Projeto Computacional

Método Simplex implementado em Python

Gabriel Cunha Marchetti
g251055@dac.unicamp.br
251055

Vécio Alves Packer
v251681@dac.unicamp.br
251681

Pedro Francisco Godoy Bernardinelli
p251570@dac.unicamp.br
251570

Matheus Queiroz Mota
m251495@dac.unicamp.br
251495

Introdução

Nosso projeto tem por finalidade implementar o algoritmo Primal Simplex para resolver problemas de Programação Linear. Para facilitar a utilização do algoritmo e torná-lo mais compreensível faremos uma documentação dele, explicando o seu funcionamento, de modo a relacionar com a teoria e também explicitando as suposições feitas pelo algoritmo para seu funcionamento. Tais suposições precisam ser cumpridas para permitir que o programa funcione corretamente. Além disso, tal documentação permite qualquer pessoa que deseje melhorar o código a fazê-lo.

Com isso, explicaremos cada função feita dentro do código em Python através de blocos de texto, relacionando-as com os métodos vistos na teoria. Assim que forem abordadas todas as funções, apresentaremos o código como um todo e traremos alguns exemplos de problemas de Programação Linear, que mostram a utilidade de nosso código e que ele cobre todo o aspecto teórico abordado.

Para ter acesso ao código, que contém todas as funções apresentadas na documentação, acesse o link GitHub

Documentação

Antes de iniciar a documentação do algoritmo, é muito importante ressaltar que o algoritmo espera que o problema de programação linear (PL) a ser resolvido está na forma padrão, ou seja, espera-se que:

- O problema é de minimização da função objetivo dadas restrições para as variáveis de decisão;
- Todas as restrições são de igualdade, com quaisquer variáveis de folga já inseridas previamente pelo usuário antes da utilização do programa;
- As variáveis são não-negativas.

É extremamente importante que essas condições sejam cumpridas, pois o código utiliza-se de tais para a resolução do problema e ao decorrer do texto são realizadas diversas referências ao fato do PL em questão estar na forma padrão.

Ressaltada essa suposição feita pelo algoritmo, podemos prosseguir para a documentação das funções implementadas no código.

```
def transpose(A):
    # transposta da matriz A
    A_t = []
    A_t_linha = []
    if type(A) == "int":
        m_A = 1
```

```
7       else:
8           m_A = len(A)
9       if type(A[0]) == "int":
10          n_A = 1
11      else:
12          n_A = len(A[0])
13
14      for j in range(n_A):
15          A_t_linha = []
16          for i in range(m_A):
17              A_t_linha.append(A[i][j])
18          A_t.append(A_t_linha)
19      return A_t
```

Listing 1: Primeira função usada dentro do Python: a de transpor matrizes.

Essa função tem um objetivo simples, queremos que ela retorne a matriz transposta, ou seja, queremos que transpose$(A) := A^T$. Pela definição, queremos que para todos i e j válidos, desejamos $a_{ij} = a_{ji}$, onde i representa as linhas da matriz A, enquanto j representa as colunas da matriz A. Notemos que a condicional na linha 5, trabalha apenas o caso especial em que a entrada fornecida pelo usuário é um número inteiro. Enquanto a condicional fornecida na linha 9 serve para o caso de uma matriz coluna.

Depois de verificados esses casos especiais, nos resta apenas fazer a mudança: $a_{ij} := a_{ji}$. Para isso, rodamos dois laços do tipo *for*, um para o elemento i e outro para o elemento j.

Observações: Note que uma implementação que pode ser feita para otimizar os laços for é através da condição $i = j$, uma vez que os elementos da diagonal principal não vão se alterar de lugar.

```
1  def vectorProduct(A, B):
2      # produto do vetor linha A e do vetor coluna B, o que resulta em um n mero
3      # Note, que A vai ser uma lista normal, enquanto B vai ser uma lista de listas
4      m_A = 1
5      n_A = len(A)
6      m_B = len(B)
7      n_B = 1
8      c = 0
9      if n_A == m_B:
10         for i in range(n_A):
11             c += A[i] * B[i][0]
12     else:
13         c = "Tamanhos incompat veis"
14     return c
```

Listing 2: Função que tem por finalidade retornar o produto de matriz.

Já esse código tem por finalidade introduzir o produto entre vetores, isto é. Suponha que tenhamos dois vetores do tipo:

$$\begin{pmatrix} x_1 & x_2 & x_3 & \cdots & x_n \end{pmatrix} \cdot \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix} = x_1 \cdot y_1 + x_2 \cdot y_2 + \cdots + x_n \cdot y_n$$

Assim, veja que a condicional na linha 9 apenas verifica se os dois vetores possuem dimensão compatível. Assim como, c é uma variável auxiliar que retorna as somas parciais.

```
1  def solucaobasica(B, b):
2      # encontra a solu  o b sica da parti  o b sica atual
3      x = np.linalg.solve(B, b)
4      for j in range(len(x)):
5          if x[j] < 0:
6              return 0
7      return x
```

Listing 3: função que retorna a solução básica do nosso problema parcial de PL

Para o propósito de encontrar uma solução básica, é utilizada a função da biblioteca numpy que resolve um sistema linear. Essa função retorna a solução de um problema, de posto completo, envolvendo uma matriz e um vetor solução. Ou seja, desejamos fazer:

$$A \cdot x = b \tag{1}$$

Notemos que o laço 'for' contido na linha 4 serve justamente para verificar se a solução é básica. Lembremos que uma condição para o problema padrão de programação linear é que as variáveis sejam

todas positivas. Desse modo, esse laço faz justamente a verificação dessa condição, pois tendo um valor negativo dentro do vetor solução, teremos que a função nos retorna 0. Em outras palavras, nos retorna que não é uma solução básica. Caso contrário, estamos justamente em uma solução básica e, desse modo, nos retorna o próprio vetor solução da equação 1.

```
1 def funcaoObjetivo(X, c):
2     f = vectorProduct(X, c)
3     return f
```

Listing 4: função que retorna o valor da função objetivo do nosso problema de PL

Para o nosso problema de PL padrão, sempre associamos um problema de minimização do valor de uma função objetivo dada certas restrições para as variáveis de decisão. Logo, usando a função definida em 2 temos justamente o cálculo do valor da função objetivo para um vetor **x** contendo as variáveis de decisão. Uma vez que o problema de PL esperado pelo algoritmo está na forma padrão, então ele consiste em:

$$\textbf{Minimize } f(x) = \mathbf{c}^T \cdot \mathbf{x} \tag{2}$$

Basicamente, a função definida em 4 nos retorna justamente o valor de f(x).

```
1 def vetormultiplicador(B, c_B):
2     B_t = transpose(B)
3     Lambda = np.linalg.solve(B_t, c_B)
4     return Lambda
```

Listing 5: Função que calcula o vetor multiplicador do método Simplex.

```
1 def custosRelativos(c_N, Lambda, N):
2     # transpondo N, para ficar mais f cil de trabalhar
3     N_t = transpose(N)
4     # lista com os custos relativos
5     c_r = []
6     for i in range(len(c_N)):
7         c_r_elem = c_N[i][0] - vectorProduct(N_t[i], Lambda)
8         c_r.append(c_r_elem)
9     return c_r
```

Listing 6: Função que calcula os custos relativos do método Simplex.

Essas duas funções apresentadas sequencialmente acima tem por objetivo executar o segundo passo dentro do método primal simplex, por tal razão são apresentadas juntas:

```
1         Passo 2: Calcular os custos relativos.
2             2.1) Resolver o sistema B^T λ = C_B
3             2.2) Calcular os custos relativos:
4                 Ĉ_Nj = C_Nj − λ^T a_Nj
5             2.3) Comparar os custos relativos:
6                 Escolher a variavel, nao negativa, de menor custo
7
```

Observe, que para a primeira função, novamente, utilizamos a função do numpy que resolve um sistema linear, para encontrar o vetor multiplicador simplex λ, o qual necessita a solução do sistema $\mathbf{B^T}\lambda = \mathbf{C_B}$. Ou seja, usamos a função vetorMultiplicador em 5 para retornar esse vetor `Lambda` no argumento da função.

Já para a segunda função, é utilizada novamente a função `vectorProduct` para realizar o produto na linha 4. No final, devemos ter uma lista de custos relativos e posteriormente trabalharemos com essa lista para definir quem entra na base.

Entradas: Para a função que calcula o vetor multiplicador são recebidas como entradas a matriz B da partição básica factível e C_B, o vetor de custo da função objetivo relacionado as variáveis básicas. Já para a segunda função, as entradas consistem, basicamente, no vetor custo relacionado as variáveis não-básicas agora e a matriz não-básica N. Além disso, a saída da função anterior, Lambda, também é utilizada como entrada, relacionando as duas funções.

```
1 def direcaoSimplex(B, a_N):
2     d = np.linalg.solve(B, a_N)
3     return d
```

Listing 7: Função que calcula o vetor direção do nosso problema de PL, ou seja, o vetor que indica uma direção que melhorará a solução.

A função direçãoSimplex tem por objetivo dar procedência ao Passo 4 do método primal. Portanto, lembrando desse passo, temos que:

```
1         Passo 4) Resolver o sistema B·y = a_Nk
2
```

Assim, usamos a função `linalg.solve` do numpy para resolver esse sistema. Lembremos que ele só traz solução para problemas de posto completo que é o caso usual dos problemas de PL.

```
1  def tamanhoPasso(d, x_B):
2      # encontrando o tamanho do passo e quem sai da base. epsilon_ind guarda as posi  es
        dos elementos do vetor dire  o positivos
3      epsilon_ind = []
4      # epsilon_list guarda os epsilons poss veis
5      epsilon_list = []
6      for i in range(len(d)):
7          if d[i] > 0:
8              epsilon_ind.append(i)
9      if epsilon_ind == []:
10         # nesse caso, nenhum elemento do vetor diretor    positivo, ent o a fun  o
       retorna 2 n meros negativos
11         # para indicar que o problema    ilimitado
12         return -42, -42
13     else:
14         for j in epsilon_ind:
15             epsilon_list.append(x_B[j]/d[j])
16         epsilon = min(epsilon_list)
17         x_sai_ind = epsilon_ind[epsilon_list.index(epsilon)]
18         return epsilon, x_sai_ind
```

Listing 8: Função que lista os possíveis tamanhos de passo, assim como retorna o valor mais adequado.

Notemos que o laço da linha 6 serve justamente para circularmos por todos os elementos do passo proposto pela direção simplex em 7. Assim, para cada valor positivo, pois lembremos que se $y \leq 0$ então estamos em um problema de PL ilimitado, sem uma solução ótima finita, associamos a uma lista de candidatos para saírem da base. Notemos que esse argumento justifica a condicional proposta na linha 9, uma vez que caso a lista seja vazia, então retornamos um valor negativo apenas como indício de que o problema não tem solução ótima.

Assim, continuamos com o nosso problema adicionando o nosso ϵ normalizado, isto é:

$$\hat{\epsilon} = \frac{\hat{x_{Bl}}}{y_l} = \min\left\{\frac{\hat{x_{Bl}}}{y_l}, \text{para } y_i > 0\right\}$$

Justamente a condição $y_i > 0$ nos fez introduzir a condicional da linha 7. Na linha 17 queremos justamente descobrir o índice do vetor que deve sair da base, enquanto na linha 16 calculamos o tamanho do passo.

```
1  def verificafase1(A, b):
2      m = len(A)
3      n = len(A[0])
4      q = n - m
5      r = n - q
6      count_lines = 0
7      numbers_line = 0
8      indexes = []
9      for i in range(m):
10         # reseta o contador de n meros positivos na linha
11         numbers_line = 0
12         for j in range(r):
13             # verifica quantos n o zeros tem na linha e em quais posi  es (algo
       semelhante a uma identidade)
14             if A[i][j+q] > 0:
15                 if j+q not in indexes:
16                     numbers_line += 1
17                     indexes.append(j+q)
18         # primeira linha s  tem um n mero nas n-m colunas finais
19         if numbers_line == 1:
20             count_lines += 1
21     # se todas as linhas tem exatamente um elemento n o-nulo
22     if count_lines == r:
23         # gerando a parti  o b sica e listas para armazenar seus  ndices
24         B = []
25         N = []
26         indB = []
27         indN = []
```

```
        for i in range(m):
            B.append(A[i][q:])
            N.append(A[i][:q])
        for j in range(r):
            indB.append(j+q)
        for k in range(q):
            indN.append(k)
        # resolvendo o sistema linear para encontrar a solu  o b sica
        x = np.linalg.solve(B, b)
        # verificando se a solu  o b sica    fact vel
        for j in range(len(x)):
            if x[j] < 0:
                return 0, 0, 0, 0
        return B, N, indB, indN
    else:
        return 0, 0, 0, 0
```

Listing 9: Função que da o passo inicial para o método das duas fases, inicialmente verificamos a fase 1

O objetivo dessa função é identificar se já há uma partição básica factível nas últimas colunas da matriz A, já que dependendo da quantidade de variáveis de folga é possível que já haja tal partição presente em A, como em casos em que as restrições são do tipo $Ax \leq b$, sem haver necessidade de resolver o método das duas fases para encontrar uma partição básica factível inicial para o algoritmo primal simplex.

Para isso, essa função olha para as últimas m colunas de A e verifica se há ou não algo semelhante a uma matriz identidade. Para tal verificação são utilizadas duas variáveis count_lines e numbers_lines. Essa última conta a quantidade de elementos positivos presentes nas últimas m colunas de cada linha de A (nesse caso, o vetor b é sempre positivo e o algoritmo já verifica isso logo após a entrada do vetor b pelo usuário e, caso esse vetor não seja estritamente não-negativo, então o algoritmo já multiplica aquele elemento e a correspondente linha em A por -1), assim, essa variável muda para cada linha da matriz. Além disso, é verificado se na coluna em questão já há um valor positivo presente nas linhas verificadas anteriormente (os índices dessas colunas são armazenados na lista indexes), caso afirmativo, então esse valor positivo não é incluído no number_lines. Quando esse número for 1, então essa linha está no formato desejado e ela é adicionada no count_lines. Do contrário, essa linha não é computada e será necessário o método das duas fases. Se o valor do count_lines for igual ao número de linhas, então temos uma boa chance de termos uma partição básica factível presente no interior de A, embora seja possível que ela não seja factível, o que é verificado posteriormente. Primeiramente apresentaremos exemplos de matrizes que são identificadas ou não pelo função. Observe que estamos olhando apenas para as últimas colunas de A e, assim, estamos trabalhando com uma parte quadrada da matriz A. Nos exemplos apresentados abaixo, estaremos trabalhando com matrizes 2x2, mas os exemplos são naturalmente estendidos para dimensões maiores, além disso, as variáveis number_lines serão concatenadas com um número natural para indicar de qual linha estamos falando

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \xRightarrow{\text{deve nos retornar}} \begin{cases} \texttt{count\_lines} \Leftarrow 2 \\ \texttt{numbers\_lines1} \Leftarrow 1 \\ \texttt{numbers\_lines2} \Leftarrow 1 \end{cases}$$

Naturalmente, a identidade é identificada pela função, pois count_lines é igual ao número de linhas (m).

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \xRightarrow{\text{deve nos retornar}} \begin{cases} \texttt{count\_lines} \Leftarrow 2 \\ \texttt{numbers\_lines1} \Leftarrow 1 \\ \texttt{numbers\_lines2} \Leftarrow 1 \end{cases}$$

Analogamente, uma múltipla por linhas da identidade também é verificada. Observe que isso corresponde, por exemplo, a um problema cuja segunda restrição já é naturalmente de igualdade, mas a primeira é de menor ou igual.

$$\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \xRightarrow{\text{deve nos retornar}} \begin{cases} \texttt{count\_lines} \Leftarrow 1 \\ \texttt{numbers\_lines1} \Leftarrow 1 \\ \texttt{numbers\_lines2} \Leftarrow 0 \end{cases}$$

Nesse caso, como count_lines = $1 \neq 2$, então a função não identifica essa matriz como uma possível partição básica factível e o método das duas fases deverá ser executado.

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \xRightarrow{\text{deve nos retornar}} \begin{cases} \texttt{count\_lines} \Leftarrow 2 \\ \texttt{numbers\_lines1} \Leftarrow 1 \\ \texttt{numbers\_lines2} \Leftarrow 1 \end{cases}$$

Finalmente, nesse caso, o algoritmo identifica essa matriz como possível, pois count_lines = 2. Todavia, dependendo dos valores do vetor b, podemos ter uma solução básica factível ou não. Destarte, o restante do código faz a verificação, por garantia, que a solução básica é realmente factível e retorna a partição básica em questão, junto com uma lista com as colunas básicas e não-básicas. Caso a solução não seja factível, então o método das duas fases será executado. A verificação se a solução básica é factível consiste simplesmente em resolver o sistema linear (novamente com o auxílio da biblioteca numpy) $\mathbf{B} \cdot \mathbf{x_B} = \mathbf{b}$, onde B são as últimas m colunas de A caso o algoritmo tenha as identificado como possíveis e verificar que o vetor $\mathbf{x_B}$ é não-negativo.

No caso, os laços em 9 e 12 juntos tentam ver qual é a semelhança com uma linha de uma matriz identidade. No caso, temos a condicional na linha 15 para verificar justamente quais linhas tem as propriedades desejadas. Caso esse número seja igual ao total de linhas, então a partição básica em questão é calculada. Se essa solução básica for factível, então é retornada a partição básica e duas listas com os índices das colunas básicas e não-básicas na matriz original A (ou seja, retorna quais são as variáveis básicas e não-básicas respectivamente). Já se a solução for infactível ou count_lines $\neq r$ (ou m, pois $r = m$), então é retornado o valor 0 para todas as variáveis e será necessária a realização do método das duas fases por meio da inserção de variáveis artificiais no problema de PL.

```
1  def simplex(A, b, c, B, N, indB, indN):
2      # vari vel contadora de intera  es
3      inte = 1
4      while True:
5          # os par metros do problema e uma solu  o b sica fact vel inicial, al m das
       colunas da parti  o
6          c_B = []
7          c_N = []
8          for j in indB:
9              c_B.append(c[j])
10         for k in indN:
11             c_N.append(c[k])
12         x_B = solucaobasica(B, b)
13         Lambda = vetormultiplicador(B, c_B)
14         c_r = custosRelativos(c_N, Lambda, N)
15         # custo relativo m nimo
16         c_N_k = min(c_r)
17         if c_N_k >= 0:
18             return x_B, indB, indN
19         else:
20             # encontrando o  ndice  que entrar  na base na pr xima itera  o
21             var_N = 0
22             for j in range(len(c_r)):
23                 if c_N_k == c_r[j]:
24                     var_N = indN[j]
25         # pegando a coluna a_N_k
26         A_t = transpose(A)
27         a_N_k = A_t[var_N]
28         # vetor diretor simplex: d (y ser  para vari veis artificiais)
29         d = direcaoSimplex(B, a_N_k)
30         # tamanho do base e  ndice  da vari vel b sica que sair  da base
31         epsilon, x_sai_ind = tamanhoPasso(d, x_B)
32         if epsilon < 0:
33             return 0, 0, 0
34         # atualiza  o da base
35         else:
36             # atualizando as listas com os  ndices  b sicos e n o-b sicos
37             indB_entra = var_N
38             indB_sai = indB[x_sai_ind]
39             indB.remove(indB_sai)
40             indB.append(indB_entra)
41             indN.remove(indB_entra)
42             indN.append(indB_sai)
43             B = []
44             N = []
45             B_t = []
46             N_t = []
47             for i in range(len(indB)):
48                 B_t.append(A_t[indB[i]])
49             B = transpose(B_t)
```

```
            for j in range(len(indN)):
                N_t.append(A_t[indN[j]])
            N = transpose(N_t)
            inte += 1
```

Listing 10: Função do método simplex

Esse código, apesar de ser grande, não apresenta nenhuma novidade. Uma vez introduzida todas as funções no documento, temos que esse praticamente compila elas em uma ordem procedural. Ou seja, esse código tem por finalidade, através das funções definidas, usar suas informações para resolver o problema de programação linear. Notemos que já colocamos algumas condições dentro do esperado, por exemplo a linha 33 que tem por objetivo notificar um problema de solução infactível. Assim como, a partir da linha 35, temos que apenas trabalhar com os indexes para deixar a matriz básica como uma melhora da iteração anterior.

Em resumo, podemos perceber que a variável `inte` conta o número de iterações que devemos fazer para resolver o nosso problema de PL. Assim como, as linhas seguintes, 5 e 6, servem para definir tanto o custo básico como o custo não-básico. Notemos que, os laços em 8 e 10 tem os índices retornados pela função anterior 9. Assim, após definirmos as primeiras variáveis básicas e não básicas iniciamos o processo do método simplex.

Portanto, primeiro resolvemos o sistema $\mathbf{B} \cdot \hat{\mathbf{x_B}} = b$ na linha 12. Assim como, em seguida, fazemos o processo de definir o vetor multiplicador simplex, os custos relativos e retorna o mínimo dos custos relativos. No caso, notemos que a condição na linha 17 serve justamente para verificar o teste de otimalidade. Notemos que se o teste falha, então devemos seguir fazendo o cálculo da direção simplex. Por fim, temos que a função do `else:` na linha 35 é justamente para mudarmos a base que sai e o não-básico que entra. Finalmente, quando o teste de otimalidade é cumprido, a função retorna a solução do problema.

```
#Input para a matriz A#
Rest = int(input("N mero de restri  es: "))

Vari = int(input("N mero de vari veis: "))
print("---------------------------------------")
print("informe os coeficientes da matriz A:")
A = []


for i in range(Rest):

    MatJunt = list(map(float, input().split()))

    A.append(MatJunt)
#Input para o vetor b#

b = []
Restb = Rest
Varib = 1
print("---------------------------------------")
print("informe os coeficientes do vetor b: ")

for i in range(Restb):
    MatJuntb = list(map(float, input().split()))
    b.append(MatJuntb)


for j in range(len(b)):
    if b[j][0] < 0:
        b[j][0] = -b[j][0]
        for i in range(len(A[0])):
            A[j][i] = -A[j][i]
#Input para o custo#
c = []
Restc = Vari
Varic = Vari
print("---------------------------------------")
print("informe os coeficientes da Fun  o Objetivo: ")

for i in range(Restc):
    MatJuntc = list(map(float, input().split()))
    c.append(MatJuntc)
print("---------------------------------------")
m = len(A)  # n mero de linhas
n = len(A[0])  # n mero de colunas
B, N, indB, indN = verificafase1(A, b)
if B != 0:
```

```
    x_B, indB, indN = simplex(A, b, c, B, N, indB, indN)
    if indB == 0:
        print("O problema em quest o    ilimitado")
    else:
        # criando um vetor X para armazenar a resposta
        X = []
        for i in range(n):
            X.append(0)
        for j in range(len(indB)):
            X[indB[j]] = x_B[j][0]
        f = funcaoObjetivo(X, c)
        print("--------------------------------")
        print("Solu   o B sica fact vel  tima :")
        print(X)
        print("--------------------------------")
        print("Valor da solu   o:")
        print(f)
        print("--------------------------------")
else:
    # m todo das duas fases
    c_aux = []
    A_aux = copy.deepcopy(A)
    for i in range(len(c)):
        c_aux.append([0])
    # criando as vari veis artificiais
    for j in range(m):
        for i in range(m):
            A_aux[j].append(0)
        c_aux.append([1])
    for i in range(m):
        A_aux[i][i+n] = 1
    print(A_aux)
    A_aux_t = transpose(A_aux)
    # parti   o b sica inicial para o m todo das duas fases
    B_duas, N_duas, indB_duas, indN_duas = verificafase1(A_aux, b)
    # executa o m todo simplex para o problema auxiliar
    x_B_duas, indB_duas, indN_duas = simplex(
        A_aux, b, c_aux, B_duas, N_duas, indB_duas, indN_duas)
    # criando um vetor X_duas para armazenar a resposta do m todo das duas fases
    X_duas = []
    for i in range(len(A_aux[0])):
        X_duas.append(0)
    for j in range(len(indB_duas)):
        X_duas[indB_duas[j]] = x_B_duas[j][0]
    g = funcaoObjetivo(X_duas, c_aux)
    # verifica se o valor   timo     diferente de 0
    if g == 0:
        lista_aux = []
        for j in range(len(indN_duas)):
            if indN_duas[j] >= len(A[0]):
                lista_aux.append(indN_duas[j])
        for elem in lista_aux:
            indN_duas.remove(elem)
        # criando a parti   o b sica fact vel encontrada pelo m todo das duas fases
        B = []
        N = []
        B_t = []
        N_t = []
        A_t = transpose(A)
        for i in range(len(indB_duas)):
            B_t.append(A_t[indB_duas[i]])
        B = transpose(B_t)
        for j in range(len(indN_duas)):
            N_t.append(A_t[indN_duas[j]])
        N = transpose(N_t)
        # executando o m todo simplex para o problema original a partir da parti   o
    b sica encontrada pelo m todo
        x_B, indB, indN = simplex(A, b, c, B, N, indB_duas, indN_duas)
        # criando um vetor X para armazenar a resposta
        if indB == 0:
            print("O problema em quest o    ilimitado")
        else:
            # criando um vetor X para armazenar a resposta
            X = []
            for i in range(n):
                X.append(0)
```

```
            for j in range(len(indB)):
                X[indB[j]] = x_B[j][0]
            f = funcaoObjetivo(X, c)
            print("--------------------------------")
            print("Solu  o B sica fact vel  tima :")
            print(X)
            print("--------------------------------")
            print("Valor da solu  o:")
            print(f)
            print("--------------------------------")
    else:
        print("Problema infact vel")
```

Listing 11: Função do programa final

Essa é a composição final do nosso código, que também não traz nada de novo com relação ao processo simplex e o método das duas fases. Assim, os comentários devem ilustrar o que cada bloco de código tem por finalidade. Como pode ser visto, temos que os primeiros blocos de código são usados para o incremento do problema de programação linear pelo usuário. Após isso, iniciamos o processo de verificar a fase 1 e, caso seja necessário, o método das duas fases é realizado, por meio da resolução do problema auxiliar (por meio do algoritmo primal simplex) com as variáveis artificiais. Nesse caso, a partição básica fatível inicial é a solução do método das duas fases. Com a partição básica factível inicial, o método simplex é iniciado. Para cada caso usamos a folha que contém o método primal simplex em que foi baseado nosso código. Desse modo, segue como referência o link para AlgoritmoPrimalSimplex-MS428.pdf

Exemplos testados.

Com o código aparentemente funcional para qualquer saída de problemas de otimização, selecionamos alguns exemplos feitos passo-a-passo para comparar a saída do código com a formulada.

Primeiro testamos para o seguinte exemplo:

$$\textbf{Minimize:} \qquad f(x) = 4x_1 + 4x_2 + x_3$$

Sujeito a:

$$\begin{cases} x_1 + x_2 + x_3 \leq 2 \\ 2x_1 + x_2 \leq 3 \\ 2x_1 + x_2 + 3x_3 \geq 3 \\ x_1 \geq 0, x_2 \geq 0, x_3 \geq 0 \end{cases}$$

Transformamos as restrições em forma padrão:

Sujeito a:

$$\begin{cases} 1 \cdot x_1 + 1 \cdot x_2 + 1 \cdot x_3 + 1 \cdot x_4 = 2 \\ 2 \cdot x_1 + 1 \cdot x_2 + 1 \cdot x_5 = 3 \\ 2 \cdot x_1 + 1 \cdot x_2 + 3 \cdot x_3 - 1 \cdot x_6 = 3 \\ x_1 \geq 0, x_2 \geq 0, x_3 \geq 0, x_4 \geq 0, x_5 \geq 0, x_6 \geq 0 \end{cases}$$

Transformando tudo em forma matricial e adicionando as variáveis de folga (criadas para transformar o problema em forma padrão), temos:

$$\textbf{Minimize: } f(x) = \begin{bmatrix} 4 & 4 & 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix}$$

Sujeito a:

$$\left\{ \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 2 & 1 & 0 & 0 & 1 & 0 \\ 2 & 1 & 3 & 0 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 3 \end{bmatrix} ; x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \right.$$

Deste modo, temos matriz custo (que multiplica as variáveis na função objetivo, ou seja, *coeficientes da Função Objetivo*), A (isto é, a matriz dos coeficientes das variáveis de decisão nas restrições) e o *vetor b* (lado direito da igualdade das restrições, na forma padrão).

Com isso, temos que o tamanho da matriz coluna b, neste caso, é 3 e, portanto, temos 3 restrições. Outrossim, o tamanho do vetor das variáveis de decisão é 6, havendo 6 variáveis no problema.

Portanto, temos o necessário para escrever o *input* para o código. Primeiro vamos explicar como devemos implementar nosso problema de PL dentro do código. Notamos que o primeiro passo essencial é transformar em um problema da forma padrão. Portanto, devemos usar as condições explicitadas no começo da documentação. Assim, inserimos os elementos da matriz com os próprios valores dos coeficientes de cada variável do problema e cada variável de folga, para isso é usado um primeiro *input* definindo o número de restrições(ou linhas) e o número de colunas(ou variáveis do problema mais variáveis de folga), notemos que mesmo que não apareça a variável, ainda sim é necessário adicionar os coeficientes zero, assim como a separação entre cada número deve ser com o espaço do teclado, após terminar de colocar uma linha, aperte *Enter* para definir a nova linha. Em seguida, o programa irá sugerir a adição do vetor custo, a qual é feita da mesma maneira, só que com apenas um elemento em cada linha.

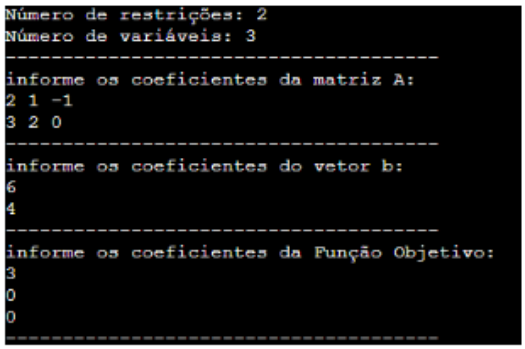
```
Número de restrições: 3
Número de variáveis: 6
------------------------------------
informe os coeficientes da matriz A:
1 1 1 1 0 0
2 1 0 0 1 0
2 1 3 0 0 -1
------------------------------------
informe os coeficientes do vetor b:
2
3
3
------------------------------------
informe os coeficientes da Função Objetivo:
4
4
1
0
0
0
------------------------------------
```

(a) Entrada do primeiro exemplo

```
--------------------------------
Solução Básica factível ótima:
[0, 0, 1.0, 1.0, 3.0, 0]
--------------------------------
Valor da solução:
1.0
--------------------------------
```

(b) Saída do primeiro exemplo

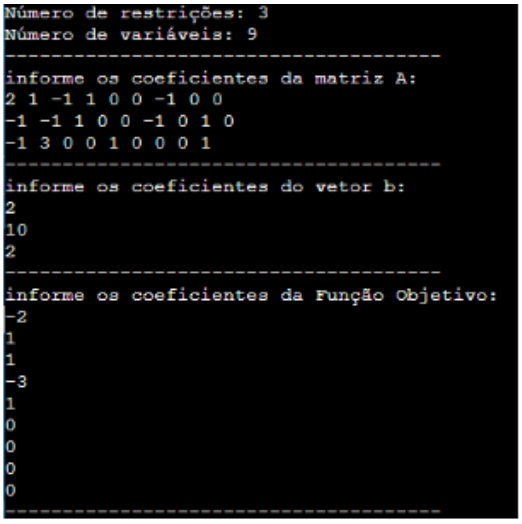
Figura 1: Exemplo 1



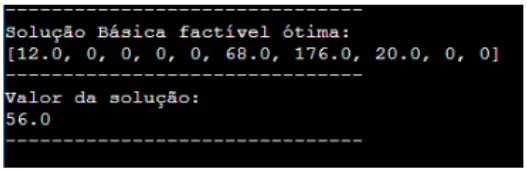
(a) Entrada do segundo exemplo

(b) Saída do segundo exemplo

Figura 2: Exemplo 2

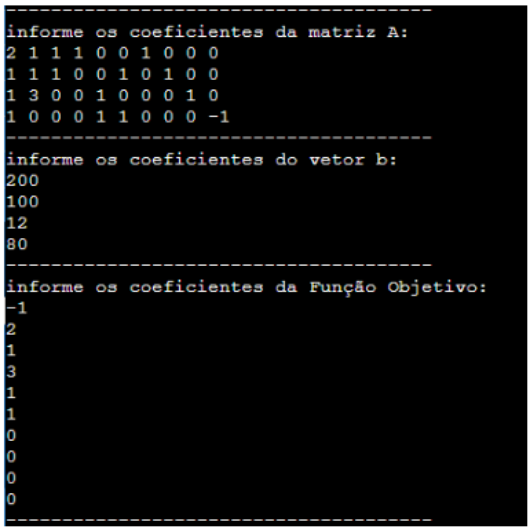


(a) Entrada do Terceiro exemplo

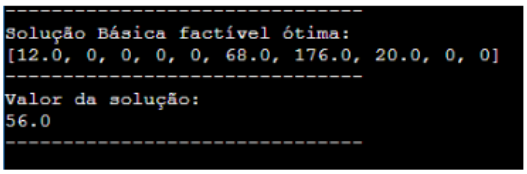


(b) Saída do Terceiro exemplo

Figura 3: Exemplo 3

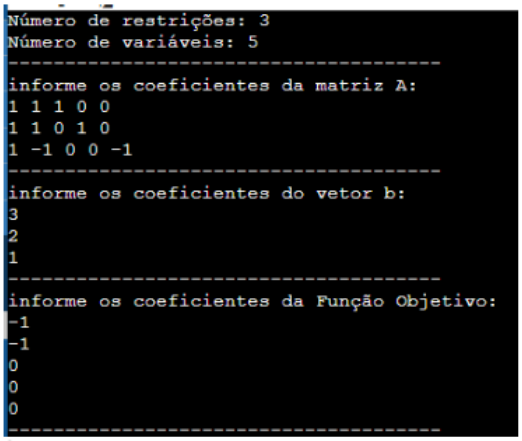


(a) Entrada do Quarto exemplo

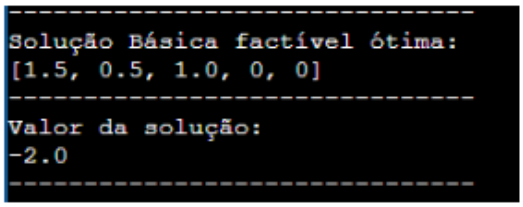


(b) Saída do Quarto exemplo

Figura 4: Exemplo 4

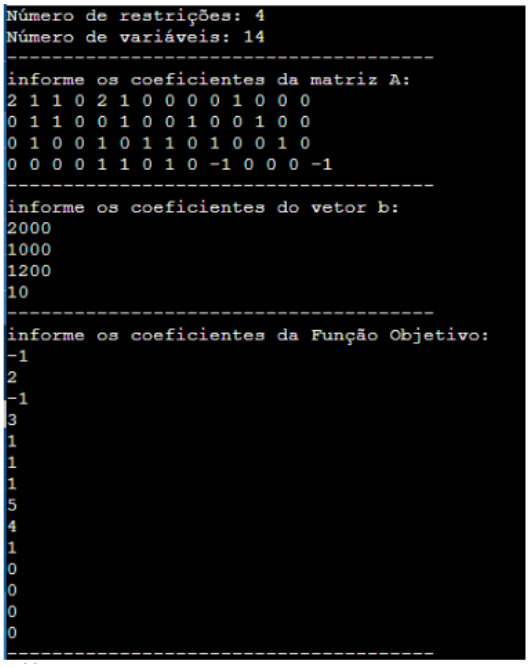


(a) Entrada do Quinto exemplo

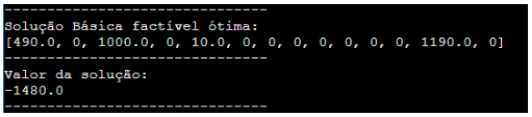


(b) Saída do Quinto exemplo

Figura 5: Exemplo 5



(a) Entrada do Sexto exemplo



(b) Saída do Sexto exemplo

Figura 6: Exemplo 6

Assim, verificamos que os nossos problemas de PL escolhidos foram solucionados pelo programa corretamente. Observe que foram utilizados problemas de PL de diversos tipos e tamanhos (como visto pelo exemplo 6, com 14 variáveis). Destarte, o grupo acredita que a utilização desse código para a solução de problemas de otimização linear é viável, embora haja espaço para melhorias na otimização do código, como na verificação da fase 1.