Obliczenia Naukowe Lista 4

Paweł Prusisz

05.12.2021

1 Zadanie 1

1.1 Opis problemu

Naszym zadaniem jest napisanie programu, który w sposób efektywny oblicza oraz przechowuje równanie $Ax = b$, gdzie dla danej macierzy współczynników $A \in \mathbb{R}^{n \times n}$ i wektora prawych stron $b \in \mathbb{R}^n$. Macierz A jest rzadką, tj. mającą dużą elementów zerowych i blokową o następującej strukturze:

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

$v = n$, zakładając, że n jest podzielne przez l, gdzie $l \geq 2$ jest rozmiarem wszystkich kwadratowych macierzy wewnętrznych (bloków): A_k, B_k i C_k. Mianowicie, $A_k \in \mathbb{R}^{l \times l}$, $k = 1, ..., v$ jest macierzą gęstą, 0 jest kwadratową macierzą zerową stopnia l macierz $B_k \in \mathbb{R}^{l \times l}$, $k = 2, ..., v$ jest następującej postaci:

$$B_k = \begin{bmatrix} 0 & \cdots & 0 & b^k_{1,l-1} & b^k_{1,l} \\ 0 & \cdots & 0 & b^k_{2,l-1} & b^k_{2,l} \\ \vdots & & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & b^k_{l,l-1} & b^k_{l,l} \end{bmatrix} \tag{2}$$

B_K ma tylko dwie ostatnie kolumny niezerowe. Natomiast $C_k \in \mathbb{R}^{l \times l}$

$k = 1, ..., v-1$ jest macierzą diagonalną:

$$C_k = \begin{bmatrix} c_1^k & 0 & 0 & 0 & 0 \\ 0 & c_2^k & 0 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & c_{l-1}^k & 0 \\ 0 & \cdots & 0 & 0 & c_l^k \end{bmatrix} \tag{3}$$

1.2 Rozwiązanie

1.2.1 Przechowywanie macierzy rzadkich

Jako, że rozmiar macierzy A jest w założeniu bardzo duży, przechowywanie w pamięci tablicy dwuwymiarowej $n \times n$ jest bardzo niewydajną metodą, biorąc pod uwagę, że znacząca część elementów to 0. Aby rozwiązać ten problem i przechowywać A efektywniej używam pakietu SparseArrays w języku Julia. Umożliwia on efektywne zapamiętywanie macierzy rzadkich. Zamiast tablicy dwuwymiarowej, której złożoność pamięciowa to $O(n^2)$, pamiętamy 3 tablice jednowymiarowe: columns, rows i values, a więc złożoność pamięciowa $O(3n) = O(n)$. Odwołując się do $columns[i], rows[i], values[i]$ możemy odczytać współrzędne, gdzie znajduje się czytany element oraz wartość macierzy w tym punkcie.

1.2.2 Algorytm Eliminacji Gaussa

Do obliczenia równania $Ax = b$ będziemy stosować algorytm eliminacji Gaussa, z modyfikacją uwzględniającą specyficzną postać macierzy A w naszym zadaniu. Za pomocą elementarnych operacjach elementarnych: odejmowania, dodawania lub mnożenia przez stałą, modyfikujemy macierz A do postaci macierzy trójkątnej, czyli zerując wszystkie komórki macierzy pod przekątną. W tym celu będziemy iteracyjnie zerować wszystkie elementy z kolejnych wierszy znajdujące się pod przekątną. Przy rozwiązywaniu macierzy A będziemy przeprowadzać jednoczesnie te same operacje na wektorze stron prawych b. Otrzymany w ten sposób układ równań jest równoważny z układem startowym. Do obliczenia wektora wynikowego x skorzystamy z algorytmu podstawiania wstecz, który ma następującą postać:

$x_i = \frac{b_i - \sum_{j=i+1}^{n} a_{ij}}{aii}$

Powyższy sposób zakłada, że wszystkie elementy znajdujące się na przekątnej są niezerowe, w przypadku kiedy nie ma to miejsca należy zastosować częściowy

wybór elementu głównego. Polega on na wybieraniu w każdym kroku rzędu z elementem głównym, czyli elementem o największej wartości bezwzględnej, który podmieniamy z aktualnie rozpatrywanym rzędem. W ten sposób zapobiegamy występowaniu zer na przekątnej kosztem wykonywania dodatkowych obliczeń.

1.2.3 Optymalizacja Algorytmu Gaussa pod macierze rzadkie

W naszym przypadku, macierz A ma swoją specyficzną postać, którą możemy wykorzystać do wprowadzenia optymalizacji w naszym algorytmie. Mianowicie będziemy korzystać z faktu, iż prawie wszytkie elementy pod przekątną macierzy A są już wyzerowane, a my musimy wyzerować tylko komórki pod samą przekątną maksymalnie o wielkość bloku l. Dodatkowo wiemy, że wszystkie elementy w kolumnie pod elementem, który zerujemy są już wyzerowane, co oznacza, że na wyzerowanie danego rzędu potrzeba $O(l^2)$ operacji, co przy zerowaniu wszystkich rzędów daje złożoność $O(nl^2)$. W przypadku kiedy potraktujemy wielkość bloku jako wartość stałą dostaniemy złożoność $O(n)$.

Analogiczne optymalizacje można zastosować w przypadku wyboru elementu głównego, który nie musi rozpatrywać wszytkich elementów, gdyż są one w większości zerowe, a dodawanie elementów zerowych nie wpływa na wynik.

1.3 Testowanie rozwiązania

Wyniki dla testów podanych na stronie prezentują się następująco.

Dla metody Gaussa z częściowym wyborem elementu głównego:

n	16	10000	50000	100000
czas	5.3551e-5	0.18797914	8.759997823	64.511041926
pamięć	3680	1839872	800160	12799968
błąd względny	2.8576114088871287e-16	4.443944163031559e-16	4.556461336521988e-16	4.487710435687954e-16

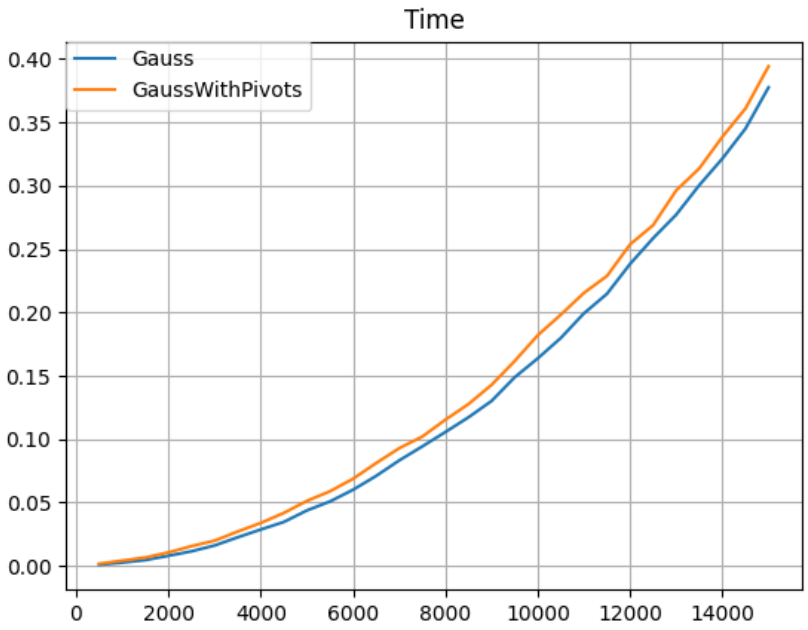
Dla metody Gaussa:

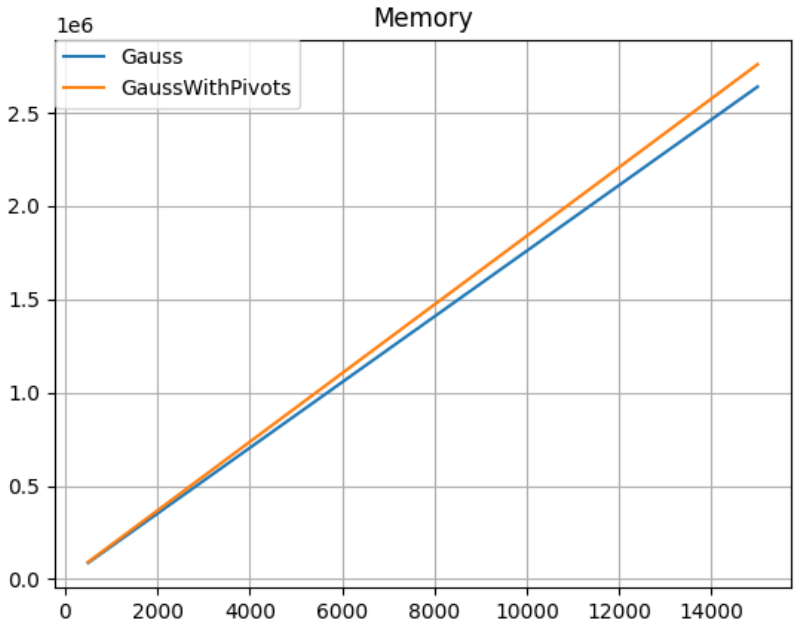
n	16	10000	50000	100000
czas	2.657e-5	0.163386146	5.995779368	62.467678887
pamięć	3472	1759792	5999888	11999888
błąd względny	1.3189010295585304e-15	5.166229763031875e-12	4.9209223217318804e-14	3.9634335264150364e-14

Jak widać metoda Gaussa zużywa nieco mniej pamięci oraz jest minimalnie szybsza niż metoda Gaussa z częściowym wyborem elementu głównego. Wynika

to z faktu, iż wybieranie elementu głównego wymaga dodatkowych obiczeń i pamięci.

Poniżej zamieszczam wykresy czasu i pamięci w zależności od wielkości macierzy A. Testy generowane były za pomocą funkcji blockmat z pakietu matrixgen. Wywołania blockat miały postać blockmat(size, blockSize, 1.0, "tmp.txt"), gdzie size zmieniała się od 500 do 15000, a blockSize miał wartość 4, podobnie jak w testach zamieszczonych na stronie.





Wykresy były generowane za pomocą funkcji plot z pakiety PyPlot.

Złożoność pamięciowa rośnie liniowo zgodnie z przewidywaniami teoretycznymi. Co do złożoność obliczeniowej widzimy, że wykres nie przedstawia funkcji liniowej. Wynika to z faktu, iż w naszych założeniach poczyniliśmy błąd, mianowicie dostęp do elementu macierzy przedsawionej jako SparseArray nie jest w czasie stałym tylko liniowym.