MOWNIT – Laboratorium 4
Metoda dekompozycji LU

Mikołaj Wróblewski

1. Pierwszym zadaniem w ćwiczeniu o metodzie LU było napisanie jej implementacji w oparciu o rozkład Doolittle’a. Metoda LU polega na takim rozbiciu macierzy współczynników A, by A = L * U. W rozkładzie Doolittle’a postać macierzy L oraz u wygląda następująco oraz opisana jest następującymi wzorami:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ l_{n1} & l_{n2} & \cdots & 1 \end{bmatrix} \cdot \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix}.$$

Wyznaczanie kolejnych elementów macierzy $\mathbf{L}$ i $\mathbf{U}$ robi się naprzemiennie, tj. raz wyznacza wiersz macierzy $\mathbf{U}$, raz kolumnę macierzy $\mathbf{L}$.

Wzory ogólne na poszczególne elementy macierzy rozkładu przedstawiają się następująco:

dla wszystkich $i \in \{1, 2, \ldots, n\}$:

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \text{ dla } j \in \{i,\ i+1, \ldots,\ n\},$$

$$l_{ji} = \frac{1}{u_{ii}} \left(a_{ji} - \sum_{k=1}^{i-1} l_{jk} u_{ki} \right) \text{ dla } j \in \{i+1,\ i+2, \ldots,\ n\}.$$

Z ostatniego równania wynika, że metoda nie zadziała, gdy $u_{ii} = 0$.

Liczba działań potrzebna do rozkładu[5]:

- mnożenia: $\frac{1}{3}n^3 - \frac{1}{3}n$,
- dodawania: $\frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n$.

Jak widzimy metoda jest obarczona błędem, kiedy któryś z elementów diagonalnych macierzy U jest zerem nasz algorytm nie zadziała. Rozkład Doolittle’a – podobnie jak podstawowa metoda eliminacji Gaussa – nie chroni nas przed dzieleniem przez zero. Dla rozkładu Doolittle’a wykonałem analogiczne testy jak w poprzednim sprawozdaniu. W kodzie funkcje z poprzedniego sprawozdania są niemal identyczne – poza małymi zmianami związanymi z użyciem macierzy z biblioteki numerycznej numpy - poprawność implementacji sprawdzam oczywiście w oparciu o funkcje biblioteczne. Poniżej zamieszczam zrzut ekranu po testach, w których nie wystąpiło dzielenie przez 0:

```
[mikolaj@mikolaj-pc lab4]$ python lu.py
Tests passed with usage of <function LU_decomposition_doolitle at 0x7f0d2657cf28>
```

Oczywiście jesteśmy w stanie zapobiegać dzieleniu przez 0 poprzez zastosowanie piwotowania.
Następnie dokonałem testów wydajnościowych metody LU z rozkładem Doolittle’a, a także metody Gaussa.

```
[mikolaj@mikolaj-pc lab4]$ python lu.py
DOOLITTLE PERFORMANCE TEST:
4 , 0.000285387
8 , 0.0009691715
16 , 0.0049202442
32 , 0.035033226
64 , 0.1911771297
128 , 1.3973851204
256 , 10.4740922451
512 , 81.2797431946
1024 , 657.9091079235
GAUSS PERFORMANCE TEST:
4 , 0.00020051
8 , 0.0007703304
16 , 0.0045497417
32 , 0.0336232185
64 , 0.2155456543
128 , 1.6690649986
256 , 12.8546085358
512 , 103.6272597313
1024 , 820.7282788754
[mikolaj@mikolaj-pc lab4]$
```

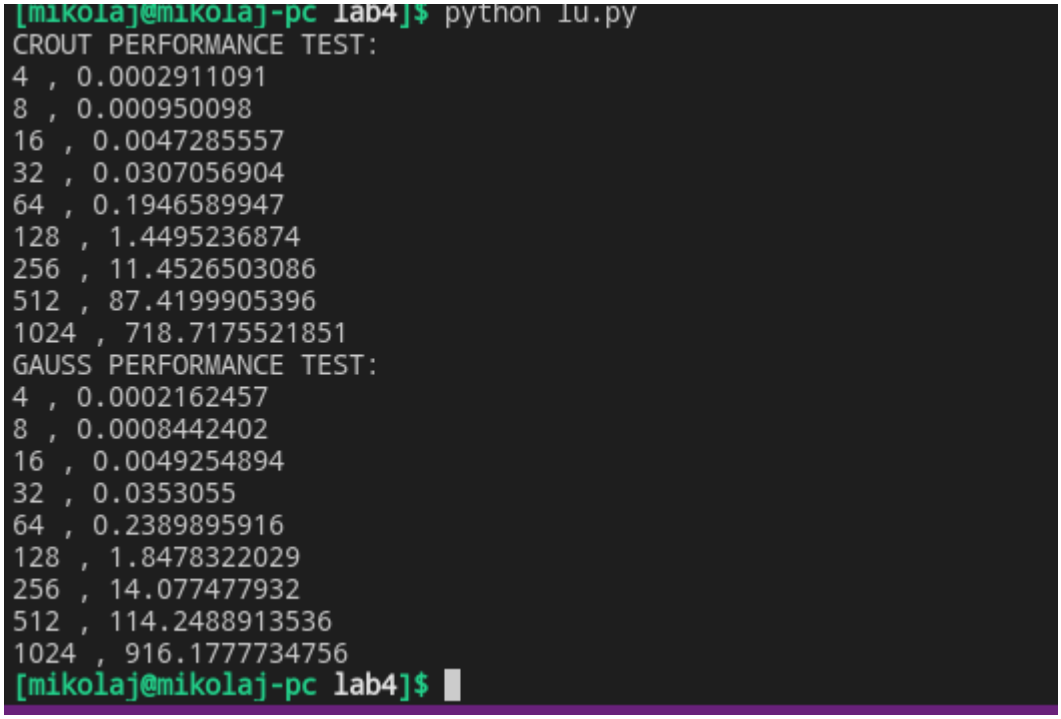
Widzimy, iż dla większych macierzy nasze wyniki zaczynają znacznie się od siebie oddalać, przy większych macierzach na korzyść zdecydowanie wypadła metoda LU.

2. Następnie zmodyfikowałem algorytm rozkładu Doolittle’a w taki sposób, by uzyskać rozkład Crout’a. Różnica w obu rozkładach polega na tym, że w rozkładzie Doolittle’a macierz L ma na diagonali jedynki, a w rozkładzie Crout’a to macierz U ma jedynki na diagonali. Oczywiście w kodzie najpierw wyznaczamy raz wiersz macierzy L, raz macierzy U, a nie raz macierzy U, a raz macierzy L, jak to było w przypadku rozkładu Crout’a. Oczywiście indeksowanie również uległo zmianie.Przetestowałem daną metodę pod kątem poprawności:

```
[mikolaj@mikolaj-pc lab4]$ python lu.py
Tests passed with usage of <function LU_decomposition_crout at 0x7efbf872a048>
```

Przy obu metodach rozwiązanie wyznaczamy w sposób następujący:
 - wyznaczamy ‘z’ z równania: L * z = y (forward substitution)
- następnie wyznaczamy wektor wynikowy ‘x’ z równania:
    U * x = z (backward substitution)

Poniżej zamieszczam zrzut ekranu pomiarów wydajnościowych z użyciem metody dekompozycji Crout’a zestawionych obok metody Gaussa.



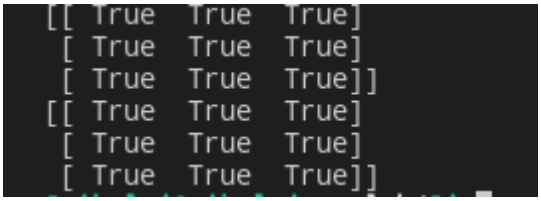
Znowu metoda LU wypada lepiej od metody Gaussa.

3. Następnie zmieniając implementację uzyskałem rozkład Choleskiego. Co ważne rozkład Choleskiego jest aplikowalny tylko i wyłącznie wtedy, kiedy mamy do czynienia symetryczną, dodatnio określoną macierzą. Wygenerowanie losowej macierzy w sposób taki jak w poprzednich przykładach by można było zastosować do niej rozkład Choleskiego jest bardzo trudne, dlatego test poprawności opieram o przykłady z internetu. Warto powiedzieć, że gdy rozkład Choleskiego jest aplikowalny, osiąga wyniki czasowe ok. 2 razy lepsze aniżeli klasyczna metoda LU. Przykłady, których użyłem (wynikiem przedstawionym jest macierz L):

```
25  15  -5              5   0   0
15  18   0       -->    3   3   0
-5   0  11             -1   1   3
```

$$\begin{pmatrix} 4 & 12 & -16 \\ 12 & 37 & -43 \\ -16 & -43 & 98 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 6 & 1 & 0 \\ -8 & 5 & 3 \end{pmatrix} \begin{pmatrix} 2 & 6 & -8 \\ 0 & 1 & 5 \\ 0 & 0 & 3 \end{pmatrix}.$$

Wyniki porównałem oczywiście z wynikami zwracanymi przez funkcję numpy.linalg.cholesky(matrix), program zwraca, co następuje:



Moje testy się zatem powiodły.

Metoda dekompozycji Choleskiego znacznie różni się od dwóch poprzednich metod dekompozycji, po pierwsze ma węższe zastosowanie (co wcale nie oznacza, że wąskie), po drugie zwraca nam de facto jedną macierz, macierz U to po prostu macierz L po transpozycji. Postać naszej macierzy prezentuje się następująco:

Rozpisując iloczyn $A = LL^T$, otrzymujemy:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} & \cdots & l_{n1} \\ 0 & l_{22} & \cdots & l_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & l_{nn} \end{bmatrix}$$

Współczynniki macierzy A są zatem równe:

$$\begin{aligned} a_{11} &= l_{11}^2 && \rightarrow\ l_{11} = \sqrt{a_{11}} \\ a_{21} &= l_{21} l_{11} && \rightarrow\ l_{21} = \frac{a_{12}}{l_{11}} \\ a_{22} &= l_{21}^2 + l_{22}^2 && \rightarrow\ l_{22} = \sqrt{a_{22} - l_{21}^2} \\ a_{32} &= l_{31} l_{21} + l_{32} l_{22} && \rightarrow\ l_{32} = \frac{a_{23} - l_{31} l_{21}}{l_{22}} \\ &\dots \end{aligned}$$

W ogólności[2]:

$$\begin{aligned} l_{ii} &= \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2}, \\ l_{ji} &= \frac{a_{ji} - \sum_{k=1}^{i-1} l_{jk} l_{ik}}{l_{ii}}. \end{aligned}$$

Wspominając o nieważkich zastosowaniach metody Choleskiego, warto byłoby je wymienić, są to między innymi: nieliniowe optymalizacje, symulacje Monte-Carlo, filtry Kalmana, a także odwracanie macierzy.

Wspomnieć należałoby również, kiedy dekompozycja LU istnieje. Jeżeli wyznacznik macierzy jest różny od zera (macierz jest odwracalna), to dekompozycja LU istnieje tylko wtedy, kiedy wiodące minory główne macierzy są różne od 0. Jeżeli macierz jest nieodwracalna, wtedy nie wiemy czy istnieje dekompozycja LU.

Reasumując, nie widać przyspieszenia, jeżeli chodzi o użycie metody rozkładu Crout’a, wręcz lekkie opóźnienie, oczywiście może to wynikać ze sprzętowych uwarunkowań. Użycie metody rozkładu Choleskiego było – niestety – dla losowo generowanych macierzy niemożliwe.