Obliczenia Naukowe 5

Arian Herda

7 January 2024

1 Wstęp

Problem przedstawiony na liście polegał na rozwiązaniu układu równań liniowych przedstawionego równaniem $Ax = b$, gdzie $A = IR^{n \times n}$ jest macierzą rzadką i blokową:

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

$v = \frac{n}{l}$ gdzie l jest rozmiarem macierzy wewnętrznych (bloków).

- $A_i = IR^{n \times n}$ jest macierzą gęstą oraz kwadratową
- $B_i = IR^{n \times n}$ jest macierzą której wyłącznie ostatnia kolumna składa się z wartości niezerowych
- $V_i = IR^{n \times n}$ jest macierzą diagonalną

Wszytskie pozostałe komórki tej macierzy są zerami. Naszym zadaniem było znalezienie sposobu na przechowywanie i przetwarzanie danej macierzy w jak najszybszy i najmniej kosztowny obliczeniowo sposób.

2 Rozwiązanie

2.1 Struktura

Zaproponowany przez mnie sposób przechowyania takiej macierzy jest zaimplementowany w pliku "sparse$_m atrix.jl$"$orazwmoduleSparseMatrix.JesttostrukturaSparseMatrixMyopolach$

n - wielkość macierzy

l - wielkość macierzy wewnętrznej (bloku)

data - słownik języka Julia, który jest zaimplementowany poprzez hash mapę o kluczach typu (UInt64, Uint64) i wartościach typu Float64

Pomysł na wykorzystanie słownika wywodzi się z tego że hash mapa jest doskonałym typem jeżeli cenimy sobie czas dostępu i zmiany wartości dla danych, średnia asymptotyka tych czynności to $O(1)$ a w najgorszym wypadku $O(n)$ w wypadku kolizji, ale takie przypadki zdarzają się bardzo rzadko.

W podanym wyżej pliku została zaimplemetowana także struktura SparseMatrixPlus która wykorzystuje SparseMatrixCSC z biblioteki SparseArrays. Służyłą ona do porównywania i sprawdzania czy SparseMatrixMy działa poprawnie i odpowiednio szybko.

2.2 Algorytmy

2.2.1 Metoda eliminacji Gaussa

Podastawową funkcją która będzie obliczałą nam rozwiązanie jest metoda eliminacji Gaussa. Algorytm ten polega na modyfikacji macierzy tak aby stworzyć macierz trójkątną górną. Aby otrzymać taki efekt wykonujemy odejmowanie kolejnych wierszy pomnożonych przez odpoweidnio wyliczony współczynnik:

$$[a_{j1}, a_{j2}, ..., a_{jl}, b_j] = [a_{j1}, a_{j2}, ..., a_{jl}, b_j] - \frac{a_{ji}}{a_{ii}}[a_{i1}, a_{i2}, ..., a_{il}, b_i]$$

Ważnym faktem jest że algorytm ten zawodzi w wypadku gdy wartość a_{ii} wynosi 0. Po wykonaniu n - 1 iteracji takiego algorytmu otrzymujemy macierz trójkątną górną której rozwiązanie będzie już dużo szybsze i łatwiejsze.

Po ukończeniu tej części należy faktyczne rozwiązanie układu równań z powstałą macierzą i wektorem prawych stron. Aby tego dokonać iterujemy w dół od n do 1 wyliczając ostatni współczynnik wektora rozwiązać:

$$x_n = \frac{b_n}{a_{nn}}$$

a następnie wyliczamy wcześniejsze współczynniki odejmując od odpowiedniego współczynnika b sumę już wyliczonych współczynników rozwiązania i dzieląc przez odpowiedni współczynnik z macierzy A:

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

Algorytm ma złożoność $O(n^3)$. Ale po analizie i zrozumieniu jak wygląda nasza macierz możemy zauważyć że w każdej kolumnie będzie conajwyżej l wartości niezerowych pod główną diagonalą. Na dodatek do tego w kązdym

kroku k k-ty wiersz ma po lewej od głównej przekoątnej już same zera, a po prawej znajduje się dokłądnie l ciągłych niezerowych wartości. Taka obserwacja pozwala nam na odejmowanie tylko l wierszy w każdym kroku. Podobnie w drugim etapie wyliczania wektora rozwiązań zauważamy że w każdym wierszu mamy maksymalnie $l - 1$ elementów. Po ograniczeniu sumowania w drugiej fazie algorytmu dostajemy złożoność $O(n)$ każdej z cześci a tym samym całego algorytmu.

2.2.2 Metoda eliminacji Gaussa z częściowym wyborem elementu głównego

Algorytm ten jest rozszerzeniem metody Gaussa z poprzedniego podpunktu. Rowziązuje on kilka problemów, między innymi ten z niemożliwością korzystania z poprzedniego algorytmu jeśli którykolwiek współczynnik na głównej przekątnej jest zerem, a także problem z dokładnością działąń przy małaych współczynnikach na głównej diagonali. Element główny to współczynnik z którego korzystamy w k-tym kroku do wyzerowania wszytskich innych współczynników w k-tej kolumnie. Dla podstawowej metody Gaussa zawsze wybieraliśmy a_{kk} a teraz wybieramy największy element w kolumnie k poniżej głównej diagonali czyli taki element a_{pk}, że:

$$|a_{pk}| = \max_{k \leq i \leq n} |a_{ik}|$$

Następnie dokonujemy zamiany wierszy p oraz k (lub zapamiętujemy tylko p by oszczędzić czas kosztem minimalnej pamięci) i wykonujemy dalszą część algorytmu jak w noramlnym algorytmie eliminacji Gaussa.

Większość optymalizacji z popszedniego podpunktu cały czas mogą być zaimplementowanie i przy szukaniu największego elemntu ze złożonością $O(n)$ otrzymujemy złożoność całego algorytmu $O(n^2)$. Natomiast podobnie jak w poprzednim algorytmie możemy zauważyć, że ilość niezerowych wartości poniżej głównej diagonali może być maksymalnie l. Po takiej obserwacji możemy zawęzić liczbę porównań do l. Przy wcześniej wymienionych optymalizacjach algorytmu Gaussa powstaje jednak problem, ze względu na to, że w k-tym kroku element główny może być w $k + l$-tym wierszu, wartości niezerowe mogą występować aż do $k + 2l$-tej kolumny. Mimo że jest tych obliczeń dwa razy więcej więcej niż w przy metodzie Gaussa, cały czas jest to stałą ilość. Wszytskie te optymalizacje sprawiają że algorytm ma również złożoność $O(n)$.

2.2.3 Rozkład LU

Rozkład LU to rozkład macierzy A na macierz trójkątną dolną L (Lower) i trójkątną górną U (Upper) tak że $A = LU$. W wyniku takiego rozkładu mamy

$$Ax = LUx = b$$

a następnie dzieli się to nam na dwa dużo prosztsze układy

$$Lz = b$$

$$Ux = z$$

Przy analizowaniu metody Gaussa widać na pierwszy rzut oka, że wykonując eliminacje tworzymy od razu macierz górnotrójkątną U. Zapisując współczynniki odejmowania wierszy w dolnej części macierzy tworzymy macierz dolnotrójkątną L i oszczędzamy pamięć nie musząc przechowywać nowej, osobnej macierzy. Współczynnik użyty do wyzerowania a_{ij} zostaje zapisany w tym samym miejscu i w ten sposób cały rozkład LU mamy zapisany w jednej macierzy.

Część druga tego algorytmu składa się z wyznaczania wektora rozwiązań x poprzez rozwiązywanie układu równań $Lz = b$ oraz $Ux = z$.

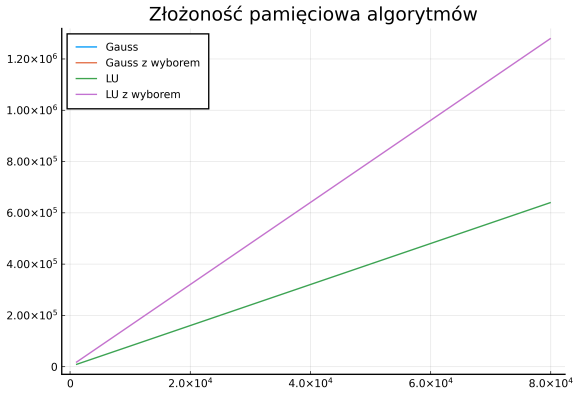
Z racji na podobieństwo tej metody do normalnej metody eliminacji Gaussa asymptotyka jest taka sama czyli $O(n^3)$ a po zastosowaniu dokładnie tych samych optymalizacji co w przypadku podstawowego algorytmu również otrzymujemy asymptotykę $O(n)$.

2.2.4 Rozkład LU z częściowym wyborem elementu głównego

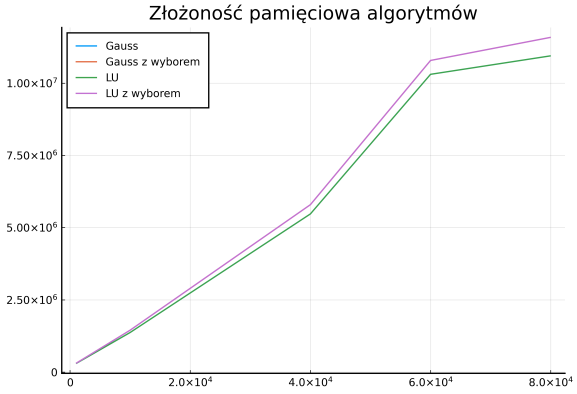
Metoda ta jest połączeniem metod LU z metodą Gaussa z cześciowym wyborem elemntu głównego. Wykorzystuje oba koncepty, wybierania największego elementu z kolumny jako element główny,następbnie tworzenie z niego współczynnik do zerowania innych współczynników a następnie podstawaia ten współczynnik tworząc w ten sposób rozkład LU. Ważne jest by zapamiętywać punkty główne gdyż są one potrzebne przy wyznaczaniu wektora rozwiązań x.

Z racji że metoda ta jest połączeniem wcześniej już tłumaczonych przeze mnie metod można szybko wywnioskować że jej złożoność to $O(n^3)$ a po zastosowaniu optymalizacji jak w podpunkcie 2.2.2 zmniejszamy tą złożoność ponowanie do $O(n)$.

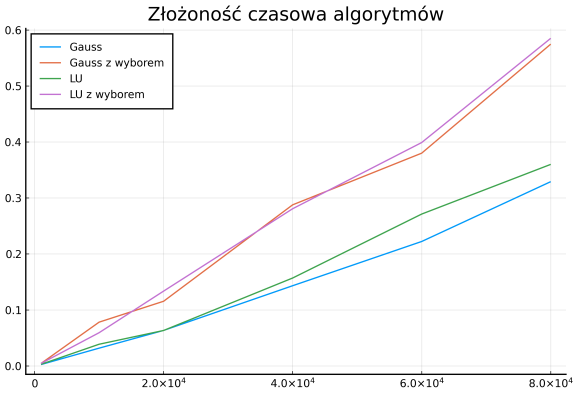
3 Wyniki i interpretacja



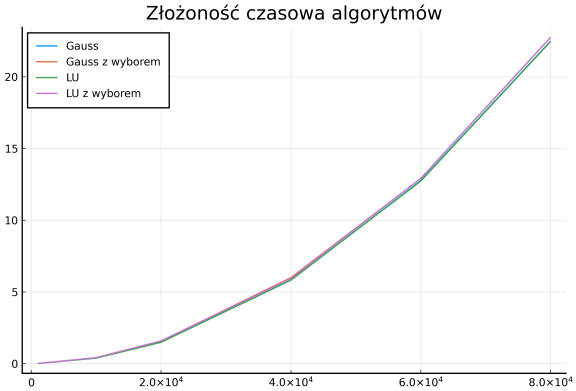
Rysunek 1: Wykres Pokazujący zużycie pamieći przy użyciu mojej struktury.



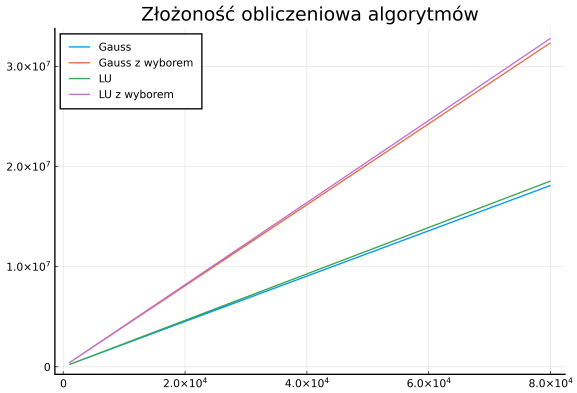
Rysunek 2: Wykres Pokazujący zużycie pamieći przy użyciu struktury z biblioteki SparseArrays.



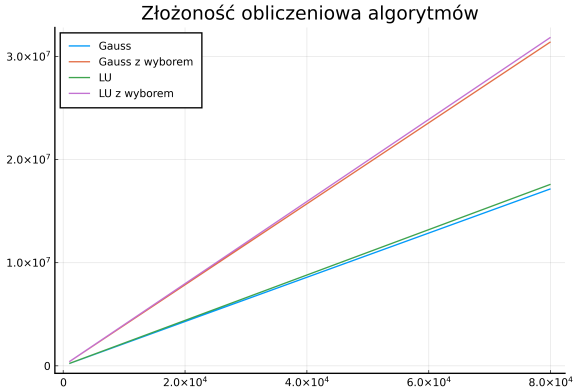
Rysunek 3: Wykres pokazujący złożoność czasową przy użyciu mojej struktury.



Rysunek 4: Wykres pokazujący złożoność czasową przy użyciu struktury z biblioteki SparseArrays.



Rysunek 5: Wykres pokazujący złożoność obliczeniową przy użyciu mojej struktury.



Rysunek 6: Wykres pokazujący złożoność czasową przy użyciu struktury z biblioteki SparseArrays.

Jak widać obie struktury są praktycznie takie same jeśli chodzi o liczbę wykonanych operacji. Jednakże już na wykresie wykorzystanej pamięci widać małą różnicę na korzyść mojej struktury SparseMatrixMy. Patrząc na wykres złożoności czasowej widać już jednoznacznie jaką różnicę robi wykorzystanie hash mapy, dzięki niej dostęp do elementów wykonuje się w stałym czasie, w przeciwieństwie do SparseMatrixCSC w której dostęp do elementów jest w czasie $O(n)$. Różnica ta sprawia że złożoność czasowa algorytmów wykorzystujących tą drugą strukturę zmienia się z $O(n)$ na $O(n^2)$. Pomijając tą różnicę widać na wykresach stworzonych przy użyciu mojej struktury że pamięć czas oraz operacje zwiększają się liniowo względem ilości elementów w macierzy co było celem wszytskich optymalizacji.

Przy poszczególnych funkcjach widać takie różnice, że zazwyczaj funkcje

z wyborem miejsc głównych zajmują więcej miejsca w pamięci a także wykonuje się przy nich więcej operacji. W związku z tym że wykonują więcej operacji zajmują też więcej czasu.

4 Wnioski

Przy rozwiązywaniu problemu należy dokładnie analizować dane z jakimi mamy doczynienia. Dzięki takiej analizie i odpowiedniej strukturze, w przypadku podanym na liście byliśmy w stanie zaoszczędzić ogromną ilość cennej pamięci a także mocy obliczeniowej która może przydać się do szybszego rozwiązywania takich problemów. Dzięki wcześniejszej analizie danych byliśmy w stanie dopasować znane już nam algorytmy do naszych potrzeb i tym samym zmieniliośmy ich złożoność z $O(n^3)$ do $O(n)$ oszczędzając ogromną ilość czasu i przyspieszając nasz program. Taka wcześniejsza analiza danych pozwala na efektywne rozwiązywanie problemów.