UNIVERSIDADE FEDERAL DE PELOTAS - UFPEL
CONCEITOS DE LINGUAGENS DE PROGRAMAÇÃO

Implementação da Eliminação de Gauss

Marina de Souza Brum - 19200391
Pelotas, março de 2025.

1. Introdução

O presente trabalho tem como objetivo explorar e comparar as características de três linguagens de programação populares – C, Rust e Golang – ao implementarem um algoritmo clássico de álgebra linear: o algoritmo de Eliminação de Gauss. O foco está em compreender as diferenças e semelhanças entre essas linguagens, levando em consideração aspectos como a estrutura do código, tipos de dados, gestão de memória, controle de fluxo e desempenho.

A implementação de um algoritmo de eliminação de Gauss permite observar claramente as particularidades de cada linguagem em relação à sintaxe, à maneira como lidam com variáveis e funções, e a eficiência de suas operações. O estudo comparativo entre C, Rust e Golang ajudará a entender as vantagens e desvantagens de cada linguagem para esse tipo específico de tarefa computacional.

2. Eliminação de Gauss

O algoritmo de Eliminação de Gauss é um método utilizado para resolver sistemas lineares de equações. O procedimento envolve transformar uma matriz aumentada (representando o sistema de equações) em uma forma triangular superior, utilizando operações elementares sobre as linhas da matriz. Este processo facilita a resolução das equações por substituição regressiva, uma vez que as variáveis já estão em uma ordem que permite a solução sequencial.

O algoritmo é fundamental em várias áreas da matemática e da computação, como na solução de sistemas lineares em problemas de álgebra linear e na análise numérica. Sua implementação eficiente e sua adaptação em diferentes linguagens de programação fornecem insights valiosos sobre o desempenho e as características de cada linguagem.

3. Comparações entre o código de cada linguagem

Linguagem	C	GO	RUST
Função para gerar aleatórios	rand()	rand.Float64()	rng.gen_range(0.0..1.0)
Intervalo de valores	De 0 a 1 após normalização (dividido por 32768.0)	De 0 a 1	De 0 a 1
Tipos de Dados	float	float64	f32
Alocação de	Usando arrays	Usando fatias	Usando Vec, que é

memória	estáticos ou dinâmicos	(slices), que são dinâmicas e flexíveis	uma estrutura dinâmica e redimensionável
Segurança de memória	Não oferece verificação de segurança de memória (risco de estouro de buffer e corrupção)	Não tem verificação explícita de segurança de memória (risco de corrupção, mas coleta de lixo para gerenciamento)	Garantia de segurança de memória através do sistema de posse e referências (sem riscos de estouro de buffer)
Mensuração de tempo	Usando gettimeofday() (para tempo real) e clock() (para tempo de CPU)	Usa time.Now() para tempo atual e time.Since() para medir o tempo decorrido	Usa SystemTime e Instant para medir o tempo de forma precisa e segura (geralmente mais confiável)

3.1. Diferença nas matrizes geradas aleatoriamente:

- C: Utiliza a função rand() que gera números inteiros aleatórios (geralmente no intervalo de 0 a 32767, dependendo da implementação do compilador e da plataforma). A função utiliza números inteiros aleatórios e requer normalização para gerar um valor de ponto flutuante entre 0 e 1. Isso pode ser impreciso e não tão eficiente.
- Go: O Go usa a função rand.Float64() para gerar números aleatórios de ponto flutuante diretamente no intervalo de 0 a 1, com precisão dupla (64 bits), sem a necessidade de normalização.
- Rust: A função rng.gen_range(0.0..1.0) é usada para gerar números aleatórios de ponto flutuante entre 0 e 1. Aqui, rng é um gerador de números aleatórios (como o StdRng), que é configurado previamente. Rust oferece uma abordagem similar à de Go, onde a geração do número aleatório já é feita no intervalo correto, com alta precisão e maior flexibilidade devido ao controle do gerador de números aleatórios.

3.2. Diferença nas sementes aleatórias:

- C: Utiliza srand(time(NULL)) para gerar uma semente baseada no tempo atual (timestamp). Isso depende da precisão do relógio do sistema e pode ser vulnerável à falta de aleatoriedade se o programa for executado rapidamente em sequência.
- Go: Utiliza rand.Seed(time.Now().UnixNano()), que também é baseado no tempo atual, mas usa uma precisão maior (nanossegundos). Isso garante uma semente mais precisa e variável entre execuções, mas ainda depende do relógio do sistema.
- Rust: Utiliza StdRng::seed_from_u64(seed) para configurar uma semente a partir de um valor de 64 bits, o que proporciona um controle preciso sobre a sequência gerada e a reprodutibilidade dos resultados. Isso dá ao programador mais controle, especialmente quando comparado ao que é visto em C e Go

3.3. Tipos de dados

- C: Utiliza o tipo float, que é de precisão simples (32 bits). Isso significa que os números gerados podem não ter a mesma precisão e faixa que os números de precisão dupla, como float64 em outras linguagens. É mais eficiente em termos de memória, mas pode ter limitações em casos que exigem maior precisão.
- Go: Utiliza float64, que é de precisão dupla (64 bits). Isso oferece uma maior precisão e uma faixa mais ampla de valores, permitindo um controle mais preciso sobre os números gerados. No entanto, consome mais memória em comparação com o tipo float ou f32.
- Rust: Utiliza f32, que é de precisão simples (32 bits), similar ao tipo float em C. Isso proporciona uma economia de memória, mas com menor precisão em comparação com o float64 de Go. O tipo f32 é adequado para a maioria dos casos quando a precisão dupla não é necessária.

3.4. Alocação de memória

- C: A alocação de memória pode ser feita com arrays estáticos ou dinâmicos. O gerenciamento de memória em C é manual, o que pode ser propenso a erros como vazamentos ou estouros de buffer.
- Go: Usa fatias (slices), que são estruturas dinâmicas que podem crescer ou encolher conforme necessário. Não há necessidade de gerenciar manualmente a alocação ou desalocação de memória, já que o garbage collector faz isso automaticamente. isso facilita bastante o gerenciamento de memória em Go, embora o custo do gerenciamento automático de memória possa impactar um pouco o desempenho
- Rust: Usa Vec, que é uma estrutura de dados dinâmica e redimensionável que aloca memória na heap. Rust gerencia a memória de forma segura e automática, usando o sistema de posse e empréstimo (ownership and borrowing), o que elimina o risco de vazamentos de memória sem a necessidade de um coletor de lixo.

3.5. Segurança de memória

- C: Exige que o programador tenha muito cuidado ao gerenciar a memória. A falta de verificações automáticas pode levar a falhas de segurança, como estouros de buffer.
- Go: Facilita por meio de garbage collection (GC), que automaticamente gerencia a alocação e desalocação de memória. No entanto, ainda está suscetível a problemas como estouros de buffer ou distorção de dados caso seja feita manipulações incorretas de slices. Mesmo com o GC, o gerenciamento de fatias (slices) em Go exige que o programador tenha cuidado ao acessar os limites e tamanhos das fatias para evitar problemas de segurança.
- Rust: Oferece a maior segurança de memória entre as três linguagens, com um sistema rigoroso de verificação em tempo de compilação, sem a necessidade de um coletor de lixo.

3.6. Número de linhas

- C: Ocupa 122 linhas de código.
- GO: Ocupa 136 linhas de código.
- RUST: Ocupa 116 linhas de código.

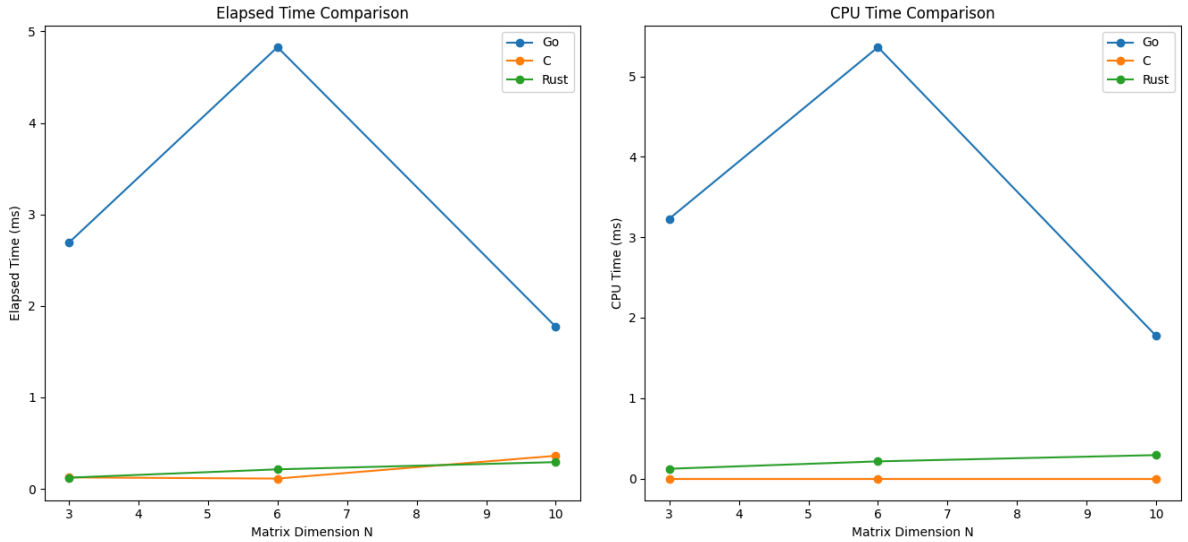
4. Relevância das Diferenças para a Análise de Desempenho

Ao comparar os desempenhos das linguagens C, Go e Rust, buscamos adaptar os códigos para cada linguagem de maneira a respeitar suas características e boas práticas. Embora as implementações tenham diferenças inerentes às particularidades de cada linguagem, essas variações foram feitas com o intuito de garantir uma análise justa e equilibrada, sem que o impacto dessas diferenças prejudique a comparação de desempenho.

- Semente aleatória: Cada linguagem gera a semente de maneira distinta, o que pode resultar em sequências de números aleatórios diferentes. No entanto, essa diferença na geração da semente não afeta substancialmente o desempenho. A sequência aleatória é usada para inicializar o gerador de números, mas não compromete a **eficiência** do código de maneira significativa.
- Medição de Tempo: Cada linguagem utiliza um mecanismo diferente para medir o tempo de execução, o que pode gerar pequenas variações nas medições. Essas diferenças são devidas à precisão e aos métodos internos de cálculo empregados por cada linguagem. Embora as variações existam, elas são mínimas e não impactam de forma substancial a análise de desempenho.
- Alocação de Memória: A forma como a memória é alocada varia entre as linguagens: manual em C, garbage collection em Go e o sistema de posse e empréstimo em Rust. Embora as abordagens de gerenciamento de memória sejam distintas, o impacto no desempenho, especialmente em relação à alocação de memória, tende a ser pequeno na maioria dos cenários típicos. A menos que o código envolva um uso intensivo de memória ou manipulação de grandes volumes de dados, as diferenças na alocação de memória não devem afetar de maneira significativa o desempenho geral.

5. Análise de desempenho - Teste 1

Nesta análise de desempenho, comparamos o tempo de execução (Elapsed Time) e o tempo de CPU (CPU Time) para as linguagens Rust, Go e C, utilizando uma seed aleatória e diferentes valores de n (3, 6 e 10).



Resultados de Tempo de Execução (Elapsed Time)

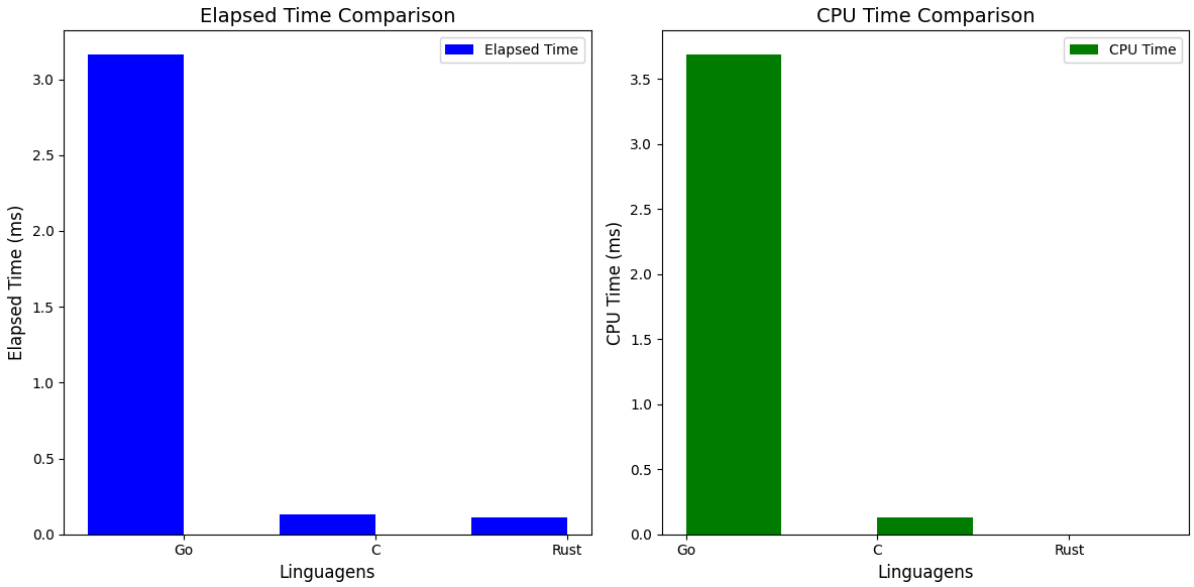
- C: teve tempos de execução significativamente mais altos, com valores variando de 0.113s a 0.362s. A diferença pode ser atribuída ao uso de alocação manual de memória e a natureza de programação mais detalhada de C, que, embora eficiente, exige mais controle e configuração.
- Go: Apresentou tempos de execução variando entre 1.772 ms e 4.828 ms, com um desempenho mais consistente em relação aos diferentes valores de N. Esses tempos são mais elevados, principalmente devido à utilização das slices, que gerenciam dinamicamente a memória e podem adicionar uma sobrecarga de alocação e coleta de lixo. A implementação do garbage collection e o gerenciamento dinâmico de memória são fatores que contribuem para esses resultados.
- Rust: Apresentou tempos de execução superiores aos de Go, com variações entre 0.123s e 0.293s. Rust tem a vantagem de não depender de garbage collection, mas o tempo de execução pode ser impactado pelo processo de inicialização e pela configuração do gerador de números aleatórios.

Resultados de Tempo de CPU (CPU Time)

- C: Teve tempos de CPU constantes de 0.000 ms para todos os valores de N, o que indica que a alocação de memória manual em C não introduziu latência significativa para o gerenciamento de memória no intervalo de N testado.
- Go: apresentou tempos de CPU variando entre 1.7728 ms e 5.3631 ms, exibindo um comportamento não linear com os diferentes valores de N. Isso sugere que o impacto do gerenciamento de memória em Go não segue um padrão linear, podendo ser influenciado por fatores específicos da implementação do garbage collector e pela alocação dinâmica de memória (slices).
- Rust: Apresentou tempos de CPU mais elevados, variando entre 0.122s e 0.292s, o que sugere que o sistema de gerenciamento de memória de Rust, com seu rigoroso modelo de posse e empréstimo, impôs um custo maior no tempo de CPU, especialmente em matrizes maiores.

6. Análise de desempenho - Teste 2

Nesta análise de desempenho, comparamos o tempo de execução (Elapsed Time) e o tempo de CPU (CPU Time) para as linguagens Rust, Go e C, utilizando uma seed com valor definido (12345) e n com apenas um valor (5).



Resultados de Tempo de Execução (Elapsed Time)

- Go: Apresentou o maior tempo de execução, com 3.163 segundos. Isso pode ser atribuído ao overhead do gerenciamento de memória, particularmente devido ao uso de fatias (slices) e à coleta de lixo (garbage collection). O gerenciamento dinâmico de memória, como ocorre com as fatias, pode gerar um overhead adicional, especialmente em cenários que envolvem frequentes alocações e desalocações de memória. Esse comportamento dinâmico pode explicar o maior tempo de execução observado em comparação com as outras linguagens.
- C: Demonstrou um desempenho significativamente mais rápido, com 0.128 segundos. Esse tempo reduzido reflete a execução mais direta e eficiente do código em C, sem a sobrecarga do gerenciamento de memória dinâmico, permitindo que a execução seja mais rápida e com menor impacto de operações internas.
- Rust: Teve um tempo de execução de 0.110 segundos, muito próximo ao desempenho de C. Embora Rust utilize um sistema de gerenciamento de memória dinâmico, ele não conta com garbage collection, o que pode explicar o desempenho semelhante ao de C, já que o gerenciamento de memória é feito de forma mais controlada e sem overhead adicional de coleta de lixo.

Resultados de Tempo de CPU (CPU Time)

- C: Demonstrou um tempo de CPU de 0.128 segundos, o que reflete a execução eficiente do código sem a sobrecarga do garbage collector. O código em C, com gerenciamento manual de memória, tende a ser mais direto e eficiente em termos de utilização da CPU, especialmente em comparação com Go.
- GO: Apresentou um tempo de CPU de 3.688 segundos, significativamente maior do que C e Rust. Esse aumento no tempo de CPU pode ser atribuído ao overhead do gerenciamento de memória dinâmico e ao funcionamento do garbage collector, que pode exigir mais processamento, especialmente em operações frequentes de alocação e desalocação de memória.
- **Rust:** Apresentou 0.000 segundos de tempo de CPU, indicando que o código em Rust teve um impacto mínimo na CPU para o valor de n = 5. Isso sugere que o gerenciamento de memória em Rust, baseado no sistema de posse e empréstimo (ownership and borrowing), não gerou sobrecarga significativa na CPU.

7. Conclusão

Após a realização dos testes de desempenho com a eliminação de Gauss nas linguagens Go, C e Rust, foi possível observar diferenças notáveis em relação aos tempos de execução e uso de CPU. Nos testes iniciais, realizados sem a utilização de slices em Go, o desempenho dessa linguagem foi bastante competitivo, com tempos de execução e uso de CPU próximos aos de C e Rust, que são conhecidos por sua eficiência. No entanto, quando slices foram introduzidos em Go, que envolvem alocação dinâmica de memória, o desempenho caiu consideravelmente, afastando-se dos resultados observados nas outras duas linguagens.

C demonstrou o melhor desempenho geral, com os menores tempos de execução e uso de CPU, devido ao seu controle direto sobre a memória. Já Rust apresentou um desempenho semelhante ao de C, beneficiando-se de seu sistema de alocação de memória sem coleta de lixo, que oferece segurança e controle mais rigoroso, sem comprometer tanto a performance. Em contraste, Go, ao usar slices, apresentou maiores tempos de execução e maior uso de CPU, evidenciando a sobrecarga da alocação dinâmica de memória. Com base nos resultados obtidos, C e Rust se mostraram mais adequados para cenários de alto desempenho, sendo Rust uma opção interessante por sua segurança e controle de memória sem a necessidade de coleta de lixo.