UNIVERSIDADE FEDERAL DE PELOTAS
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

DANIEL LISBOA PEREIRA
JOÃO VITOR LAIMER
daniel.lp@inf.ufpel.edu.br
jvlaimer@inf.ufpel.edu.br

IMPLEMENTAÇÃO DA ELIMINAÇÃO DE GAUSS EM GO E RUST

Pelotas

2025

SUMÁRIO

1 INTRODUÇÃO

A eliminação de Gauss é um dos algoritmos fundamentais para a resolução de sistemas lineares de equações. Sua implementação pode variar significativamente dependendo da linguagem de programação utilizada, devido a diferenças na manipulação de memória, estruturas de controle e chamadas de função. Neste trabalho, analisamos e comparamos três implementações do algoritmo, escritas em C, Go e Rust, destacando suas características sintáticas e de desempenho.

A comparação entre essas linguagens será realizada sob duas perspectivas principais. Primeiramente, analisaremos as diferenças estruturais do código, considerando métricas como número de linhas, quantidade de comandos, uso de ponteiros, gerenciamento de memória e controle de fluxo. Em seguida, realizaremos testes de desempenho para avaliar a eficiência de cada implementação, medindo o tempo de execução do algoritmo para diferentes tamanhos de matriz.

Os experimentos de desempenho serão conduzidos em um ambiente controlado, especificando detalhes da máquina utilizada, como CPU, quantidade de memória RAM e sistema operacional. Os resultados serão apresentados em tabelas e gráficos para facilitar a interpretação das diferenças de tempo de execução entre as linguagens.

Com este estudo, buscamos entender como as escolhas de linguagem impactam a implementação de algoritmos numéricos, considerando não apenas o desempenho, mas também a facilidade de desenvolvimento e manutenção do código.

2 COMPARAÇÃO

2.1 Comparação Desempenho

C é conhecido por seu alto desempenho, oferecendo controle total sobre otimizações e manipulação de memória. Sua ausência de verificações de segurança e coleta de lixo permite que a execução ocorra com máxima eficiência, desde que bem programado.

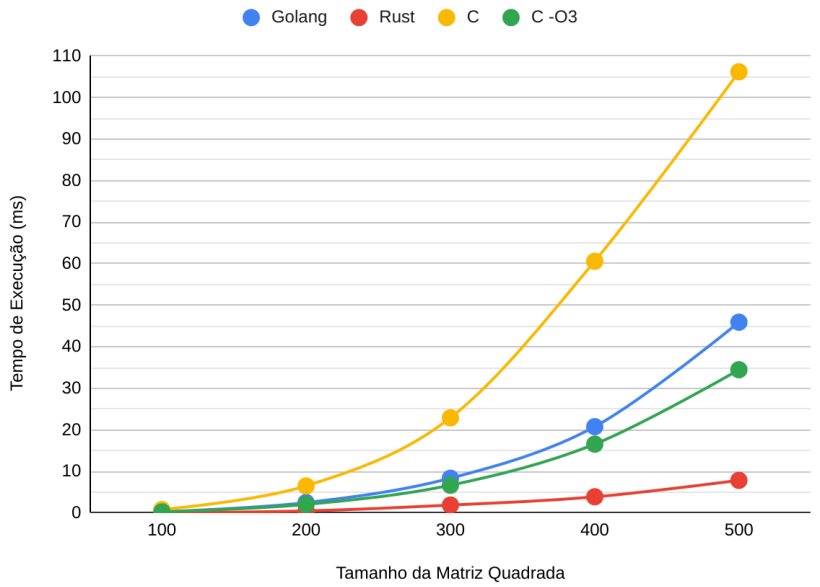
Rust oferece desempenho comparável ao de C, mas com a vantagem de evitar acessos inválidos e problemas de memória por meio de suas verificações em tempo de compilação. Contudo, quando compilado com `--release`, o desempenho melhora significativamente, aplicando otimizações no código.

Golang oferece bom desempenho, mas é projetado para ser eficiente em concorrência e simplicidade, não necessariamente para cálculos numéricos pesados. Seu coletor de lixo (GC) e abstrações mais altas geralmente tornam seu desempenho inferior ao de C e Rust para este tipo específico de processamento intensivo.

Finalmente, quando o código em C é compilado com a flag `-O3`, ele atinge seu desempenho máximo, pois o compilador aplica otimizações agressivas, incluindo *loop unrolling*, *inlining* de funções, eliminação de código morto e melhor utilização de registradores, melhorando consideravelmente o desempenho.

Com isso, realizando os testes da implementação da eliminação de gauss em cada cenário descrito acima, executando o programa compilado de cada linguagem em um processador **Intel I7 10750h**, com **16gb de memória RAM**, no sistema operacional **Debian 12** (bookworm), podemos exemplificar as diferenças de performance de cada linguagem observando a tabela e o gráfico abaixo:

	Tempo de Execução (ms)			
Tamanho Matriz	GOLANG	RUST	C	C -O3
100	0,308	0,063	0,837	0,266
200	2,507	0,511	6,521	2,064
300	8,388	1,914	22,911	6,689
400	20,792	3,892	60,608	16,582
500	45,949	7,875	106,194	34,489



2.2 Comparação Código

2.2.1 Paradigma de Linguagem

C é uma linguagem de programação de baixo nível amplamente utilizada para desenvolvimento de sistemas, proporcionando controle direto sobre a memória e hardware. Rust, também uma linguagem de sistemas, prioriza a segurança de memória sem precisar de um coletor de lixo (GC). Já Go busca simplicidade e concorrência, adotando um modelo de gerenciamento de memória automático por meio de coleta de lixo.

2.2.2 Alocação e Gerenciamento de Memória

Em C, a alocação de memória pode ser feita tanto na stack, com arrays de tamanho fixo, quanto na heap, por meio de `malloc()` e `free()`, exigindo gerenciamento manual da memória. Rust utiliza um modelo baseado em ownership e borrowing, garantindo segurança em tempo de compilação e dispensando `free()`, pois a liberação ocorre automaticamente quando as variáveis saem de escopo. Em Go, a alocação é feita com `make()`, e o coletor de lixo gerencia automaticamente a memória.

Enquanto Rust impede acessos inválidos por meio de verificações rigorosas em tempo de compilação, C permite manipulação direta de ponteiros, o que pode levar a vazamentos ou acessos inválidos se não for gerenciado corretamente. Já Go, apesar de oferecer um gerenciamento automático mais seguro, pode sofrer pequenas perdas de desempenho devido à coleta de lixo em tempo de execução.

2.2.3 Manipulação de Matrizes e Vetores

C trabalha com arrays de tamanho fixo e alocação dinâmica via `malloc()`, exigindo liberação manual com `free()`. A manipulação dos dados ocorre diretamente por meio de ponteiros e aritmética de memória.

Rust permite tanto arrays fixos quanto estruturas alocadas dinamicamente com `Box`, combinando segurança de memória com controle eficiente. Go utiliza slices e arrays, onde

`make()` aloca memória dinamicamente e as verificações de índice são feitas automaticamente.

Enquanto C oferece máxima flexibilidade no controle da memória, essa liberdade exige maior cuidado para evitar acessos inválidos. Rust equilibra flexibilidade e segurança, enquanto Go privilegia simplicidade na manipulação de estruturas de dados.

2.2.4 Geração de Números Aleatórios

Em C, a geração de números aleatórios é feita com `rand()` da biblioteca padrão `<stdlib.h>`, sendo necessário definir uma semente com `srand(time(NULL))` para obter diferentes sequências.

Rust usa o pacote `rand`, com uma API moderna e segura para manipulação de números aleatórios. Go, por sua vez, adota `rand.Seed()` com `time.Now().UnixNano()`. Tanto Rust quanto Go garantem reprodutibilidade ao definir explicitamente a semente, enquanto em C o controle é mais manual.

2.2.5 Medição de Tempo

C mede tempo de execução utilizando `clock()` da biblioteca `<time.h>`, fornecendo o tempo de CPU gasto pelo processo. A precisão pode variar conforme a plataforma.

Rust adota `std::time::Instant` para medições precisas e eficientes. Go usa `time.Now()` e `time.Since()`, permitindo medir a duração com alta precisão.

Enquanto Rust e Go oferecem soluções modernas e diretas para cronometragem, C exige mais configurações e conversões para obter tempos precisos.

2.2.6 Expressividade e Sintaxe

C tem uma sintaxe minimalista, mas exige mais código boilerplate para manipulação de memória e tratamento de erros. Rust possui uma sintaxe mais moderna e rigorosa, garantindo segurança com regras mais complexas de propriedade e tipos. Go, por outro lado, aposta em uma sintaxe simples e direta, tornando o código mais enxuto e fácil de entender.

Embora C seja mais flexível, sua simplicidade pode resultar em maior risco de erros. Rust adiciona segurança com um custo de complexidade sintática. Go sacrifica algumas garantias para oferecer uma abordagem mais pragmática e acessível.

Por fim, Rust tem uma sintaxe concisa e expressiva, oferecendo segurança de memória sem sacrificar o desempenho. Com um sistema de tipos forte e recursos como pattern matching *e* ownership, permite escrever código seguro e eficiente, embora com uma curva de aprendizado mais acentuada.

3 CONCLUSÃO

Cada linguagem apresenta vantagens e desvantagens na implementação do algoritmo de eliminação de Gauss. C é a escolha ideal para máxima performance e controle total, mas exige mais cuidado no gerenciamento de memória. Rust combina eficiência com segurança, oferecendo verificações rigorosas que evitam erros comuns de C sem comprometer muito o desempenho. Go prioriza simplicidade e facilidade de uso, sendo menos eficiente que C e Rust, mas proporcionando um ambiente mais produtivo para desenvolvimento. A escolha entre as linguagens depende do contexto: C é indicado quando performance extrema e controle absoluto sobre a memória forem essenciais, Rust se destaca quando segurança de memória e robustez são prioridades sem comprometer muito o desempenho, e Go é mais adequado quando a simplicidade e a facilidade de manutenção são mais importantes que a otimização de baixo nível.