Obliczenia naukowe - lista piąta

Bartosz Rajczyk

6 stycznia 2020

1 Wstęp

Problem przedstawiony na liście zadań polegał na rozwiązywaniu układu równań liniowych postaci

$$Ax = b$$

gdzie A jest macierzą kwadratową liczb rzeczywistych o rozmiarze n, a b jest wektorem stron prawych. Macierz A posiada blokową, rzadką strukturę prezentującą się nastepująco:

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

gdzie A_i, B_i oraz C_i są podmacierzami kwadratowymi o rozmiarze $l \geqslant 2$ (rozmiar ten nazywany jest także "rozmiarem bloku") o określonych charakterystykach:

- A_i są w całości niezerowe
- B_i mają niezerowe dwie ostatnie kolumny
- C_i mają niezerową przekątną

reszta komórek tych macierzy jest zerami. Problemem postawionym przed nami jest znalezienie wydajnego sposobu przechowywania i rozwiązywania przestawionych równań.

2 Rozwiązanie

2.1 Przechowywanie

Jako że rozmiar macierzy A jest w założeniu bardzo duży, klasyczny sposób przechowywania jej w pamięci w postaci tablicy dwuwymiarowej zajmowałby $O(n^2)$ miejsca. Nie jest to sposób wydajny, jako że z treści problemu wiemy, że macierz jest bardzo rzadka i większość tego miejsca wypełniałyby zera nie niosące żadnej informacji. Rozwiązaniem tego problemu może być zapamiętywanie jedynie komórek niezerowych.

Język Julia udostępnia w bibliotece standardowej pakiet SparseArrays, który umożliwia wydajne pamięciowo przechowywanie takich macierzy w strukturze danych nazywanej SparseMatrixSCS. Przetrzymuje ona trzy tablice o równych rozmiarach, z czego jedna zawiera numery kolumn, druga numery rzędów, a trzecia wartości. Czytając każdą z nich na tym samym indeksie otrzymujemy wartość przechowywanej macierzy na przechowywanej kolumnie i wierszu - w ten sposób możemy także wydajnie iterować po macierzy, nie musząc uwzględniać w ogóle komórek zerowych.

2.2 Podstawy teoretyczne

2.2.1 Eliminacja Gaussa

Podstawowym narzędziem wykorzystywanym w rozwiązaniu jest metoda eliminacji Gaussa. Jest to algorytm o wielu zastosowaniach, umożliwiający zarówno rozwiązywanie układów równań liniowych, jak i wyznaczanie rozkładku LU. Wykorzystuje on jedynie operacje elementarne, jak odejmowanie czy dodawanie wielokrotności rzędów i kolumn macierzy.

Rozwiązywanie przy tej pomocy modyfikuje macierz w celu uzyskania macierzy trójkątnej, tj. zeruje wszystkie komórki macierzy znajdujące się pod jej przekątną. W tym celu iteracyjnie zerować będziemy kolejne znajdujące się tam elementy, dla przykładu w celu wyzerowania elementów w pierwszym wierszu poniżej diagonali odejmiemy od wiersza i wiersz 1 przemnożony przez $z = a_i 1/a_1 1$. Przy rozwiązywaniu układu

równań takie przeszktałcenia uwzględniają także odejmowanie i dodawanie w ramach wektora stron prawych. Otrzymywany w ten sposób układ równań z macierzą trójkątną jest równoważny układowi pierwotnemu.

Zauważmy jednak że powyższy sposób działać będzie jedynie w przypadku, kiedy elementy znajdujące się na przekątnej (w powyższym przykładzie chociażby a_11, w ogólności jakieś a_kk) nie są zerowe. W takim przypadku dodaje się tzw. częściowy wybór elementu głównego. Polega on na wybieraniu w każdym kroku rzędu z elementem głównym, czyli elementem o największej wartości bezwzględnej, który wtedy podmieniamy z aktualnie rozpatrywanym rzędem. W ten sposób zapobiegamy występowaniu zera na diagonali, lecz znalezienie i podmiana zajmuje trochę czasu.

W celu rozwiązania układu równań po przekształceniu go przez eliminację Gaussa korzystamy z algorytmu podstawiania wstecz, który zwraca wektor wynikowy $x_1, x_2, x_3, ..., x_n$ gdzie:

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

W swojej standardowej postaci algorytm rozwiązywania układów równań metodą eliminacji Gaussa ma złożoność czasową $O(n^3)$, a algorytm podstawiania wstecz - $O(n^2)$, co daje łączną złożoność $O(n^3)$.

2.2.2 Rozkład LU

Korzystając z alogorytmu eliminacji Gaussa możemy także wykonać tzw. rozkład LU, część metody LU będącej innym sposobem rozwiązywania układu równań liniowych. Rozkład ten polega na utworzeniu dwóch macierzy, L oraz U, będących kolejno macierzą trójkątną dolną oraz górą takich, że $A = L*U$, gdzie produkty są postaci:

$$L = \begin{pmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ l_{n1} & l_n2 & \cdots & l_{nn} \end{pmatrix}$$

$$U = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{pmatrix}$$

Wtedy rozwiązanie równania

$$A * x = y$$

$$L * U * z = y$$

gdzie x, y to wektory, sprowadza się do rozwiązania dwóch, prostych równań (bo z macierzami trójkątnymi)

- $L * z = y$
- $U * x = z$

Rozkładu tego dokonuje się właśnie przy użyciu metody eliminacji Gaussa, gdzie U jest macierzą uzyskiwaną w klasycznym rozkładzie, a L zapełniana jest przez zapamiętywanie mnożników użytych do zerowania elementów; mnożnik użyty do wyzerowania a_{ij} zostanie zapamiętany w analogicznej komórce l_{ij} w macierzy L. Także dla tej metody istnieje modyfikacja z wyborem elementu głównego w celu uniknięcia błędów związanych z zerami i jest ona analogiczna jak w poprzedniej metodzie.

Złożoność obliczeniowa rozkładu jest równa złożoności eliminacji Gaussa, czyli wynosi $O(n^3)$, natomiast złożoność rozwiązania otrzymanego układu z U, L wynosi $O(n^2)$. Dużą zaletą tej metody jest natomisat to, że wektor stron prawych pojawia się jedynie w jednym z pary równań; oznacza to, że przy jego zmianie nie musimy dokonywać ponownych obliczeń dla obu równań, a jedynie trzeba przeliczyć jedno z nich, drugie zachowując jako stałą.

2.3 Optymalizacja pod nasz przypadek

2.3.1 Eliminacja Gaussa

Charakterystyczna struktura rozpatrywanej macierzy umożliwi nam poczynienie znacznych optymalizacji używanych przez nas sposobów. Należy zwrócić przede wszystkim uwagę na to, że nie musimy zerować *wszystkich* elementów kolumn znajdujących się pod diagonalą, jako że większość z nich jest już na początku zerowa. Rozpatrzmy więc, jakie tak naprawdę powinny być bezpieczne zakresy naszej iteracji.

Poruszając się po kolejnych kolumnach nie musimy rozpatrywać wszystkich rzędów w tych kolumnach; maksymalnym wychyleniem pod diagonalę będzie rozmiar bloku (rozmiar macierzy blokowych tworzących naszą macierz). Równocześnie musimy odjąć nasz mnożnik jedynie tyle razy od kolejnych rzędów, aby poprawnie przeprowadzić eliminację, jako że rzędy niższe, znów, już są zerowe. To przekłada się na złożoność eliminacji postaci $O(nl^2)$, a jeżeli l jest stałe - $O(n)$. Analogiczne optymalizacje można poczynić dla wyboru elementu głównego. Rozwiązywanie znowu nie musi rozpatrywać w tworzonej sumie wszystkich rzędów, jako że dodawanie zera nie wpływa na wynik.

2.3.2 Rozkład LU

Ponownie jak w poprzedniej metodzie, ważne jest uwzględnienie specyficznej postaci rozpatrywanej macierzy i ustalenie indeksów, po których należy iterować, które będą analogiczne jak w poprzednim punkcie. Znowu daje nam to złożoność liniową, różniącą się jedynie o stałą, bowiem rozkład LU musi wykonać trochę więcej operacji.

3 Testy i wyniki

3.1 Metodologia

W celu poprawnego zmierzenia wydajności otrzymanych algorytmów została opracowana metodologia badań mająca na celu wyeliminowanie możliwych nieścisłości w badaniach. W tym celu:

- został zmodyfikowany algorytm generowania macierzy dostarczony w module `matrixgen` przez doktora Zielińskiego w taki sposób, aby nie zapisywał on macierzy do pliku, a zwracał obiekt `SparseMatrixSCS` używany przez algorytmy; oszczędza to nam używania obciążających operacji IO
- algorytmy zostały przystosowane do działania *in-place*, tj. bez kopiowania i tworzenia własnych tablic, a raczej modyfikowania przekazywanych im jako argumenty; kopiowanie i tworzenie nowych macierzy znajduje się poza testami wydajnościowymi
- przeprowadzenie testów dla powiększających się o określony krok rozmiarów macierzy; od 400 do 15 000 z krokiem 400
- 25-krotne powtórzenie każdego testu w celu stabilizacji otrzymywanych wyników

W ten sposób udało się uzyskać możliwie zgodne z rzeczywistością wyniki.

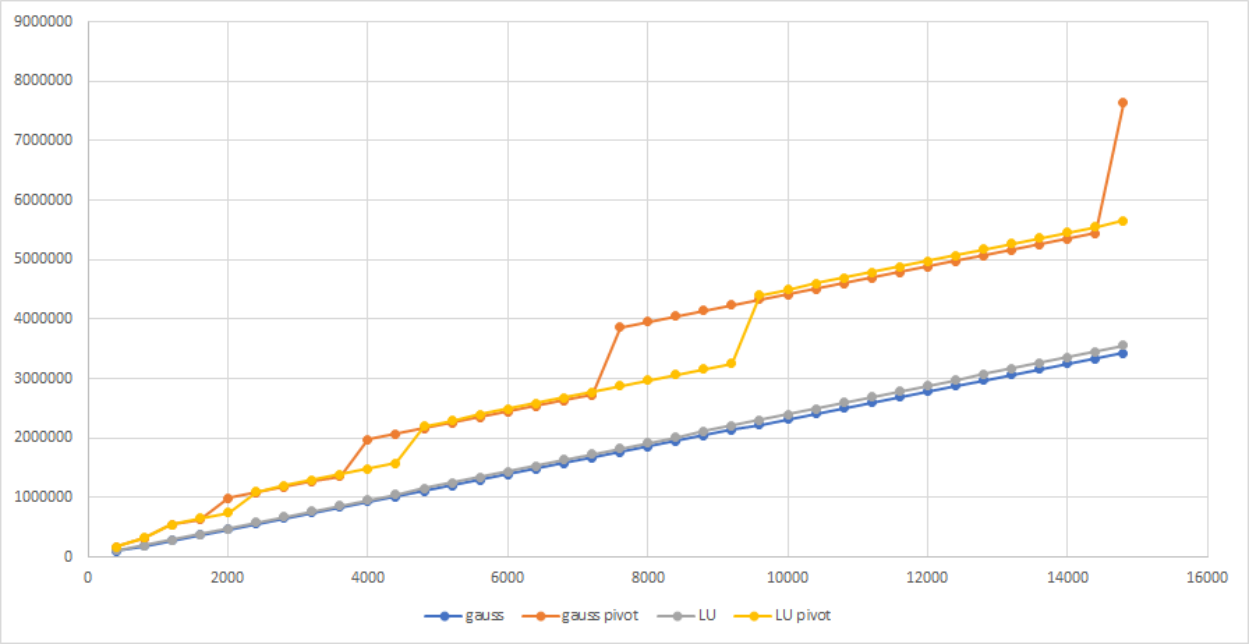
3.2 Wyniki

Poniższe wykresy prezentują wizualizację wyników uzyskanych przy pomocy metodologii opisanej powyżej.

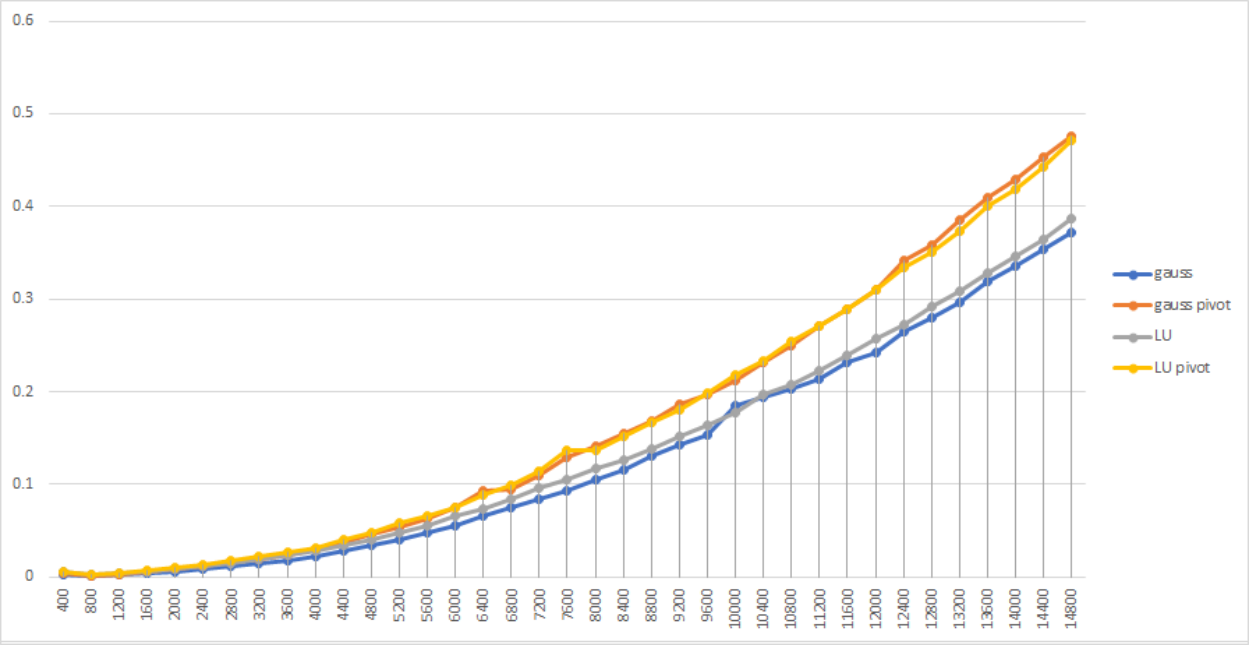
3.3 Interpretacja

3.3.1 Analiza wyników

Założeniem optymalizacji przeprowadzonych dla konkretnego przypadku naszej macierzy było doprowadzenie wydajności algorytmów do poziomu $O(n)$, co jest widoczne na wykresie pamięci, ale czemu zdaje zaprzeczać się wykres czasu. Nie prezentuje on także $O(n^3)$ oczekiwanego jeżeli zastosowalibyśmy klasyczne, niedostosowane algorytmy.



Rysunek 1: Wykres pamięci



Rysunek 2: Wykres czasu

Drugą kwestią jest zależność pomiędzy czasami kolejnych użytych algorytmów; każdy z nich należy teoretycznie do $O(n)$, ale każdy wykonuje też inną liczbę operacji; algorytmy z wyborem elementu głównego (oznaczone przez *pivot*) na wykresach muszą przez obliczeniem wykonać permutacje rzędów macierzy, stąd ich niższa wydajność względem odpowiedników bez tego. Natomiast algorytmy LU przegrywają z używaną przez nas modyfikacją Gaussa z powodu tego, że muszą one realnie wypełniać dwie macierze zamiast jednej, a następnie rozwiązywać obie. LU zyskałyby względem Gaussa, gdybyśmy używali wspomnianej wcześniej optymalizacji z liczeniem tylko równania ze stroną prawą przy jej zmianach. Nie było to jednak możliwe w zadanym środowisku testowym, gdzie strony prawe zależały od oryginalnej macierzy.

3.3.2 Powody rozbieżności od oczekiwań

Powodu rozbieżności pomiędzy wydajnością liniową a tą uzyskiwaną przez nas należy doszukiwać się najpewniej w kwestiach czysto technicznych; zakładamy bowiem, że używana przez nas struktura danych `SparseMatrixSCS` ma dostępy rzędu $O(1)$, co niekoniecznie musi być prawdą - trudno jednak znaleźć jakieś precyzyjne informacje o jej wydajności w specyfikacji.

Wypada także zaadresować "schodki" pojawiające się na wykresach dotyczących pamięci przy funkcjach z wyborem elementu głównego. Przy dokładniejszym przyjrzeniu się ich rozkładowi względem rozmiarowi macierzy możemy zaobserwować, chociażby dla `gaussPivot`:

Rozmiary	**rozmiar skoku**	**rozmiar względny**
1600 → 2000	634240 → 989312	1.59
3600 → 4000	1360464 → 1977680	1.47
7200 → 7600	2720080 → 3861584	1.41
14400 → 14800	5439056 → 7629008	1.40

Dla porównania średni skok dla wszystkich pozostałych par sąsiednich punktów wynosi około 92000 w przeciwieństwie do większych i rosnących skoków tu. Częstotliwość skoków także zdaje się rosnąć ekspotencjalnie, mniej więcej w okolicach c^n, gdzie c jest jakąś stałą. Najbardziej prawdopodobnym więc wytłumaczeniem jest to, że język Julia implementuje tablice jako tablice dynamiczne powiększające swój rozmiar o pewien współczynnik przy przekroczeniu danej liczby elementów; to wyjaśniałoby skoki w ekspotencjalnych interwałach o jakiś współczynnik względem poprzedniego; algorytmy z wyborem elementu głównego używają jednej więcej tablicy do trzymania permutacji.

4 Wnioski

Z przeprowadzonej analizy i testów możemy wyciągnąć wywnioskować, jak ważne jest dostosowanie istniejących metod rozwiązywania problemów pod zastane przez nas warunki. Przy pomocy krótkiej analizy indeksów używanych przez nas w iterowaniu przez macierze mogliśmy zaoszczędzić znaczną ilość czasu potrzebnego na dokończenie obliczeń.