Metody Numeryczne Projekt 2
Układy równań liniowych

Michał Bałdyga 184523 gr.1 Informatyka

1 Wstęp

Celem projektu była implementacja metod iteracyjnych (Jacobiego i Gaussa-Seidla) i bezpośrednich (faktoryzacja LU) rozwiązywania układów równań liniowych. Działanie metod iteracyjnych polega na wyznaczaniu coraz dokładniejszych przybliżeń rzeczywistego rozwiązania wraz z kolejnymi iteracjami, natomiast metody bezpośrednie pozwalają na otrzymanie dokładnego rozwiązania.

2 Konstruckcja układu równań

Układ równań liniowych ma następującą postać:

$$Ax = b$$

gdzie:

- A - macierz systemowa
- b - wektor pobudzenia
- x - wektor rozwiązań

Na potrzeby testów przyjmijmy, że A jest tzw. macierzą pasmową o rozmiarze NxN:

$$A = \begin{bmatrix} a_1 & a_2 & a_3 & \dots & 0 \\ a_2 & a_1 & a_2 & \dots & 0 \\ a_3 & a_2 & a_1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & a_1 \end{bmatrix}$$

Macierz A zawiera więc pięć diagonali - główna z elementami $a1$, dwie sąsiednie z elementami $a2$ i dwie skrajne diagonale z elementami $a3$.
Prawa strona równania to wektor b o długości N.
W wyniku rozwiązania układu równań $Ax = b$ otrzymujemy wektor x.

[1]:
```
from math import sin

def create_matrix(n, a1, a2, a3):
    matrix = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
```

```
        for j in range(n):
            if i == j:
                matrix[i][j] = a1
            elif i == j + 1 or i == j - 1:
                matrix[i][j] = a2
            elif i == j + 2 or i == j - 2:
                matrix[i][j] = a3
    return matrix

def create_vector(n):
    vector = []
    for i in range(n):
        vector.append(sin(i*5))
    return vector
```

3 Wektor residuum

Ważnym elementem algorytmów iteracyjnych (np. Jacobiego i Gaussa-Seidla) jest określenie w której iteracji algorytm powinien się zatrzymać. W tym celu najczęściej korzysta się z tzw. wektora residuum, który dla k – tej iteracji przyjmuje postać:

$$res^{(k)} = Ax^{(k)} - b$$

Badając normę euklidesową wektora residuum, możemy w każdej iteracji algorytmu obliczyć jaki błąd wnosi wektor $x^{(k)}$. Jeżeli algorytm zbiegnie się do dokładnego rozwiązania, residuum powinno być wektorem zerowym.

[2]:
```
def subtract_vectors(v1, v2):
    result = []
    for i in range(len(v1)):
        result.append(v1[i] - v2[i])
    return result

def residuum_vector(A, x, b):
    N = len(A)
    res = [0 for _ in range(N)]
    for i in range(N):
        for j in range(N):
            res[i] += A[i][j] * x[j]
    return subtract_vectors(res, b)
```

4 Zadania

4.1 Zadanie A - układ równań dla a1 = 10, a2 = a3 = -1 i N = 923

[3]:
```
N = 923
a1 = 10
a2 = a3 = -1
A = create_matrix(N, a1, a2, a3)
b = create_vector(N)
```

$$A = \begin{bmatrix} 10 & -1 & -1 & \dots & 0 \\ -1 & 10 & -1 & \dots & 0 \\ -1 & -1 & 10 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 10 \end{bmatrix}$$

4.2 Zadanie B - implementacja metod iteracyjnych

Funkcje pomocnicze:

[4]:
```
from math import sqrt

def copy_vector(vector):
    copy = []
    for value in vector:
        copy.append(value)
    return copy

def norm(vector):
    result = 0.0
    for value in vector:
        result += pow(value, 2)
    return sqrt(result)
```

Implementacja metody Jacobiego:

[5]:
```
import time

def jacobi(A, b):
    starting_time = time.time()
    N = len(A)
    residuum_value = pow(10, -9)
    x = [0 for _ in range(N)]
    x_tmp = [0 for _ in range(N)]
    k = 0

    while True:
        for i in range(N):
            result = b[i]
```

```
            for j in range(N):
                if i != j:
                    result -= A[i][j] * x[j]
            result /= A[i][i]
            x_tmp[i] = result
        x = copy_vector(x_tmp)
        res = residuum_vector(A, x, b)

        if norm(res) < residuum_value:
            break
        k += 1

    computation_time = time.time() - starting_time
    print("Jacobi's method")
    print('Time:', computation_time)
    print('Iterations:', k)

    return computation_time
```

Implementacja metody Gaussa-Seidla:

[6]:
```
def gauss_seidel(A, b):
    starting_time = time.time()
    N = len(A)
    x = [0 for _ in range(N)]
    residuum_value = pow(10, -9)
    k = 0

    while True:
        for i in range(N):
            result = b[i]
            for j in range(N):
                if i != j:
                    result -= A[i][j] * x[j]
            result /= A[i][i]
            x[i] = result
        res = residuum_vector(A, x, b)
        if norm(res) < residuum_value:
            break
        k += 1

    computation_time = time.time() - starting_time
    print("Gauss-Seidel's method")
    print("Time:", computation_time)
    print("Iterations:", k)

    return computation_time
```

Porównanie obu metod pod względem wydajności:

[7]:
```
jacobi(A, b)
gauss_seidel(A, b)
```

[7]:
```
Jacobi's method
Time: 3.4627633094787598
Iterations: 18

Gauss-Seidel's method
Time: 2.591036081314087
Iterations: 13
```

Łatwo zauważyć, iż metoda Gaussa-Seidla potrzebowała mniej iteracji niż metoda Jacobiego, aby osiągnąć zadaną dokładaność przybliżenia wyniku rzeczywistego. Spowodowane jest to tym, że w odróżnieniu od metody Jacobiego, metoda Gaussa-Seidla uwzględnia dotychczas wyznaczone elementy wektora wynikowego w kolejnych iteracjach.

4.3 Zadanie C - układ równań dla a1 = 3, a2 = a3 = -1 i N = 923

[8]:
```
a1 = 3
C = create_matrix(N, a1, a2, a3)
jacobi(C, b)
gauss_seidel(C, b)
```

$$A = \begin{bmatrix} 3 & -1 & -1 & \dots & 0 \\ -1 & 3 & -1 & \dots & 0 \\ -1 & -1 & 3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 3 \end{bmatrix}$$

Próba obliczenia układu równań Cx = b kończyła się błędem OverflowError: ('Result too large') (podczas liczenia normy wektora residuum; odpowiednio po 1264 iteracjach metodą Jacobiego i 521 metodą Gaussa-Seidla). Można z tego wywnioskować, iż metody iteracyjne dla takich wartości nie zbiegają się.

4.4 Zadanie D - metoda faktoryzacji LU

Funkcje pomocnicze:

[9]:
```
def copy_matrix(matrix):
    copy = []
    for row in matrix:
        copied_row = []
        for value in row:
            copied_row.append(value)
        copy.append(copied_row)
    return copy
```

```
def create_diagonal_matrix(vector):
    n = len(vector)
    matrix = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        matrix[i][i] = vector[i]
    return matrix
```

Implementacja metody faktoryzacji LU:

[10]:
```
def lu_factorization(A, b):
    starting_time = time.time()
    N = len(A)
    x = [1 for _ in range(N)]
    y = [0 for _ in range(N)]
    U = copy_matrix(A)              # Upper Triangular Matrix
    L = create_diagonal_matrix(x)   # Lower Triangular Matrix

    # creating matrices L and U; A = L * U
    for i in range(N):
        for j in range(i+1, N):
            L[j][i] = U[j][i] / U[i][i]
            for k in range(i, N):
                U[j][k] = U[j][k] - L[j][i]*U[i][k]

    # solving L * y = b
    for i in range(N):
        result = b[i]
        for j in range(i):
            result -= L[i][j] * y[j]
        y[i] = result / L[i][i]

    # solving U * x = y
    for i in range(N-1, -1, -1):
        result = y[i]
        for j in range(i+1, N):
            result -= U[i][j] * x[j]
        x[i] = result / U[i][i]

    res = residuum_vector(A, x, b)

    computation_time = time.time() - starting_time
    print("LU Factorization:")
    print("Time:", computation_time)
    print("Norm(res):", norm(res))

    return computation_time
```

Zastosowanie metody faktoryzacji LU do przypadku C:

[11]:
```
lu_factorization(C, b)
```

[11]:
```
LU Factorization:
Time: 38.319488763809204
Norm(res): 4.5378182421815034e-13
```

W zadanym przypadku norma z residuum wynosi 4.5378182421815034e-13. Jest to wynik bardzo bliski 0, co oznacza wysoką dokładaność obliczeń.

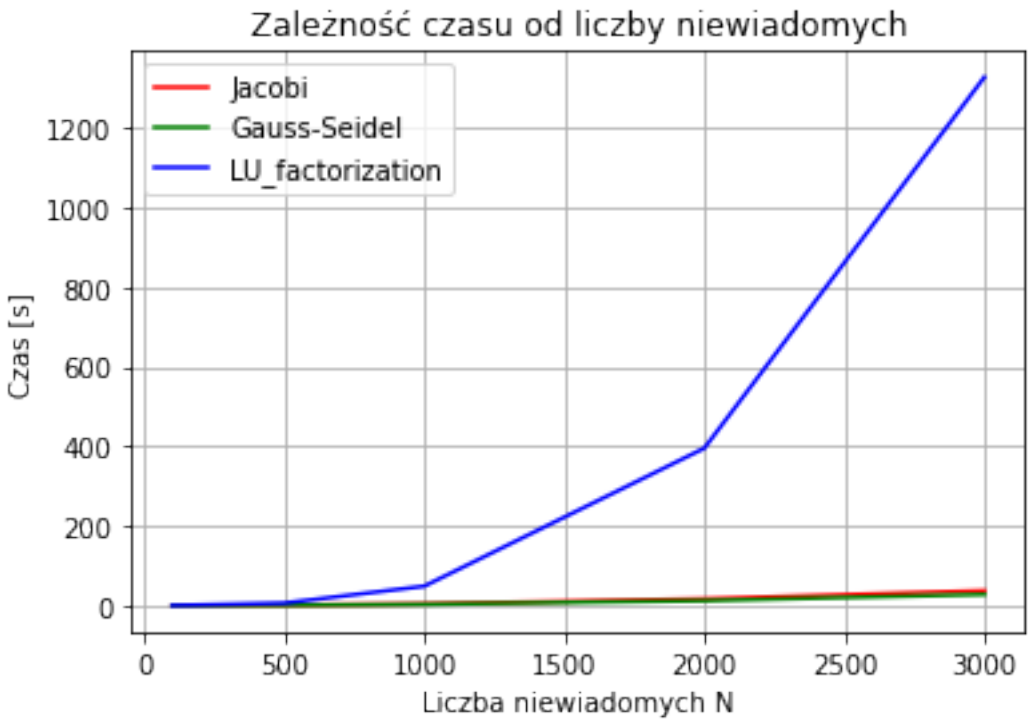
4.5 Zadanie E - porównanie zależności czasu trwania poszczególnych algorytmów od liczby niewiadomych N = {100, 500, 1000, 2000, 3000}

[12]:
```
from matplotlib import pyplot

a1 = 10
N = [100, 500, 1000, 2000, 3000]
jacobi_time = []
gauss_seidel_time = []
lu_factorization_time = []

for n in N:
    A = create_matrix(n, a1, a2, a3)
    b = create_vector(n)
    jacobi_time.append(jacobi(A, b))
    gauss_seidel_time.append(gauss_seidel(A, b))
    lu_factorization_time.append(lu_factorization(A, b))

pyplot.plot(N, jacobi_time, label="Jacobi", color="red")
pyplot.plot(N, gauss_seidel_time, label="Gauss-Seidel", color="green")
pyplot.plot(N, lu_factorization_time, label="LU_factorization", color="blue")
pyplot.legend()
pyplot.grid(True)
pyplot.ylabel('Czas [s]')
pyplot.xlabel('Liczba niewiadomych N')
pyplot.title('Zależność czasu od liczby niewiadomych')
pyplot.show()
```



Na podstawie wykresu widać, że wraz ze wzrostem liczby niewiadomych, wzrasta również czas wykonywanych obliczeń. Faktoryzacja LU jest metodą najdokładniejszą, ale również najbardziej czasochłonną. Metody iteracyjne nie są tak dokładne, lecz wykonują się znacząco szybciej. Z powyższych obserwacji wynika, że metody iteracyjne zdecydowanie bardziej nadają się do rozwiązywania układów równań z dużą liczbą niewiadomych. Warto mieć jednak na uwadzę, iż metody te mogą się nie zbiegać i wtedy konieczne będzie zastoswanie metody bezpośredniej - faktoryzacji LU.