****

**Universidade Estadual de Santa Cruz – UESC**

**Relatórios de Implementações de Métodos da Disciplina Análise Numérica**

**Relatório de implementações realizadas por Iago Gomes Santana**

**Disciplina Análise Numérica.**

**Curso Ciência da Computação**

**Semestre 2031.1**

**Professor Gesil Sampaio Amarante II**

**Ilhéus – BA**

**2023**

ÍNDICE

[**Estratégia de Implementação: 6**](#_heading=h.591oldvrkm39)

[**Estrutura dos Arquivos de Entrada/Saída  
Entrada: 6**](#_heading=h.e3zz9zq3c1cg)

[**Saída: 6**](#_heading=h.30j0zll)

[**Problema teste 1, 2, 3... 7**](#_heading=h.cekoel9npo7y)

[**Dificuldades enfrentadas 7**](#_heading=h.hao5zwmylzps)

[**Estratégia de Implementação: 7**](#_heading=h.qqdjqt5cb7x7)

[**Estrutura dos Arquivos de Entrada/Saída  
Entrada: 8**](#_heading=h.a66bfiz80a7g)

[**Saída: 8**](#_heading=h.nsr6rrd4ngl5)

[**Problema teste 1, 2, 3... 8**](#_heading=h.s6snnm5tu3by)

[**Dificuldades enfrentadas 9**](#_heading=h.2vdsnpshb34a)

[**Estratégia de Implementação: 9**](#_heading=h.giwoe3ftqmyb)

[**Estrutura dos Arquivos de Entrada/Saída  
Entrada: 9**](#_heading=h.k4jdnt78tvfc)

[**Saída: 9**](#_heading=h.yupicc66cxha)

[**Problema teste 1, 2, 3... 10**](#_heading=h.2ira81hobgby)

[**Dificuldades enfrentadas 10**](#_heading=h.i0q6n41xueh5)

[**Estratégia de Implementação: 10**](#_heading=h.w0jca3lwm9pd)

[**Estrutura dos Arquivos de Entrada/Saída  
Entrada: 11**](#_heading=h.o2m0pxqnpd1v)

[**Saída: 11**](#_heading=h.gnqg51hwozas)

[**Problema teste 1, 2, 3... 11**](#_heading=h.fan34nb3mi50)

[**Dificuldades enfrentadas 11**](#_heading=h.r588ip8mzkrg)

[**Estratégia de Implementação: 11**](#_heading=h.wy806i1570as)

[**Estrutura dos Arquivos de Entrada/Saída  
Entrada: 12**](#_heading=h.cqa2z1p7rn7j)

[**Saída: 12**](#_heading=h.afz6jgvbsr0s)

[**Problema teste 1, 2, 3... 12**](#_heading=h.7ehw8qptjn4p)

[**Dificuldades enfrentadas 13**](#_heading=h.1fob9te)

[**Estratégia de Implementação: 13**](#_heading=h.3d27s5nbf1kp)

[**Estrutura dos Arquivos de Entrada/Saída  
Entrada: 14**](#_heading=h.k8o57bucjraq)

[**Saída: 14**](#_heading=h.ml45a8yiitdx)

[**Problema teste 1, 2, 3... 14**](#_heading=h.3znysh7)

[**Dificuldades enfrentadas 15**](#_heading=h.2et92p0)

[**Estratégia de Implementação: 15**](#_heading=h.77yz7bm92fkx)

[**Estrutura dos Arquivos de Entrada/Saída  
Entrada: 16**](#_heading=h.wzgnolp60qyc)

[**Saída: 16**](#_heading=h.u5fcevjod9tp)

[**Problema teste 1, 2, 3... 17**](#_heading=h.cnvt4xuj9lnt)

[**Dificuldades enfrentadas 17**](#_heading=h.ydw996sqq8ii)

[**Estratégia de Implementação: 17**](#_heading=h.wgax0spd3sql)

[**Estrutura dos Arquivos de Entrada/Saída  
Entrada: 18**](#_heading=h.fgolw7gkoplm)

[**Saída: 19**](#_heading=h.hs7p0a7bcg4e)

[**Problema teste 1, 2, 3... 19**](#_heading=h.7ta2342ybupw)

[**Dificuldades enfrentadas 19**](#_heading=h.lcvvy0gh2cgq)

[**Estratégia de Implementação: 20**](#_heading=h.cu9xyidnmuju)

[**Estrutura dos Arquivos de Entrada/Saída  
Entrada: 20**](#_heading=h.a2cko557vgb1)

[Saída: 21](#_heading=h.umv2qmcppsjy)

[**Problema teste 1, 2, 3... 21**](#_heading=h.of4rojarhsqz)

[**Dificuldades enfrentadas 21**](#_heading=h.m7bcltlimgek)

Linguagem Escolhida e justificativas:

A linguagem escolhida foi o Python (versão 3.8.10) por sua facilidade em lidar com tipos diferentes de dados e praticidade ao programar algumas funções matemáticas. Outro ponto importante para a escolha foi com relação a leitura do .txt de entrada e a conversão para f(x), algo que eu tive muita dificuldade em implementar, mas pelo fato da linguagem possuir uma vasta comunidade ativa, consegui lidar com este problema.  
Uma desvantagem que percebi no python foi que, em alguns casos, quando envolvem operações com números decimais com muitos dígitos após a vírgula, os resultados podem ser imprecisos. Porém é possível contornar esse déficit utilizando a libs específicas para cálculo de alta precisão.

No mais, python é uma boa escolha no geral por ser uma linguagem de sintaxe enxuta e ágil para se programar, não chega a ser tão eficiente quanto C, mas para o nosso escopo de problemas, essa perda na eficiência não será impactante.

Arquitetura:

**Estratégia de Implementação:**

Por ser um projeto com cada método tendo sua especificidade a ser implementada, ao mesmo tempo que cada um dispõe de semelhanças tais quais a leitura e escrita em arquivo e a função que calcula f(x), optei por uma estrutura que se baseia em:

* Ter um diretório raiz chamado “metodos\_numericos” e dentro dele tendo mais 3 diretórios
* implementations:
  + Aqui eu tenho 1 arquivo python para cada método implementado. Nesses arquivos temos apenas a implementação das especificidades do método em si.
  + Temos também o diretório “utils” que possui 2 arquivos python, com a implementação de uma classe para leitura e escrita em arquivo de texto, tal como um pacote para realizar o cálculo de uma função de X (ou f(x) ), á partir de uma string e um valor para X
* Inputs:
  + Diretório onde devem ser criados os arquivos de inputs para cada método. A princípio cada input deve ter o nome de seu respectivo método, mas isso pode ser mudado no decorrer do desenvolvimento.
* Outputs:
  + Diretório de saída onde o programa escreverá os resultados dos cálculos. Novamente cada arquivo terá o nome de seu respectivo método, mas isso também poderá mudar até a data de entrega do relatório.

Esta é a estrutura básica do projeto, vale salientar mais 2 arquivos na raiz que podem ser úteis, o ‘README.md’, que irá conter tutoriais e explicações. E o ‘makefile’, que auxilia na execução dos programas.

**OBS.:** os comandos make só funcionam no linux, caso use windows, pode aproveitar a linha do comando dentro do makefile para executar o código.

Bissecção

## Estratégia de Implementação:

Esse método me lembrou bastante uma busca binária, então eu acabei usando a mesma estratégia para implementar separando os valores de a, b, f(a), f(b) e calculando c e f(c) com base neles. A partir daí, foi só fazer as verificações para saber se f(c) era menor que meu erro, se o intervalo [a,b] era menor que meu erro, e por fim, se c ficaria no lugar de a ou b.

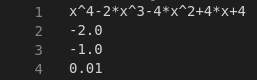
A verificação que indica se C assume o lugar de A ou B é feita ao multiplicar f(a) por f(c), caso haja troca de sinal, ou seja, caso o resultado dessa multiplicação seja negativo, C assume o lugar de B. Caso contrário, C assume o lugar de A

## Estrutura dos Arquivos de Entrada/Saída Entrada:

Me esforcei ao máximo para que o formato das funções f(x) ficassem o mais semelhantes ao formato que usamos ao escrever essas funções a mão. Os diferenciais mais relevantes são a necessidade de adicionar um ‘\*’ para explicitar a multiplicação de X, e a exponenciação sendo denotada por ‘^’.

Para este método, o nome do arquivo de entrada deve ser ‘bissection.txt’ e deve seguir a ordem de: função f(x); a; b; erro;

segue abaixo um exemplo:



## Saída:

Para os arquivos de saída, teremos o mesmo nome que o arquivo de entrada, porém no diretório específico de output. A formatação da saída é: função f(x) : X = valor de x. Segue abaixo o output para a entrada acima.



## Problema teste 1, 2, 3...

* 1. 0.9-(1+x+((x^(2))/2))\*e^(-x):  
      X9 = 1.10216796875
  2. 0.1-(1+x+((x^(2))/2))\*e^(-x)  
     X10 = 5.322265625

1. tan(40\*pi/180)\*((4/5)-cos(x)^2)-sin(x)\*cos(x)  
   X13 = 1.0237915039062502
   1. (140/26.50)\*((1+x)^10)-((140/26.50)+1)\*((1+x)^9)+1   
      X12 = 0.12216796875
   2. (140/21.50)\*((1+x)^13)-((140/21.50)+1)\*((1+x)^12)+1  
        
      X7 = 0.109375

## 

## 

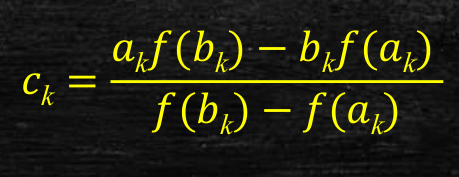
## Dificuldades enfrentadas

Quanto à implementação do método, foi tudo tranquilo.

Posição Falsa

## Estratégia de Implementação:

Para este método, eu reaproveitei toda a implementação do método da secante, alterando somente a forma como eu escolho o próximo valor de C, saindo do método que se assemelha a uma busca binária (forma usada na bissecção), para a forma que usa a fórmula dada em sala de aula:

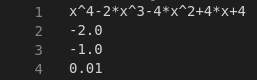


E para calcular a fórmula acima, também foi necessário calcular o valor de f(b), que na bissecção não era necessário.

## Estrutura dos Arquivos de Entrada/Saída Entrada:

Para este método, o nome do arquivo de entrada deve ser regula\_falsi.txt’ e deve seguir a ordem de: função f(x); a; b; erro;

segue abaixo um exemplo:



## Saída:

Para os arquivos de saída, teremos o mesmo nome que o arquivo de entrada, porém no diretório específico de output. A formatação da saída é: função f(x) : X = valor de x. Segue abaixo o output para a entrada acima.



# 

## Problema teste 1, 2, 3...

* 1. 0.9-(1+x+((x^(2))/2))\*e^(-x):  
      X6 = 1.101869571108599
  2. 0.1-(1+x+((x^(2))/2))\*e^(-x)  
     X6 = 5.323450063238776

1. tan(40\*pi/180)\*((4/5)-cos(x)^2)-sin(x)\*cos(x)  
   X5 = 1.023760230610521
   1. (140/26.50)\*((1+x)^10)-((140/26.50)+1)\*((1+x)^9)+1   
      X12 = 0.10734603160934908
   2. (140/21.50)\*((1+x)^13)-((140/21.50)+1)\*((1+x)^12)+1  
      X12 = 0.10145015206680909

## Dificuldades enfrentadas

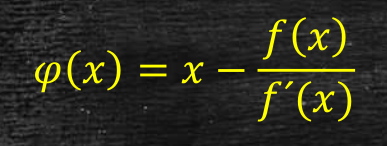
Tal como o método da bissecção, posição falsa foi um método tranquilo de ser implementado.

Newton-Raphson

## Estratégia de Implementação:

Para a implementação deste método, foi necessário instalar e importar uma biblioteca externa, a **sympy.** Essa biblioteca tem uma função que me permite calcular derivadas que serão usadas no método. A lib pode ser instalada usando o comando ‘pip install sympy’

Seguindo a lógica do método Newton-Raphson, eu só preciso de um valor inicial para x, que será enviado no input de dados, e calcular a derivada da função f(x), que será feito através da lib sympy. Tendo essas duas informações, basta substituir na fórmula e iterar até que o resultado tenha um erro aceitável:



## Estrutura dos Arquivos de Entrada/Saída Entrada:

Para este método, o nome do arquivo de entrada deve ser newton\_raphson.txt’ e deve seguir a ordem de: função f(x); x1; erro;

segue abaixo um exemplo:



## Saída:

Para os arquivos de saída, teremos o mesmo nome que o arquivo de entrada, porém no diretório específico de output. A formatação da saída é: função f(x) : X = valor de x. Segue abaixo o output para a entrada acima.



## Problema teste 1, 2, 3...

* 1. 0.9-(1+x+((x^(2))/2))\*e^(-x):  
      X6 = 1.101869571108599
  2. 0.1-(1+x+((x^(2))/2))\*e^(-x)  
     X6 = 5.323450063238776

1. tan(40\*pi/180)\*((4/5)-cos(x)^2)-sin(x)\*cos(x)  
   X5 = 1.023760230610521
   1. (140/26.50)\*((1+x)^10)-((140/26.50)+1)\*((1+x)^9)+1   
      X3 = 0.12224831517905059
   2. (140/21.50)\*((1+x)^13)-((140/21.50)+1)\*((1+x)^12)+1  
      X2 = 0.10937840270843556

## Dificuldades enfrentadas

Para este método, encontrei certa dificuldade para lidar com as derivadas. Após um tempo tentando implementar decidi optar pela solução mais produtiva, que foi encontrar uma lib que lide com isso. Queria muito evitar o uso de pacotes externos, mas não achei sábio insistir tanto ao ponto de perder muito tempo buscando outra solução.  
Porém, mesmo após derivar, tive problema em lidar com os componentes não numéricos (log, tang, sin, etc)

Secante

## Estratégia de Implementação:

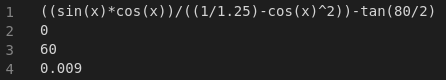
Primeiro eu calculo o número máximo de iterações usando a fórmula: ceil(log((b-a)/tol)/log((1+sqrt(5))/2))

Em seguida apenas calculo o resultado das funções f(x0) e f(x1), uso seus resultados na fórmula do método da secante, conseguindo assim o valor de xk+2. Repito processo até que o valor de f(xk+2) seja menor que meu erro ou que eu exceda o número máximo de iterações calculado anteriormente

## Estrutura dos Arquivos de Entrada/Saída Entrada:

Para este método, o nome do arquivo de entrada deve ser ‘secante.txt’ e deve seguir a ordem de: função f(x); a; b; erro;

segue abaixo um exemplo:



## Saída:

Para os arquivos de saída, teremos o mesmo nome que o arquivo de entrada, porém no diretório específico de output. A formatação da saída é: função f(x) : X = valor de x. Segue abaixo o output para a entrada acima.



## Problema teste 1, 2, 3...

* 1. 0.9-(1+x+((x^(2))/2))\*e^(-x):  
      X2 = 0.9685968893052013
  2. 0.1-(1+x+((x^(2))/2))\*e^(-x)  
     X2 = 5.263022237438587

1. tan(40\*pi/180)\*((4/5)-cos(x)^2)-sin(x)\*cos(x)  
   X2 = 0.9688391894365167
   1. (140/26.50)\*((1+x)^10)-((140/26.50)+1)\*((1+x)^9)+1   
      X2 = 0.1013822448853539
   2. (140/21.50)\*((1+x)^13)-((140/21.50)+1)\*((1+x)^12)+1  
      X2 = 0.10025661836037342

## 

## Dificuldades enfrentadas

Eliminação de Gauss

## Estratégia de Implementação:

Implementei 2 funções auxiliares, uma para calcular M e outra para calcular L, não foi algo tão necessário, mas eu achei mais organizado fazer assim. Usei um for e uma variável de controle (mx) para calcular m com base no estado atual da matriz e usei esse m na função l(x, m, y) para ter l.

A partir daí, foi só fazer a substituição voltando da linha final até a inicial da matriz, substituindo os valores de x por seus respectivos valores descobertos na iteração anterior

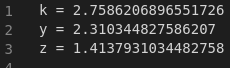
## Estrutura dos Arquivos de Entrada/Saída Entrada:

Para a entrada desse programa, espera-se um arquivo na pasta de “inputs” chamado "gauss.txt" e esse arquivo deve seguir o padrão de escrita do sistema linear linha a linha, substituindo x1, x2, x3… por x,y,z… Segue abaixo um exemplo:

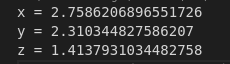
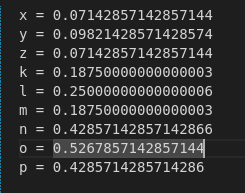


## Saída:

Para a saída, teremos uma escrita semelhante à da entrada, com a diferença de que, para cada linha do sistema linear teremos uma incógnita (x,y,z) e seu respectivo valor. Segue abaixo um exemplo:



## Problema teste 1, 2, 3...

1. Questão 4.1  
   
2. Questão 4.3  
   
3. Questão 4.6  
   

## Dificuldades enfrentadas

Fatoração LU

## Estratégia de Implementação:

Para a implementação desse método, eu desenvolvi 2 funções auxiliares para o cálculo do somatório para a matriz L e U, como elas tem uma pequena diferença, achei melhor deixá-las separadamente.

Eu criei uma matriz de tamanho igual a da questão e inicializei ela com zeros. Usei as fórmulas para preencher a primeira linha da matriz U e a primeira coluna da matriz L. Após isso, usei a outra fórmula para preencher as outras posições de ambas matrizes, de modo a preencher apenas a diagonal principal e o que estiver acima para U, e a parte abaixo da diagonal principal de L, deixando assim zeros nas posições que fogem dessa regra.

A partir daí, foi só fazer a substituições e encontrar o resultado para o sistema

## Estrutura dos Arquivos de Entrada/Saída Entrada:

Para a entrada desse programa, espera-se um arquivo na pasta de “inputs” chamado "LU.txt" e esse arquivo deve seguir o padrão de escrita do sistema linear linha a linha, substituindo x1, x2, x3… por x,y,z… Segue abaixo um exemplo:

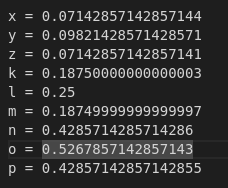
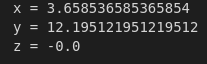


## Saída:

Para a saída, teremos uma escrita semelhante à da entrada, com a diferença de que, para cada linha do sistema linear teremos uma incógnita (x,y,z) e seu respectivo valor. Segue abaixo um exemplo:

# 

## Problema teste 1, 2, 3...

1. Questão 4.1  
   
2. Questão 4.3  
   
3. Questão 4.6  
   

## Dificuldades enfrentadas

Matrizes com incógnitas que fazem operações enquanto estão sendo multiplicadas por uma mesma constante.

Jacobi

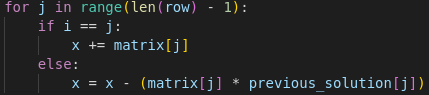
## Estratégia de Implementação:

O método de Jacobi foi um pouco mais complicado, para simular a aplicação da fórmula geral, eu precisei usar vetores e fazer loopings encadeados, mas tentando explicar de modo objetivo:

Eu armazeno cada linha do sistema linear em uma lista de floats de forma que o B\_n seja sempre o último elemento da lista.

Então para resolver cada X\_n, eu faço uma cópia da lista que representa a linha original, e troco o valor do índice n pelo valor do último ítem, substituindo assim B\_n pelo A\_nn \* X\_n que estamos tentando encontrar.

Após ter feito esta substituição, basta usar um segundo looping percorrendo cada elemento da lista, somando o valor de B\_n (posição na qual o iterador dos loopings aninhados é igual) e decremento os “valores restantes”, como ilustrado abaixo.



Após executar todo esse looping, o valor de X\_n é dividido pelo valor de seu A\_nn.

Vale ressaltar que “os valores restantes” são compostos pelos outros elementos da lista \* seu respectivo elemento na lista de soluções.

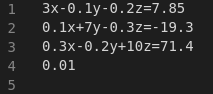
A lista de soluções por sua vez, é iniciada com todos os valores de X valendo zero, e vai sendo atualizada a cada iteração do método (após o método percorrer toda a matriz)

Este processo se repete até que a variância absoluta seja menor que o erro ou que seja atingido um número máximo de iterações que é definido no programa como sendo 30.

Outro detalhe da implementação é o uso de uma função que antes mesmo de executar o método de Jacobi, verifica se o sistema linear irá convergir. Essa função faz nada mais que verificar se o elemento da diagonal principal é maior que a soma dos módulos dos elementos de sua respectiva linha tal como, maior que a soma dos módulos dos elementos de sua respectiva coluna.

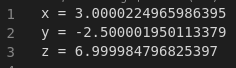
## Estrutura dos Arquivos de Entrada/Saída Entrada:

Para a entrada desse programa, espera-se um arquivo na pasta de “inputs” chamado "jacobi.txt" e esse arquivo deve seguir o padrão de escrita do sistema linear linha a linha, substituindo x1, x2, x3… por x,y,z… e após escrever todo o sistema linear, na linha seguinte, adicionar o valor de tolerância do erro. Segue abaixo um exemplo:

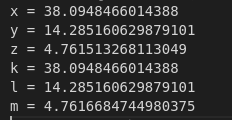


## Saída:

Para a saída, teremos uma escrita semelhante à da entrada, com a diferença de que, para cada linha do sistema linear teremos uma incógnita (x,y,z) e seu respectivo valor. Segue abaixo um exemplo:



## Problema teste 1, 2, 3...

1. Questão 5.1  
   
2. Questão 5.2  
   
3. Questão 5.5  
   Não converge

## 

## 

## Dificuldades enfrentadas

Gauss-Seidel

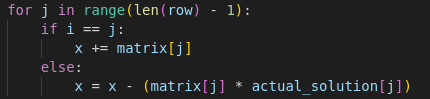
## Estratégia de Implementação:

O método de Jacobi foi um pouco mais complicado, para simular a aplicação da fórmula geral, eu precisei usar vetores e fazer loopings encadeados, mas tentando explicar de modo objetivo:

Eu armazeno cada linha do sistema linear em uma lista de floats de forma que o B\_n seja sempre o último elemento da lista.

Então para resolver cada X\_n, eu faço uma cópia da lista que representa a linha original, e troco o valor do índice n pelo valor do último ítem, substituindo assim B\_n pelo A\_nn \* X\_n que estamos tentando encontrar.

Após ter feito esta substituição, basta usar um segundo looping percorrendo cada elemento da lista, somando o valor de B\_n (posição na qual o iterador dos loopings aninhados é igual) e decremento os “valores restantes”, como ilustrado abaixo.



Após executar todo esse looping, o valor de X\_n é dividido pelo valor de seu A\_nn.

Vale ressaltar que “os valores restantes” são compostos pelos outros elementos da lista \* seu respectivo elemento na lista de soluções, que nesse método é atualizada a cada linha do sistema linear, e não mais a cada iteração por todo o sistema como era no caso de Jacobi.

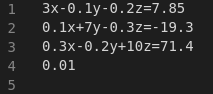
A lista de soluções por sua vez, é iniciada com todos os valores de X valendo zero, e vai sendo atualizada a cada iteração do método (após o método percorrer toda a matriz)

Este processo se repete até que a variância absoluta seja menor que o erro ou que seja atingido um número máximo de iterações que é definido no programa como sendo 30.

Outro detalhe da implementação é o uso de uma função que antes mesmo de executar o método de Jacobi, verifica se o sistema linear irá convergir. Essa função faz nada mais que verificar se o elemento da diagonal principal é maior que a soma dos módulos dos elementos de sua respectiva linha tal como, maior que a soma dos módulos dos elementos de sua respectiva coluna.

## Estrutura dos Arquivos de Entrada/Saída Entrada:

Para a entrada desse programa, espera-se um arquivo na pasta de “inputs” chamado "jacobi.txt" e esse arquivo deve seguir o padrão de escrita do sistema linear linha a linha, substituindo x1, x2, x3… por x,y,z… e após escrever todo o sistema linear, na linha seguinte, adicionar o valor de tolerância do erro. Segue abaixo um exemplo:

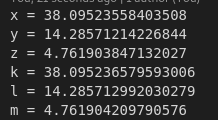
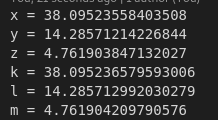


## Saída:

Para a saída, teremos uma escrita semelhante à da entrada, com a diferença de que, para cada linha do sistema linear teremos uma incógnita (x,y,z) e seu respectivo valor. Segue abaixo um exemplo:



## Problema teste 1, 2, 3...

1. Questão 5.1  
   
2. Questão 5.2  
   
3. Questão 5.5  
   Não converge

## Dificuldades enfrentadas

Inversão

## Estratégia de Implementação:

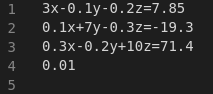
O código começa realizando o pivotamento da matriz na diagonal principal, a fim de evitar divisões por zero no próximo passo. Em seguida, ele cria uma matriz identidade do mesmo tamanho que a matriz original e concatena as duas matrizes em uma matriz ampliada.

O próximo passo é reduzir a matriz ampliada à forma escalonada por meio de operações elementares de linha, que transformam a matriz ampliada em uma matriz triangular superior. Então, ele realiza operações elementares adicionais de linha para transformar a matriz triangular superior em uma matriz diagonal.

Por fim, ele extrai a matriz inversa da parte da matriz ampliada que corresponde à identidade original e retorna essa matriz inversa. O código lê a matriz a ser invertida de um arquivo de entrada, calcula a inversa para cada matriz no arquivo e grava as matrizes inversas em um arquivo de saída.

## Estrutura dos Arquivos de Entrada/Saída Entrada:

Para a entrada desse programa, espera-se um arquivo na pasta de “inputs” chamado "jacobi.txt" e esse arquivo deve seguir o padrão de escrita do sistema linear linha a linha, substituindo x1, x2, x3… por x,y,z… e após escrever todo o sistema linear, na linha seguinte, adicionar o valor de tolerância do erro. Segue abaixo um exemplo:



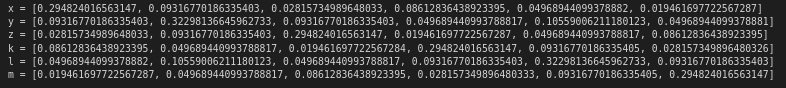
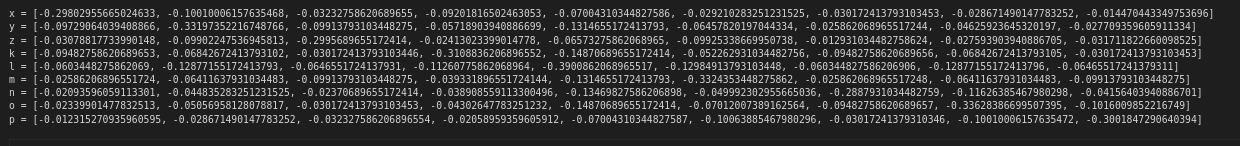
## Saída:

Para a saída, teremos uma escrita semelhante à da entrada, com a diferença de que, para cada linha do sistema linear teremos uma incógnita (x,y,z) e seu respectivo valor. Segue abaixo um exemplo:



## 

## Problema teste 1, 2, 3...

1. Questão 5.1  
   
2. Questão 5.2  
   
3. Questão 5.5  
   

## 

## 

## Dificuldades enfrentadas

## 

Considerações Finais

Neste relatório de implementação, foram apresentados nove métodos numéricos amplamente utilizados em diferentes áreas da engenharia e ciência: bissecção, regula falsi, Newton-Raphson, secante, eliminação de Gauss, fatoração LU, Jacobi, Gauss-Seidel e inversão de matriz pelo método de Gauss-Jordan.

A implementação dos métodos numéricos aqui apresentados foi realizada em Python 3. Foram utilizadas bibliotecas padrão do Python, como math e numpy, além de funções auxiliares implementadas especificamente para cada método.

Foram apresentados exemplos de aplicação de cada método numérico, incluindo o cálculo de raízes de funções, sistemas lineares e inversão de matrizes. Os resultados obtidos foram comparados com soluções analíticas sempre que possível e foram considerados satisfatórios.

A implementação dos métodos numéricos aqui apresentados permitiu a compreensão teórica de cada método e forneceu uma base sólida para a aplicação desses métodos em problemas do mundo real. Além disso, a implementação em Python forneceu uma visão mais prática e didática de como esses métodos podem ser implementados em um ambiente computacional.

Em conclusão, a implementação dos nove métodos numéricos apresentados neste relatório demonstrou sua relevância e aplicabilidade em diferentes áreas da engenharia e ciência. Esperamos que este relatório possa ser útil para aqueles que desejam compreender melhor a teoria por trás desses métodos e como eles podem ser implementados de forma prática em um ambiente computacional.