Obliczenia Naukowe - laboratorium 5

Łukasz Machnik

8 stycznia 2024

1 Problem

Zadanie polega na rozwiązywaniu układu równań zadanego przez wektor prawych stron b oraz bardzo rzadką macierz A.

$$A = \begin{bmatrix} A_1 & C_1 & 0 & 0 & 0 & \dots & 0 \\ B_2 & A_2 & C_2 & 0 & 0 & \dots & 0 \\ 0 & B_3 & A_3 & C_3 & 0 & \dots & 0 \\ & & & \dots & & & \\ 0 & \dots & 0 & 0 & B_{v-1} & A_{v-1} & C_{v-1} \\ 0 & \dots & 0 & 0 & 0 & B_v & A_v \end{bmatrix}$$

Taka, że $v = n/l$ przy założeniu że n jest podzielne przez l. $A_k, B_k, C_k \in \mathbb{R}^{l \times l}$. A_k jest macierzą gęstą, B_k i C_k są następującej postaci:

$$B_k = \begin{bmatrix} 0 & \dots & 0 & b_1^k \\ 0 & \dots & 0 & b_2^k \\ & \dots & & \\ 0 & \dots & 0 & b_l^k \end{bmatrix}$$

$$C_k = \begin{bmatrix} c_1^k & 0 & \dots & 0 \\ 0 & c_2^k & \dots & 0 \\ & & \dots & \\ 0 & 0 & \dots & c_l^k \end{bmatrix}$$

W związku z taką budową im większa macierz i im mniejszy współczynnik l tym więcej miejsca się "marnuje" na przechowywanie zer.

2 Przechowywanie tablicy

Aby rozwiązać problem marnowania miejsca do przechowywania tablicy zaproponowałem własną strukturę danych przechowującą jak najmniej danych.

```
struct MyMatrix <: AbstractMatrix{Float64}
    n::Int
    l::Int
    A::Matrix{Float64}
    B::Vector{Float64}
    C::Matrix{Float64}
end
```

Struktura ta przechowuje rozmiar macierzy(n), współczynnik l oraz w dwóch osobnych macierzach wszystkie "pod-macierze" A_k i C_k, a także w jednym wektorze wszystkie "pod-macierze" B_k (wystarczy pamiętać tylko ostatnią kolumnę). Do tej struktury danych dopisałem też funkcje ułatwiające pracę na niej tak, aby można było używać składni analogicznej do tej jakbyśmy korzystali ze zwykłej macierzy (całość jest w pliku *matrixstruct.jl*). W ten sposób znacznie ograniczyłem wymagania pamięciowe mojego programu w porównaniu do używania zwykłych macierzy a nawet wbudowanych SparseArrays.

3 Metoda eliminacji Gaussa

Do rozwiązania zadania można wykorzystać metodę eliminacji Gaussa, t.j. odpowiednimi przekształceniami doprowadzić macierz A do postaci macierzy trójkątnej:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22} & a_{23} & \dots & a_{2n} \\ 0 & 0 & a_{33} & \dots & a_{3n} \\ & & & \dots & \\ 0 & 0 & 0 & \dots & a_{nn} \end{bmatrix}$$

A następnie idąc od ostatniego wiersza obliczać wartość wektora x.

```julia
function solveAxb(A::matrixstruct.MyMatrix, b::Vector, withPartialChoice::Bool = false)
    # Inicjalizacja zmiennych
    n = A.n
    l = A.l
    P = ones(Int64, n)  # Tablica do zapamiętywania permutacji
    for i = 2:n
        P[i] = i
    end

    lastI = l # Ostatni rząd potencjalnie mający w i-tej kolumnie coś innego niż 0
    for i = 1:(n-1)
        if i % l == 0 && lastI < n
            lastI += l
        end

        # Wybór częściowy (wykonywany dla pierwszych l-1 elementów każdej "sekcji")
        if withPartialChoice && i % l != 0
            biggestElement = abs(A[i, i])       # Największy element (na moduł) z dotychczas prze
            rowId = i                           # Rząd w którym ten element się znajduje

            # Szukanie największego elementu w danej "sekcji"
            for ti = (i+1):lastI
                if abs(A[ti, i]) > biggestElement
                    biggestElement = abs(A[ti, i])
                    rowId = ti
                end
            end

            # Jeżeli największy element nie jest w i-tym rzędzie to następuje zamiana rzędów w ma
            if rowId != i
                swapRow(A, rowId, i)
                swapRow(b, rowId, i)
                swapRow(P, rowId, i)
            end
        end

        # Eliminacja Gaussa
        if abs(A[i, i]) < eps()
            throw("Niepoprawna macierz: A[$i, $i] = 0")
        end

        for k = (i+1):lastI
            I = A[k, i] / A[i, i]

            section = div(k - 1, l) + 1
            firstJ = (section - 1) * l
            lastJ = (section + 1) * l
            if firstJ <= i
                firstJ = i
```

```
50             end
51             if lastJ > n
52                 lastJ = n
53             end
54 
55             for j = firstJ:lastJ
56                 A[k, j] = A[k, j] - A[i, j] * I
57             end
58             A[k, i] = 0.0
59 
60             b[k] = b[k] - b[i] * I
61         end
62     end
63 
64     # Wyznaczenie wektora x
65     x = zeros(Float64, n)
66     lastJ = n
67 
68     for i = 1:n
69         row = n - i + 1
70 
71         x[row] = b[row]                  # x_n = b_n
72         for j = (row+1):lastJ
73             x[row] -= x[j] * A[row, j]   # x_n = b_n - a_{n+1}x_{n+1} - ...
74         end
75         x[row] /= A[row, row]            # x_n = (b_n - a_{n+1}x_{n+1} - ...) / a_n
76 
77         if row % l == 1 && row < n - l
78             lastJ -= l
79         end
80     end
81 
82     # Odpermutowanie wektora x
83     if withPartialChoice
84         unpermutate(P, x)
85     end
86 
87     return x
88 end
```

Jak widać algorytm ten dzieli się na dwie części. W pierwszej z nich (*11-62*) idąc kolejno wierszami przekształcamy macierz w macierz trójkątną. W tym celu w każdym wierszu bierzemy element znajdujący się na przekątnej (a_{ii}) i eliminujemy współczynniki w tej samej kolumnie w rzędach poniżej. W tym celu dla każdego z tych wierszów wyznaczamy $l_{ki} = \frac{a_{ki}}{a_{ii}}$ mnożymy tę wartość przez wiersz i oraz odejmujemy otrzymany wiersz od wiersza k (*42-61*). Analogicznie przy każdej takiej operacji mnożymy i odejmujemy też wektor prawych stron (*60*).

Następnym etapem jest z otrzymanej macierzy trójkątnej oraz wektora b wyznaczenie wektora x (*68-80*). W tym celu zaczynając od ostatniego wiersza wyznaczamy $x_n = \frac{b_n}{u_{nn}}$ oraz $x_k = \frac{b_k - \sum_{j=k+1}^{n} u_{kj} x_j}{u_{kk}}$ (*71-75*).

Łatwo zauważyć że przy wyznaczaniu współczynników $l_k i$ dzielimy przez elementy znajdujące się na przekątnej. W alternatywnej wersji mojego algorytmu (przełączanej flagą przekazywaną jako argument funkcji) upewniamy się że elementy na przekątnej są co do modułu jak największe (*17-35*). W tym celu przed wybraniem elementu a_{ii} odpowiednio zamieniamy wiersze biorąc pod uwagę wiersze poniżej i (nie zamieniamy kolumn a zatem proces ten nazywamy częściowym wyborem). Aby zachować specyficzną strukturę macierzy największego elementu szukamy tylko w tej samej pod-macierzy A_k. Jeżeli dokonywaliśmy częściowego wyboru to na końcu zamieniamy miejscami elementy wektora x tak aby były na poprawnym miejscu.

4 Rozkład LU

Jeżeli chcemy wielokrotnie obliczać rozwiązanie układu równań dla tej samej macierzy A ale różnych wektorów prawych stron b to przydatny może się okazać tzw. rozkład LU. Jest on niemalże wynikiem wykonywania algorytmu eliminacji Gaussa. Różnica polega na tym że jako że w wyniku algorytmu z poprzedniego zadania otrzymaliśmy macierz trójkątną górną ze współczynnikami równań (U) to w tej samej macierzy, jako macierz trójkątną dolną (L) możemy przechowywać wyznaczane w poprzednim zadaniu współczynniki $l_{ki} = \frac{a_{ki}}{a_{ii}}$ tak aby przy zmianie wektora b nie musieć od nowa przeliczać całej macierzy tylko móc przekształcić wektor b z pomocą macierzy L i od razu móc otrzymać rozwiązanie. Kod jest w tym wypadku bardzo podobny do tego z poprzedniego zadania i wygląda następująco:

```
function LUdistribution(M::matrixstruct.MyMatrix, withPartialChoice::Bool = false)
    # Inicjalizacja zmiennych
    n = M.n
    l = M.l
    P = ones(Int64, n)  # Tablica do zapamiętywania permutacji
    for i = 2:n
        P[i] = i
    end

    lastI = l # Ostatni rząd potencjalnie mający w i-tej kolumnie coś innego niż 0
    for i = 1:(n-1)
        if i % l == 0 && lastI < n
            lastI += l
        end

        # Wybór częściowy (wykonywany dla pierwszych l-1 elementów każdej "sekcji")
        if withPartialChoice && i % l != 0
            biggestElement = abs(M[i, i])       # Największy element (na moduł) z dotychczas prze
            rowId = i                           # Rząd w którym ten element się znajduje

            # Szukanie największego elementu w danej "sekcji"
            for ti = (i+1):lastI
                if abs(M[ti, i]) > biggestElement
                    biggestElement = abs(M[ti, i])
                    rowId = ti
                end
            end

            # Jeżeli największy element nie jest w i-tym rzędzie to następuje zamiana rzędów w ma
            if rowId != i
                swapRow(M, rowId, i)
                swapRow(P, rowId, i)
            end
        end

        # Eliminacja Gaussa
        if abs(M[i, i]) < eps()
            throw("Niepoprawna macierz: M[$i, $i] = 0")
        end
        for k = (i+1):lastI
            I = M[k, i] / M[i, i]

            section = div(k - 1, l) + 1
            firstJ = (section - 1) * l
            lastJ = (section + 1) * l
            if firstJ <= i
                firstJ = i + 1
            end
            if lastJ > n
```

```
                lastJ = n
            end

            for j = firstJ:lastJ
                M[k, j] = M[k, j] - M[i, j] * I
            end
            M[k, i] = I
        end
    end
    return P
end
```

Algorytm zwraca wektor zapamiętujący wykonane permutacje w przypadku aktywowania opcji z częściowym wyborem.

5 Rozwiązywanie równania LUx=b

Jak już wspomniałem wcześniej rozkład LU jest pomocny jeśli chcemy rozwiązywać układ równań dla wielu różnych wektorów prawych stron. Należy w tym celu najpierw z macierzy A wygenerować macierz LU za pomocą algorytmu z poprzedniego zadania, a następnie rozwiązać równanie $LUx = b$. Dla ułatwienia można to rozbić na dwa równania: $Ly = b$ oraz $Ux = y$. Zarówno macierz L jak i U są macierzami trójkątnymi (warto tu nadmienić, że mimo że nie są one zapamiętywane w strukturze danych to macierz L ma na przekątnej same jedynki) zatem obliczenie obu tych równań jest trywialne i analogiczne do obliczeń wykonywanych w algorytmie eliminacji Gaussa.

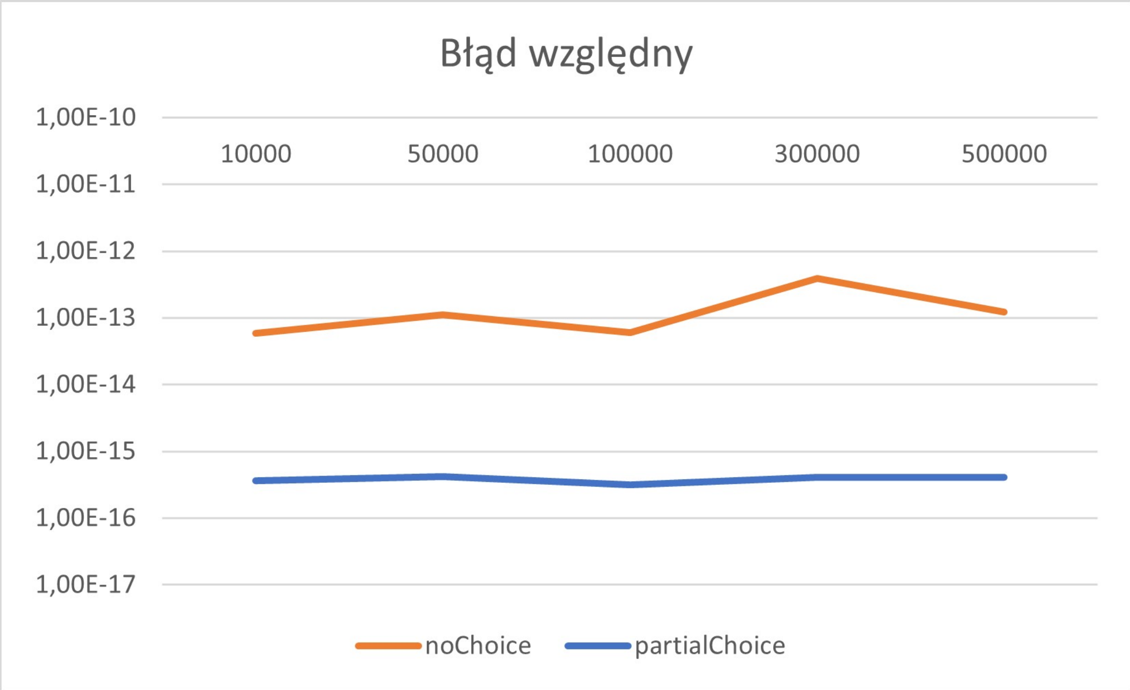
```
function solveLUxb(LU::matrixstruct.MyMatrix, b::Vector, Permutation::Vector = [])
    n = LU.n
    l = LU.l

    # Jeżeli rozkład na macierz LU był z częściowym wyborem to odpowiednio permutuję b
    if !isempty(Permutation)
        for i = 1:n
            next = i
            while Permutation[next] >= 1
                temp = Permutation[next]
                swapRow(b, i, temp)
                Permutation[next] -= n
                next = temp
            end
        end

        for i = 1:n
            if Permutation[i] < 1
                Permutation[i] += n
            end
        end
    end

    # Ly = b
    firstJ = 1

    for i = 1:n
        sum = 0.0
        for j = firstJ:(i-1)
            sum += b[j] * LU[i, j]
        end
        b[i] -= sum

        if i % l == 0
            firstJ += l
            if i == l
```

```
                firstJ -= 1
            end
        end
    end

    # Ux = y
    x = zeros(Float64, n)
    lastJ = n

    for i = 1:n
        row = n - i + 1
        x[row] = b[row]                 # x_n = b_n

        for j = (row+1):lastJ
            x[row] -= x[j] * LU[row, j]  # x_n = b_n - a_{n+1}x_{n+1} - ...
        end

        x[row] /= LU[row, row]           # x_n = (b_n - a_{n+1}x_{n+1} - ...) / a_n
        if row % l == 1 && row < n - l
            lastJ -= l
        end
    end

    # Odpermutowanie wektora x
    if !isempty(Permutation)
        unpermutate(Permutation, x)
    end
    return x
end
```

Algorytm przyjmuje opcjonalny argument - tablicę z permutacją, tak aby móc odpowiednio zmienić kolejność w wektorze b.

6 Testowanie i wydajność

Algorytmy testowałem podstawiając jako rozwiązanie x składający się z samych jedynek i na tej podstawie obliczając wektor b. Obliczone w ten sposób rozwiązania oraz błędy względne otrzymanych wyników (względem wektora $[1, 1, \ldots, 1]$) zapisywałem do plików. Przy użyciu algorytmów "bez wyboru" błąd względny był rzędu mniej więcej 10^{-13} a przy użyciu algorytmów "z wyborem" błąd ten polepszył się do wartości rzędu 10^{-16}. Zależność błędu względnego od rozmiaru danych prezentuje wykres:



Poniższy wykres przedstawia czas wykonywania się funkcji w zależności od rozmiaru danych. Widać że algorytm jest bardzo szybki - wydaje się być liniowy. A zarazem jest dużo mniej wymagający niż podejście "domyślne". Pokazuje to fakt, że przy próbach przechowania tak dużej macierzy jako zwykła macierz już dla rozmiaru 50000×50000 program zwrócił błąd *Out of memory*.