

ANIMA EDUCAÇÃO
UNIVERSIDADE SÃO JUDAS TADEU

PAULO EMÍLIO NERY ASSIS OLIVEIRA
NOEMI SOARES GONÇALVES DA SILVA

HUB QUANTUM LAB - COMPUTAÇÃO QUÂNTICA
LINGUAGEM JULIA NA PROGRAMAÇÃO QUÂNTICA

SÃO PAULO – SP

2023

PAULO EMÍLIO NERY ASSIS OLIVEIRA
NOEMI SOARES GONÇALVES DA SILVA

HUB QUANTUM LAB - COMPUTAÇÃO QUÂNTICA
LINGUAGEM JULIA NA PROGRAMAÇÃO QUÂNTICA

**Trabalho de pesquisa apresentado ao Hub
Quantum Lab - Computação Quântica, sob
orientação da Professora Dra M. Ines Brosso.**

SÃO PAULO – SP

2023

Índice

Introdução.....	4
Linguagem Julia: Visão Geral.....	5
Fundamentos da Programação Quântica.....	6
Bibliotecas de Programação Quântica em Julia.....	7
Exemplos Práticos.....	8
Jupyter e Julia.....	11
Conclusão.....	13
Referências.....	14

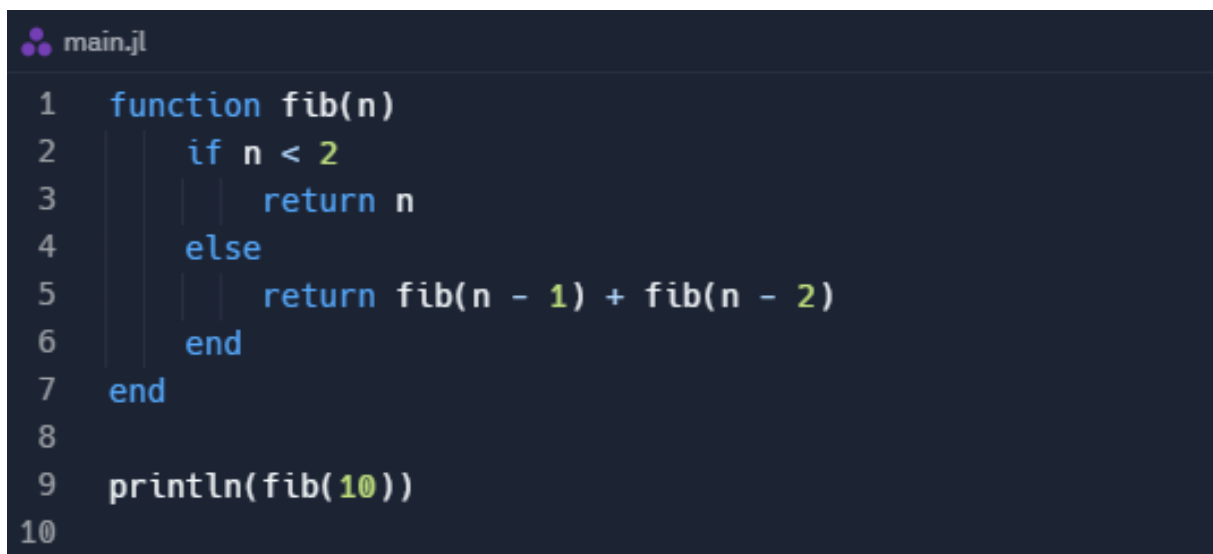
Introdução

A linguagem Julia tem se destacado como uma poderosa ferramenta para a programação quântica, oferecendo um ambiente altamente eficiente e de fácil utilização. Combinando as características das linguagens dinâmicas com a eficiência das linguagens compiladas, Julia apresenta uma sintaxe simples e flexível que torna a implementação de algoritmos complexos mais acessível, semelhante ao MATLAB. Sua infraestrutura otimizada permite lidar com operações matemáticas complexas de forma eficiente, enquanto sua sintaxe amigável e expressiva facilita a codificação e a leitura do código. Além disso, a linguagem Julia possui uma vasta coleção de pacotes e bibliotecas especializadas, permitindo que os desenvolvedores aproveitem recursos adicionais sem precisar reinventar a roda. Neste artigo, exploraremos a visão geral da linguagem Julia, abordando seus fundamentos, bibliotecas de programação quântica e exemplos práticos de algoritmos quânticos implementados em Julia.

Linguagem Julia: Visão Geral

A linguagem Julia foi desenvolvida com o propósito de oferecer um ambiente de programação altamente eficiente e de fácil utilização para computação quântica. Ao combinar as características das linguagens dinâmicas com a eficiência das linguagens compiladas, Julia apresenta uma sintaxe simples e flexível, semelhante ao MATLAB, o que torna a implementação de algoritmos complexos mais acessível.

Figura 1: Exemplo de código em Julia

A screenshot of a code editor window titled 'main.jl'. The code is written in Julia and implements a Fibonacci function. The code is as follows:

```
1 function fib(n)
2     if n < 2
3         return n
4     else
5         return fib(n - 1) + fib(n - 2)
6     end
7 end
8
9 println(fib(10))
10
```

Neste exemplo, demonstramos a implementação da função de Fibonacci em Julia. A linguagem permite uma escrita clara e concisa, facilitando a compreensão do algoritmo em questão.

No código acima, a função `fibonacci` recebe um número `n` como argumento e retorna o n-ésimo número de Fibonacci. A estrutura de controle `if-else` é utilizada para tratar os casos base, em que `n` é menor ou igual a 1, e a recursão é empregada para calcular os valores para `n` maiores que 1.

A principal vantagem da linguagem Julia é sua combinação única de desempenho e facilidade de uso. Ao ser projetada para computação quântica, Julia consegue lidar com operações matemáticas complexas de forma eficiente, graças à sua infraestrutura otimizada. Além disso, sua sintaxe amigável e expressiva torna a codificação e a leitura do código mais intuitivas.

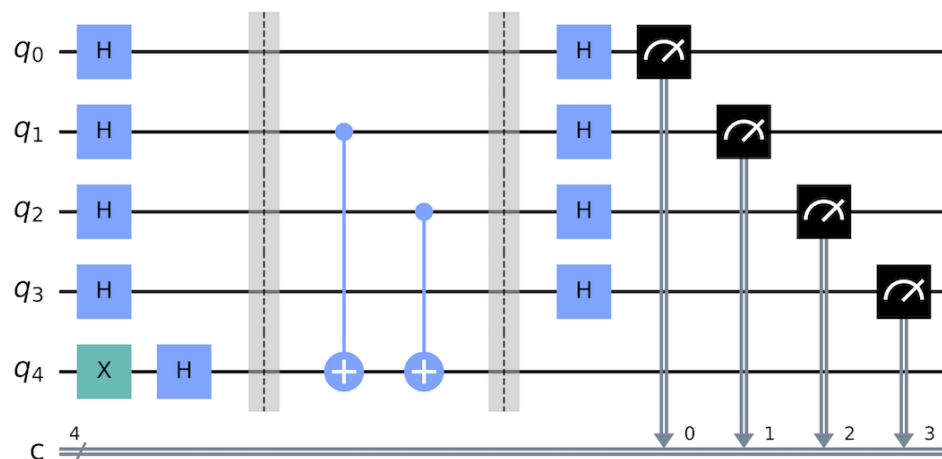
Julia também possui uma ampla gama de pacotes e bibliotecas disponíveis, permitindo que os desenvolvedores aproveitem funcionalidades adicionais sem precisar reinventar a roda. Essa extensa coleção de recursos é um dos pontos fortes da linguagem, facilitando a implementação de soluções inovadoras em diversas áreas, como ciência de dados, aprendizado de máquina e simulações físicas.

Em suma, a linguagem Julia oferece uma combinação poderosa de desempenho e usabilidade, tornando-a uma opção atraente para projetos que envolvem computação quântica e algoritmos complexos. Sua sintaxe simples e flexível, aliada à vasta coleção de pacotes disponíveis, torna Julia uma ferramenta valiosa para cientistas e engenheiros que buscam explorar e inovar na área da programação científica.

Fundamentos da Programação Quântica

Antes de explorarmos o uso da linguagem Julia na programação quântica, é essencial compreender os conceitos fundamentais dessa área. A computação quântica é baseada nos princípios da mecânica quântica, que descreve o comportamento das partículas em escalas atômicas e subatômicas.

Figura 2: Circuito quântico



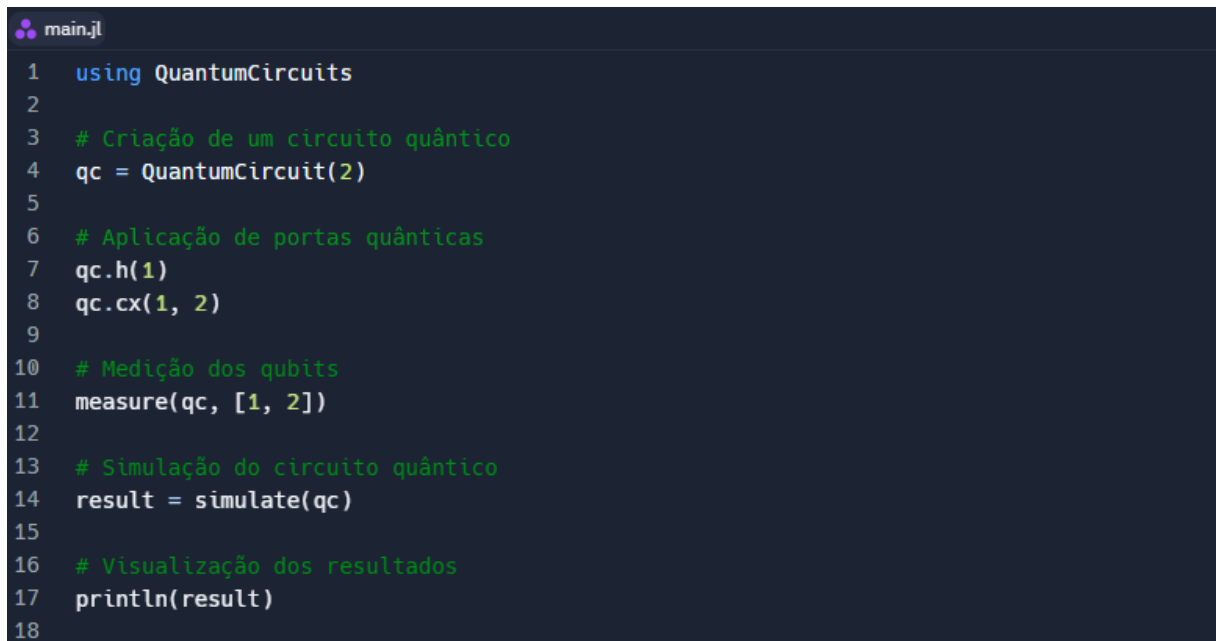
A figura anterior ilustra um circuito quântico, uma representação visual de um algoritmo quântico. Nesse circuito, são aplicadas portas quânticas (simbolizadas por símbolos específicos) em qubits, que são os equivalentes quânticos dos tradicionais bits clássicos. Ao

contrário dos bits clássicos, os qubits podem estar em uma superposição de estados, o que possibilita a execução de cálculos paralelos em um único circuito. Essa característica intrínseca da computação quântica permite explorar vastas quantidades de informações simultaneamente, oferecendo o potencial para resolver problemas complexos de maneiras eficientes e revolucionárias.

Bibliotecas de Programação Quântica em Julia

Uma das principais vantagens de utilizar a linguagem Julia para programação quântica é a disponibilidade de bibliotecas especializadas. Essas bibliotecas fornecem ferramentas e funções específicas para a manipulação de qubits, simulação de circuitos quânticos e implementação de algoritmos quânticos.

Figura 3: Exemplo de código utilizando a biblioteca QuantumCircuits.jl



```
main.jl
1  using QuantumCircuits
2
3  # Criação de um circuito quântico
4  qc = QuantumCircuit(2)
5
6  # Aplicação de portas quânticas
7  qc.h(1)
8  qc.cx(1, 2)
9
10 # Medição dos qubits
11 measure(qc, [1, 2])
12
13 # Simulação do circuito quântico
14 result = simulate(qc)
15
16 # Visualização dos resultados
17 println(result)
18
```

Nesse exemplo da figura anterior, utilizamos a biblioteca QuantumCircuits.jl para criar um circuito quântico de 2 qubits. Inicializamos o circuito usando a função QuantumCircuit() e especificamos o número de qubits como 2. Em seguida, aplicamos uma porta Hadamard

(representada por `h()`) no primeiro qubit e uma porta CNOT (representada por `cx()`) entre os dois qubits.

Para realizar a medição dos qubits, utilizamos a função `measure()` passando como argumento o circuito `qc` e uma lista dos qubits a serem medidos. Por fim, realizamos a simulação do circuito usando a função `simulate()` e armazenamos o resultado na variável `result`.

Para visualizar os resultados, utilizamos a função `println()` para imprimir o conteúdo da variável `result`.

Essa biblioteca é apenas uma das várias opções disponíveis em Julia para programação quântica, e cada uma delas oferece recursos específicos e vantagens distintas. Explorar essas bibliotecas pode ser fundamental para o desenvolvimento de aplicações quânticas eficientes e poderosas utilizando a linguagem Julia.

Exemplos Práticos

Para ilustrar de forma mais abrangente o uso da linguagem Julia na programação quântica, apresentaremos alguns exemplos práticos de algoritmos quânticos populares.

Algoritmo de Grover:

Figura 4: Exemplo de código para o algoritmo de Grover

```
main.jl
1  using QuantumCircuits
2
3  # Definição do número de qubits e o estado desejado
4  n = 3
5  target = [0, 0, 1]
6
7  # Criação do circuito quântico
8  qc = QuantumCircuit(n)
9
10 # Implementação do algoritmo de Grover
11 grover(qc, target)
12
13 # Simulação e visualização dos resultados
14 result = simulate(qc)
15 println(result)
16
```


Neste exemplo específico, estamos utilizando novamente a biblioteca `QuantumCircuits.jl` para implementar o algoritmo de Grover, um algoritmo de busca quântica amplamente conhecido e utilizado. Inicialmente, definimos o número de qubits desejado para o circuito, no caso, são utilizados 3 qubits. Em seguida, especificamos o estado desejado, representado aqui pelo vetor $[0, 0, 1]$, indicando que o estado alvo é o terceiro estado quântico possível.

Com base nessas definições, criamos o circuito quântico utilizando a função `QuantumCircuit(n)`, onde n representa o número de qubits do circuito. Uma vez criado o circuito, aplicamos a função `grover(qc, target)` para realizar a busca quântica utilizando o algoritmo de Grover. Essa função é responsável por aplicar as transformações necessárias no circuito para buscar o estado desejado.

Por fim, simulamos o circuito quântico utilizando a função `simulate(qc)` e armazenamos o resultado na variável `result`. Para exibir os resultados da simulação, utilizamos o comando `println(result)`, que mostra as informações relevantes obtidas durante a execução do algoritmo.

Dessa forma, podemos visualizar e analisar os resultados da busca quântica realizada pelo algoritmo de Grover, fornecendo uma demonstração prática do uso da linguagem Julia na programação quântica.

Calculo de Massa Corporal:

Figura 5: Exemplo de Cálculo de Massa Corporal

```
main.jl
1  function calcular_massa_corporal(peso, altura)
2      altura_metros = altura / 100
3      imc = peso / (altura_metros^2)
4      return imc
5  end
6
7  peso = 70
8  altura = 170
9
10 massa_corporal = calcular_massa_corporal(peso, altura)
11 println("A massa corporal é:", massa_corporal)
12
```

Neste exemplo, a função `calcular_massa_corporal` recebe o peso em quilogramas e a altura em centímetros como argumentos. Em seguida, converte a altura para metros e calcula o índice de massa corporal (IMC) usando a fórmula $\text{peso} / (\text{altura_metros}^2)$. O resultado é retornado e impresso na tela.

Cifra de César:

Figura 6: Criptografia - Cifra de César

```
main.jl
1  function cifra_cesar(texto, deslocamento)
2      alfabeto = 'A':'Z'
3      texto_cifrado = ""
4      for caractere in texto
5          if caractere in alfabeto
6              indice = findfirst(isequal(caractere), alfabeto)
7              novo_indice = mod(indice + deslocamento - 1, length(alfabeto)) + 1
8              novo_caractere = alfabeto[novo_indice]
9              texto_cifrado *= novo_caractere
10         else
11             texto_cifrado *= caractere
12         end
13     end
14     return texto_cifrado
15 end
16
17 mensagem = "HELLO WORLD"
18 deslocamento = 3
19
20 mensagem_cifrada = cifra_cesar(mensagem, deslocamento)
21 println("Mensagem cifrada:", mensagem_cifrada)
```

Neste exemplo, a função `cifra_cesar` implementa a cifra de César, um método de criptografia simples que desloca cada letra do alfabeto em um determinado número de posições. O texto a ser cifrado e o deslocamento são passados como argumentos. A função percorre cada caractere do texto, verifica se está presente no alfabeto e realiza o deslocamento. O texto cifrado é construído e retornado.

Criptografia por Substituição:

Figura 7: Criptografia por Substituição

```
main.jl
1 function criptografia_substituicao(texto, chave)
2     alfabeto = 'A':'Z'
3     texto_cifrado = ""
4     for caractere in texto
5         if caractere in alfabeto
6             indice = findfirst(isequal(caractere), alfabeto)
7             novo_caractere = chave[indice]
8             texto_cifrado *= novo_caractere
9         else
10            texto_cifrado *= caractere
11        end
12    end
13    return texto_cifrado
14 end
15
16 mensagem = "HELLO WORLD"
17 chave = "QWERTYUIOPASDFGHJKLZXCVBNM"
18
19 mensagem_cifrada = criptografia_substituicao(mensagem, chave)
20 println("Mensagem cifrada:", mensagem_cifrada)
```

Neste exemplo, a função `criptografia_substituicao` implementa um método de criptografia por substituição, onde cada letra do alfabeto é substituída por outra letra de acordo com uma chave fornecida. O texto a ser cifrado e a chave de substituição são passados como argumentos. A função percorre cada caractere do texto, verifica se está presente no alfabeto e realiza a substituição. O texto cifrado é construído e retornado.

Esses exemplos adicionais demonstram algumas aplicações práticas da linguagem Julia, abrangendo cálculos de massa corporal e técnicas de criptografia. A linguagem Julia oferece uma ampla variedade de recursos e bibliotecas para abordar problemas em diversas áreas, permitindo aos desenvolvedores explorar e inovar em suas aplicações.

Jupyter e Julia

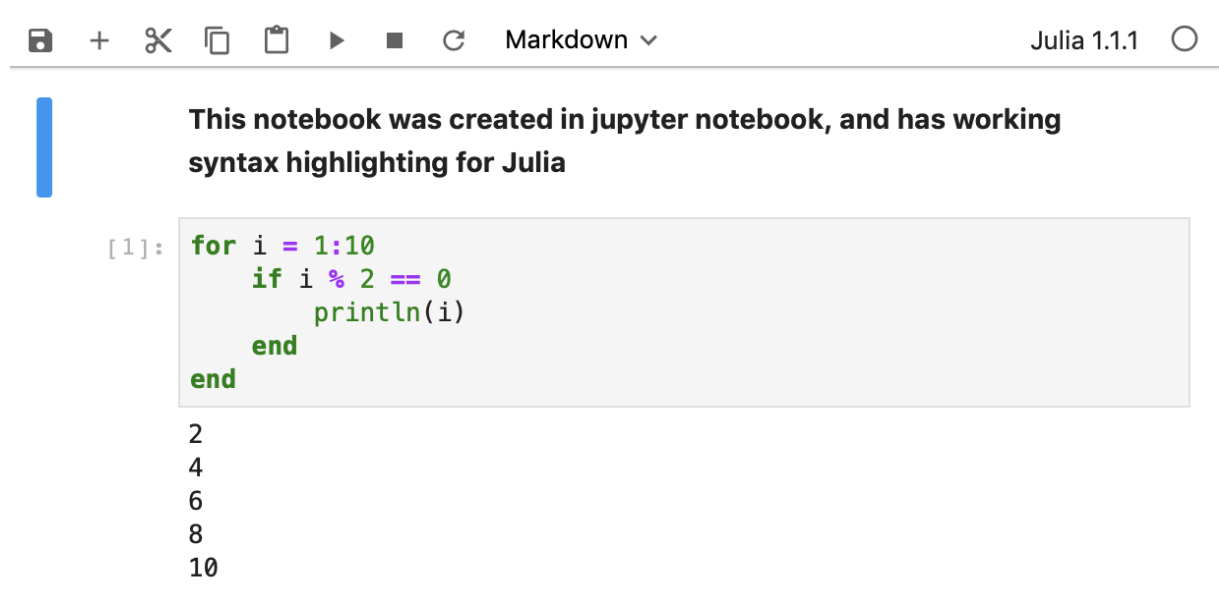
Uma das vantagens de usar a linguagem Julia é sua compatibilidade com o ambiente Jupyter. O Jupyter permite a programação interativa por meio de notebooks com código, visualizações e texto explicativo. Com essa integração, os usuários de Julia podem aproveitar

recursos como execução de código em células, visualização de gráficos e criação de relatórios interativos.

A combinação do Jupyter com Julia facilita a exploração e experimentação de algoritmos quânticos. Os notebooks Jupyter permitem escrever e executar código Julia em células individuais, tornando a experimentação iterativa e a depuração mais eficientes. Além disso, é possível criar visualizações interativas para representar circuitos quânticos e resultados de simulações, auxiliando na compreensão e comunicação dos conceitos da programação quântica.

Outra vantagem é a capacidade de incluir texto explicativo, formatação rica e equações matemáticas nos notebooks Jupyter. Isso permite documentar descobertas, explicar conceitos e fornecer contexto para o código implementado. A combinação do Jupyter com a linguagem Julia é uma ferramenta poderosa para o desenvolvimento e colaboração em projetos de programação quântica, oferecendo uma forma eficiente e interativa de explorar, experimentar e documentar algoritmos quânticos.

Figura 8: Exemplo linguagem Julia no Jupyter



Nesse exemplo temos um Markdown dizendo: “Este notebook foi criado no Jupyter Notebook e possui destaque de sintaxe funcional para Julia”, e um código em Julia que imprime os números pares de 1 a 10.

Conclusão

A linguagem Julia se destaca como uma escolha atraente para a programação quântica devido à sua combinação única de desempenho e facilidade de uso. Ao oferecer uma sintaxe simples e flexível, Julia simplifica a implementação de algoritmos complexos, tornando a programação quântica mais acessível para cientistas e engenheiros. Sua infraestrutura otimizada proporciona eficiência na execução de operações matemáticas complexas, enquanto sua vasta coleção de pacotes e bibliotecas especializadas expande as possibilidades da programação quântica em Julia. Com Julia, os desenvolvedores podem explorar todo o potencial da computação quântica, implementando soluções inovadoras em áreas como ciência de dados, aprendizado de máquina e simulações físicas. Com sua combinação poderosa de desempenho e usabilidade, a linguagem Julia está impulsionando a programação científica para novos patamares, oferecendo oportunidades emocionantes para avanços na computação quântica.

Referências

- Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. SIAM review, 59(1), 65-98.
- Gottesman, D. (1999). The Heisenberg representation of quantum computers. arXiv preprint quant-ph/9807006.
- QuantumCircuits.jl Documentation. Disponível em: <https://quantumcircuitsjl.readthedocs.io/en/latest/>. Acesso em: 28 mai. 2023.
- Grover, L. K. (1996). A fast quantum mechanical algorithm for database search. In Proceedings of the twenty-eighth annual ACM symposium on Theory of computing (pp. 212-219).
- QISKIT. Advanced Circuit Visualization. Disponível em: https://qiskit.org/documentation/stable/0.26/locale/pt_BR/tutorials/circuits_advanced/03_advanced_circuit_visualization.html. Acesso em: 06 jun. 2023.
- JULIA, The Julia Language. The Julia Language Documentation. São Francisco: Julia Computing, 2021.
- JULIA QUANTUM. Julia Quantum. Disponível em: <https://juliaquantum.github.io/>. Acesso em: 07 jun. 2023.