Relatório Técnico de Desenvolvimento

Projeto: Calculadora Matricial em Python

Charles Ribeiro Chaves - 122086950
Filipe Viana da Silva - 121050053
Vinícius Brasil de Oliveira Barreto - 120029237

8 de junho de 2025

Sumário

1 Introdução

Este documento detalha o processo de desenvolvimento do projeto "Calculadora Matricial", uma aplicação em Python que implementa os princípios do Paradigma Orientado a Objetos para realizar operações matriciais. O principal objetivo foi desenvolver um sistema coeso e eficiente, que otimiza o uso de memória e o tempo de execução ao lidar com tipos especiais de matrizes, como as Diagonais e Triangulares (Inferior e Superior).

A aplicação foi estruturada em torno de uma hierarquia de classes, utilizando herança e polimorfismo para prover implementações especializadas e otimizadas para cada tipo de matriz, ao mesmo tempo em que oferece uma interface de usuário interativa via console.

2 Estruturas de Dados Utilizadas

A escolha das estruturas de dados internas foi um aspecto crucial do projeto para garantir a otimização de espaço de memória, evitando o armazenamento de um grande volume de elementos nulos.

- **Matriz Geral (`MatrizGeral`):** Utiliza uma lista de listas (`List[List[float]]`) para representar uma matriz densa 'm x n'. Esta é a estrutura mais geral, servindo como base e fallback para operações entre tipos distintos de matrizes.
- **Matriz Diagonal (`MatrizDiagonal`):** Emprega uma lista simples (`List[float]`) para armazenar unicamente os 'n' elementos da diagonal principal. Esta é a representação mais econômica em termos de espaço.
- **Matrizes Triangulares (`MatrizTriangularInferior/Superior`):** Utilizam uma lista de listas com tamanhos variáveis (*jagged array*). Esta abordagem armazena apenas os elementos não nulos, resultando em uma economia de espaço de quase 50% em comparação com a representação densa de uma matriz 'n x n'.

3 Divisão de Módulos e Arquitetura

A arquitetura do sistema foi projetada em torno de uma hierarquia de classes e uma classe controladora, separando a lógica de negócio da interação com o usuário.

3.1 Hierarquia de Classes

O núcleo do programa é um sistema de classes que explora herança e polimorfismo, a partir de uma classe base abstrata.

- `Matriz`: Uma Classe Base Abstrata (ABC) que define a interface comum para todas as matrizes. Ela utiliza o decorador `@abstractmethod` para forçar as classes filhas a implementarem métodos essenciais como `__add__`, `__mul__`, `transposta`, etc.

- Classes Concretas: `MatrizGeral`, `MatrizDiagonal`, `MatrizTriangularInferior` e `MatrizTriangularSuperior` herdam de `Matriz` e fornecem implementações concretas e otimizadas dos métodos abstratos.

3.2 Classe Controladora (`CalculadoraMatricial`)

A classe `CalculadoraMatricial` atua como o módulo principal da aplicação. Suas responsabilidades incluem:

- Gerenciar uma lista de objetos de matriz (`self.lista_matrizes`).
- Apresentar uma interface de menu de texto para o usuário.
- Orquestrar as operações, como ler os dados de uma nova matriz, selecionar operandos e invocar os métodos de cálculo.

Diferente da abordagem em C++, o código Python optou por não ter um *Factory Method* explícito, delegando a criação do tipo correto ao usuário no momento da entrada de dados, o que simplifica o fluxo de criação.

4 Descrição das Rotinas e Funções Principais

4.1 Sobrecarga de Operadores ("Dunder Methods")

O Python permite uma sintaxe natural para operações matemáticas através da implementação de "dunder methods"(métodos com duplo underscore).

- `__add__`, `__sub__`, `__mul__`: Foram sobrecarregados em cada classe. A lógica interna verifica se a operação está ocorrendo entre matrizes de mesmo tipo para usar um algoritmo otimizado. Caso contrário, a operação é delegada para a `MatrizGeral`, que converte as matrizes especializadas para o formato denso antes de calcular.

```
def __add__(self, other):
    if not isinstance(other, MatrizDiagonal):
        # Fallback: converte para geral e usa a soma genérica
        return MatrizGeral(self.linhas, self.colunas, self.to_array()) +
    other
    if self.linhas != other.linhas:
        raise ValueError("Dimensoes incompativeis para soma")
    # Via rapida: opera apenas nos vetores 1D
    resultado = MatrizDiagonal(self.linhas)
    for i in range(self.linhas):
        resultado.diagonal[i] = self.diagonal[i] + other.diagonal[i]
    return resultado
```

Listing 1: Exemplo de soma otimizada e com fallback em MatrizDiagonal.

4.2 Métodos Especializados (`traço` e `determinante`)

As operações de `traço` e `determinante` foram implementadas de forma otimizada nas classes onde fazem sentido.

- **Traço**: Implementado em todas as classes de matrizes quadradas, somando os elementos da diagonal principal.
- **Determinante**: Implementado de forma eficiente para `MatrizDiagonal` e as classes Triangulares, calculando o produto dos elementos da diagonal. A classe base lança um `NotImplementedError` para indicar que a operação não é universal.

5 Análise de Complexidade

5.1 Complexidade de Espaço

A otimização de memória é um dos pontos fortes do projeto. Para uma matriz quadrada de ordem ‘n‘:

- **MatrizGeral:** $O(n^2)$
- **MatrizTriangular:** $O(\frac{n(n+1)}{2}) = O(n^2)$
- **MatrizDiagonal:** $O(n)$

5.2 Complexidade de Tempo

O polimorfismo permitiu a implementação de algoritmos com complexidade de tempo reduzida para casos específicos.

- **Soma de Matrizes (A + B):**
  - Genérica: $O(n^2)$
  - Diagonal + Diagonal: $O(n)$
  - Triangular + Triangular: $O(n^2)$ (mas com menos operações que a genérica)
- **Determinante:**
  - Genérico (não implementado): Normalmente $O(n^3)$
  - Triangular ou Diagonal: $O(n)$
- **Multiplicação (Matriz x Matriz):**
  - Genérica: $O(n^3)$ para matrizes quadradas.

6 Problemas e Observações Encontrados

1. **Lógica de Transposição Incorreta:** Um bug foi identificado na rotina de transposição da classe `MatrizTriangularSuperior` e `MatrizTriangularInferior`, onde a atribuição dos índices estava incorreta para a estrutura de dados otimizada. A correção envolveu garantir que o elemento `dados[i][j]` da matriz original fosse corretamente mapeado para a posição `dados[j][i]` na estrutura da matriz transposta.

2. **Cálculo do Traço em `MatrizTriangularSuperior`:** A implementação inicial do método `traço` estava incorreta, pois não considerava o mapeamento de índices da estrutura otimizada. A correção foi acessar `self.dados[i][0]`, pois o elemento da diagonal principal de uma linha 'i' sempre corresponde ao primeiro elemento de seu vetor interno.

3. **Design da Hierarquia:** O código Python reflete uma implementação direta da hierarquia solicitada. A decisão de não ter uma classe `MatrizQuadrada` explícita, mas sim uma verificação com o método `eh_quadrada()`, é uma abordagem válida e idiomática em Python, embora diferente da implementação em C++ que se beneficia de uma hierarquia mais rígida para o sistema de tipos.

7 Conclusão

O projeto da Calculadora Matricial em Python foi implementado com sucesso, cumprindo todos os requisitos funcionais e de otimização. A utilização de uma Classe Base Abstrata e a sobrecarga de operadores resultaram em um código limpo, legível e que demonstra claramente os benefícios da programação orientada a objetos.

Os resultados confirmam que a arquitetura escolhida é eficiente tanto em uso de memória quanto em tempo de execução para os casos especializados, ao mesmo tempo que mantém a flexibilidade para operar com matrizes de tipos diferentes através de um mecanismo de fallback. O projeto serve como um excelente exemplo prático da aplicação de herança e polimorfismo para resolver um problema computacional de forma elegante e eficaz.