IMPLEMENTAÇÃO DO ALGORITMO QUÂNTICO DEUTSCH-JOZSA EM LINGUAGEM FUNCIONAL E NO SIMULADOR IBM Q *EXPERIENCE*

Lucas Lacava[1]
Mariana Vazquez Miano[2]

Resumo
O objetivo deste artigo é mostrar a implementação do algoritmo quântico Deutsch-Jozsa em linguagem funcional e a simulação deste no IBM Q *Experience*. Para tanto, são apresentados conceitos matemáticos fundamentais e conceitos específicos da Computação Quântica, tal como sobreposição de estados, Bit Quântico, Esfera de Bloch e Portas Quânticas. Também serão apresentadas definições de programação imperativa, linguagem Python e seu caráter multi-paradigma, utilizada pela IBM no simulador *IBM Q Experience,* para a simulação de circuitos quânticos. Por fim, são exibidas definições e características da programação funcional, da linguagem Haskell e da linguagem Quipper, onde se apresenta a implementação detalhada e comentada do algoritmo quântico de Deutsch-Jozsa juntamente com o pseudocódigo.
Palavras chave: computação quântica; linguagem funcional; quipper; python; haskell**.**

Abstract
The goal of this article is to show the implementation of the quantum algorithm of Deutsch-Jozsa in functional programming language and the simulation in IBM Q Experience simulator, therefore, are presented fundamental mathematical concepts and specific concepts of Quantum Computation, such as superposition of states, Quantum Bit, Bloch Sphere and Quantum Gates. Then, are presented definitions of imperative programming, the Python language and its multi-paradigm feature, which is the reason it was adopted by IBM in the simulator IBM Q Experience, used for the simulation of quantum circuits. Then, definitions and characteristics of the functional programming, the Haskell language and the language Quipper are presented, in which the detailed and commented implementation of the Deutsch-Jozsa quantum algorithm along with the pseudocode.
Keywords: *quantum computing; functional programming language; quipper; python; haskell.*

Introdução

A Física e a Computação Quântica estão em constante evolução e têm despertado interesse de pesquisadores de diversas áreas, especialmente por serem consideradas o futuro da Computação. Por isso, é importante que programadores compreendam a lógica deste novo paradigma, que segundo José, Piqueira e Lopes (2013), possui três etapas: preparação dos estados iniciais, realização das transformações unitárias e execução das medições.

O estudo deste tema justifica-se pela obsolescência da Lei de Moore que previu o crescimento de 100% da capacidade de processamento a cada 18 meses enquanto seu custo permaneceria constante. Segundo Nielsen e Chuang (2010), uma possível explicação para a falha na lei de Moore é a mudança de paradigma, uma vez que a Computação Quântica se baseia em mecânica quântica ao invés de Física Clássica, possibilitando a expansão de horizontes e da pesquisa.

[1] Discente do curso de Análise e Desenvolvimento de Sistemas da Fatec Americana. E-mail: lacaval@tcd.ie
[2] Docente e pesquisadora da Fatec Americana. E-mail: vazquez.prof@gmail.com

O objetivo deste trabalho é o estudo de conceitos da Computação Quântica e demonstração prática da implementação de um algoritmo quântico em linguagem funcional e no simulador *IBM Q Experience*. Com este fim, nas seções 1 e 2 foram realizadas pesquisas específicas sobre definições físico-matemático-computacionais que embasam o tema, tal como esfera de Bloch, bits quânticos e portas quânticas. Na seção 3, ocorre o aprofundamento de conhecimentos rotineiros, tal como programação imperativa e linguagem Python e também a demonstração do algoritmo quântico escolhido, Deutsch-Jozsa, no simulador *IBM Q Experience*. Por fim, na seção 4, conceitos menos cotidianos, tal como programação funcional, linguagem Haskell e Quipper são apresentados, definidos e exemplificados.

1 Definições matemáticas de álgebra linear

É necessário compreender alguns conceitos matemáticos de Álgebra Linear, ramo da matemática que estuda equações lineares, que terá alguns de seus principais conceitos e estruturas fundamentais demonstrados por Steinbruch e Winterle (1987). Posteriormente também serão apresentados conceitos como a Lei de Moore, Notação de Dirac e Espaço de Hilbert.

1.1. Espaço Vetorial Real

Um conjunto *V*, não vazio, onde estão definidas as operações de adição e multiplicação por escalar é considerado um espaço vetorial real caso se verifiquem os axiomas:

Quanto à adição, $\forall\, u, v, w \in V$:

A_1) $(u + v) + w = u + (v + w)$

A_2) $u + v = v + u$

A_3) $\exists\, 0 \in V, u + 0 = u$

A_4) $\exists\, (-u) \in V, u + (-u) = 0$

Quanto à multiplicação, $\forall\, u, v \in V$ e $\forall\, \alpha, \beta \in \mathbf{R}$:

M_1) $(\alpha . \beta)u = \alpha(\beta . u)$

M_2) $(\alpha + \beta)u = \alpha u + \beta u$

M_3) $\alpha(u + v) = \alpha u + \alpha v$

M_4) $1u = u$

- Os elementos $u, v, w, \ldots,$ de um espaço vetorial *V* são denominados vetores.
- Se a definição de espaço vetorial usasse o conjunto $\mathbf{C}$ dos números complexos como escalares, *V* seria chamado de espaço vetorial complexo.

1.2. Subespaço Vetorial

Considerando que *S*, um subconjunto não vazio do espaço vetorial *V*, seja um espaço vetorial em relação à adição e à multiplicação por escalar. Para que S seja um subespaço de *V*, deve satisfazer as seguintes condições:

I) Para quaisquer $u, v \in S, u + v \in S$.

II) Para quaisquer $\alpha \in \mathbb{R}, u \in S, \alpha . u \in S$

Todo espaço vetorial $V \neq \{0\}$ admite no mínimo dois subespaços, o subconjunto nulo e o próprio espaço vetorial *V*. Esses dois são os subespaços triviais, enquanto os outros são denominados subespaços próprios.

1.3. Combinação Linear de Vetores

Sejam os vetores $v_1, v_2, ..., v_n$ do espaço vetorial V e os escalares $a_1, a_2, ..., a_n$. Qualquer $v \in V$ da forma $v = v_1a_1 + v_2a_2 + ... + v_na_n$ é uma combinação linear dos vetores $v_1, v_2, ..., v_n$.

1.4. Produto Interno em Espaço Vetorial

Se define como produto escalar (ou interno) no espaço vetorial V uma aplicação de $V \times V$ em $\mathbf{R}$ que a todo par de vetores $(u, v) \in V \times V$ associa um número real, indicado por $u.v$ ou por $< u, v >$, tal que os seguintes axiomas sejam verificados:

P_1) $u.v = v.u$

P_2) $u.(v + w) = u.v + u.w$

P_3) $(\alpha.u).v = \alpha(u.v)$

P_4) $u.u \geq \mathbf{0}$ e $u.u = \mathbf{0}$, se $u = \mathbf{0}$

O número real $u.v$ é também chamado de produto interno dos vetores u e v. Ou seja, o produto interno resulta em um número real.

1.5. Coordenadas Polares

Coordenadas polares são um sistema de coordenadas bidimensional que descreve um ponto no espaço como um ângulo de rotação ao redor da origem e um raio a partir dela (SHIFFMAN, 2003). Pensando nisso nos termos de um vetor, coordenadas cartesianas possuem componentes x e y de um vetor, e coordenadas polares, possuem a magnitude (comprimento) e direção (ângulo) de um vetor.

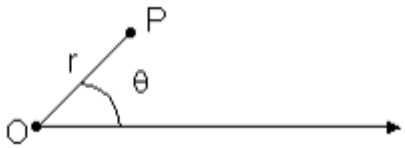
A letra grega θ (teta) é usada frequentemente para simbolizar um ângulo, dessa forma, uma coordenada polar pode ser mencionada convenientemente como (r, θ) ao invés de (x, y).

De acordo com Nascimento (2005) dado um ponto *P* do plano, utilizando coordenadas cartesianas (retangulares), descrevemos sua localização no plano escrevendo *P* = (a,b) onde a é a projeção de P no eixo x e b, a projeção no eixo y. Podemos também descrever a localização de *P*, a partir da distância de *P* à origem *O* do sistema, e do ângulo formado pelo eixo x e o segmento *OP*, caso *P*≠*O*. Denotamos *P* = (r,θ) onde r é a distância de *P* a *O* e θ o ângulo tomado no sentido anti–horário, da parte positiva do eixo *O*x ao segmento *OP*, caso *P*≠*O*. Se *P* = *O*, denotamos *P* = (0,θ), para qualquer θ.

Para representar pontos em coordenadas polares, necessitamos somente de um ponto *O* do plano e uma semirreta com origem em *O*. Representamos abaixo um ponto *P* de coordenadas polares (r,θ), tomando o segmento *OP* com medida r.

Na figura 1 pode-se ver o ponto fixo O que é chamado polo e a semirreta, eixo polar, observe:

Figura 1 - Representação de coordenadas polares no eixo de θ



Fonte: Nascimento (2005).

Em coordenadas polares, há possibilidade de representações diferentes para um mesmo ponto, isto é, podemos ter *P* = (r,θ) e P = (s, α) sem que r = s e θ = α, ou seja (r,θ) = (s,α) não implica em r = s e θ = α. Assim, (r,θ) não representa um par ordenado, mas sim uma classe de pares ordenados, representando um mesmo ponto. Denotamos um ponto *P* por (r,–θ), para r e

θ positivos, se θ é tomado no sentido horário. Assim, (r,–θ) = (r,2π–θ) e (r,–θ) é o simétrico de (r,θ) em relação à reta suporte do eixo polar.

1.6. Números complexos

Os números complexos são elementos que habitam o conjunto $\mathbf{C}$, onde existe possibilidade de raízes negativas. Cerri e Monteiro (2001) nos mostra que essa abordagem trouxe controvérsias, mas Girolamo Cardano demostrou tal capacidade. Assim, interessado pelos estudos de Cardano, outros matemáticos como, Descartes, Euler e Gauss, estudaram esses novos números e consolidaram o conceito de números complexos.

1.7. Forma algébrica

O número complexo possui a definição dada pela equação 1:

$$z = a + bi \ (a \in R, b \in R \ e \ i^2 = -1) \qquad (1)$$

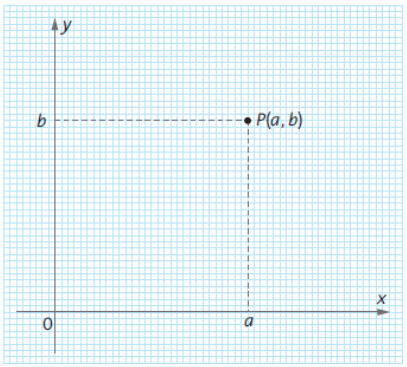
Essa representação é conhecida como forma binomial ou forma algébrica, onde a é a parte real de z, b é a parte imaginária de z e i é uma unidade imaginária. A unidade i é o fator que possibilita a existência de raiz quadrada negativa, por exemplo, na equação 2:

$$\text{Se } x \in C \ e \ x^2 = -4 \text{, então } x = \pm 2i \qquad (2)$$

1.8. Representação geométrica

Outra forma de representar os números complexos, é através de um par ordenado de números reais $z = (a, b)$, onde a e b são coordenadas de um plano cartesiano, entretanto denominado como Plano complexo ou plano de Argand-Gauss. Como no exemplo da figura 2:

Figura 2 - Representação geométrica de um número complexo



Fonte: elaborada pelo autor.

1.9. Lei de Moore

O físico estadunidense Gordon Moore estimou em 1965 que a cada 18 meses a capacidade de processamento iria aumentar em 100% enquanto seu custo permaneceria constante, posteriormente esta profecia provou-se real e passou a ser conhecida como Lei de Moore.

Atualmente a Lei de Moore deixou de ser uma regra e se tornou uma meta para as empresas de tecnologia que buscam evoluir usando diferentes componentes, elementos e mais

recentemente, paradigmas, tal qual a Computação Quântica. E de acordo com Nielsen e Chuang (2010) uma possível explicação para a falha na lei de Moore é a mudança de paradigma, uma vez que a Computação Quântica se baseia em mecânica quântica ao invés de Física Clássica.

1.10. Notação de Dirac ou Bra-ket

Segundo Strubell (2011), a notação de Dirac ou Bra-ket é a mais usada para descrever sistemas mecânicos quânticos e também registradores quânticos, e possui esse nome em homenagem ao ganhador do Nobel Paul Dirac. A notação é um outro modo de descrever vetores. A coluna vetorial $\square|v\rangle$ é chamada de "v-ket". Na notação de Dirac:

$$v = \begin{bmatrix} v0 \\ v1 \\ \cdot \\ \cdot \\ \cdot \\ vn \end{bmatrix} = \square|v\rangle$$

1.11. Espaço de Hilbert

De acordo com Strubell (2011), espaço de Hilbert é um vetor espacial com um produto interno e uma regra definida pelo produto interno. O produto interno de um vetor é uma operação que define um valor escalar para cada par de vetores u e v no espaço vetorial, e o produto interno de dois vetores no espaço de Hilbert em C^n é representado utilizando a notação Dirac $\langle u|v\rangle$. Portanto o produto interno $\langle u|v\rangle$ de dois vetores no Espaço complexo de Hilbert é dado pelo produto interno dos vetores v e u^t, o conjugado transposto de *u*:

$$\langle u|v\rangle = u^t v = [u_0\ u_1 \dots u_n] \begin{bmatrix} v0 \\ v1 \\ \cdot \\ \cdot \\ \cdot \\ vn \end{bmatrix} = u_0 \cdot v_0 + u_1 \cdot v_1 + \cdots + u_n \cdot v_n$$

2. Conceitos de computação quântica

A seguir são apresentados conceitos básicos importantes para a compreensão da Computação Quântica, tal como: sobreposição de estados, bit quântico, esfera de Bloch, as principais portas quânticas: operadores de Pauli, portas de fase, porta de Hadamard e por fim, o algoritmo quântico de Deutsch-Jozsa.

2.1. Sobreposição de estados

Para melhor entender o conceito de sobreposição de estados Nielsen e Chuang (2010) comparam Computação Quântica e Computação Clássica. Um bit clássico funciona de forma similar a uma moeda, ao lançá-la o resultado será ou cara ou coroa, e o bit clássico pode ser zero ou um. Porém, no caso do lançamento de uma moeda irregular pode haver estados intermediários, como por exemplo, equilibrada em uma de suas arestas. Da mesma maneira, paralelamente, qubits podem existir continuamente em estados entre $\square|0\rangle$ e $\square|1\rangle$, até serem estimados.

É importante enfatizar que quando um qubit é medido o resultado será a probabilidade de zero ou um, por exemplo, o estado de um qubit pode ser $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$, no momento da

medição há possibilidade de o resultado ser zero durante 50% do tempo e de ser um durante 50% do tempo.

2.2. Bit quântico

A Computação Clássica utiliza-se dos bits, unidades de informação que trabalham no estado 0 ou 1. Entretanto, a computação quântica substituiu os bits pelo bit quântico, o qubit, que pode ser encontrado nos estados 0, 1 ou em uma superposição ($|\psi\rangle$) entre os dois estados, sendo assim, pode existir nos dois estados ao mesmo tempo em $|0\rangle$ e $|1\rangle$, até ser observado. Os estados 0 e 1 podem ser apresentados como vetores, conforme apresentado na equação 3:

$$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix} \text{ e } |1\rangle = \begin{bmatrix}0\\1\end{bmatrix} \quad (3)$$

Enquanto a superposição é um estado onde os qubits podem ser 0 e 1 simultaneamente, ele é definido na equação 4 desta maneira:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \quad (4)$$

Onde α e β são números complexos que representam a amplitude dos estados, utilizados para medir os estados. A probabilidade de um estado resultar 0 é $|\alpha|^2$, a probabilidade de resultar 1 é $|\beta|^2$ e a soma dos dois resultados deve ser 1, assim como determina a equação 5:

$$|\alpha|^2 + |\beta|^2 = 1 \quad (5)$$

Um qubit também pode ser expresso de diversas outras maneiras para ajudar na compreensão da Esfera de Bloch, local geométrico do mesmo, utilizando apenas de Transformadores Unitários *(T)*, que mudam a dimensão desse qubit sem mudar sua estrutura. O exemplo mais famoso disso é a representação Polar do bit quântico que é determinada pela equação 6:

$$|\psi\rangle = e^{i\gamma}\left[\cos(\xi)|0\rangle + e^{i\varphi}\sin(\xi)|1\rangle\right] \quad (6)$$

Onde $\xi = \frac{\theta}{2}$ e i é a representação de um número imaginário. Também podemos usar como exemplos as transformações *(T)* onde os vetores unitários $\mathbf{C^2}(\mathbb{R})$ e $\mathbf{R^4}(\mathbb{R})$ tornam-se os modelos matemáticos para um qubit. Sendo a representação deles dada respectivamente pela equação 7 e equação 8:

$$|\psi\rangle = a\begin{bmatrix}1\\0\end{bmatrix} + b\begin{bmatrix}i\\0\end{bmatrix} + c\begin{bmatrix}0\\1\end{bmatrix} + d\begin{bmatrix}0\\i\end{bmatrix} \quad (7)$$

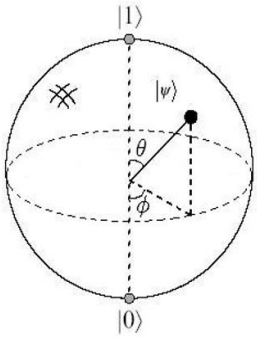
$$|\psi\rangle = a\begin{bmatrix}1\\0\\0\\0\end{bmatrix} + b\begin{bmatrix}0\\1\\0\\0\end{bmatrix} + c\begin{bmatrix}0\\0\\1\\0\end{bmatrix} + d\begin{bmatrix}0\\0\\0\\1\end{bmatrix} \quad (8)$$

Dessa forma, pudemos estabelecer algumas fórmulas básicas para entender melhor o ambiente onde se encontram todos os Vetores que compõem os qubits e assim compreender como essas transformações ocorrem dentro da Esfera de Bloch.

2.3. Esfera de Bloch

Segundo Vignatti, Netto e Bittencourt (2004), um qubit pode ser geometricamente representado em três dimensões. Esta representação é chamada Esfera de Bloch, representada na figura 3:

Figura 3 - Esfera de Bloch: representação 3D do qubit.



Fonte: Vignatti, Netto, Bittencourt (2004)

De acordo com Vignatti, Netto e Bittencourt (2004), fazer a medição de um qubit num dado estado irá retornar 0 com probabilidade de α^2 e 1 com probabilidade de β^2, e mais importante que isso, o estado da medição será $|0\rangle$ ou $|1\rangle$.

2.4. Portas quânticas

Em circuitos clássicos de computação, "são utilizadas portas lógicas para manipular n bits de entrada e computar a saída de m bits" (Grilo, 2014), porém as operações de circuitos quânticos são reversíveis, sendo assim é necessário que o número de qubits da entrada seja igual ao número de qubits de saída. Serão explanados a seguir as principais portas lógicas: operadores de Pauli, porta de Hadamard e portas de fase.

2.4.1. Operadores de Pauli

Três das principais portas quânticas são conhecidas como os operadores de Pauli, e recebem a notação *X, Y* e *Z*.

"A porta *X* é o *bit-flip* quântico, alterando o estado de |0⟩ para |1⟩ ou de |1⟩ para |0⟩, pode ser comparado ao *NOT* na Computação Clássica, que possui a função de inverter o estado do bit em que é aplicado" (GRILO, 2014).

A representação matemática da porta *X* é a matriz quadrada:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

De acordo com Barbosa (2005, p. 14) "a porta *Y* realiza uma rotação de **π** ao redor do eixo *Y*. Realiza tanto o *bit flip* quanto uma inversão de fase".

A representação matemática da porta *Y* é a matriz quadrada:

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

A porta *Z* é uma rotação de **π** ao redor do eixo *Z* e também alterna a fase do qubit. Sua representação matemática é a matriz quadrada:

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

2.4.2. *Portas de Fase*

As portas de fase são responsáveis por realizar alterações na fase do qubit, possuindo quatro variações, *S*, , *T e* .

A porta *S*, realiza uma rotação de $\frac{\pi}{2}$ e é equivalente a $\sqrt{Z}$, uma vez que *Z* realiza uma rotação de **π**. A representação matemática de *S*, é descrita por Nielsen e Chuang (2010) da seguinte maneira:

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$$

A porta é a matriz *S* adjunta portanto, realiza uma rotação de $-\left(\frac{\pi}{2}\right)$, e sua representação matemática, é a seguinte:

$$S^{\dagger} = \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix}$$

A porta *T*, de acordo com Nielsen e Chuang (2010) realiza uma rotação de $\frac{\pi}{4}$ e é equivalente a $\sqrt{S}$, uma vez que *S* realiza uma rotação de $\frac{\pi}{2}$. A representação matemática de *T* é descrita abaixo:

$$T = \begin{pmatrix} 1 & 0 \\ 0 & \frac{1+i}{\sqrt{2}} \\ 1 & 0 \end{pmatrix}$$

A porta é a matriz *T* adjunta, portanto, realiza uma rotação de $-\left(\frac{\pi}{4}\right)$, e sua representação matemática é a seguinte:

2.4.3. *Porta de Hadamard*

A porta de Hadamard é uma das portas quânticas mais úteis, pois é responsável por levar um estado a uma superposição. Segundo Barbosa (2005) sua representação matemática é a seguinte:

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

Segundo Nielsen a Chuang (2010) quando aplicada ao estado |00⟩, Hadamard o transforma em (|0⟩ + |1⟩) |0⟩ /√2, e quando aplicada ao estado |10⟩, Hadamard o transforma em (|00⟩ - |11⟩)/√2 . É importante ressaltar que caso Hadamard seja aplicado duas vezes em um estado o resultado permanecerá como era inicialmente.

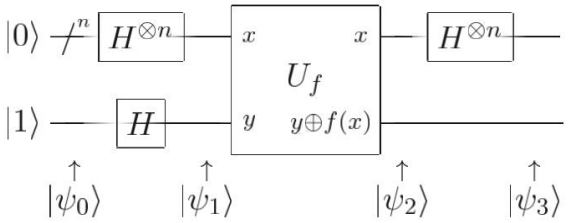
2.5. *Algoritmo de Deutsch-Jozsa*

De acordo com Nielsen e Chuang (2010) o algoritmo de Deutsch-Jozsa descreve um problema onde é necessário definir se uma função $f(x)$ é constante ou balanceada, caso seja constante, todos os resultados serão iguais a 0 ou iguais a 1, caso balanceada, metade dos resultados serão iguais a 0 e a outra metade igual a 1.

Na Computação Clássica, na pior das hipóteses seria necessário $\left(\frac{2n}{2}\right) + 1$ tentativas para definir o tipo da função, por exemplo: se a situação estudada possui 3 bits, há oito combinações possíveis: 000, 001, 010, 011, 100, 101, 110, 111. Considerando n igual ao

número de bits, temos que: $\left(\frac{2^3}{2}\right) + 1 = \left(\frac{8}{2}\right) + 1 = 5$. Desta forma, seriam necessários cinco parâmetros para ter certeza do tipo da função, porém, a grande vantagem neste algoritmo é que o resultado é descoberto com apenas um parâmetro. A figura 4 exemplifica o circuito com o algoritmo de Deutsch-Jozsa aplicado:

Figura 4 - Circuito quântico implementando algoritmo de Deutsch-Jozsa



Fonte: Nielsen e Chuang (2010).

O circuito acima considera *n* qubits no estado |0⟩ e 1 qubit no estado |1⟩, a porta de Hadamard é aplicada em ambas. Aplica-se uma função oráculo em ambos os estados e por fim, Hadamard é aplicado em todos os qubits no estado |0⟩. Deste circuito é possível visualizar 4 estados: $|\psi_0\rangle, |\psi_1\rangle, |\psi_2\rangle, |\psi_3\rangle$, sendo $|\psi_0\rangle$ o estado inicial, $|\psi_1\rangle$ o estado após a aplicação de Hadamard em todos os qubits, $|\psi_2\rangle$ o estado após a aplicação da função oráculo e finalmente, $|\psi_3\rangle$ após a aplicação de Hadamard nos qubits em |0⟩ e o estado final.

Como citado anteriormente, sabe-se que |ψ0⟩ é o estado inicial, desta forma, pode ser definido como $|\psi\rangle = |0\rangle^{\otimes n}|1\rangle$. Então, é aplicada a porta de Hadamard em todos os qubits, portanto $|\psi_1\rangle$ é representado pela equação 9:

$$|\psi_1\rangle = \sum_{x\in\{0,1\}^n} \frac{|x\rangle}{\sqrt{2^n}} \left[\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right] \quad (9)$$

A seguir, é aplicada a função oráculo em ambos os qubits, resultando em $|\psi_2\rangle$ na equação 10:

$$|\psi_1\rangle = \sum_{x} \frac{(-1)^{f(x)}|x\rangle}{\sqrt{2^n}} \left[\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right] \quad (10)$$

E por fim, é aplicada a porta de Hadamard aos qubits no estado |0⟩, resultando na equação 11 resumida:

$$H^{\otimes n}|x\rangle = \frac{\sum_z (-1)^{x\cdot z}|z\rangle}{\sqrt{2^n}} \quad (11)$$

Sendo x.z o produto interno de x e z, podemos então chegar ao resultado de $|\psi_3\rangle$ representada na equação 12:

$$|\psi_3\rangle = \sum_z \sum_x \frac{(-1)^{x\cdot z + f(x)}|z\rangle}{2^n} \left[\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right] \quad (12)$$

Segundo Nielsen e Chuang (2010), na fase final, a partir do resultado de $|\psi_3\rangle$ é então mensurado o estado $|z\rangle$, e caso $f(x)$ seja constante, a amplitude do estado |0⟩ é 0. Se $f(x)$ for

balanceada, a amplitude do estado |0⟩ é ± 1 . Ou seja, é necessário apenas mensurar o estado z para chegar ao resultado desejado.

3. Paradigmas de programação

Neste tópico são apresentados paradigmas de computação, o imperativo e o funcional, mostrando quais suas principais características. Detalha-se a linguagem de programação Python, e porque é considerada multi-paradigma, sendo utilizada tanto em programação imperativa quanto em funcional. Também se discorre sobre o *IBM Quantum Experience*, simulador da IBM para processamento quântico.

Por fim é apresentada a linguagem Haskell e Quipper, utilizadas posteriormente para implementação do algoritmo de Deutsch-Jozsa.

3.1. Programação imperativa

De acordo com José, Piqueira e Lopes (2013), linguagens imperativas também são conhecidas como procedurais e são constituídas pelo uso de declarações que mudam o estado global do programa ou variáveis do sistema. Exemplos de linguagens imperativas clássicas são: FORTRAN, Pascal, C, Java e Python.

Segundo Tucker e Noonan (2007),

> [...] todas as linguagens imperativas possuem atribuição como um elemento central, adicionalmente suportam declaração de variáveis, expressões, declarações condicionais, *loops* e abstração de processos. Declarações definem nomes a posições de memória e associam tipos com valores armazenados, expressões são interpretadas quando retornam seus valores atuais das variáveis do respectivo espaço de memória e computam um resultado a partir destes valores. [...] Comandos são normalmente executados na ordem em que aparecem na memória, enquanto fluxos paralelos condicionais e incondicionais podem interromper o fluxo padrão de execução.

E por fim, Castro (2002) caracteriza que "no paradigma imperativo a ênfase é em "como" fazer.

3.2. Python

Segundo Venners (2003), a linguagem Python foi concebida no fim dos anos 80 por Guido van Rossum no *Centrum voor Wiskunde en Informatica (CWI)*, Instituto Nacional de Pesquisa de Matemática e Ciência da Computação da Holanda, porém a primeira versão foi lançada em 1991, e desde então vem se popularizando em meio aos programadores, por possuir uma sintaxe limpa e reputação de produtividade.

Algumas linguagens de programação são procedurais, por exemplo C e Pascal, ou seja, o programa é uma lista de instruções que dita o que o computador deve fazer com o *input* recebido.

Outras linguagens são declarativas, como SQL, o programador escreve a especificação que descreve o problema a ser resolvido e a linguagem implementada descobre como executar a computação de dados de forma eficiente.

Linguagens Orientadas a Objeto, tal como Java, C++ e Python, manipulam conjuntos de objetos, que possuem estados e suportam funções que modificam os estados de alguma forma.

De acordo com Kuchling (2001), programação funcional decompõe o problema em uma série de funções, que recebem os *inputs* e produzem *output,* sem realizar mudança de estado internamente. Haskell é um exemplo de linguagem funcional muito conhecida.

Porém, também há linguagens multi-paradigmas, é possível escrever programas que são procedurais, orientadas a objeto ou funcionais, como é o caso de Python.

Segundo Tucker e Noonan (2007), inúmeros autores consideram Python uma linguagem multi-paradigma suportando os seguintes estilos de programação: procedural, orientado a objeto e funcional. Tal como Perl, Python é uma linguagem de *script*, sendo na maioria das vezes interpretado, ao invés de compilado.

Justamente por ser multi-paradigma, a linguagem Python é utilizada em ambientes de programação funcional e foi escolhida como linguagem do simulador de processadores quânticos IBM *Quantum Experience*, abordado no tópico 3.3.

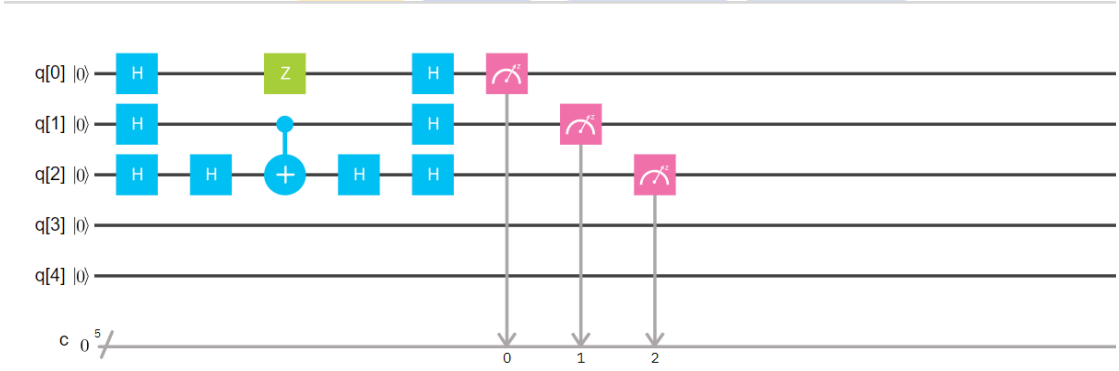
3.3. IBM Quantum Experience

O simulador IBM *Quantum Experience*, permite que seus usuários se conectem aos processadores quânticos através *do* IBM *Cloud*, executem algoritmos e programas e também apresenta um manual completo de uso, para iniciantes ou usuários avançados.

A interação pode acontecer de duas maneiras, através de um modelo de circuitos, onde o usuário aplica portas quânticas em um circuito, ou através de linguagem de programação utilizando uma *API* Python.

Na figura 5 é apresentado o algoritmo de Deutsch-Jozsa implementado no simulador *Quantum Experience*, com 3 qubits (n = 3):

Figura 5 - Algoritmo de Deutsch-Jozsa no IBM Quantum Experience



Fonte: Disponível em: <https://quantumexperience.ng.bluemix.net/qx >

A seguir é apresentado a mesma implementação do algoritmo através da *API* Python:

```
include "qelib1.inc";
qreg q[5];
```

```
creg c[5];
h q[0];
h q[1];
h q[2];
h q[2];
z q[0];
cx q[1], q[2];
h q[2];
h q[0];
h q[1];
h q[2];
measure q[0] -> c[0];
measure q[1] -> c[1];
measure q[2] -> c[2];
```

No tópico a seguir são apresentados os conceitos e fundamentos de programação funcional, a linguagem Haskell, alguns exemplos de sua sintaxe e uso e por fim, a linguagem Quipper, baseada em Haskell e voltada para Computação Quântica.

3.4. Programação funcional e principais linguagens funcionais

De acordo com Tucker e Noonan (2007), a programação funcional surgiu como um paradigma da computação no inicio dos anos 60, motivada pela necessidade dos pesquisadores da área de inteligência artificial e derivados, cuja suas necessidades não eram sanadas pelas linguagens imperativas da época.

José, Piqueira e Lopes (2013) conceituam linguagens funcionais como um tipo de paradigma para programação de computadores em que o foco das funções é puramente nas entradas e saídas, tal qual ocorre na Matemática, em oposição às linguagens imperativas que alteram estados e dados.

Para compreender o que é uma linguagem funcional é importante conceituar o que é uma função, de acordo com Thompson (2011), "função é algo que podemos imaginar como uma caixa que possui entradas e saídas, que retorna um valor de saída de acordo com os valores de entrada, o termo resultado pode ser usado para representar a saída e os termos parâmetros ou argumentos para representar as entradas.

Dentre as principais linguagens funcionais, destaca-se Lisp e seus derivados como Clojure e Scheme, Scala, Elixir, R e Haskell, linguagem utilizada e objeto de estudo deste artigo.

3.5. Linguagem funcional Haskell

Segundo Thompson (2011), Haskell foi nomeada em homenagem à Haskell B. Curry, um dos pioneiros no cálculo de lambda, uma teoria matemática de funções e inspiração para diversas linguagens funcionais. Foi criado no início dos anos 80 e desde então passou por diversas revisões até alcançar seu estado "padrão".

Thompson (2011) compara "computação funcional com uma calculadora eletrônica, o programador provê uma expressão, e o sistema a estima para retornar seu valor, portanto a tarefa do programador é escrever as funções que modelam a área do problema. Um programa funcional é feito por uma série de definições de funções e outros valores"

Um programa funcional em Haskell consiste em uma série de definições, associando o nome ao valor de um tipo específico, basicamente funciona da seguinte maneira:

nome :: tipo
nome = expressão
O exemplo posto em prática, funcionaria conforme a expressão a seguir:
tamanho :: Int
tamanho = 2+3

Também é possível definir funções, como no exemplo a seguir é definido a função quadrado, responsável por elevar um número ao quadrado, ou seja, multiplicando o número por ele mesmo:

quadrado :: Int -> Int
quadrado n = n*n

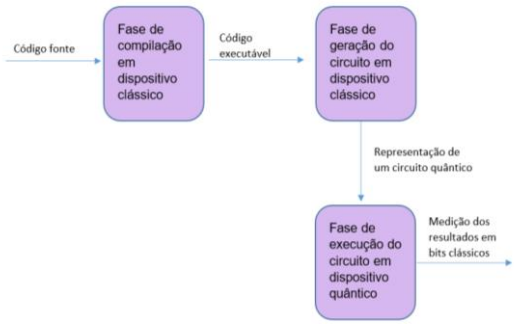
O código acima demonstra na primeira linha que quadrado é uma função, devido a presença da seta (->), que possui um argumento do tipo inteiro (antes da seta) e retorna um resultado também do tipo inteiro (depois da seta). A segunda linha define a função, que quando é aplicada a uma variável desconhecida n o resultado será n * n.

3.6. Linguagem Quipper

De acordo com Green et. al (2013), "Quipper é uma linguagem de programação funcional integrada para Computação Quântica e foi desenvolvida de um projeto da IARPA (Intelligence Advanced Research Projects Activity)", desenvolvido por Richard Eisenberg, Alexander S. Green, Peter LeFanu Lumsdaine, Keith Kim, Siun-Chuon Mau, Baranidharan Mohan, Won Ng, Joel Ravelomanantsoa-Ratsimihah, Neil J. Ross, Artur Scherer, Peter Selinger, Benoît Valiron, Alexandr Virodov and Stephan A. Zdancewic, e seu lançamento foi em Junho de 2013. Quipper é baseado em Haskell, portanto pode ser considerada como uma série de data types, combinadores e bibliotecas de funções Haskell.

Segundo Siddiqui et. al (2014), programas em Quipper são executados em três fases, compilação, geração do circuito e execução do circuito. As duas primeiras etapas são executadas em um computador clássico, e a última é executada em um computador quântico. Na primeira etapa, compilação, Quipper analisa o código fonte, compila os parâmetros de tempo e utiliza o compilador Haskell para gerar um código de objeto para output. A segunda fase, geração do circuito, recebe como inputs o código de objeto da fase anterior, e parâmetros do circuito e gera a representação de um circuito quântico como output. Por fim, a terceira fase, execução do circuito, recebe o circuito quântico e algumas informações do circuito como input e retorna a medição dos resultados das sub-rotinas quânticas, a figura 7 ilustra esse processo:

Figura 7 - Modelo de execução Quipper



Fonte: Siddiqui et al (2014)

4. Implementação do algoritmo deutsch-jozsa em quipper/haskell

Neste tópico será apresentado o pseudocódigo do circuito quântico e então a sua implementação em Quipper, um *framework* baseado em Haskell, assim como partes do código comentado.

4.1. Pseudocódigo

Baseando-se na figura 2 supracitada, é possível simplificar o algoritmo de Deutsch-Jozsa no seguinte pseudocódigo comentado, onde o texto após "//" deve ser interpretado como comentário

```
q0 = Hadamard          // Porta de Hadamard (definida na equação 9) é aplicada no
qubit 0
q1 = Hadamard          // Porta de Hadamard é aplicada no qubit 1
q0 = Oraculo      // Função oráculo (definida na equação 10) é aplicada no qubit 0
q1 = Oraculo      // Função oráculo é aplicada no qubit 1
q0 = Hadamard          // Porta de Hadamard é aplicada novamente no qubit 0
```

4.2. Implementação em Quipper/Haskell

A seguir é apresentada a implementação do algoritmo de Deutsch-Jozsa na linguagem Quipper, que é um *framework* baseado em Haskell e focada em desenvolvimento para computação quântica. O código foi desenvolvido por Siddiqui, Islam e Shehab (2014), e cada trecho será devidamente comentado e explicado.

Na primeira linha do trecho seguinte é importada a biblioteca Quipper, então cria-se um tipo de dado chamado *Oracle*, e define-se que qubit_num é do tipo inteiro, e a função oráculo f(x), que é necessária para o algoritmo.

```
import Quipper
data Oracle = Oracle {
qubit_num :: Int,
function :: ([Qubit], Qubit) → Circ ([Qubit], Qubit)
}
```

A seguir é declarada a função deutsch_jozsa_circuit, e então é inicializada a *string* de qubits top_qubits e bottom_qubit, a primeira no estado $|0\rangle$ e a segunda no estado $|1\rangle$.

```
deutsch_jozsa_circuit :: Oracle → Circ [Bit]
deutsch_jozsa_circuit oracle = do
top_qubits ← qinit (replicate (qubit_num oracle) False)
bottom_qubit ← qinit True
label (top_qubit, bottom_qubit) ("|0⟩ ","|1⟩ ")
```

Então, é aplicada a primeira porta de Hadamard tanto em top_qubits quanto em bottom_qubits além de ser adicionado um comentário que aparecerá no diagrama final "antes do oráculo".

```
mapUnary hadamard top_qubits
hadamard_at bottom_qubit
comment "before oracle"
```

Depois é aplicada a função oráculo em top_qubits e bottom_qubit, além de ser adicionado o comentário "após oráculo".

```
function oracle (top_qubits, bottom_qubit)
```

```
        comment "after oracle"
```

Então, é aplicada a porta de Hadamard apenas nos *top*_qubits.

```
        mapUnary hadamard top_qubits
```

E é realizada a medição dos qubits do circuito, tanto de *top*_qubits, quanto de *bottom*_qubit:

```
        (top_qubits, bottom_qubit) ← measure (top_qubits, bottom_qubit)
```

Por fim, é criada a função *main*, responsável por executar todo o código, declarar uma função oráculo vazia (*empty*_oracle) e definir o número de qubits (5):

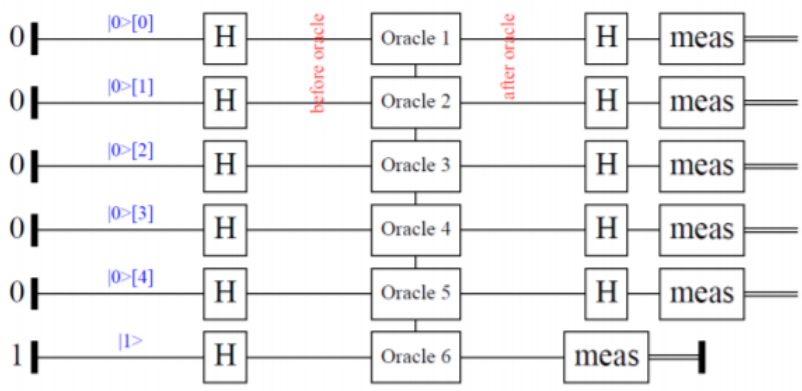
```
        main = print_generic Preview (deutsch_jozsa_circuit empty_oracle)
        where
        empty_oracle :: Oracle
        empty_oracle = Oracle {
        qubit_num = 5,
        function = empty_oracle_function
        }
```

Então, a função '*empty*_oracle' é inicializada e nomeada como '*Oracle'*.

```
        empty_oracle_function:: ([Qubit],Qubit) → Circ ([Qubit],Qubit)
        empty_oracle_function (ins,out) = named_gate "Oracle" (ins,out)
```

Sendo o resultado do algoritmo apresentado da seguinte forma:

Figura 8 - Output do algoritmo Deutsch-Jozsa em Quipper



Fonte: Siddiqui et al (2014)

Porém, para executar o algoritmo é necessário criar outra função para simular a execução e um novo circuito para demonstrar os resultados. Para tanto, são importadas as bibliotecas necessárias e implementada a função *simulate* que utiliza o operador *map* para

aplicar a operação *not* a cada um dos elementos do oráculo e negar seus valores, então é adicionado o operador *and* aos resultados e finalmente é retornado um valor booleano.

```
import qualified Data.Map as Map
import QuipperLib.Simulation
import System.Random
simulate :: Circ [Bit] → Bool
simulate oracle = and (map not (run_generic (mkStdGen 1) (1.0::Float) oracle))
```

A função *circuit* vai usar o booleano resultante da operação anterior, sendo que para oráculos constantes, booleano = verdadeiro, e para oráculos balanceados, booleano = falso.

```
circuit :: (Circ [Bit] → Bool) → Oracle → IO ()
circuit run oracle =
if run (deutsch_jozsa_circuit oracle)
then putStrLn "constant"
else putStrLn "balanced"
```

Por fim, a função *main* é usada para iniciar todo o programa, e exibe o resultado final se a função é constante ou balanceada:

```
main = do
circuit simulate constant_oracle
circuit simulate balanced_oracle
```

De forma condensada, é possível definir o funcionamento do algoritmo em sua primeira etapa como a importação das bibliotecas necessárias, criação de função oráculo, inicialização dos qubits, aplicação de Hadamard em todas os qubits, aplicação da função oráculo em todas os qubits, aplicação de Hadamard no conjunto *top_qubits* e por fim, a medição dos resultados.

Na segunda etapa é criada uma função *simulate* para realizar a simulação que retornará um valor booleano, são importadas bibliotecas e utilizados operadores como *map, not* e *and.* A função seguinte, *circuit*, utiliza o resultado da função *simulate*, caso seja verdadeiro, a função é constante, e caso seja falso, é balanceada.

Na etapa final, a função *main* inicia o algoritmo e exibe o resultado final, se a função é balanceada ou constante.

5. Conclusão

O principal objetivo deste artigo introdutório acerca de Computação Quântica foi apresentar os principais conceitos deste tema atual e complexo, desde sua fundamentação matemática, conceitos físico-computacionais e sua inserção nos paradigmas de computação, assim como código fonte e aplicação prática, através da implementação do algoritmo de Deutsch-Jozsa. E demonstrar a implementação do algoritmo quântico de Deutsch-Jozsa em linguagem funcional e no simulador IBM Q *Experience*, mostrando suas diferenças e vantagens.

Na seção 1, foram apresentados conceitos matemáticos; na seção 2, apresentou-se conceitos específicos de Computação e Física Quântica, tal como bit quântico, sobreposição de estados, Esfera de Bloch, as portas quânticas mais utilizadas (posteriormente representadas no simulador *IBM Q Experience*), e por fim, o algoritmo de Deutsch-Jozsa, um exemplo clássico de como a Computação Quântica pode resolver um problema mais rapidamente e de forma mais eficaz do que a Computação Clássica, uma vez que através do paradigma clássico, seriam necessárias $\left(\frac{2n}{2}\right) + 1$ tentativas para definir o tipo da função, balanceada ou constante, e através do paradigma da Computação Quântica, esse definição é feita apenas com um parâmetro.

A seção 3 conceituou paradigmas da computação, apresentou a programação imperativa, amplamente utilizada academicamente e no mercado de trabalho e discorreu sobre a linguagem Python, que é multi-paradigma, motivo pelo qual foi escolhida para ser a linguagem base do simulador *IBM Q Experience*, também apresentado neste tópico. Posteriormente, tratou-se sobre outro paradigma da computação, a programação funcional, apresentando seus fundamentos e origem; para exemplificar, foram utilizadas as linguagens funcionais Haskell e Quipper.

Inicialmente, a seção 4 apresentou o pseudocódigo da implementação do algoritmo de Deutsch-Jozsa, objetivando a compreensão de usuários sem conhecimento específico da linguagem trabalhada. Em seguida, é realizada a implementação em Quipper, um *framework* baseado em Haskell, que demonstrou quão importante é a utilização de linguagens funcionais para o avanço da Computação Quântica, visto que o desempenho e eficiência são superiores em relação ao equivalente em Computação Clássica.

Como trabalhos futuros, pretende-se a implementação do algoritmo Deutsch-Jozsa em linguagem Python, e de outros algoritmos quânticos, no sentido de demonstrar a eficiência da Computação Quântica em aspectos de desempenho, quando comparada à Computação Clássica. Há também a intenção de se desenvolver aplicações práticas da teoria quântica com a utilização da montagem e simulação de circuitos quânticos no *IBM-Q Experience*.

Referências

BARBOSA, A. A. **Introdução a Circuitos Quânticos**. 2005. 22 f. Dissertação (Mestrado) - Curso de Ciência da Computação, Departamento de Sistemas e Computação, Universidade Federal de Campina Grande, Campina Grande, 2005. Disponível em: < http://lad.dsc.ufcg.edu.br/arq/AC - Escrito - Circuitos Quanticos.pdf>

CASTRO, T. C. et al. Utilizando programação funcional em disciplinas introdutórias de computação. In: **X WORKSHOP SOBRE EDUCAÇÃO EM COMPUTAÇÃO. EVENTO INTEGRANTE DO XXII CONGRESSO DA SBC.** 2002, Florianópolis. Anais... Florianópolis: WEI, 2002.

CERRI, C. e MONTEIRO, M. S. (Org.). **História dos Números Complexos**. 2001. Disponível em: <https://www.ime.usp.br/~martha/caem/complexos.pdf >. Acesso em: 01 dez. 2017.

GREEN, A. S., LUMSDAINE, P. L., ROSS, N. J., SELINGER, P., VALIRON, B. *An introduction to quantum programming in Quipper*. In: **INTERNATIONAL CONFERENCE ON REVERSIBLE COMPUTATION**, 5th., 2013, Victoria. Proceedings**.** Victoria, BC, Canadá, 2013. Disponível em: <doi10.1007/978-3-642-38986-3_10>. Acesso em: 28 maio 2018.

GRILO, A. B. **Computação quântica e teoria de computação**. 2014. 155 p. Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação, Campinas, SP. Disponível em: < http://repositorio.unicamp.br/jspui/handle/REPOSIP/275508>. Acesso em: 2 nov. 2017.

JOSÉ, M. A.; PIQUEIRA, J. R. C.; LOPES, R. D. **Introdução à programação quântica.** Revista Brasileira de Ensino de Física, [s.l.], v. 35, n. 1, p.2-9, mar. 2013. FapUNIFESP (SciELO). http://dx.doi.org/10.1590/s1806-11172013000100006.

KUCHLING, A. M.. **Functional Programming HOWTO**. 2001. Disponível em: <https://docs.python.org/3/howto/functional.html>. Acesso em: 28 maio 2018.

SHIFFMAN, D. **Coordenadas polares**. 2003. Disponível em: <https://pt.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-angular-movement/a/polar-coordinates>. Acesso em: 01 de dez de 2017.

STEINBRUCH, A.; WINTERLE, P. **Introdução à álgebra linear**. 2°. ed. [S.l.]: Makron Books, 1995. 245 p.

STRUBELL, E. **An Introduction to Quantum Algorithms**, Springer, vol. COS498, 2011.

NASCIMENTO, M C.. **Coordenadas Polares**. 2005. Disponível em: <http://wwwp.fc.unesp.br/~mauri/Down/Polares.pdf>. Acesso em: 01 de dez de 2017.

NIELSEN, M. A.; CHUANG, I. L. **Quantum Computation and Quantum Information**. Cambridge: Cambridge University Press, 2010. 698 p.

ÖMER, B. **Structured Quantum Programming**. 2003. 130 f. Dissertação, TU Vienna, 2003.

SIDDIQUI, S.; ISLAM, M. J.; SHEHAB, O. **Five Quantum Algorithms Using Quipper**. 2014. 27 f. Tese (Doutorado) - Curso de *Quantum Physics, University Of Maryland, Maryland*, 2014. Disponível em: <https://arxiv.org/abs/1406.4481>. Acesso em: 08 jun. 2018.

THOMPSON, S. **Haskell: the craft of functional programming**. 3. ed. Boston: Addison-Wesley, 2011.

TUCKER, A. B.; NOONAN, R. E. **Programming Languages: Principles and Paradigms**. 2. ed. Boston: McGraw-Hill Higher Education, 2007.

VENNERS, B. **The Making of Python**. 2003. Disponível em: <https://www.artima.com/intv/pythonP.html>. Acesso em: 28 maio 2018.

VIGNATTI, A. L., NETTO, F. S., BITTENCOURT, L. F. **Uma introdução à Computação Quântica**, 2004 (Trabalho de conclusão de curso).