UNIVERSIDADE FEDERAL DE PELOTAS
CENTRO DE DESENVOLVIMENTO TECNOLÓGICO
CIÊNCIA DA COMPUTAÇÃO
Conceitos de Linguagens de Programação

Comparação do Algoritmo da Eliminação de Gauss em C, Golang e Rust: Análise da Linguagem, Desempenho e Eficiência

Integrantes: Arthur Alves, Artur Gruppelli, Fabricio Bartz e Victor Reis

Pelotas, 2025

Introdução

A disciplina de Conceitos de Linguagens de Programação desempenha um papel fundamental na formação de profissionais da Ciência da Computação, pois fornece subsídios para a compreensão das diferenças e similaridades entre diversas linguagens, bem como suas aplicações em diferentes contextos.

Neste trabalho, será realizada uma comparação da implementação do algoritmo da Eliminação de Gauss em três linguagens distintas: C, Golang e Rust. O algoritmo de Eliminação de Gauss é amplamente utilizado para a resolução de sistemas de equações lineares, sendo uma ferramenta essencial em diversas áreas da matemática aplicada e da engenharia computacional.

A comparação será baseada em métricas como tempo de execução, uso de memória, complexidade de implementação e segurança da linguagem.

Ao final deste estudo, espera-se identificar as vantagens e limitações de cada linguagem no contexto da resolução de sistemas lineares, contribuindo para uma compreensão mais profunda de como diferentes abordagens impactam a eficiência e a segurança da programação em aplicações matemáticas.

Vamos comparar as implementações do algoritmo de Eliminação de Gauss em C, Golang e Rust, destacando as diferenças entre as linguagens em termos de tipos de dados, acesso às variáveis, organização de memória, chamadas de função, controle de fluxo e métricas como número de linhas e comandos.

Tipos de Dados

Em **C**, os tipos de dados são bastante simples e diretos. Para representar números reais, utiliza-se o tipo `float`. Matrizes e vetores são declarados como arrays estáticos, como `float A[MAXN][MAXN]` para uma matriz e `float B[MAXN]` para um vetor. O tamanho máximo da matriz é definido por uma constante `MAXN`, que é determinada em tempo de compilação. Essa abordagem é eficiente, mas limita a flexibilidade do código, pois o tamanho da matriz não pode ser alterado em tempo de execução.

Em **Golang**, os tipos de dados são um pouco mais modernos. Para números reais, utiliza-se `float64`, que oferece maior precisão em comparação ao `float` do C. Matrizes e vetores também são declarados como arrays, mas com uma sintaxe ligeiramente diferente, como `[MAXN][MAXN]float64` para uma matriz e `[MAXN]float64` para um vetor. Assim como em C, o tamanho máximo da matriz é definido por uma constante `MAXN`. Golang também oferece slices, que são mais flexíveis que arrays estáticos, mas na implementação do algoritmo de Gauss, arrays estáticos são suficientes.

Em **Rust**, os tipos de dados são semelhantes aos de Golang, mas com uma sintaxe mais moderna. Para números reais, utiliza-se `f64`, que é equivalente ao `float64` de Golang. Matrizes e vetores são declarados como arrays estáticos, como `[[f64; MAXN]; MAXN]` para uma matriz e `[f64; MAXN]` para um vetor. O tamanho máximo da matriz também é definido por uma constante `MAXN`. Rust, assim como C e Golang, utiliza arrays estáticos, mas com uma sintaxe mais explícita e segura.

Observação: Enquanto C e Rust utilizam arrays estáticos de forma semelhante, Golang oferece uma sintaxe um pouco diferente, mas ainda mantém a simplicidade e a eficiência dos arrays estáticos.

Acesso às Variáveis

Em **C**, o acesso às variáveis é direto e sem restrições. Isso significa que o programador pode acessar e modificar qualquer variável sem grandes obstáculos. No entanto, essa liberdade pode levar a problemas de segurança, especialmente quando se trata de variáveis globais. No caso do algoritmo de Gauss, a matriz e os vetores são frequentemente declarados como globais, o que facilita o acesso, mas também aumenta o risco de erros, como vazamentos de memória ou acesso indevido.

Em **Golang**, o acesso às variáveis também é direto, mas a linguagem não permite o uso de ponteiros brutos como em C. Isso reduz o risco de erros de memória, mas ainda permite que o programador acesse e modifique variáveis globais com facilidade. Golang possui um garbage collector, que gerencia automaticamente a memória, o que ajuda a evitar vazamentos de memória. No entanto, o uso de variáveis globais ainda pode levar a problemas de design, como acoplamento excessivo entre funções.

Em **Rust**, o acesso às variáveis é mais restrito devido ao sistema de ownership e borrowing da linguagem. Rust não permite o acesso direto a variáveis globais sem o uso de blocos `unsafe`, o que garante maior segurança de memória. Isso significa que, para acessar ou modificar variáveis globais, o programador precisa explicitamente marcar o código como `unsafe`, o que aumenta a complexidade do código, mas também garante que o acesso seja feito de forma segura e controlada.

Organização de Memória

Em **C**, a memória é gerenciada manualmente. Isso significa que o programador é responsável por alocar e liberar memória conforme necessário. No caso do algoritmo de Gauss, os arrays são alocados estaticamente, o que simplifica o gerenciamento de memória, mas também limita a flexibilidade do código.A falta de

um garbage collector em C exige que o programador gerencie manualmente a memória, o que pode levar a vazamentos se não for feito corretamente..

Em **Golang**, a memória é gerenciada automaticamente pelo garbage collector. Isso significa que o programador não precisa se preocupar com a alocação e liberação de memória, o que reduz a complexidade do código e minimiza o risco de vazamentos de memória. No entanto, o garbage collector pode introduzir uma pequena sobrecarga de desempenho, especialmente em aplicações que exigem alta performance.

Em **Rust**, a memória é gerenciada em tempo de compilação através do sistema de ownership e borrowing. Isso significa que o compilador garante que a memória seja gerenciada de forma segura, sem a necessidade de um garbage collector. No entanto, o sistema de ownership pode ser complexo para programadores iniciantes, e o uso de blocos `unsafe` para acessar variáveis globais pode aumentar a complexidade do código.

Chamadas de Função

Em **C**, as funções são chamadas diretamente, sem restrições. Por exemplo, a função `gauss()` pode ser chamada diretamente no `main`. Essa simplicidade torna o código fácil de entender, mas também pode levar a problemas de segurança, especialmente se as funções não forem bem documentadas ou se houver dependências ocultas entre elas.

Em **Golang**, as funções também são chamadas diretamente, mas todas as funções devem ser associadas a um pacote. Isso ajuda a organizar o código e a evitar conflitos de nomes. Por exemplo, a função `eliminacaoGaussiana(n)` pode ser chamada no `main`, mas ela deve estar dentro de um pacote específico. Essa abordagem promove uma melhor organização do código, mas também pode aumentar a verbosidade.

Em **Rust**, as funções são chamadas diretamente, mas o acesso a variáveis globais exige o uso de blocos `unsafe`. Por exemplo, a função `eliminacao_gaussiana(n)` pode ser chamada dentro de um bloco `unsafe` se precisar acessar variáveis globais. Isso garante que o código seja seguro, mas também aumenta a complexidade, especialmente para programadores que não estão familiarizados com o sistema de ownership de Rust.

Comandos de Controle de Fluxo

Em **C**, os comandos de controle de fluxo são bastante tradicionais. Loops `for` e condicionais `if` são usados de forma clássica.
Essa sintaxe é simples e eficiente, mas pode ser verbosa em comparação com linguagens mais modernas.

Em **Golang**, os comandos de controle de fluxo são semelhantes aos de C, mas com uma sintaxe mais simplificada.
A sintaxe é mais limpa e fácil de ler, o que torna o código mais acessível para programadores iniciantes.

Em **Rust**, os comandos de controle de fluxo são modernos e expressivos.
A sintaxe é mais concisa e moderna, o que torna o código mais legível e menos propenso a erros.

Métricas de Código

Métrica	C	Golang	Rust
Número de Linhas	143	99	91
Número de Funções	7	4	4
Memória	Manual	Garbage Collector	Ownership/Borrowing

Análise Geral

C:
- É a linguagem mais baixo nível, oferecendo controle total sobre a memória e o hardware. No entanto, essa liberdade vem com o custo de maior complexidade e risco de erros, especialmente relacionados ao gerenciamento de memória. O código em C tende a ser mais verboso e propenso a erros, mas é extremamente eficiente em termos de desempenho bruto.

Golang:
- É uma linguagem mais moderna e segura que C, com um garbage collector que gerencia automaticamente a memória. Isso reduz a complexidade do código e

minimiza o risco de vazamentos de memória. O código em Golang é mais conciso e fácil de entender, mas pode ter uma pequena sobrecarga de desempenho devido ao garbage collector.

Rust:

- Oferece um equilíbrio entre desempenho e segurança, com um sistema de ownership que garante segurança de memória em tempo de compilação. No entanto, o uso de blocos `unsafe` para acessar variáveis globais pode aumentar a complexidade do código. Rust é uma linguagem moderna e poderosa, mas com uma curva de aprendizado mais íngreme.

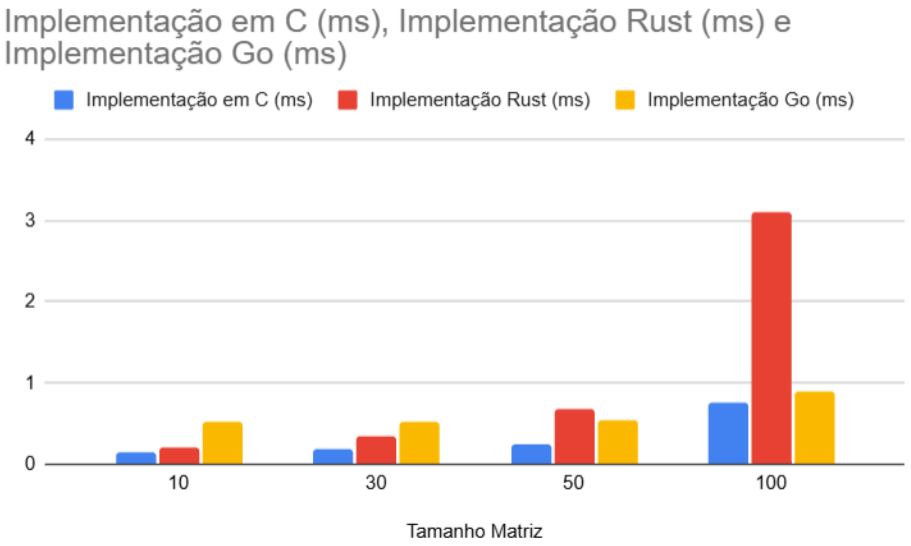
Levantamento de dados

Tabela

Tamanho Matriz	Implementação em C (ms)	Implementação Rust (ms)	Implementação Go (ms)
10	0,1379	0,2095	0,516
30	0,1791	0,342	0,530
50	0,2423	0,681	0,535
100	0,764	3,1032	0,9045
500	64,8050	298,1562	23,004

Esses valores são a média de cinco execuções, todos os tempos estão em milissegundos (ms).

Gráfico



Especificações da máquina utilizada:

CPU: AMD Ryzen 5 5600
Memória: 32GB 3200Mhz DDR4
Armazenamento: SSD 1TB
Placa de Vídeo: Nvidia RTX 4060
Sistema Operacional: Windows 11

Conclusão

Cada linguagem tem suas vantagens e desvantagens, e a escolha da linguagem depende das necessidades do projeto. **C** é a mais rápida em termos de desempenho bruto, mas exige cuidado com o gerenciamento de memória. **Golang** é a mais simples e segura, com desempenho próximo ao de C. **Rust** oferece um equilíbrio entre desempenho e segurança, mas com uma curva de aprendizado mais íngreme. A escolha final dependerá do contexto do projeto e da experiência da equipe de desenvolvimento.