Metody Numeryczne – Zadanie NUM3

Bartosz Bochniak

Wstęp

Polecenie

Wyznacz $\boldsymbol{y = A^{-1}x}$, dla

$$\boldsymbol{A} = \begin{pmatrix} 1.01 & \frac{0.2}{1} & \frac{0.15}{1^3} & & & & & \\ 0.3 & 1.01 & \frac{0.2}{2} & \frac{0.15}{2^3} & & & & \\ & 0.3 & 1.01 & \frac{0.2}{3} & \frac{0.15}{3^3} & & & \\ & & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & & & 0.3 & 1.01 & \frac{0.2}{N-2} & \frac{0.15}{(N-2)^3} \\ & & & & & 0.3 & 1.01 & \frac{0.2}{N-1} \\ & & & & & & 0.3 & 1.01 \end{pmatrix}$$

Oraz $\boldsymbol{x} = (1, 2, \dots, N)^T$. Ustalamy *N* = 300. Oblicz również wyznacznik macierzy **A**. Zadanie rozwiąż właściwą metodą (uzasadnij wybór) i wykorzystaj strukturę macierzy. Algorytm proszę zaprogramować samodzielnie; wyjątkowo nie należy stosować procedur bibliotecznych z zakresu algebry liniowej ani pakietów algebry komputerowej (chyba, że do sprawdzenia swojego rozwiązania, co zawsze jest mile widziane). Ponadto, potraktuj *N* jako zmienną i zmierz czas działania swojego programu w funkcji *N*. Wynik przedstaw na wykresie. Jakiej zależności się spodziewamy?

Rozwiązanie

Równanie postaci $y = A^{-1}x$ możemy od razu znacznie uprościć, poprzez pomnożenie lewostronne przez macierz **A**, co przekształci równanie do formy $Ay = x$. Oszczędzi nam to obliczania $\mathbf{A^{-1}}$, co bardzo redukuje czas, który program będzie potrzebować do rozwiązania problemu.

Macierz **A** posiada cztery diagonale, co sprawia, że jest macierzą rzadką. Biorąc pod uwagę strukturę macierzy **A**, można dojść do wniosku, że rozkład *LU* tej macierzy będzie bardzo nieefektywny, gdyż jego złożoność obliczeniowa wynosi $O(n^3)$, pomimo tego, że większość czasu zostanie spędzona na zbędnym dodawaniu zer.

Wykorzystując budowę macierzy, możemy utworzyć algorytm o złożoności O(n), traktując macierz **A** jako macierz (4 × N), gdzie każda diagonala jest jednym z wierszy.

Poniżej podane są wzory na macierze **U** oraz **L**, stosowane do rozkładu *LU*:

$$U_{ij} = \begin{cases} A_{ij} - \sum_{k=1}^{i-1} L_{ik}U_{kj} & dla\ i \le j \\ 0 & dla\ i > j \end{cases}$$

$$L_{ij} = \begin{cases} 1 & dla\ i = j \\ \frac{A_{ij} - \sum_{k=1}^{j-1} L_{ik}U_{kj}}{U_{jj}} & dla\ i > j \\ 0 & dla\ i < j \end{cases}$$

Stosując te wzory, możemy otrzymać specjalny wzór przystosowany dla macierzy o kilku diagonalach, który pomija zbędne operacje zawierające zera. Ten wzór zostanie zastosowany w naszej implementacji, oraz obniży złożoność obliczeniową z $O(n^3)$ do O(n).

Następny powód dla którego użycie rozkładu *LU* jest najbardziej optymalne, to rozwiązanie samego równania $\boldsymbol{Ay} = \boldsymbol{x}$. Podstawiając za A = LU, otrzymujemy następujący rachunek:

$$Ay = x \ \rightarrow \ LUy = x$$

Do rozwiązania powyższego równania korzystamy z forward substitution, oraz backward substitution:

- $Lz = x$ (Forward substitution)

- $Uy = z$ (Backward substitution)

Powyższe metody również możemy dostosować specjalnie pod naszą macierz z czterema diagonalami, co także sprawi, że ich złożoność obliczeniowa „spadnie” do O(n).

Na koniec należy policzyć wyznacznik macierzy **A**. Korzystamy także z rozkładu LU:

$$\det(A) = \det(L) * \det(U) = 1 * \det(U) = \det(U) = \prod_{i=1}^{n} U_{ii}$$

Oznacza to, że wyznacznik macierzy **A** jest równy iloczynowi wszystkich elementów diagonali macierzy **U**. Obliczanie tego posiada złożoność obliczeniową O(n).

Implementacja

Zaczynamy od zdefiniowania macierzy **A** w kodzie jako struktury, która zawiera cztery tablice, każda z nich reprezentująca jedną z diagonali. Następnie tworzymy funkcję `LUDecomposition` bazując na podanych wzorach. Zauważamy, że te wzory korzystają z sum, które dla czterech diagonali nie będą posiadać więcej niż cztery niezerowe elementy, dlatego rozpisujemy te wzory dla tych elementów, co pozwala nam pominąć wszystkie zbędne zera.

Dzięki temu otrzymujemy złożoność O(n). Bazując na tej samej zasadzie, zapisujemy wzory dla `Forward_Substitution` oraz `Backward_Substitution`.

Ostatecznie tworzymy także funkcję `program(int size, ...)`, która wywołuje podany algorytm zależnie od liczby size, która symbolizuje rozmiar macierzy wobec której ten algorytm zostanie wykonany. Dzięki temu potrafimy zmierzyć czas, który minie podczas wykonywania tego algorytmu w zależności od rozmiaru macierzy.

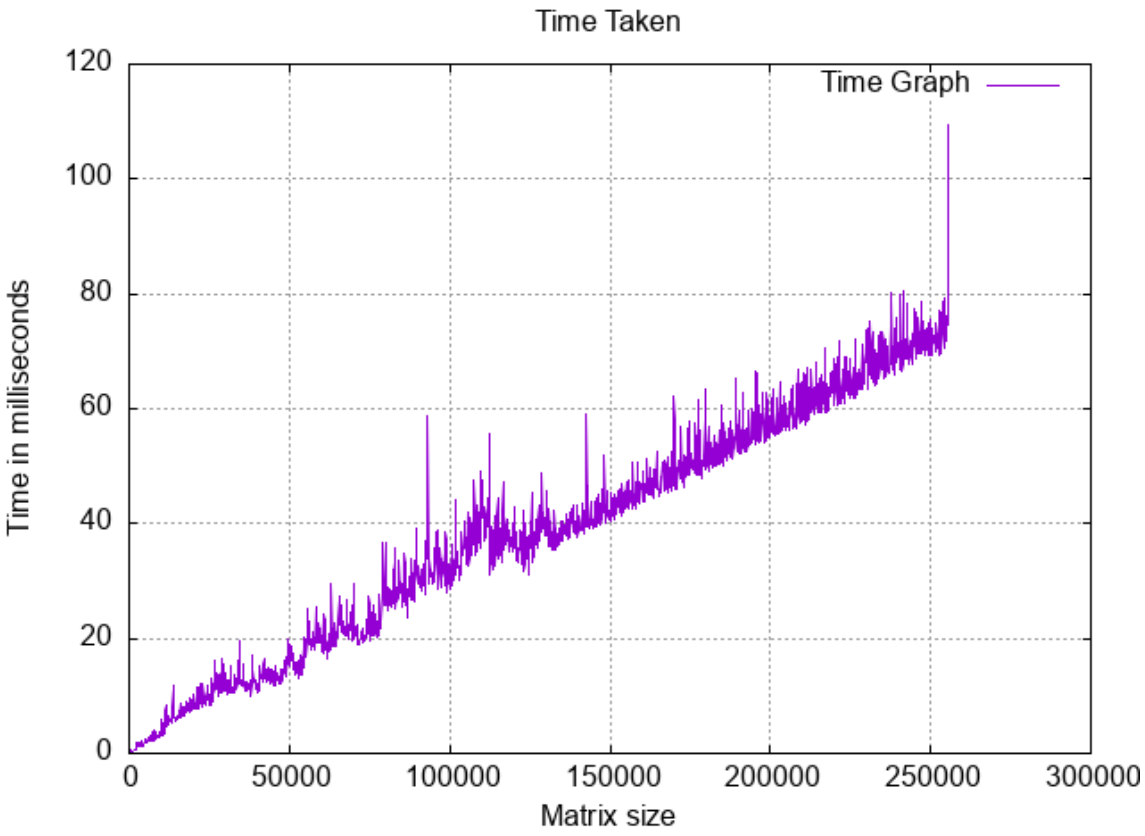
Wynik

Dla N = 300, otrzymujemy poniższy wektor **y**, oraz wyznacznik macierzy **A**:

```
Rozwiazanie to:
(0.336377; 1.59818; 2.27082; 3.08484; 3.84497; 4.61666; 5.38248; 6.14867; 6.91388; 7.6788;
8.44339; 9.20777; 9.97199; 10.7361; 11.5001; 12.264; 13.0278; 13.7916; 14.5554; 15.3191;
16.0827; 16.8464; 17.61; 18.3736; 19.1372; 19.9007; 20.6643; 21.4278; 22.1913; 22.9548;
23.7183; 24.4818; 25.2453; 26.0087; 26.7722; 27.5357; 28.2991; 29.0626; 29.826; 30.5895;
31.3529; 32.1163; 32.8798; 33.6432; 34.4066; 35.17; 35.9334; 36.6969; 37.4603; 38.2237;
38.9871; 39.7505; 40.5139; 41.2773; 42.0407; 42.8041; 43.5675; 44.3309; 45.0943; 45.8577;
46.6211; 47.3845; 48.1479; 48.9113; 49.6747; 50.438; 51.2014; 51.9648; 52.7282; 53.4916;
54.255; 55.0184; 55.7817; 56.5451; 57.3085; 58.0719; 58.8353; 59.5986; 60.362; 61.1254;
61.8888; 62.6522; 63.4155; 64.1789; 64.9423; 65.7057; 66.469; 67.2324; 67.9958; 68.7592;
69.5225; 70.2859; 71.0493; 71.8127; 72.576; 73.3394; 74.1028; 74.8661; 75.6295; 76.3929;
77.1563; 77.9196; 78.683; 79.4464; 80.2097; 80.9731; 81.7365; 82.4999; 83.2632; 84.0266;
84.79; 85.5533; 86.3167; 87.0801; 87.8434; 88.6068; 89.3702; 90.1335; 90.8969; 91.6603;
92.4236; 93.187; 93.9504; 94.7137; 95.4771; 96.2405; 97.0038; 97.7672; 98.5306; 99.2939;
100.057; 100.821; 101.584; 102.347; 103.111; 103.874; 104.638; 105.401; 106.164; 106.928;
107.691; 108.454; 109.218; 109.981; 110.744; 111.508; 112.271; 113.035; 113.798; 114.561;
115.325; 116.088; 116.851; 117.615; 118.378; 119.141; 119.905; 120.668; 121.432; 122.195;
122.958; 123.722; 124.485; 125.248; 126.012; 126.775; 127.538; 128.302; 129.065; 129.829;
130.592; 131.355; 132.119; 132.882; 133.645; 134.409; 135.172; 135.935; 136.699; 137.462;
138.226; 138.989; 139.752; 140.516; 141.279; 142.042; 142.806; 143.569; 144.332; 145.096;
145.859; 146.623; 147.386; 148.149; 148.913; 149.676; 150.439; 151.203; 151.966; 152.729;
153.493; 154.256; 155.019; 155.783; 156.546; 157.31; 158.073; 158.836; 159.6; 160.363;
161.126; 161.89; 162.653; 163.416; 164.18; 164.943; 165.707; 166.47; 167.233; 167.997;
168.76; 169.523; 170.287; 171.05; 171.813; 172.577; 173.34; 174.104; 174.867; 175.63;
176.394; 177.157; 177.92; 178.684; 179.447; 180.21; 180.974; 181.737; 182.5; 183.264;
184.027; 184.791; 185.554; 186.317; 187.081; 187.844; 188.607; 189.371; 190.134; 190.897;
191.661; 192.424; 193.188; 193.951; 194.714; 195.478; 196.241; 197.004; 197.768; 198.531;
199.294; 200.058; 200.821; 201.585; 202.348; 203.111; 203.875; 204.638; 205.401; 206.165;
206.928; 207.691; 208.455; 209.218; 209.981; 210.745; 211.508; 212.272; 213.035; 213.798;
214.562; 215.325; 216.088; 216.852; 217.615; 218.378; 219.142; 219.905; 220.669; 221.432;
222.195; 222.959; 223.722; 224.485; 225.249; 226.012; 226.775; 227.539; 228.302; 229.217)

Wyznacznik macierzy A to: 13.8264
```

Wykres czasu spędzonego na obliczeniu wektora **y** oraz wyznacznika w zależności od wielkości N:



Wnioski

Analiza wykresu zależności rozmiaru od czasu obliczeń wykazała, że przebieg wykresu przypomina funkcję liniową, przy czym dla większych rozmiarów macierzy zauważalne są rosnące odchylenia od tej tendencji. To potwierdza hipotezę o złożoności obliczeniowej klasy O(n).

Odkrycie to wskazuje jednocześnie na możliwość znacznego zmniejszenia złożoności obliczeniowej poprzez optymalizację algorytmów przetwarzających macierze rzadkie o specyficznych strukturach. Tego rodzaju optymalizacje oferują potencjał do istotnej redukcji czasu wymaganego na różne obliczenia, co może mieć kluczowe znaczenie w kontekście zaawansowanych obliczeń matematycznych oraz innych procesów wymagających dużych zasobów obliczeniowych.

Notatka z poprawki

Błąd który wywołał zły wynik w poprzedniej implementacji tkwił w mojej implementacji funkcji „backward_substitution”, gdzie przypadkowo pominięty został jeden z elementów.