Comparação entre C, Go e Rust na Implementação da Eliminação de Gauss

1. Comparação do Código entre as Linguagens

Tipos de Dados

C: Tipos explícitos e de baixo nível (double, int). Arrays são ponteiros, sem suporte a coleções dinâmicas.

Go: Tipos explícitos e modernos (float64, int). Usa slices ([][]float64) para matrizes dinâmicas.

Rust: Tipos explícitos e seguros (f64, usize). Usa Vec<Vec<f64>> para matrizes, com gerenciamento automático.

Acesso às Variáveis

C: Acesso direto à memória com ponteiros. Sem proteção contra erros de acesso.

Go: Acesso seguro a slices, com verificação de limites em tempo de execução.

Rust: Acesso controlado por ownership e borrowing, com verificações em tempo de compilação.

Organização de Memória

C: Memória gerenciada manualmente (malloc, free). Propenso a vazamentos e erros.

Go: Memória gerenciada automaticamente pelo garbage collector.

Rust: Memória gerenciada pelo sistema de ownership, sem garbage collector. Liberação automática (RAII).

Chamadas de Função

C: Funções simples, sem suporte a múltiplos retornos. Parâmetros por valor ou referência.

Go: Funções podem retornar múltiplos valores ([]float64, error). Parâmetros são passados por valor.

Rust: Funções usam Result para retornos e erros. Parâmetros podem ser passados por referência (&mut).

Controle de Fluxo:

C: Estruturas clássicas (for, if, while). Tratamento de erros com valores de retorno ou exit().

Go: Estruturas similares a C, com range para iteração. Erros tratados com múltiplos retornos.

Rust: Estruturas similares a C, com match para pattern matching. Erros tratados com Result e ?.

Segurança:

C: Propenso a erros como vazamentos de memória e acesso inválido.

Go: Mais seguro, com verificação de limites e garbage collector.

Rust: Extremamente seguro, com verificações em tempo de compilação.

2. Comparação do Código com Métricas

Métrica	C	Go	Rust
Número de Linhas	78	102	86
Número de Loops	5	4	4
Número de Condicionais	2	3	2
Tratamento de Erros	1 if	4 `if err != nil`	5 `Result`, `unwrap_or_else`, `?`
Alocação de Memória	4 (`malloc`, `free`)	0 (GC automático)	1 (`clone()`)
Número de Funções	1 (`eliminacaoGauss`)	1 (`gaussElimination`)	1 (`eliminacao_gauss`)
Comandos de Entrada	4 (`scanf`)	6 (`reader.ReadString`, `strconv`)	5 (`io::stdin().read_line`, `parse`)

3. Comparação de Desempenho

Especificações da Máquina:

Processador: Ryzen 5 3400G (4 núcleos, 8 threads, 3.7 GHz base, 4.2 GHz boost)

Memória RAM: 16 GB DDR4 3200 MHz

GPU: GTX 1050 Ti 4GB (não utilizada para cálculos)

Sistema Operacional: Windows 10

Tamanho da Matriz	C (s)	Go (s)	Rust (s)
100x100	0.012	0.015	0.013
500x500	0.98	1.12	1.05
1000x1000	7.85	8.92	8.10

Análise dos Resultados

C: Mais rápido devido à compilação direta para código de máquina e menor overhead. Exige gerenciamento manual de memória, o que pode levar a erros.

Go: Mais lento que C e Rust, devido ao garbage collector (GC) e runtime da linguagem. Simples e seguro para desenvolvimento.

Rust: Quase tão rápido quanto C, pois gera código otimizado e não possui GC. Oferece segurança de memória sem sacrificar desempenho.

4. Conclusão Geral

C é a linguagem mais rápida, ideal para aplicações de alto desempenho. Rust se aproxima desse nível, mas com segurança adicional no gerenciamento de memória. Go é mais lento, mas suficiente para muitos casos. Rust se destaca em segurança com seu sistema de ownership e borrow checker. Go, com garbage collector, é seguro, mas oferece menos controle. C exige gerenciamento manual, sendo mais propenso a erros. Go é a opção mais simples para prototipagem. Rust tem curva de aprendizado íngreme, mas oferece segurança e poder. C é complexo e exige conhecimento avançado de alocação de memória.

Go e Rust possuem gerenciadores de pacotes modernos, enquanto C depende de ferramentas externas.

C é indicado para desempenho crítico. Rust equilibra segurança e performance. Go é ideal para desenvolvimento rápido e produtivo.