UNIVERSIDADE ESTADUAL DE SANTA CRUZ
MESTRADO EM MODELAGEM COMPUTACIONAL

MÉTODOS DE SOLUÇÃO DE SISTEMAS DE EQUAÇÕES LINEARES E ALGÉBRICAS

Mateus Chaves Barbosa (202210182)

Ilhéus-Bahia
2022

Mateus Chaves Barbosa (202210182)

Métodos de solução de sistemas de equações lineares e algébricas.

Relatório apresentado como parte dos critérios de avaliação da disciplina: Metodos numéricos I.

Professor(a): Dany Sanchez Dominguez.

Ilhéus-Bahia

2022

Sumário

1 Introdução

Os métodos de solução de sistemas lineares do tipo $Ax = b$ são de extrema importância para as diversas áreas da ciência e da tecnologia. Existem aplicações desses métodos em áreas como matemática, engenharia, biologia e até em economia onde existem sistemas que podem ser dos mais simples e menores, até os problemas de grande porte com milhares de equações e que exigem um grande poderio computacional.

Estes sistemas surgem da necessidade de se modelar e resolver problemas práticos, como balenceamento de equações químicas, circuitos elétricos e até mesmo para o cálculo de uma alimentação diária e equilibrada. De maneira geral, os sitemas de equações lineares possuem a seguinte forma:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2 \\ a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n = b_m \end{cases} \tag{1}$$

Na matemática, o estudo de sistemas lineares é um dos pilares da da álgebra linear, sendo este presente em várias vertentes. Com o avanço nos estudos da álgebra linear, foram desenvolvidos métodos computacionais para a resolução de problemas envolvendo a mesma incluindo os sistemas lineares, o uso de resursos computacionais para a resolução de problemas em álgebra linerar dá a base para uma área da matemática chamada álgebra linear computacional.

Um sistema linear pode ser classificado também, como um sistema de equações de grau um, ou seja, um sistema onde as equações possuem apenas polinômios em que cada parcela possui apenas uma incógnita. Portanto num sistema linear não há potências diferentes de um ou zero e consequentemente não há multiplicação entre incógnitas.

Em geral, os coeficientes das equações são valores reais ou complexos e as suas soluções são encontradas no mesmo conjunto de números, mas os métodos, sejam eles teóricos ou computacionais, aplicam os coeficientes e soluções em qualquer campo (conjunto).

De forma geral, os métodos para a resolução desses sistemas se dividem em duas partes, os métodos diretos e os iterativos. Ambos serão abordados nesse trabalho para um sistema específico, gerado por uma matriz específica, chamada matriz de Hilbert. Dentre esses métodos estão: A eliminação de Gauss com substituição recessiva, com pivotamento parcial, total, Gauss-jacobi, Gauss-Seidel, sob-relaxamento entre outros. Todos eles tem seus pontos positivos e negativos e esses pontos serão discutidos no corpo desse trabalho.

2 Problema1: Matriz de Hilbert

A matriz de Hilbert é uma matriz que segue uma determinada regra de formação. É uma matriz de ordem n e é composta por elementos h_{ij}, onde esses elementos são dados pela seguinte lei de formação:

$$h_{ij} = \frac{1}{i+j-1} \tag{2}$$

Para uma matriz de Hilbert de ordem n, teremos a seguinte forma:

$$H_n = \begin{pmatrix} 1 & 1/2 & 1/3 & \cdots & 1/n \\ 1/2 & 1/3 & 1/4 & \cdots & 1/n+1 \\ 1/3 & 1/4 & 1/5 & \cdots & 1/n+2 \\ \vdots & \vdots & \vdots & & \vdots \\ 1/n & 1/n+1 & 1/n+2 & \cdots & 1/2n-1 \end{pmatrix}$$

Os métodos que serão aplicados a seguir utilizarão esta matriz para compor o sistema, portanto, teremos um sistema linear da forma $Hx = b$, onde a matriz A foi subistituída pela matriz de Hilbert H.

A implementação dessa matriz para a sua utilização nos métodos é bastante simples. Utilizando a linguagem de programação Python foi feito um algoritmo para gerar a matriz de Hilibert de ordem $n = 5$, $n = 9$ e $n = 15$. O código para tal algoritmo segue abaixo:

```
import numpy as np
def Hilbert(n):

  M_hilbert = np.zeros((n, n))


  for linha in range(n):
    for coluna in range(n):
      M_hilbert[linha][coluna] = 1 / (((linha + 1) + (coluna + 1)) - 1)
```

```python

    return M_hilbert.tolist()
```

No código acima, o retorno dessa função é uma matriz de Hilbert na ordem em que foi dada no parâmetro da função. E esta função foi utilizada em todos os métodos de solução do problema 1.

No código anterior foi gerada a matriz de Hilbert que será utilizada nos métodos a seguir, porém também devemos gerar o vetor b, que pelo enunciado do problema, cada elemento do vetor é dado pela soma das linhas da matriz de Hilbert. O algoritmo que gera este vetor b foi implementado na linguagem python e o mesmo segue abaixo:

```python
import numpy as np

def vetor_soma(m,l):
    soma = []
    for linha in range(len(m)):
        for coluna in range(len(m)):
            if(linha == l) and (coluna >=0) :
                soma.append(m[linha][coluna])

    return sum(soma)

def vetor_b(n):
    vetor_b = []
    for i in range(n):
        vetor_b.append(vetor_soma(Hilbert(n), i))
    return vetor_b
```

Segue abaixo os terminais com as matrizes de Hilbert geradas pelo algoritimo acima, para $n = 5$, $n = 9$ e $n = 15$:

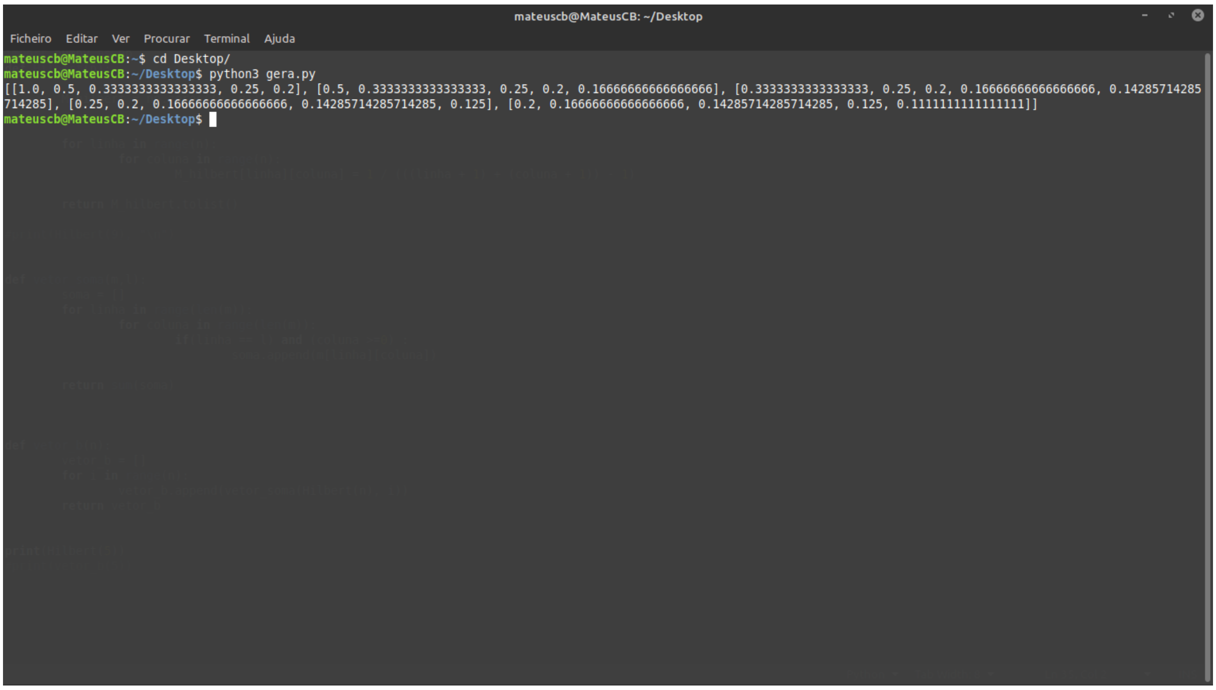

Figura 1: Terminal de execução gerando uma matriz de Hilbert $n = 5$

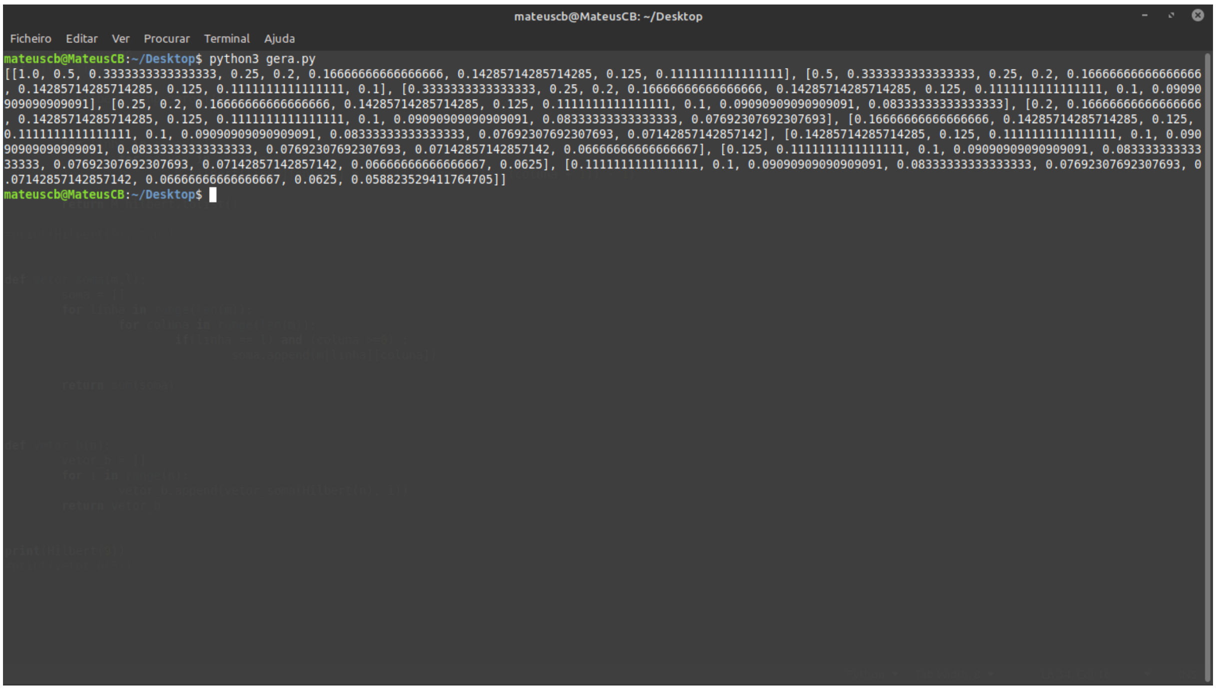

Figura 2: Terminal de execução gerando uma matriz de Hilbert $n = 9$

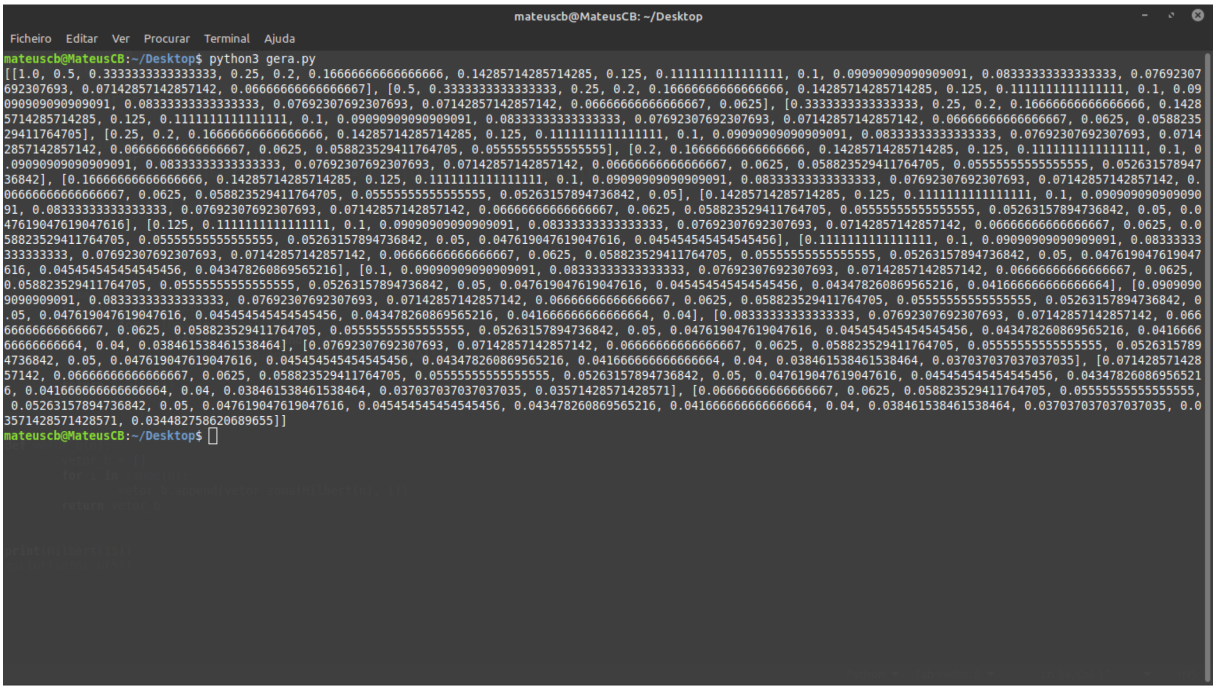

Figura 3: Terminal de execução gerando uma matriz de Hilbert $n = 15$

3 Problema1: Métodos de solução de sistemas de equações lineares do tipo $Hx = b$

Com os elementos do sistema $Hx = b$ em mãos, podemos agora aplicar os métodos de solução de sistemas lineares diretos e iterativos e analisar os seus comportamentos, em relação a precisão, tempo de execução e dificuldade de implementação.

3.1 Métodos diretos: Eliminação de Gauss com substituição regressiva

Este método consiste em evitar o cálculo de forma direta da matriz inversa, que obtemos a partir de um determinado sistema de equações linerares. Para este método de solução, vamos transformar o sistema linear original em um outro sistema linear equivalente com a matriz dos coeficientes sendo agora uma matriz triangular superior.

Em outras palavras, se temos um sistema do tipo $Ax = b$, onde A é uma matriz, b e x são vetores, vamos fazer a transformação $Ax = b \rightarrow Ux = d$ onde U é uma matriz triangular superior. Podemos notar que, após as operações elementares os vetores b e d não serão mais os mesmos, porém o vetor solução x continuará sendo o mesmo.

E uma vez encontrando o sistema equivalente $Ux = d$, podemos encontrar o vetor solução (solução do sistema) fazendo as substituições retroativas (regressivas).

A implementação do método de Gauss com substituição regressiva segue da seguinte forma:

```python

def subs_retro(M, b):
  n = len(M)
  x = n * [0]

  for i in range(n-1, -1, -1):
    S = 0
    for j in range(i+1, n):
      S = S + M[i][j] * x[j]
    x[i] = (b[i] - S) / M[i][i]

  return x


def Gauss(M, b):
  n = len(M)
  for k in range(0, n-1):
    for i in range(k+1, n):
      m = -M[i][k] / M[k][k]
      for j in range(k+1, n):
        M[i][j] = m * M[k][j] + M[i][j]
      b[i] = m * b[k] + b[i]
      M[i][k] = 0

  x = subs_retro(M,b)
  return x
```

A primeira função implementa o algoritmo para fazer as substituições de forma regressiva. As substituições retroativas são feitas seguindo a equação:

$$x_i = \frac{b_i - \displaystyle\sum_{j=i+1}^{n} a_{ij}x_j}{a_{ii}}, i = n, n-1, \ldots, 1. \tag{3}$$

Implementando essa equação na função 'subs_ retro' fazemos um laço for para percorrer as linhas da matriz em ordem decrescente e aplicando a cada uma delas essa equação acima. Segue abaixo o resultado deste método para a matriz de Hilbert com $n = 5$, $n = 9$ e $n = 15$:

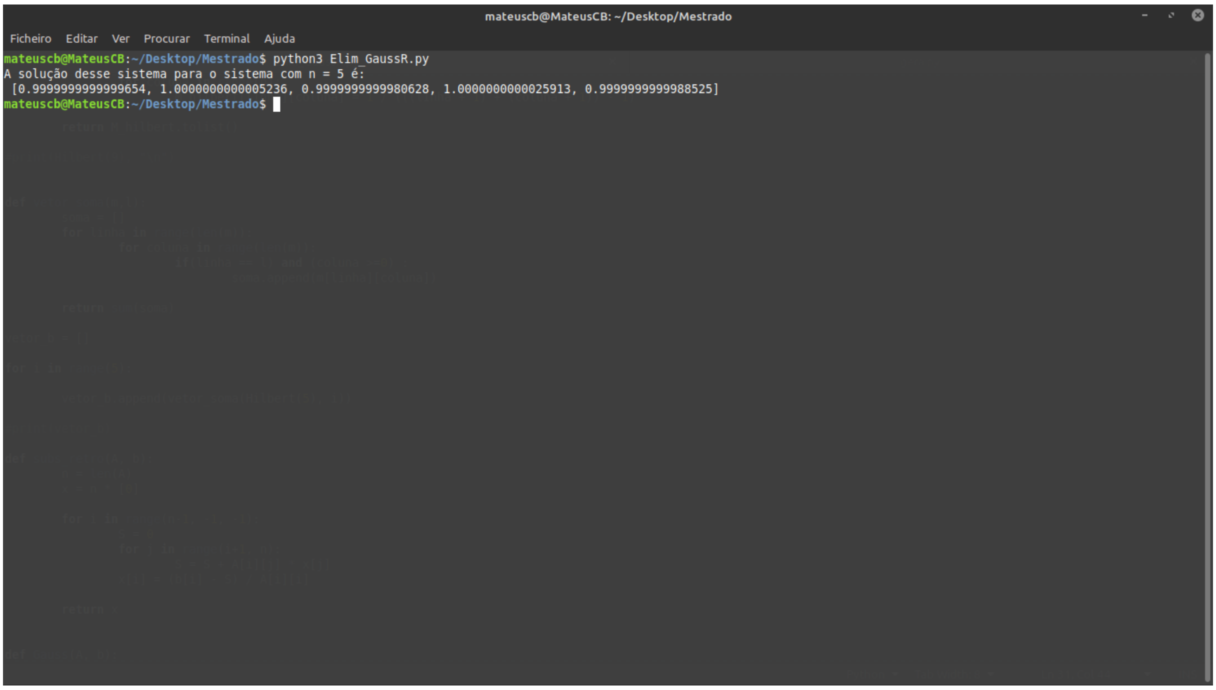

Figura 4: Terminal de execução do método de Gauss por retrosubstituição para $n = 5$

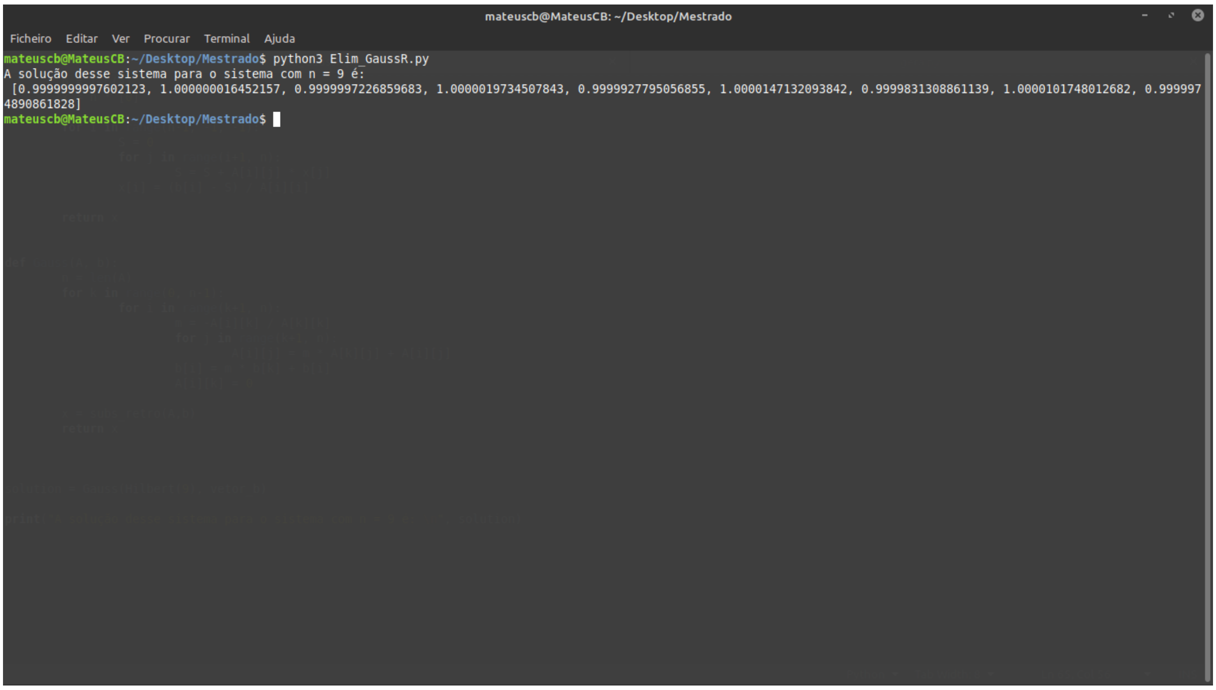

Figura 5: Terminal de execução do método de Gauss por retrosubstituição para $n = 9$

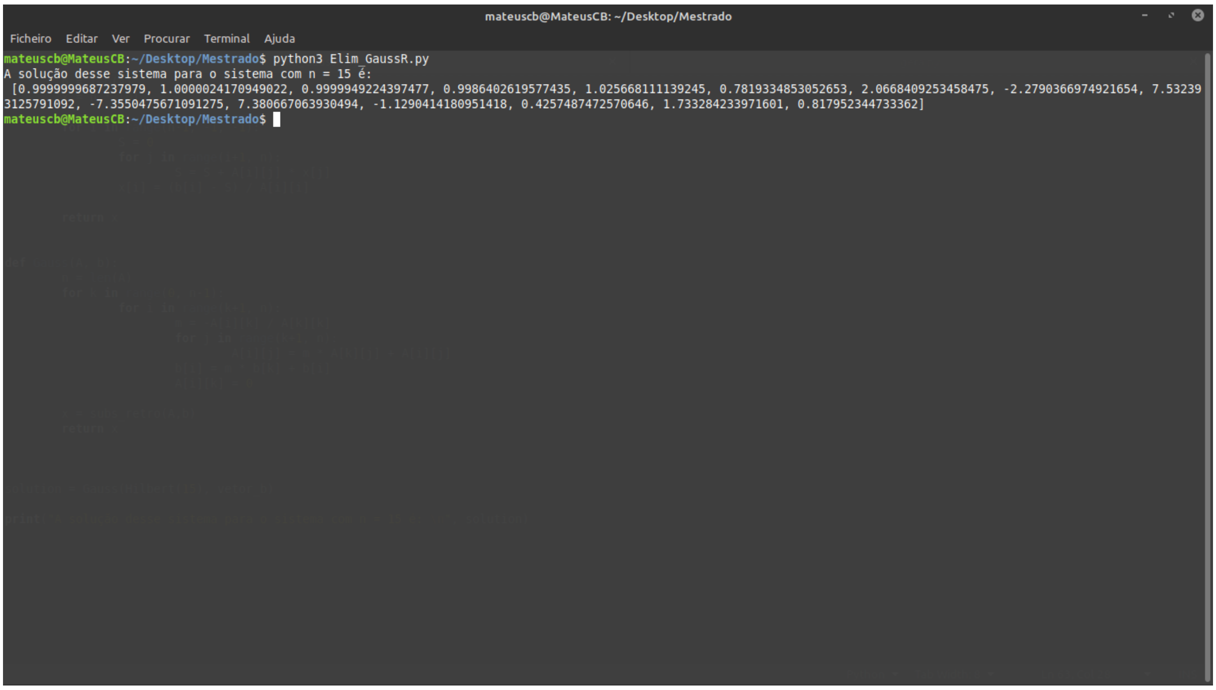

Figura 6: Terminal de execução do método de Gauss por retrosubstituição para $n = 15$

3.2 Métodos diretos: Eliminação de Gauss com pivoteamento parcial

Os método da eliminação gaussiana por pivotamento são utilizados para fazer com que não apareçem zeros no denominador durante a eliminação. Este método pode ser feito de forma parcial, parcial com escala e total. O pivotamento parcial consiste em fazer a troca de linhas para escolher o maior em cada passo. Vale salientar que, para a escolha desse maior pivô devemos considerar o valor dele em módulo. Segue o algoritmo do método abaixo:

```

def Gausspivot(A, b):
  for i in range(len(A)):
    pivo = math.fabs(A[i][i])
    linhaPivo = i
    for j in range(i+1, len(A)):
      if math.fabs(A[j][i]) > pivo:
        pivo = math.fabs(A[j][i])
        linhaPivo = j

    if linhaPivo != i:
      linhaAuxiliar = A[i]
      A[i] = A[linhaPivo]
      A[linhaPivo] = linhaAuxiliar

      bAuxiliar = b[i]
      b[i] = b[linhaPivo]
      b[linhaPivo] = bAuxiliar

    for m in range(i + 1, len(A)):
      multiplicador = A[m][i]/A[i][i]
      for n in range(i, len(A)):
        A[m][n]-=multiplicador*A[i][n]
      b[m] -= multiplicador*b[i]

  return solucao(A, b)


```

```python
def solucao(A, b):
  vetorSolucao = []
  for i in range(len(A)):
    vetorSolucao.append(0)
  linha = len(A) - 1
  while linha >= 0:
    x = b[linha]
    coluna = len(A) - 1
    while coluna > linha:
      x -= A[linha][coluna]*vetorSolucao[coluna]
      coluna -= 1
    x /= A[linha][coluna]
    linha -= 1
    vetorSolucao[coluna] = x

    for j in range(len(vetorSolucao)):
      print(vetorSolucao[j])
```

O algoritmo foi dividido em duas funções, uma para gerar a matriz escalonada e a outra para, por fim, encontrar o vetor solução do sistema.

Segue abaixo os terminais com os vetores solução para cada caso da matriz de Hilbert.

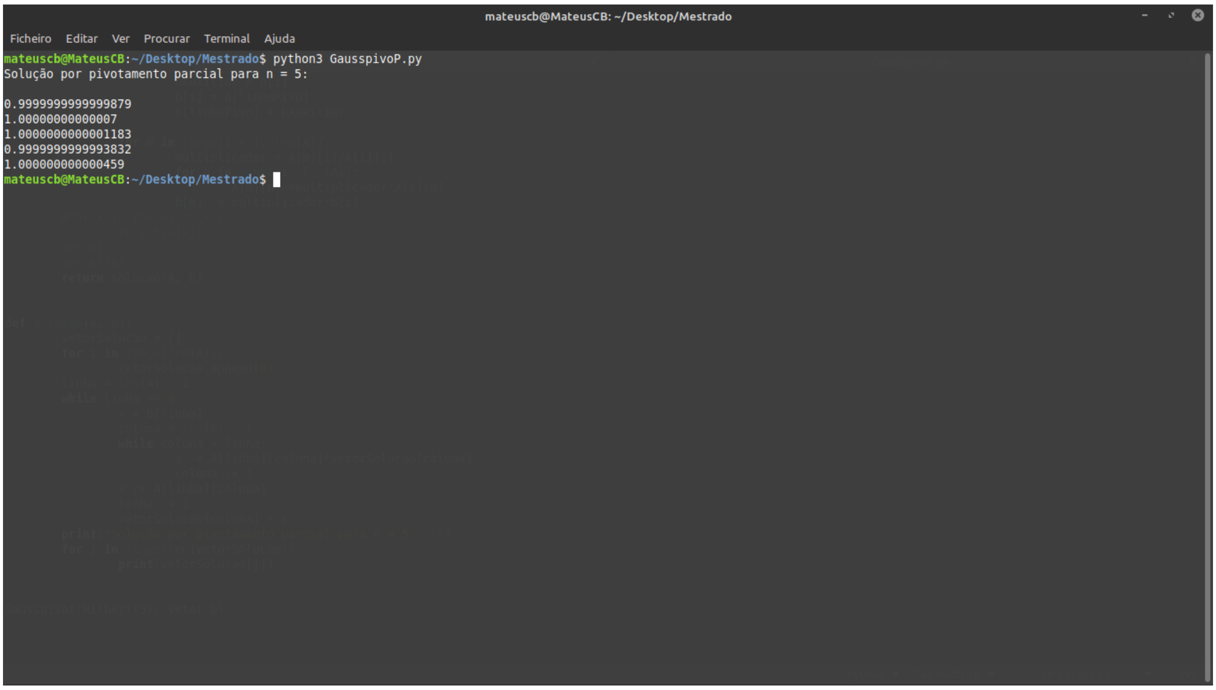

Figura 7: Terminal de execução do método de Gauss por pivotamento parcial para $n = 5$

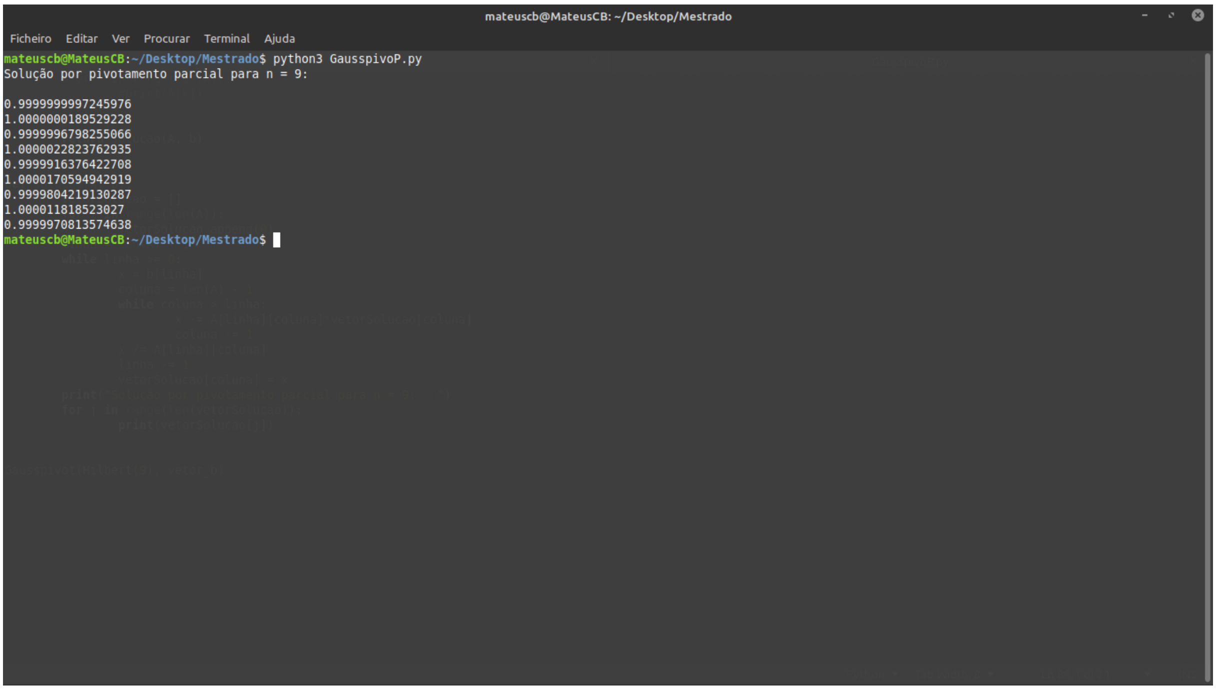

Figura 8: Terminal de execução do método de Gauss por pivotamento parcial para $n = 9$

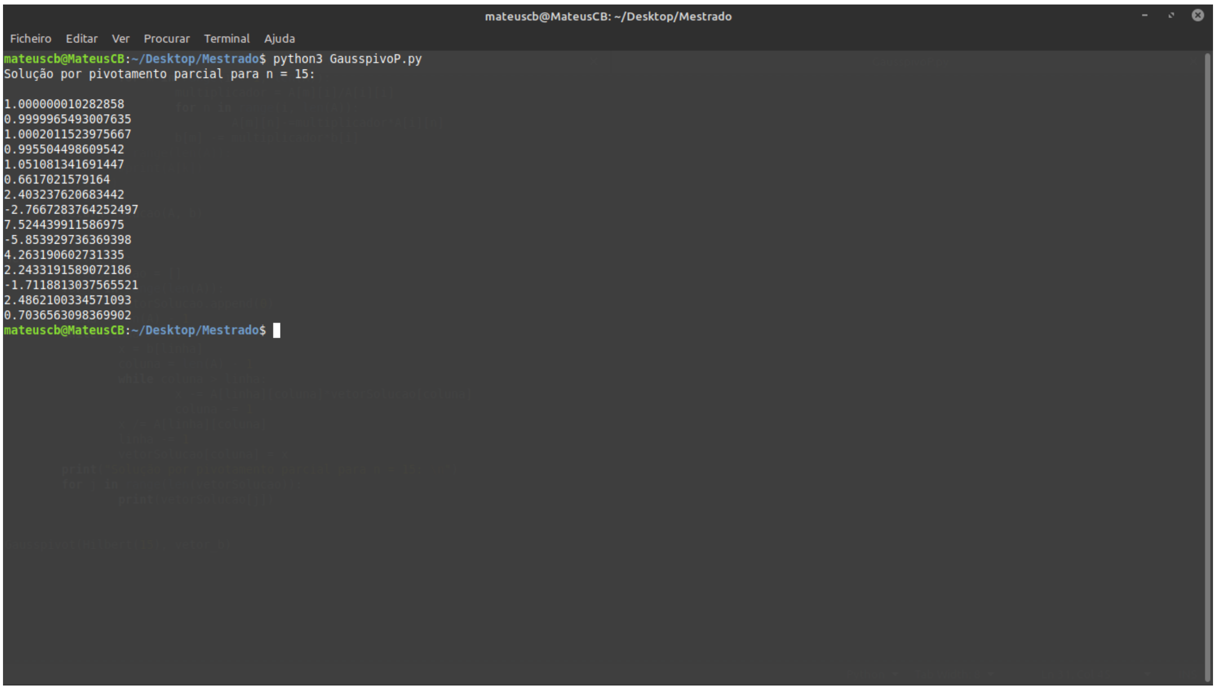

Figura 9: Terminal de execução do método de Gauss por pivotamento parcial para $n = 15$

3.3 métodos diretos: Eliminação de Gauss com pivoteamento parcial com escala

O método de eliminação de Gauss com pivotamento parcial com escala é uma estratégia a de pivotamento coloca na posição do pivô o elemento em módulo que é o maior em relação aos elementos de sua linha. O efeito desta mudança de escala é garantir que o maior elemento em cada linha tenha módulo relativo 1 antes que a comparação para troca de linhas seja efetuada.

Segue abaixo o algoritmo da implementação deste método:

```
def Trocalinha(v, i, j):
  arr = np.array(v)
  if len(arr.shape) == 1:
    arr[i], arr[j] = arr[j], arr[i]
  else:
    arr[[i,j], :] = arr[[j,i], :]

def GausspivotE(A, b):
  n = len(b)
  s = np.zeros(n)

  for i in range(n):
    s[i] = max(np.abs(a[i,:]))
  for k in range(0, n-1):
    p = np.argmax(np.abs(A[k:n,k])/s[k:n]) + k
    if p != k:
      Trocalinha(b, k, p)
      Trocalinha(s, k, p)
      Trocalinha(A, k, p)
    for i in range(k+1, n):
      if A[i, k] != 0.0:
        lambda = A[i, k] / A[k, k]
        A[i, k+1: n] = A[i, k+1:n] - lambda*A[k, k+1:n]
        b[i] = b[i] - lam*b[k]

  b[n-1] = b[n-1] / A[n-1, n-1]
  for k in range(n-2, -1, -1):
    b[k] = (b[k] - np.dot(A[k,k+1:n], b[k+1:n])) / A[k, k]
```

```python
    return b
```

Primeiramente, foi gerada uma função que faz as trocas das linhas, essa função é chamada de Trocalinha. Em seguida, essa função é chamada dentro do método GausspivotE, que é onde efetivamente ocorre o pivotamento com escala. O último bloco do código retorna calcula e retorna o vetor b, que no caso é o nosso vetor solução.

As saídas geradas por esse método são iguais as geradas pelo método do pivotamento parcial.

Os vetores solução que foram obtidos após a execução do algoritmo acima seguem indicados nos teminais abaixo:

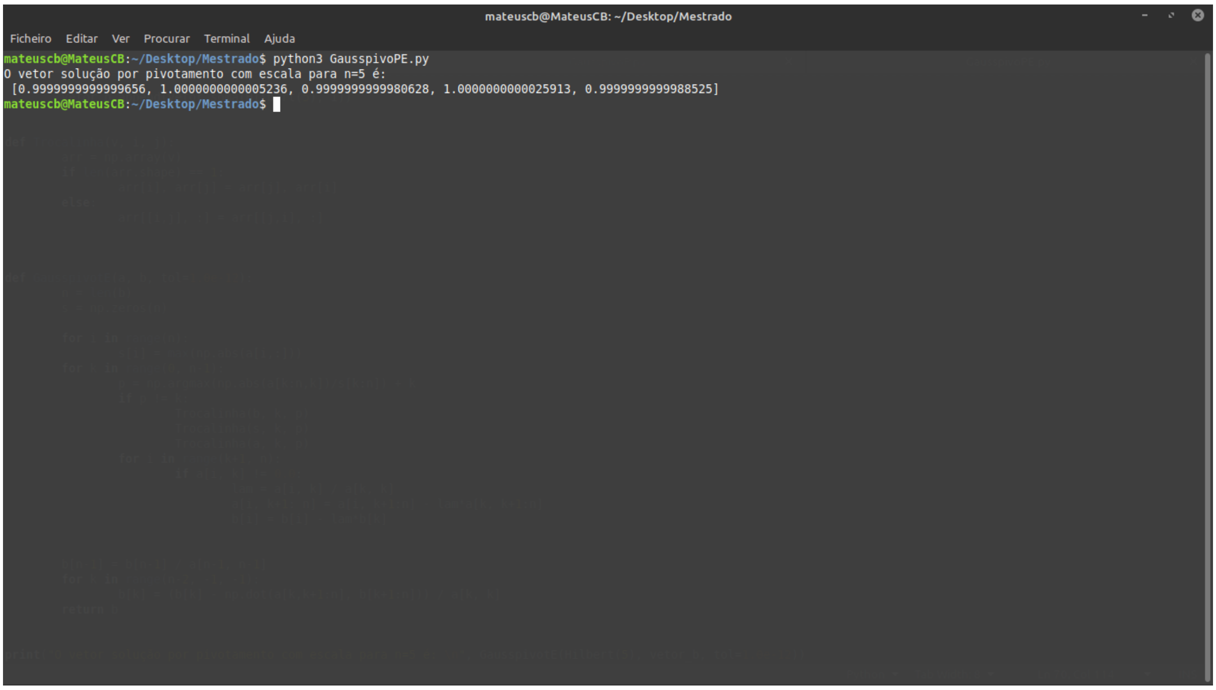

Figura 10: Terminal de execução do método de Gauss por pivotamento parcial com escala para $n = 5$

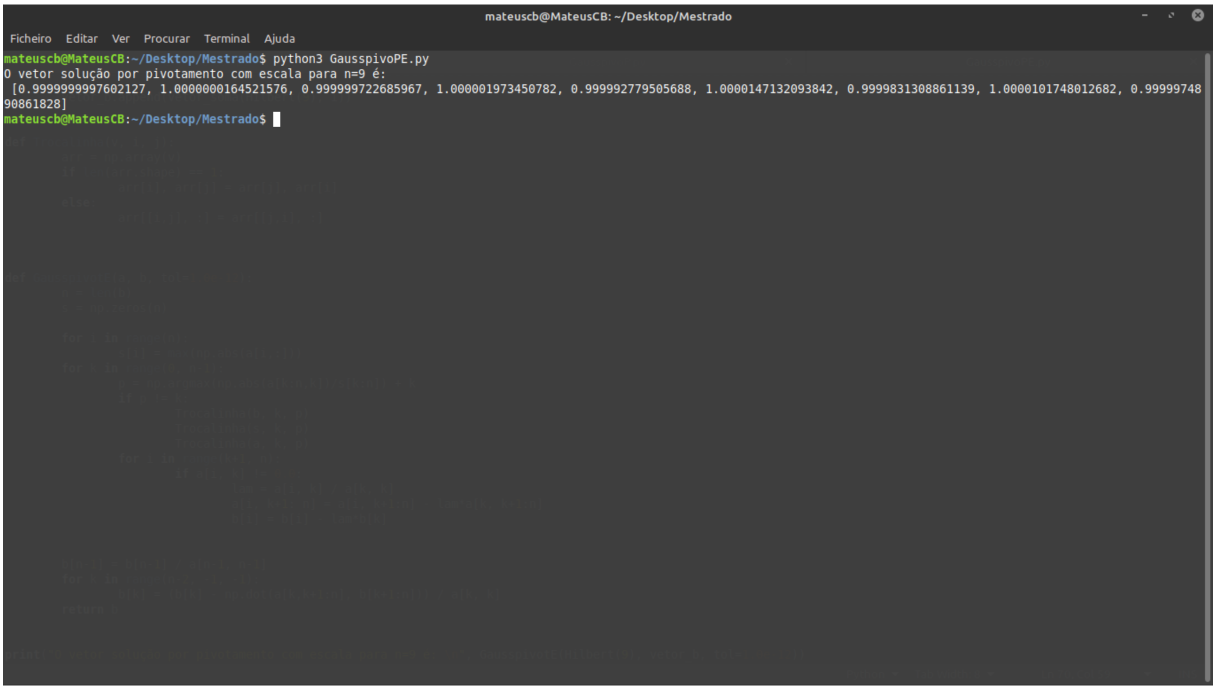

Figura 11: Terminal de execução do método de Gauss por pivotamento parcial com escala para $n = 9$

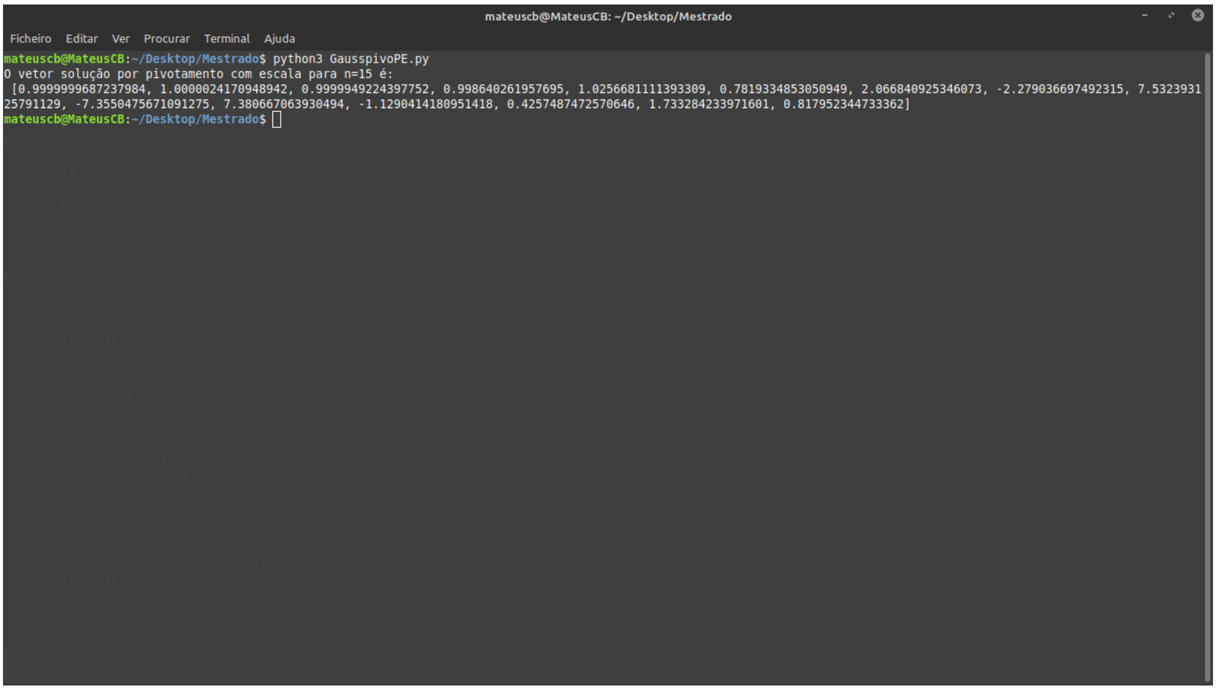

Figura 12: Terminal de execução do método de Gauss por pivotamento parcial com escala para $n = 15$

3.4 Métodos iterativos: Eliminação de Gauss-Jacobi

O método iterativo de Gauss-Jacobi tem como objetivo transformar um sistema linear que possui a forma $Ax = b$ em $x = Cx + g$ e para este método (asism como para outros métodos iterativos) os pivôs de cada coluna, ou seja, os elementos das diagonais principais devem ser diferentes de zero. Podemos dizer que por se tratar de um método iterativo, ele possui a ideia geral de generalizar o método do ponto fixo, que é utilizado para encontrar as raízes de equações, técnica que foi vista nos temas anteriores.

A equação que nos dá os elementos do vetor solução deste método é dada por:

$$x_i{}^{(k+1)} = \frac{1}{a_{ii}} \left[- \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k)} + b_i \right] \tag{4}$$

Este método é muito parecido com o método de Gauss-Seidel porém, em Gauss-Jacobi, temos uma linha de raciocínio um pouco diferente, não são "usados"os valores de x de forma imediata, mas calcula-se um conjunto de novos valores para x com base em um conjunto de $x's$ anteriores. Segue a implementação do método no algoritmo abaixo:

```python
def jacobi(tol, erro, k_max):
  n = len(Hilbert()) # entre parenteses entra o valor de n da matriz
  x = np.zeros(n)
  x_1 = np.copy(x)
  k = 1
  vec_erro = [0]


  while k < k_max and erro > tol:

        for i in range(n):
            soma = 0
            for j in range(n):
                  if i != j:
                      soma += (Hilbert()[i][j] * x[j])
                      x[i] = (vetor_b[i]-soma) / Hilbert()[i][i]


        erro = np.linalg.norm(x-x_1)
        vec_erro.append(erro)
        x_1 = np.copy(x)


        k += 1
  print("iteracoes: ",k)
  print("O erro associado a este metodo e: \n", vec_erro[]) # entre colchetes entra o numero de iteracoes
  print("O vetor solucao e dado por: \n ")
  for i in range(len(x_1)):
    print(x_1[i])
```

Primeiramente nós estabelecemos um valor máximo para a tolerância e um valor para ser comparado no final do algoritmo com a tolerância. Também antes da função do método é estabelecido o numero de iterações. Já dentro da função, no comando While, nós calculamos o somatório da equação de Gauss-Jacobi e na linha abaixo, a equação de Gauss-Jacobi é calcudada fazendo uso dessa soma.

E por fim, temos o erro associado a cada iteração. Os terminais com os vetores solução e os erros a cada iteração seguem abaixo:

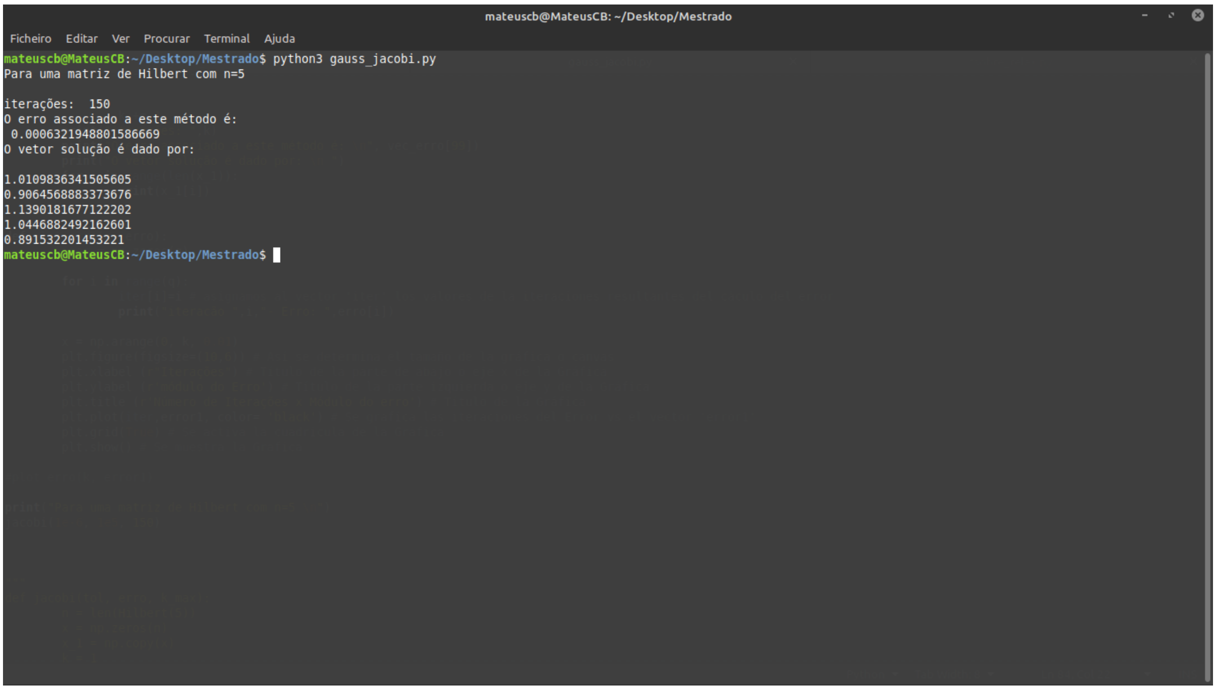

Figura 13: Terminal de execução do método de Gauss-jacobi para $n = 5$

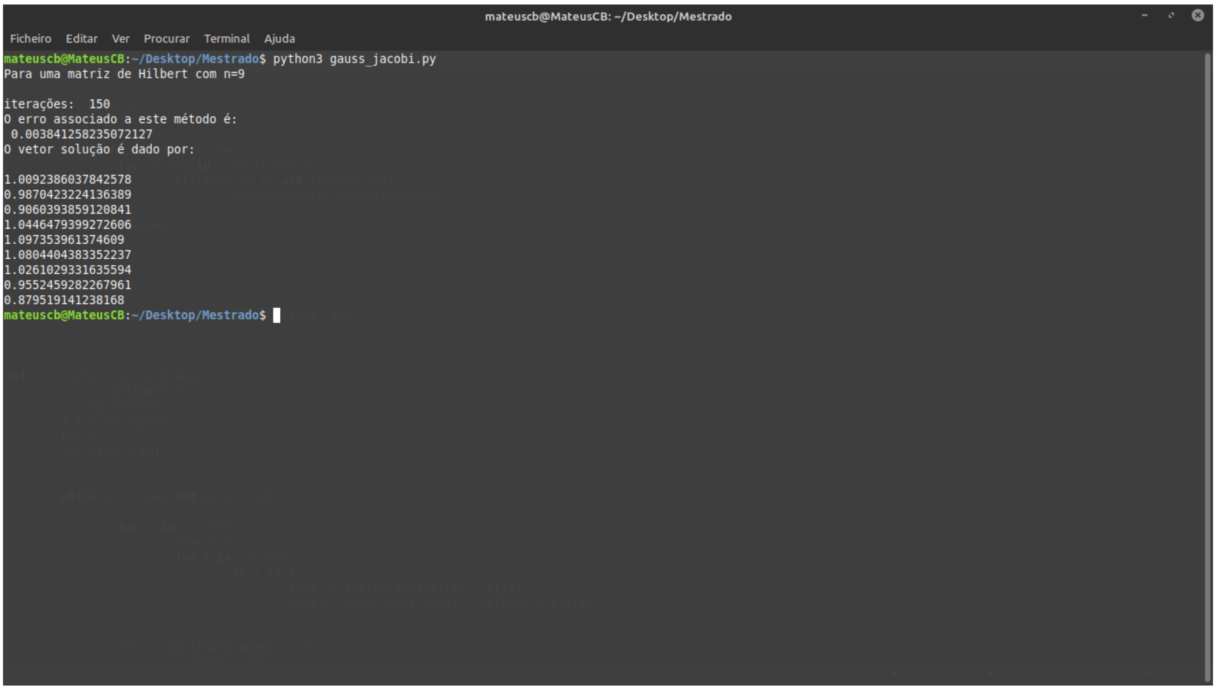

Figura 14: Terminal de execução do método de Gauss-jacobi para $n = 9$

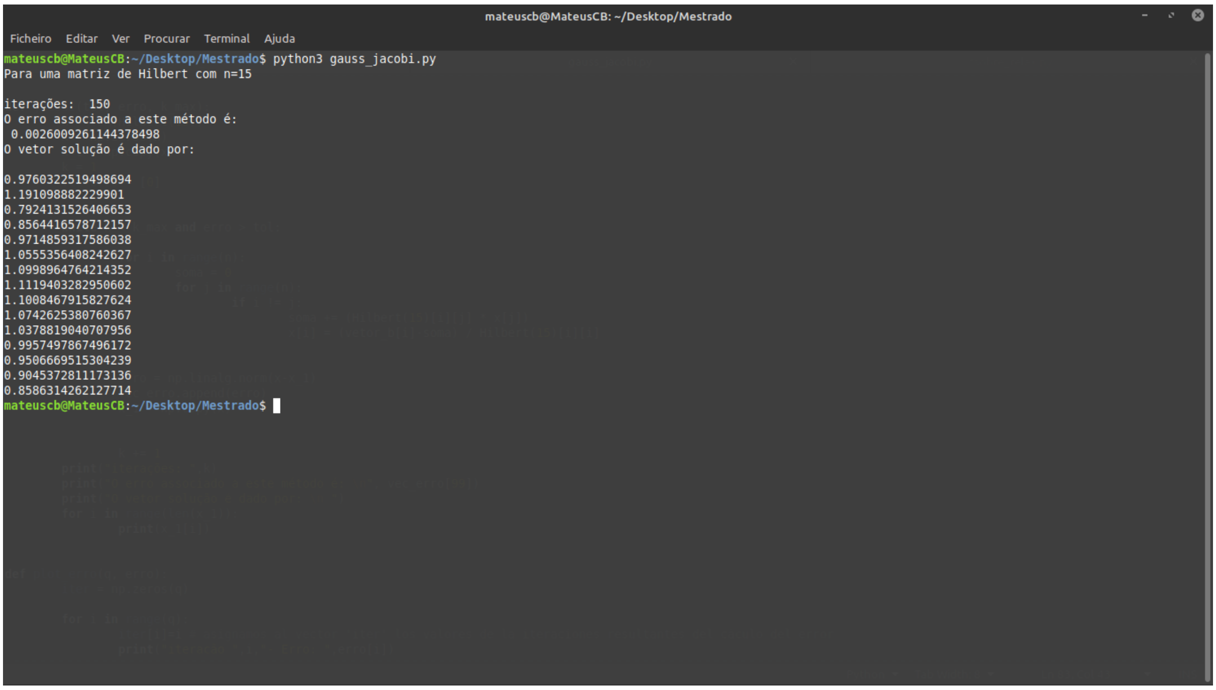

Figura 15: Terminal de execução do método de Gauss-jacobi para $n = 15$

Para os métodos iterativos, podemos gerar também gráficos que relacionam o módulo dos valores dos erros obtidos com o número de iterações, assim podemos ver melhor o comportamento do algoritmo ao longo de sua execução. Para gerar o gráfico, foi feito o seguinte código:

```python

def plot_erro(q, erro):
  n = np.zeros(q)

  for i in range(q):
        n[i]=i

  x = np.arange(0, q, 0.01)
  plt.figure(figsize=(10,6))
  plt.xlabel (r"Iteracoes")
  plt.ylabel (r'modulo do Erro')
  plt.title (r'Numero de Iteracoes x Modulo do erro ()')
  plt.plot(n, erro, color= 'red')
  plt.grid(True)
  plt.show()
```

A função para a construção do gráfico recebe como parâmetro o numero de iterações

(q) e o vetor com os valores dos erros. Os Gráficos para $n = 5$, $n = 9$ e $n = 15$ seguem abaixo:

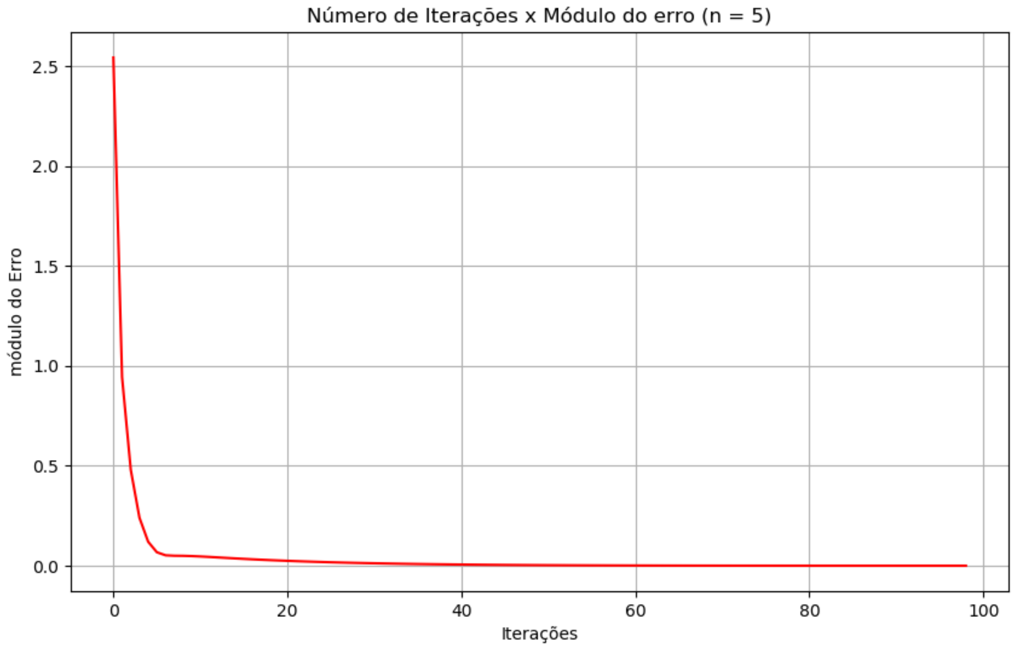

Figura 16: Gráfico erro vs iterações de Gauss-jacobi para $n = 5$

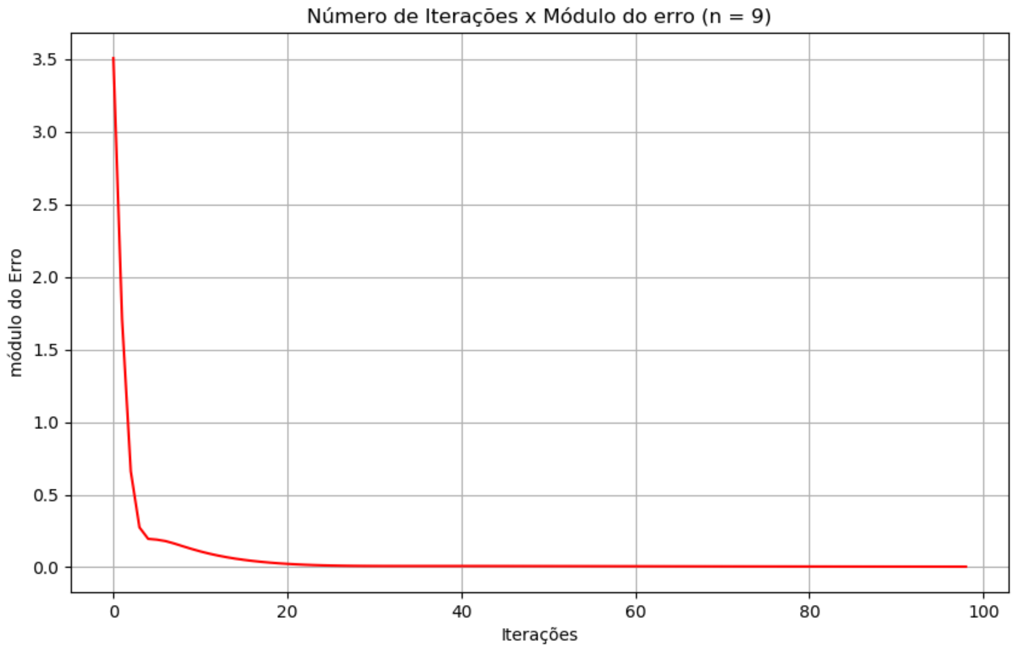

Figura 17: Gráfico erro vs iterações de Gauss-jacobi para $n = 9$

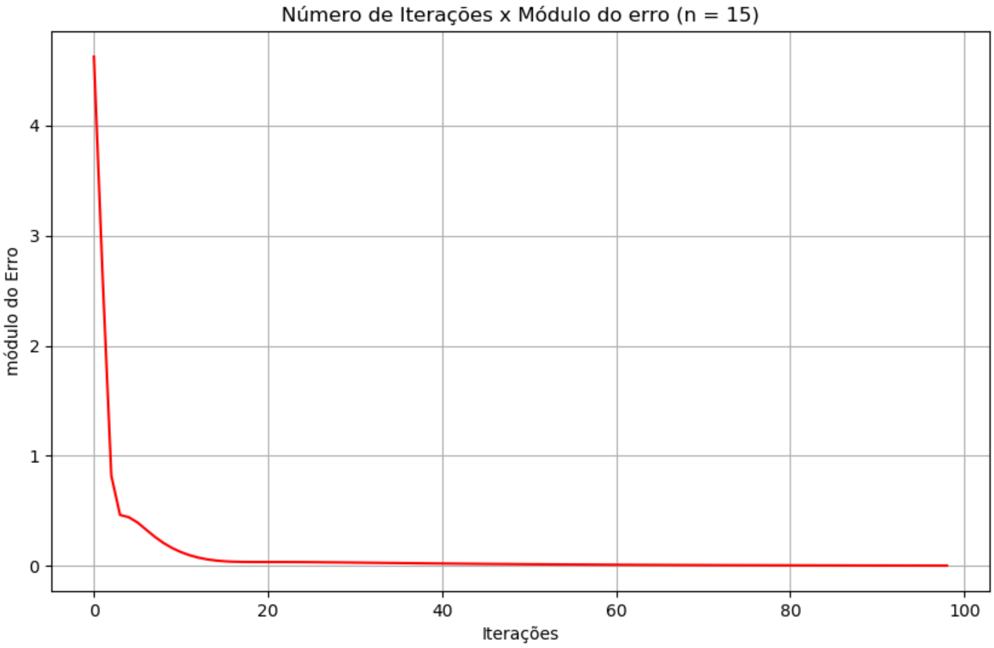

Figura 18: Gráfico erro vs iterações de Gauss-jacobi para $n = 15$

3.5 Eliminação de Gauss-Seidel

Este método é bastante semelhante ao anterior, como foi dito a diferença principal entre o método de Gauss-jacobi e o método de Gauss-Seidel é que o valor de x é calculado e logo substituído na equação de Gauss-Seidel para se encontrar o vetor solução. A equação para se encontrar a solução segundo este método é dada por:

$$x_i{}^{(k)} = \frac{1}{a_{ii}}\left[-\sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k-1)} + b_i\right] \tag{5}$$

O algoritmo implementado segue abaixo:

```
def seidel(tol, erro, k_max):
  n = len(Hilbert()) # entre parenteses entra o valor de n da matriz
  x = np.zeros(n)
  x_1 = np.copy(x)
  k = 1
  vec_erro = [0]


```

```
9  while k < k_max and erro > tol:
10
11     for i in range(n):
12         soma = 0
13         for j in range(n):
14             if i != j:
15                 soma += (Hilbert()[i][j] * x[j])
16          x[i] = (vetor_b[i]-soma) / Hilbert(5)[i][i]
17
18
19     erro = np.linalg.norm(x-x_1)
20     vec_erro.append(erro)
21     x_1 = np.copy(x)
22
23
24     k += 1
25 print("iteracoes: ",k)
26 print("O erro associado a este metodo e: \n", vec_erro[]) # entre colchetes entra o numero de iteracoes
27 print("O vetor solucao e dado por: \n ")
28 for i in range(len(x_1)):
29   print(x_1[i])
```

Podemos notar que, a diferença, em se tratando de código, deste método para o método anterior está na linha 16, onde podemos notar que o cálculo do vetor x[i] está fora do segundo for (que segue em j).

Os terminais com os resultados desse método seguem abaixo:

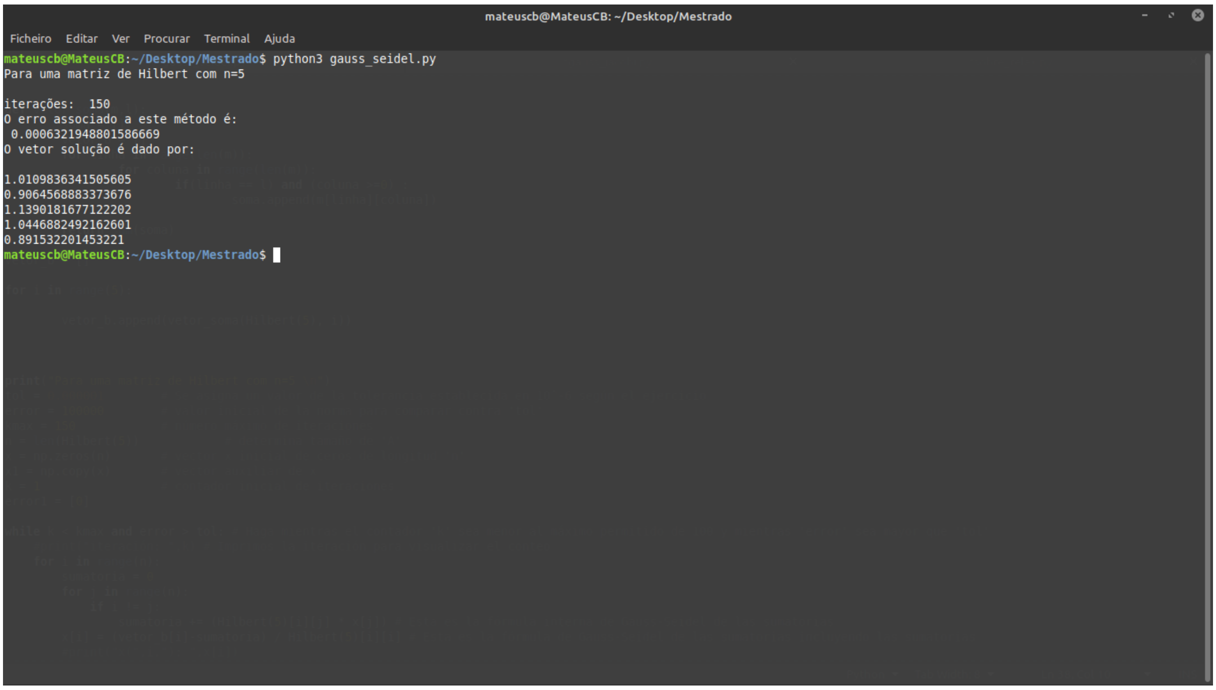

Figura 19: Terminal de execução do método de Gauss-seidel para $n = 5$

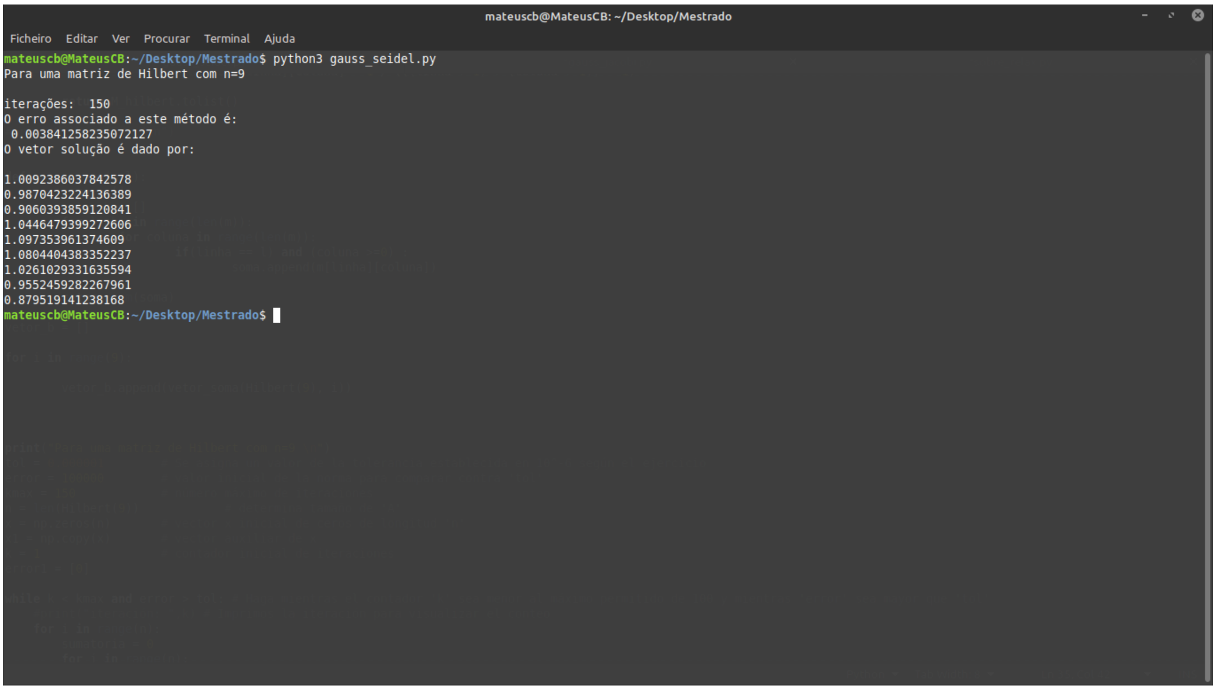

Figura 20: Terminal de execução do método de Gauss-seidel para $n = 9$

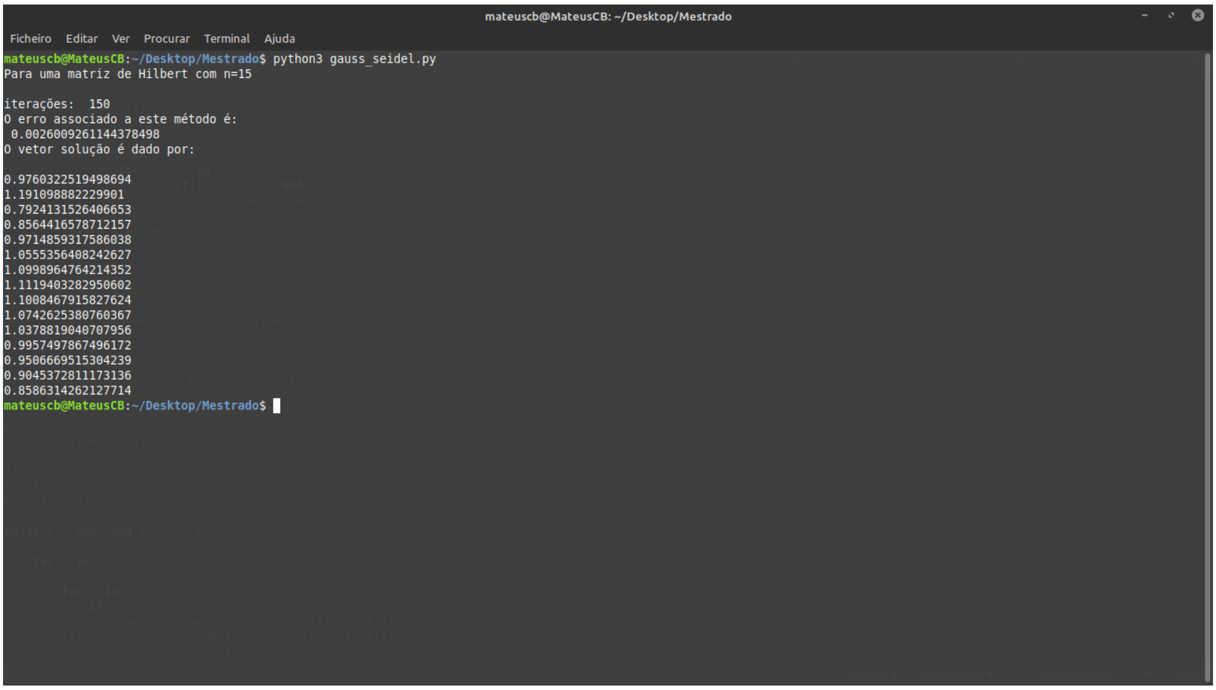

Figura 21: Terminal de execução do método de Gauss-seidel para $n = 15$

Também foram gerados os gráficos de erro por número de iterações utilizando o mesmo código mostrado anteriormente, então segue abaixo os gráficos para o método de Gauss-seidel.

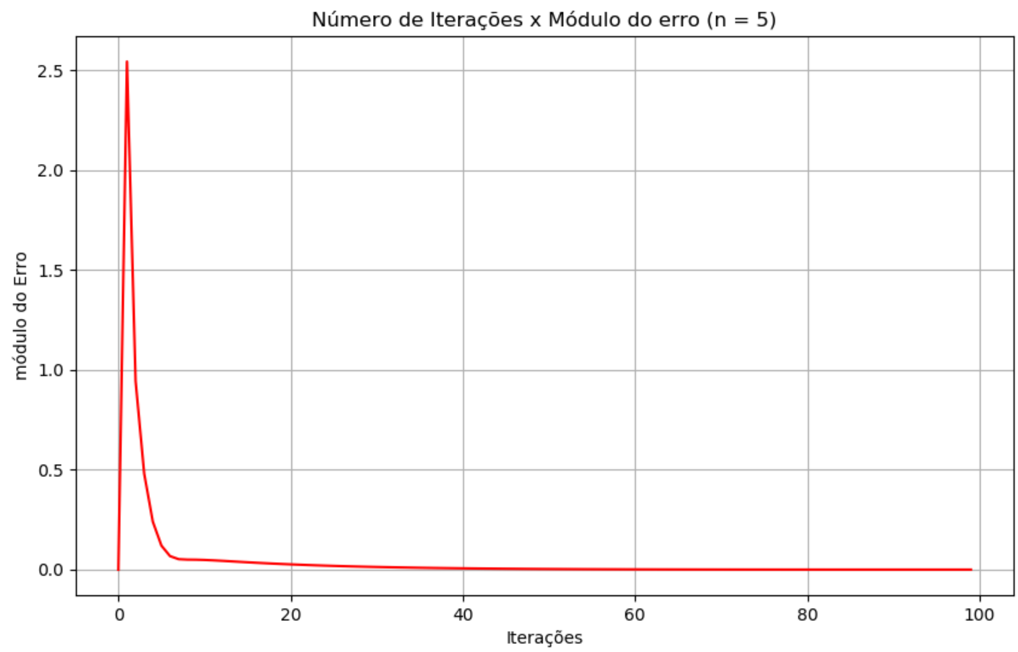

Figura 22: Gráfico erro vs iterações de Gauss-seidel para $n = 5$

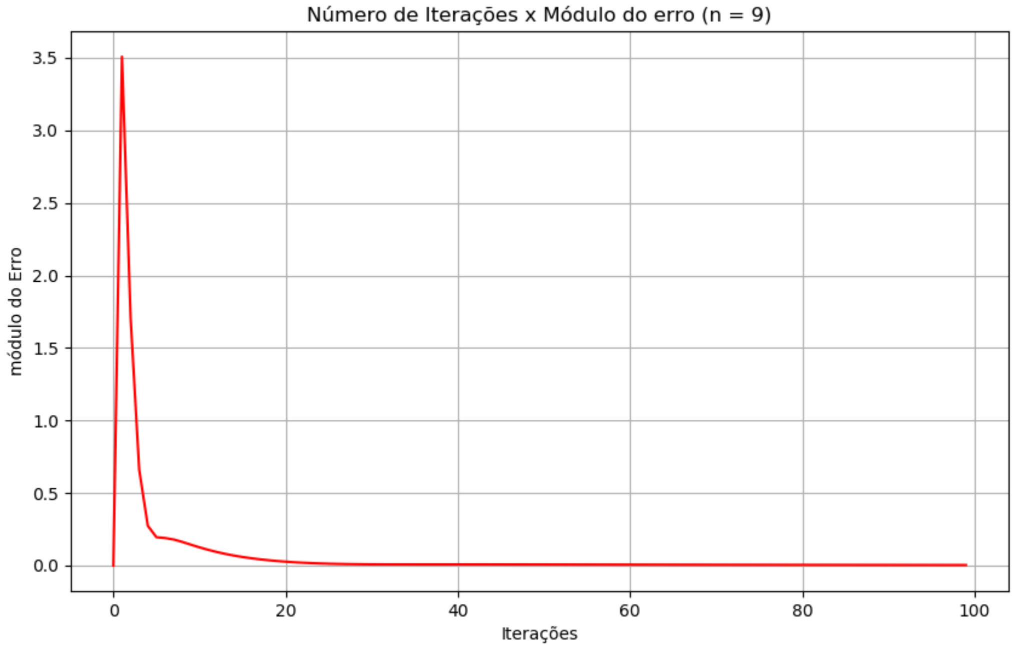

Figura 23: Gráfico erro vs iterações de Gauss-seidel para $n = 9$

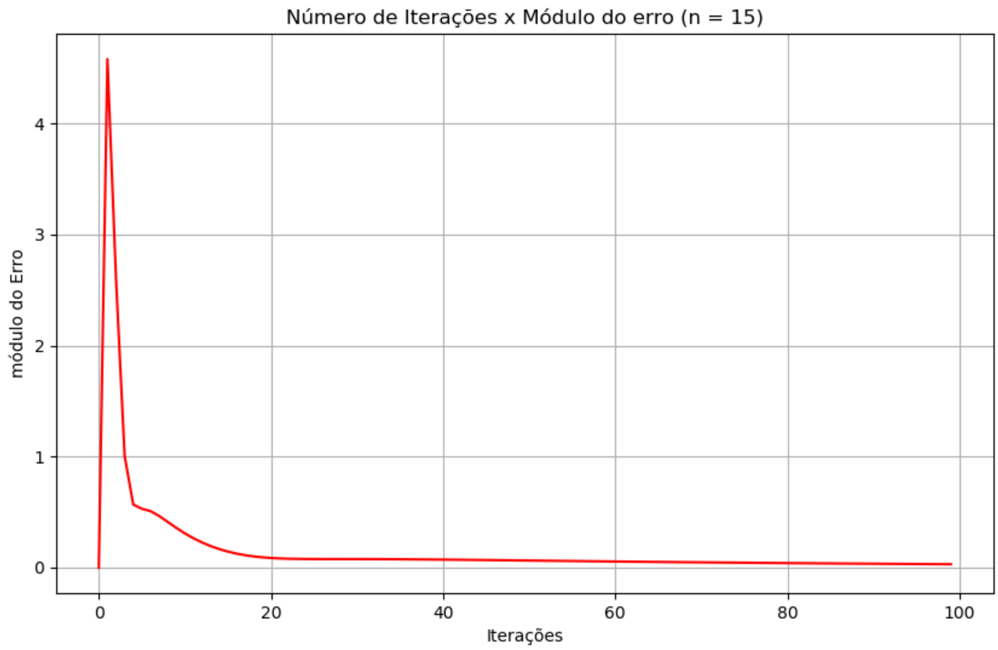

Figura 24: Gráfico erro vs iterações de Gauss-seidel para $n = 15$

3.6 Métodos iterativos: Sobre-relaxamento

O método sobre-relaxamento é calculado seguindo a seguinte equação:

$$x_i^{(k)} = \lambda \left[\frac{1}{a_{ii}} \left(-\sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k-1)} + b_i \right) \right] + (1-\lambda) x_i^{k-1} \tag{6}$$

Para cada valor de x, este valor é modificado através de uma média ponderada utilizando o resultado das iterações anteriores da seguinte forma:

$$\begin{aligned} x_i^{atual} &= \lambda x_i^{atual} + (1-\lambda) x_i^{ant} \\ x_i^k &= \lambda x_i^k + (1-\lambda) x_i^{k-1} \end{aligned} \tag{7}$$

Disso temos que se λ for um valor entre 0 e 1, o resultado final será uma média ponderada entre os resultados anteriores e os atuais e é justamente essa modificação que é conhecida como sub-relaxamento. Se o valor de λ estiver entre 1 e 2, um peso maior é dado para os valores atuais neste caso, há uma superposição em que o novo valor está seguindo na direção correta em relação à solução, mas muito lentamente. Portanto, a ponderação adicional de λ tenta melhorar a aproximação. Com isso, esse tipo de modificação, que é chamado de sobrerelaxamento, permite acelerar a convergência de um sistema que já é convergente.

O algoritmo para este método segue abaixo:

```
def relax(tol, erro, k_max):
  n = len(Hilbert()) # entre parenteses entra o valor de n da matriz
  x = np.zeros(n)
  x_1 = np.copy(x)
  k = 1
  vec_erro = [0]
  lamb = 1.5


  while k < k_max and erro > tol:

        for i in range(n):
            soma = 0
            for j in range(n):
```

```
                if i != j:
                    soma += (Hilbert()[i][j] * x[j])
            x[i] = (lam*((vetor_b[i]-soma) / Hilbert()[i][i]))+(1-lam)*x[i]


        erro = np.linalg.norm(x-x_1)
        vec_erro.append(erro)
        x_1 = np.copy(x)


        k += 1
print("iteracoes: ",k)
print("O erro associado a este metodo e: \n", vec_erro[]) # entre colchetes entra o numero de iteracoes
print("O vetor solucao e dado por: \n ")
for i in range(len(x_1)):
  print(x_1[i])
```

Código também muito semelhante aos anteriores (dos métodos iterativos), seguindo apenas a mudança na linha onde agora aos cálculos é adicionado o termo 'lam', que para a matriz de Hilbert usamos como sendo 1.5. Segue abaixo os terminais com os resultados da implementação.

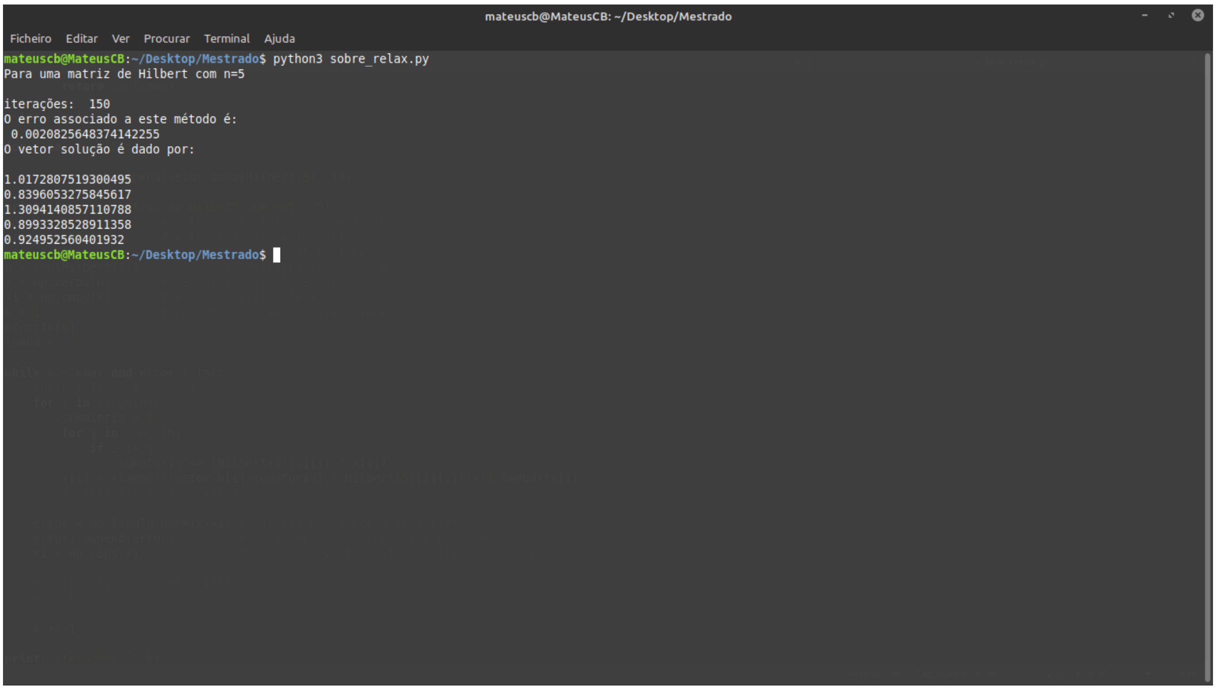

Figura 25: Terminal de execução do método do sobre-relaxamento para $n = 5$

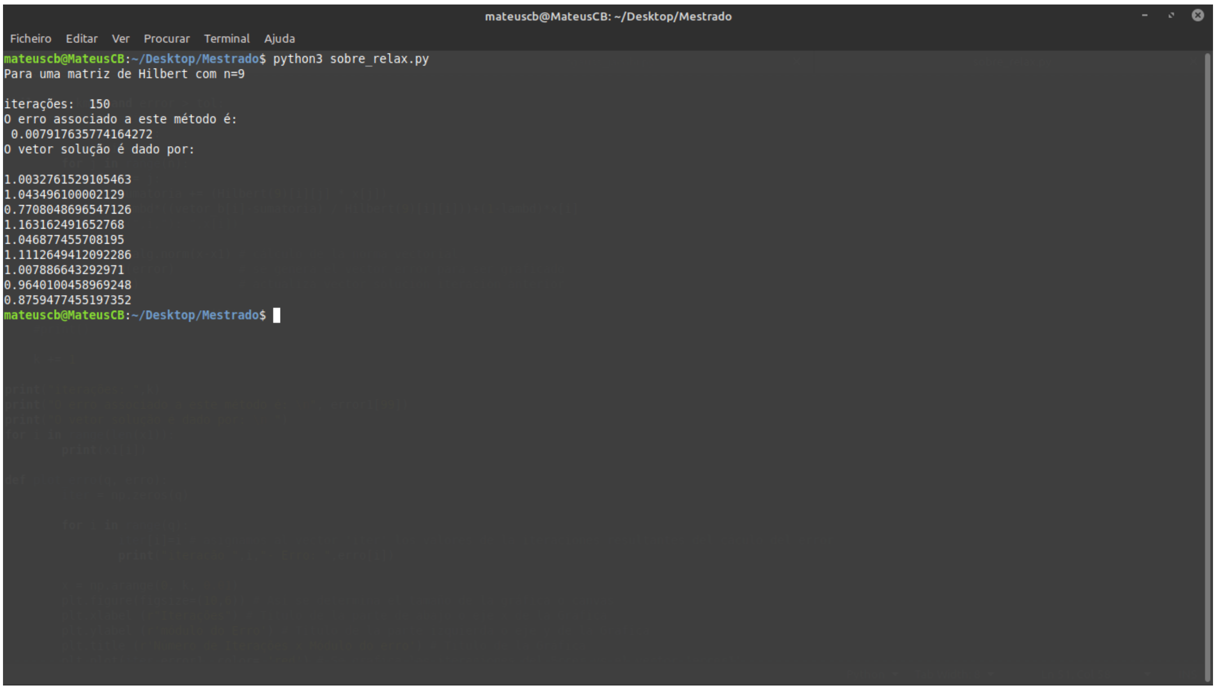

Figura 26: Terminal de execução do método do sobre-relaxamento para $n = 9$

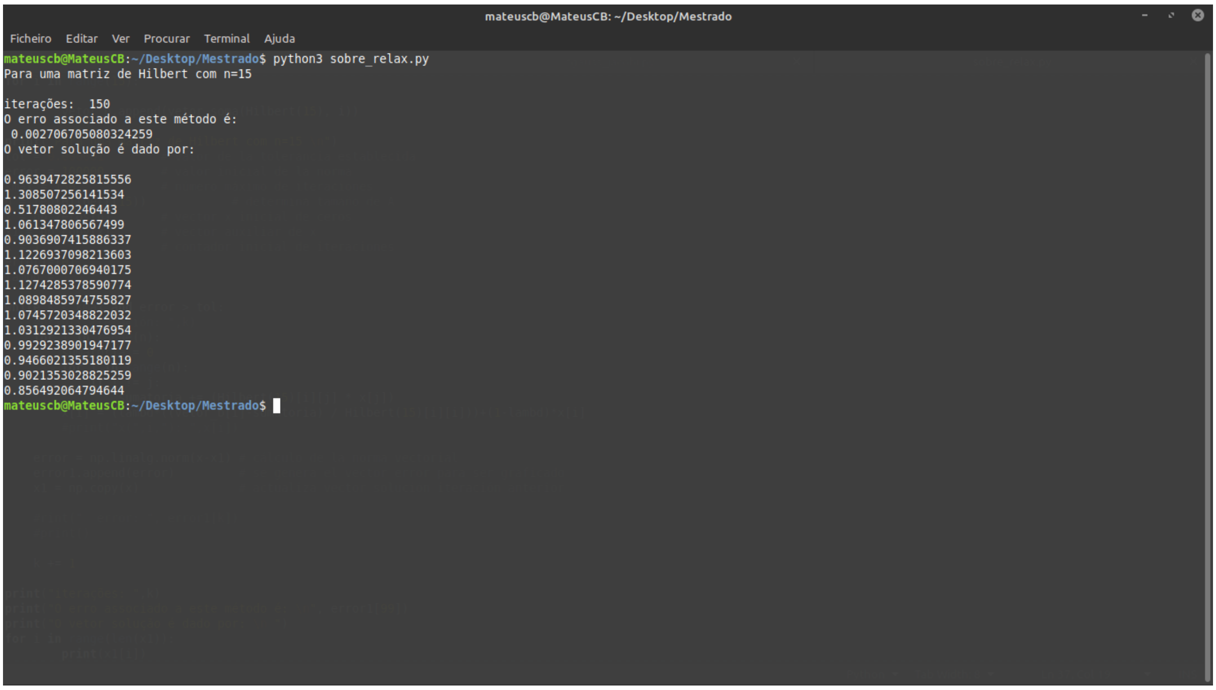

Figura 27: Terminal de execução do método do sobre-relaxamento para $n = 15$

E como já está sendo de costume para os métodos iterativos, segue também agora os gráficos de erro x iterações para o método do sobre relaxamento:

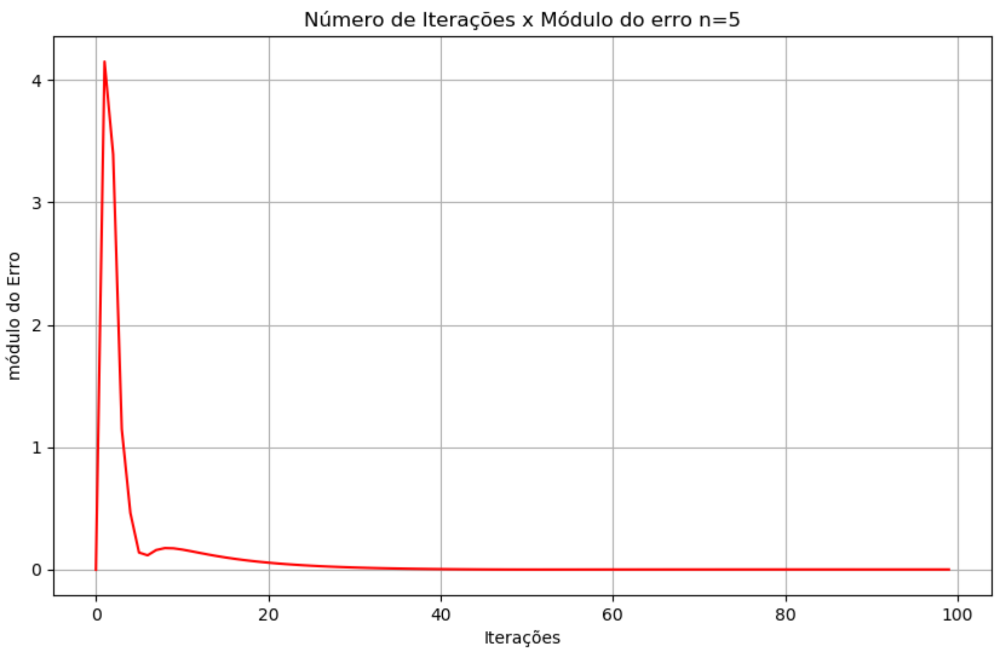

Figura 28: Gráfico erro vs iterações de Sobre-relaxamento para $n = 5$

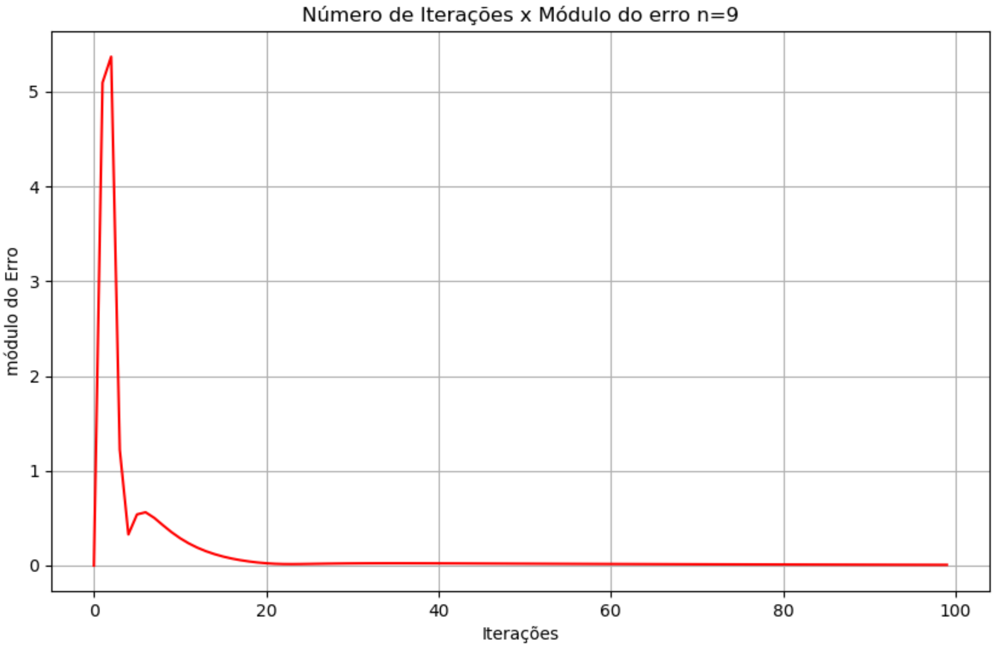

Figura 29: Gráfico erro vs iterações de Sobre-relaxamento para $n = 9$

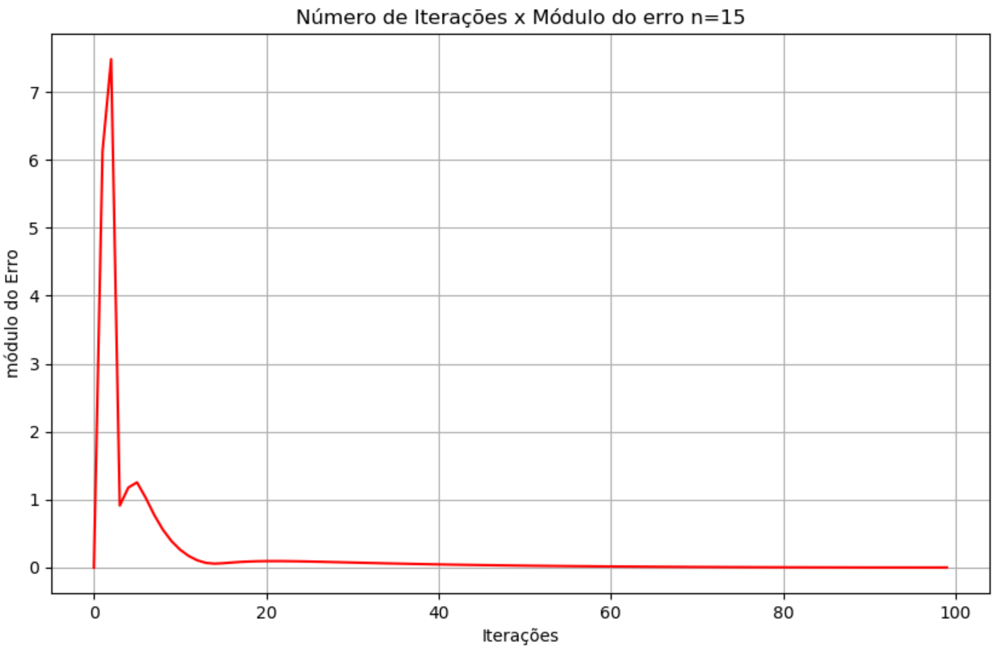

Figura 30: Gráfico erro vs iterações de Sobre-relaxamento para $n = 15$

4 Problema 2: Decomposição LU

Para se resolver um sistema de equações lineares do tipo $Ax = b$, podemos fazer a fatoração da matriz A como o produto de uma matriz L triangular inferior e uma matriz U triangular superior, ou seja, teremos então $A = LU$. Portanto, o sistema pode ser escrito da segiunte forma:

$$\begin{aligned} Ax &= b \\ (LU)x &= b \\ L(Ux) &= b \end{aligned} \tag{8}$$

Com isso, nós vamos ter que: $Ly = b$ e $Ux = y$. Isto significa que, ao invés de resolvermos o sistema original, devemos resolver o sistema triangular inferior $Ly = b$ e o sistema triangular superior $Ux = y$, o qual nos fornece a solução de $Ax = b$.

Para este problema, devemos aplicar esse método para encontrar a inversa de uma matriz. Para tal, nós temos que realizar o escalonamento de uma matriz A tal que $[A][I]$ onde I é a matriz identidade. Após fazer esse escalonamento o bloco direito irá conter a matriz A^{-1}. Podemos dizer também que, realizar este processo é o mesmo que resolver n sistemas com a mesma matriz A e os vetores da base canônica $e_i = [0, ..., 0, 1, ..., 0]^T$ tal que:

$$Ax_i = e_i, \qquad i = 1 : n \tag{9}$$

As matrizes que iremos usar para implementar o método afim de se obter a inversa das mesmas são:

$$A = \begin{pmatrix} 10 & 2 & -1 \\ -3 & -6 & 2 \\ 1 & 1 & 5 \end{pmatrix} \tag{10}$$

$$B = \begin{pmatrix} 1 & 4 & 9 & 16 \\ 4 & 9 & 16 & 25 \\ 9 & 16 & 25 & 36 \\ 16 & 25 & 36 & 49 \end{pmatrix} \tag{11}$$

Conhecendo o problema, agora devemos implementar o método, o algoritmo da decomposição LU segue abaixo:

```python

def subs_sucess(A, b):

  n = len(A)

  x = n * [0]
  for i in range(0, n):
    S = 0
    for j in range(0, i):
      S = S + A[i][j] * x[j]
    x[i] = (b[i] - S) / A[i][i]

  return x


def iden(n):
  m = []
  for i in range(0, n):
    linha = [0] * n
    linha[i] = 1
    m.append(linha)
  return m

def lu(A):
  n = len(A)

  L = iden(n)

  for k in range(0, n-1):
    for i in range(k+1, n):
      m = -A[i][k] / A[k][k]
```

```
      L[i][k] = -m
      for j in range(k+1, n):
        A[i][j] = m*A[k][j] + A[i][j]
      A[i][k] = 0

  return (L, A)

def subs_retro(A, b):
  n = len(A)
  x = n * [0]

  for i in range(n-1, -1, -1):
    S = 0
    for j in range(i+1, n):
      S = S + A[i][j] * x[j]
    x[i] = (b[i] - S) / A[i][i]

  return x

def lux(L, U, b):

  y = subs_sucess(L, b)
  x = subs_retro(U, y)

  return x
```

A primeira função faz a substituições sucessivas, a segunda função retorna a matriz identidade. O método de fato está na função lu, pois é nela que ocorre o escalonamento.

As matrizes foram geradas manualmente, segue a implementação das mesmas abaixo:

```

A = [[10, 2, -1],
     [-3, -6, 2],
     [1, 1, 5]]

B = [[1, 4, 9, 16],
     [4, 9, 16, 25],
     [9, 16, 25, 36],
     [16, 25, 36, 49]]

```

```python
A_arr = np.array(A)
B_arr = np.array(B)
```

As duas ultimas linhas do código convertem as matrizes, que estão em forma de listas, para um formato de dados que no python se chamam nparray, isso porque algumas funções que foram utilizadas só funcionam para dados que estão neste formato.

Para critério de comparação, também foi calculada a função inversa das matrizes utilizando funções prontas da linguagem Python, essas funções estão presentes na biblioteca numpy, o produto das duas matrizes também foi calculado utilizando uma função da biblioteca numpy. Abaixo seguem os comandos que fazem estas operações:

```python
import numpy as np

np.linalg.inv(A_arr) #Comando que retorna a inversa da matriz A.

A_arr.dot(A_i) #Comando que faz o produto entre a matriz A e a sua
    inversa.
```

Começando a análise para a matriz A, os resultados de saída dos métodos está no terminal abaixo:

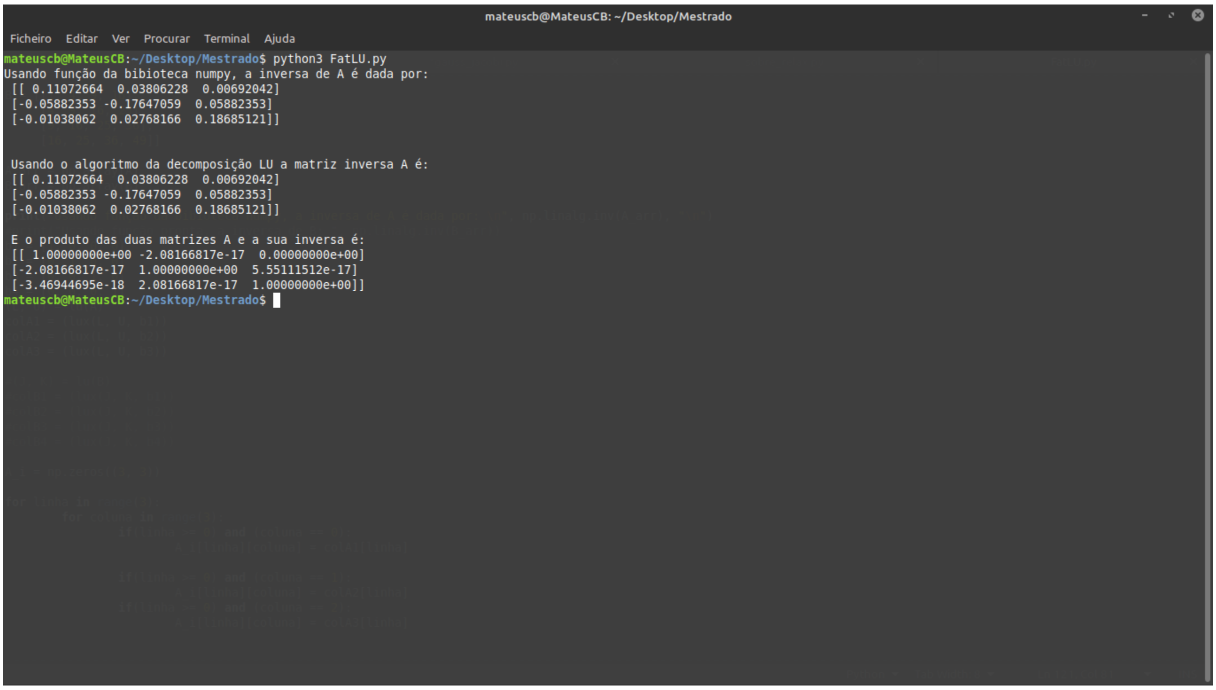

Figura 31: Matriz inversa de A e produto de A com A^{-1}

A matriz B não possui inversa, já que o determinante dessa matriz é muito próximo de zero. A biblioteca numpy também possui uma função para se calcular a determinante de uma matriz e este cálculo é feito da seguinte forma:

```python
import numpy as np

print("O determinante da matriz B eh: \n", np.linalg.det(B))
```

O resultado deste determinate é: 7.347657835701079^{-15}

5 Conclusões

5.1 Problema 1

O que podemos constatar é que para os métodos diretos, quanto maior é o n da matriz de Hilbert mais os valores do vetor solução se afastam do valor real da solução, embora os meus resultados também não foram exatos mesmo nos métodos diretos. Estes métodos possuem uma maior "facilidade"em sua implementação, mas não muita, pois um dos métodos eu não consegui implementar. Em relação a velocidade de execução ela se mostrou rápida, o que já era esperado.

Já para os métodos iterativos, podemos ver pelos gráficos que quanto maior é a matriz de Hilbert, temos um maior valor de erro nas iterações inicias , porém temos uma suavização conforme vão se aumentando o número de iterações. Esses métodos apresentaram uma maior dificuldade na implementação e a sua convergencia também se mostrou um pouco mais lenta, já que para as matrizes de ordem 9 e 15, tivemos mais calculos a serem computados.

Com isso, o resultado desse trabalho sugerem que, para uma analise númerica de sistemas de equações lineares o mais indicado são os métodos iterativos.

5.2 Problema 2

Para a matriz A, o método foi de uma eficácia satisfatória encontrou a matriz inversa de forma quase que exata ao que foi calculada usando uma função pronta do python, como também o produto da matriz A com a sua inversa gerou a matriz identidade, já que temos a diagonal principal com os valores 1 e os demais elementos muito próximos de zero como podemos ver nos terminais acima.

Já para a matriz B não foi possível encontrar uma matriz inversa, já que o determinante dessa matriz é um valor muito próximo de zero, então ao fazer a decomposição o algoritmo encontra problemas e retorna uma matriz inversa completamente sem sentido e o seu produto com a matriz original B não resulta na identidade.