Metody rozwiązywania układów równań liniowych

Piotr Pesta, 184531

Maj 2022

1. Wstęp

Celem projektu była implementacja i porównanie działania trzech różnych metod rozwiązywania układów równań liniowych. W projekcie zrealizowałem dwie metody iteracyjne oraz jedną metodę bezpośrednią. Były to odpowiednio: metoda Jacobiego oraz metoda Gaussa-Seidla (iteracyjne) i metoda faktoryzacji LU (bezpośrednia). Zadanie zreazliowane zostało w języku Python z użyciem bibliotek math, time, matplotlib oraz biblioteki Numpy, która została wykorzystana jedynie do porównania efektywności mojej implementacji metod z implementacją biblioteczną.

Wszystkie funkcje doyczące działań na macierzach, takie jak ich mnożenie, mnożenie przez skalar, dodawanie i odejmowanie oraz metody podstawiania w przód i w tył znajdują się w module Functions.py. Implementacje metod będących tematem projetku, znajdują się w modułach o nazwach: Jacobi.py, Gauss-Seidel.py oraz LU.py.

2. Zadanie A

Dane dla zadania A (nr indeksu: 184531):

- $a_1 = 10$
- $a_2 = a_3 = -1$
- $N = 931$

Wygenerowana macierz systemowa $\mathbf{A}$ (o rozmiarze 931×931):

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

Wektor $\mathbf{b}$ określony jest wzorem:

$$\boldsymbol{b}_i = sin(i \cdot (f + 1))$$

Gdzie $f = 4$, dla numeru indeksu 184531.

Za generowanie macierzy pasmowej oraz wektora współczynników $\mathbf{b}$ odpowiadają odpowiednie funckje w module Functions.py

3. Zadanie B

Implementacje metod iteracyjnych znajdują się w modułach o nazwach: Jacobi.py oraz Gauss-Seidel.py. Metody iteracyjne nie dają nam dokładnego wyniku, a przybliżenie, którego dokładność specyfikujemy poprzez podanie parametru epsilon. Sprawia to, że mają one lepszą złożoność obliczeniową niż metody bezpośrednie - $O(n^2)$ zamiast $O(n^3)$.
Wadą metod iteracyjnych jest fakt, że nie zawsze będą się zbiegać - zależy to od własności macierzy systemowej.
W moich implementacjach, po 100 iteracjach sprawdzamy czy norma wektora residuum wzrosła - jeżeli tak, to metoda się rozbiega.
W metodach iteracyjnych wykorzystujemy podział macierzy **M** na macierze **L**, **U** oraz **D**,
takie, że $\mathbf{M} = \mathbf{L} + \mathbf{D} + \mathbf{U}$.

Rysunek 1: Podział macierzy systemowej w metodach iteracyjnych.(źródło: instrukcja do laboratorium nr 3)

Rozpatrzamy układ równań postaci:

$$\boldsymbol{Mx} = \boldsymbol{b}$$

Gdzie **M** jest macierzą systemową, **x** wektorem rozwiązań i **b** jest wetkorem współczynników.

3.1. Wektor residuum

Wektor residuum jest bardzo istotnym elementem metod iteracyjnych. Badając jego normę euklidesową możemy okreslić moment, w którym przybliżenie wyniku jest zadowalające i algorytm może zakończyć działanie. Zakończenie algorytmu następuje w momencie, gdy norma ta jest mniejsza niż podany parametr epsilon.
Wektor residuum obliczamy ze wzoru:

$$\boldsymbol{res}^k = \boldsymbol{M}\boldsymbol{x}^k - \boldsymbol{b}$$

Norma euklidesowa wektora residuum ma postać:

$$||res||_2 = \sqrt{\sum_{i=1}^{n} res_i^2}$$

3.2. Metoda Jacobiego

Podstawiając do powyższego równania $\mathbf{M} = \mathbf{L} + \mathbf{D} + \mathbf{U}$, po przekształceniach otrzymujemy następujący schemat iteracyjny:

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

k - przybliżenie wyniku po k-tej iteracji
Możemy zauważyć, że czynniki $-\boldsymbol{D}^{-1}(\boldsymbol{L} + \boldsymbol{U})$ oraz $\boldsymbol{D}^{-1}\boldsymbol{b}$ możemy obliczyć jeden raz, zamiast w każdej iteracji osobno. Obliczenie pierwszego z tych czynników może być kosztowne, ponieważ dokonujemy mnożenia dwóch macierzy (złożoność - $O(n^3)$), a nie mnożenia $macierz \times wektor$ (złożoność - $O(n^2)$). W przypadku naszego zadania warto zauważyć, że wszystkie elementy na głównej przekątnej macierzy D są takie same. Możemy zatem przedstawić macierz D w postaci: $\boldsymbol{D} = d_{00}\boldsymbol{I}$, a następnie korzystając z własności macierzy jednostkowej obliczyć nasz czynnik jako: $-\boldsymbol{D}^{-1}(\boldsymbol{L} + \boldsymbol{U}) = -d_{00}^{-1} \cdot (\boldsymbol{L} + \boldsymbol{U})$. Wykorzystamu zatem mnożenie skalarane macierzy zamiast zwykłego, co pozwala zoptymalizować działanie algorytmu.

Jako, że macierz $\boldsymbol{D}$ jest macierzą diagonolną, możemy dokonać jej jawnego odwrócenia bez dużych kosztów pamięciowych: wystarczy elementy na przekątnej zamienić na ich odwrotności.

Po powyższych obliczeniach, działania, które należy wykonać podaczas każdej iteracji to:

- sprawdzenie czy norma wektora residuum jest odpowienio niska, jeżeli nie to:
- $-\boldsymbol{D}^{-1}(\boldsymbol{L} + \boldsymbol{U}) \cdot \boldsymbol{x}_{k-1}$
- $\boldsymbol{x}_k =< wynik\ poprzedniego\ punktu > + \boldsymbol{D}^{-1}\boldsymbol{b}$

Działania te powtarzamy, aż do uzyskania przybliżenia wyniku, które spełnia nasze oczekiwania co do dokładności.

3.3. Metoda Gaussa-Seidla

Analogicznie do metody Jacobiego uzyskujemy schemat iteracyjny:

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

W metodzie Gaussa-Seidla każda pętla algorytmu jest bardziej kosztowna obliczeniowo niż w przypadku metody Jacobiego. Jest to spowodowane faktem, że w metodzie Gaussa-Seidla przed pętlą możemy obliczyć tylko jeden wyraz: $(\boldsymbol{D} + \boldsymbol{L})^{-1}\boldsymbol{b}$. Jako, że macierz $(\boldsymbol{D} + \boldsymbol{L})$ jest macierzą dolną trójkątną, możemy w tym celu wykorzystać metodę podstawiania w przód.
Pierwszy wyraz, $-(\boldsymbol{D} + \boldsymbol{L})^{-1}(\boldsymbol{U}\boldsymbol{x}^k)$, trzeba obliczać podaczas każdej iteracji. W tym przypadku nie możemy jawnie odwrócić macierzy $(D+L)^{-1}$, ponieważ wiązałoby się to z dużymi kosztami pamięciowymi. Musimy więc rozwiązać układ równań liniowych postaci: $-(\boldsymbol{D}+\boldsymbol{L})^{-1}(\boldsymbol{U}\boldsymbol{x}^k)$, a następnie do otrzymanego wyniku dodać obliczony na początku algorytmu wyraz. Można zauważyć, że macierz $(\boldsymbol{D}+\boldsymbol{L})$ jest macierzą dolną trójkątną oraz wyraz $(\boldsymbol{U}\boldsymbol{x}^k)$ jest wekotrem. Możemy zatem wykorzystać metodę podstawiania w przód do rozwiązania tego układu.

3.4. Wyniki obliczeń

Rozwiązując układ równań z zadania A, uzyskane wyniki są następujące:

```
Porównanie wyników:
Jacobi: czas - 5.291513442993164s , iteracje - 29, norma z wektora residuum: 5.226877059481407e-10
Gauss-Seidl: czas - 4.408907890319824s , iteracje - 19, norma z wektora residuum: 6.572617688841712e-10
```

Rysunek 2: Porównanie działania metod iteracyjnych

Jak widać na powyższym zrzucie ekranu, metoda Jacobiego porzebuje większej liczby iteracji niż metoda Gaussa-Seidla. Jednak to metoda Gaussa-Seidla szybciej zakończyła obliczenia. Metoda Gaussa-Seidla jest szybsza i potrzebuje mniej iteracji do uzyskania zadowalającego przybliżenia wyniku. Obie metody działają poprawnie.

4. Zadanie C

Dane dla zadania C (nr indeksu: 184531):

- $a_1 = 3$
- $a_2 = a_3 = -1$
- $N = 931$

Wygenerowana macierz systemowa C (o rozmiarze 931×931):

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

Za generowanie macierzy oraz wektora współczynników B odpowiadają odpowiednie funckje w module Functions.py
Metody iteracyjne dla tej macierzy nie zbiegają się. W przypadku metody Gaussa-Seidla wynika to prawdopodobnie z faktu, że macierz $\mathbf{C}$ nie jest dodatnio określona, a w przypadku metody Jacobiego z faktu, że promień spektralny macierzy $\boldsymbol{D}^{-1}(\boldsymbol{L} + \boldsymbol{U})$ jest większy niż 1.

5. Zadanie D

W faktoryzacji LU przedstawiamy macierz systemową w postaci iloczynu dwóch macierzy trójkątnych.

$$\boldsymbol{A} = \boldsymbol{LU}$$

Gdzie **L** to macierz trójkątna dolna, zawierającą na przekątnej 1 i **U** to macierz trójkątna górna.
Najbardziej kosztowną operacją w metodzie LU jest uzyskanie macierzy **L** i **U**.
Gdy dokonamy już dekompozycji macierzy **A**, wykonujemy następujące działania:

- $\boldsymbol{LUx} = \boldsymbol{b}$
- tworzymy wektor pomocniczy: $\boldsymbol{y} = \boldsymbol{Ux}$
- rozwiązujemy układ równań: $\boldsymbol{L}^{-1}\boldsymbol{b} = \boldsymbol{y}$ metodą podstawiania w przód
- rozwiązujemy układ równań: $\boldsymbol{U}^{-1}\boldsymbol{y} = \boldsymbol{x}$ metodą podstawiania w tył

Metoda ta ma złożoność obliczeniową $O(n^3)$. Zaletą faktoryzacji LU jest fakt, że w przypadku gdy chcemy rozwiązać układ równań dla wielu prawych stron i tej samej macierzy systemowej, najbardziej kosztowną część algorytmu - dekompozycje macierzy - wystarczy wykonać jeden raz.

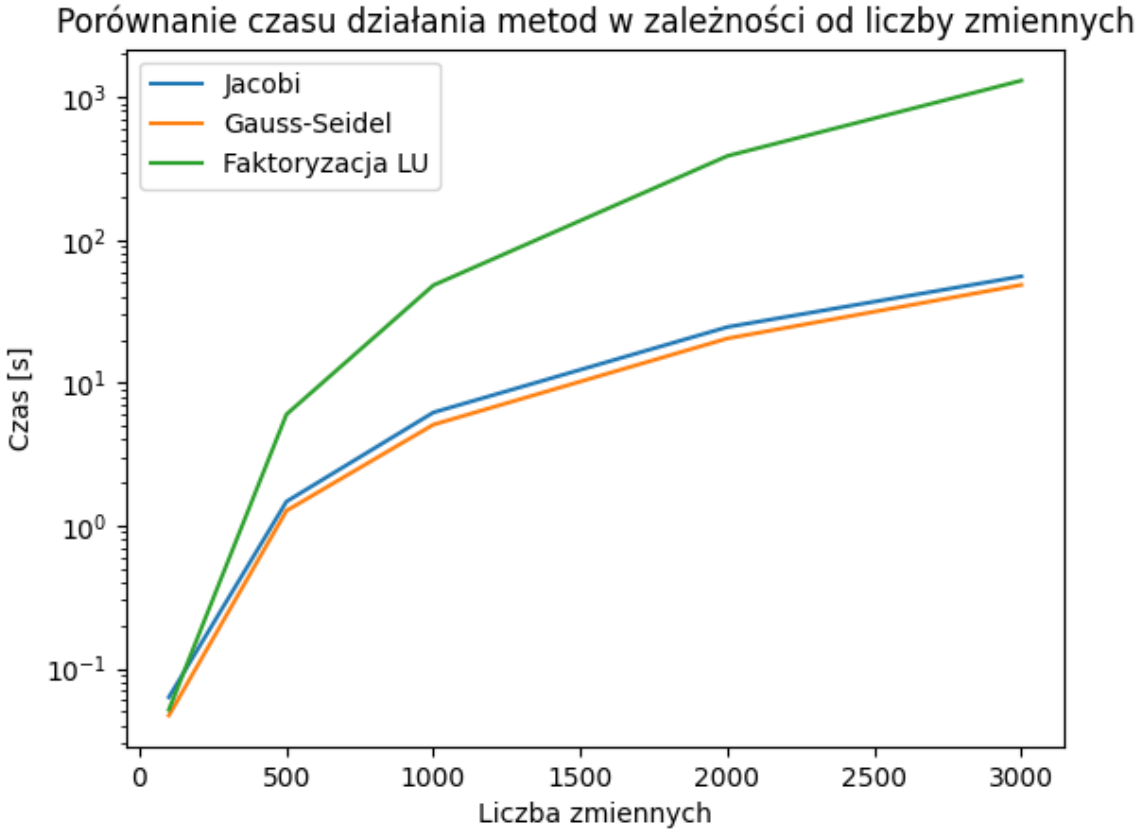
```
Metoda Jacobiego rozbiega się
Metoda GS rozbiega się.
Faktoryzacja LU: czas -  38.823699951171875s, norma z wektora residuum: 4.775610437404033e-13
```

Rysunek 3: Zadanie C - jak widać metody iteracyjne rozbiegają się

Norma z wektora residuum wynosi ok. $4.8 \cdot 10^{-14}$, a więc jest ok. 10^5 razy niższa niż w przypadku metod iteracyjnych, jednak zwróćmy uwagę, że w ich przypadku dokładność, a wraz z nią czas działania, możemy dostosować do naszych aktualnych potrzeb.
Jednak metody bezpośrenie charakteryzują się tym, że dadzą nam wynik po określonej liczbie operacji, podczas gdy metody iteracyjne mogą się rozbiegać. W przypadku faktoryzacji LU mogą wystąpić problemy jeżeli macierz systemowa zawiera 0 na głównej przekątnej. Rozwiązaniem tych problemów jest pivoting, czyli zamienianie wierszy macierzy miejscami w celu przeniesienia zer z głównej przekątnej w inne miejsca.

6. Zadanie E

Utworzony wykres dla liczby zmiennych N = {100, 500, 1000, 2000, 3000}. W celu lepszego zobrazowania wyników, dla osi y użyłem skali logarytmicznej.



Rysunek 4: Zadanie E - wykres czasu działania różnych metod

7. Zadanie F

Analizując powyższy wykres możemy zauważyć, że metoda faktoryzacji LU jest znacząco wolniejsza niż obie metody iteracyjne.

```
Porównanie wyników:
Jacobi: czas - 5.290364503860474s , iteracje - 29, norma z wektora residuum: 5.226877059481407e-10
Gauss-Seidl: czas - 4.408808708190918s , iteracje - 19, norma z wektora residuum: 6.572617688841712e-10
Faktoryzacja LU: czas -  38.854411125183105 s, norma z wektora residuum: 2.5024296770990176e-15
```

Rysunek 5: Porównanie czasów działania dla układu z zadania A

Gdy uruchomimy wszystkie 3 metody na układzie z zadania A, wyniki odzwierciedlają to co widzimy na wykresie - dla 931 zmiennych najszybsza jest metoda Gaussa-Seidla, niewiele wolniejsza jest metoda Jacobiego, a metoda faktoryzacji LU potrzebuje znacznie więcej czasu do uzyskania wyniku. Metoda Jacobiego potrzebuje więcej iteracji niż metoda Gaussa-Seidla, ale działa podobnie szybko. Iteracje metody Gaussa-Seidla są bardziej kosztowne, ponieważ w każdej iteracji musimy dokonać mnożenia macierzy przez wektor.
Warto jednak zauważyć, że faktoryzacja LU uzyskuje znacznie dokładniejszy wynik. Tak dokładny wynik rzadko jest nam potrzebny, a w metodach iteracyjnych możemy sterować dokładnością przez parametr epsilon.
Przeprowadziłem eksperyment i ustawiłem jako epsilon dla metod iteracyjnych wartośc 10^{-14}, czyli taką samą jak daje faktoryzacja LU.

```
Porównanie wyników:
Jacobi: czas - 7.262004852294922s , iteracje - 41, norma z wektora residuum: 9.126792692602138e-15
Gauss-Seidl: czas - 6.419880390167236s , iteracje - 28, norma z wektora residuum: 3.4665973299223347e-15
Faktoryzacja LU: czas -  38.85623240470886 s, norma z wektora residuum: 2.5024296770990176e-15
```

Rysunek 6: Czasy działania przy takiej samej dokładności wyniku

Jak widać po wynikach, metody iteracyjne są znacznie szybsze, nawet przy takiej samej dokładności. Jest to maksymalna dokładność z jaką możemy uzyskać wynik dla metod iteracyjnych. Przy wartościach niższych metody rozbiegają się.

8. Porównanie z biblioteką Numpy

Sprawdziłem działanie metody numpy.linalg.solve() dla układu z zadania A i działa ona znacznie szybciej od wszystkich zaimplementowanych przeze mnie metod.

```
Porównanie wyników:
Jacobi: czas - 5.306983232498169s , iteracje - 29, norma z wektora residuum: 5.226877059481407e-10
Gauss-Seidl: czas - 4.449592113494873s , iteracje - 19, norma z wektora residuum: 6.572617688841712e-10
Faktoryzacja LU: czas -  39.790077209472656 s, norma z wektora residuum: 2.502429677099017 6e-15
Implementacja metody linalg.solve() z biblioteki Numpy: czas - 0.27208256721496582s, norma z wektora residuum: 2.543675169694721e-15
```

Rysunek 7: Porównanie działania metody z biblioteki Numpy

Jak widać na powyższym obrazku, czas działania jest znacznie krótszy, a norma z wektora residuum jest na poziomie 10^{-15}, czyli wynik jest dokładny.
Lepsza wydajność pakietu Numpy jest spowodowany wykorzystaniem tablic zamiast list. Tablice (jak w C/C++) zawierają dane jednego typu i są upakowane w pamięci komputera. Listy, których używam w projekcie, mogą zawierać elementy różnego typu i być rozrzucone w pamięci. Implementacja zastosowana w Numpy powoduje znacznie większą szybkość wykonywania operacji na wekotrach i macierzach, a w rezultacie również szybsze działanie algorytmów odpowiadających za rozwiązywanie układów równań liniowych.