Universidade Federal de Pelotas

Ciência da Computação

Relatório de Análise e Implementação da Eliminação de Gauss em C, GoLang e Rust

Prof. Gerson Geraldo Cavalheiro

Gabriel Moura

Luis Eduardo Rasch

Renan Pinho

Repositório: https://github.com/pinhorenan/eliminacao_gaussiana

1. Implementações

Ao traduzir o algoritmo de eliminação Gaussiana para diversas linguagens de programação, surgem diferenças notáveis relacionadas às características fundamentais de cada uma.

Em C, a abordagem reflete sua essência de baixo nível e domínio total sobre os recursos computacionais. A alocação de memória é feita diretamente, com malloc e free, oferecendo adaptabilidade, mas demandando atenção para evitar problemas como perdas de memória. A falta de um coletor de lixo (GC) permite otimizações avançadas, mas também abre caminho para erros graves, como acesso a endereços de memória inválidos. A medição do tempo, por exemplo, depende diretamente de APIs do Windows (como QueryPerformanceCounter), mostrando a ligação com o sistema operacional.

Em GoLang, a implementação se destaca pela simplicidade e rapidez no desenvolvimento. A linguagem simplifica o gerenciamento de memória com slices dinâmicos e coleta de lixo, eliminando a necessidade de alocação manual. Isso resulta em um código mais fácil de entender e com menos chances de erros, como perdas de memória, mas introduz uma possível demora devido ao GC. A biblioteca padrão completa — com pacotes como time para medir o desempenho e math/rand para gerar números aleatórios — facilita tarefas comuns sem precisar de bibliotecas externas.

Já em Rust, a implementação equilibra segurança e desempenho de forma singular. O sistema de ownership e borrow checker garante que as operações de memória sejam seguras durante a compilação, eliminando erros comuns como ponteiros soltos sem comprometer o desempenho. A alocação dinâmica é feita com Vec, uma estrutura segura e flexível, enquanto as interações com o sistema (como a medição do tempo via GetProcessTimes) exigem blocos unsafe, definindo claramente onde a segurança pode ser afetada. A dependência de crates como rand e winapi mostra seu ecossistema modular, onde as funcionalidades são adicionadas conforme a necessidade.

2. Tabela Comparativa

	Melhor Tempo	Pior Tempo	Número de Linhas	Número de Comandos
C	104865 ms	108179 ms	151	128
Go	59927 ms	60820 ms	105	70
Rust	40635 ms	47699 ms	114	114

3. Conclusão

A análise dos dados da tabela comparativa revela diferenças significativas entre as linguagens C, Go e Rusty em termos de desempenho, concisão e complexidade de implementação. Rust destaca-se como a linguagem mais rápida, com o melhor tempo de execução (40.635 ms) e o pior tempo (47.699 ms), superando Go e C por uma margem expressiva. Essa performance superior é resultado da combinação de segurança de memória e otimizações de baixo nível, características intrínsecas do Rust. Go, por sua vez, ocupa a

segunda posição em desempenho, com tempos próximos entre o melhor e o pior caso (59.927 ms a 60.820 ms), demonstrando consistência e eficiência, especialmente considerando sua simplicidade e foco em produtividade. Já C apresenta os piores tempos (104.865 ms a 108.179 ms), o que pode ser atribuído à ausência de otimizações automáticas e à sobrecarga do gerenciamento manual de memória, que exige maior atenção do desenvolvedor para evitar erros.

Em termos de concisão, Go é a linguagem mais enxuta, com apenas 105 linhas e 70 comandos, refletindo sua sintaxe simplificada e abstrações eficientes, como o uso de slices e a coleta de lixo automática. Rust, por outro lado, requer 114 linhas e 114 comandos, equilibrando segurança (através do sistema de ownership) e expressividade, mas com uma verbosidade um pouco maior devido às verificações em tempo de compilação. C é a mais verbosa, com 151 linhas e 128 comandos, exigindo código explícito para alocação de memória, loops e funções de sistema para medição de tempo, o que aumenta a complexidade e o risco de erros.

Quando se trata de complexidade versus controle, C prioriza o controle absoluto sobre o hardware, mas paga o preço em complexidade e risco de erros, como possíveis vazamentos de memória. Go, por outro lado, oferece produtividade e legibilidade, sendo ideal para prototipagem rápida e desenvolvimento ágil, embora com desempenho inferior ao Rust, que por sua vez combina alta performance com segurança de memória, sendo ideal para sistemas críticos, ainda que exija mais atenção a detalhes de implementação.