Obliczenia Naukowe

Lista 5

Yuliia Melnyk
246202

11 stycznia 2023

1 Załączone pliki

- blocksys.jl - zawiera moduł z implementacjami zadanych na liście algorytmów
- unitTests.jl - zawiera testy jednostkowe algorytmów
- TMtests.jl - zawiera testy przeznaczone do wyliczenia straconego czasu i pamięci dla każdego algorytmu

2 Problem

Musimy rozwiązać układ równań $Ax = b$. Mamy podaną macierz współczynników kwadratową. Dla tego że macierz jest rzadka mamy wymyślić jakiś sposób na efektywne obliczanie, gdyż korzystanie z zwykłego sposobu liczenia bardzo szybko powoduje zużycie bardzo dużej ilości czasu i pamięci komputera.

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

Macierz zawiera v bloków (gdzie $v = n/l$), każdy ma swoją specyficzną strukturę. Każdy z bloków również jest kwadratową macierzą, co ułatwia nam liczenie. Pod-macierz B_k ma postać:

$$B_k = \begin{bmatrix} b_{11}^k & \dots & b_{1l-2}^k & b_{1l-1}^k & b_{1l}^k \\ 0 & \dots & 0 & 0 & b_{2l}^k \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & 0 & 0 & b_{ll}^k \end{bmatrix}$$

C_k jest postaći:

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

Natomiast macierz A_k jest gęsta, tzn. niezerowa. Symbole n i l będą używane jako rozmiar macierzy i rozmiar bloków. Do efektywnego przechowywania macierzy rzadkiej A została wykorzystana biblioteka języku Julia *SparseArrays*. Pamięta ona tylko niezerowe elementy. Możemy założyć że

dostęp do elementów macierzy mamy wtedy w czasie stałym. (powszechnie wiadomo jednak, że tak nie jest).

3 Eliminacja Gaussa

3.1 Bez wyboru elementu głównego

3.1.1 Opis algorytmu

Ten algorytm można nazwać najłatwiejszym z zaimplementowanych. Polega on na tym żeby wyeliminować wszystkie elementy pod przekątną, tworząc macierz górnotrójkątną. Eliminacja przechodzi w kilka kroków.

1) Dzielenie elementu eliminowanego przez element na przekątnej. Dla pierwszej kolumny będzie to wyglądać następująco:

$$l_{i1} = \frac{a^{(1)}_{i1}}{a^{(1)_{11}}} i = 2, \ldots, n$$

2) Przemnażanie pierwszego równania przez wynik dzielenia i odejmowanie od pozostałych

Uogólniając dostaniemy równanie takiej postaci:

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

dla każdego $j > i$. Należy pamiętać że algorytm zawodzi gdy mamy sprawę z zerową lub bardzo niewielką wartością leżącą na przekątnej ponieważ występuje ona w mianowniku czynnika, przez który mnożony jest odejmowany wiersz, co może powodować błędy numeryczne. Żeby uniknąć danej sytuacji istnieje wersja algorytmu z częściowym wyborem elementu głównego, opisana dalej.

Po wyliczeniu macierzy przechodzimy do rozwiązania przekształconego układu. Można to łatwo zrobić korzystając z wzorów podanych na wykładzie:

$$1) x_n = \frac{b_n}{a_{nn}}$$

$$2) x_i = \frac{b_i - \sum_{j=i+1}^{n} x_j}{a_{ii}}$$

3.1.2 Złożoność i optymalizacje

Standardowa wersja eliminacji Gaussa ma złożoność $O(n^3)$ dla tego że k-ty wiersz (do $n-1$) musimy odjąć od $(n-k)$ wierszy poniżej, a każde takie odejmowanie to aktualizacja każdej wartości tego wierszu, czyli n.

Możemy jednak zauważyć że w każdej kolumnie mamy maksymalnie l niezerowych wartości pod przekątną (Przez specyficzną strukturę B_k, w punktach skrajnych lewych i prawych pod-macierzy, w innych miejscach jest ich nawet mniej: do spodu bloku A i jeden wiersz niżej). Zera nie ma sensu "zerować", ponieważ cel jest już osiągnięty. Możemy więc ograniczyć proces eliminacji wyłącznie do eliminacji elementów niezerowych pod przekątną, a gdy zauważymy że po prawej dla każdego k mamy l wartości, ostatnia z których jest wartość z bloku C_k odejmowanie od pozostałych elementów też możemy ograniczyć. Więc możemy w każdym kroku odejmować tylko od l wierszy, w których ma to sens i aktualizować tylko l wartości, które muszą ulegnąć zmianie. W ten sposób doprowadzamy złożoność algorytmu do $O(n)$

Jako ograniczenie do wyznaczania ostatniego wiersza dla każdej kolumny został wygenerowany taki wzór:

$$min((k+l) - (k \mod l)) - l, n)$$

który w przypadku przybliżenia do końca tablicy zwraca n, a w każdym innym wynik policzonego wzoru. Warto zauważyć że dla tworzenia takich wzorów ograniczających macierz musi posiadać

jakąś cykliczność bloków, powtarzające się kształty. Ze względu na to że ostatnia macierz po prawej to C_k i jej diagonal jest zawsze w równej odległości od każdego elementu a_{kk} możemy również wyprowadzić wzór dla wyznaczania ostatniej niezerowej kolumny w wierszu:

$$min(k + l, n)$$

Rozumowanie jest takie same jak do poprzedniego wzoru.

3.2 Z częściowym wyborem elementu głównego

3.2.1 Opis algorytmu

Uzupełnienie metody eliminacji Gaussa o częściowy wybór elementu głównego pomaga rozwiązać problem z bardzo małymi wartościami na przekątnej macierzy, które mogą powodować błędy w obliczeniach. Początkowym elementem głównym jest wartość, którą w k-tym kroku używamy do wyzerowania pozostałych wartości w kolumnie. W podstawowej wersji algorytmu korzystaliśmy zawsze z a_{kk}.
Częściowy wybór polega na wyznaczaniu takiego wiersza p, że:

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

a następnie zamianie w macierzy A wierszy p i k. Warto zauważyć że w faktycznej implementacji algorytmu wiersze nie są przestawiane, tylko są zamieniane wartości w wektorze permutacji $perm$, w którym początkowo każdy k-ty element oznacza k-ty wiersz. Jeśli program znajdzie wiersz p z większą wartością w kolumnie, dokonuje on zamianę $perm[k]$ na $perm[p]$.
Efekt fazy eliminacji jest identyczny jak w podstawowej wersji - otrzymujemy układ z macierzą górno trójkątną który rozwiązujemy jak wcześniej.

3.2.2 Złożoność i optymalizacje

Nietrudno zauważyć że procedura znalezienia elementu głównego, wykonywana raz w każdym z $(n-1)$ kroków eliminacji, jest $O(n)$, zatem złożoność niezoptymalizowanego algorytmu pozostaje $O(n^3)$. W fazie wyznaczania rozwiązania nie zachodzą większe zmiany, zatem również złożoność się nie zmienia

Stosujemy analogiczne techniki optymalizacji jak w podstawowej wersji. Pod diagonalną mamy maksymalnie l niezerowych wartości, zatem możemy zawęzić pole poszukiwania elementu głównego do stałej liczby iteracji. Podobnie ponieważ wiersz z elementem głównym w k-tym kroku się zamienia, a pod k jest maksymalnie l niezerowych wartości możemy również określić ograniczenie przy odejmowaniu które będzie łagodniejsze niż dopiero. Będzie ono równe:

$$min((k + 2l), n)$$

Wynika to z tego że maksymalny element w innym wierszu w tej samej kolumnie jest w odległości od elementu przekątnej swojej kolumny, a to znaczy że żeby dostać się ostatniego elementu wiersza będziemy potrzebować większego kroku niż l. W skrajnych przypadkach gdy maksymalna będzie w lewym górnym rogu macierzy B_k lub w jej prawym dolnym rogu odległość do ostatniego niezerowego elementu w C_k będzie wynosić dokładnie $k + 2l$. Rozwiązania te pozwalają nam ponownie osiągnąć złożoność $O(n)$, nawet przy łagodniejszym ograniczeniu $k + 2l$

3.3 LU

Rozkładem LU nazywamy taki podział $A = LU$, że L jest macierzą dolno trójkątną, a U - górno trójkątną. Układ równań $Ax = LUx = b$ możemy wówczas rozwiązać jako dwa proste układy z macierzami trójkątnymi:

$$Lz = bUx = z$$

ta dwustopniowość pozwala nam również na wielokrotne rozwiązanie układów dla różnych wektorów prawych stron b, wyznaczając sam rozkład LU tylko raz.

Algorytm opiera się na spostrzeżeniu że metoda eliminacji Gaussa konstruuje macierz U, a po niewielkiej modyfikacji również L. Proces sprowadzania macierzy $A^{(1)}$ do macierzy górno trójkątnej $A^{(n)}$ w $n-1$ krokach możemy zapisać:

$$A^{(n)} = L^{(n-1)} \dots L^{(2)} L^{(1)} A^{(1)}$$

Oznaczając $A^{(n)}$ przez U oraz z tego, że $A = A^{(1)}$ mamy

$$U = L^{(n-1)} \dots L^{(2)} L^{(1)} A$$

$$A = (L^{(n-1)} \dots L^{(2)} L^{(1)})^{-1} U$$

$$A = L^{(1)^{-1}} L^{(2)^{-1}} \dots L^{(n-1)^{-1}} U$$

W k-tym kroku dostajemy:

$$L^{(k)} \cdot \begin{bmatrix} w_1^{(k)} \\ w_2^{(k)} \\ \vdots \\ w_n^{(k)} \end{bmatrix} = \begin{bmatrix} w_1^{(k)} \\ \vdots \\ w_k^{(k)} \\ w_{k+1}^{(k)} - m_{k+1,k} \cdot w_k^{(k)} \vdots \\ w_n - z_{n,k} \cdot w_k^{(k)} \end{bmatrix}$$

gdzie $w_i^{(k)}$ jest i-tym wierszem macierzy A w k-tym kroku eliminacji, a $z_{i,k} = a_{i,k}/a_{k,k}$. Wówczas gdy złożymy wszystkie macierzy przekształcenia dostaniemy:

$$L^{(n-1)^{-1}} \dots L^{(2)^{-1}} L^{(1)^{-1}} = L = \begin{bmatrix} 1 & & & & \\ z_{21} & 1 & & & \\ z_{31} & z_{32} & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ z_{n1} & z_{n2} & \dots & z_{n,n-1} & 1 \end{bmatrix}$$

Jest to macierz dolno trójkątna. Do efektywnego pamiętania LU zamiast korzystać z dwóch tablic możemy:

- Zamiast zerowania wartości pod przekątna zapisywać w tych miejscach mnożniki z.
- Nie zmieniać wartości wektora b w trakcie wyznaczenia rozkładu, tylko przy rozwiązywaniu równania.

Faza wyznaczania wektora składa się z dwóch pętli rozwiązujących $Lz = b$ i $Ux = z$. Są analogiczne do opisanej przy algorytmie eliminacji Gaussa bez wyboru.

3.3.1 Złożoność i optymalizacje

Z uwagi na to że wyznaczanie rozkładu LU a faza eliminacji Gaussa są bardzo podobne, mają one tą samą złożoność $O(n^3)$. Z tego samego powodu korzystając z tych samych optymalizacji możemy zredukować ją do $O(n)$
Pamiętając że L ma na przekątnej same jedynki, wiemy że każdy z_i zawiera składnik b_i. Możemy zatem celem zaoszczędzenia pamięci nadpisywać wektor b, umieszczając w nim rozwiązania pierwszego układu. Macierz przechowująca rozkład LU ma dokładnie tą samą strukturę co macierz A, więc złożoność rozwiązania układów możemy jak w przypadku zwykłej eliminacji Gaussa zmniejszyć do $O(n)$

3.4 Z częściowym wyborem elementu głównego

Jak w przypadku zwykłej eliminacji Gaussa, częściowe wybieranie elementu głównego pomaga nam uniknąć błędy numeryczne w przypadku niewielkiej wartości na przekątnej macierzy A. Dokonujemy podobnych zmian jak w wersji LU bez wyboru i dodajemy wybór elementu głównego podobnie jak w eliminacji Gaussa z wyborem. Przy wyznaczaniu rozkładu musimy też uwzględnić wektor permutacji, będzie on potrzebny do obliczania układów równań.

3.4.1 Złożoność i optymalizacje

Niezoptymalizowana wersja algorytmu ma złożoność $O(n^3)$, optymalizacje jak opisano wcześniej redukują złożoność do $O(n)$. Warto zauważyć że tak samo jak w algorytmie Gaussa z częściowym wyborem musi tutaj wystąpić łagodniejsze ograniczenie z powodu możliwości oddalenia się ostatniego niezerowego elementu w C_k

3.5 Badanie złożoności algorytmów

Korzystając z faktu że macierzy są podanej postaci możemy znacznie zmniejszyć zużycie pamięci komputera i mocno zmniejszyć czas ich wykonania. Też poprawne zaimplementowanie algorytmów pozwala na rozwiązanie układów równań na bardzo dużych macierzach

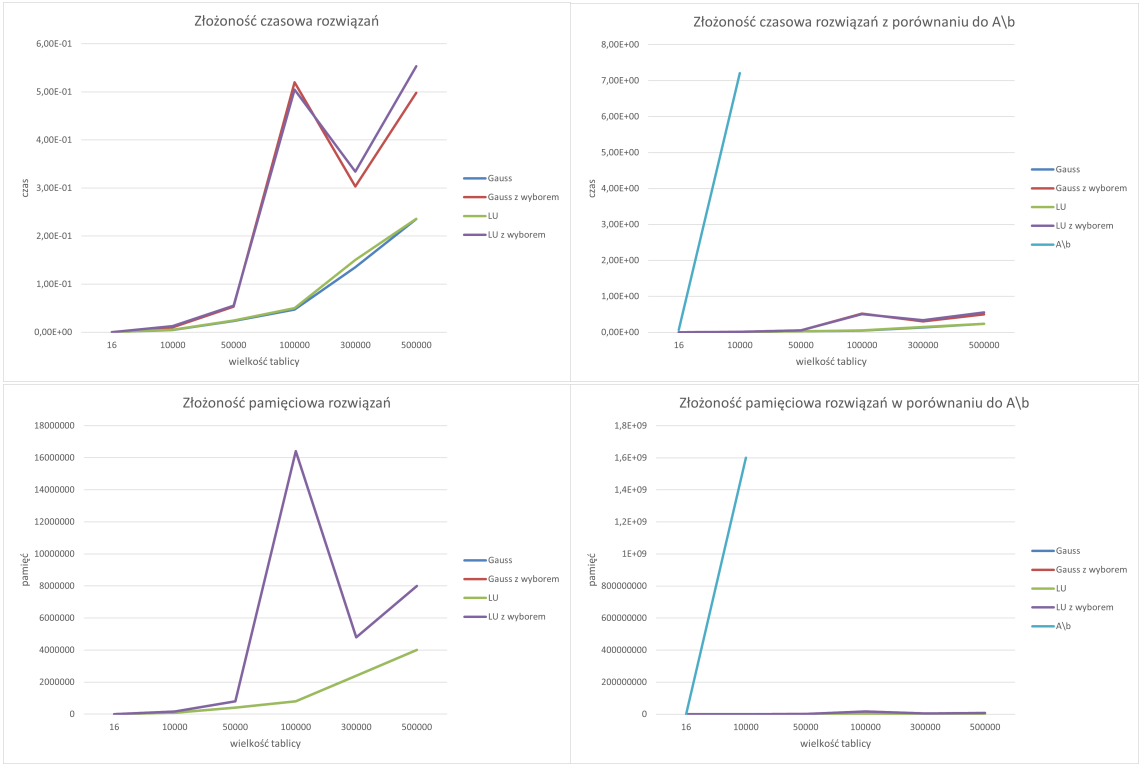
Złożoność czasowa algorytmów					
n	eliminacja Gaussa	eliminacja Gaussa z częściowym wyborem	LU	LU z częściowym wyborem	A\b
16	1.66333333333333332e-5	2.9100000000000003e-5	1.73e-5	3.1333333333333334e-5	0.06471443333333333
10 000	0.004700333333333333	0.010327266666666666	0.004936133333333334	0.012497933333333334	7.210426333333334
50 000	0.023264566666666667	0.053248333333333335	0.024451499999999998	0.0551517	OutOfMemory
100 000	0.04694913333333334	0.5199331333333334	0.0502147	0.5044943000000001	OutOfMemory
300 000	0.13540343333333332	0.30321776666666667	0.15040716666666668	0.3339341	OutOfMemory
500 000	0.23545676666666668	0.49791216666666666	0.23608583333333333	0.5534834666666666	OutOfMemory

Złożoność pamięciowa algorytmów					
n	eliminacja Gaussa	eliminacja Gaussa z częściowym wyborem	LU	LU z częściowym wyborem	A\b
16	192.0	384.0	192.0	384.0	4736.0
10 000	80048.0	160096.0	80048.0	160096.0	1.600160192e9
50 000	400048.0	800096.0	400048.0	800096.0	OutOfMemory
100 000	800048.0	1.6402912e7	800048.0	1.6402912e7	OutOfMemory
300 000	2.400048e6	4.800096e6	2.400048e6	4.800096e6	OutOfMemory
500 000	4.000048e6	8.000096e6	4.000048e6	8.000096e6	OutOfMemory

Błędy względne algorytmów				
n	eliminacja Gaussa	eliminacja Gaussa z częściowym wyborem	LU	LU z częściowym wyborem
16	1.466850003875454e-15	3.3766115072321297e-16	1.466850003875454e-15	3.3766115072321297e-16
10 000	2.6979457399889918e-14	4.952490752895815e-16	2.6979457399889918e-14	4.952490752895815e-16
50 000	5.983696068447594e-14	5.378168315102863e-16	5.983696068447594e-14	5.378168315102863e-16
100 000	7.422229851374492e-13	5.199630892522782e-16	7.422229851374492e-13	5.199630892522782e-16
300 000	6.521306370065581e-14	4.554950254748486e-16	6.521306370065581e-14	4.554950254748486e-16
500 000	8.329169633245345e-14	4.507270850385053e-16	8.329169633245345e-14	4.507270850385053e-16

Obliczenia zostały wykonane przy użyciu makra @timed

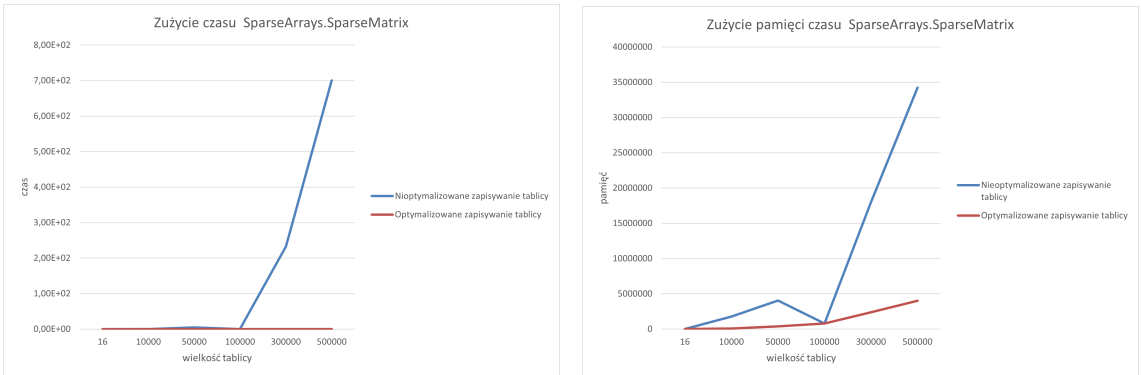
Widzimy że naiwne rozwiązywanie układu A\b bardzo szybko sprowadzi do zużycia całej pamięci, a ponad to już na małych liczbach liczenie zajmuje dużo czasu. Najszybciej działają wersję algorytmów bez wyboru elementu głównego, przy tym eliminacja Gaussa jest prawie dwa razy szybsza od rozkładu LU. Liczenie algorytmów z częściowym wyborem także potrzebuje więcej pamięci, ale jeżeli popatrzymy na błędy względne zobaczymy że faktycznie wybieranie elementu głównego zmniejsza błędy przy obliczaniu z bardzo małymi liczbami na przekątnej.



Wykresy wykonane w Microsoft Excel

3.6 Optymalizowane tworzenie SparseArrays

Sparse Arrays efektywnie przechowuje macierze, pobierając wiersze, kolumny i wartości elementów niezerowe. Gdy użytkownik potrzebuję skorzystać z elementu niezerowego wewnątrz SparseMatrix, struktura dosyć szybko zwraca dane, ale problem występuję wtedy gdy jest potrzeba iterowania po elementach pustych. Strasznie długo zwraca dane, powoduje większe zużycie pamięci i nawet lekko zaburza wyniki, powodując większe błędy względne.
Na pierwszy pogląd paradoksalnym rozwiązaniem danego problemu jest wpisanie ręcznie elementów zerowych do wektorów tworzących SparseMatrix.
Nie sprawia to większych trudności dla tego że wiemy że zerowymi elementami występującymi w naszym programie będą wartości między podmacierzą A_x, a przekątna w podmacierzy C_x, a także l elementów od przekątnej C_k wpawo, dla tego że przy wyborze elementu głównego zawsze robimy odejmowanie na $k + 2l$ elementach, przez możliwe wybranie dolnego wiersza, które od danego indeksu k do przekątnej w podmacierzy C_k może mieć $2l$ elementów.
W tym celu przy wczytywaniu macierzy z pliku dopisano kilka dodatkowych pętel które do całej macierzy generują potrzebne elementy zerowe. Efekty danej optymalizacji pokaże na przykładzie optymizowanego algorytmu eliminacji Gaussa bez wyboru elementu głównego:



Wykresy wykonane w Microsoft Excel

3.7 Wnioski

Odpowiednia analiza problemu może spowodować, że stosunkowo łatwo otrzymamy rozwiązanie, które w naiwnym podejściu byłoby kosztownym lub niemożliwym do uzyskania.
Drobne modyfikacje algorytmów umożliwiły nam znaczne ograniczenie ich złożoności obliczeniowej. Znając strukturę danych udało nam się również ograniczyć zapotrzebowanie na pamięć, odpowiednio wybierając ich sposób przechowywania.