correspondente, Alice envia seu qubit para o laboratório de Bob. Uma vez recebido o qubit de Alice, Bob, em posse dos dois qubits, precisa fazer a **decodificação** da informação clássica transmitida. Ele realiza esse procedimento por meio de uma **medição na base de Bell** (MBB), a qual projeta o sistema em um dos quatro estados dessa base: $\{|\Phi_+\rangle, |\Psi_+\rangle, |\Phi_-\rangle, |\Psi_-\rangle\}$.

No IBMQ, a medição está restrita à base computacional. Para dois qubits, isso corresponde ao conjunto $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$. Assim, para realizar uma medição na base de Bell, é necessário implementar, previamente, uma mudança de base. Esse procedimento consiste em aplicar, inicialmente, uma porta CNOT entre os dois qubits e, em seguida, uma porta de Hadamard no qubit de Bob. Essa sequência de operações transforma os estados da base de Bell em estados da base computacional, permitindo que a medição realizada no IBMQ reproduza efetivamente uma medição na base de Bell. Matematicamente, temos

$$(H\otimes\mathbb{I})\,C_X^{A\to B}\,|\Phi_+\rangle = (H\otimes\mathbb{I})\frac{1}{\sqrt{2}}\big(C_X^{A\to B}|00\rangle + C_X^{A\to B}|11\rangle\big) = (H\otimes\mathbb{I})\frac{1}{\sqrt{2}}\big(|00\rangle + |10\rangle\big) \tag{2.17}$$

$$= (H\otimes\mathbb{I})\left(\frac{|0\rangle+|1\rangle}{\sqrt{2}}\otimes|0\rangle\right) = |0\rangle\otimes|0\rangle = |00\rangle,$$

$$(H\otimes\mathbb{I})\,C_X^{A\to B}\,|\Psi_+\rangle = (H\otimes\mathbb{I})\frac{1}{\sqrt{2}}\big(C_X^{A\to B}|01\rangle + C_X^{A\to B}|10\rangle\big) = (H\otimes\mathbb{I})\frac{1}{\sqrt{2}}\big(|01\rangle + |11\rangle\big) \tag{2.18}$$

$$= (H\otimes\mathbb{I})\left(\frac{|0\rangle+|1\rangle}{\sqrt{2}}\otimes|1\rangle\right) = |0\rangle\otimes|1\rangle = |01\rangle,$$

$$(H\otimes\mathbb{I})\,C_X^{A\to B}\,|\Phi_-\rangle = (H\otimes\mathbb{I})\frac{1}{\sqrt{2}}\big(C_X^{A\to B}|00\rangle - C_X^{A\to B}|11\rangle\big) = (H\otimes\mathbb{I})\frac{1}{\sqrt{2}}\big(|00\rangle - |10\rangle\big) \tag{2.19}$$

$$= (H\otimes\mathbb{I})\left(\frac{|0\rangle-|1\rangle}{\sqrt{2}}\otimes|0\rangle\right) = |1\rangle\otimes|0\rangle = |10\rangle,$$

$$(H\otimes\mathbb{I})\,C_X^{A\to B}\,|\Psi_-\rangle = (H\otimes\mathbb{I})\frac{1}{\sqrt{2}}\big(C_X^{A\to B}|01\rangle - C_X^{A\to B}|10\rangle\big) = (H\otimes\mathbb{I})\frac{1}{\sqrt{2}}\big(|01\rangle - |11\rangle\big) \tag{2.20}$$

$$= (H\otimes\mathbb{I})\left(\frac{|0\rangle-|1\rangle}{\sqrt{2}}\otimes|1\rangle\right) = |1\rangle\otimes|1\rangle = |11\rangle.$$

Sendo assim, mapeamos a base de Bell em estados da base computacional de dois qubits $|\Phi_+\rangle \mapsto |00\rangle$, $|\Psi_+\rangle \mapsto |01\rangle$, $|\Phi_-\rangle \mapsto |10\rangle$, $|\Psi_-\rangle \mapsto |11\rangle$.

Então, uma vez que Bob aplique $C_X^{A\to B}$, seguido de $H\otimes\mathbb{I}$, e a medição na base computacional, ele aprende os dois bits de informação clássica que Alice gostaria de comunicar a ele.

Vamos agora simular o protocolo de codificação superdensa no IBMQ. Como vimos, um terceiro elemento, Charlie, precisa preparar um estado emaranhado de dois qubits e enviar um qubit para Alice e o outro para Bob. Partindo do estado padrão no IBMQ, que é $|00\rangle$, podemos obter o estado emaranhado de Bell $|\Phi_+\rangle$ da seguinte forma:

$$C_X^{A\to B}(H\otimes\mathbb{I})|00\rangle = C_X^{A\to B}|+0\rangle = C_X^{A\to B}\frac{1}{\sqrt{2}}\big(|00\rangle + |10\rangle\big) = \frac{1}{\sqrt{2}}\big(|00\rangle + |11\rangle\big) = |\Phi_+\rangle. \tag{2.21}$$

Vale observar que o protocolo de CSD envia 2 cbits ao enviar 1 qubit, mas, para isso, precisa do par emaranhado compartilhado entre Alice e Bob. Então, no final do dia, 2 qubits foram enviados para Bob. Tem-se a questão da ordem temporal a ser levada em conta aqui, mas não discutiremos esse ponto.

Primeiro, definimos a função que realiza a operação U_j no qubit de Alice, encarregada de codificar os 2 cbits que ela deseja transmitir a Bob. Isso é feito por meio da seguinte função:

```
def qc_alice_encoding(cb):
    qc = QuantumCircuit(1, name='AE')
    if cb == '00':
        qc.id(0)
    elif cb == '01':
        qc.x(0)
    elif cb == '10':
        qc.z(0)
    elif cb == '11':
        qc.x(0); qc.z(0)
    return qc
```

Abaixo, implementamos o protocolo de codificação superdensa (CSD) conforme discutido na parte teórica. Para uma separação mais interessante na visualização das etapas do protocolo CSD, remova os comentários (símbolo de #) nas linhas 4 e 7. É importante deixar essas linhas comentadas quando se for executar o experimento em chips quânticos reais:

```
def qc_dense_coding(cb):
    qc = QuantumCircuit(2, name='DC')
    qc.h(0); qc.cx(0,1) # compartilha o par emaranhado
    #qc.barrier()
    qcae = qc_alice_encoding(cb)
    qc.append(qcae, [0])
    #qc.barrier()
    qc.cx(0,1); qc.h(0)
    return qc
```

No código abaixo, o comando `QuantumCircuit(2,2)` cria um circuito com dois qubits (primeira entrada no comando) e dois registros clássicos (segunda entrada no comando), sendo, respectivamente, os dois qubits destinados às operações quânticas e os dois registros clássicos destinados ao armazenamento dos resultados das medições. O comando `qcdc = qc_dense_coding('01')` chama a função previamente definida que implementa o protocolo de codificação superdensa, retornando o circuito quântico correspondente à codificação da mensagem clássica `'01'`. Esse circuito é, então, incorporado ao circuito principal por meio do comando `qc.append(qcdc, [0,1])`, que aplica as operações do protocolo aos qubits q_0 e q_1. Por fim, o comando `qc.measure([0,1],[0,1])` realiza a medição dos dois qubits na base computacional, armazenando os resultados nos bits clássicos correspondentes, o que permite recuperar a informação clássica transmitida após a execução do circuito. A primeira lista desse último comando corresponde aos dois qubits, enquanto a segunda se refere aos dois registros clássicos. Como podemos alterar a ordem de ambos, é essencial ter um cuidado extra para interpretar corretamente o resultado da medição. Vamos ao código.

```
from qiskit import QuantumCircuit
qc = QuantumCircuit(2,2)
qcdc = qc_dense_coding('01'); qc.append(qcdc, [0,1])
qc.measure([0,1],[0,1])
```

Por fim, com o código `qc.decompose().draw('mpl')`, renderizamos o circuito quântico. O circuito produzido pode ser visualizado na Figura 2.3. Note que o circuito aqui está com as barreiras no código, que são as barras cinzas verticais com uma linha pontilhada. Experimente remover o `.decompose()` ou incluir `.decompose().decompose()` e executar o código para ver o efeito no circuito gerado.

Para a simulação clássica, vamos inserir o Código 1.3. Vamos, agora, utilizar o *Sampler*, que será o responsável por obter as contagens. Para isso, precisamos enviar um circuito como uma lista decomposta para a correta interpretação e execução do circuito. Como a escolha da Alice está encapsulada em uma função e ainda há uma função para implementar o protocolo, precisamos utilizar o `.decompose()` duas vezes. Por fim, utilizamos o *Sampler* para extrair as contagens. No caso, utilizamos $= 2^{12} = 4096$ execuções (aqui, denominadas `shots` e definidas no Capítulo 1) do mesmo circuito para obter as contagens e armazenamos o resultado bruto dessas execuções na variável `job`, que representa uma requisição enviada ao serviço de execução: ela contém o identificador do experimento, incluindo as contagens da medição e metadados da simulação. Sendo assim, o código fica como segue: