AGH

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

Metody obliczeniowe w nauce i technice

Laboratorium 3

Jan Gołda, WIET 2018/2019

1. Wprowadzenie

Laboratorium miało na celu zbadanie dokładności i wydajności implementacji algorytmu eliminacji Gaussa.

Zadanie wykonano w Jupyter Notebook, pisząc w Pythonie i używając biblioteki do obliczeń naukowych o nazwie NumPy.

Wykorzystany kod oraz wyniki pomiarów są dostępne w repozytorium.

Dla uproszczenia przyjęto że macierz współczynników jest kwadratowa.

1.1. Funkcje pomocnicze

W programie wykorzystano następujące funkcje pomocnicze:

```
# tworzy rozszerzoną macierz [A|B]
def extended_matrix(A,B):
    return np.append(A, B, axis=1)
```

```
# macierz jest osobliwa <=> detA == 0
def is_singular(A):
    return np.isclose(0, np.linalg.det(A))
```

```
# porównuje dwie tablice liczb zmiennopozycyjnych
def compare(A, B):
    return np.all(np.isclose(A, B))
```

2. Naiwny algorytm eliminacji Gaussa

Algorytm ten opiera się tylko na odejmowaniu wielokrotności wierszy, w celu uzyskania macierzy ukośnej.

Algorytm ten nie wykorzystuje zamian wierszy/kolumn, przez to podatny na błąd dzielenia przez zero jeżeli na przekątnej macierzy znajduje się zero. Jako że ma to być prosta wersja eliminacji Gaussa założone że takich danych nie będzie.

2.1. Implementacja

```
def gauss(A, B):

    # check for singularity
    if is_singular(A):
        raise ValueException("Cannot solve equation for singular...")

    # create extended matrix
    U = extended_matrix(A, B)

    # assuming that matrix A is n by n
    n = U.shape[0]

    # trianguralize matrix
    for i in range(n):
        for j in range(i + 1, n):
            factor = U[j,i] / U[i,i]

            for k in range(n + 1):
                U[j,k] -= U[i,k] * factor

    # diagonalize matrix
    for i in range(n - 1, -1, -1):
        for j in range(i - 1, -1, -1):
            factor = U[j,i] / U[i,i]

            for k in range(n + 1):
                U[j,k] -= U[i,k] * factor

    # divide to acquire ones at diagonal
    for i in range(n):
        U[i,n] /= U[i,i]
        U[i,i] /= U[i,i]

    # return last column as matrix X
    return U[:,n:n+1]
```

2.2. Weryfikacja poprawności działania dla losowych macierzy

W celu sprawdzenia poprawności działania powyższego kodu porównano jego działanie z funkcją z biblioteki NumPy o nazwie *np.linalg.solve*.

Testy przeprowadzono na losowym układzie 5 równań z 5 niewiadomymi:

```
A = np.random.randn(5, 5).astype("float32")
B = np.random.randn(5, 1).astype("float32")

compare(np.linalg.solve(A, B), gauss(A, B))
# -> True
```

Wyniki okazały się spójne (z dokładnością do epsilona), a zatem na razie można uznać za program działa poprawnie.

2.3. Weryfikacja poprawności działania dla problematycznych macierzy

Program przetestowano również dla układu równań w którym poszczególne współczynniki różnią się o kilka rzędów wielkości:

```
A = np.array([
      [-0.004,  -0.0001,   10.0],
      [   9.0,   0.0004,   -4.0],
      [  11.0,     -2.5,  0.005]
]).astype("float32")

B = np.array([
      [    8],
      [8.001],
      [  2.5]
]).astype("float32")

compare(np.linalg.solve(A, B), gauss(A, B))
# -> False
```

W tym przypadku wynik testu jest negatywny, a wyniki poszczególnych algorytmów wyglądają następująco:

NumPy	Gauss
1.24459780	1.24485790
4.47783140	4.46975200
0.80054260	0.80054265

Jak widać mamy tu do czynienia z błędem, który przypomina błędy z jakimi mieliśmy do czynienia przy operacjach na liczbach zmiennopozycyjnych. Jest to uzasadnione podejrzenie ponieważ metoda eliminacji Gaussa dokonuje znacznej liczby operacji dzielenia, mnożenia i odejmowania, co przy liczbach różniących się o kilka rzędów doprowadza do zauważalnych strat na dokładności.

3. Poprawiony algorytm eliminacji Gaussa

Rozwiązaniami mającymi zmniejszyć błąd z poprzedniego punktu są: skalowanie macierzy i pivoting.

Skalowanie macierzy – polega na wykorzystaniu możliwości przemnażania wierszy tak aby największą wartością w wierszu była jedynka.

Pivoting – polega na takiej zamianie wierszy aby na przekątnej znajdowała się największa możliwa wartość z danej kolumny.

Obydwa te rozwiązania pozwalają na zniwelowanie konieczności dokonywania operacji na liczbach znacznie się różniących cechą, a zatem pozwalają zredukować błąd całej metody.

3.1. Implementacja

```
def improved_gauss(A, B):

    # check for singularity
    if is_singular(A):
        raise ValueException("Cannot solve equation for singular matr
ix")

    # create extended matrix
    U = extended_matrix(A, B)

    # assuming that matrix A is n by n
    n = U.shape[0]

    # scale matrix
    factor = np.max(np.absolute(A), axis = 1)
    for i in range(n):
        for k in range(n+1):
            U[i,k] /= factor[i]

    # trianguralize matrix
    for i in range(n):

        # pivoting - swapping rows
        row = i
        for j in range(i+1, n):
            if abs(U[j,i]) > abs(U[row,i]):
                row = j
        U[[i,row]] = U[[row, i]]

        # same as in normal gaussian
        for j in range(i + 1, n):
            factor = U[j,i] / U[i,i]

            for k in range(n + 1):
                U[j,k] -= U[i,k] * factor

    # diagonalize matrix
    for i in range(n - 1, -1, -1):
        for j in range(i - 1, -1, -1):
            factor = U[j,i] / U[i,i]

            for k in range(n + 1):
                U[j,k] -= U[i,k] * factor

    # divide to acquire ones at diagonal
    for i in range(n):
        U[i,n] /= U[i,i]
        U[i,i] /= U[i,i]

    # return last column as matrix X
    return U[:,n:n+1]
```

3.2. Weryfikacja działania algorytmu

Przetestowano działanie poprawionego algorytmu na tych samych danych na których testowano naiwny wariant. Zestawienie wyników wygląda następująco:

NumPy	Gauss	Poprawiony Gauss
1.24459780	1.24485790	1.24459780
4.47783140	4.46975200	4.47783100
0.80054260	0.80054265	0.80054270

Jak widać poprawiony algorytm co prawda nie daje idealnych wyników, ale wykazuje znaczną poprawę względem naiwnego algorytmu.

4. Porównanie wydajności algorytmów

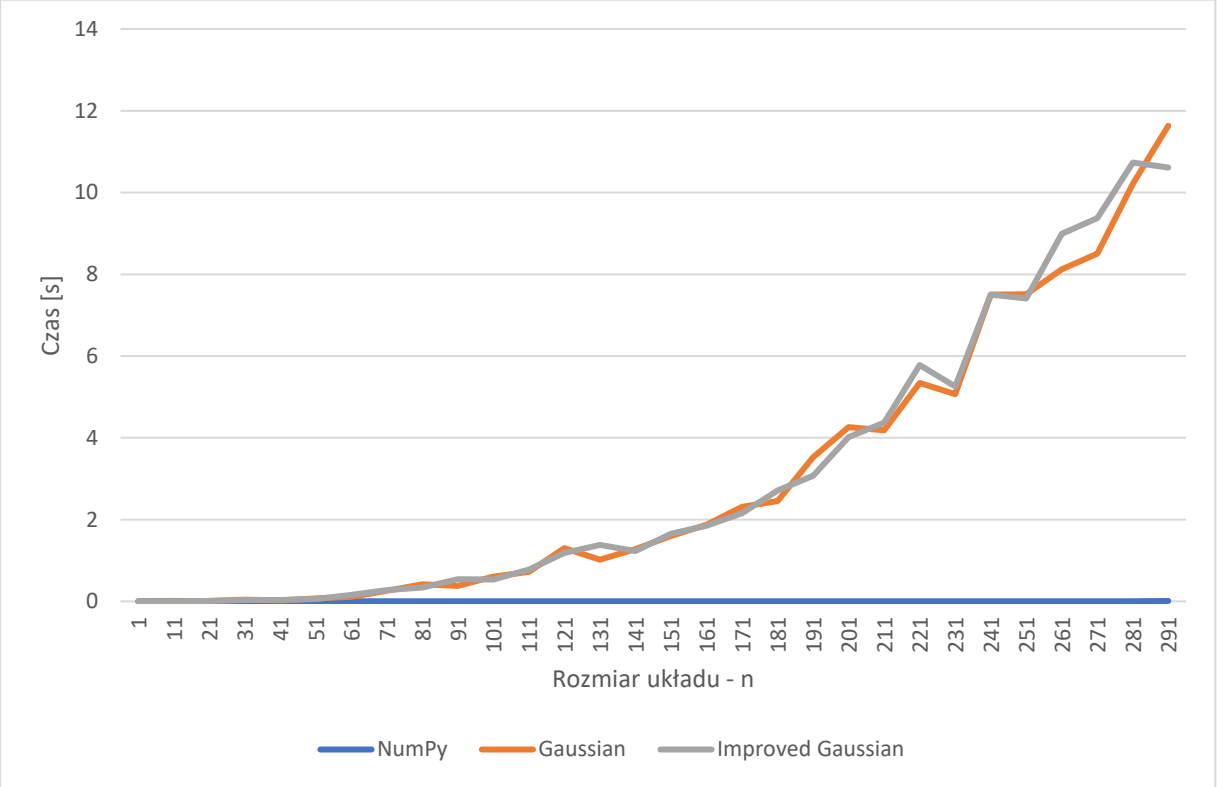
Pomiarów wydajności dokonano przy użyciu funkcji **%timeit** dostępnej w Jupyter Notebook.

Pomiary wykonywane były dla losowego układu n równań o n nie wiadomych.

Każdy pomiar został powtórzony trzy razy, a wynik końcowy jest średnią z tych trzech pomiarów.

Pełne dane pomiarowe dostępne są tutaj.

4.1. Wyniki pomiarów



5. Wnioski

Najważniejszym wnioskiem jest to, że funkcja biblioteczna jest wielokrotnie szybsza od implementacji eliminacji Gaussa, w tej skali wydaje się wręcz, że ma ona stałą złożoność czasową (co oczywiście nie jest prawdą).

Przy tak nie dokładnych pomiarach czasu trudno określić która wersja eliminacji Gaussa jest szybsza, ich czasy są bardzo zbliżone. Jeżeli jednak weźmiemy pod uwagę dokładność obliczeń można stwierdzić że ulepszony algorytm radzi sobie zacznie lepiej.

Ostatnim mnie istotnym spostrzeżeniem jest to czasy wykonania zależą delikatnie od wartości w macierzy (wynika to z operacji na liczbach zmiennopozycyjnych).