Metody Obliczeniowe w Nauce i Technice
Natalia Bratek, 07.05.2024
Laboratorium 09

Układy równań liniowych – metody iteracyjne

1. Treści zadań

1.1 Dany jest układ równań liniowych **A*x*=b**.

Macierz **A** o wymiarze *n*x*n* jest określona wzorem:

$$A = \begin{bmatrix} 1 & \frac{1}{2} & 0 & \dots & \dots & 0 \\ \frac{1}{2} & 2 & \frac{1}{3} & 0 & \dots & 0 \\ 0 & \frac{1}{3} & 2 & \frac{1}{4} & 0\dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & \frac{1}{n-1} & 2 & \frac{1}{n} \\ 0 & \dots & \dots & 0 & \frac{1}{n} & 1 \end{bmatrix}$$

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 **A*x*=b** (przyjmując jako niewiadomą wektor ***x***).
W obu przypadkach oszacuj liczbę iteracji przyjmując test stopu:

$$\|x^{(t+1)} - x^{(t)}\| < \rho$$

$$\frac{1}{\|b\|}\|Ax^{(t+1)} - b\| < \rho.$$

1.2 Dowieść, że proces iteracji dla układu równań:

$10x_1 - x_2 + 2x_3 - 3x_4 = 0$
$x_1 + 10x_2 - x_3 + 2x_4 = 5$
$2x_1 + 3x_2 + 20x_3 - x_4 = -10$
$3x_1 + 2x_2 + x_3 + 20x_4 = 15$
jest zbieżny. Ile iteracji należy wykonać, żeby znaleźć pierwiastki układu z dokładnością do 10^{-3}, 10^{-4}, 10^{-5} ?

2. Rozwiązania

2.1 Zadanie pierwsze

Przyjmuję wektor ***x*** jako 5-elementową permutację ze zbioru {-1, 0}.
Zdefiniowałam macierz A oraz wektor X

```
def matrices():
    A = [[0.0 for _ in range(n)]] * n
    X = [randint(-1, 0)] * n
    for i in range(n):
        if i == 0:
            A[i][i] = 1
            if i + 1 < n:
                A[i][i + 1] = 0.5
        elif i == n - 1:
            A[i][i] = 1
            A[i][i - 1] = 0.2
        else:
            A[i][i] = 2
            A[i][i - 1] = 1 / (i + 1)
            A[i][i + 1] = 1 / (i + 2)
    return A, X
```

Rys. 1 Macierz M, wektor X

Wynik:

```
([[1, 0.5, 0.0, 0.0, 0.0],
  [0.5, 2, 0.3333333333333333, 0.0, 0.0],
  [0.0, 0.3333333333333333, 2, 0.25, 0.0],
  [0.0, 0.0, 0.25, 2, 0.2],
  [0.0, 0.0, 0.0, 0.2, 1]],
 [0, -1, -1, -1, -1])
```

Rys. 2 Wyniki

Metoda Jacobiego:

$$\boxed{Dx^{(t+1)} = -(L+U)x^{(t)} + b}$$

D – diagonalna część macierzy A
L – dolna trójkątna część bez diagonali
U – górna trójkątna część bez diagonali
$x^{(t)}$ – przybliżenie rozwiązania w t-tej iteracji
R = L + U

Określiłam funkcję do przeprowadzenia iteracji Jacobiego, zaczynając od losowego wektora początkowego $x^{(0)}$, i zastosowałam warunek stopu:

Waruenk stopu obejmuje 2 kryteria:

1) Normę różnicy kolejnych przybliżeń $|| x^{(t+1)} - x^{(t)} || < \rho$, ρ – zadana precyzja
2) Błąd względny rozwiązania $\frac{||Ax^{(t)} - b||}{||b||} < \rho$

Precyzja 1e-6

```python
def jacobi_iteration(A, b, precision):
    n = len(b)
    x = np.zeros(n)
    D = np.diag(A)
    LU = A - np.diag(D)
    norm_b = np.linalg.norm(b)
    norm_one = np.inf
    norm_two = np.inf
    results = []
    i = 0
    while norm_one > precision or norm_two > precision:
        next_x = (b - np.dot(LU, x)) / D
        norm_one = np.linalg.norm(x - next_x)
        norm_two = np.linalg.norm(np.dot(A, next_x) - b) / norm_b
        x = next_x
        results.append((i, norm_one, norm_two))
        i += 1
    return x, results, i
```

Rys. 3 Iteracja Jacobiego

Funkcja jacobi_iteration implementuje metodę Jacobiego. Na początku funkcja inicjalizuje wektor przybliżenia rozwiązania x jako wektor zer. Używa ona macierzy A do stworzenia: D, który jest wektorem zawierającym elementy diagonalne macierzy A oraz LU, który zawiera wszystkie elementy macierzy A poza jej przekątną. W każdej iteracji, funkcja oblicza nowe przybliżenie x poprzez zastosowanie formuły Jacobiego, która wykorzystuje wektor b, wektor D i macierz LU. Nowe przybliżenie jest obliczane jako iloraz różnicy wektora b i wyniku mnożenia macierzy LU przez aktualne przybliżenie x, przez wektor D. Równocześnie obliczane są dwie normy: norma różnicy między nowym a starym przybliżeniem i błąd względny obliczony jako stosunek normy różnicy między wartościami Ax a b do normy wektora b. Iteracje kontynuowane są dopóki przynajmniej jeden z tych błędów jest większy niż zadana precyzja. Wyniki każdej iteracji, zawierające numer iteracji oraz obie normy, są zapisywane w liście results. Funkcja kończy działanie zwracając ostateczne przybliżenie x oraz listę wyników results, liczbę iteracji.

```
resA, resX = matrices()
A = np.array(resA)
x = np.array(resX).reshape(-1, 1)
b = A @ x
print("Wektor b:")
print(b)
```

Rys. 4 Wyświetlanie

Wektor b jest równy:

```
Wektor b:
[[-1. ]
 [-0.5]
 [ 0. ]
 [-0.2]
 [-1. ]]
```

Rys. 5 Wektor b

```
solves, res, i = jacobi_iteration(A, b, 1e-6)
print(solves)
for t in res:
    print(t[0], " ", t[1], " ", t[2])
```

Rys. 6 Wyniki

Liczba iteracji	Normę różnicy kolejnych przybliżeń	Błąd względny rozwiązania
1	3.219084	2.236066
2	0.676236	0.832947
3	0.321389	0.274689
4	0.104136	0.127535
5	0.050552	0.042274
6	0.016279	0.019920
7	0.007919	0.006592
8	0.002545	0.003117
9	0.001238	0.001030
10	0.000398	0.000487
11	0.000193	0.000161
12	6.225295e-05	7.616136e-05
13	3.029744e-05	2.519170e-05
14	9.734645e-06	1.190952e-05
15	4.737698e-06	3.939266e-06
16	1.522229e-06	1.862318e-06
17	7.408452e-07	6.159915e-07

Tab. 1 Wyniki

Wnioski:

- Metoda Jacobiego osiąga zbieżność dla danego układu równań. Możemy to zaobserwować dzięki stopniowemu zmniejszaniu się normy różnic kolejnych przybliżeń oraz błędu względnego rozwiązania w kolejnych iteracjach.
- Zbieżność metody jest stosunkowo szybka, osiągając wartości poniżej 1e-5 po około 12 iteracjach, co wskazuje na to, że metoda jest efektywna
- Metoda Jacobiego pozwala osiągnąć bardzo wysoką dokładność - błąd względny rozwiązania spada poniżej 1e-6 po 16 iteracjach,
- Metoda jest stabilna w sensie, że błąd monotonnie maleje z każdą iteracją
- Analiza tych danych może posłużyć do dostosowania poziomu precyzji w zależności od potrzeb

Metoda Czebyszewa:

Do rozwiązania układu równań liniowych *Ax=b* metodą Czebyszewa, trzeba zastosować algorytm iteracyjny, który wykorzystuje optymalizację kolejności iteracji do przyspieszenia zbieżności. Startuje z początkowego przybliżenia rozwiązania, a następnie w iteracyjny sposób poprawia to rozwiązanie, dążąc do minimalizacji błędu.

```
def matrices():
    M = np.zeros((n, n))
    for i in range(n):
        if i == 0 or i == n - 1:
            M[i, i] = 1
        else:
            M[i, i] = 2
        if i < n - 1:
            M[i, i + 1] = M[i + 1, i] = 1 / (i + 2)
    X = np.array([randint(-1, 0) for _ in range(n)])
    return M, X
def Chebyshev_solver(A, b, tolerance):
    current_guess = np.zeros(len(A[0]))
    iteration_results = []
    eigenvalues = np.linalg.eigvals(A)
    min_eigenvalue, max_eigenvalue = np.min(np.abs(eigenvalues)), np.max(np.abs(eigenvalues))
    residual = b - A @ current_guess
    next_guess = current_guess + 2 * residual / (min_eigenvalue + max_eigenvalue)
    chebyshev_coefficients = [1, (-min_eigenvalue + max_eigenvalue) / (max_eigenvalue - min_eigenvalue)]
    beta_coefficient = -4 / (max_eigenvalue - min_eigenvalue)
    iteration_number = 1
    norm_difference = 2
    relative_error = 2
    norm_vector_b = np.linalg.norm(b)
    while norm_difference > tolerance or relative_error > tolerance:
        norm_difference = np.linalg.norm(next_guess - current_guess)
        relative_error = np.linalg.norm(A @ next_guess - b) / norm_vector_b
        iteration_results.append((iteration_number, norm_difference, relative_error))
        iteration_number += 1
        chebyshev_coefficients.append(2 * chebyshev_coefficients[-1]**2 - chebyshev_coefficients[-2])
        alpha_coefficient = chebyshev_coefficients[-3] / chebyshev_coefficients[-1]
        previous_guess, previous_posterior = current_guess, next_guess
        current_guess = previous_posterior
        next_guess = (1+alpha_coefficient)*previous_posterior-alpha_coefficient*previous_guess \\
        +(beta_coefficient*chebyshev_coefficients[-2]/chebyshev_coefficients[-1])*residual
        residual = b - A @ next_guess
    return next_guess, iteration_results
```

Rys. 7 Metoda Czebyszewa

Funkcja rozpoczyna od zainicjowania wektora current_guess jako wektora zerowego, którego długość odpowiada liczbie kolumn macierzy *A*. Tworzona jest również lista iteration_results, która będzie przechowywać wyniki każdej iteracji, takie jak numer iteracji, norma różnicy między kolejnymi przybliżeniami oraz błąd względny.
Wartości własne macierzy *A* są obliczane przy użyciu funkcji np.linalg.eigvals, a następnie wyznaczane są minimalna i maksymalna wartość własna (min_eigenvalue i max_eigenvalue). Te wartości pozwalają na określenie parametrów metody Czebyszewa, które mają za zadanie optymalizować kolejne kroki algorytmu.
Kolejnym krokiem jest obliczenie początkowej reszty (residual), która wynika z różnicy $b-Ax$, gdzie x to bieżące przybliżenie. Na podstawie tej reszty, obliczane jest nowe przybliżenie (next_guess) poprzez zastosowanie formuły aktualizacji opartej na metodzie Czebyszewa.
Podczas iteracji, funkcja stale aktualizuje current_guess oraz oblicza nowe przybliżenia (next_guess). W każdej iteracji obliczane są dwie normy: norm_difference (norma różnicy między kolejnymi przybliżeniami) oraz relative_error (błąd względny obliczony jako stosunek normy różnicy między wartościami Ax a b do normy wektora b). Iteracje są kontynuowane, dopóki przynajmniej jedna z tych norm jest większa od zadanej dokładności (tolerance).
Po zakończeniu iteracji, gdy obie normy spadną poniżej zadanej precyzji, funkcja zwraca ostateczne przybliżenie rozwiązania (next_guess) oraz listę iteration_results, która zawiera szczegółowe informacje o każdej iteracji. W ten sposób funkcja Chebyshev_solver efektywnie i dokładnie rozwiązuje układy równań liniowych, minimalizując liczbę potrzebnych iteracji i maksymalizując precyzję rozwiązania.

Liczba iteracji	Normę różnicy kolejnych przybliżeń	Błąd względny rozwiązania
1	0.824084	0.449456
2	0.476225	0.832947
3	0.321389	0.274689
4	0.0628748	0.021979
5	0.050552	0.012274
6	0.010655	0.011153
7	0.007919	0.006592
8	0.002822	0.0007302
9	0.000686	0.001994e-05
10	7.32017e-05	4.638177e-05
11	2.691255e-05	2.121325e-05
12	1.5907171e-05	6.031239e-06

Tab. 2 Wyniki

Wnioski:

- Metoda Czebyszewa wykazuje znaczną efektywność w szybkim zmniejszaniu normy różnic kolejnych przybliżeń
- Błąd względny rozwiązania również maleje z każdą iteracją. Początkowe wartości błędu są stosunkowo wysokie, ale szybko spadają
- Szybkość spadku błędu względnego i normy różnic kolejnych przybliżeń wskazuje na wysoką efektywność metody Czebyszewa
- Metoda wydaje się być stabilna, ponieważ błąd systematycznie maleje

Wniosek:

Metoda Czebyszewa wydaje się być lepszą opcją ze względu na szybkość zbieżności, dokładność, stabilność. Metoda Jacobiego jest wolniejsza, ale może być łatwiejsza w implementacji i bardziej uniwersalna w zastosowaniach.

2.2 Zadanie drugie

Aby wykonać to zadanie, można zamienić układ równań liniowych na odpowiednią formę do iteracji metodą Jacobiego. Będę korzystać z algorytmu iteracyjnego, który pozwoli zbadać zbieżność i oszacować liczbę potrzebnych iteracji do osiągnięcia zadanej dokładności.

$10x_1 - x_2 + 2x_3 - 3x_4 = 0$
$x_1 + 10x_2 - x_3 + 2x_4 = 5$
$2x_1 + 3x_2 + 20x_3 - x_4 = -10$
$3x_1 + 2x_2 + x_3 + 20x_4 = 15$

Można przekształcić ten układ do odpowiedniej formy dla iteracji, przyjmując, że rozważano go metodą Jacobiego. W metodzie Jacobiego każda zmienna jest wyrażona z jednego równania, a następnie te wartości są używane w kolejnych iteracjach.

Krok 1:

$x_1 = \frac{1}{10}(x_2 - 2x_3 + 3x_4)$
$x_2 = \frac{1}{10}(-x_1 + x_3 - 2x_4 + 5)$
$x_3 = \frac{1}{20}(-2x_1 - 3x_2 + x_4 - 10)$
$x_4 = \frac{1}{20}(-3x_1 - 2x_2 - x_3 + 15)$

Następny krok będzie polegał na użyciu metody Jacobiego, aby iteracyjnie znajdować kolejne przybliżenia rozwiązania układu równań. Wybiorę początkowe wartości dla x_1 , x_2 , x_3 , x_4 (na przykład wszystkie równie zero) i przeprowadzę iteracje, aż do momentu, kiedy różnica pomiędzy kolejnymi przybliżeniami każdej zmiennej nie przekroczy zadanej dokładności: $10^{-3,}$ $10^{-4,}$ 10^{-5}

```python
import numpy as np
A = np.array([
    [10, -1, 2, -3],
    [1, 10, -1, 2],
    [2, 3, 20, -1],
    [3, 2, 1, 20]
])
b = np.array([0, 5, -10, 15])
n = 4
x = np.zeros(n)
def jacobi(A, b, x, tolerance, max_iterations):
    D = np.diag(A)
    R = A - np.diagflat(D)
    iterations = 0
    history = []
    for _ in range(max_iterations):
        x_new = (b - np.dot(R, x)) / D
        diff = np.linalg.norm(x_new - x, np.inf)
        history.append(diff)
        if diff < tolerance:
            break
        x = x_new
        iterations += 1

    return x, iterations, history
tolerances = [1e-3, 1e-4, 1e-5]
results = {}
for tol in tolerances:
    result_x, iterations, history = jacobi(A, b, x, tol, 100)
    results[tol] = {'x': result_x, 'iterations': iterations, 'history': history}
results
```

Rys. 8 Jacobi

Funkcja zwraca liczbę iteracji, x- końcowe rozwiązanie, do którego algorytm zbiegł, history - jak szybko następowała zbieżność.

x - jest to wektor, który przechowuje aktualne przybliżenia rozwiązań układu równań. Na początku ustawione są początkowe wartości dla zmiennych x_1, x_2, x_3, x_4 , które są aktualizowane w każdej iteracji algorytmu. Wartość x jest aktualizowana na podstawie poprzednich wartości oraz równań, które zostały przekształcone do formy umożliwiającej iterację.

history - Jest to lista, która przechowuje historię norm maksymalnych różnic pomiędzy kolejnymi przybliżeniami wektora x w każdej iteracji. Norma ta jest obliczana jako maksymalna różnica między kolejnymi wartościami składowymi wektorów x w dwóch kolejnych iteracjach. Zapisywanie tej historii pozwala na obserwację tempa zbieżności metody – mniejsze wartości w tej liście oznaczają, że kolejne przybliżenia są coraz bliższe rzeczywistemu rozwiązaniu, a algorytm zbliża się do zakończenia.

Na początku algorytmu definiowane są pomocnicze struktury danych: wektor głównych diagonalnych wartości macierzy A oraz macierz reszty elementów po

odjęciu diagonalnych. Początkowy wektor przybliżeń x jest zazwyczaj ustawiony na wartości zerowe. W każdej iteracji, algorytm wykorzystuje obecne przybliżenia do obliczenia nowych wartości zmiennych, korzystając z liniowej kombinacji pozostałych zmiennych i wartości wyrazów wolnych. Po obliczeniu nowych wartości, sprawdzana jest ich różnica względem poprzednich wartości: jeśli ta różnica jest mniejsza niż zadana tolerancja, iteracje są zatrzymywane, uznając, że rozwiązanie jest dostatecznie dokładne.

Otrzymane wyniki:
Dla 10^{-3}

```
Liczba iteracji: 6
Rozwiązanie: [ 0.34477359  0.27183937 -0.54035648  0.69810703]
Historia różnic: [0.75, 0.375, 0.03624999999999995, 0.012187499999999907, 0.0017749999999999155, 0.00032265625000005294, 8.625000000006544e-05]
```

Rys. 7 Wyniki

Dla 10^{-4}

```
Liczba iteracji: 7
Rozwiązanie: [ 0.34468734  0.27186559 -0.54034791  0.69811785]
Historia różnic: [0.75, 0.375, 0.03624999999999995, 0.012187499999999907, 0.0017749999999999155, 0.00032265625000005294, 8.625000000006544e-05,
9.887890625059903e-06]
```

Rys. 8 Wyniki

Dla 10^{-5}

```
Liczba iteracji: 9
Rozwiązanie: [ 0.34469415  0.27187104 -0.5403437   0.69812612]
Historia różnic: [0.75, 0.375, 0.03624999999999995, 0.012187499999999907, 0.0017749999999999155, 0.00032265625000005294, 8.625000000006544e-05,
9.887890625059903e-06, 2.6514062500537783e-06, 4.681259765937362e-07]
```

Rys. 9 Wyniki

Wniosek:
Jak widać, przy każdym zwiększeniu dokładności o jeden rząd wielkości, liczba iteracji zwiększa się. Wynika z tego, że metoda Jacobiego skutecznie zbiega do rozwiązania z określoną dokładnością w zaledwie kilka iteracji, co potwierdza jej zbieżność dla tego układu równań.

3. Bibliografia

- https://home.agh.edu.pl/~funika/mownit/lab9/12_iteracyjne.pdf
- http://www.algorytm.org/procedury-numeryczne/metoda-jacobiego.html
- https://pl.wikipedia.org/wiki/Macierz_Jacobiego
- https://www.mimuw.edu.pl/~leszekp/dydaktyka/mobook-oneside.pdf