Metody Numeryczne - Projekt 2

Układy równań liniowych

Kacper Barański 182613

Informatyka | Grupa 1 | Semestr 4

Sprawozdanie

Celem drugiego projektu była implementacja metod iteracyjnych: *Jacobiego* i *Gaussa-Seidla* oraz metody bezpośredniej : *faktoryzacji LU* do rozwiązywania układów równań liniowych.

Konstrukcja układu równań Układ równań liniowych ma następującą postać:

$$Ax = b$$

gdzie :
A jest macierzą systemową,
b jest wektorem pobudzenia ,
x jest wektorem rozwiązań reprezentującym szukaną wielkość fizyczną.

Na potrzeby testów przyjmujemy, że macierz A jest tzw. macierzą pasmową o rozmiarze $N \times N$.

$$\mathbf{A} = \begin{bmatrix} a1 & a2 & a3 & 0 & 0 & 0 & 0 & \dots & 0 \\ a2 & a1 & a2 & a3 & 0 & 0 & 0 & \dots & 0 \\ a3 & a2 & a1 & a2 & a3 & 0 & 0 & \dots & 0 \\ 0 & a3 & a2 & a1 & a2 & a3 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 & 0 & a3 & a2 & a1 \end{bmatrix}$$

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.

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:

$$\mathbf{res}^{(k)} = \mathbf{A}\mathbf{x}^{(k)} - \mathbf{b}.$$

Badając normę euklidesową wektora residuum $(norm(\mathbf{res}^{(k)}))$, możemy w każdej iteracji algorytmu obliczyć jaki błąd wnosi wektor $\mathbf{x}^{(k)}$. Jeżeli algorytm zbiegnie się do dokładnego rozwiązania, residuum powinno być wektorem zerowym. Przeważnie jako kryterium stopu przyjmuje się normę z residuum o wartości mniejszej niż 10^{-6} .

Zadania

Zadanie A

Celem zadania A było stworzenie układu równań dla odpowiednich wartości $a1, a2, a3$ zgodnych z Moim indeksem.
Stąd:

$$c = 1 \qquad d = 3 \qquad e = 5 \qquad f = 2 \qquad N = 913$$

Wartości : $a1 = 5 + e \qquad a2 = -1 \qquad a3 = -1$

Otrzymana $macierz\ A$:

$$A = \begin{bmatrix} 11 & -1 & -1 & \dots & 0 \\ -1 & 11 & -1 & \dots & 0 \\ -1 & -1 & 11 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 11 \end{bmatrix}$$

Funkcje potrzebne do stworzenia macierzy oraz wektora :

```python
def create_matrix(a1, a2, a3, N):
    matrix = []
    for i in range(N):
        row = []
        for j in range(N):
            if i == j:
                row.append(a1)
            elif i - 1 <= j <= i + 1:
                row.append(int(a2))
            elif i - 2 <= j <= i + 2:
                row.append(int(a3))
            else:
                row.append(int(0))
        matrix.append(row)
    return matrix


def create_vector(f, N):
    return [sin(i * (f + 1)) for i in range(N)]
```

Wywołanie funkcji tworzących macierz A oraz wektor b :

```python
A = create_matrix(e+5, -1, -1, N_A)
b = create_vector(f, N_A)
```

Zadanie B

Celem zadania B była implementacja metod iteracyjnych rozwiązywania układów równań liniowych: *Jacobiego* i *Gaussa–Seidla*.

Metoda *Jacobiego :*

```
def jacobi(a, b, N):
    start_time = time.time()
    length = len(a)
    iter_count = 0
    x = [0] * length

    while 1:
        for i in range(length):
            x[i] = b[i]
            for j in range(length):
                if i != j:
                    x[i] -= a[i][j] * x[j]
            x[i] /= a[i][i]
        res = sub(dot(a, x), b)

        if norm(res) < norm_res:
            break
        iter_count += 1
    time_taken = time.time() - start_time
    print(" For N = ", N)
    print(" Jacobi iterative method ")
    print("     Time(s): ", time_taken)
    print("     Number of iterations: ", iter_count)
    print("---------------------------")
    return time_taken
```

Metoda *Gaussa – Seidla :*

```
def gauss_seidel(a, b, N):
    start_time = time.time()
    length_b = len(b)
    length = len(a)
    iter_counter = 0
    known_x = [0] * length_b

    while 1:
        iter_counter += 1
        for i in range(length):
            new_x = b[i]
            for j in range(length):
                if i != j:
                    new_x -= a[i][j] * known_x[j]
            new_x /= a[i][i]
            known_x[i] = new_x
        residuum = sub(dot(a, known_x), b)
        if norm(residuum) < norm_res:
            break

    time_taken = time.time() - start_time
    print(" For N = ", N)
    print(" Gauss-Seidel iterative method ")
    print("     Time(s) : ", time_taken)
    print("     Number of iterations : ", iter_counter)
    print("---------------------------")
    return time_taken
```

Do realizacji zadania B zaimplementowałem dodatkowe funkcje pomocnicze :

```
def fill(rows, cols, value):
    matrix = []
    for r in range(0, rows):
        matrix.append([value for _ in range(0, cols)])
    return matrix


def sub(a, b):
    return [a[i]-b[i] for i in range(len(a))]


def norm(vector):
    norm_res = 0.0
    for elem in vector:
        norm_res += pow(elem, 2)
    return pow(norm_res, 0.5)


def dot(matrix, vector):
    length = len(matrix)
    n = len(vector)
    multiplied = [0] * length
    for i in range(length):
        for j in range(n):
            multiplied[i] += matrix[i][j] * vector[j]
    return multiplied
```

Wywołanie funkcji w głównym pliku programu :

```
jacobi(A, b, N_A)
gauss_seidel(A, b, N_A)
```

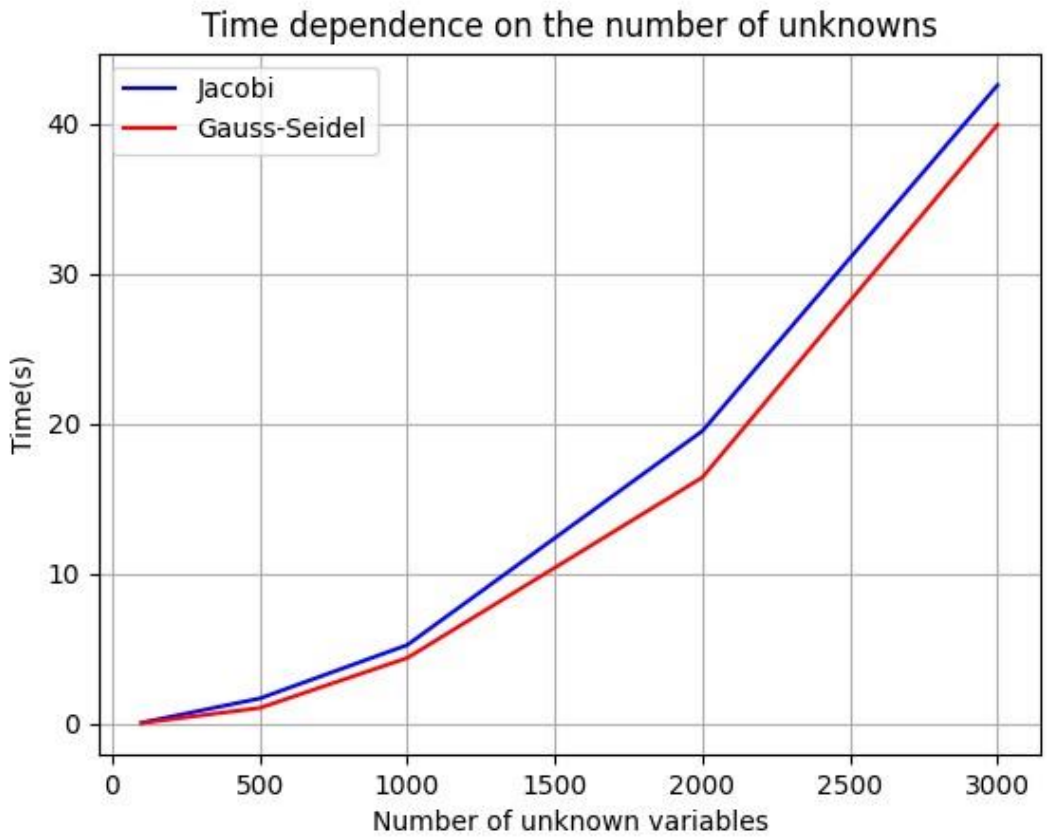
Porównanie czasu trwania algorytmów

oraz liczby iteracji :

```
For N =  913
Jacobi iterative method
    Time(s):  4.8474345207214355
    Number of iterations:  11
---------------------------
For N =  913
Gauss-Seidel iterative method
    Time(s) :  4.517552852630615
    Number of iterations :  11
---------------------------
```

Wnioski do zadania B

Po wielu przeprowadzonych próbach łatwo zauważyć, że Metoda Jacobiego potrzebuje więcej czasu na wykonanie obliczeń.
Dosyć dobrze prezentuje to poniższy wykres wykonany w oparciu o dane z zadania E :



Zadanie C

Celem zadania C było stworzenie układu równań dla $a1 = 3,\ a2 = a3 = -1\ oraz\ N = 9cd$.

Wywołanie funkcji potrzebnych do realizacji zadania :

```
C = create_matrix(3, -1, -1, N_A)
jacobi(C, b,N_A)
gauss_seidel(C, b, N_A)
```

Rezultat działania funkcji :

Dla metody *Jacobiego* :

```
Traceback (most recent call last):
  File "C:\Users\kacpe\Desktop\Projekt_2\main.py", line 26, in <module>
    jacobi(C, b,N_A)
  File "C:\Users\kacpe\Desktop\Projekt_2\IterationMethods.py", line 23, in jacobi
    if norm(res) < norm_res:
  File "C:\Users\kacpe\Desktop\Projekt_2\MatrixMethods.py", line 40, in norm
    norm_res += pow(elem, 2)
OverflowError: (34, 'Result too large')
```

Dla metody *Gaussa – Seidla* :

```
Traceback (most recent call last):
  File "C:\Users\kacpe\Desktop\Projekt_2\main.py", line 27, in <module>
    gauss_seidel(C, b,N_A)
  File "C:\Users\kacpe\Desktop\Projekt_2\IterationMethods.py", line 52, in gauss_seidel
    if norm(residuum) < norm_res:
  File "C:\Users\kacpe\Desktop\Projekt_2\MatrixMethods.py", line 40, in norm
    norm_res += pow(elem, 2)
OverflowError: (34, 'Result too large')
```

Wnioski do zadania C

Próba rozwiązania układu równań $Cx = b$ zakończyła się dla obu algorytmów błędem. Zatem łatwo zauważyć, że zarówno metody *Jacobiego* jak i metoda *Gaussa – Seidla* nie zbiega się dla takich wartości macierzy C i wektora b.

Zadanie D

Celem zadania D jest zaimplementowanie metody bezpośredniego rozwiązania układów równań liniowych: metodę *faktoryzacji LU* i zastosowanie jej do przypadku z zadania C.

Metoda *faktoryzacji LU :*

```
def factorization_LU(a, b, N):
    start_time = time.time()
    length = len(a)
    L = fill(length, length, 1)
    U = fill(length, length, 0)
    x = [1.0] * length
    y = [0] * length

    # Tworzymy macierze L i U, takie, że: LUx = b
    for i in range(length):
        for j in range(i + 1):
            U[j][i] += a[j][i]
            for k in range(j):
                U[j][i] -= L[j][k] * U[k][i]

        for j in range(i + 1, length):
            for k in range(i):
                L[j][i] -= L[j][k] * U[k][i]
            L[j][i] += a[j][i]
            L[j][i] /= U[i][i]

    # Rozwiązujemy układ równań: Ly = b za pomocą podstawiania wprzód
    for j in range(length):
        value = b[j]
        for i in range(j):
            value -= L[j][i] * y[i]
        y[j] = value / L[j][j]

    # Rozwiązujemy układ równań: Ux = y za pomocą podstawiania wstecz
    for j in range(length - 1, -1, -1):
        value = y[j]
        for i in range(j + 1, length):
            value -= U[j][i] * x[i]
        x[j] = value / U[j][j]

  res = sub(dot(a, x), b)
  time_taken = time.time() - start_time
  print(" For N = ", N)
  print(" LU factorization method ")
  print("     Time(s): ", time_taken)
  print("     Residuum norm: ", norm(res))
  print("-------------------------")
  return time_taken
```

Wywołanie metody *:*

```
factorization_LU(C, b, N_A)
```

Rezultat wywołania metody :

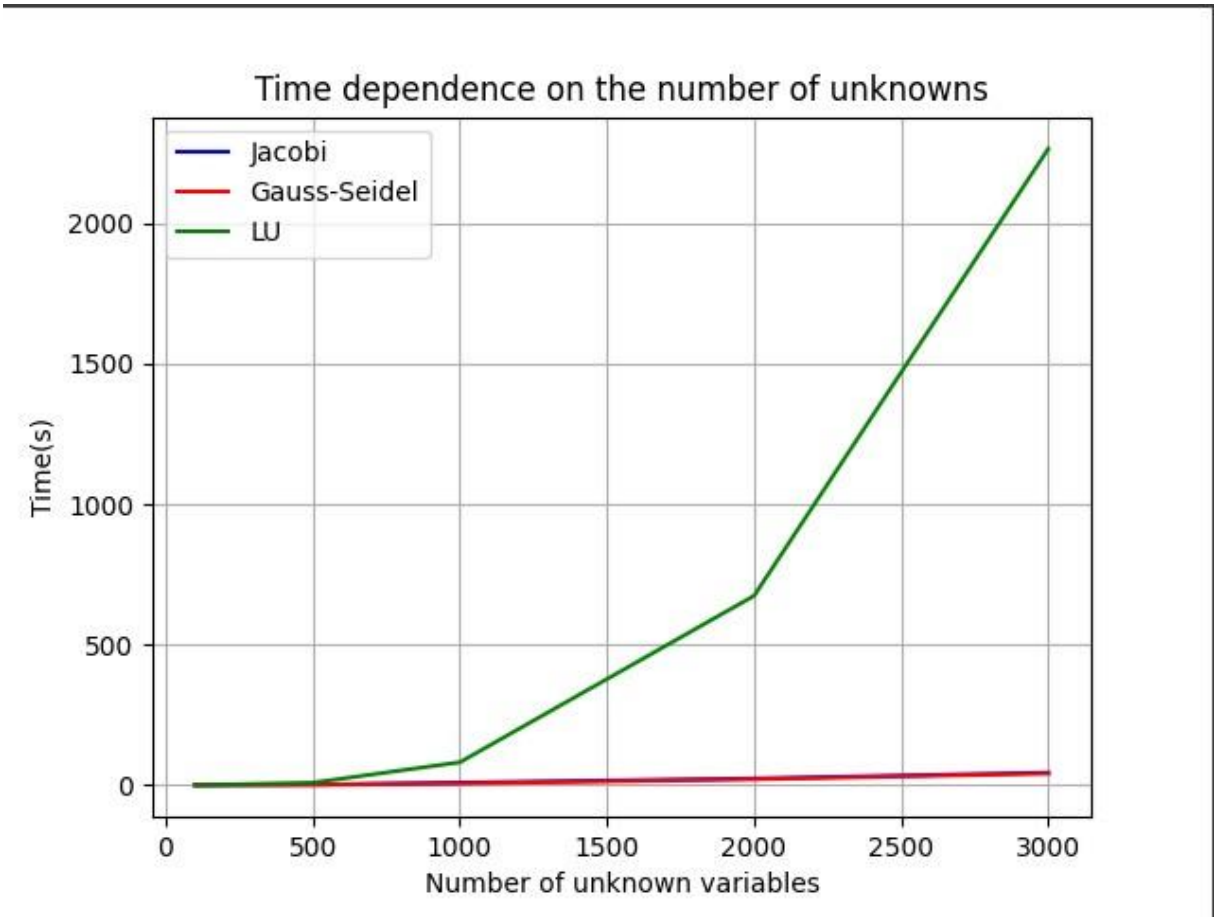
```
 For N =  913
 LU factorization method
     Time(s):  48.790733098983765
     Residuum norm:  4.176783511951184e-13
-------------------------
```

Wnioski do zadania D

Norma z residuum wynosi: 4.176783511951184e-13
Jest to wynik zbliżony do zera, co świadczy o wysokiej dokładności obliczeń.

Zadanie E

Celem zadania E było stworzenie wykresu zależności czasu trwania poszczególnych algorytmów od liczby niewiadomych $N = \{100, 500, 1000, 2000, 3000 \dots\}$ dla przypadku z punktu A.



Implementacja :

```
for number_of_variables in N:
    A = create_matrix(e+5, -1, -1, number_of_variables)
    b = create_vector(f, number_of_variables)

    jacobi_time_taken.append(jacobi(A, b, number_of_variables))
    gauss_seidel_time_taken.append(gauss_seidel(A, b, number_of_variables))
    LU_time_taken.append(factorization_LU(A, b, number_of_variables))

pyplot.plot(N, jacobi_time_taken, label="Jacobi", color="blue")
pyplot.plot(N, gauss_seidel_time_taken, label="Gauss-Seidel", color="red")
pyplot.plot(N, LU_time_taken, label="LU", color="green")
pyplot.legend()
pyplot.grid(True)
pyplot.xlabel('Number of unknown variables')
pyplot.ylabel('Time taken (s)')
pyplot.title('Time dependence on the number of unknowns')
pyplot.show()
```

Wnioski do zadania D

Na podstawie wykresu można zauważyć, że wzrost liczby niewiadomych, powoduje wydłużenie czasu wykonywania obliczeń. Metody iteracyjne są mniej precyzyjne, ale czas ich wykonania jest krótszy niż dla *faktoryzacji LU*, zatem metody iteracyjne nadają się do obliczeń na dużej liczbie niewiadomych, a *faktoryzacja LU* , może być wykorzystana gdy potrzebujemy dokładnych rezultatów. Należy zauważyć ,że metody iteracyjne mogą się nie zbiegać i wtedy konieczne będzie zastosowanie metody bezpośredniej - *faktoryzacji LU*.