UNIVERSIDADE FEDERAL DE PELOTAS
Centro de Desenvolvimento Tecnológico
Ciência da Computação
Conceitos de Linguagem de Programação

Implementação da Eliminação de Gauss
C, Rust e Golang

Bianca Waskow
Rafael Siqueira

Pelotas, 2023.

Introdução

O desenvolvimento de software tem evoluído constantemente, com o surgimento de novas linguagens de programação que buscam solucionar problemas de performance, segurança e escalabilidade. Entre as linguagens de programação mais recentes, destacam-se Rust e Golang. Ambas foram desenvolvidas para lidar com desafios da programação moderna, como a concorrência e a segurança de memória. Nesse sentido, é importante compreender as características dessas linguagens e compará-las com uma linguagem mais antiga como a C, que é uma das linguagens de programação mais consolidadas.

A eliminação de Gauss é um algoritmo bem conhecido para solução de sistemas de equações lineares. Neste trabalho, comparamos as linguagens de programação C, Rust e Golang, implementando o algoritmo de Eliminação de Gauss em cada uma delas. Esse é um algoritmo relativamente simples e bem definido, o que nos permite comparar de forma justa as linguagens em termos de clareza e legibilidade do código.

Objetivo

O objetivo deste estudo é compreender as características de Rust e Golang comparando uma implementação da eliminação de Gauss nessas linguagens com a implementação em linguagem C, analisando as diferenças entre as três linguagens em termos de eficiência, desempenho e facilidade de implementação do algoritmo.

Para isso, serão realizados testes de performance, análise dos tipos de dados, acessos às variáveis, organização de memória, chamadas de função e comandos de controle de fluxo. Ainda, serão analisados os aspectos de sintaxe e semântica dessas linguagens para entender como elas diferem da linguagem C comparando seu número de linhas, comandos e número médio de lexemas por linha.

Ao final deste trabalho, esperamos ter uma compreensão clara das vantagens e desvantagens de cada linguagem de programação para a implementação de algoritmos numéricos, bem como suas limitações e possíveis aplicações em outras áreas da computação.

Eliminação de Gauss

A Eliminação de Gauss é um método utilizado para transformar um sistema linear em um sistema triangular superior equivalente, que pode ser resolvido diretamente por meio da resolução retroativa. O objetivo é simplificar o sistema original por meio de sucessivas operações elementares, mantendo as mesmas soluções.

Embora existam várias maneiras de escalonar uma matriz, o algoritmo da Eliminação de Gauss é o mais eficiente computacionalmente [5]. Ao utilizar a Eliminação de Gauss para resolver um sistema linear, pode-se obter uma solução precisa e eficiente, o que é crucial em muitas áreas da ciência e engenharia.

Para aplicar a eliminação de Gauss, é preciso representar o sistema de equações como uma matriz ampliada. Essa matriz é formada pelos coeficientes das variáveis do sistema, seguidas pelos valores constantes das equações. A matriz ampliada é chamada de Ab na imagem abaixo e é usada como entrada nos algoritmos que implementamos.

Seja um sistema linear:

$$\begin{cases} 3x_1 + 2x_2 + 4x_3 = 1 \\ x_1 + x_2 + 2x_3 = 2 \\ 4x_1 + 3x_2 - 2x_3 = 3 \end{cases}$$

Podemos escrevê-lo na forma matricial do tipo $Ax = b$, onde:

$$A = \begin{bmatrix} 3 & 2 & 4 \\ 1 & 1 & 2 \\ 4 & 3 & -2 \end{bmatrix} \quad b = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}.$$

E a partir delas escrevemos a matriz aumentada do sistema:

$$Ab = \left[\begin{array}{ccc|c} 3 & 2 & 4 & 1 \\ 1 & 1 & 2 & 2 \\ 4 & 3 & -2 & 3 \end{array}\right].$$

O primeiro passo da eliminação de Gauss é usar operações elementares de linha para transformar a matriz ampliada em uma matriz escalonada. Para fazer isso, é necessário escolher uma linha como a linha pivô e eliminar a variável correspondente nas outras linhas. A escolha do pivô pode ser feita de diversas maneiras, sendo a mais comum a escolha do elemento de maior valor absoluto na coluna correspondente.

O segundo passo é resolver o sistema escalonado. A partir da matriz escalonada, pode-se obter a solução das variáveis do sistema por meio de substituições regressivas. Começa-se pela última equação do sistema e, a partir dela, substitui-se o valor da variável desconhecida encontrada em equações posteriores.

$$\left[\begin{array}{ccc|c} 3 & 2 & 4 & 1 \\ 0 & \frac{1}{3} & \frac{2}{3} & \frac{5}{3} \\ 0 & 0 & -8 & 0 \end{array}\right]$$

Matriz escalonada

$$\begin{cases} 3x_1 + 2x_2 + 4x_3 = 1 \\ \frac{1}{3}x_2 + \frac{2}{3}x_3 = \frac{5}{3} \\ -8x_3 = 0 \end{cases}$$

Sistema a ser resolvido facilmente por substituição regressiva

Linguagens, códigos e suas comparações

A linguagem de programação **C** foi inicialmente criada para ser uma linguagem de programação de sistema, para a implementação do sistema operacional UNIX. Ela oferece recursos de baixo nível que permitem que os programadores escrevam código de sistema de *alto desempenho*. **Rust** também é uma linguagem de programação de sistema, projetada para ser *segura, concorrente e de alto desempenho*, foi lançada em 2010 e vem ganhando popularidade desde então. **Golang**, também conhecida como Go, é uma linguagem de programação de código aberto criada em 2007 na Google e anunciada publicamente em 2009. Foi projetada para oferecer um equilíbrio entre *eficiência de programação e desempenho de execução*.

C, Rust e Go são linguagens de tipagem estática, o que significa que elas precisam conhecer os tipos de todas as variáveis em tempo de compilação. Mas, Rust e Go possuem o recurso de inferência de tipos, o que permite que os tipos das variáveis e constantes sejam definidos de maneira automática pelo compilador com base no valor que atribuímos à variável e como a usamos.

Por padrão Rust utiliza variáveis *imutáveis*, isso quer dizer que após a inicializarmos não é possível modificar o seu valor. Essa é uma forma de manter seu código seguro e estabilizar a concorrência. Porém temos a opção de tornar uma variável mutável e para isso adicionamos a palavra chave `mut`.

Mesmo sabendo que o tipo pode ser inferido automaticamente, temos a opção de explicitar o seu tipo da forma:

```
let x: i32 = 10;
```

Na linguagem Go essa linha ficaria assim:

```
var x int = 10
```

A tabela a seguir mostra os tipos inteiros em Rust e Go:

Rust			Go		
Tamanho	Signed	Unsigned	Tamanho	Signed	Unsigned
8-bit	`i8`	`u8`	8-bit	`int8`	`uint8`
16-bit	`i16`	`u16`	16-bit	`int16`	`uint16`
32-bit	`i32`	`u32`	32-bit	`int32`	`uint32`
64-bit	`i64`	`u64`	64-bit	`int64`	`uint64`
arch	`isize`	`usize`			

Os tipos `isize` e `usize` do Rust dependem do computador em que o programa está rodando: 64 bits se estiver em uma arquitetura de 64-bit e 32 bits se sua arquitetura for 32-bit. A principal situação para usar `isize` ou `usize` é indexar algum tipo de coleção.

Rust e Go também possuem dois tipos primitivos para números de ponto flutuante. Os pontos flutuantes do Rust são `f32` e `f64` e do Go são `float32` e `float64`, que têm respectivamente os tamanhos de 32 e 64 bits. Os tipos *default* no Rust são `f64` e `i32`, já no Go são `float32` e `int32`.

Strings

Em C as strings são implementadas como vetores de caracteres (char), onde cada elemento do vetor corresponde a um caractere da string e o último elemento é sempre um caractere nulo ‘\0’ que indica o término da string. Em contraste com Rust e Golang.

Rust tem dois tipos principais de strings: `&str` e `String`. `&str` são chamados de "fatias de string". Uma fatia de string tem tamanho fixo. Não se pode acessar um `str` diretamente, mas apenas por meio de uma referência `&str`. Isso porque `str` é um tipo não dimensionado que requer informações de tempo de execução adicionais para ser utilizável.

`String`s são implementadas como uma coleção de bytes. Os dados que as coleções apontam estão guardados na *heap*, que significa que a quantidade de dados não precisa ser conhecida em tempo de compilação e pode aumentar ou diminuir conforme a execução do programa.

Já em Go uma `string` é na verdade um *slice* de bytes somente leitura. Se utilizarmos o `Println` ele mostra seu valor decimal, isso porque Go utiliza um `int32` para representar todos os caracteres do UTF8.

Vetores e matrizes

Em Rust há a coleção `Vec<T>`, que possibilita guardar um número variável de valores um ao lado do outro e assim como a String os dados que essa coleção aponta está guardado na *heap*, ou seja, a quantidade de dados não precisa ser conhecida em tempo de compilação e pode aumentar ou diminuir conforme a execução do programa. Além dessa coleção há a declaração de vetor como um *array* de tamanho fixo, assim como em C e em Go.

```
int X[5]; // C

let mut x: [i32; 5]; // Rust

let x: Vec<i32> = Vec::new(); // Rust

var X [5]int32 // Go
```

Matrizes em C, Rust e Golang são de tamanhos fixos: uma vez declarado, eles não podem aumentar ou diminuir de tamanho.

```
int Ab[5][5]; // C

let mut ab: [[i32; 5]; 5] = [[0.0; 5]; 5]; // Rust

var Ab [5][5]int64 // Go
```

No exemplo em Rust `let mut A: [[f32; 20]; 20] = [[0.0; 20]; 20];`, estamos criando uma matriz A de 20 linhas e 20 colunas, onde cada elemento é um número inteiro de 32 bits (`i32`). A primeira parte da declaração, `[[i32; 20]; 20]`, especifica o tipo de cada elemento na matriz e o tamanho da matriz. A segunda parte, `[[0.0; 20]; 20]`, inicializa todos os elementos da matriz com o valor 0.0. Essa sintaxe é usada para simplificar a inicialização de matrizes com um valor padrão.

Rust: *Ownership*

A característica mais singular do Rust é a propriedade ownership, que permite que a linguagem ofereça segurança de memória sem a necessidade de um coletor de lixo.

Todos os programas precisam definir como irão gerenciar a memória do computador durante a execução. Algumas linguagens, como C deixam isso totalmente por conta do programador, que deve alocar e liberar memória de forma explícita. Outras como Golang, possuem garbage collection (coleta de lixo), que constantemente busca segmentos de memória que já não são mais utilizados enquanto o programa é executado.

Em Rust, a gestão de memória é realizada por meio de um sistema de posse, que conta com um conjunto de regras validadas no momento da compilação. Não há nenhum impacto no tempo de execução em relação às funcionalidades relacionadas à posse.

Todo objeto na *heap* é de propriedade de uma, e somente uma, variável. Quando esta variável sai de contexto, Rust automaticamente remove o objeto da memória.

É possível devolver ao sistema operacional a memória utilizada por uma `String` em Rust, de forma natural, quando a String sai do escopo.. Quando uma variável sai de escopo, o Rust chama para nós uma função especial. Essa função é chamada `drop`, e é aí que o autor da `String` pode colocar o código que retorna a memória. Rust chama `drop` automaticamente ao fechar chaves (`}`).

Ponteiros e referência

O tipo mais comum de ponteiro em Rust é a referência, que é um ponteiro verificado pelo compilador. Referências são indicadas pelo símbolo &, e “pegam emprestado” o valor para o qual apontam, são um tipo de ponteiro que têm como única habilidade referir-se a dados. E ao contrário de outros tipos de ponteiros, as referências não têm custos adicionais e são frequentemente utilizadas.

Elas não têm nenhuma outra habilidade senão referir-se a dados. Além disso, elas não têm nenhum custo adicional e são o tipo de ponteiro usado com maior frequência.

A grande diferença em comparação com C é que o Rust verifica SEMPRE se a referência é válida e se não está apontando para lixo na memória em tempo de compilação, assim os erros sempre vão aparecer na compilação e não na execução.

Já em Go, os ponteiros são inicializados com um "valor zero", que é nulo para tipos de ponteiros. Portanto, desreferenciar um ponteiro nulo resultará em um *runtime panic*. Além disso, o Go possui certos recursos para ajudar a evitar o uso de ponteiros não inicializados ou pendentes. Por exemplo, o *garbage collector* do Go pode ajudar a detectar e liberar memória que não está mais sendo usada. No entanto, ainda é possível criar ponteiros pendentes ou acessar memória inválida em Go, assim como em C.

Em Rust, assim como as variáveis são *imutáveis* por padrão, as referências também são. Não temos permissão para modificar algo que temos apenas a referência e podem ter várias referências para um mesmo objeto.

Rust: referências mutáveis

As referências mutáveis têm uma limitação importante: é permitido ter somente uma referência mutável para uma parte específica dos dados em um escopo específico. Essa limitação é necessária para permitir a mutabilidade de forma segura e controlada. É algo diferente, porque a maioria das linguagens, como C e Golang, permitem que se faça mudanças sempre que desejar.

Com essa restrição, Rust consegue evitar conflitos de dados já durante a compilação. Tais conflitos podem levar a comportamentos indefinidos e, muitas vezes, são difíceis de serem diagnosticados e corrigidos em tempo de execução. Ao não permitir esses problemas desde a compilação, Rust aumenta a segurança do código e reduz a possibilidade de erros difíceis de serem resolvidos.

Nós usamos referências mutáveis para “passar” os arrays para as funções.

```
fn gauss(n: usize, ab: &mut[[f64; MAXN+1]; MAXN], x: &mut [f64; MAXN])
{

      // código

}
```

Argumentos de linha de comando

Para habilitar a leitura dos valores dos argumentos de linha de comando em C usa-se `int argc, char** argv`.

Em Rust precisa-se de `std::env::args` função fornecida na biblioteca padrão do Rust. Esta função retorna um iterador dos argumentos de linha de comando passados. O iterador produz uma série de valores e podemos chamar o método `collect` em um iterador para transformá-lo em uma coleção, como um vetor, que contém todos os elementos que o iterador produz.

```
use std::env;

let args: Vec<String> = env::args().collect();
```

Em Golang, há um pacote chamado “os” que contém um *array* denominado “Args”. `Args` é um *array* de string que contém todos os argumentos de linha de comando passados.

```
package main

import (

    "fmt"

    "os"

    "strconv"
```

```
)

func main() {

    if len(os.Args) != 3 {

        fmt.Println("Incorrect number of arguments")

        os.Exit(1)

    }

    N, _ = strconv.Atoi(os.Args[2])

    readMatrix(N, os.Args[1])

    // . . .

}
```

O package main define qual é o package principal do programa. O nome do package é sempre declarado na primeira linha de cada arquivo Go.

O import fmt prover a formatação dos valores de entrada e saída do Go. As suas funções se assemelham muito ao printf e scanf da linguagem C.

Lendo as matrizes de arquivos texto

Para ler a matriz de entrada de um arquivo texto em Rust nós usamos os módulos `std::fs::File` e `std::io::{BufRead, BufReader}`

```
fn read_matrix(n: usize, arg: &str, ab: &mut[[f64; MAXN+1]; MAXN]){

      let file = File::open(arg).unwrap();

      let reader = BufReader::new(file);

      for (i, line) in reader.lines().enumerate() { // read matrix
elements

          let line = line.unwrap();

          let mut nums = line.split_whitespace().map(|num|
num.parse().unwrap());

          for j in 0..n+1 {

              ab[i][j] = nums.next().unwrap();

          }

      }

}
```

A primeira linha da função abre o arquivo em “arg”. A função `unwrap` é usada para lidar com possíveis erros na abertura do arquivo. Se ocorrer algum erro, o programa irá encerrar com uma mensagem de erro.

Em seguida, a função `BufReader::new` é usada para criar um objeto BufReader, que permite a leitura eficiente de um arquivo. Dentro do loop for, O objeto `reader` é usado para ler o arquivo linha por linha.

Dentro do loop for, a função `enumerate` é usada para obter o índice i e a linha `line` atual. A linha lida é dividida em números usando a função `split_whitespace`. Em seguida, a função `map` é usada para converter cada número em `f64`. Finalmente, dentro do segundo loop for, cada número é atribuído ao elemento apropriado da matriz ab.

Lendo a matriz de entrada em Go:

```
func readMatrix(N int, arg string) {
    file, err := os.Open(arg)
    if err != nil {
        panic(err)
    }
    defer file.Close()
    scanner := bufio.NewScanner(file)
    for i := 0; i < N; i++ {
        line := ""
        if scanner.Scan() {
            line = scanner.Text()
        } else {
            fmt.Println("Error reading file")
            os.Exit(1)
        }
        tokens := strings.Fields(line)
        if len(tokens) != N+1 {
            fmt.Println("Invalid line in file:", line)
            os.Exit(1)
        }
        for j := 0; j <= N; j++ { // read matrix elements
            value, err := strconv.ParseFloat(tokens[j], 64)
            if err != nil {
                fmt.Println("Invalid value in file:", tokens[j])
                os.Exit(1)
            }
```

```
            Ab[i][j] = float64(value)
        }
    }
}
```

A função abre o arquivo usando a função os.Open e cria um scanner de texto para ler as linhas do arquivo com `bufio.NewScanner`. O arquivo é fechado no final da função usando defer `file.Close()`. A função itera pelas linhas do arquivo usando um loop for. A cada iteração, a função lê uma linha do arquivo usando `scanner.Scan()` e armazena o resultado na variável `line`.

Em seguida, a função verifica se a linha contém exatamente N+1 elementos usando a função `strings.Fields`, que separa a linha em uma lista de strings contendo os elementos da linha. Se a linha não contém N+1 elementos, a função exibe uma mensagem de erro e sai do programa usando `os.Exit(1)`.

Por fim, a função itera pelos elementos da linha com outro loop for. A cada iteração, a função converte o valor da string para um `float64` usando `strconv.ParseFloat`, e armazena o resultado na matriz `Ab[i][j]`. Se a conversão não for bem sucedida, a função exibe uma mensagem de erro e sai do programa.

Funções

Em Rust as funções são declaradas usando a palavra reservada `fn`. Deve-se especificar o tipo de dado de cada parâmetro. Se a função retornar um valor, o tipo de retorno deve ser especificado após uma seta ->. A expressão final na função será usada como valor de retorno, a não ser que a instrução `return` seja usada para retornar um valor anterior de dentro da função.

```
fn cinco() -> i32 { // Função que retorna o número 5
    5
}
```

Em Go uma função é definida usando a palavra reservada `func`. Assim como em C e (talvez) Rust, para retornar um valor usa-se `return`. Também deve-se especificar o tipo de dado de cada parâmetro e o tipo de dado retornado (caso houver). Se uma função especifica um retorno, ela deve fornecer um retorno, se não, haverá um erro de compilação.

```
func double(x int) int {
    y := x * 2
    return y
```

```
}
```

Comandos de controle de fluxo: for

Em Rust o `for in` pode ser usado para iterar por meio de um Iterator. Uma das maneiras mais fáceis de criar um iterador é usar a notação de intervalo `a..b`. Isso produz valores de `a` (inclusivo) a `b` (exclusivo) pulando de um em um. Como alternativa, `a..=b` pode ser usado para um intervalo inclusivo em ambas as extremidades. Já em Golang o `for` não foge muito ao estilo de C.

```
// Rust:
// . . .
for i in (0..n-1).rev() { // this loop is for backward substitution
    let mut sum = 0.0;
    for j in i+1..n {
        sum = sum + ab[i][j] * x[j];
    }
    x[i] = (ab[i][n] - sum) / ab[i][i];
}
// Golang:
// . . .
for i:=N-2; i>=0; i-- { // this loop is for backward substitution
    sum = 0
    for j := i + 1; j < N; j++ {
        sum = sum + Ab[i][j]*X[j]
    }
    X[i] = (Ab[i][N] - sum) / Ab[i][i]
}
```

Os loops `for` são amplamente utilizados em Rust devido à sua segurança e concisão, tornando-os o tipo de loop mais frequente na linguagem.

Assim seria a contagem regressiva usando um loop `for` e o método `rev`, para reverter o intervalo:

```
fn main() {
    for numero in (1..4).rev() {
        println!("{}!", numero);
```

```
    }
}
```

Comparações gerais: Rust e Golang

Rust e Golang são linguagens de programação modernas, projetadas para alto desempenho e segurança de memória. Rust é considerada mais desafiadora para os desenvolvedores devido ao seu ambiente de código mais seguro e gerenciamento de memória estático, enquanto Golang é mais fácil de trabalhar graças ao seu eficiente coletor de lixo e abordagem menos rígida em relação ao controle de memória.

Na verificação de erros, Rust oferece uma opção de verificação por meio do compilador, enquanto Golang dá ao desenvolvedor a escolha de realizar a verificação ou não. Ambas as linguagens lidam com o problema de segurança de memória de maneiras diferentes, mas oferecem soluções inteligentes e seguras para o gerenciamento de memória.

Ambas as linguagens são compiladas, o que significa que os programas são traduzidos diretamente para código de máquina executável, permitindo a implantação como um único arquivo binário. Isso simplifica a distribuição, sem a necessidade de distribuir interpretadores ou muitas bibliotecas e dependências, tornando a distribuição mais vantajosa. Isso também resulta em um melhor desempenho em comparação com as linguagens interpretadas, como Python e Ruby.

C é uma linguagem de programação de baixo nível, útil para escrever drivers de dispositivo e outras aplicações de baixo nível. No entanto, sua flexibilidade pode levar a erros de segurança, como estouros de buffer, e ela não possui recursos avançados de segurança de memória.

Já Rust foi projetada para ser segura, concorrente e de alto desempenho, usando o sistema de gerenciamento de memória "*ownership*" para evitar problemas de segurança de memória. Golang, por outro lado, usa *garbage collector* automatizado e foi criada para equilibrar eficiência de programação e desempenho de execução, com ênfase em simultaneidade e paralelismo. Ambas são projetadas para alto desempenho, mas Rust tem uma vantagem devido ao seu refinado controle sobre a alocação e desalocação de memória.

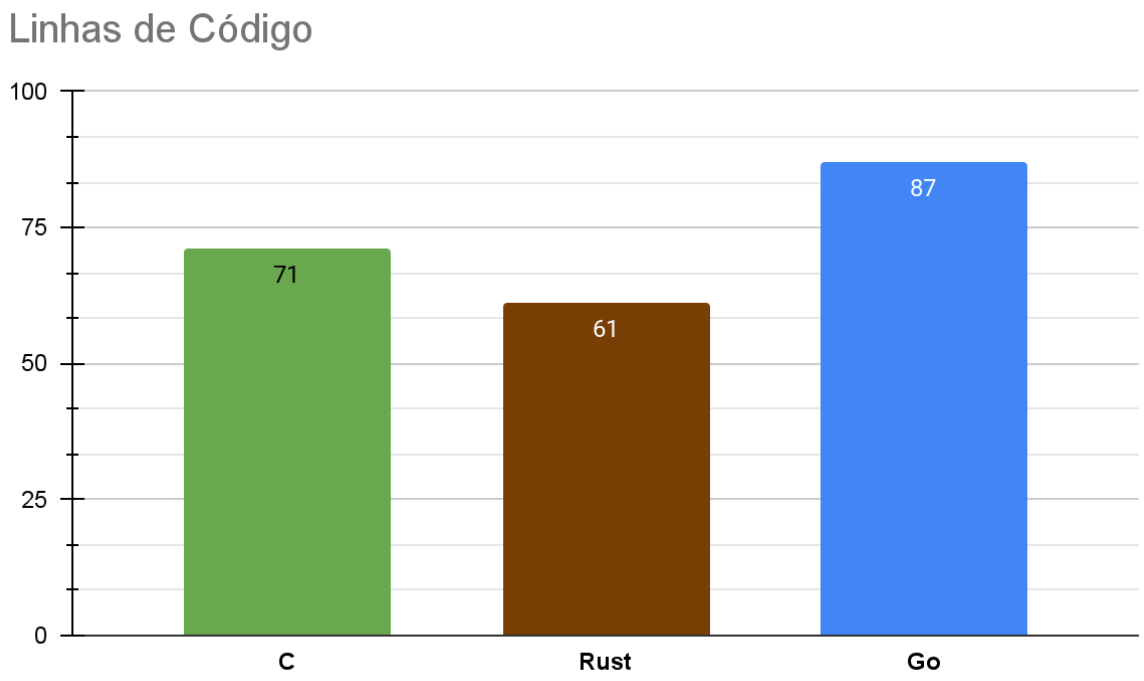
Em resumo, enquanto C oferece flexibilidade e recursos de baixo nível, sua falta de recursos avançados de segurança de memória a torna menos segura que Rust. Rust é projetada para segurança de memória, concorrência e eficiência, enquanto Golang busca um equilíbrio entre eficiência de programação e desempenho de execução, com ênfase em simultaneidade e paralelismo.

Quadros Comparativos

Quadro com a quantidade de linhas de código escritas para implementação do algoritmo de eliminação de Gauss:

Número de Linhas de Código		
C	**Rust**	**Go**
71	61	87

Representação gráfica do quadro acima:

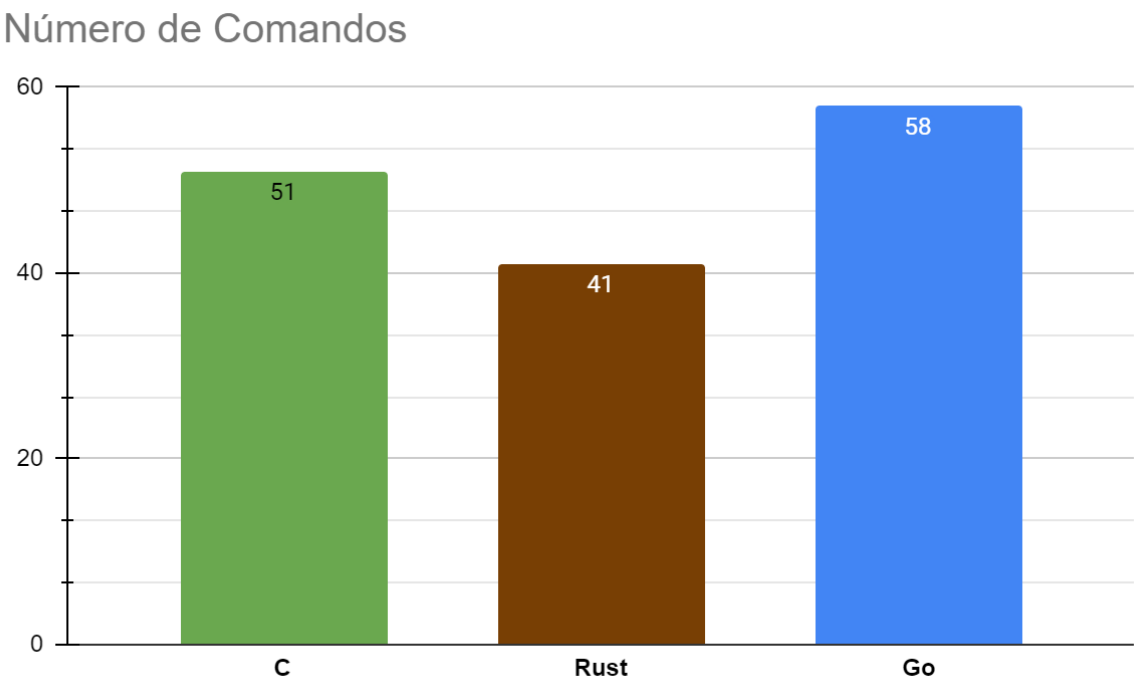


Utilizando a mesma política de indentação em todas as implementações, Rust foi a que menos consumiu linhas com um total de 61, enquanto que Go foi a que mais consumiu linhas com um total de 87. C ficou próximo da média entre as duas outras linguagens, com um total de 71 linhas.

Quadro com a quantidade de comandos utilizados para implementação do algoritmo de eliminação de Gauss:

Número de Comandos		
C	**Rust**	**Go**
51	41	58

Representação gráfica do quadro acima:

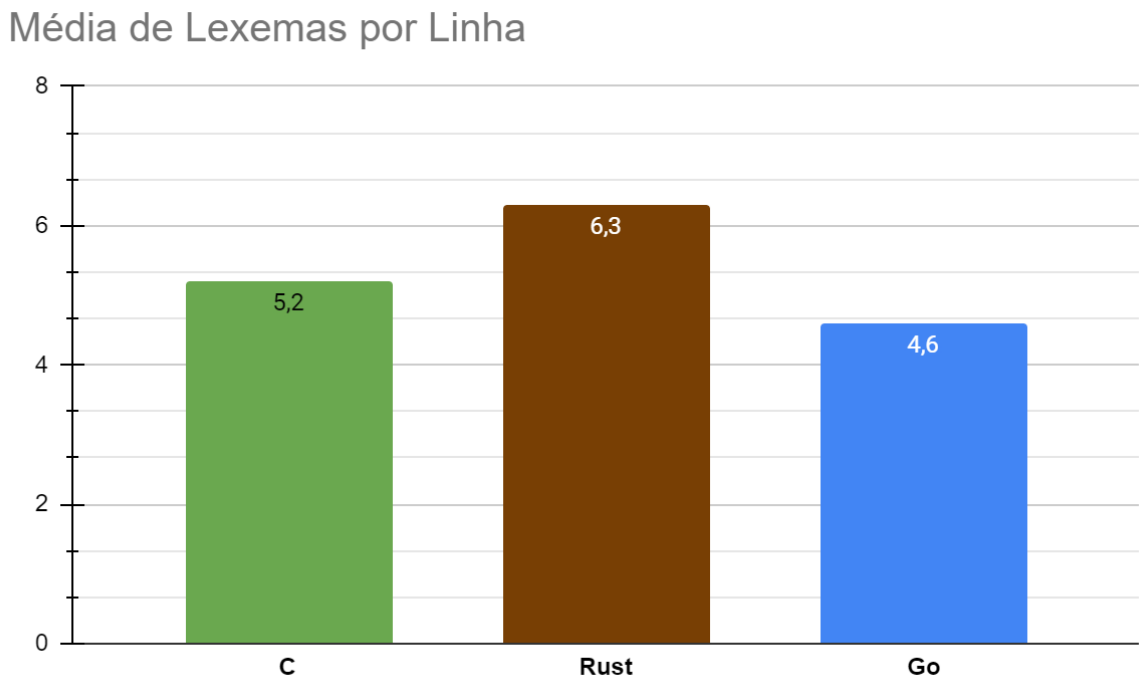


Utilizando os comandos oferecidos nas bibliotecas padrão de cada linguagem, Rust foi a que menos necessitou comandos, ficando com um total de 41. Go apresentou o maior número de comandos para o mesmo algoritmo, totalizando 58. C manteve-se próximo do valor médio entre os resultados das outras linguagens, com um total de 51 comandos.

Quadro com a quantidade média de lexemas utilizados por linha de código na implementação do algoritmo de eliminação de Gauss:

Número Médio de Lexemas por Linha		
C	**Rust**	**Go**
5,2	6,3	4,6

Representação gráfica do quadro acima:



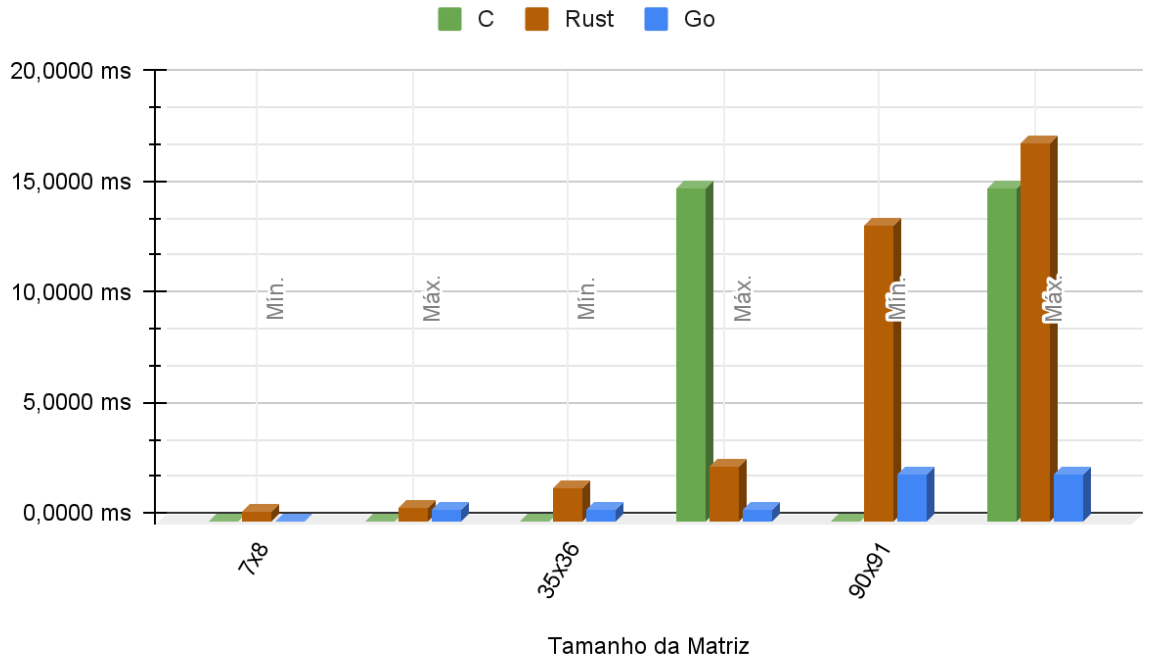
Na avaliação da média de lexemas por linha a linguagem Go apresentou o menor resultado, necessitando apenas de 4,6 lexemas em média por linha de código, enquanto Rust apresentou a quantidade mais elevada, com um resultado de 6,3. A linguagem C demonstrou um resultado novamente próximo da média entre as linguagens, com um total de 5,2.

Desempenho

Comparação do tempo de execução da implementação do algoritmo de eliminação de Gauss nas diferentes linguagens, considerando diferentes tamanhos de matriz.

Tamanho da Matriz	7x8		35x36		90x91	
	Mín.	Máx.	Mín.	Máx.	Mín.	Máx.
C	0	0	0	15ms	0	15ms
Rust	0,42ms	0,61ms	1,467ms	2,448ms	13,3489ms	17,078ms
Go	0	0,52ms	0,515ms	0,529ms	2,0646ms	2,1309ms

Representação gráfica do quadro acima:



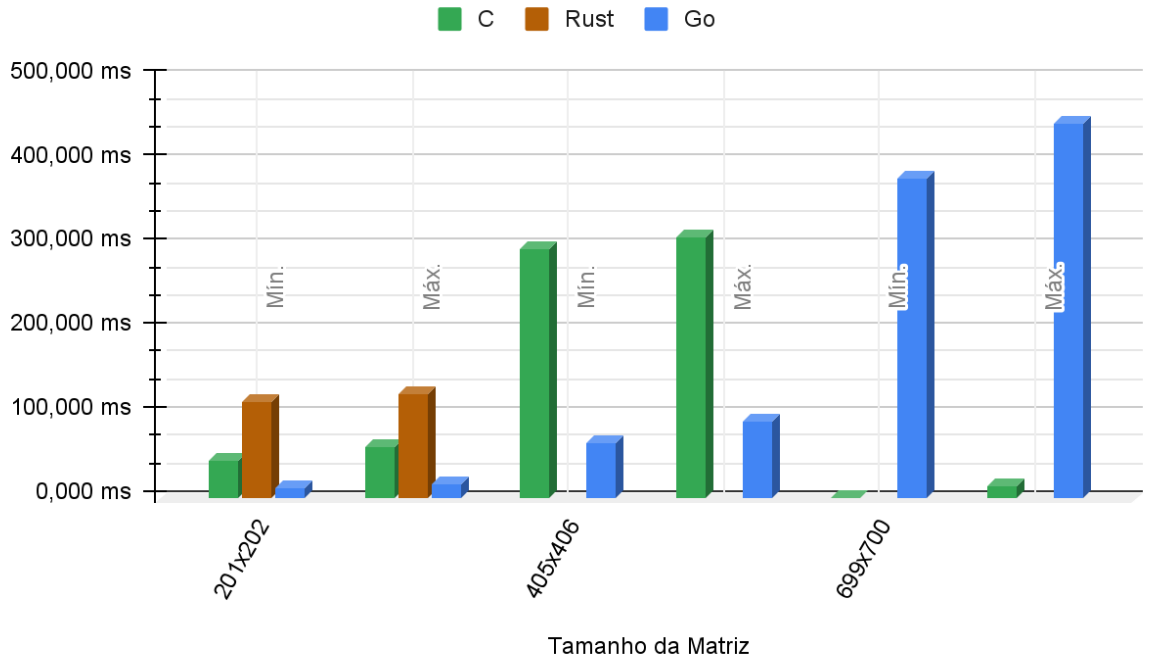
Características da máquina utilizada:

Processador: Intel Core i3-4005U 1.70GHz, cache de 3M, 2 núcleos e 4 threads

Mémoria RAM: 8,00GB

Tamanho da Matriz	201x202		405x406		699x700	
	Mín.	Máx.	Mín.	Máx.	Mín.	Máx.
C	46ms	62ms	296ms	312ms	0	15ms
Rust	115,794ms	123,869ms	X	X	X	X
Go	12,155ms	19.563ms	65,581ms	92,455ms	382,048ms	447,195ms

Representação gráfica do quadro acima:



Características da máquina utilizada:

Processador: Intel Core i3-4005U 1.70GHz, cache de 3M, 2 núcleos e 4 threads

Mémoria RAM: 8,00GB

Conclusão

Comparando a implementação de um algoritmo de eliminação de Gauss nas três linguagens de programação podemos observar, a partir dos resultados obtidos, que a melhor linguagem para ser aplicada é de fato uma resposta subjetiva, sujeita ao foco de cada projeto em particular. Rust, por exemplo, apresentou a menor necessidade de linhas de código, porém exigiu mais lexemas em média por linha, enquanto que Go apresentou um resultado completamente inverso nesse aspecto. A linguagem C por sua vez produziu resultados constantes bem próximos dos valores médios das outras duas linguagens. Isto pode ser um demonstrativo do resultado das diferentes abordagens praticadas por cada uma.

No quesito desempenho, a linguagem C é amplamente conhecida por ser rápida, já que permite um controle direto sobre a memória e o processamento. Rust e Go têm um desempenho comparável, mas Rust é mais adequado para aplicações de alto desempenho, enquanto Go é mais adequado para aplicações de concorrência e escalabilidade.

Quanto à facilidade de uso, a linguagem C possui uma curva de aprendizado mais íngreme do que Rust e Go, por exigir um conhecimento mais profundo sobre gerenciamento de memória e tipos de dados. Entre Rust e Go, Rust é considerada de mais difícil aprendizagem devido ao seu ambiente de código mais seguro e gerenciamento de memória estático, enquanto Golang é mais fácil de trabalhar graças a sua abordagem menos rígida em relação ao controle de memória.

O fator de segurança é um quesito fortemente reconhecido de Rust pelo seu sistema de tipo estático que impede muitos erros de memória e segurança em tempo de compilação. Go apresenta recursos de segurança através da coleta de lixo automática e do controle de limites de memória. Já em C, grande parte da segurança dos processos é delegada ao programador.

Concluindo, os resultados reafirmam a ideia de que a escolha da linguagem de programação depende das necessidades específicas do problema a ser resolvido. Se o desempenho é uma preocupação primordial, a linguagem C continua a ser uma escolha sólida. Se for a segurança, então Rust será uma boa escolha. Se a escalabilidade é importante, Go pode ser a escolha certa. Em última análise, a resposta final vai depender do conhecimento e preferência do programador associada ao objetivo alvo da aplicação a ser desenvolvida.

Referências

[1] C Program for Gauss Elimination Method. Code with C, March 28, 2014. Disponível em: <https://www.codewithc.com/c-program-for-gauss-elimination-method/>. Acesso em: 28 de Abril de 2023.

[2] gaussian-elimination-pthreads-openmp/gauss.c at master. gmendonca via github, Oct 8, 2015. Disponível em: <https://github.com/gmendonca/gaussian-elimination-pthreads-openmp/blob/master/gauss.c>. Acesso em: 28 de Abril de 2023.

[3] The Rust Programming Language. Disponível em: <https://doc.rust-lang.org/book/>. Acesso em: 28 de Abril de 2023.

[4] Effective Go - The Go Programming Language. Disponível em: <https://go.dev/doc/effective_go>. Acesso em: 28 de Abril de 2023.

[5] Sistemas Lineares - Método de Gauss. 19 de fev. de 2021. Disponível em: <https://cn.ect.ufrn.br/index.php?r=conteudo%2Fsislin-gauss>. Acesso em: 26 de Abril de 2023.