МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное

учреждение высшего образования

«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

**Институт компьютерных технологий и информационной безопасности**

**Кафедра Математического обеспечения и применения ЭВМ**

 

**ОТЧЁТ**

по лабораторной работе № 1

по курсу «Практикум по ПМОиРД»

Выполнили:

студенты группы КТмо2-3

Шепель И.О.

Куприянова А.А.

Проверила:

доцент каф. МОП ЭВМ

Пирская Л.В.

Оценка

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2017 г.

Таганрог 2017

Оглавление

[1. Постановка задачи 2](#_Toc493100438)

[В соответствии с вариантом №6 ставится следующая задача: 2](#_Toc493100439)

[2. Математическая модель 2](#_Toc493100440)

[2.1. Переменные задачи. 2](#_Toc493100441)

[2.2. Ограничения. 2](#_Toc493100442)

[2.3. Цeль задачи. 3](#_Toc493100443)

[2.4. Приведение задачи к канонической форме и построение симплекс-таблицы. 3](#_Toc493100444)

[3. Алгоритм решения задачи симплекс-методом 3](#_Toc493100445)

[3.1. Симплекс-таблица 3](#_Toc493100446)

[3.2. Реализация решения задачи симплекс-методом на языке Python. 4](#_Toc493100447)

[4. Результат работы программы 8](#_Toc493100448)

[5. Заключение 8](#_Toc493100449)

[Приложение. Листинг программы. 9](#_Toc493100450)

# Постановка задачи

# В соответствии с вариантом №6 ставится следующая задача:

1. На швейной фабрике для изготовления четырех видов изделий может быть использована ткань трех артикулов. Нормы расхода тканей всех артикулов на пошив одного изделия, имеющееся в распоряжении фабрики общее количество тканей каждого артикула и цена одного изделия данного вида представлены в таблице:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Артикул ткани | Норма расхода ткани, м,  на одно изделие вида | | | | Общее количество ткани,  м |
| 1 | 2 | 3 | 4 |
| I | 1 | – | 2 | 1 | 180 |
| II | – | 1 | 3 | 2 | 210 |
| III | 4 | 2 | – | 4 | 800 |
| Цена одного изделия, р. | 9 | 6 | 4 | 7 |  |

Определить, сколько изделий каждого вида должна произвести фабрика, чтобы стоимость изготовленной продукции была максимальной.

# Математическая модель

## Переменные задачи.

x1, x2, x3, x4 – свободные переменные (количество изготавливаемых изделий).

x5, x6, x7 – базисные переменные.

f = 9x1 + 6x2 + 4x3 + 7x4 – функция цели (стоимость изготовленной продукции).

с = (c1, c2, c3, c4, c5, c6, c7) = (9, 6, 4, 7, 0, 0, 0) – вектор весовых коэффициентов (исходя из цены каждого изделия).

А0  = (180, 210, 800) – вектор ограничений (исходя из общего количества ткани).

а =  – матрица норм расхода (матрица условий).

## Ограничения.

1. Количество изделий неотрицательно:

x1 ≥ 0, x2 ≥0, x3 ≥ 0, x4 ≥ 0.

1. Ограничения, следующие из ограниченности ресурсов (ткани):



## Цeль задачи.

Цель задачи состоит в том, чтобы найти такой вектор x=(x1, x2, x3, x4, x5, x6, x7), при котором значение f максимально, то есть f = c⋅x→max.

## Приведение задачи к канонической форме.

Чтобы преобразовать в уравнения неравенства ограничений пункта 2 параграфа 2.2, необходимо прибавить к левым частям неравенств базисные переменные:



Приведём полученную систему уравнений к каноническому виду:



Коэффициенты левой части системы представляют собой матрицу норм расхода:

а = .

# Алгоритм решения задачи симплекс-методом

## 3.1. Симплекс-таблица

Симлекс-таблица, для данного варианта в общем виде выглядит следующим образом:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| БП | СБ | А0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | Симплексные  отношения |
| c1 | c2 | c3 | c4 | c5 | c6 | c7 |
| x5 | 0 | b1 = А0[1] | a11 | a12 | a13 | a14 | 1 | 0 | 0 | b1/a1r |
| x6 | 0 | b2 = А0[2] | a21 | a22 | a23 | a24 | 0 | 1 | 0 | b2/a2r |
| x7 | 0 | b3 = А0[3] | a31 | a32 | a33 | a43 | 0 | 0 | 1 | b3/a3r |
| Zj - cj | | Δ0 | Δ1 | Δ2 | Δ3 | Δ4 | Δ5 | Δ6 | Δ7 |  |

Последняя строка симлекс-таблицы – индексная строка (она же строка оценок). Значения в ней рассчитываются по формулам: Δ0 = *СБ* ∙ А0, Δ*j* = *СБ* ∙ 𝐴𝑗 − 𝑐𝑗.

На начальном этапе решения задачи симплекс-таблица для данного варианта выглядит следующим образом:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| БП | СБ | А0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | Симплексные  отношения |
| 9 | 6 | 4 | 7 | 0 | 0 | 0 |
| x5 | 0 | 180 | 1 | 0 | 2 | 1 | 1 | 0 | 0 |  |
| x6 | 0 | 210 | 0 | 1 | 3 | 2 | 0 | 1 | 0 |  |
| x7 | 0 | 800 | 4 | 2 | 0 | 4 | 0 | 0 | 1 |  |
| Zj - cj | | 0 | -9 | -6 | -4 | -7 | 0 | 0 | 0 |  |

## 3.2. Реализация решения задачи симплекс-методом на языке Python.

Для решения задачи симплекс-методом, был разработан класс SimplexTable, который содержит следующие поля и методы:

free\_cnt – количество свободных переменных

basis\_cnt – количество базисных переменных

free\_coef – вектор свободных коэффициентов в симплекс-таблице (вектор с).

basis\_coef – вектор базисных коэффициентов (вектор СБ).

cond\_table – матрица норм расхода а.

cond\_coef – вектор ограничений b (при инициализации равен А0).

delta\_coef – индексная строка (вектор Δ).

Z – значение целевой функции (функции f).

simplex\_div – вектор симплексных отношений.

basis\_indexes – индексы переменных, входящих в список базисных.

iter\_cnt – счётчик итераций.

\_\_init\_\_ – конструктор (инициализация полей при создании экземпляра класса).

Вначале вызывается метод init, который принимает на вход сперва вектор с в качестве вектора свободных коэффициентов, матрицу норм расхода а, а также вектор ограничений А0. Количество базисных и свободных переменных, а также индексы переменных, входящих в состав список базисных, определяются исходя из размерности матрицы а. Номеру итерации присваивается нулевое значение.

def init(self, free\_coef, cond\_table, cond\_coef):

self.free\_cnt = cond\_table.shape[1] - cond\_table.shape[0]

self.basis\_cnt = cond\_table.shape[0]

self.free\_coef = free\_coef

self.basis\_coef = np.array(free\_coef[-self.basis\_cnt:])

self.cond\_table = cond\_table

self.cond\_coef = cond\_coef

self.delta\_coef = np.zeros(self.free\_cnt + self.basis\_cnt)

self.simplex\_div = np.zeros(self.basis\_cnt)

self.basis\_indexes = np.zeros(self.basis\_cnt)

for i in range(self.basis\_cnt):

self.simplex\_div[i] = np.nan

self.basis\_indexes[i] = self.free\_cnt + i

self.iter\_cnt = 0

Основной метод класса fit вначале вычисляет значение целевой функции (метод calc\_z) на данной итерации, затем вычисляет строку оценок (метод calc\_delta), после чего вычисляет разрешающий элемент (метод find\_resolving\_element), а также номер разрешающей строки и разрешающего столбца. После этого пересчитывается значение целевой функции. После чего метод iterate либо повторяет итерацию, либо выполнение метода fit завершается.

def fit(self):

self.calc\_z()

self.calc\_delta()

e, r, c = self.find\_resolving\_element()

print("\x1b[31;1m" + str(self.iter\_cnt) + "\x1b[0m")

print(self)

self.recalculate\_table(e, r, c)

self.iter\_cnt = 1

self.calc\_z()

while not self.iterate():

pass

Метод iterate вначале проверяет условие выхода из цикла (метод check\_delta), затем после вычисления разрешающего элемента пересчитывает строку оценок (recalculate\_delta), затем матрицу норм расхода (recalculate\_table), вычисляет значение целевой функции (calc\_z) и если условие выхода из цикла выполняется, то печатает результат (print\_res), затем наращивает счётчик итераций и возвращает флаг выхода из цикла.

def iterate(self):

print("\x1b[31;1m" + str(self.iter\_cnt) + "\x1b[0m")

res = False

if self.check\_delta():

res = True

e, r, c = self.find\_resolving\_element()

print(self)

self.recalculate\_delta(e, r, c)

self.recalculate\_table(e, r, c)

self.calc\_z()

if res:

self.print\_res()

self.iter\_cnt += 1

return res

Метод find\_resolving\_element находит разрешающий элемент как элемент матрицы cond\_table, соответсвующий разрешающему столбцу r\_col и разрешающей строке r\_row. Разрешающий столбец вычисляется по минимуму массива delta\_coef. После этого рассчитываются симплексные отношения simplex\_div. Для предотвращения деления на 0 и отрицательных симплексных отношений соответствующим таким случаям элементам массива simplex\_div присваивается значение nan. Среди остальных ищется минимум и по нему определяется разрешающая строка r\_row.

def find\_resolving\_element(self):

# индекс разрешающего столбца

r\_col = np.argmin(self.delta\_coef)

for i, ac in enumerate(self.cond\_coef):

if self.cond\_table[i][r\_col] == 0:

self.simplex\_div[i] = np.nan

continue

self.simplex\_div[i] = ac / self.cond\_table[i][r\_col]

if self.simplex\_div[i] < 0:

self.simplex\_div[i] = np.nan

r\_row = np.nanargmin(self.simplex\_div)

return self.cond\_table[r\_row][r\_col], r\_row, r\_col

Метод triangle\_rule пересчитывает таблицу a по правилу треугольника. Сперва создаётся новая матрица new\_table той же размерности, что и матрица cond\_table. Разрешающая строка переносится в новую матрицу без изменений. Элемент, стоящий на месте разрешающего, принимает нулевое значение. Все остальные элементы рассчитываются по правилу треугольника. Затем матрице cond\_table присваивается значение new\_table.

def triangle\_rule(self, res\_elem, res\_row, res\_col):

new\_table = np.zeros(self.cond\_table.shape)

for i in range(self.cond\_table.shape[0]):

if i == res\_row:

new\_table[i] = self.cond\_table[i]

continue

for j in range(self.cond\_table.shape[1]):

if j == res\_col:

new\_table[i][j] = 0

continue

new\_table[i][j] = self.cond\_table[i][j] -\(self.cond\_table[res\_row][j] \* self.cond\_table[i][res\_col]) / res\_elem

self.cond\_table = new\_table

Метод recalculate\_table пересчитывает всю оставшуюся часть симплекс-таблицы: заменяет базисную переменную, соответствующую разрешающей строке, на переменную, соответствующую разрешающему столбцу. Следом за этим заменяет соответствующие замененной переменной коэффициенты. Также пересчитывает разрешающую строку и коэффициенты ограничивающих условий.

def recalculate\_table(self, res\_elem, res\_row, res\_col):

# заменяем базисную переменную

self.basis\_indexes[res\_row] = res\_col

self.basis\_coef[res\_row] = self.free\_coef[res\_col]

res\_cond = self.cond\_coef[res\_row]

# по правилу треугольника пересчитываем коэффициенты ограничивающих условий

for i in range(self.basis\_cnt):

if i == res\_row:

continue

self.cond\_coef[i] -= res\_cond \* self.cond\_table[i][res\_col] / res\_elem

self.triangle\_rule(res\_elem, res\_row, res\_col)

# делим разрешающую строку на разрешающий элемент

self.cond\_coef[res\_row] /= res\_elem

self.cond\_table[res\_row] /= res\_elem

Метод recalculate\_delta пересчитывает строку оценок, при этом элементу этой строки, который соответствует разрешающему столбцу, присваивается нулевое значение.

def recalculate\_delta(self, res\_elem, res\_row, res\_col):

res\_delta = self.delta\_coef[res\_col]

for i in range(self.free\_cnt + self.basis\_cnt):

if i == res\_col:

self.delta\_coef[i] = 0

continue

self.delta\_coef[i] -= self.cond\_table[res\_row][i] \* res\_delta / res\_elem

Метод check\_delta проверяет условия выхода из цикла: все элементы строки оценок неотрицательны.

def check\_delta(self):

return np.all(self.delta\_coef >= 0)

Метод print\_res выводит на печать максимум целевой функции и вектор х, при котором он достигается.

def print\_res(self):

res\_str = "SOLVED: f = " + str(self.Z) + " at ("

for i in range(self.free\_cnt):

if i >= self.basis\_cnt:

res\_str += '0'

continue

if len(self.cond\_coef[np.argwhere(self.basis\_indexes==i)])!= 0:

res\_str += str(self.cond\_coef[np.argwhere(self.basis\_indexes == i)][0][0])

else:

res\_str += '0'

res\_str += ', '

res\_str += ")\n"

print(res\_str)

Метод \_\_str\_\_ выводит на печать симплекс-таблицу.

def \_\_str\_\_(self):

s = "\x1b[31;1m" + "B\_i\tC\_b\tA\_j\t" + "\x1b[0m"

for i in self.free\_coef:

s += "\x1b[32;1m" + str(i) + "\x1b[0m" + "\t"

s += "\x1b[31;1m" + "Simplex\n" + "\x1b[0m"

for i in range(self.cond\_table.shape[0]):

s += "\x1b[32;1m" + str(self.basis\_indexes[i]) + "\t" + str(self.basis\_coef[i]) + "\x1b[0m" + "\t" +\

str(self.cond\_coef[i]) + "\t"

for j in range(self.cond\_table.shape[1]):

s += str(self.cond\_table[i][j]) + "\t"

s += "\x1b[34;1m" + str(self.simplex\_div[i]) + "\x1b[0m" + "\n"

s += "\x1b[31;1m" + " \tZ\_j\t" + "\x1b[0m" + "\x1b[34;1m" + str(self.Z)

for i in self.delta\_coef:

s += "\t" + str(i)

s += "\x1b[0m" + "\n"

return s

# Результат работы программы

# Заключение

В результате выполнения лабораторной работы был реализован симплекс-метод решения задачи линейного программирования на языке высокого уровня Python.

Для работы с векторами и матрицами была использована библиотека numpy. Методы библиотеки применялись для:

инициализации массивов,

работы с размерностью,

перебора элементов.

В ходе выполнения лабораторной работы были получены навыки решения задачи линейного программирования при помощи симплекс-таблиц.

# Приложение. Листинг программы.

import numpy as np

# Problem No. 6

# начальные условия

c = np.array([9., 6., 4., 7., 0., 0., 0.])

A0 = np.array([180., 210., 800.])

a = np.array([

[1., 0., 2., 1., 1., 0., 0.],

[0., 1., 3., 2., 0., 1., 0.],

[4., 2., 0., 4., 0., 0., 1.]

])

# флаг печати таблиц после каждой итерации

PRNT\_DBG = True

class SimplexTable:

# инициализация по умолчанию

def \_\_init\_\_(self):

self.free\_cnt = 0

self.basis\_cnt = 0

self.free\_coef = np.array([])

self.basis\_coef = np.array([])

self.cond\_table = np.array([])

self.cond\_coef = np.array([])

self.delta\_coef = np.array([])

self.Z = 0

self.simplex\_div = np.array([])

self.basis\_indexes = np.array([])

# создание экземпляра класса

def init(self, free\_coef, cond\_table, cond\_coef):

self.free\_cnt = cond\_table.shape[1] - cond\_table.shape[0]

self.basis\_cnt = cond\_table.shape[0]

self.free\_coef = free\_coef

self.basis\_coef = np.array(free\_coef[-self.basis\_cnt:])

self.cond\_table = cond\_table

self.cond\_coef = cond\_coef

self.delta\_coef = np.zeros(self.free\_cnt + self.basis\_cnt)

self.simplex\_div = np.zeros(self.basis\_cnt)

self.basis\_indexes = np.zeros(self.basis\_cnt)

for i in range(self.basis\_cnt):

self.simplex\_div[i] = np.nan

self.basis\_indexes[i] = self.free\_cnt + i

self.iter\_cnt = 0

# вычисление значения целевой функции

def calc\_z(self):

self.Z = 0

for b, A in zip(self.basis\_coef, self.cond\_coef):

self.Z += b \* A

# вычисление элементов строки оценок

def calc\_delta(self):

for i, c in enumerate(self.free\_coef):

self.delta\_coef[i] = self.Z - c

# нахождение разрешающего элемента и симплексных отношений

def find\_resolving\_element(self):

# индекс разрешающего столбца

r\_col = np.argmin(self.delta\_coef)

for i, ac in enumerate(self.cond\_coef):

if self.cond\_table[i][r\_col] == 0:

self.simplex\_div[i] = np.nan

continue

self.simplex\_div[i] = ac / self.cond\_table[i][r\_col]

if self.simplex\_div[i] < 0:

self.simplex\_div[i] = np.nan

r\_row = np.nanargmin(self.simplex\_div)

return self.cond\_table[r\_row][r\_col], r\_row, r\_col

# пересчёт матрицы а по методу треугольника

def triangle\_rule(self, res\_elem, res\_row, res\_col):

new\_table = np.zeros(self.cond\_table.shape)

for i in range(self.cond\_table.shape[0]):

if i == res\_row:

new\_table[i] = self.cond\_table[i]

continue

for j in range(self.cond\_table.shape[1]):

if j == res\_col:

new\_table[i][j] = 0

continue

new\_table[i][j] = self.cond\_table[i][j] - (self.cond\_table[res\_row][j] \*\

self.cond\_table[i][res\_col]) / res\_elem

self.cond\_table = new\_table

# пересчет симплекс таблицы

def recalculate\_table(self, res\_elem, res\_row, res\_col):

# заменяем базисную переменную

self.basis\_indexes[res\_row] = res\_col

self.basis\_coef[res\_row] = self.free\_coef[res\_col]

res\_cond = self.cond\_coef[res\_row]

# по правилу треугольника пересчитываем коэффициенты ограничивающих условий

for i in range(self.basis\_cnt):

if i == res\_row:

continue

self.cond\_coef[i] -= res\_cond \* self.cond\_table[i][res\_col] / res\_elem

self.triangle\_rule(res\_elem, res\_row, res\_col)

# делим разрешающую строку на разрешающий элемент

self.cond\_coef[res\_row] /= res\_elem

self.cond\_table[res\_row] /= res\_elem

# перерасчет строки оценок

def recalculate\_delta(self, res\_elem, res\_row, res\_col):

res\_delta = self.delta\_coef[res\_col]

for i in range(self.free\_cnt + self.basis\_cnt):

if i == res\_col:

self.delta\_coef[i] = 0

continue

self.delta\_coef[i] -= self.cond\_table[res\_row][i] \* res\_delta / res\_elem

# проверка условия выхода: неотрицательность элементов строки оценок

def check\_delta(self):

return np.all(self.delta\_coef >= 0)

# итерация

def iterate(self):

print("\x1b[31;1m" + str(self.iter\_cnt) + "\x1b[0m")

res = False

if self.check\_delta():

res = True

e, r, c = self.find\_resolving\_element()

print(self)

self.recalculate\_delta(e, r, c)

self.recalculate\_table(e, r, c)

self.calc\_z()

if res:

self.print\_res()

self.iter\_cnt += 1

return res

# главная функция, выполняющая максимизацию по методу симплекс таблицы

def fit(self):

self.calc\_z()

self.calc\_delta()

e, r, c = self.find\_resolving\_element()

print("\x1b[31;1m" + str(self.iter\_cnt) + "\x1b[0m")

print(self)

self.recalculate\_table(e, r, c)

self.iter\_cnt = 1

self.calc\_z()

while not self.iterate():

pass

# функция печати результата

def print\_res(self):

res\_str = "SOLVED: f = " + str(self.Z) + " at ("

for i in range(self.free\_cnt):

if i >= self.basis\_cnt:

res\_str += '0'

continue

if len(self.cond\_coef[np.argwhere(self.basis\_indexes == i)]) != 0:

res\_str += str(self.cond\_coef[np.argwhere(self.basis\_indexes == i)][0][0])

else:

res\_str += '0'

res\_str += ', '

res\_str += ")\n"

print(res\_str)

# перегрузка функции печати для класса

def \_\_str\_\_(self):

s = "\x1b[31;1m" + "B\_i\tC\_b\tA\_j\t" + "\x1b[0m"

for i in self.free\_coef:

s += "\x1b[32;1m" + str(i) + "\x1b[0m" + "\t"

s += "\x1b[31;1m" + "Simplex\n" + "\x1b[0m"

for i in range(self.cond\_table.shape[0]):

s += "\x1b[32;1m" + str(self.basis\_indexes[i]) + "\t" + str(self.basis\_coef[i]) + "\x1b[0m" + "\t" +\

str(self.cond\_coef[i]) + "\t"

for j in range(self.cond\_table.shape[1]):

s += str(self.cond\_table[i][j]) + "\t"

s += "\x1b[34;1m" + str(self.simplex\_div[i]) + "\x1b[0m" + "\n"

s += "\x1b[31;1m" + " \tZ\_j\t" + "\x1b[0m" + "\x1b[34;1m" + str(self.Z)

for i in self.delta\_coef:

s += "\t" + str(i)

s += "\x1b[0m" + "\n"

return s

st = SimplexTable()

st.init(c, a, A0)

st.fit()