Федеральное государственное автономное образовательное учреждение
высшего образования
«Национальный исследовательский университет ИТМО»
Факультет программной инженерии и компьютерной техники

Дисциплина «Вычислительная математика»

Лабораторная работа №1

Вариант 13

Выполнил:
Студент группы P3206
Сорокин Артём Николаевич
Преподаватель:
Рыбаков Степан Дмитриевич

г. Санкт-Петербург
2024 год

Оглавление

1. Текст задания

Решить СЛАУ методом простых итераций.

Для итерационных методов должно быть реализовано:

- Точность задается с клавиатуры/файла
- Проверка диагонального преобладания (в случае, если диагональное преобладание в исходной матрице отсутствует, сделать перестановку строк/столбцов до тех пор, пока преобладание не будет достигнуто). В случае невозможности достижения диагонального преобладания - выводить соответствующее сообщение.
- Вывод вектора неизвестных: $x_1,\ x_2,\ \ldots, x_n$
- Вывод количества итераций, за которое было найдено решение.
- Вывод вектора погрешностей: $|x_i^{(k)} - x_i^{(k-1)}|$

Рисунок 1 – Задание

2. Исходный код

```
from io_matrix import read_matrix
from simple_iteration import simple_iteration
if __name__=="__main__":
print("CompLab1 by LocalPiper")
print("#13: Simple Iteration method")
try:
matrix = read_matrix()
simple_iteration(matrix)
except FileNotFoundError:
print("Requested file was not found")

class Matrix:
def __init__(self):
self.dimension = 0
self.presicion = 0
self.matrix = []
self.expansion = []
self.solution = []

def print_matrix(self):
print("Matrix of dimension: " + str(self.dimension))
for i in range(self.dimension):
```

```
s = ""
for j in range(self.dimension):
s += "{:14.12f} ".format(self.matrix[i][j])
s += "| {:14.12f}".format(self.expansion[i])
print(s)
print("\n")

def get_diagonal(self):
diag = []
for i in range(self.dimension):
diag.append(self.matrix[i][i])
return diag
def print_solution(self):
print("Solution:")
for i in range(self.dimension):
print("x" + str(i+1) + " = " + "{:14.12f}".format((self.solution[i])) + "+-" + str(self.presicion))


from matrix import Matrix

FILE_MODE = "f"
CONSOLE_MODE = "m"

def read_matrix_from_file():
print("Your file structure should look like this:")
print("n p")
print("a11 a12 ... a1n b1 ")
print("a21 a22 ... a2n b2 ")
print("... ... ... ... ...")
print("an1 an2 ... ann bn ")
print("where n - dimension of matrix, p - presicion")
print("Example:")
print("3 0.00001")
print("1 2 3 1")
print("4 5 6 2")
print("7 8 9.1 3\n")
filename = input("Input file name: ")
matrix = Matrix()
with open(filename, "r") as f:
data = f.readline().strip().split(" ")
matrix.dimension = int(data[0])
matrix.presicion = float(data[1])
for _ in range(matrix.dimension):
row = [float(x) for x in f.readline().strip().split(" ")]
res = row.pop()
matrix.expansion.append(res)
matrix.matrix.append(row)

matrix.print_matrix()
```

```
return matrix
def read_matrix_from_console():
matrix = Matrix()
done = False
while not done:
try:
matrix.dimension = int(input("Input dimension: "))
if matrix.dimension <= 0:
raise ValueError
except ValueError:
print("Wrong dimension value! Try again")
continue

try:
matrix.presicion = float(input("Input presicion: "))
if matrix.presicion >= 1:
raise ValueError
except ValueError:
print("Wrong presicion value! Try again")
continue
try:
for i in range(matrix.dimension):
row = [float(x) for x in input("Input " + str(i + 1) + " row of matrix: ").split(" ")]
if len(row) != matrix.dimension:
raise ValueError
matrix.matrix.append(row)
except ValueError:
print("Wrong values! Try again")
continue

res = [float(x) for x in input("Input vector of answers: ").split(" ")]
for a in res:
matrix.expansion.append(a)
done = True

matrix.print_matrix()
return matrix

def read_matrix():
print("Choose preferred option:")
print("m - input matrix manually (for experienced users only)")
print("f - input matrix by file")

running = True
mode = ''
while (running):
mode = input()
if ((mode != FILE_MODE) and (mode != CONSOLE_MODE)):
print("Wrong input! Try again")
```

```
else:
running = False
if (mode == FILE_MODE):
return read_matrix_from_file()
else:
return read_matrix_from_console()

from matrix import Matrix

def check_zeroes(matrix: Matrix):
for i in range(matrix.dimension):
if sum(matrix.matrix[i]) == 0:
print("Seems like this matrix has a full-zero row: " + str(i + 1))
return True
return False

def check_linear_dependency(matrix: Matrix):
m = matrix.matrix
det = determinant_fast(m)
if det == 0:
print("Seems like this matrix has a linear dependency")
return True
return False

def determinant_fast(A : list):
n = len(A)
AM = A
for fd in range(n):
for i in range(fd+1,n):
if AM[fd][fd] == 0:
AM[fd][fd] = 1.0e-18
crScaler = AM[i][fd] / AM[fd][fd]
for j in range(n):
AM[i][j] = AM[i][j] - crScaler * AM[fd][j]

product = 1.0
for i in range(n):
product *= AM[i][i]
return product


def check_health(matrix: Matrix):
diag = matrix.get_diagonal()
if diag.count(0) != 0:
return True
return False


from matrix import Matrix
```

```
import solvability

def check_diagonal_dominance(matrix: Matrix):
for i in range(matrix.dimension):
if abs(matrix.matrix[i][i]) < (sum([abs(x) for x in matrix.matrix[i]]) - abs(matrix.matrix[i][i])):
return False
return True

def diagonal_dominant(l : list):
for i in range(len(l)):
if abs(l[i]) >= sum([abs(x) for x in l]) - abs(l[i]):
return i
return -1

def try_sort_by_diagonal(matrix: Matrix):
# finding dominating elements
sorting = [-1 for _ in range(matrix.dimension)]
success = True

for i in range(matrix.dimension):
res = diagonal_dominant(matrix.matrix[i])
sorting[i] = res
#trying to sort the matrix
raw_sorted_matrix = [[] for _ in range(matrix.dimension)]
raw_sorted_matrix_expansion = [0 for _ in range(matrix.dimension)]
buffer = []
for i in range(len(sorting)):
if sorting[i] != -1:
if len(raw_sorted_matrix[sorting[i]]) == 0:
raw_sorted_matrix[sorting[i]] = matrix.matrix[i]
raw_sorted_matrix_expansion[sorting[i]] = matrix.expansion[i]
else:
buffer.append(i)
else:
buffer.append(i)
#pushing leftovers into our matrix
if (len(buffer) != 0):
success = False
for i in range(len(buffer)):
for j in range(matrix.dimension):
if len(raw_sorted_matrix[j]) == 0:
raw_sorted_matrix[j] = matrix.matrix[buffer[i]]
raw_sorted_matrix_expansion[j] = matrix.expansion[buffer[i]]
break
#creating new matrix out of the monstrosity we just did
new_matrix = Matrix()
new_matrix.dimension = matrix.dimension
new_matrix.presicion = matrix.presicion
new_matrix.matrix = [x for x in raw_sorted_matrix]
```

```
new_matrix.expansion = [x for x in raw_sorted_matrix_expansion]
return new_matrix, success

def create_normalized_matrix(matrix : Matrix):
n_matrix = Matrix()
n_matrix.dimension = matrix.dimension
for i in range(matrix.dimension):
n_matrix.expansion.append(matrix.expansion[i] / matrix.matrix[i][i])
m = []
for j in range(matrix.dimension):
if (i == j):
m.append(0)
else:
m.append(matrix.matrix[i][j] / matrix.matrix[i][i])
n_matrix.matrix.append(m)
return n_matrix

def simple_iteration(matrix : Matrix):
#create a copy for solvability checks
copy = Matrix()
copy.dimension = matrix.dimension
copy.presicion = matrix.presicion
copy.expansion = [x for x in matrix.expansion]
copy.matrix = [[y for y in x] for x in matrix.matrix]

#run solvability checks
if solvability.check_zeroes(copy) or solvability.check_linear_dependency(copy):
print("Matrix cannot be solved!")
return
running = True
it = 0
# checking if matrix is already dominant
if check_diagonal_dominance(matrix):
print("Matrix is already diagonally dominant")
else:
# trying to transform matrix to diagonal dominance
matrix, result = try_sort_by_diagonal(matrix)
if not result:
print("Matrix diverges! Trying to perform 100 computations...")

print("After transforming, matrix looks like this:")
matrix.print_matrix()
try:
# check main diagonal
need_restructure = solvability.check_health(matrix)
if need_restructure:
matrix, success = rearrange_matrix(matrix)
if not success:
print("Matrix cannot be solved!")
```

```
return
# trying to normalize matrix
n_matrix = create_normalized_matrix(matrix)
print("After normalizing, matrix looks like this:")
n_matrix.print_matrix()
vector = [x for x in matrix.expansion]

for i in range(matrix.dimension):
matrix.solution.append(n_matrix.expansion[i])
# running computations...
while ((running) and (it < 100)):
new_vector = []

for i in range(matrix.dimension):
val = n_matrix.expansion[i]
for j in range(matrix.dimension):
val -= n_matrix.matrix[i][j] * vector[j]
new_vector.append(val)

for i in range(matrix.dimension):
matrix.solution[i] = new_vector[i]

delta = []
for i in range(matrix.dimension):
delta.append(abs(new_vector[i] - vector[i]))
vector = [x for x in new_vector]
if max(delta) < matrix.presicion:
running = False
it += 1

matrix.print_solution()
print("Iterations performed: " + str(it))
print("Errors: ")
for i in range(len(delta)):
print("delta x" + str(i+1) + ": " + str(delta[i]))

except ZeroDivisionError:
print("It seems like either you printed in the wrong matrix \nor my dumbass author screwed up while
restructuring the matrix, therefore creating a zero division")

def rearrange_matrix(m : Matrix):
matrix = m.matrix
n = len(matrix)
sorted_matrix = sorted(matrix, key=lambda x: x.count(0))

for i in range(n):
if sorted_matrix[i][i] == 0:
for j in range(i+1, n):
if sorted_matrix[j][i] != 0:
```

```
sorted_matrix[i], sorted_matrix[j] = sorted_matrix[j], sorted_matrix[i]
break
else:
print("It's impossible to rearrange matrix - main diagonal still has zeroes")
return m, False
m.matrix = sorted_matrix
return m, True
```

3. Заключение

В ходе выполнения данной лабораторной работы я познакомился с итерационными методами решения систем линейных уравнений, программно реализовал один из таких методов, исследовал особые случаи работы метода простой итерации.