Liss Vadym

257264

Dane dotyczące macierzy potrzebnej do wykonania zadań

Dana macierz współczynników $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ i wektor prawych stron $\boldsymbol{b} \in \mathbb{R}^{n}$, $n \geq 4$. Macierz $\boldsymbol{A}$ jest rzadką, tj. mającą dużo elementów zerowych i blokową o następującej strukturze:

$$\boldsymbol{A} = \begin{pmatrix} \boldsymbol{A_1} & \boldsymbol{C_1} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \cdots & \boldsymbol{0} \\ \boldsymbol{B_2} & \boldsymbol{A_2} & \boldsymbol{C_2} & \boldsymbol{0} & \boldsymbol{0} & \cdots & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{B_3} & \boldsymbol{A_3} & \boldsymbol{C_3} & \boldsymbol{0} & \cdots & \boldsymbol{0} \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \boldsymbol{0} & \cdots & \boldsymbol{0} & \boldsymbol{B_{v-2}} & \boldsymbol{A_{v-2}} & \cdots & \boldsymbol{0} \\ \boldsymbol{0} & \cdots & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{B_{v-1}} & \boldsymbol{A_{v-1}} & \boldsymbol{C_{v-1}} \\ \boldsymbol{0} & \cdots & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{B_v} & \boldsymbol{A_v} \end{pmatrix},$$

$v = \frac{n}{\ell}$, zakładając, że n jest podzielne przez ℓ, gdzie $\ell \geq 2$ jest rozmiarem wszystkich kwadratowych macierzy wewnętrznych (bloków): $\boldsymbol{A_k}, \boldsymbol{B_k}$ *i* $\boldsymbol{C_k}$.

- $\boldsymbol{A_k} \in \mathbb{R}^{\ell \times \ell}, k = 1, \dots, v$ jest macierzą gęstą
- $\boldsymbol{0}$ jest kwadratową macierzą zerową stopnia ℓ
- $\boldsymbol{B_k} \in \mathbb{R}^{\ell \times \ell}, k = 2, \dots, v$ jest następującej postaci:

$$\begin{pmatrix} 0 & \cdots & 0 & b^k_{1\ell-1} & b^k_{1\,\ell} \\ 0 & \cdots & 0 & b^k_{2\ell-1} & b^k_{2\,\ell} \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & b^k_{\ell\ell-1} & b^k_{\ell\ell} \end{pmatrix}$$

- $\boldsymbol{C_k} \in \mathbb{R}^{\ell \times \ell}, k = 1, \dots, v-1$ jest macierzą diagonalną postaci:

$$\begin{pmatrix} c^k_1 & 0 & 0 & \cdots & 0 \\ 0 & c^k_2 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & c^k_{\ell-1} & 0 \\ 0 & \cdots & 0 & 0 & c^k_{\ell} \end{pmatrix}$$

Sposób przechowywania macierzy:

Ważnym zastrzeżeniem w zadaniu było użycie specjalnej struktury, która pozwoli na zapamiętanie tylko niezerowych wartości macierzy *A* w ramach oszczędności pamięci. W języku Julia została ona nazwana SparseArrays. Implementacja tej struktury pozwala na szybszy dostęp do elementów poprzez kolumny niż przez wiersze. Na potrzeby obliczania złożoności czasowej powyższych algorytmów przyjęto, że dostęp do elementu macierzy przechowywanej w tej strukturze jest stały.

Zadanie 1

Napisać funkcję rozwiązującą układ Ax = b metodą eliminacji Gaussa uwzględniającą specyficzną postać (1) macierzy A dla dwóch wariantów:

- *bez wyboru elementu głównego,*
- *z częściowym wyborem elementu głównego.*

Opis algorytmu

Klasyczna wersja algorytmu eliminacji Gaussa polega na sprowadzeniu układu równań do macierzy trójkątnej wykonując przy tym tylko operacje takie jak dodawanie, odejmowanie czy mnożenie przez niezerową stałą na kolumnach i wierszach, które dodatkowo można dowolnie przestawiać. Wynikiem tych operacji jest macierz, dzięki której wraz z wektorem prawych stron można obliczyć wektor rozwiązań układu równań.

Kluczowym elementem algorytmu jest przetworzenie macierzy wejściowej na trójkątną. Aby tego dokonać należy wyeliminować z niej elementy niezerowe znajdujące się pod diagonalną. Przypatrując się strukturze macierzy podanej w zadaniu można zauważyć, że dla l=4 w kolejnych kolumnach musimy wyzerować kolejno 3,2,5,4,3,2,5,4… elementów, które występują pod diagonalną jedna pod drugą. Wiedząc to, przy oznaczeniu k jako numer kolumny możemy stwierdzić, że w każdej kolejnej kolumnie należy wyeliminować elementy w wierszach od k do $k+l-modulo(k,l)$.
Budowa funkcji $modulo(k,l)$ wynika z faktu, że dwie ostatnie kolumny w macierzy Bk są niezerowe (gdyby tylko ostatnia kolumna była niezerowa eliminowalibyśmy tylko elementy na pozycjach od k do $k+l-k\ mod\ l$).

Aby wyeliminować element a_{ik} należy od i-tego wiersza odjąć k-ty wiersz pomnożony przez współczynnik $a \leftarrow \frac{a_{\mathrm{ik}}}{a_{\mathrm{kk}}}$. Warto zauważyć, że może tu dojść do dzielenia przez zero. Aby tego uniknąć należy zamienić ze sobą miejscami kolumny lub wiersze. Należy pamiętać, aby równocześnie dokonywać zmian również w wektorze prawych stron.
Złożoność algorytmu eliminacji Gaussa wynosi $O(n)$, ponieważ główna pętla wykona się dokładnie n razy, natomiast zagnieżdżone w niej pętle co najwyżej l razy, co daje złożoność liniową.

Eliminacja Gaussa z częściowym wyborem elementu głównego

Algorytm ten jest modyfikacją powyższego algorytmu, która ma na celu zapobieganie występowania zer na przekątnej macierzy. Polega ona na wyborze elementu głównego w kolumnie i odpowiednim przestawieniu wierszy macierzy, aby znalazł się on na przekątnej. Element ten (a dokładnie jego wartość bezwzględna) jest zawsze największym elementem w kolumnie. Przestawianie elementów w kolumnie można wyrazić następującym wzorem: $|akk|=|as(k),k|=\max\{|aik|: i=k,\ldots,n\}$
gdzie $s(k)$ jest wektorem permutacji.

Ze względu na swoje podobieństwo do algorytmu eliminacji Gaussa, algorytm ten ma również podobną złożoność. Jest ona liniowa, jednak ze względu na obecność dodatkowej pętli, w której wybierany jest główny element, wykonujemy dodatkowe operacje, które powodują, że złożoność ta jest większa, jednak z dokładnością do stałej.

Poniżej znajduje się pseudokod algorytmów rozwiązujących układ równań przy pomocy odpowiednio eliminacji Gaussa oraz eliminacji Gaussa z częściowym wyborem elementu głównego.

```
rozwiazUkladGauss(A,b,n,l)
  A',b' ← eliminacjaGaussa(A,b,n,l,false)
  for i ← n to 1 do
    suma ← 0
    for j ← i + 1 to min(n, i + l) do
      suma ← suma + A'_ji * x_j
    end for
    x_i ← (b'_i − suma) / A'_ii
  end for
  return x
end function
```

```
rozwiazUkladGaussZWyborem(A,b,n,l)
  A',perm,b'←eliminacjaGaussaZWyborem(A,b,n,l,false)
  for i ← n to 1 do
    suma ← 0
    for j ← i+1 to min (n, i + 2 * l + 1)do
      suma ← suma + A'_{j,perm_i} * x_j
    end for
    x_i ← (b'_{perm_i} − suma) / A'_{i,perm_i}
  end for
  return x
end function
```

Rozwiązaniem tego zadania jest implementacja powyższego pseudokodu w języku Julia w postaci funkcji odpowiednio ***eliminacjaGaussa***, ***eliminacjaGaussaZWyborem***, ***rozwiazUkladGauss*** oraz ***rozwiazUkladGaussZWyborem*** w module ***blocksys***.

Zadanie 2

Napisać funkcję wyznaczającą rozkład LU macierzy A metodą eliminacji Gaussa uwzględniającą specyficzną postać (1) macierzy A dla dwóch wariantów:

- *bez wyboru elementu głównego,*
- *z częściowym wyborem elementu głównego. Rozkład LU musi być efektywne pamiętany!*

Opis algorytmu

Rozkład LU jest algorytmem silnie związanym z algorytmem eliminacji Gaussa, w którym wyznacza się nie jedną a dwie macierze trójkątne: górną (U) oraz dolną (L). W ten sposób otrzymujemy układ równań $LUx=b$, którego rozwiązanie sprowadza się do rozwiązania następujących dwóch układów równań:

$$Ux=b'$$
$$Lb'=b$$

Do wyznaczenia obu tych macierzy posłuży nieco zmodyfikowany algorytm eliminacji Gaussa, w którym zamiast zerować elementy w dolnej macierzy trójkątnej, wstawiamy w ich miejsce wartość a użytą do wyeliminowania tego elementu. Dodatkowo pomijana jest aktualizacja wektora b, ponieważ następuje to dopiero w momencie rozwiązywania układu przy użyciu rozkładu LU. Dotyczy to zarówno rozkładu LU metodą eliminacji Gaussa jak również metodą eliminacji Gaussa z częściowym wyborem elementu głównego. Złożoność tych algorytmów jest taka sama jak złożoność algorytmu eliminacji Gaussa i wynosi $O(n)$.

Rozwiązaniem tego zadania są funkcje ***rozkladLU*** oraz ***rozkladLUZWyborem***, które uruchamiają odpowiednio funkcje ***eliminacjaGaussa*** oraz ***eliminacjaGaussaZWyborem*** z flagą ***zRozkladem*** ustawioną na wartość **true**.

Zadanie 3

Napisać funkcję rozwiązującą układ równań Ax = b (uwzględniającą specyficzną postać (1) macierzy A) jeśli wcześniej został już wyznaczony rozkład LU przez funkcję z punktu 2.

Opis algorytmu

Jak już zostało wspomniane opisie algorytmu **Zadanie2** rozkładu LU, aby rozwiązać układ równań z wyznaczonym rozkładem LU należy rozwiązać dwa układy: $Ux=b'$ oraz $Lb'=b$. Algorytm rozwiązywania tych układów (przy obliczeniu rozkładu poprzez eliminację Gaussa) przedstawia pseudokod poniżej.

```
rozwiazUkladLU(A,b,n,l)
  for i←1 to n do
    suma←0
    for j←max(1,l∗⌊(i−1)/l⌋)to i−1 do
      suma←suma+A_ji∗bj
    end for
    b_i←b_i −suma
  end for
  for i←n to 1 do
    suma←0
    for j←i+1 to min(n,i+l+1) do
      suma←suma+A_ji∗x_j
    end for
    x_i←(b_i−suma)/A_ij,
  end for
  return x
end function
```

Algorytm rozwiązywania tych układów (przy obliczeniu rozkładu poprzez eliminację Gaussa z częściowym wyborem elementu głównego) przedstawia poniższy pseudokod.

```
rozwiazUkladLUZWyborem(A,perm,b,n,l)
  for i←1 to n do
    suma←0
    for j←max(1,l∗⌊(i−1)/l⌋)to i−1 do
      suma←suma+A_{j,perm_i}∗bj
    end for
    b_i←b_{perm_i} −suma
  end for
  for i←n to 1 do
    suma←0
    for j←i+1 to min(n,i+2∗l+1) do
      suma←suma+A_{j,perm_i}∗x_j
    end for
    x_i←(b_i−suma)/A_{i,perm_i},
  end for
  return x
end function
```

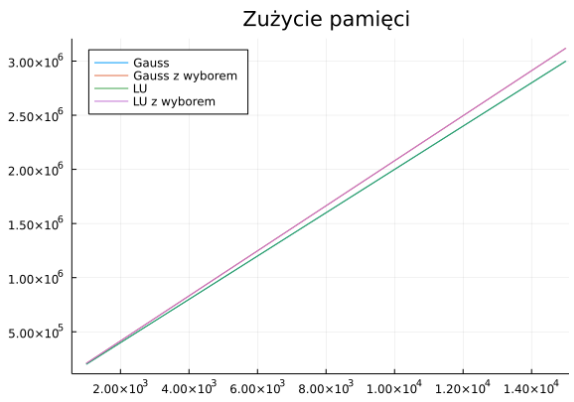
Złożoność przedstawionych powyżej algorytmów wynosi $O(n)$.

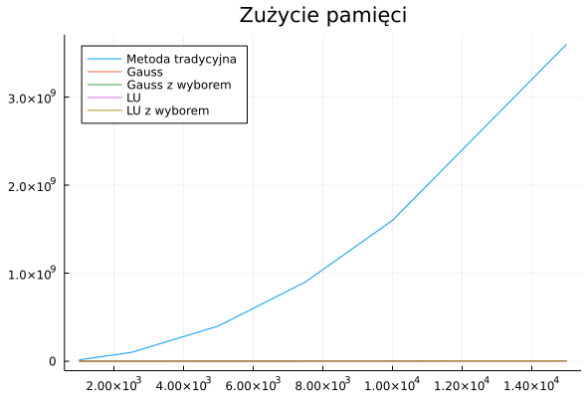
Rozwiązaniem tego zadania jest implementacja powyższego pseudokodu w języku Julia w funkcjach odpowiednio ***rozwiazUkladLU*** oraz ***rozwiazUkladLUZWyborem*** w module ***blocksys***.

Wyniki

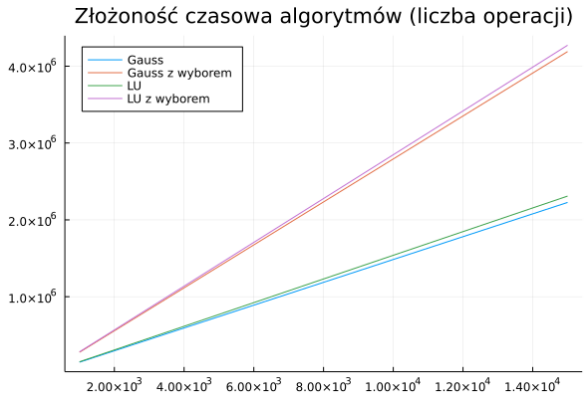
Korzystając z faktu, że macierze postaci opisanej na liście zawierają bardzo dużo elementów zerowych, złożoność pamięciowa zaimplementowanych algorytmów została znacząco ograniczona poprzez zastosowanie macierzy rzadkich z modułu SparseArrays. Ma to znaczenie szczególnie dla bardzo dużych *n*.

Zaimplementowane algorytmy zostaną porównane z ”metoda tradycyjna”, to jest $A\backslash b$ dla A przechowywanego jako gęste. Wykresy porównujące złożoność pamięciowa umieszczone zostały na podanym rysunku.

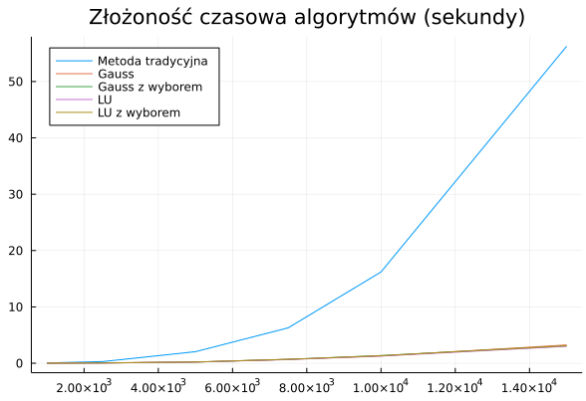


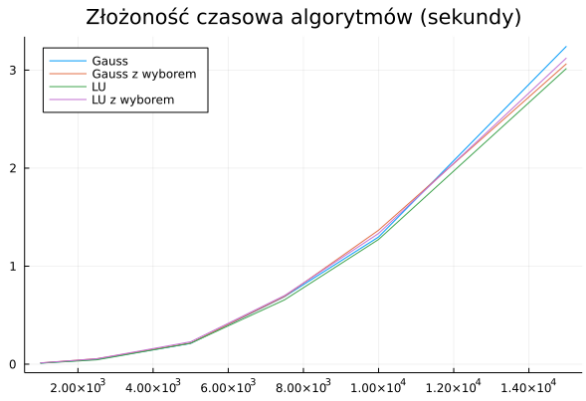


Patrząc na wyniki, widzimy pomiary zużycia pamięci w bajtach w zależności od rozmiaru macierzy dla zaimplementowanych algorytmów, uzyskane za pomocą makra @timed. Jak widać, wykorzystanie macierzy rzadkich znacznie ogranicza zapotrzebowanie na pamięć. Różnice między wariantami z wyborem i bez wynikają z konieczności przechowywania wektora *P*.
Złożoność obliczeniowa zaimplementowanych algorytmów została zmierzona na dwa sposoby: z użyciem makra *@timed* oraz poprzez zliczanie odwołana do elementów macierzy *A*. W przeprowadzonych analizach złożoności zakładaliśmy, że dostęp do elementu macierzy odbywa się w czasie stałym. W rzeczywistości tak nie jest, co widać na wykresach z rysunków porównania złożoności czasowej algorytmów, jednak drugi sposób z następnego rysunku pozwala nam na zignorowanie tego kosztu, zasadniczo potwierdzając poprawność naszych analiz.



Na podanym rysunku są pomiary liczby odwołana do elementów macierzy w zależności od jej rozmiaru dla zaimplementowanych algorytmów. Zgodnie z oczekiwaniami, wszystkie osiągają złożoność $O(n)$. Warianty z wyborem mają większe złożoności niż te bez, co spowodowane jest łagodniejszymi ograniczeniami na zakresy iteracji. Pozwalają one jednak na pracę z macierzami, w których na przekątnej występują niewielkie wartości. Warianty LU mają większe złożoności niż ich odpowiedniki, ponieważ w drugiej fazie rozwiązują dwa układy równań. Z drugiej jednak strony ich pierwsza faza może być przeprowadzona tylko raz i użyta wielokrotnie dla różnych wektorów *b*.





Na tych dwóch wykresach przedstawiamy pomiary czasu pracy w sekundach w zależności od rozmiaru macierzy dla zaimplementowanych algorytmów, uzyskane za pomocą makra *@timed*. Ograniczenie zakresów iteracji do niezerowych kolumn i wierszy przynosi ogromne zyski czasowe w stosunku do naiwnego podejścia. Zaimplementowane algorytmy osiągają bardzo podobne do siebie wyniki.

Wnioski

Odpowiednia analiza problemu może spowodować, że stosunkowo łatwo otrzymamy rozwiązanie, które w naiwnym podejściu byłoby kosztowne lub niemożliwe do uzyskania. Także podany problem zwraca szczególną uwagę na fakt, że w niektórych sytuacjach mała modyfikacja klasycznego algorytmu daje możliwość znajdowania rozwiązań dla skomplikowanych problemów w szybki i łatwy sposób.
Drobne modyfikacje oryginalnych algorytmów umożliwiły nam znaczne ograniczenie ich złożoności obliczeniowej. Znając strukturę danych, udało nam się również ograniczyć zapotrzebowanie na pamięć, odpowiednio wybierając możliwie oszczędny sposób ich przechowywania.
Zauważę, że zużycie pamięci rośnie liniowo wraz ze wzrostem wartości n. Czas działania natomiast, mimo iż powinien być liniowy, bardziej przypomina wykres funkcji kwadratowej. Wynika to z faktu, że w rzeczywistości dostęp do elementów macierzy nie jest stały, co powoduje wydłużenie czasu działania algorytmów. Można zauważyć również, że rozwiązywanie układu przy pomocy rozkładu *LU* jest na ogół szybsze niż przy pomocy eliminacji Gaussa.