ONLl5

Feofilaktow Iwan

January 2021

1 Zadanie

Zaimplementować efektywne algorytmy i struktury danych dla przechowywania i rozwiazania układu równań liniowych w nastepnej postaci $Ax = b$ $A \in R^{n \times n}$ gdzie

$$A = \begin{bmatrix} A_1 & C_1 & 0 & 0 & 0 & ... & 0 \\ B_1 & A_2 & C_2 & 0 & 0 & ... & 0 \\ 0 & B_2 & A_3 & C_3 & 0 & ... & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & ... & 0 & B_{v-3} & A_{v-2} & C_{v-2} & 0 \\ 0 & ... & 0 & 0 & B_{v-2} & A_{v-1} & C_{v-1} \\ 0 & ... & 0 & 0 & 0 & B_{v-1} & A_v \end{bmatrix}$$

$$B_i = \begin{bmatrix} 0 & ... & 0 & b_1^i \\ 0 & ... & 0 & b_2^i \\ \vdots & \ddots & \vdots & \vdots \\ 0 & ... & 0 & b_l^i \end{bmatrix}$$

$$C_i = \begin{bmatrix} c_1^i & 0 & 0 & ... & 0 \\ 0 & c_2^i & 0 & ... & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & ... & 0 & c_{l-1}^i & \\ 0 & ... & 0 & 0 & c_l^i \end{bmatrix}$$

$n = v * l$

2 Algorytmy dla macierzy gestych

2.1 Eliminacja Gaussa

Eliminacja Gaussa jest algorytmem dla rozwiazywania nieosobliwego układu równań liniowych.

$a_{11}x_1 + a_{12}x_2 + ... + a_{1n-1}x_{n-1} + a_{1n}x_n = b_1$
...
$a_{n1}x_1 + a_{n2}x_2 + ... + a_{n1}x_{n-1} + a_{n2}x_n = b_n$

lub $Ax = b$ w postaci macierzowej. Polega algorytm na postepowej eliminacji niezerowych elementów macierzy znajdujacych sie pod przekatna i sprowadzaniu zadania do rozwiazania trywialnego układu $Ux = b^{'}$ gdzie U jest macierza górna trójkatna a $b^{'}$ odpowiednio zmodyfikowanym wektorem b. W pierwszym kroku trzeba eliminowac zmienna x_1 ze wszystkich równań oprócz pierwszego. $L_1 A^{(1)} = A^{(2)}$ gdzie $A^{(1)}$ i $A^{(2)}$ sa macieżami przed i pierwszym kroku eliminacji.

$$A^{(1)} = \begin{bmatrix} a^{(1)}_{1,1} & a^{(1)}_{1,2} & ... & a^{(1)}_{1,n-1} & a^{(1)}_{1,n} \\ a^{(1)}_{2,1} & a^{(1)}_{2,2} & & a^{(1)}_{2,n-1} & a^{(1)}_{2,n} \\ \vdots & & \ddots & & \vdots \\ a^{(1)}_{n-1,1} & a^{(1)}_{n-1,2} & & a^{(1)}_{n-1,n-1} & a^{(1)}_{n-1,n} \\ a^{(1)}_{n,1} & a^{(1)}_{n,2} & ... & a^{(1)}_{n,n-1} & a^{(1)}_{n,n} \end{bmatrix}$$

$$A^{(2)} = \begin{bmatrix} a^{(2)}_{1,1} & a^{(2)}_{1,2} & ... & a^{(2)}_{1,n-1} & a^{(2)}_{1,n} \\ 0 & a^{(2)}_{2,2} & & a^{(2)}_{2,n-1} & a^{(2)}_{2,n} \\ \vdots & & \ddots & & \vdots \\ 0 & a^{(2)}_{n-1,2} & & a^{(2)}_{n-1,n-1} & a^{(2)}_{n-1,n} \\ 0 & a^{(2)}_{n,2} & ... & a^{(2)}_{n,n-1} & a^{(2)}_{n,n} \end{bmatrix}$$

Macierz L_1 wybieramy w nastepny sposób

$$L_1 = \begin{bmatrix} 1 & 0 & ... & 0 & 0 \\ l^{(1)}_{2,1} & 1 & & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ l^{(1)}_{n-1,1} & 0 & & 1 & 0 \\ l^{(1)}_{n,1} & 0 & ... & 0 & 1 \end{bmatrix}$$

wtedy $a^{(2)}_{i,1} = a^{(1)}_{i,j} + l^{(1)}_{i,1} a^{(1)}_{1,1} = 0 \quad l^{(1)}_{i,1} = \frac{a^{(1)}_{i,j}}{a^{(1)}_{1,1}}$ Z czego wynika że algorytm bez przestawiania jest wykonywalny tylko wtedy, gdy $a^{(i)}_{i,i} \neq 0$ Analogiczne postepowania dla eliminacji pozostałych kolumn, co prowadzi do postaci
$U = L_n L_{n-1} ... L_2 L_1 A$
$L_n L_{n-1} ... L_2 L_1 b = Ux$ co prowadzi do nastepnego algorytmu.

Algorithm 1 ToUpperTriangular(A,b,n)

```
for k ← 1 to n-1 do
    if A[k,k] = 0 then
        return 1
    end if
    for i ← k+1 to n do
        A[i,k] := 0
        b[i] := b[i] − b[k] ∗ A[i,k]/A[k,k]
        for j ← k+1 to n do
            A[i,j] := A[i,j] − A[k,j] ∗ A[i,k]/A[k,k]
        end for
    end for
end for
return 0
```

Algorytm eliminacji w analogiczny sposób może tworzyć macierzy dolne trójkatne. Przy użyciu takiej modyfikacji z macierza U algorytm wygeneruje układ równań z macierza przekatniowa. $Dx = b^{''}$ Dla wyznaczenia x zostaje tylko użyć równości $x_i = \frac{b_i^{''}}{d_{i,i}}$. Co prowadzi do algorytmu poniżej.

Algorithm 2 SolveUpperTriangular(A,b,n)

```
for k ← n to 2  do
    if A[k,k] = 0 then
        return 1
    end if
    for i ← k-1 to 1 do
        b[i] := b[i] − b[k] ∗ A[i,k]/A[k,k]
    end for
    b[k] := b[k]/A[k,k]
end for
return 0
```

2.2 LU Rozkład

Z eliminacji do macierzy górnej trójkatnej $U = L_n L_{n-1} ... L_2 L_1 A$ Dla macierzy trójkatnych dolnych ich iloczyny i odwrotności także sa trójkatne dolne. Z tego wynika, że dla klasy macierzy, dla której możliwa jest eliminacja bez przestawiania, istnieja takie L, U, bedace macierza dolno- i górno-trójkatna, że $LU = A$ $(L = (L_n L_{n-1} ... L_2 L_1)^{-1})$ Z mnożenia macierzy $a_{i,j} = \sum_{k=1}^{n} l_{i,k} u_{k,j}$. Z trójkatności U i L $a_{i,j} = \sum_{k=1}^{min(i,j)} l_{i,k} u_{k,j}$ Powstaje układ z n^2 równań i $n^2 + n$ niewiadomymi. Zakładamy że $l_{i,i} = 1$. Wtedy dla $a_{i,i} = \sum_{k=1}^{i} l_{i,k} u_{k,i} = \sum_{k=1}^{i-1} l_{i,k} u_{k,i} + u_{i,i}$
$u_{i,i} = a_{i,i} - \sum_{k=1}^{i-i} l_{i,k} uk, i$
$u_{i,j} = a_{i,j} - \sum_{k=1}^{i-1} l_{i,k} u_{k,j}$ dla $i < j$
$l_{i,j} = (a_{i,j} - \sum_{k=1}^{j-1} l_{i,k} u_{k,j}) / u_{j,j}$ dla $i > j$

Z powyższych rekurencyj można zauważyć, że dla obliczenia $l_{i,j}$ ui, j wystarczy wyznaczyć wszystkie elementy $l_{i,k}$ i $u_{k,j}$ takie że $k < i, j$. To determinuje kolejność wyznaczania elementów. Niech $L^{(m)}$ i $U^{(m)}$ beda macierzami na m-tym kroku algorytmu wtedy

$$L^{(m)} = \begin{bmatrix} 1 & 0 & & & & & & \\ l_{2,1} & 1 & & & & & & \\ \vdots & \vdots & \ddots & 0 & & & & \\ l_{m-1,1} & l_{m-1,2} & & 1 & 0 & & & \\ l_{m,1} & l_{m,2} & & l_{m,m-1} & 1 & & & \\ \vdots & \vdots & & \vdots & \vdots & \ddots & 0 & \\ l_{n-1,2} & l_{n-1,2} & ... & l_{n-1,m-1} & l_{n-1,m}^{(m)} & ... & 1 & 0 \\ l_{n,1} & l_{n,2} & ... & l_{n,m-1} & l_{n,m}^{(m)} & ... & l_{n,n-1}^{(m)} & 1 \end{bmatrix}$$

$$U^{(m)} = \begin{bmatrix} u_{1,1} & u_{1,2} & ... & u_{1,m-1} & u_{1,m} & ... & u_{1,n-1} & u_{1,n} \\ 0 & u_{2,2} & ... & u_{2,m-1} & u_{2,m} & ... & u_{2,n-1} & u_{2,n} \\ & 0 & \ddots & \vdots & \vdots & & \vdots & \vdots \\ & & & u_{m-1,m-1} & u_{m-1,m} & & u_{m-1,n-1} & u_{m-1,n} \\ & & & 0 & u_{m,m-1}^{(m)} & & u_{m,n-1}^{(m)} & u_{m,n}^{(m)} \\ & & & & 0 & \ddots & \vdots & \vdots \\ & & & & & & u_{n-1,n-1}^{(m)} & u_{n-1,n}^{(m)} \\ & & & & & & 0 & u_{n,n}^{(m)} \end{bmatrix}$$

Algorithm 3 LUFactorizing(A,n)

```
for k in 1 to n do
    L[k,k] := 1
    U[k,k] := A[k,k]
    for i in k+1 to n do
        L[i,k] := A[i,k]
        U[k,i] := A[k,i]
    end for
end for
for k ← n to 2 do
    if A[k,k] = 0 then
        return 1
    end if
    for i in k+1 to n do
        L[i,k] := L[i,k]/U[k,k]
    end for
    for j in k+1 to n do
        for i in j+1 to n do
            L[i,j] = L[i,j] − L[i,k] * U[k,j]
        end for
        for i in k+1 to j do
            U[i,j] = U[i,j] − L[i,k] * U[k,j]
        end for
    end for
end for
return 0
```

O ile $a_{i,j}$ jest potrzebne tylko do obliczenia elementu $l_{i,j}^{(m)}$ lub $ul_{i,j}^{(m)}$, za warunkiem opuszczenia przekatnej L algorytm można przerobić na in sito bez użycia dodatkowej pamieci.

$$A^{(m)} = \begin{bmatrix} u_{1,1} & u_{1,2} & \dots & u_{1,m-1} & u_{1,m} & \dots & u_{1,n-1} & u_{1,n} \\ l_{2,1} & u_{2,2} & \dots & u_{2,m-1} & u_{2,m} & \dots & u_{2,n-1} & u_{2,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots & \vdots \\ l_{m-1,1} & l_{m-1,2} & & u_{m-1,m-1} & u_{m-1,m} & & u_{m-1,n-1} & u_{m-1,n} \\ l_{m,1} & l_{m,2} & & l_{m,m-1} & u_{m,m-1}^{(m)} & & u_{m,n-1}^{(m)} & u_{m,n}^{(m)} \\ \vdots & \vdots & & \vdots & \vdots & \ddots & \vdots & \vdots \\ l_{n-1,2} & l_{n-1,2} & \dots & l_{n-1,m-1} & l_{n-1,m}^{(m)} & \dots & u_{n-1,n-1}^{(m)} & u_{n-1,n}^{(m)} \\ l_{n,1} & l_{n,2} & \dots & l_{n,m-1} & l_{n,m}^{(m)} & \dots & l_{n,n-1}^{(m)} & u_{n,n}^{(m)} \end{bmatrix}$$

Algorithm 4 LUFactorizingInPlace(A,b,n)

```
for k ← n to 2 do
    if A[k,k] = 0 then
        return 1
    end if
    for i in k+1 to n do
        A[i,k] = A[i,k]/A[k,k]
    end for
    for j in k+1 to n do
        for i in k+1 to n do
            A[i,j] = A[i,j] − A[i,k] ∗ A[k,j]
        end for
    end for
end for
return 0
```

Po wyprowadzeniu LU rozkładu macierze trójkatne moga być użyte do rozwiazania układu równań w nastepny sposób $LUx = b$ zamieniamy Ux na z i otrzymujemy dwa układy trójkatne $Lz = b$ i $Ux = z$. Rozwiazanie obu układów po kolei daje poszukiwany wektor x. O ile rozwiazanie dwu układów trójkatnych jest preferowane rozwiazaniu zwyczajnego układu rozkład LU jest lepszy dla wyznaczania x w wielokrotnych układów z niezmiennym A.

2.3 Pivoting

Żeby zapewnić lepsza stabilność można skorzystać z wybory w kolumnie. Przed każdym krokiem eliminacji wybiera sie bezwzglednie najwiekszy element w kolumnie eliminacji k wiersz tego elementu wymienia sie z wierszem k. Polepszenie stabilności powiazane z ograniczeniem wzrostu maksymalnego elementu macierzy a razem z tym i błedów obliczeniowych. Również taka modyfikacja może być użyta do algorytmu rozkładu LU.

Algorithm 5 ToUpperTriangularPivoting(A,b,n)

```
for k ← 1 to n-1 do
    pivot := i such that k ≤ i ≤ n, |A[i,k]| = max_j|A[j,k]|
    swap(A[k,:],A[pivot,:])
    if A[k,k] = 0 then
        return 1
    end if
    for i ← k+1 to size(A)[1] do
        A[i,k] := 0
        b[i] := b[i] − b[k] ∗ A[i,k]/A[k,k]
        for j ← k+1 to size(A)[1] do
            A[i,j] := A[i,j] − A[k,j] ∗ A[i,k]/A[k,k]
        end for
    end for
end for
return 0
```

Algorithm 6 LUFactorizingPivoting(A,b,n)

```
for k ← 1 to n  do
    pivot := i such that k ≤ i ≤ n, |A[i,k]| = max_j|A[j,k]|
    swap(A[k,:],A[pivot,:])
    if A[k,k] = 0 then
        return 1
    end if
    if A[k,k] = 0 then
        return 1
    end if
    for i in k+1 to n do
        A[i,k] = A[i,k]/A[k,k]
    end for
    for j in k+1 to n do
        for i in k+1 to n do
            A[i,j] = A[i,j] − A[i,k] ∗ A[k,j]
        end for
    end for
end for
return 0
```

3 Specjalne struktury danych

Zauważmy, że dla rzadkiej macierzy z zadania zachodzi nastepne $a_{i,j} = 0$ dla $|i-j| > l$ czyli jest on w całości zawarte w pasie przekatnym. To oznacza że można użycz macierzy pasmowych i odpowiednio zmodyfikowanych algorytmów. Niech p i q sa szerokościami pod- i nad-przekatna macierzy A odpowiednio. Wtedy dla macierzy zachodzi $i - j >= p \rightarrow a_{i,j} = 0$ i $j - i >= q \rightarrow a_{i,j} = 0$.

Przykładowo p = 2,q = 3 X - dowolny element

$$A = \begin{bmatrix} X & X & X & & & & \\ X & X & X & X & & & \\ & X & X & X & X & & \\ & & X & X & X & X & \\ & & & X & X & X & X \\ & & & & X & X & X \\ & & & & & X & X \end{bmatrix}$$

Jeśli przesunać każdy niezerowy element $a_{i,j}$ w takiej macierzy do komórki o współczynnikach $(i, j - i + p)$ ograniczenia zmienia sie na nastepne $i - (j + i - p) >= p \to a_{i,j} = 0$ i $j + i - p - i >= q \to a_{i,j} = 0$

$-j >= 0 \to a_{i,j} = 0$ i $j >= q + p \to a_{i,j} = 0$

$$A' = \begin{bmatrix} & X & X & X & & & \\ X & X & X & X & & & \\ X & X & X & X & & & \\ X & X & X & X & & & \\ X & X & X & X & & & \\ X & X & X & & & & \\ X & X & & & & & \end{bmatrix} \to T = \begin{bmatrix} & X & X & X \\ X & X & X & X \\ X & X & X & X \\ X & X & X & X \\ X & X & X & X \\ X & X & X & \\ X & X & & \end{bmatrix}$$

Wykorzystanie takiego przesuniecia pozwana na przechowywanie wszystkich elementów w prostokatnej macierzy $T \in R^{n \times p+q-1}$. Z p = q = l+1 otrzymujemy złożoność pamieciowa O(ln). Niżej sa zmienione podstawowe operacji macierzowe.

Algorithm 7 get(T,i,j,p)

```
i := T.P[i]
return T[i,j-i+p]
```

Algorithm 8 set(T,i,j,p,v)

```
i: = T.P[i]
T[i,j-i+p] := v
```

Algorithm 9 swap(T,row1,row2)

```
T.P[row1] := row2
T.P[row2] := row1
```

4 Algorytmy zmodyfikowane

4.1 Eliminacja Gaussa

Głowna zmiana w porównaniu do startowego algorytmu jest wziecie pod uwage ograniczeń pasmowych. i zmniejszenie liczby iteracyj petel wewnetrznych. Krok eliminacji jest przeprowadzany tylko dla wierszy które moga mieć niezerowy element w kolumnie eliminacji. Również odejmowanie wierszy macierzy ograniczone go elementów niezerowych. Algorytm eliminacji nie dodaje niezerowych elementów pod pasem, ponieważ modyfikuje tylko elementy zlewa od kolumny eliminacji. Również dodani górnego wiersza od dolnego nie może spowodować dodanie elementów niezerowych na góre pasu. To oznacza że wynik działania algorytmu jest identyczny do niezmodyfikowanej wersji.

Algorithm 10 ToUpperTriangular(A,b,n,q,p)

```
for k ← 1 to n-1 do
    if A[k,k] = 0 then
        return 1
    end if
    for i ← k+1 to min(k+p-1,n) do
        A[i,k] := 0
        b[i] := b[i] − b[k] ∗ A[i,k]/A[k,k]
        for j ← k+1 to min(k+q-1,n) do
            A[i,j] := A[i,j] − A[k,j] ∗ A[i,k]/A[k,k]
        end for
    end for
end for
return 0
```

Algorytm zawiera 3 petli włożone o liczbach iteracyj O(n),O(p) i O(q) pozostałe instrukcje sa wykonywana w O(1), a wiec złożoność obliczeniowa jest O(npq). Rozwiazanie układu trójkatnego ma złożoność O(n), co oznacza, że całkowita złożoność obliczeniowa rozwiazania układu równań jest O(nqp) Macierz pasmowa dla danych z zadania ma p = q = l+1 i nie potrzebuje dodatkowego pasu na rozwiazanie. Dla stałej l złożoność jest O(n).

4.2 LU Rozkład

Zmniejszenie liczby iteracyj petel poprzez wykluczanie operacyj z elementami zerowymi. Zmiana wartości komórki (i,j) nie może wystapić jeżeli dla każdego k¿0 wszystkie elementy U[i-k,j] lub L[i,j-k] sa zerowe. Z ograniczenia pasmowego takimi komórkami sa wszystkie elementy poza macierza pasmowa. To oznacza że algorytm działa tak jak niezmodyfikowana wersja.

Algorithm 11 LUFactorizingInPlace(A,b,n,q,p)

```
for k ← n to 2 do
    if A[k,k] = 0 then
        return 1
    end if
    for i in k+1 to min(k+p-1,n) do
        A[i,k] = A[i,k]/A[k,k]
    end for
    for j in k+1 to min(k+q-1,n) do
        for i in k+1 to min(k+p-1,n) do
            A[i,j] = A[i,j] − A[i,k] ∗ A[k,j]
        end for
    end for
end for
return 0
```

Obliczenie złożoności obliczeniowej jak w algorytmie powyżej. $O(nl^2)$.O(n) dla stałej l.

4.3 Pivoting

Dodanie wybory elementu głównego może zepsuć pasmowa strukture macierzy, a wiec trzeba dodać dodatkowy pas dla przechowywania elementów które moga wyjść spoza macierz. W najgorszym przypadku stanie sie zamiana najniższego możliwego wiersza.

$$
\begin{bmatrix}
X & X & X & & & & \\
 & X & X & X & & & \\
 & X & X & X & X & & \\
 & & X & X & X & X & \\
 & & & X & X & X & X \\
 & & & & X & X & X \\
 & & & & & X & X
\end{bmatrix}
\rightarrow
\begin{bmatrix}
X & X & X & & & & \\
 & X & X & X & X & & \\
 & X & X & X & & & \\
 & & X & X & X & X & \\
 & & & X & X & X & X \\
 & & & & X & X & X \\
 & & & & & X & X
\end{bmatrix}
$$

W takim wypadku w wierszu pojawi sie p elementów niezerowych poza zakresem macierzy. Każdy wiersz w algorytmie jest zamieniany tylko 1 raz wiec wystarczy zwiekszyć szerokość pasu nad-przekatniowego o szerokość pasu pod-przekatniowego. $p = l + 1$, $q = 2l + 1$ Takie rozszerzenie nie spowoduje zmiany klasy złożoności pamieciowej i obliczeniowej.

Algorithm 12 ToUpperTriangular(A,b,n,q,p)

```
for k ← 1 to n-1 do
    pivot := i such that k ≤ i ≤ n, |A[i,k]| = max_j |A[j,k]|
    swap(A[k,:],A[pivot,:])
    if A[k,k] = 0 then
        return 1
    end if
    for i ← k+1 to min(k+p-1,n) do
        A[i,k] := 0
        b[i] := b[i] − b[k] ∗ A[i,k]/A[k,k]
        for j ← k+1 to min(k+q-1,n) do
            A[i,j] := A[i,j] − A[k,j] ∗ A[i,k]/A[k,k]
        end for
    end for
end for
return 0
```

Podobnie z algorytmem eliminacji, rozkład LU potrzebuje pas pomocnicza. Również w odróżnieniu od eliminacji elementy zlewa od elementu głównego moga być niezerowe. Razem z faktem, że wiersz może być zamieniony z wierszem niżej wiecej niż jeden raz, powoduje niemożliwość użycia tego algorytmu z macierza pasmowa w ogólnym przypadku. Na szczeście blokowa struktura macierzy z zadania pozwala na ominiecie tego problemu kosztem niewielkiej utraty stabilności numerycznej. Można zauważyć że dla wybór elementu głównego w bloku A_i nie powoduje problemów, ponieważ nie może wyjść z zakresu macierzy. A wiec jedyny problem może wyniknać przy wyborze elementu z B_i. Rozwiazaniem jest niewybieranie elementu głównego z B_i

$$\begin{bmatrix} X & X & X & & & & \\ X & X & X & X & & & \\ & X & X & X & X & & \\ & & X & X & X & X & \\ & & & X & X & X & X \\ & & & & X & X & X \\ & & & & & X & X \end{bmatrix} \rightarrow \begin{bmatrix} X & X & X & & & & \\ & X & X & X & X & & \\ X & X & X & X & & & \\ & & X & X & X & X & \\ & & & X & X & X & X \\ & & & & X & X & X \\ & & & & & X & X \end{bmatrix} \rightarrow ... \rightarrow \begin{bmatrix} X & X & X & & & & \\ X & X & X & X & & & \\ & & X & X & X & X & \\ & & & X & X & X & X \\ & & & & X & X & X \\ & & & & & X & X \\ & X & X & X & X & & \end{bmatrix}$$

Algorithm 13 LUFactorizingInPlace(A,b,n,q,p,l)

```
for k ← 1 to n-1 do
    if k then mod l ≠ 0
        pivot := i such that k ≤ i ≤ n, |A[i,k]| = max_j|A[j,k]|
        swap(A[k,:],A[pivot,:])
        if A[k,k] = 0 then
            return 1
        end if
        for i in k+1 to min(k+p-1,n) do
            A[i,k] = A[i,k]/A[k,k]
        end for
        for j in k+1 to min(k+q-1,n) do
            for i in k+1 to min(k+p-1,n) do
                A[i,j] = A[i,j] − A[i,k] ∗ A[k,j]
            end for
        end for

        return 0
```

Złożoność nie zmienia sie.

5 Testowanie i wyniki

Testy potwierdzaja liniowa klase złożoności pamieciowej i obliczeniowej. Algorytmy osiagaja pożadana precyzje z błedem nie wyżej 10^{-13} dla wariantów bez wyboru elementu głównego i 10^{-16} z wyborem. Naturalnie algorytmy LU potrzebuja wiecej operacji przez niezbedność obliczenia 2 układów trójkatnych ale jest lepszy przy rozwiazaniu wiekszej liczby układów. Algorytmy z wyborem potrzebuja wiecej operacyj i pamieci poprzez dodanie pasu pomocniczego do macierzy pasmowej.

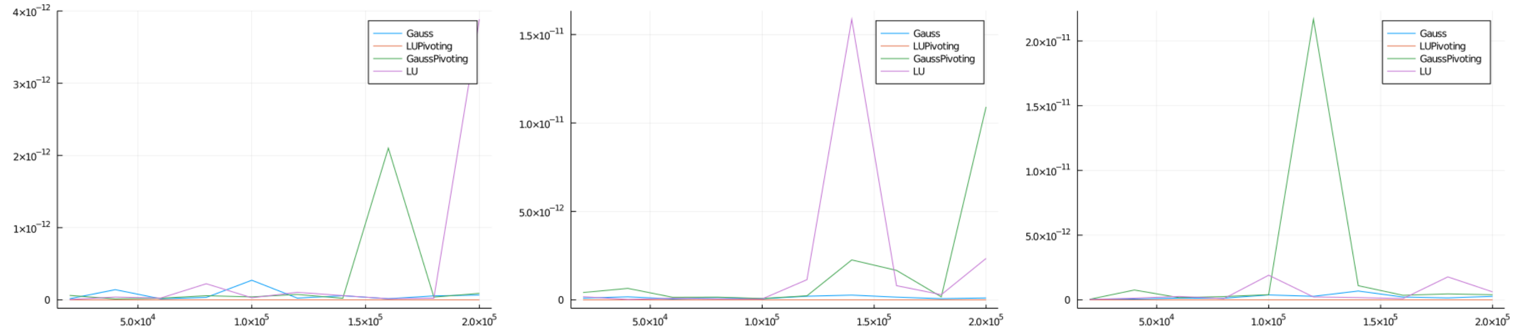

Figure 1: Blad relatywny dla l =2,5,8

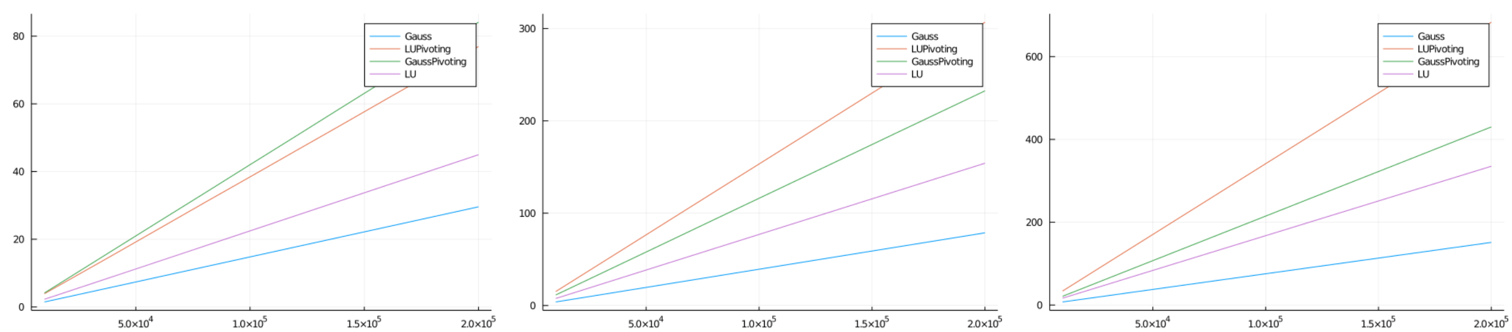

Figure 2: Zużycie pamieci dla l =2,5,8

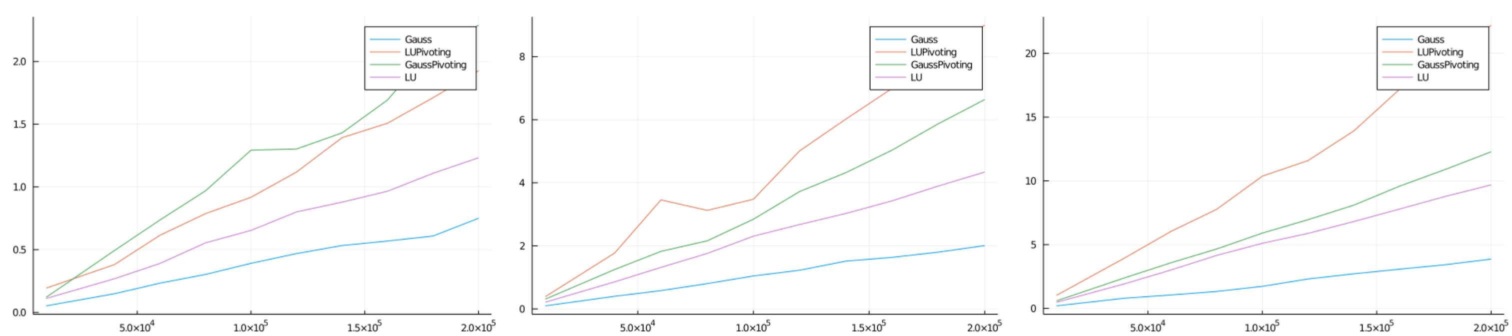

Figure 3: Czas działania dla l =2,5,8