Федеральное государственное автономное

образовательное учреждение высшего образования

Национальный Исследовательский Университет ИТМО

Лабораторная работа 1

«Решение системы линейных алгебраических уравнений СЛАУ»

Дисциплина: Вычислительная математика

Вариант 13. Метод простых итераций

Выполнил: Терехин Никита Денисович

Факультет: Программной инженерии и компьютерной техники

Группа: P3208

Преподаватель: Машина Екатерина Алексеевна

г. Санкт-Петербург, 2024 год

Оглавление

Цель работы

Составить программу для решения системы линейных алгебраических уравнений методом простых итераций. Понять как автоматизировать численные методы из математики

Текст задания

1. В программе численный метод должен быть реализован в виде отдельной подпрограммы/метода/класса, в который исходные/выходные данные передаются в качестве параметров
2. Размерность матрицы n ≤ 20 (задается из файла или с клавиатуры - по выбору конечного пользователя)
3. Должна быть реализована возможность ввода коэффициентов матрицы, как с клавиатуры, так и из файла по выбору конечного пользователя

- 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 , … x_n
- Вывод количества итераций, за которое было найдено решение
- Вывод вектора погрешностей: $|x_i^{(k)} - x_i^{(k-1)}|$

Описание методы, расчетные формулы

Из исходной системы можно выразить вектор в формате

$$x_i = \sum_{j=1}^{n} c_{ij} x_j + d_i \quad \text{где } i = 1, 2, 3, \ldots$$

$$c_{ij} = \frac{a_{ij}}{a_{ii}} \text{ при } i \neq j \quad 0 \text{ при } i = j$$

$$d_i = \frac{b_i}{a_{ii}} \quad i = 1, 2, 3, \ldots$$

Расчетная формула метода простых итераций имеет вид:

$$x_i^{(k+1)} = \frac{b_i}{a_{ii}} - \sum_{i=1\ j\neq i}^{n} \frac{a_{ij}}{a_{ii}} x_i^{(k)} \quad i = 1, 2, 3, \ldots$$

Условие преобладания диагональных элементов

$$|a_{ii}| \geq \sum_{i\neq i} |a_{ij}| \quad i = 1, 2, 3, \ldots$$

Листинг программы

```
# main.py

from typing import List, TextIO
import re
from exceptions import DiagonalDominatingError, ParsingError


def swap_rows_sort(mat: List[List[float]]) -> List[List[float]]:
    m: int = len(mat)
    sort_mat: List[List[float]] = [[]] * m
    for i in range(m):
        abs_row: List[float] = [abs(num) for num in mat[i][:-1]]
        max_num: float = max(abs_row)
        max_ind: int = abs_row.index(max_num)
        if sort_mat[max_ind]:
            raise DiagonalDominatingError("Matrix isn't diagonally
dominated")
        row_sum: int = sum(abs_row)
        if row_sum > 2 * max_num:
            raise DiagonalDominatingError(f"Diagonal dominating is
broken in row {i + 1} of your equation")
        sort_mat[max_ind] = mat[i]
    return sort_mat


def select_vector_equation(mat: List[List[float]]) ->
List[List[float]]:
    vector_mat: List[List[float]] = []
    for i in range(len(mat)):
        vector_mat.append(list(map(lambda num: num / mat[i][i] if
num != mat[i][i] else 0, mat[i])))
    return vector_mat


def get_next_approx(mat: List[List[float]], x: List[float],
precision: int) -> List[float]:
```

```
    next_approx: List[float] = []
    for i in range(len(x)):
        next_approx.append(round(mat[i][-1] - sum([mat[i][j] * x[j]
for j in range(len(x))]), precision))
    return next_approx


def get_precision(epsilon: float) -> int:
    precision: int = 2
    eps_iter: float = epsilon
    while eps_iter < 1:
        eps_iter *= 10
        precision += 1
    return precision


def do_simple_iteration(mat: List[List[float]], approx:
List[float], precision: float, step: int) -> List[float]:
    k: int = len(approx)
    next_approx: List[float] =
get_next_approx(select_vector_equation(mat), approx,
get_precision(precision))
    dispersion_vec = [round(abs(approx[i] - next_approx[i]),
get_precision(precision)) for i in range(k)]
    if max(dispersion_vec) < precision:
        print(f'Final iteration amount is {step}')
        print(f'Dispersion vector is: ', dispersion_vec)
        return next_approx
    return do_simple_iteration(mat, next_approx, precision, step +
1)


def read_matrix_from_console(lines: int) -> List[List[float]]:
    print('Input your equation as an extended matrix line by line',
          'The format is: ' + ' '.join([f'a[{i + 1}]' for i in
range(lines)]) + ' | b', sep='\n')
    while True:
        mat: List[List[float]] = []
        for i in range(lines):
            mat.append(read_row_from_console(i + 1, lines))
        if find_determinant(mat) == 0:
            print('Matrix is non-degenerate')
            continue
        try:
            return swap_rows_sort(mat)
        except DiagonalDominatingError as e:
            print(e)
```

```
def read_row_from_console(step: int, m: int) -> List[float]:
    while True:
        s: str = input(f'Line [{step} / {m}]: ')
        try:
            return parse_input_row(s, m)
        except (ParsingError, ValueError) as e:
            print(e)


def parse_input_row(row: str, m: int) -> List[float]:
    s: List[str] = list(filter(lambda a: a, re.split("[ |]+",
row)))
    if len(s) != m + 1:
        raise ParsingError('Dimension is incorrect')
    return list(map(float, s))


def read_dimension_and_precision() -> tuple[int, float]:
    dim: int = 0
    precision: float = 0
    while True:
        try:
            dim = int(input('Enter matrix dimension: '))
        except ValueError:
            print("Can't parse integer value")
            continue
        if dim > 20 or dim < 0:
            print('It should be less than or equal to 20. Try
again')
            continue
        break

    while True:
        try:
            precision = float(input('Input precision number: '))
        except ValueError:
            print("Can't parse float value")
            continue
        if precision >= 1 or precision <= 1e-10:
            print('Precision is a number less than 1. Please try
again')
            continue
        break
    return dim, precision
```

```
def read_matrix_from_file() -> tuple[int, float,
List[List[float]]]:
    while True:
        it: int = 1
        try:
            file: TextIO = open(input('Input file name with
extension: '), 'r')
            mat: List[List[float]] = []
            m = int(file.readline().strip())
            ep = float(file.readline().strip())
            while it <= m:
                row: List[float] = parse_input_row(file.readline(),
m)
                mat.append(row)
                it += 1
            if find_determinant(mat) == 0:
                print('Matrix is non-degenerate')
                continue
            return m, ep, swap_rows_sort(mat)
        except (ValueError, FileNotFoundError,
DiagonalDominatingError) as e:
            print(e)
        except ParsingError as e:
            print(e, f'Error in row {it}: check your input format
or use another file', sep='\n')


def choose_input() -> int:
    variants: List[str] = ['From file', 'Using console']
    print('How do you want to read your equation?',
          '\n'.join([f'{ind + 1}. {v}' for ind, v in
enumerate(variants)]), sep='\n')
    while True:
        try:
            var = int(input())
        except ValueError:
            print('No such variant. Try again')
            continue
        if var < 1 or var > len(variants):
            print('No such variant. Try again')
            continue
        return var


def find_determinant(mat: List[List[float]]) -> float:
    copy: List[List[float]] = [row[:-1] for row in mat]
    ind: List[int] = list(range(len(copy)))
    if len(copy) == 2 and len(copy[0]) == 2:
```

```
        return copy[0][0] * copy[1][1] - copy[1][0] * copy[0][1]
    res: float = 0
    for i in ind:
        copy = [row[:] for row in mat]
        copy = copy[1:]
        height: int = len(copy)
        for j in range(height):
            copy[j] = copy[j][0:i] + copy[j][i + 1:]
        sign: int = (-1) ** (i % 2)
        det: float = find_determinant(copy)
        res += sign * mat[0][i] * det
    return res


inp: int = choose_input()
matrix: List[List[float]]
n: int = 0
eps: float = 0

if inp == 1:
    n, eps, matrix = read_matrix_from_file()
else:
    n, eps = read_dimension_and_precision()
    matrix = read_matrix_from_console(n)

print('Answer vector is: ', do_simple_iteration(matrix, [0] * n,
eps, 1))
```

```
# exceptions.py

class DiagonalDominatingError(Exception):
    def __init__(self: Exception, message: str):
        super().__init__(message)

class ParsingError(Exception):
    def __init__(self: Exception, message: str):
        super().__init__(message)
```

Примеры и результаты работы

Чтение из файла

```
# test.py

3
0.01
2 2 10 | 14
```

```
10 1 1 | 12
2 10 1 | 13
```

Вывод программы

```
How do you want to read your equation?
1. From file
2. Using console
1
Input file name with extension: test.txt
Final iteration amount is 6
Dispersion vector is:  [0.0019, 0.0024, 0.0031]
Answer vector is:  [0.9996, 0.9995, 0.9993]

Process finished with exit code 0
```

Пример обработки ошибок

```
How do you want to read your equation?
1. From file
2. Using console
2
Enter matrix dimension: -3
It should be less than or equal to 20. Try again
Enter matrix dimension: 3
Input precision number: acb
Can't parse float value
Input precision number: .01
Input your equation as an extended matrix line by line
The format is: a[1] a[2] a[3] | b
Line [1 / 3]: 3 e 2 | 1
could not convert string to float: 'e'
Line [1 / 3]: 2 3
Dimension is incorrect
Line [1 / 3]: 1 2 10 | 14
Line [2 / 3]: 2 10 3 | 2
Line [3 / 3]: 2 4 10 | 5
Matrix isn't diagonally dominated
Line [1 / 3]: ...
```

Выводы

В ходе выполнения работы удалось изучить на примере программы линейных алгебраических уравнений методом простых итераций. Получилось понять как автоматизировать численные методы из

математики, переносить их в код с учетом ошибок ввода пользователей и погрешности

Итерационные методы имеют преимущества в виде заданной точности вычислений, что позволяет сократить на хранение всей системы, однако в подобных методах невозможно предсказать время выполнение программы, т.к количество итераций растет пропорционально количеству строк в уравнении.