Implementação da Eliminação de Gauss

Comparação entre os códigos (considerando tipos de dados, acesso às variáveis, organização de memória, chamadas de função e comandos de controle de fluxo)

1. Tipos de dados

   C

   - Usa int para dimensões (N).
   - Usa float para a matriz A, vetor B e vetor solução X.
   - Variáveis globais como volatile float A[MAXN][MAXN], o que pode evitar otimizações excessivas do compilador.

   Rust

   - Usa usize para índices e dimensões (mais seguro para indexação de arrays).
   - Usa f32 para a matriz A, vetor B e vetor X.
   - Não há variáveis globais mutáveis, pois Rust desencoraja isso.

   Go

   - Usa int para N, mantendo consistência com outras partes do código.
   - Usa float64 para A, B e X (mais preciso que float32, o que pode impactar o desempenho em algumas arquiteturas).
   - Declaração explícita de constantes MAXN e intervalos de números aleatórios.

2. Acesso a variáveis

   C

   - Usa variáveis globais (A[MAXN][MAXN], B[MAXN], X[MAXN]).
   - Permite acesso direto a arrays, sem verificação de limites.

   Rust

   - As matrizes A, B e X são armazenadas em Vec<Vec<f32>> e Vec<f32>.
   - Rust faz verificações de limites ao acessar índices de vetores (A[i][j]) pode causar panic! se o índice estiver fora dos limites.

   Go

- Usa arrays globais (var A [MAXN][MAXN]float64).
- Arrays em Go são passados por um valor, mas pode-se usar slices ([]float64) para evitar cópias desnecessárias.
- O acesso também inclui verificação de limites, mas sem panic! automático como em Rust.

3. Organização de memória

C

- Arrays são alocados estaticamente (volatile float A[MAXN][MAXN]), o que pode desperdiçar memória quando N é pequeno.
- Uso de ponteiros implícitos ao acessar arrays, sem verificações de segurança.

Rust

- Usa alocação dinâmica com Vec, ajustando a memória ao tamanho N.
- O gerenciamento de memória é feito automaticamente pelo ownership system (sem malloc ou free).

Go

- A matriz A e os vetores B e X são declarados como arrays fixos (var A [MAXN][MAXN] float64).
- Go tem garbage collector, então a memória é gerenciada automaticamente.

4. Chamadas de função

C

- Usa funções tradicionais (void gauss(), void initialize_inputs()).
- Passagem de arrays ocorre por referência, pois C trata arrays como ponteiros.

Rust

- Usa funções fn (fn gauss()), seguindo um modelo mais funcional.
- Passa arrays como &mut Vec<Vec<f32>> (referência mutável), garantindo segurança.
- Sem ponteiros explícitos, apenas referências seguras.

Go

- Usa func (func gauss()), e as variáveis globais são acessadas diretamente.
- Arrays são passados por valor, então slices são preferidos para evitar cópias.

- Go permite defer para executar código final de uma função (usado para medir tempo).

5. Comandos de controle de fluxo

   C

   - Usa for tradicional (for (i=0; i < N; i++)).
   - if e while seguem a sintaxe clássica de C.

   Rust

   - Usa for i in 0..size{} (mais intuitivo).
   - Laços invertidos usam .rev() (for row in (0..size-1).rev()).
   - if é mais restrito, exigindo bool explícito (if x > 0 {}).

   Go

   - Usa for i := 0; i < N; i++ {} para loops tradicionais.
   - Não tem while, usa for com condição (for condition {}).
   - Introduz defer para adiar execução do código.

Comparação entre os códigos (com base no número de linhas e funções)

1. Número de linhas

   Go

   - 153 linhas.

   C

   - 194 linhas.

   Rust

   - 91 linhas.

2. Número de funções

   Go

   - 7 funções

   C

   - 6 funções

   Rust

   - 2 funções

Conclusão

A solução original (em C) é altamente eficiente, mas propensa a erros de memória e difícil de manter. A versão em Rust melhora a segurança, evitando acessos inválidos e garantindo alocação segura sem comprometer o desempenho. Já a implementação em Go se destaca pela simplicidade e legibilidade, embora tenha um pequeno overhead devido ao garbage collector.

Se o foco for máximo desempenho, C ainda é uma boa escolha. Para segurança e eficiência, Rust é superior. E para um código mais simples e fácil de manter, Go oferece um ótimo equilíbrio. As soluções representam um avanço significativo em robustez e organização.

Comparação de desempenho

Linguagem	**Dimensão**	**Tempo**
C	5	0.1576 ms
	50	0.1973 ms
	100	0.6608 ms
	500	63.7529 ms
	1000	483.043 ms
	2000	3908.72 ms
Rust	5	104.00 us
	50	3.02 ms
	100	18.52 ms
	500	1.81 s
	1000	14.2 s
	2000	111.16 s
Go	5	1.5325 ms
	50	
	100	509.1 µs
	500	24.6231 ms
	1000	180.5434 ms
	2000	1.9291331 s

Observações:

- A respeito da implementação em Go, no intervalo de dimensões 10-90, apesar do tempo de processamento esperado estar na representação dos milissegundos, o tempo impresso pelo programa é zero.
- O tempo de processamento de uma matriz de 100 dimensões está muito abaixo do esperado, considerando que o tempo de processamento de uma matriz de 5 dimensões foi aproximadamente 3 vezes maior.