Obliczenia Naukowe,
Laboratorium, Lista 5

Jerzy Wroczyński

2021-01-10

1 Opis problemu

Należy efektywnie obliczyć wynik równania

$$Ax = b,$$

które oczywiście reprezentuje układ równań, gdzie macierz A przechowuje współczynniki stojące przy składowych x, sam wektor reprezentuje konkretne rozwiązanie układu równań, a wektor b przechowuje prawe strony równań.

1.1 Struktura macierzy A

Przy czym macierz A jest macierzą rzadką o specyficznej strukturze:

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

gdzie

- $v = \frac{n}{l}$ (zakładamy, że n jest podzielne przez l) jest iteratorem bloków (podmacierzy),
- l jest rozmiarem każdego z bloków (podmacierzy),
- n jest rozmiarem całej macierzy A.

1.2 Struktura podmacierzy

Podmacierze ($k = 1 \dots v$) z których składa się macierz A są następującej struktury:

- A_k są macierzami gęstymi (nie mają elementów zerowych),
- $B_k = \begin{bmatrix} 0 & \cdots & 0 & b_1^k \\ 0 & \cdots & 0 & b_2^l \\ \vdots & & \vdots & \vdots \\ 0 & \cdots & 0 & b_l^k \end{bmatrix}$,

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

2 Idea rozwiązania

Do proceduralnego rozwiązywania układów równań często wykorzystywany jest algorytm *elminacji Gaussa*. Ideą tego algorytmu jest sprowadzenie macierz A do macierzy trójkątnej, ponieważ wtedy można sukcesywnie rozwiązać układ równań, idąc od dołu do góry.
Zobaczmy, jak to wygląda na początku. Mamy:

- $A \in \mathbb{R}^{n \times n}$,
- $x \in \mathbb{R}^n$,
- $b \in \mathbb{R}^n$,
- $\det(A) \neq 0$.

Znakiem $^{(k)}$ oznaczamy k-ty krok algorytmu. Wówczas oczywiście $A^{(1)} = A,\ \ b^{(1)} = b$.

$$A^{(1)}x = b^{(1)}: \quad \begin{array}{ccccccccc} a_{11}^{(1)}x_1 & + & a_{12}^{(1)}x_2 & + & \cdots & + & a_{1n}^{(1)}x_n & = & b_1^{(1)} \\ a_{21}^{(1)}x_1 & + & a_{22}^{(1)}x_2 & + & \cdots & + & a_{2n}^{(1)}x_n & = & b_2^{(1)} \\ \vdots & & \vdots & & & & \vdots & & \vdots \\ a_{n1}^{(1)}x_1 & + & a_{n2}^{(1)}x_2 & + & \cdots & + & a_{nn}^{(1)}x_n & = & b_n^{(1)} \end{array}$$

Chcemy uzyskać macierz trójkątną, czyli wyeliminować wszystkie współczynniki umieszczone pod diagonalą macierzy. Wykonujemy $(n-1)$ kroków:

$$A^{(k)}x = b^{(k)} \quad \begin{array}{ccccccccc} a_{11}^{(1)}x_1 & + & a_{12}^{(1)}x_2 & + & \ldots & + & a_{1n}^{(1)}x_n & = & b_1^{(1)} \\ & & a_{22}^{(2)}x_2 & + & \ldots & + & a_{2n}^{(2)}x_n & = & b_2^{(2)} \\ & & & \ddots & & & \vdots & & \vdots \\ & & & & a_{kk}^{(k)}x_k & +\cdots+ & a_{kn}^{(k)}x_n & = & b_k^{(k)} \\ & & & & \vdots & & \vdots & & \vdots \\ & & & & a_{nk}^{(k)}x_k & +\cdots+ & a_{nn}^{(k)}x_n & = & b_n^{(k)} \end{array}$$

i uzyskujemy macierz trójkątną:

$$A^{(n)}x = b^{(n)} \quad \begin{array}{ccccccccc} a_{11}^{(1)}x_1 & + & a_{12}^{(1)}x_2 & + & \ldots & + & a_{1n}^{(1)}x_n & = & b_1^{(1)} \\ & & a_{22}^{(2)}x_2 & + & \ldots & + & a_{2n}^{(2)}x_n & = & b_2^{(2)} \\ & & & & \ddots & & \vdots & & \vdots \\ & & & & & & a_{nn}^{(n)}x_n & = & b_n^{(n)} \end{array}$$

Tak jak mówiliśmy wcześniej — na dole mamy trywialne równanie $a_{nn}^{(n)} \cdot x_n = b_n^{(n)}$, z którego wyznaczamy x_n. Następnie z równania powyżej wyznaczamy x_{n-1}, bo znamy już x_n itd. sukcesywnie wyznaczamy cały wektor x.

Możemy zamknąć ideę tego algorytmu w formie listy kroków:

- `for` $k := 1 \dots (n-1)$:
  - `for` $i := (k+1) \dots n$:
    - $l_{ik} \leftarrow \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}$
    - `for` $j := (k+1) \dots n$:
      - $a_{ij}^{(k+1)} \leftarrow a_{ij}^{(k)} - l_{ik} a_{kj}^{(k)}$
    - $b_i^{(k+1)} \leftarrow b_i^{(k)} - l_{ik} b_k^{(k)}$

2.1 Ograniczenie iteracji

Jednakże biorąc pod uwagę ten specyficzny układ elementów w macierzy wejściowej, standardowy algorytm *eliminacji Gaussa* będzie bardzo nieoptymalny, ponieważ bardzo dużo obliczeń zostanie wykonanych niepotrzebnie na samych zerach.

Przypadek takiej macierzy możemy porównać do macierzy diagonalnej, gdzie złożoność obliczeniowa jest liniowa, ponieważ w każdym równaniu mamy tylko jeden niezerowy współczynnik. Tutaj mamy *w pewnym sensie* podobną sytuację, przy czym skalujemy rozmiar przekątnej przez liczbę l oraz wprowadzamy pewnie nieregularności rozmieszczenia elementów w otoczeniu przekątnej macierzy (patrz: struktura podmacierzy).
Oczywiście chcemy również osiągnąć liniową złożoność obliczeniową.
W celu ograniczenia liczby operacji chcemy ograniczyć, dokąd dochodzimy w pętlach w stosowanym algorytmie.
Dla zobrazowania problemu popatrzmy jeszcze raz na standardowy algorytm eliminacji Gaussa:

```
1: for k := 1 ... (n − 1) do
2:     for i := (k + 1) ... n do
3:         l_ik := a_ik^(k) / a_kk^(k)
4:         for j := (k + 1) ... n do
5:             a_ij^(k) := a_ij^(k) − l_ik a_kj^(k)
6:         end for
7:         b_i^(k+1) := b_i^(k) − l_ik b_k^(k)
8:     end for
9: end for
```

(notacja $^{(k)}$ oznacza k-ty krok algorytmu)
Widzimy, że mamy zagnieżdżone w sobie trzy pętle `for`. Prowadzi to oczywiście do złożoności obliczeń $O(n^3)$. Chcemy tego uniknąć czyli ograniczyć liczebność iteracji przechodzenia przez elementy macierzy, która jest przecież macierzą bardzo rzadką. Jeżeli założymy, że liczba l określająca wielkość bloków (podmacierzy), z których składa się macierz A, jest stała, to jesteśmy w stanie osiągnąć złożoność obliczeniową $O(n)$ przy rozwiązywaniu układu $Ax = b$. Dzieje się tak, ponieważ w takim układzie te dwie pętle wewnętrzne (linijki 2 oraz 4) wykonają znacznie mniej iteracji — nie wychodzą dalej niż bloki (podmacierze), ich złożoność obliczeniowa będzie zależeć od stałej l.

3 Potrzebne algorytmy

W celu realizacji zadanego problemu należy zaimplementować następujące algorytmy:

- Częściowy wybór elementu głównego (wyliczenie wektora permutacji wierszy macierzy A),
- Rozkład LU macierzy A (optymalizacja algorytmu eliminacji Gaussa),

- Obliczenie rozwiązania układu $Ax = b$
- Obliczenia rozwiązania układu $LUx = b$ (kiedy rozkład LU jest już obliczony)

Oczywiście, tak jak mówiliśmy o tym wcześniej — dążymy do liniowej złożoności obliczeniowej. Dlatego też we wszystkich algorytmach stosujemy pewien *design pattern*. Otóż najwyższą pętlę (iterującą po wszystkich wierszach) dzielimy na dwie pętle: jedną idącą po blokach $k = 1 \dots \frac{n}{l}$ i drugą, wewnętrzną idącą po poszczególnych elementach danego bloku (podmacierzy) plus opcjonalnie po elementach sąsiedniego bloku (podmacierzy) w razie potrzeby (iterator: $p = (k-1) \cdot l + 1 \dots k \cdot l[+l]$). Ostatecznie otrzymujemy liniową złożoność obliczeniową, ponieważ przyjmujemy, że l jest stałą.

3.1 Częściowy wybór elementu głównego

W wykorzystywanym algorytmie Gaussa wykorzystywana jest wielokrotnie operacja dzielenia przez elementy z przekątnej macierzy A. Może to być dość problematyczne, jako, że wartości na przekątnej mogą być bardzo małe, co oznacza możliwe bardzo duże odchylenia. Celem tego algorytmu jest zmaksymalizowanie wartości liczb na przekątnej.

Żeby tego dokonać, stosujemy *częściowy wybór elementu głównego*, czyli patrząc na daną wartość na przekątnej macierzy sprawdzamy czy jest to największa wartość w tej kolumnie. Jeśli tak nie jest — permutujemy wiersze w taki sposób, żeby na diagonali znalazła się ta wybrana największa liczba. Wówczas na przekątnej będziemy mieli względnie duże liczby, które nie powinny już sprawiać problemów.

Algorithm 1 partialPivot

```
1:  pivot = [1 ... n]
2:  for k := 1 ... n/l do
3:      for p := (k-1)·l + 1 ... k·l do
4:          curr = |A[pivot[p], p]|
5:          maxvalue = curr
6:          maxindex = pivot[p]
7:          for i := (p+1) ... k·l + l do
8:              if |A[pivot[i], p]| > maxvalue then
9:                  maxvalue = |A[pivot[i], p]|
10:                 maxindex = pivot[i]
11:             end if
12:         end for
13:         if maxvalue < macheps then
14:             return wybrana wartość jest bliska zeru
15:         end if
16:         if maxvalue > curr then
17:             pivot[p], pivot[maxindex] = pivot[maxindex], pivot[p]
18:         end if
19:     end for
20: end for
21: return pivot
```

Powyższy pseudokod ukazuje ideę algorytmu. Dwie pierwsze pętle (linijki 2 oraz 3) odpowiadają za przejście po wszystkich elementach na przekątnej macierzy (z podziałem na $\frac{n}{l}$ bloków). Dla każdego przejścia pobieramy wartość aktualnie będącą na przekątnej macierzy i patrzymy (7) czy nie ma większej wartości w tej kolumnie. Oczywiście ograniczamy zasięg poszukiwania w stosunku do wielkości bloków (liczba l).

W faktycznej implementacji tego algorytmu został wprowadzony argument `limited` będący flagą, który ma wpływ na zasięg poszukiwań określony domyślnie na $k \cdot l + l$ jak widać w linijce 7 — jeśli algorytm zwróci błąd, o którym mowa w linijce 14 zalecane jest albo zrezygnowanie z częściowego wyboru dla tego przypadku lub uruchomienie algorytmu z włączoną flagą `limited`. Wówczas, zakres poszukiwań zostanie ograniczony do jednego bloku, czyli pętla kończy się na $k \cdot l$.

3.2 Rozkład LU macierzy A

Standardowy algorytm rozkładu LU, jak sama nazwa wskazuje, rozkłada zadaną macierz A na dwie macierze trójkątne. Algorytm ten wywodzi się z algorytmu rozkładu Gaussa, gdzie dodatkowo, oprócz wyliczenia macierzy U, zapisujemy w macierzy L współczynniki, przez które mnożyliśmy równania, żeby uzyskać macierz U.

Żeby dostać poniższy układ równań, należy wykonać $k-1$ kroków algorytmu eliminacji Gaussa:

$$A^{(k)}x = b^{(k)} \quad \begin{array}{rcccccccc} a_{11}^{(1)}x_1 & + & a_{12}^{(1)}x_2 & + & \dots & + & a_{1n}^{(1)}x_n & = & b_1^{(1)} \\ & & a_{22}^{(2)}x_2 & + & \dots & + & a_{2n}^{(2)}x_n & = & b_2^{(2)} \\ & & & \ddots & & & \vdots & & \vdots \\ & & & & a_{kk}^{(k)}x_k & +\dots+ & a_{kn}^{(k)}x_n & = & b_k^{(k)} \\ & & & & \vdots & & \vdots & & \vdots \\ & & & & a_{nk}^{(k)}x_k & +\dots+ & a_{nn}^{(k)}x_n & = & b_n^{(k)} \end{array}$$

gdzie za każdym razem mnożymy k-te równanie przez

$$l_{ik} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}, \quad i = (k+1), \dots, n$$

i odejmujemy od pozostałych eliminując zmienną x_k z niższych równań. I to właśnie te liczby l_{ik} tworzą nam macierz L, bo to właśnie macierz L definiuje te wszystkie kroki (k), które należy wykonać, żeby uzyskać macierz U.

Poniższy algorytm jest zoptymalizowaną wersją algorytmu standardowego rozkładu LU. Tutaj również ograniczamy iterację pętli wewnętrznych.

Algorithm 2 decomposeIntoLU

```
1:  LU := A
2:  b' := b
3:  for k := 1 ... n/l do
4:      for p := (k − 1) · l + 1 ... k · l do
5:          for i := (p + 1) ... k · l + l do
6:              l_ip = LU[pivot[i],p] / LU[pivot[p],p]
7:              for j := p ... k · l + l do
8:                  LU[pivot[i], j] = LU[pivot[i], j] − l_ip · LU[pivot[p], j]
9:              end for
10:             b'[pivot[i]] = b'[pivot[i]] − l_ip · b'[pivot[p]]
11:             LU[pivot[i], p] = l_ip
12:         end for
13:     end for
14: end for
15: return LU, b'
```

Znowu iterujemy po wszystkich współczynnikach zmiennych x_p na diagonali macierzy (linijki 3 oraz 4), ponieważ chcemy usunąć zmienne w niższych równaniach. Tak też robimy właśnie — w linijce 5 zaczynamy iterować po wszystkich

wierszach, które są niżej niż aktualny (dlatego zaczynamy iterować od $(p+1)$). Obliczamy współczynnik dla każdego wiersza (reprezentującego równanie), przez który trzeba przemnożyć p-ty wiersz (reprezentujący równanie), a następnie go odejmujemy (linijka 8). Od razu też aktualizujemy zmiany w wektorze b (linijka 10) oraz zapisujemy wykonany krok do macierzy L (linijka 11).

Warto nadmienić, że sposób przechowywania macierzy L i U jest tutaj nieco inny niż w modelu matematycznym. Jako że obie macierze są macierzami trójkątnymi (jedna górno- druga dolno-trójkątna), a macierz L ma przekątną złożoną z samych jedynek (w żadnym kroku algorytmu Gaussa nie usuwamy zmiennych z diagonali), można te macierze przechowywać razem.

3.3 Obliczenie rozwiązania układu $Ax = b$

Użyjemy tutaj wcześniej zdefiniowanej funkcji *decomposeIntoLU*, która zwraca rozkład LU macierzy A wraz ze zmienionym wektorem b'.

Mając rozkład LU mamy oczywiście dostęp do górno-trójkątnej macierzy wyjściowej U. Wówczas wyznaczamy kolejne elementy wektora wyjściowego x stopniowo: wyraz x_n mamy *za darmo*, jako że ostatnie równanie jest postaci $u_n n \cdot x_n = b'_n$. Dalej, równanie wyżej będzie zależne od kolejnej nieznanej zmiennej oraz teraz już znanej zmiennej x_n. W równaniach wyżej mamy podobną sytuację — sukcesywnie wyznaczamy kolejne zmienne x_p.

3.3.1 Funkcja pomocnicza do rozwiązywania układów z macierzą górno-trójkątną

Niniejszy algorytm wylicza wyjściowy wektor x z podanych macierzy górno-trójkątnej oraz pionowego wektora y z prawej strony układu $Ux = y$. Tutaj mówimy bardziej ogólnie o wektorze prawej strony jako o wektorze y, ponieważ jest to funkcja pomocnicza, którą wykorzystamy nie tylko do rozwiązania układu $Ax = b$, ale też do układu $LUx = b$.
Dajemy na wejście:

- macierz górno-trójkątną U
- wektor y (prawa strona układu $Ux = y$)

Dostajemy na wyjście: wektor x będący rozwiązaniem układu $Ux = y$.

Algorithm 3 solveUpperTriangularMatrix(U, y, n, l, `pivot`)

1: $x := [\overbrace{0 \dots 0}^{n \text{ zer}}]$
2: **for** $k := \frac{n}{l} \dots 1$ **do**
3: **for** $p := k \cdot l \dots (k-1) \cdot l + 1$ **do**
4: $t := y\,[\texttt{pivot}[p]]$
5: **for** $i := (p+1) \dots k \cdot l + l$ **do**
6: $t := t - x\,[\texttt{pivot}[i] \cdot U[\texttt{pivot}[p], i]]$
7: **end for**
8: $x\,[\texttt{pivot}[p]] := \frac{t}{U[\texttt{pivot}[p], p]}$
9: **end for**
10: **end for**
11: **return** x

Dla każdej zmiennej x_p (lin. 2, 3) zaczynamy od wartości w wektorze po prawej stronie (lin. 4), a następnie odejmujemy już wyliczone zmienne x_i (lin. 6). Następnie dzielimy wynik przez współczynnik przy obliczanej zmiennej i zapisujemy do wyniku (lin. 8).

3.3.2 Implementacja algorytmu do rozwiązywania układu $Ax = b$

Mając macierz górno-trójkątną U oraz zmodyfikowany wektor b' możemy użyć funkcji *solveUpperTriangularMatrix* do obliczenia rozwiązania układu $Ax = b$.

Algorithm 4 gaussianElimination(A, b, n, l, `pivot`)

1: $U, b' =$ `decomposeIntoLU`$(A, b, n, l,$ `pivot`$)$
2: $x =$ `solveUpperTriangularMatrix`$(U, b', n, l, pivot)$
3: **return** x

3.4 Obliczenie rozwiązania układu $LUx = b$

Kolejnym bliźniaczym algorytmem do wcześniej definiowanych jest algorytm rozwiązujący układ $Ax = b$, przy czym już znany jest rozkład LU macierzy A. Czyli różnica jest taka, że nie mamy zmienionego wektora $b' = b^{(k)}$ w posiadaniu — mamy podany tylko rozkład LU macierzy A. Wówczas zadanie sprowadza się do obliczenia układu równań:

$$\begin{cases} Ly = b \\ Ux = y \end{cases}$$

Tutaj musimy rozwiązać dwa równania z macierzami trójkątnymi. Pierwsze równanie z macierzą dolno-trójkątną rozwiązujemy podobnie, jak rozwiązywaliśmy równanie z macierzą górno-trójkątną — sukcesywnie odkrywamy kolejne zmienne z wektora y.

Algorithm 5 solveFromLU(LU, b, n, l, `pivot`)

1: $y := [\overbrace{0 \dots 0}^{n}]$
2: **for** $k := 1 \dots \frac{n}{l}$ **do**
3: **for** $p := (k-1) \cdot l + 1 \dots k \cdot l$ **do**
4: $t := b\,[$`pivot`$[p]]$
5: **for** $i := (p-1) \dots (k-1) \cdot l - l$ **do**
6: $t := t - y[$`pivot`$[i]] \cdot$ `L` $[$`pivot`$[p], i]$
7: **end for**
8: $y[$`pivot`$[p]] = t$
9: **end for**
10: **end for**
11: $x =$ `solveUpperTriangularMatrix`$($`LU`$, y, n, l,$ `pivot`$)$
12: **return** x

Najpierw liczymy równanie $Ly = b$.
Iterujemy znowu po wszystkich elementach diagonali macierzy (lin. 2, 3). Zaczynamy od wartości z wektora po prawej stronie (lin. 4), a następnie odejmujemy od niej wszystkie znane już zmienne pomnożone przez odpowiednie współczynniki (lin. 6).
Następnie liczymy równanie $Ux = y$ przy pomocy funkcji `solveUpperTriangularMatrix`.

3.5 Złożoność obliczeniowa

Zakładając, że liczba l jest stałą, możemy powiedzieć, że złożoność obliczeniowa wszystkich powyższych algorytmów jest liniowa i zależy od n, czyli mamy $O(n)$. Za każdym razem mamy do czynienia z pętlą iterującą po blokach (podmacierzach)

dużej macierzy $k = 1 \dots \frac{n}{l}$ i pętlą iterującą po elementach tych bloków (podmacierzy). Jako że bloki (podmacierze) te ułożone są na diagonali tej macierzy mamy właśnie do czynienia z liniową złożonością obliczeniową.

Warto nadmienić, że przyjmujemy stałą $O(1)$ złożoność obliczeniową dla operacji pobierania i zapisywania elementów w strukturze wykorzystywanej do przechowywania naszej rzadkiej macierzy.

3.6 Złożoność pamięciowa

W implementacji algorytmów została zastosowana struktura `SparseMatrixCSC`, która w efektywny sposób przechowuje macierze rzadkie, traktując wartości 0 jako brak informacji. Znacznie mniejsze zużycie pamięci będzie bardzo widoczne w testach.

3.6.1 Alternatywa do `SparseMatrixCSC`

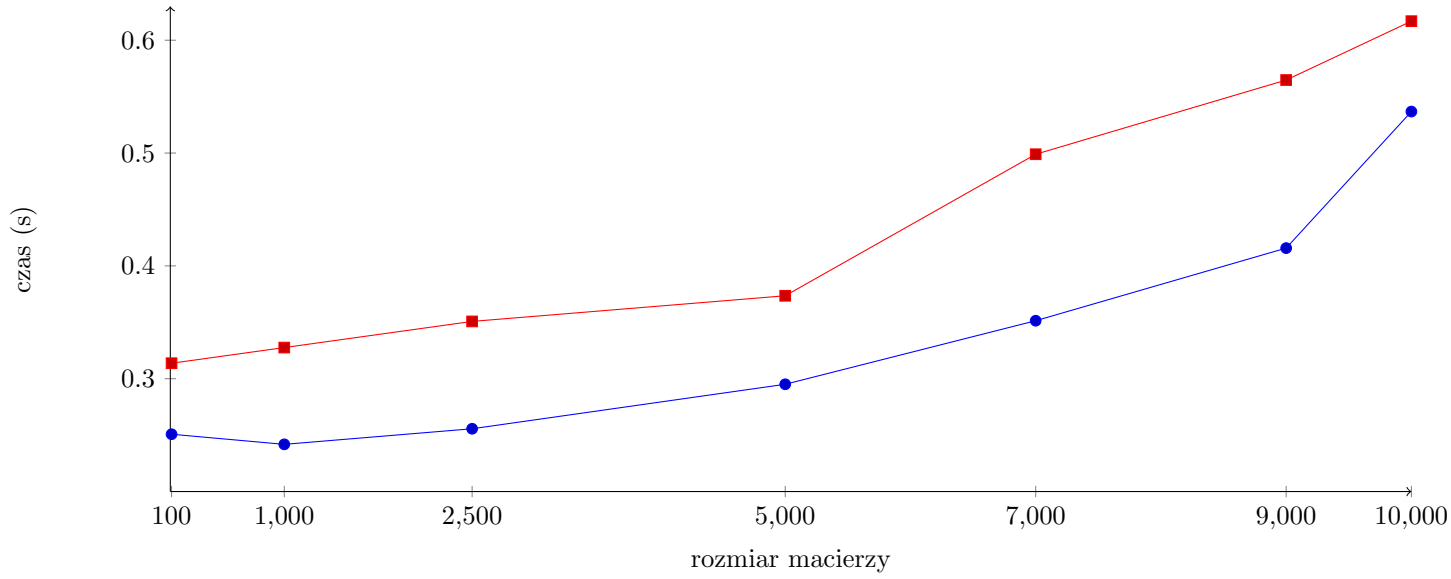
W pliku `types.jl` znajduje się struktura `SquareSparseMatrix` symulująca macierze rzadkie, oparta na strukturze `Dict` będącą implementacji mapy hashującej w języku *Julia*. Jednakże w testach okazało się, że niniejsza struktura zużywa więcej pamięci niż `SparseMatrixCSC`, a dla dostatecznie dużych rozmiarów macierzy (np. $n = 5000$) może nawet przekroczyć zużycie pamięci standardowego algorytmu rozwiązywania układu $Ax = b$ czyli `x = A\b` w *Julii*.

4 Testy

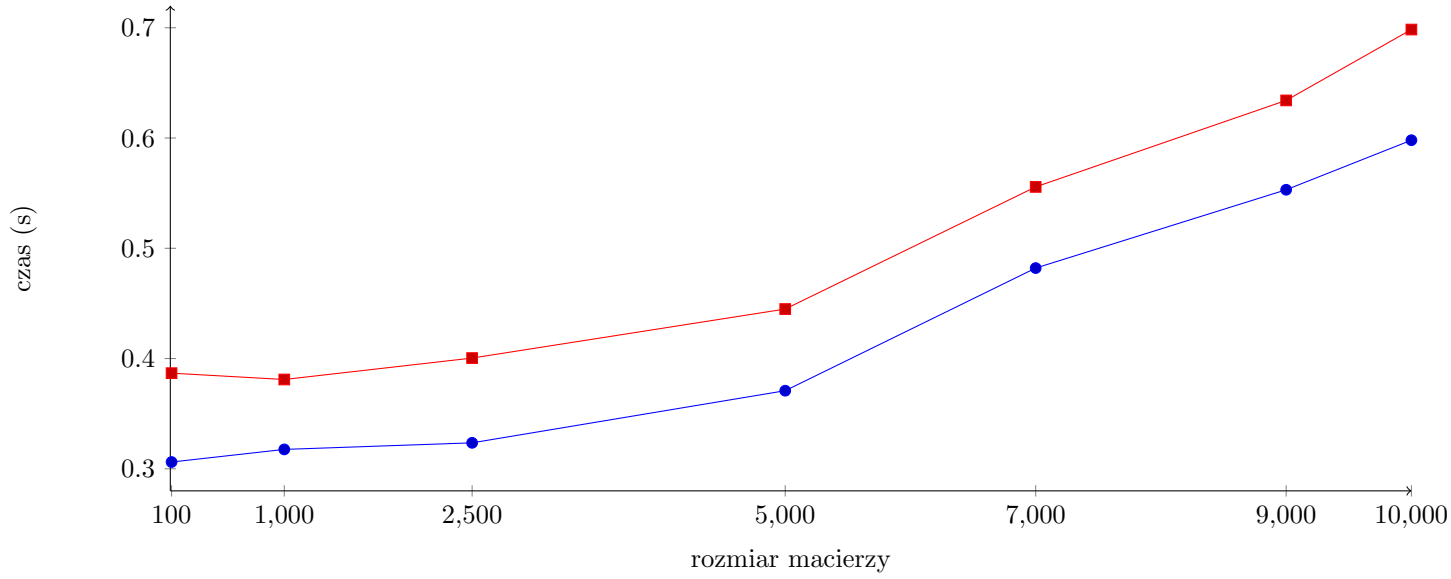
Poniższe testy zostały wykonane na komputerze wyposażonym w czterordzeniowy procesor `Intel Core i5-2400` drugiej generacji *(Sandy Bridge)* i pamięć operacyjną `12GB` z zainstalowanym pakietem *Julia* w wersji `1.5.2` na systemie operacyjnym *Linux Mint* opartym na *Ubuntu*.
Specyfikacja poniższych testów:

- Do testów wykorzystano funkcję `@time` wbudowaną w język *Julia* oraz moduł `testdata` (`code/generate-a-test.jl`) używający programu `blockmat` znajdującego się w pliku `code/test-data/external.jl`.
- Program uruchamiający funkcje z modułu `blocksys` (`code/blocksys.jl`) znajduje się w pliku `code/run-a-test.jl` i może przyjmować argumenty:
  - `classic` (użyj `x = A\b`)
  - `custom` (użyj niestandardowej struktury do przechowywania macierzy rzadkich, zamiast `SparseMatrixCSC`)
  - `pivot` (użyj funkcji `partialPivot`)
  - `lu` (najpierw zrób rozkład LU przed rozwiązaniem układu $Ax = b$)
- Dla wszystkich przeprowadzonych testów odchylenie wektora x (błąd względny) był akceptowalny (błędy rzędu 10^{-15}, 10^{-16}).

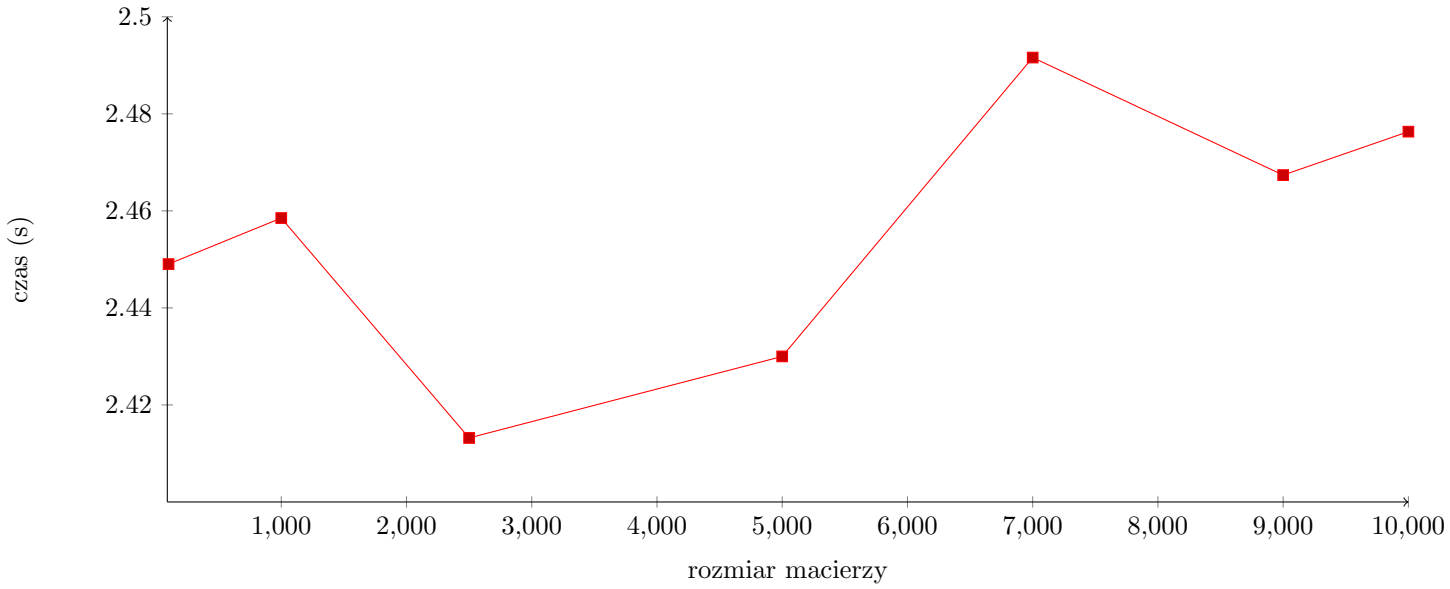
4.1 Czas działania programów



Rysunek 1: Czas działania zoptymalizowanego algorytmu eliminacji Gaussa, funkcja `gaussElimination`, bez (niebieski) i z wyborem elementu głównego (czerwony)

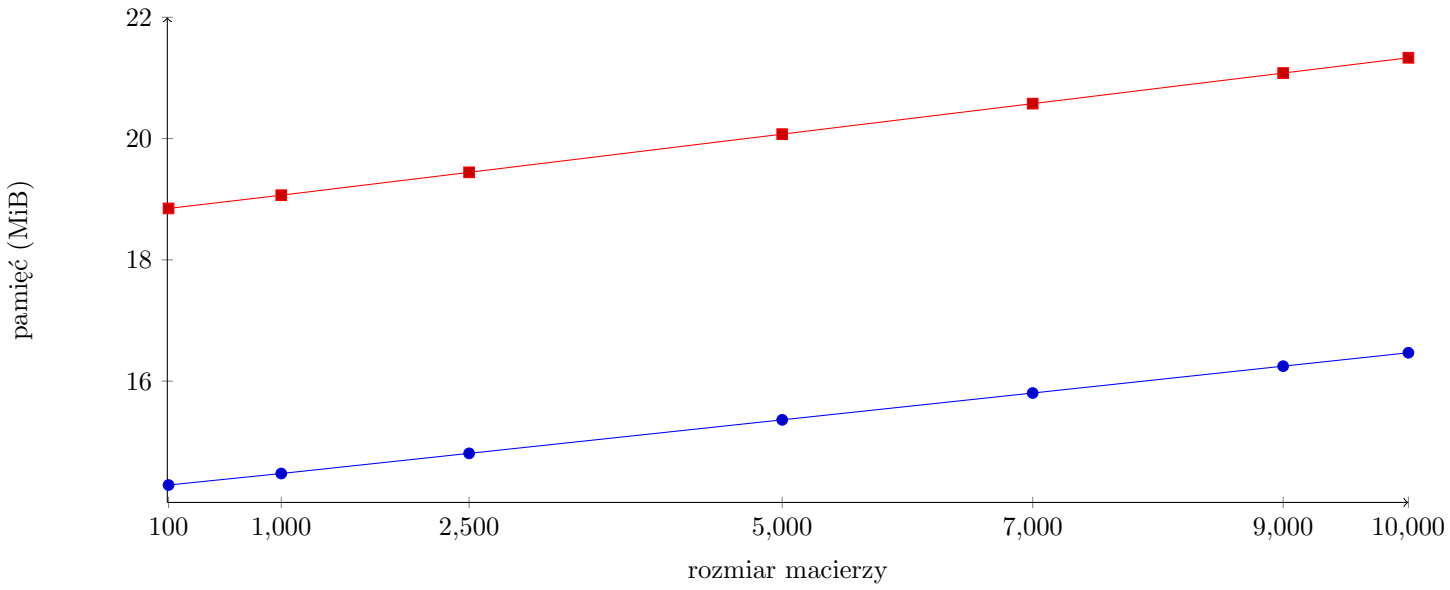


Rysunek 2: Czas działania algorytmu obliczania rozwiązania $LUx = b$, funkcje `decomposeIntoLU` oraz `solveFromLU`, bez (niebieski) i z wyborem elementu głównego (czerwony)

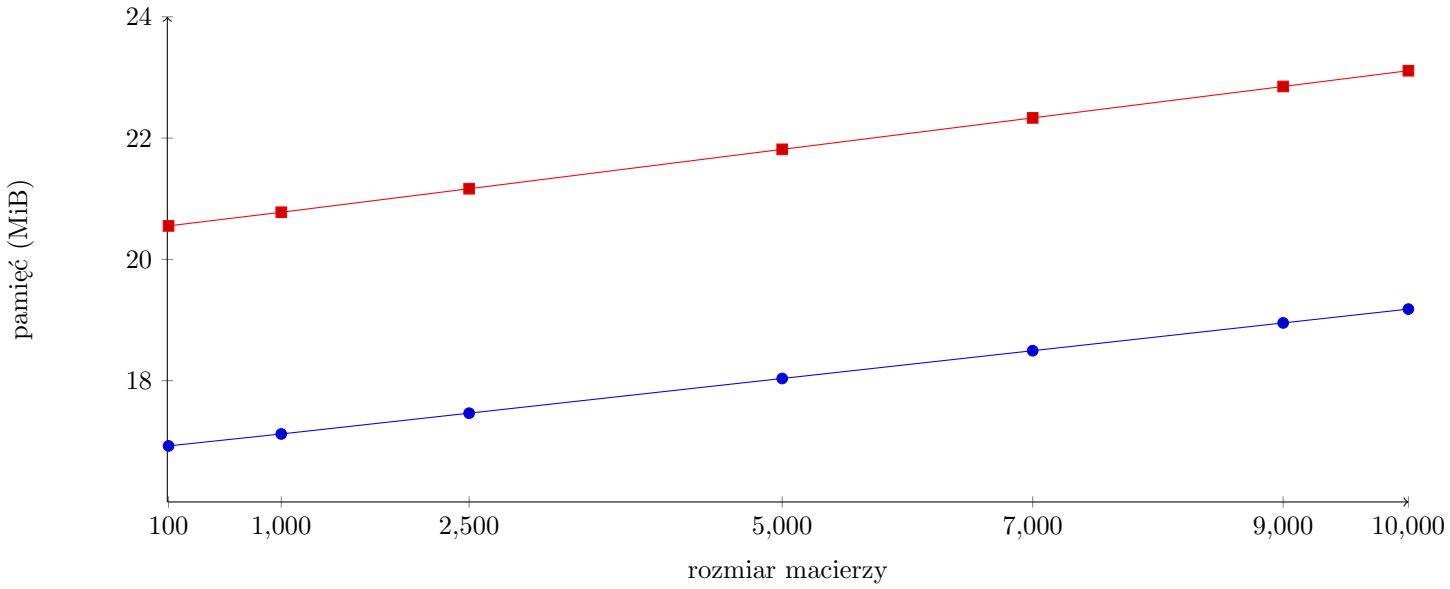


Rysunek 3: Czas działania `x = A\b`

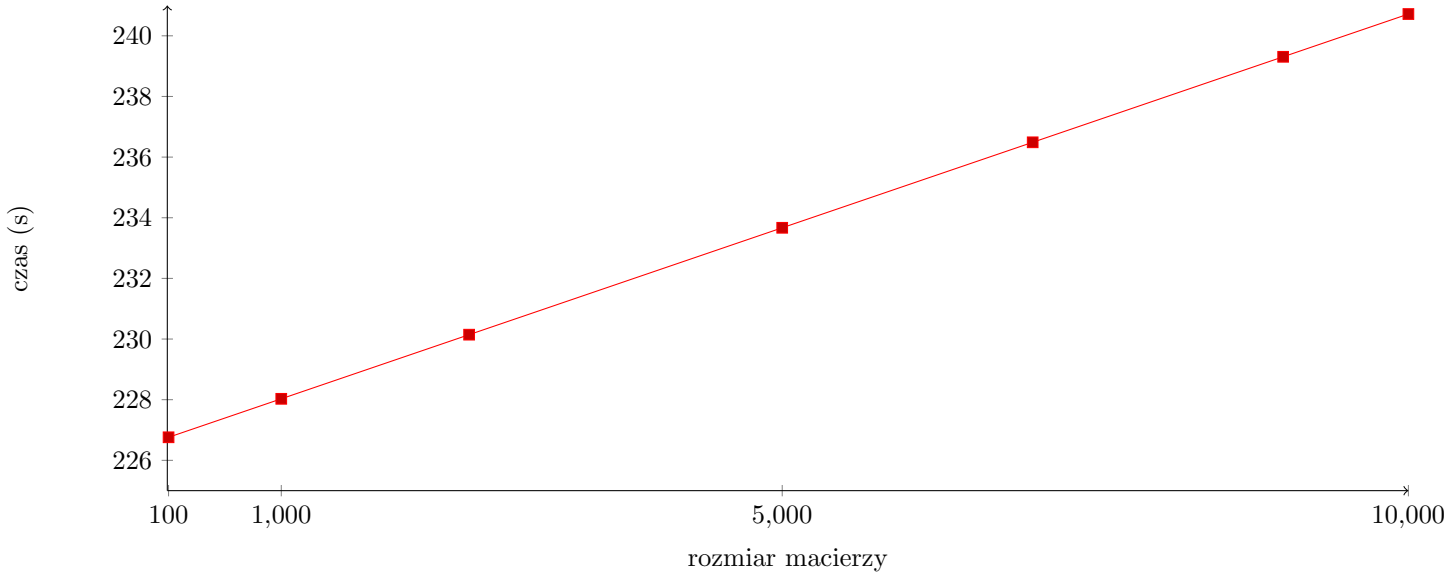
4.2 Zużycie pamięci przez programy



Rysunek 4: Pamięć zajmowana przez zoptymalizowany algorytm eliminacji Gaussa, funkcja `gaussElimination`, bez (niebieski) i z wyborem elementu głównego (czerwony)



Rysunek 5: Zużycie pamięci przez algorytm do obliczania rozwiązania $LUx = b$, funkcje `decomposeIntoLU` oraz `solveFromLU`, bez (niebieski) i z wyborem elementu głównego (czerwony)



Rysunek 6: Czas działania `x = A\b`

4.3 Interpretacja wyników testów

Patrząc ogólnie wyniki są zadowalające — za każdym razem otrzymujemy wynik znacznie szybciej (i zużywając znacznie mniej pamięci) używając zoptymalizowanego algorytmu niż w przypadku standardowego podejścia `x = A\b`. Jednakże warto zwrócić uwagę na to, że czas wykonania programu nie jest do końca liniowy. Złożoność obliczeniowa samego algorytmu jest na pewno liniowa (zakładając, że l jest stałą), więc jedynym wyjaśnieniem są pewne technikalia związane z samym językiem, w którym zostały zaimplementowane programy. Wiemy na przykład, że dostęp do poszczególnych wartości w macierzy typu `SparseMatrixCSC` nie wykonuje się w czasie stałym — być może właśnie dlatego w obserwacjach widać odchylenie od liniowości.

5 Wnioski

Porównując wyniki testów dla zoptymalizowanego algorytmu oraz dla standardowego podejścia `x = A\b` możemy zauważyć oczywiste zalety naszego zoptymalizowanego algorytmu. Zastosowany algorytm jest znacznie szybszy niż podejście klasyczne oraz zużywa znacznie mniej pamięci. Ta sytuacja pokazuje, jak ważne jest dostosowanie algorytmu do danego przypadku i ile można zaoszczędzić zasobów przy znajdowaniu rozwiązania problemu.