Rust y Álgebra Lineal: Un Enfoque Moderno para el Manejo de Matrices

Análisis y Desarrollo con Rust

***Abstract*—El álgebra lineal constituye uno de los pilares fundamentales en computación científica, gráficos computacionales, simulación física y aprendizaje automático. En programación, trabajar con matrices y vectores es esencial para modelar sistemas, transformar datos y representar operaciones matemáticas complejas. Rust, un lenguaje moderno orientado a la seguridad y el rendimiento, proporciona herramientas de bajo y alto nivel para construir estructuras matriciales sin comprometer la seguridad de memoria. Este documento presenta una introducción conceptual al álgebra lineal aplicada en Rust, describiendo cómo diseñar estructuras matriciales seguras y eficientes, y mostrando ejemplos idiomáticos de operaciones fundamentales.**

I. Introducción

El álgebra lineal en programación requiere estructuras capaces de representar arreglos multidimensionales, vectores y transformaciones. En lenguajes tradicionales como C, las matrices suelen implementarse mediante arreglos bidimensionales o punteros dobles. Sin embargo, estos enfoques implican riesgos relacionados con la gestión manual de memoria.

Rust introduce un modelo seguro basado en:
1) **Propiedad (Ownership)**
2) **Préstamos (Borrowing)**
3) **Verificaciones en tiempo de compilación (Compile-time checks)**

Estos mecanismos permiten representar matrices mediante estructuras (*structs*), arreglos estáticos (*arrays*) y vectores dinámicos (`Vec<T>`) de forma segura y eficiente.

Estructuras como `struct`, combinadas con el sistema de tipos fuerte del lenguaje, permiten abstraer matrices de tamaño fijo, mientras que tipos dinámicos como `Vec<Vec<f64>>` permiten operar con matrices de tamaño variable. En este documento se describen ambos enfoques.

II. Representación de Matrices en Rust

Existen dos aproximaciones principales para manejar matrices en Rust:
1) Matrices de tamaño fijo: utilizando `[[T; N]; M]`
2) Matrices dinámicas: utilizando `Vec<Vec<T>>`

La primera opción se utiliza comúnmente en escenarios donde el tamaño es conocido en tiempo de compilación (por ejemplo, transformaciones 3x3 o 4x4 en gráficos). La segunda es apropiada para aplicaciones más generales.

A. Ejemplo de matriz de tamaño fijo

```
struct Matrix3x3 {
    data: [[f64; 3]; 3],
}
```

Esta estructura garantiza seguridad en memoria y permite indexación como en C, pero con checado de límites.

B. Ejemplo de matriz dinámica

```
struct Matrix {
    rows: usize,
    cols: usize,
    data: Vec<Vec<f64>>,
}
```

Esto permite crear matrices de cualquier dimensión, validando durante la construcción que todos los vectores internos tienen la misma longitud.

III. Operaciones Básicas

A continuación se presentan ejemplos idiomáticos de Rust para operaciones fundamentales en álgebra lineal.

A. Suma de matrices

```
impl Matrix3x3 {
    fn add(&self, other: &Self) -> Self {
        let mut result = [[0.0; 3]; 3];
        for i in 0..3 {
            for j in 0..3 {
                result[i][j] = self.data[i][j]
                    + other.data[i][j];
            }
        }
        Self { data: result }
    }
}
```

La suma de matrices requiere que ambas matrices tengan las mismas dimensiones, lo cual el sistema de tipos garantiza al usar matrices fijas.

B. Multiplicación escalar

```
fn scalar_mul(&self, k: f64) -> Self {
    let mut result = [[0.0; 3]; 3];
    for i in 0..3 {
        for j in 0..3 {
            result[i][j] = self.data[i][j] * k
                ;
        }
```

```
    }
    Self { data: result }
}
```

C. Multiplicación matricial

```
fn mul(&self, other: &Self) -> Self {
    let mut result = [[0.0; 3]; 3];
    for i in 0..3 {
        for j in 0..3 {
            for k in 0..3 {
                result[i][j] += self.data[i][k
                    ] * other.data[k][j];
            }
        }
    }
    Self { data: result }
}
```

La multiplicación matricial es un ejemplo clásico de cómo Rust permite expresar algoritmos matemáticos de manera clara, manteniendo seguridad en el acceso a índices.

IV. Consideraciones y Buenas Prácticas

- Encapsular siempre la representación interna usando `struct`.
- Evitar exponer acceso directo al array interno; usar funciones `get` y `set`.
- Usar `Result<T, E>` al trabajar con matrices dinámicas para capturar errores de dimensión.
- Apoyarse en iteradores cuando sea posible para mejorar legibilidad y eficiencia.
- Consultar el Rust Book para fundamentos en: Ownership (Cap. 4), Structs (Cap. 5), Enums (Cap. 6) y Vectores (Cap. 8).

V. Conclusión

Rust ofrece herramientas modernas para implementar álgebra lineal de manera segura, expresiva y eficiente. El uso de matrices como arrays fijos garantiza seguridad en tiempo de compilación, mientras que las matrices dinámicas mediante `Vec` ofrecen flexibilidad. Su sistema de tipos fuerte y sus garantías de memoria convierten a Rust en una excelente opción para aplicaciones científicas, gráficas y de alto rendimiento.

Referencias

References

[1] Steve Klabnik y Carol Nichols, “The Rust Programming Language”, Disponible en: https://doc.rust-lang.org/book/

[2] Rust Standard Library Documentation, https://doc.rust-lang.org/std/