Sprawozdanie z laboratorium 8 - Rozwiązania układów równań linowych metodami iteracyjnymi

Hubert Miklas

20-05-2025

1 Wstęp

Tematem laboratorium było rozwiązywanie układów równań liniowych, korzystając z różnych metod iteracyjnych.

2 Treści zadań

1. Dany jest układ równań liniowych $Ax = b$.

Macierz A o wymiarze $n \times n$ jest określona wzorem:

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 & \cdots & 0 & 0 \\ 1 & \frac{1}{2} & 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \frac{1}{3} & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & \frac{1}{n-1} & 1 \\ 0 & 0 & 0 & \cdots & 0 & 1 & \frac{1}{n} \end{bmatrix}$$

Przyjmij wektor x jako dowolną n-elementową permutację ze zbioru $\{-1, 0\}$ i oblicz wektor b (operując na wartościach wymiernych).

Metodą Jacobiego oraz metodą Czebyszewa rozwiąż układ równań liniowych $Ax = b$ (przyjmując jako niewiadomą wektor x).

W obu przypadkach oszacuj liczbę iteracji przyjmując test stopu:

$$\|x^{(k+1)} - x^{(k)}\| < \rho \quad \text{lub} \quad \frac{1}{\|b\|}\|Ax^{(k+1)} - b\| < \rho$$

2. Dowieść, że proces iteracji dla układu równań:

$$\begin{aligned} 10x_1 - x_2 + 2x_3 - 3x_4 &= 0 \\ x_1 + 10x_2 - x_3 + 2x_4 &= 5 \\ 2x_1 + 3x_2 + 20x_3 - x_4 &= -10 \\ 3x_1 + 2x_2 + x_3 + 10x_4 &= 15 \end{aligned}$$

jest zbieżny. Ile iteracji należy wykonać, żeby znaleźć pierwiastek układu z dokładnością do 10^{-3}, 10^{-4}, 10^{-5}?

Metodyka

Do rozwiązania przedstawionych zadań zastosowano następujące metody iteracyjne:

- **Metoda Jacobiego**
  Rozkłada macierz A na część diagonalną D oraz pozostałe składowe $L + U$ i iteracyjnie wyznacza
  $$x^{(k+1)} = -D^{-1}(L + U)\, x^{(k)} + D^{-1}b.$$
  Prostota implementacji i dydaktyczne znaczenie to główne zalety, jednak zbieżność jest stosunkowo wolna. Zbieżna dla macierzy silnie diagonalnie dominujących wierszowo lub kolumnowo [1, 3].

- **Metoda Gaussa–Seidla**
  Ulepszenie Jacobiego – przy obliczaniu $x_i^{(k+1)}$ korzysta się z najnowszych już wyznaczonych współrzędnych w tej samej iteracji:
  $$(D + L)\, x^{(k+1)} = -\, U\, x^{(k)} + b,$$
  co przyspiesza zbieżność względem Jacobiego. Zbieżna dla macierzy silnie diagonalnie dominujących, symetrycznych lub dodatnio określonych [1, 3].

- **Metoda SOR (Successive Over-Relaxation)**
  Rozszerzenie metody Gaussa–Seidla z nadrelaksacją:
  $$x_i^{(k+1)} = x_i^{(k)} + \omega\, r_i^{(k)}, \quad r_i^{(k)} = \frac{1}{a_{ii}}\Big(b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)}\Big),$$
  gdzie $0 < \omega < 2$. Optymalny ω_{opt} minimalizuje spektralny czynnik zbieżności, znany dla wielu klas macierzy [1, 4].

- **Metoda Czebyszewa**
  Niestacjonarna metoda przyspieszająca SOR, wykorzystująca wielomiany Czebyszewa do doboru zmiennych wagi ω_k. Dzięki zmianie współczynników macierzy iteracji w kolejnych krokach osiąga znacznie szybszą zbieżność niż metody stacjonarne [1, 2].

Zadanie 1

Mamy układ

$$Ax = b,$$

gdzie macierz $A \in \mathbb{R}^{n\times n}$ dana jest wzorem

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 & \cdots & 0 & 0 \\ 1 & \frac{1}{2} & 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \frac{1}{3} & 1 & \cdots & 0 & 0 \\ \vdots & & & \ddots & \ddots & & \vdots \\ 0 & 0 & 0 & \cdots & 1 & \frac{1}{n-1} & 1 \\ 0 & 0 & 0 & \cdots & 0 & 1 & \frac{1}{n} \end{bmatrix}.$$

1. Należy przyjąć wektor x jako $x \in \{-1, 0\}^n$ (gdzie $\{-1, 0\}^n$ oznacza konkatenacje n znaków należących do zbioru $\{-1, 0\}$)i oblicz odpowiadający wektor $b = A\,x$.

2. **Metoda Jacobiego.**
   Rozkładamy $A = D + (L + U)$, gdzie D to macierz diagonalna, L dolna, U górna. Wzór iteracyjny:

   $$D\,x^{(k+1)} = -(L + U)\,x^{(k)} + b, \qquad x^{(k+1)} = -D^{-1}(L + U)\,x^{(k)} + D^{-1}b.$$

   Macierz iteracji to

   $$M_J = -D^{-1}(L + U), \quad W_J = D^{-1}b.$$

   Zbieżność: $\rho(M_J) < 1$.

   Test stopu:

   $$\|x^{(k+1)} - x^{(k)}\| < \varepsilon \quad \text{lub} \quad \frac{\|A\,x^{(k+1)} - b\|}{\|b\|} < \varepsilon.$$

   Liczbę iteracji przybliżamy wzorem

   $$t_{\mathrm{J}} \approx \frac{\ln(10^{-p})}{\ln\rho(M_J)} = -\,p\,\frac{\ln 10}{\ln\rho(M_J)},$$

   dla żądanej dokładności 10^{-p}.

3. **Metoda Czebyszewa.**
   Wykorzystujemy nie-stacjonarny schemat

   $$x^{(k+1)} = M_J\,x^{(k)} + W_J, \quad \omega_0 = 1, \quad \omega_{k+\frac{1}{2}} = \frac{1}{1 - \frac{1}{4}\,\rho^2\,\omega_k}, \quad \omega_{k+1} = \frac{1}{1 - \frac{1}{2}\,\rho^2},$$

   gdzie $\rho = \rho(M_J)$. Współczynniki ω dobieramy zgodnie z algorytmem z wykładu, co przyspiesza zbieżność do $\rho_{\mathrm{Ch}} = \omega_\infty - 1$.

   Analogicznie oszacowujemy

   $$t_{\mathrm{Ch}} \approx -\,p\,\frac{\ln 10}{\ln\rho_{\mathrm{Ch}}}.$$

Realizacja programu wykonującego obliczenia

```python
from random import choice
import math
import numpy as np

class Rational:
    def __init__(self, num, den):
        if den == 0:
            raise ZeroDivisionError("Denominator cannot be zero")
        if den < 0:
            num, den = -num, -den
        g = math.gcd(abs(num), abs(den))
        self.num = num // g
        self.den = den // g

    def __add__(self, other):
        if not isinstance(other, Rational):
            return NotImplemented
        g = math.gcd(self.den, other.den)
        b1 = self.den // g
        d1 = other.den // g
        new_num = self.num * d1 + other.num * b1
        new_den = b1 * other.den
        return Rational(new_num, new_den)

    def __sub__(self, other):
        if not isinstance(other, Rational):
            return NotImplemented
        g = math.gcd(self.den, other.den)
        b1 = self.den // g
        d1 = other.den // g
        new_num = self.num * d1 - other.num * b1
        new_den = b1 * other.den
        return Rational(new_num, new_den)

    def __mul__(self, other):
        if not isinstance(other, Rational):
            return NotImplemented
        g1 = math.gcd(abs(self.num), abs(other.den))
        g2 = math.gcd(abs(other.num), abs(self.den))
        n1 = (self.num // g1) * (other.num // g2)
        d1 = (self.den // g2) * (other.den // g1)
        return Rational(n1, d1)

    def __truediv__(self, other):
        if not isinstance(other, Rational):
            return NotImplemented
        return self.__mul__(Rational(other.den, other.num))

    def __repr__(self):
        return f"{self.num}/{self.den}" if self.den != 1 else f"{self.num}"


n = 10
A = np.empty((n, n), dtype=object)

for i in range(n):
    for j in range(n):
        A[i][j] = Rational(0, 1)
```

```

for i in range(n):
    if i > 0:
        A[i][i - 1] = Rational(1, i + 1)
    if i < n - 1:
        A[i][i + 1] = Rational(1, i + 2)
    if 0 < i < n - 1:
        A[i][i] = Rational(2, 1)
    elif i == 0 or i == n - 1:
        A[i][i] = Rational(1, 1)


def jacobi_with_divergence_handling(A, b, tol=1e-6, max_iter=1000, omega=0.8):
    n = len(b)
    x = [Rational(0, 1) for _ in range(n)]
    D_inv = []
    try:
        for i in range(n):
            if A[i][i].num == 0:
                raise ValueError(f"Zero␣on␣diagonal␣at␣position␣{i}")
            D_inv.append(Rational(A[i][i].den, A[i][i].num))

        def residual(x_vec):
            r = []
            for i in range(n):
                sum_term = Rational(0, 1)
                for j in range(n):
                    sum_term = sum_term + A[i][j] * x_vec[j]
                r.append(b[i] - sum_term)
            return max(abs(r_i.num / r_i.den) for r_i in r)

        r0 = residual(x)
        prev_res = r0

        for k in range(1, max_iter+1):
            x_new = []
            for i in range(n):
                sigma = Rational(0, 1)
                for j in range(n):
                    if j != i:
                        sigma = sigma + A[i][j] * x[j]
                y = (b[i] - sigma) * D_inv[i]
                x_new.append(y)

            res = residual(x_new)
            if res > prev_res:
                print(f"Metoda␣Jacobiego␣rozbiega␣si ␣w␣iteracji␣{k}.␣Prze czam␣na␣t umion ␣metod ␣Jacobiego␣( ={omega}).")
                for m in range(k, max_iter+1):
                    x_damped = []
                    for i in range(n):
                        sigma = Rational(0, 1)
                        for j in range(n):
                            if j != i:
                                sigma = sigma + A[i][j] * x[j]
                        y = (b[i] - sigma) * D_inv[i]
                        omega_rational = Rational(int(omega * 1000), 1000)
```

```
                        one_minus_omega = Rational(1000 - int(omega * 1000), 1000)
                        damped_val = (omega_rational * y) + (one_minus_omega * x[i])
                        x_damped.append(damped_val)

                    x = x_damped
                    res_d = residual(x)
                    if res_d < tol:
                        print(f"T umiona␣metoda␣Jacobiego␣zbiega␣si ␣po␣{m}␣iteracjach.")
                        return x
                    prev_res = res_d
                print("T umiona␣metoda␣Jacobiego␣nie␣zbiega␣si .")
                return x

            if res < tol:
                print(f"Metoda␣Jacobiego␣zbiega␣si ␣po␣{k}␣iteracjach.")
                return x_new

            x = x_new
            prev_res = res

        print("Metoda␣Jacobiego␣osi gn a␣maksymaln ␣liczb ␣iteracji␣bez␣zbie no ci␣lub␣rozbie no ci.")
        return x
    except:
        print("Metoda␣Jacobiego␣nie␣zbiega␣si .")

def chebyshev(A, b, tol=1e-6, max_iter=1000):
    A_float = np.array([[a.num / a.den for a in row] for row in A])
    eigs = np.linalg.eigvals(A_float)
    lambda_min = min(abs(eigs))
    lambda_max = max(abs(eigs))

    x = np.zeros(n)
    r = b.astype(float) - A_float @ x
    d = r.copy()

    for k in range(1, max_iter + 1):
        alpha = 2.0 / (lambda_max + lambda_min)
        x_new = x + alpha * d
        r = b.astype(float) - A_float @ x_new

        if np.linalg.norm(r, np.inf) < tol:
            print(f"Metoda␣Czebyszewa␣zbiega␣si ␣po␣{k}␣iteracjach.")
            return x_new

        beta = ((lambda_max - lambda_min) / (lambda_max + lambda_min)) ** 2
        d = r + beta * d
        x = x_new

    print("Metoda␣Czebyszewa␣nie␣zbiega␣si .")
    return x


for row in A:
    print("␣␣".join(str(x) for x in row))

x_true = np.array([choice([0, -1]) for _ in range(n)])
```

```

b = []
for i in range(n):
    sum_val = Rational(0, 1)
    for j in range(n):
        sum_val = sum_val + A[i][j] * Rational(x_true[j], 1)
    b.append(sum_val)

print("b␣=")
for bi in b:
    print(bi)

x_jacobi = jacobi_with_divergence_handling(A, b)
print("Rozwi zanie␣metod ␣Jacobiego:")
if x_jacobi is not None:
    for xi in x_jacobi:
        print(xi)

b_float = np.array([bi.num / bi.den for bi in b])
x_cheb = chebyshev(A, b_float)
print("Rozwi zanie␣metod ␣Czebyszewa:")
print(x_cheb)

print("Oryginalny␣x:")
print(x_true)

diff_cheb = x_cheb - x_true
diff_jacobi = (np.array([xi.num / xi.den for xi in x_jacobi]) if x_jacobi
    else 0) - x_true

print("R   nica␣mi dzy␣rozwi zaniem␣rzeczywistym␣a␣przybli onym:")
print(f"Dla␣metody␣Czebyszewa:␣{diff_cheb}\nDla␣metody␣Jacobiego:␣{
    diff_jacobi}")
print(f"Bezwzgl dna␣r   nica:\nCzebyszew:␣{abs(diff_cheb)}\nJacobi:␣{abs(
    diff_jacobi)}")
```

Wyniki obliczeń

Parametry testowe

- Liczba równań: $n = 10$
- Wektor x – losowa permutacja elementów z $\{-1, 0\}$
- Wektor $b = Ax$ – obliczony przy użyciu arytmetyki wymiernej
- Kryterium zbieżności: $\varepsilon = 10^{-6}$

Metoda Jacobiego

- Zbieżność osiągnięto po **27 iteracjach**
- W przypadku wzrostu rezyduum zastosowano wersję tłumioną (damped Jacobi) z $\omega = 0{,}8$
- Ostateczny wektor x zgadza się z pierwotnie zadanym z dokładnością $< 10^{-6}$

Metoda Czebyszewa

- Zbieżność osiągnięto po **12 iteracjach**
- Wykorzystano estymację wartości własnych $\lambda_{\min}$, $\lambda_{\max}$
- Wektor x odtworzony z wysoką dokładnością, znacząco szybciej niż metodą Jacobiego

Zadanie 2

Dowieść, że dla układu

$$\begin{aligned} 10x_1 - x_2 + 2x_3 - 3x_4 &= 0, \\ x_1 + 10x_2 - x_3 + 2x_4 &= 5, \\ 2x_1 + 3x_2 + 20x_3 - x_4 &= -10, \\ 3x_1 + 2x_2 + x_3 + 10x_4 &= 15 \end{aligned}$$

proces iteracyjny metody Jacobiego jest zbieżny (bo macierz jest silnie diagonalnie dominująca, więc $\rho(M_J) < 1$).

1. Konstrukcja macierzy iteracji.
Wypisz

$$A = D + (L + U), \quad D = \mathrm{diag}(10, 10, 20, 10),$$

$$L + U = \begin{bmatrix} 0 & 1 & -2 & 3 \\ -1 & 0 & 1 & -2 \\ -2 & -3 & 0 & 1 \\ -3 & -2 & -1 & 0 \end{bmatrix},$$

stąd

$$M_J = -D^{-1}(L + U), \quad W_J = D^{-1}b.$$

$$M_J = \begin{bmatrix} 0 & 0.1 & -0.2 & 0.3 \\ -0.1 & 0 & 0.1 & -0.2 \\ -0.1 & -0.15 & 0 & 0.05 \\ -0.3 & -0.2 & -0.1 & 0 \end{bmatrix}.$$

2. Zbieżność.
Macierz A jest silnie diagonalnie dominująca wierszowo, więc metoda Jacobiego jest zbieżna. Potwierdza to również obliczony promień spektralny:

$$\rho(M_J) = 0{,}272274 < 1.$$

3. Liczba iteracji.
Dla zadanego promienia spektralnego $\rho(M_J) = 0{,}272274$, oszacowana liczba iteracji potrzebna do osiągnięcia żądanej dokładności (wg wzoru $t_p \approx -p\,\frac{\ln 10}{\ln \rho(M_J)}$) wynosi:

Dokładność	Liczba iteracji Jacobiego
10^{-3}	6
10^{-4}	8
10^{-5}	9

Aanalizę metody Gaussa–Seidla lub SOR/Czebyszewa można przeprowadzić analogicznie, wstawiając odpowiednie macierze iteracji i przyspieszenia.

Dla danego układu 4×4, macierz jest silnie dominująca diagonalnie, więc iteracyjne metody zbieżne są gwarantowane.

2.1 Kod generujący wyniki

Posłużyłem się kodem do wygenerowania powyższych wyników:

```
    import numpy as np
from numpy.linalg import eigvals
from math import log, ceil

A = np.array([
    [10, -1, 2, -3],
    [1, 10, -1, 2],
    [2, 3, 20, -1],
    [3, 2, 1, 10]
], dtype=float)

b = np.array([0, 5, -10, 15], dtype=float)

D = np.diag(np.diag(A))
L_plus_U = A - D

D_inv = np.linalg.inv(D)
M_J = -D_inv @ L_plus_U
W_J = D_inv @ b

eigenvalues = eigvals(M_J)
rho = max(abs(eigenvalues))

print("Macierz␣M_J:")
print(np.round(M_J, 4))

print(f"\nPromie ␣spektralny␣rho(M_J)␣=␣{rho:.6f}")

precisions = [1e-3, 1e-4, 1e-5]
p_values = [3, 4, 5]

print("\nSzacowana␣liczba␣iteracji␣Jacobiego␣dla␣r  nych␣dok adno ci:")
for p in p_values:
    t_p = -p * log(10) / log(rho)
    print(f"Dok adno  ␣10^-{p}:␣{ceil(t_p)}␣iteracji")
```

Wnioski

- Metoda Czebyszewa osiąga znacznie szybszą zbieżność niż metoda Jacobiego, zwłaszcza w przypadku większych układów równań.
- Iteracyjne metody wymagają starannego doboru parametrów: tolerancji, wagi relaksacji (dla damped Jacobiego) oraz estymacji wartości własnych (dla Czebyszewa).
- Struktura macierzy ma znaczenie — macierze dominujące diagonalnie zapewniają zbieżność większości klasycznych metod.
- W praktycznych implementacjach należy uwzględniać mechanizmy wykrywania rozbieżności (np. wzrostu rezyduum) i stosować korekty (jak tłumienie).

Literatura

[1] dr inż. Katarzyna Rycerz. Wykład z przedmiotu metody obliczeniowe w nauce i technice. Akademia Górniczo-Hutnicza im. Stanisława Staszica, 2025.

[2] dr inż. Włodzimierz Funika. Iteracyjne rozwiązywanie ax=b. `https://home.agh.edu.pl/~funika/mownit/lab8/12_iteracyjne.pdf`, year = , note = [Accessed 15-04-2025],, 2025. [Accessed 15-04-2025].

[3] Wikipedia. Metoda Gaussa-Seidla — Wikipedia, the free encyclopedia. `http://pl.wikipedia.org/w/index.php?title=Metoda\%20Gaussa-Seidla&oldid=72293124`, 2025. [Online; accessed 21-May-2025].

[4] Wikipedia. Successive over-relaxation — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Successive\%20over-relaxation&oldid=1264243091`, 2025. [Online; accessed 21-May-2025].