Algorytmy algebry liniowej wraz z szacowaniem złożoności obliczeniowej

Bartłomiej Hofman Katarzyna Dawidowicz

Styczeń 2020

1 Algorytmy algebry liniowej

Celem niniejszego sprawozdania będzie porównanie dwóch metod rozwiązujących przekazany im układ równań liniowych: metody wyznacznikowej oraz metody eliminacji. Analiza zostanie przeprowadzona na serii zagadnień, które będą różniły się od siebie liczbą równań. Dodatkowo, dla każdej z metod zostanie zmierzony potrzebny czas na wykonanie obliczeń przez system komputerowy. Aby wyniki były wartościowe, pomiary zostaną wykonane dla serii danych wejściowych, w tym przypadku macierzy, które będą różniły się między sobą wymiarem (od 50 do kilkuset równań). Wszystkie pomiary wykonane zostaną na tym samym sprzęcie i systemie, co umożliwi względne porównanie ze sobą otrzymanych wyników. Uzyskane dane zostaną przedstawione w formie tabelarycznej przedstawiającej zależność między wielkością układu, a czasem potrzebnym na jego rozwiązanie.

Drugim zagadnieniem dotyczącym tej pracy będzie zbadanie złożoności obliczeniowej na podstawie pomiarów czasu i wymiaru zadania jednego z grup algorytmów omawianych na ćwiczeniach, które wykonuje się wielokrotnie dla dużych zbiorów w przypadku algorytmów sortujących czy wyszukiwania jak i np. rozmiarów układów równań liniowych w przypadku algorytmów algebry liniowej. Ze względu na charakter tego sprawozdania badaną złożoność będziemy opierać na algorytmach obliczających rozwiązania układów równań liniowych odpowiednio Metodą Wyznaczników oraz Eliminacji Gaussa-Jordana. Dane zostaną pokazane na wykresie w celu sprawdzenia, czy punkty układają się wzdłuż pewnej krzywej, którą można opisać funkcją oraz porównać do złożoności obliczeniowej algorytmu.

2 Generator macierzy i wyrazów wolnych

W celu wygenerowania serii zagadnień różniących się liczbą równań, posłużyliśmy się funkcją o nazwie *generator*, która została opracowana wcześniej na zajęciach.

Funkcja przyjmuje cztery argumenty: rozmiar macierzy, wartość, która ma znaleźć się na przekątnej oraz wartości, które znajdą się kolejno powyżej i poniżej przekątnej. Dodatkowo w sposób losowy tworzy wektor wyrazów wolnych.

2.1 Postać funkcji generatora:

```
def generator(size, diag, up, down):
    A = np.zeros((size, size))

    B = np.zeros((size, 1))
    for i in range(size):
        for k in range(size):
            A[i, i] = diag
            for j in range(i + 1, size):
                A[i, j] = up
            for j in range(i):
                A[i, j] = down

            B[k, 0] = random.randint(1, size)

    return A, B
```

Macierze oraz wektory wyrazów wolnych wygenerowane w ten sposób, posłużyły nam w kolejny etapach analizy metod rozwiązywania układów liniowych.

3 Metoda wyznaczników

Metoda Cramera wykorzystuje twierdzenie mówiące o wyznacznikach macierzy, dzięki którym jesteśmy w stanie rozwiązać układ równań liniowych.
Metody tej jednak nie da zastosować się do każdego układu. W tym przypadku jest mowa o układach, gdzie liczba niewiadomych jest równa liczbie równań. Oznacza to, że macierz współczynników równania jest macierzą kwadratową wymiaru $n \times n$. Dodatkowo należy pamiętać, że aby układ równań liniowych był układem Cramera, wyznacznik główny W musi być różny od zera. W przeciwnym wypadku układ równań jest sprzeczny lub liniowo zależny i nie posiada jednoznacznego rozwiązania.
Aby znaleźć wartości kolejnych niewiadomych, postępujemy następująco: W macierzy A współczynników dla każdej niewiadomej $x_i, i = 1, 2, \ldots, n$, zastępujemy i-tą kolumnę wektorem kolumnowym B. Następnie wyliczamy wyznacznik W_i tak zmodyfikowanej macierzy. Wartość kolejnych niewiadomych $x_i, i = 1, 2, \ldots, n$, otrzymamy za pomocą poniższego wzoru:

$$x_i = \frac{W_i}{W} \tag{1}$$

3.1 Kod programu

Postać funkcji obliczającej rozwiązanie układu równań liniowych metodą wyznaczników:

```
def cramer(m,w):
    x = np.zeros((len(m),1))
    determinant = np.linalg.det(m)
    for i in range(len(m)):
        m_1 = m.copy()
        m_1[:,i] = w[i,:]
        x[i] = (np.linalg.det(m_1)/determinant)
    return x
```

3.2 Działanie programu

1. Definiujemy funkcję o nazwie cramer, która przyjmuje dwa argumenty: m, czyli macierz kwadratową oraz w, czyli wektor wyrazów wolnych.
2. Następnie tworzymy pustą macierz x. Zmienna ta będzie przechowywać wektor rozwiązań, do której, w późniejszych etapach dopisywane będą kolejne wartości niewiadomych x_i.
3. Kolejnym krokiem jest zdefiniowanie zmiennej *wyznacznik*, która przechowuje wartość wyznacznika macierzy policzonej za pomocą gotowej funkcji *numpy.linalg.det*().
4. Tworzymy jedną pętle, które wykonają się dla i-tej kolumny macierzy tyle razy, ile posiadamy wyrazów wolnych. Tutaj następuje skopiowanie wartości z macierzy m do roboczej zmiennej m_1 funkcją *copy*.
5. Idać dalej w każdej iteracji kolejna kolumna jest modyfikowana poprzez podstawienie do niej wektora wyrazów wolnych, która przy następnym obrocie pętli wraca do swojej pierwotnej postacji, ponieważ kolejny raz wykona się skopiowanie macierzy funkcją *copy* $*$.
6. Przy pomocy funkcji *linalg.det* z modułu *numpy* obliczane są wyznaczniki. Następnie według wzorów Cramera wyznaczane są kolejne niewiadome x_i, a ich wartości są dodawane do pustej macierzy przechowywanej w zmiennej x.
7. Na koniec funkcja zwraca nam wektor rozwiązań układów równań.

4 Metoda eliminacji Gaussa

Metoda z uwagi na łatwość implementacji programowej jest bardzo szeroko rozpowszechnioną metodą rozwiązywania układów liniowych. Przebiega ona w dwóch głównych etapach:

- Sprowadzenie macierzy do postaci macierzy trójkątnej
- Redukcja wsteczna, która ma na celu wyliczenie wartości poszukiwanych zmiennych

Jednym z niebezpieczeństw tej metody jest eliminacja zmiennych, która czasem może prowadzić do dzielenia przez 0. Wówczas trzeba wziąć pod uwagę, że zamiana wierszy miejscami nie będzie wpływać na rozwiązanie układu i w ten sposób można zażegnać niebezpieczeństwo dzielenia przez 0. Warto jednak zaznaczyć, że zamiana wierszy nie zawsze może okazać się możliwa ze względu na niespełnienie warunku, jakim jest znalezienie poniżej wybranego wiersza takiego, który ma zmienną różną od 0. W takim przypadku układ nie będzie miał jednego rozwiązania, ponieważ może ich mieć nieskończenie wiele.

4.1 Kod programu

Postać funkcji sprowadzającej macierz do postaci trójkątnej oraz obliczającej rozwiązanie układu równań liniowych metodą eliminacji Gaussa.

```
def gauss_elimination(A,B):
    n = len(B)
    x = numpy.zeros(n,float)

    #Elimination
    for k in range(n-1):
        #Protection if zeros are on diagonal
        if A[k,k] == 0:
            for i in range(k+1, n):
             if A[i,k] != 0:
                A[[k,i]] = A[[i,k]]
                B[[k,i]] = B[[i,k]]
                break #Brake when the rows are changed
        #Setting values to 0 at the diagonal
        for i in range(k+1,n):
            wspl = A[k,k] / A[i,k]
            for j in range(k,n):
                A[i,j] = A[k,j] - (A[i,j] * wspl)
            B[i] = B[k] - (B[i]*wspl)

    #Backward solution
    x[n-1] = B[n-1] / A[n-1,n-1]
    for i in range(n-2,-1,-1):
        sums = 0
        for j in range(i+1,n):
            sums = sums + A[i,j] * x[j]
        x[i] = (B[i] - sums) / A[i,i]

    return x
```

4.2 Działanie kodu

4.2.1 Eliminacja

1. Definiujemy funkcję *eliminacjagaussa*, w której również jako parametr przekazujemy układ równań w postaci macierzy kwadratowej oraz wektor wyrazów wolnych.

2. Instrukcje zaczynają się od zdefiniowania długości wektora rozwiązań oraz pustej macierzy x, która przechowywać będzie wektor rozwiązań układu równań

3. Proces eliminacji wymaga czterech niezbędnych pętli for:

   - Pierwsza pętla przechodząca przez wszystkie wiersze.
   - Druga, która odpowiada za zamianę wierszy, gdy na przekątnej napotkamy 0.
   - Trzecia przechodząca przez wszystkie kolumny
   - Czwarta, która wykonuje operacje zerowania poszczególnych wartości w k-tym wierszu, i-tej kolumnie

4. Druga iteracja jest poprzedzona warunkiem, który ma sprawdzić czy wartość bezwzględna z k-tego wiersza k-tej kolumny jest równa zeru. Gdy warunek jest spełniony, to następuje sprawdzenie, czy następny wiersz jest większy od wiersza bieżącego. Wówczas następuje zamiana wierszy miejscami oraz przejście do następnej operacji. Taki zabieg zamiany wierszy w których występuje liczba 0 na przekątnej jest konieczny w celu prawidłowego wyzerowania macierzy pod przekątną.

5. Podczas trzeciej iteracji dla wartości niezerowych wyliczana jest wartość wspl, która jest wykorzystywana w dalszych instrukcjach celem zerowania wartości pod przekątną macierzy.

6. W końcowym etapie procesu sprowadzenia macierzy do postaci trójkątnej górnej jest również niezależnie modyfikowany wektor rozwiązań układu.

4.2.2 Redukcja wsteczna

Druga część kodu odpowiada za obliczenie wektora rozwiązań metodą podstawiania, która zaczyna się od końca. Od końca, ponieważ mając macierz trójkątną górną wartość znajdująca się w ostatnim wierszu w ostatniej kolumnie jest jednocześnie rozwiązaniem jednej z niewiadomych.

1. Właściwa procedura zaś rozpoczyna się od obliczenia ostatniej wartości niewiadomej.

2. Następnie w celu przejścia przez cały układ równań konieczne jest użycie dwóch pętli:

- Pierwsza, która od końca, przesuwa się w górę układu z krokiem -1
- Druga, która sumuje wartości niewiadomych pomnożone, przez współczynnik, który przy nim się znajduje

3. Będąc już poza wewnętrzną pętlą wartość każdej kolejnej niewiadomej x_i jest obliczana oraz dołączana do wektora rozwiązań.

4. Instrukcją return zwracamy z funkcji wektor rozwiązań układu równań.

5 Pomiary czasowe działania poszczególnych metod dla różnych wielkości układu równań

Liczba równań	Czas wykonywania		
	M. Wyznaczników	M. Elim. Gaussa	M. Biblioteczna
50	0.005142567000802956	0.0029755089999525808	0.000131563999989885
100	0.034331930000917055	0.009966904999600956	0.005070820006949361
150	0.11526559199955955	0.021535487998335157	0.00582989800022915
200	0.33505518800120626	0.04106358100034413	0.02255249599875242
250	0.7443063360005908	0.05719664899879717	0.008132040000418783
300	2.498606536999432	0.08035564400051953	0.006313509000392514
500	12.300197139000375	0.2769476490029774	0.018050721000690828
600	21.238422148999234	0.38976130100127193	0.023209494998809532
700	27.029957020000438	0.580783371002326	0.03068042099948798
800	44.714774731000944	0.6933282669997425	0.05641992500022752
1000	109.30554160599968	1.0908481719998235	0.1014957190000132
1100	167.02598880200094	2.0726115268	0.14332926600036444

Tabela 1: Tabela pomiarów czasowych różnych metod dla poszczególnych wielkości układu równań

5.1 Wnioski

Wszystkie pomiary zostały wykonane w tych samych warunkach tj. tym samym interpreterem, używając tych samych funkcji oraz realizując obliczenia we względnie tych samych warunkach pod takim samym obciążeniem procesora.

Wraz ze wzrostem liczby równań dla każdej z metod, wzrasta czas ich rozwiązywania. Jednak w zależności od użytej metody, czas ten wzrasta w mniejszym lub większym stopniu. Największe przyrosty możemy zaobserwować dla metody wyznaczników. W jej przypadku dla liczby równań rzędu tysiąca czas działania programu jest ponad 80 razy dłuższy niż dla metody eliminacji przy tej samej liczbie równań. Z kolei gotowa funkcja biblioteki numpy robi to 15 razy szybciej niż metoda eliminacji a prawie 1200 razy szybciej niż metoda wyznacznikowa dla tej samej liczby równań.

Może to wynikać z faktu, iż w metodzie Cramera istnieje konieczność obliczania wyznaczników macierzy z każdym podstawieniem w iteracji, co tylko dodaje nam operacji arytmetycznych do wykonania. W efekcie dla układów równań rzędu kilkuset lub kilku tysięcy, metoda jest niewydajna, dosyć czasochłonna i może stanowić jego główne wady.

Bazując na wiedzy na temat dwóch metod oraz uzyskanych wynikach, możemy stwierdzić, że to metoda eliminacji Gaussa jest dużo lepszym rozwiązaniem, pomimo tego że w pierwszej kolejności należy sprowadzić postać macierzy do postaci trójkątnej górnej, lub dolnej. Sprawia to, że operacji do wykonania przez całą funkcję jest więcej niż w metodzie Cramera, lecz ze względu na obliczanie wektora rozwiązań metodą „od końca" mając macierz trójkątną, z każdym obrotem pętli rozwiązujemy kolejną niewiadomą układu.

Jednak najefektywniejszą metodą spośród wymienionych, okazała się metoda linalg.solve(), czyli gotowa funkcja obliczająca wektor rozwiązań układów równań liniowych znajdująca się w module numpy. Metoda ta również opiera się na sprowadzeniu macierzy do postaci trójkątnej i jej rozwiązaniu. Działa jednak o rząd szybciej, ponieważ jej kod został wstępnie skompilowany.

6 Spełnienie układów równań

W celu sprawdzenia, czy wektor rozwiązań spełnia układ równań użyliśmy napisanej podczas zajęć funkcji $MxW(m, w)$ służącej do mnożenia dwóch macierzy. Kod programu jest zamieszczony poniżej:

```
def MxW(a, b):
    nw, nk = a.shape
    assert b.shape == (nk,)
    w = numpy.zeros((nw,))
    for i in range(nw):
        cos = 0.0
        for j in range(nk):
            cos += a[i,j]*b[j]
        w[i] = cos
    return w
```

Po wyznaczeniu wektora rozwiązań Metodą Cramera oraz Metodą Gaussa, chcieliśmy sprawdzić czy układ równań jest spełniony. Korzystając z własności ukł. równań w postaci macierzowej tj.

$$MX = W \tag{2}$$

wiemy, że

$$\varepsilon = MX - W \tag{3}$$

Gdzie ε jset błędem, który został popełniony podczas wykonywanych obliczeń. W podany sposób jesteśmy w stanie obliczyć błąd z jakim obarczone jest rozwiązanie układu równań. Obliczając wynik wektora niewiadomych dla każdej

z metod, następnie przemnażając przez siebie macierz początkową z wektorem niewiadomych otrzymujemy wszystkie potrzebne składowe do policzenia błędu i wykonania przedstawionej powyższej operacji. Poniżej przedstawiamy wykonanie opisanych powyżej czynności:

```
macierz = generator(10,4,2,3)

m = macierz[0]
w = macierz[1]

m_kopia = m #skopiowanie macierzy wyjściowej
r = w.T #transponowanie wektora z postaci kolumny do wiersza

bledy_cramer = []
bledy_gauss = []

#obliczenia wektorów
X = eliminacja_gaussa(m,w)
Y = cramer(m,w)

#mnożenie skopiowanych macierzy wyjściowych przez wektor rozwiązań
AX = mxw(m_kopia,X) #Gauss
AY = mxw(m_kopia,Y) #Cramer
#Obarczenie błędu:
epsilon_gauss = AX - r[0]
epsilon_cramer = AY - r[0]

bledy_gauss.append(max(abs(epsilon_gauss)))
bledy_cramer.append(max(abs(epsilon_cramer)))
```

Sprawdzenia błędu zostały wykonane na różnych wielkościach macierzy. Dla ich największych wartości bezwzględnych stworzyliśmy tabelę, która obrazuje jej wielkość w zależności od układu równań:

W przypadku obydwu metod zauważamy, że błąd obliczeń rośnie wzraz ze wzrostem rozmiaru układu równań, dla którego obliczenia mają się wykonać. Widać jednak, że w przypadku metody Cramera ten błąd rośnie zdecydowanie szybciej niż w przypadku metody Elim. Gaussa ze względu na ilość operacji, którą algorytm musi wykonać, aby dojść do rozwiązania układu.

7 Szacowanie złożoności obliczeniowej na podstawie pomiarów czasu

We wcześniejszym fragmencie niniejszego sprawozdania, została przedstawiona tabela zawierająca czas potrzebny na rozwiązanie układu równań trzema różny-

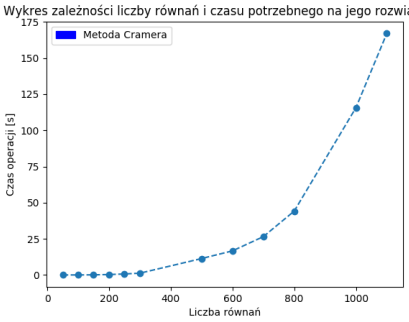
Liczba równań	Wielkość błędu	
	M. Wyznaczników	M. Elim. Gaussa
50	4.376943252282217e-12	5.684341886080802e-14
100	1.2317258324401337e-11	1.2789769243681803e-13
150	2.688693712116219e-11	2.2737367544323206e-13
200	8.290612640848849e-11	3.115907697472721e-13
250	2.2469137661573768e-10	5.765876499528531e-13
300	3.8191672047105385e-10	8.526512829121202e-13
500	2.1856294551980682e-09	1.9326762412674725e-12

Tabela 2: Tabela błędów rozwiązań dwóch metod w zależności od wielkości układu

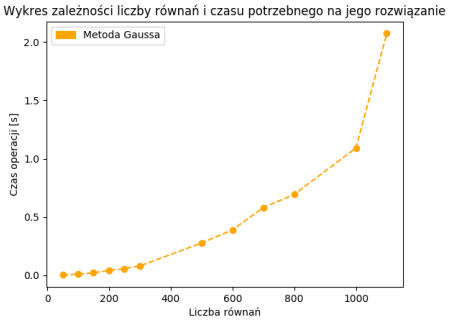
mi metodami dla różnych wielkościach macierzy. Na jej podstawie stworzyliśmy wykresy zależności rozmiaru układu równań, a czasu potrzebnego na jego rozwiązanie dla każdej z trzech metod.

7.1 Wnioski

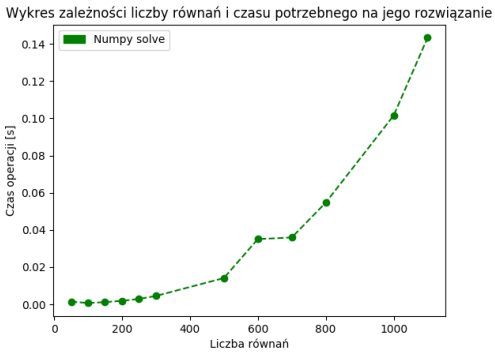
Jak widać na wykresach i co zostalo zasugerowane na ćwiczeniach, punkty układają się wzdłuż pewnej krzywej dla każdej z metod. Zauważamy również pewną zależność: Im większy układ równań, tym więcej czasu potrzebują wszystkie metody na obliczenie wektorów ich rozwiązań.



(a) M. Cramera

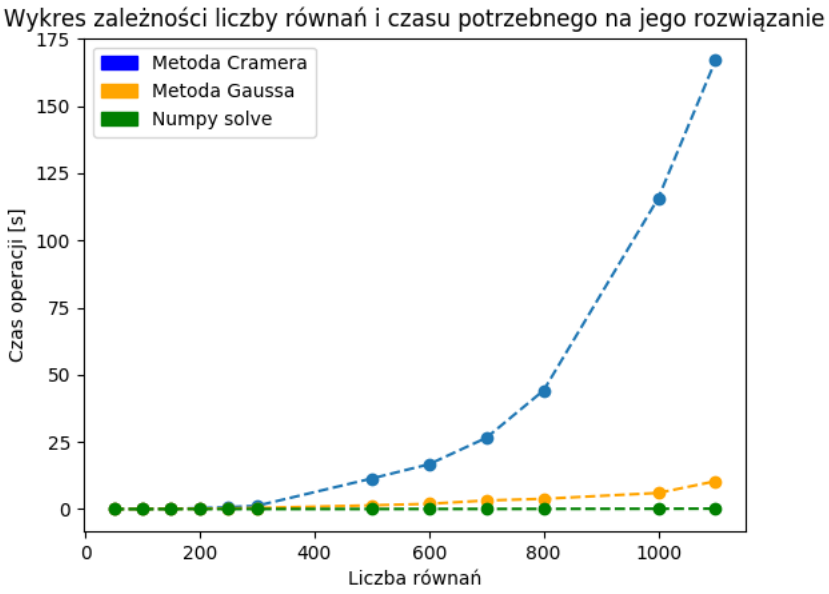


(b) M. Gaussa-Jordana



(c) Numpy solve

Zbiórczy wykres porównujący wszystkie trzy metody:



Rysunek 1: Zbiórczy wykres zależności liczby równań i czasu na jego rozwiązanie

Warto dodać, że na tle metody Gaussa oraz metody Cramera funkcja wbudowana *numpy.linalg.solve*() nie wzrasta w miarę zwiększania układu równań.

W przypadku metody wyznaczników obserwujemy bardzo dużą złożoność obliczeniową. Stosowanie wzorów Cramera nakłada konieczność obliczenia $n+1$ wyznaczników stopnia n. Bazując na klasycznej metodzie obliczania wyznacznika, należy wykonać $n!$ dodawań oraz $n \times n!$ mnożeń. Łącznie daje nam to $(n+1)!$ działań arytmetycznych zmiennoprzecinkowych. Aby otrzymać rozwiązanie układu równań tą metodą, należy obliczyć $n+1$ wyznaczników stopnia n oraz wykonać n dzieleń. Całkowita liczba działań arytmetycznych jest zatem równa $(n+1) \times (n+1)! + n$. Co z kolei powoduje, że metoda Cramera ma klasę złożoności obliczeniowej równą $O(n^4)$. Klasa ta pokrywa się z krzywą uzyskaną na wykresie. Na tej podstawie możemy wyciągnąć wniosek, że metoda wyznacznikowa nie jest wskazana do numerycznego rozwiązywania układów równań liniowych.

W ogólnym rozrachunku, bardzo dobrą metodą w porównaniu do tej poprzedniej, okazuje się metoda eliminacji Gaussa. W jej przypadku operacja wyznaczania niewiadomych zawiera trzy zagnieżdżone w sobie pętle, zatem ma klasę złożoności obliczeniowej równą $O(n^3)$. Uzyskana krzywa dla tej metody nie do końca pokrywa się ze złożonością obliczeniową tego algorytmu. Jednak zestawiając ze sobą te dwie metody i porównując ich czas potrzebny na rozwiązanie układu równań, można stwierdzić, że metoda eliminacji jest lepsza do numerycznego rozwiązywania układów równań liniowych.

Warto jednak zaznaczyć, że podawanie złożoności czasowej w jednostkach

czasu jest niewygodne, ponieważ wynik zależy od szybkości komputera, na którym dokonano pomiarów - trudno takie wyniki odnieść do innych komputerów, szczególnie wyposażonych w inne procesory, gdzie czas wykonania podobnych operacji może znacznie się różnić. Dlatego częściej złożoność czasową wyrażamy w liczbie operacji dominujących, gdyż każdy komputer, bez względu na swoje własności, operacje te musi wykonać. Dzięki temu wynik może zostać uniezależniony od faktycznej szybkości komputerów.

8 Aproksymacja funkcji potęgowej za pomocą funkcji Optimize Curve Fit z modułu SciPy

W celu dopasowania funkcji potęgowej do danych sprawdzających czas działania naszych funkcji na odpowiednio dużych rozmiarach macierzy posłużyliśmy się funkcją $optimize.curve\_fit$ z modułu $SciPy$, która pozwala na odpowiednie dobranie parametrów modelu. Funkcja jako argumenty przyjmuje postać funkcji, zmienną niezależną oraz zmienną zależną. Następnie wykorzystaliśmy te parametry w celu aproksymacji otrzymanych punktów na wykresie.

8.1 Postać funkcji

Postać funkcji potęgowej prezentuje się następująco:

$$y = x^a * b$$

Gdzie:

- y - zmienna zależna
- x - zmienna niezależna
- a, b - parametry modelu

8.2 Wyznaczanie parametrów oraz wizualizacja

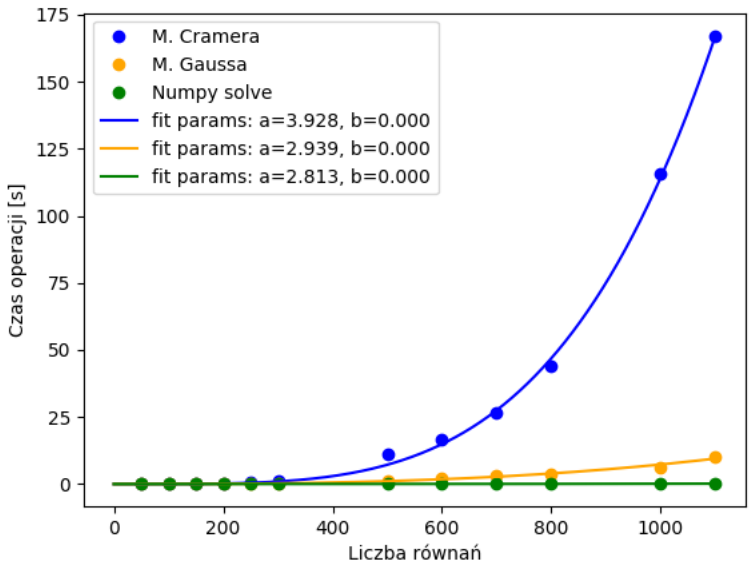
Kod programu:

```
#Metoda Cramera
plt.plot(l_rownan,czas_cramer, 'bo', label='M. Cramera', color='blue')
popt1, pcov1 = curve_fit(func, l_rownan, czas_cramer)
#[3.99063767e+00 1.22019764e-10]

#Metoda Gaussa
plt.plot(l_rownan,czas_gauss, 'bo', label='M. Gaussa', color='orange')
popt2,pcov2 = curve_fit(func,l_rownan,czas_gauss)
#[2.93937347e+00 2.14140260e-09]

#Metoda numpy
```

```
plt.plot(l_rownan,czas_numpy, 'bo', label='Numpy solve', color='green')
popt3,pcov3 = curve_fit(func,l_rownan,czas_numpy)
#[2.81300951e+00 3.87866680e-10]

xfit = numpy.arange(0, 1100, 0.01)

#M.Cramera
plt.plot(xfit, func(xfit, *popt1),
'blue', label='fit params: a=%5.3f, b=%5.3f'
% tuple(popt1))

#M.Gaussa
plt.plot(xfit, func(xfit, *popt2),
'orange', label='fit params: a=%5.3f, b=%5.3f'
% tuple(popt2))

#Numpy Solve
plt.plot(xfit, func(xfit, *popt3),
'green', label='f
```

Odpowiednio w zmiennych *pop*1, *popt*2, *popt*3 przechowywane są wyliczone parametry a oraz b dla każdej z metod.

Dopasowanie oraz aproksymacja punktów funkcji potęgowej dla każdej z metod:



Rysunek 2: Aproksymacja funkcji potęgowej wzdłuż punktów

8.3 Wnioski

W poszukiwaniu rozwiązania przyjmuje się pewną znaną funkcję, a następnie dopasowuje parametry w taki sposób, aby wynik jak najbardziej „pasował" do zadanych punktów, które pochodzą z pewnych pomiarów i już ze swej natury są obarczone błędami.

W naszym przypadku, dla układu parametrów:

$$a = 3.92756930, b = 1.86485975e^{-10}$$

$$a = 2.93937347, b = 2.14140260e^{-09}$$

$$a = 2.81300951, b = 3.87866680e^{-10}$$

oraz przy zadanych ogólnych postaciach naszych funkcji zależności czasu od licz- by równań, średni błąd kwadratowy dopasowania przebiegu funkcji do posiada- nych danych jest możliwie najmniejszy. Na podstawie powyższej wizualizacji, możemy stwierdzić, że dopasowanie to jest bardzo dobre, ponieważ wszystkie punkty leżą na wyznaczonych liniach lub są bardzo do nich zbliżone. Ponadto wyznaczone wartości współczynników a dla każdego z algorytmów zgadzają się z ich złożonością obliczeniową.