MOwNiT laboratorium 4. - Sprawozdanie

Michał Szczurek
Informatya, WIEiT

Grupa poniedziałek 12:50

1 Wstęp

Celem laboratorium było przetestowanie różnych metod aproksymacji średniokwadratowej, czyli przybliżenia funkcji mającego na celu minimalizację kwadratu różnic funkcji aproksymującej i apryksomowanej z uwzględnieniem wag.

W tym celu aproksymowano funkcję $\frac{1}{1+25x^2}$ na przedziale $[-1, 1]$ wielomianem 10 stopnia korzystając z 6 różnych metod. W laboratorium użyłem biblioteki functools, która pozwala na użycie dekoratora funkcji @lru_cache(maxsize=None). Dekorator ten dodaje automatyczne chache'owanie wyników funkcji co jest szczególnie przydatne podczas wyliczania wartości zadanych rekurencyjnie i przyspiesza działanie programu (szczególnie zauważalna była poprawa działania aproksymacji dla aproksymacji wielomianami Czebyszewa z użyciem wzoru Clenshawa.

2 Aproksymacja średniokwadratowa punktowa

Aproksymacji dokonano na dwa sposoby jako punkty aproksymacji wybierając 21 punktów równomiernie rozmieszczonych w przedziale [-1,1].

2.1 Metoda wykorzystująca zmienne pomocnicze S_k i T_k

Metoda polega na wprowadzeniu dwóch zmiennych pomocniczych:

$$S_k = \sum_{i=0}^{n} {x_i}^k, k = 0, 1...2m$$

$$T_k = \sum_{i=0}^{n} {x_i}^k \cdot y_i, k = 0, 1...m$$

Gdzie n+1 to liczba punktów aproksymacji, a m - – stopień wielomianu aproksymującego (m<n), a y_i to wartość funkcji aproksymowanej w punkcie x_i. Dzięki wprowadzeniu zmiennych pomocniczych, można obliczyć współczynniki

wielomianu aproksymującego $Q_m(x) = \sum_{i=0}^{m} a_i x^i$ korzystając z następującego układu równań:
$a_0 \cdot S_0 + a_1 \cdot S_1 + ... + a_m \cdot S_m = T_0$
$a_0 \cdot S_1 + a_1 \cdot S_2 + ... + a_m \cdot S_{m+1} = T_1$
$a_0 \cdot S_2 + a_1 \cdot S_3 + ... + a_m \cdot S_{m+2} = T_2$
$\vdots$
$a_0 \cdot S_m + a_1 \cdot S_{m+1} + ... + a_m \cdot S_{2m} = T_m$

Realizuje to poniższa funkcja:

```
def approx_1a(func, k, deg):

    def S_k(k, x_vals):
        return np.sum(x_vals ** k)

    def T_k(k, x_vals, y_vals):
        return np.sum(y_vals * x_vals ** k)


    x_vals = np.linspace(-1,1,k)
    y_vals = np.array([func.replace(x,v) for v in x_vals])

    S_vector = [S_k(i, x_vals) for i in range(2*deg + 1)]
    T_vector = np.array([T_k(i, x_vals, y_vals) for i in range(deg + 1)], dtype='float')\
    \.reshape(-1,1)

    matrix = np.empty((deg+1, deg+1), dtype='float')

    for i in range(deg+1):
        matrix[i] = S_vector[i:i+deg+1]

    a_vector = np.linalg.solve(matrix, T_vector)
    a_vector = np.ndarray.flatten(np.flip(a_vector))
    poly = Poly.from_list(a_vector, gens=x).as_expr()

    return poly, lambdify(x, poly)
```

Funkcja jako argumenty przyjmuje funkcje aproksymowaną, liczbę punktów aproksymacji i stopień wielomianu aproksymującego, a zwraca wielomian aproksymujący i funkcję realizująca ten wielomian, przydatną przy rysowaniu wykresu.
Poniżej znajduje się wykres prezentujący wynik aproksymacji:

Wielomian aproksymujący (z dokładnością do 3 cyfr znaczących) ma postać:

$$Q(x) = -36.5x^{10} + 6.23 \cdot 10^{-10}x^9 + 101x^8 - 1.39 \cdot 10^{-9}x^7 - 102x^6 + 1.02 \cdot 10^{-9}x^5 \\ +47.3x^4 - 2.74 \cdot 10^{-10}x^3 - 9.99x^2 + 1.97 \cdot 10^{-11}x + 0.901$$

2.2 Metoda niewykorzystująca zmiennych pomocniczych

Metoda polega na obliczeniu współczynników wielomianu $Q_n(x)$ korzystając ze wzoru:

$$A = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & \cdots & x_0^m \\ 1 & x_1 & x_1^2 & \cdots & \cdots & x_1^m \\ 1 & x_2 & x_2^2 & \cdots & \cdots & x_2^m \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & x_n & x_n^2 & \cdots & \cdots & x_n^m \end{bmatrix}, c = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \cdots \\ \cdots \\ a_m \end{bmatrix} y = \begin{bmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ \cdots \\ \cdots \\ f(x_n) \end{bmatrix}$$

$$A^T Ac = A^T y$$

Dla aproksymacji przy użyciu n+1 punktów wielomianem stopnia m. Poniżej znajduje się kod funkcji:

```
def approx_1b(func, k, deg):
x_vals = np.linspace(-1,1,k)
y_vals = np.array([func.replace(x,v) for v in x_vals], dtype='float')

matrix = np.empty((k, deg+1), dtype='float')
for i in range(deg+1):
    matrix[:, i] = x_vals**i
```

```
a_vector = np.linalg.solve(matrix.T @ matrix, matrix.T @ y_vals)
a_vector = np.ndarray.flatten(np.flip(a_vector))
poly = Poly.from_list(a_vector, gens=x).as_expr()

return poly, lambdify(x, poly)
```

Wykres apromsymacji wygląda następująco:

A wielomian (z dokładnością do 3 miejsc znaczących ma postać):

$$Q(x) = -36.5x^{10} + 1.46 \cdot 10^{-10}x^9 + 101x^8 - 3.30 \cdot 10^{-10}x^7 - 102x^6 - 2.49 \cdot 10^{-10}x^5 \\ +47.3x^4 + 6.96 \cdot 10^{-11}x^3 - 9.99x^2 - 5.32 \cdot 10^{-12}x + 0.901$$

2.3 Podsumowanie

Wielomian uzyskany w obu metodach jest tym samym wielomianem, jednak różniącym się ze względu na błędy numeryczne wynikające z różnych sposobów obliczania wyniku. Różnice widoczne są dla małych współczynników współczynników ($< 10^8$). Pozostałe współczynniki nie różnią się dla początkowych cyfr znaczących.

3 Aproksymacja ciągła

Aby obliczyć wielomian aproksymujący n-tego stopnia należy rozwiązać układ równań:

$$\begin{bmatrix} < \phi_0, \phi_0 > & \cdots & < \phi_0, \phi_n > \\ < \phi_1, \phi_0 > & \cdots & < \phi_1, \phi_n > \\ \cdots & \cdots & \cdots \\ < \phi_m, \phi_0 > & \cdots & < \phi_m, \phi_m > \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \cdots \\ c_m \end{bmatrix} = \begin{bmatrix} < f, \phi_0 > \\ < f, \phi_1 > \\ \cdots \\ < f, \phi_m > \end{bmatrix}$$

gdzie

$$< f, g >= \int_{-1}^{1} w(x) f(x) g(x) dx$$

Funkcja aproksymująca ma wówczas postać:

$$p = \sum_{k=0}^{m} c_k \phi_k$$

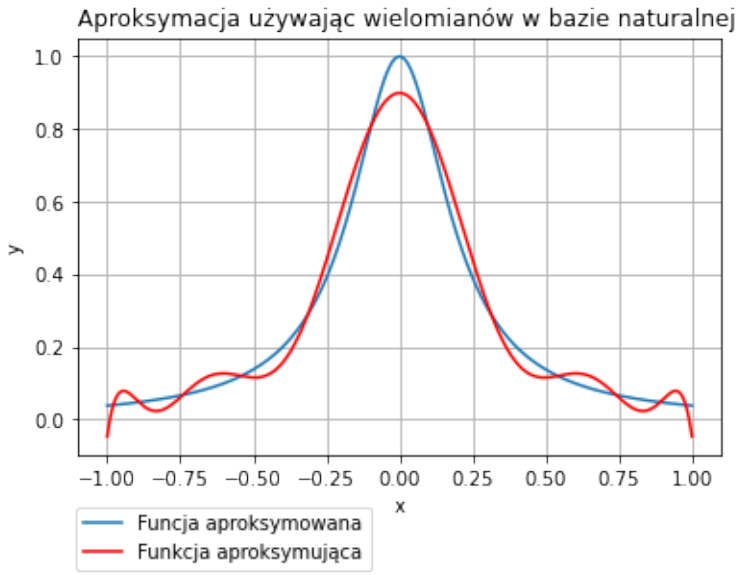
3.1 Aproksymacja przy użyciu wielomianów w bazie naturalnej i macierzy Hilberta

Macierz Hilberta to macierz, w której element na pozycji (i, j) wynosi $\frac{1}{i+j+1}$. Macierz ta zawiera więc na pozycji (i,j) wartości równe $\int_0^1 x^i x^j dx$. Macierz można zmodyfikować mnożąc wartości przez 2 tam, gdzie i+j jest parzyste i zerując w pozostałych. Wówczas macierz odpowiada wartością $\int_{-1}^1 x^i x^j dx$. Jeśli za ϕ_k przyjmiemy x^k, to macierz będzie macierzą z powyższego układu równań. W oparciu o tą obserwację napisałem następującą funkję:

```
def approx_2(func, deg):
matrix = np.empty((deg+1, deg+1), dtype='float')
for i in range (deg+1):
    for j in range(deg+1):
        if (i+j)%2 == 0:
            matrix[i][j] = 2 / (i + j + 1)
        else:
            matrix[i][j] = 0

def integral(f, k):
    # <f, (k)>
    return quad(lambdify(x, f * x**k), -1, 1)[0]

res_vector = np.array([integral(func, i) for i in range(deg+1)], dtype='float')/
/.reshape(-1,1)
c_vector = np.linalg.solve(matrix, res_vector)
```

```
    c_vector = np.ndarray.flatten(c_vector)
    res = 0
    for i in range(deg + 1):
        res += c_vector[i] * x**i

    return simplify(res), lambdify(x, res)
```

Poniżej znajduje się wykres wykonanej aproksymacji:



A wielomian (z dokładnością do 3 miejsc znaczących ma postać):

$$Q(x) = -36.0x^{10} + 98.5x^8 - 100x^6 + 46.5x^4 - 9.89x^2 + 0.899$$

3.2 Aproksymacja przy użyciu wielomianów Legendre'a

Funkcję można aproksymować przyjmując za ϕ_k n-ty wielomian Legendre'a czyli wielomian, który można obliczyć wzorem rekurencyjnym:
$P_{n+1}(x) = \frac{2n+1}{n+1}xP_n(x) - \frac{n}{n+1}xP_{n-1}(x)$ (n=2,2,...)
$P_0 = 1$
$P_1 = x$
Wielomiany Legendre'a są ortogonalne na przedziale $[-1, 1]$. Dzięki temu człon $< \phi_i, \phi_j >= 0$ dla $i \neq j$ oraz $< \phi_i, \phi_i >= \frac{2}{2i+1}$. Znacząco upraszcza to układ równań jako, że macierz staje się przekątniowa, wobec czego $c_k = \frac{<f,\phi_k>(2k+1)}{2}$.
Funkcja wagowa w tym przypadku wynosi 1. Korzystając z powyższych faktów napisałem funkcję:

```
def approx_3(func, deg):

@lru_cache(maxsize=None)
def Legendre(n):
    if n == 0:
        return 1

    if n == 1:
        return x

    return (2*n - 1)/n * x * Legendre(n-1) - (n - 1)/n * Legendre(n-2)

def c_k(k, f):
    return (2*k+1)/2 * quad(lambdify(x, f * Legendre(k)), -1, 1)[0]

res = 0
for i in range(deg + 1):
    res += c_k(i, func) *  Legendre(i)

return simplify(res), lambdify(x, res)
```

Poniżej znajduje się wykres przedstawiający uzyskaną aproksymację:

Wielomian aproksymujący ma wówczas postać (z dokładnością do 3 miejsc znaczących ma postać):

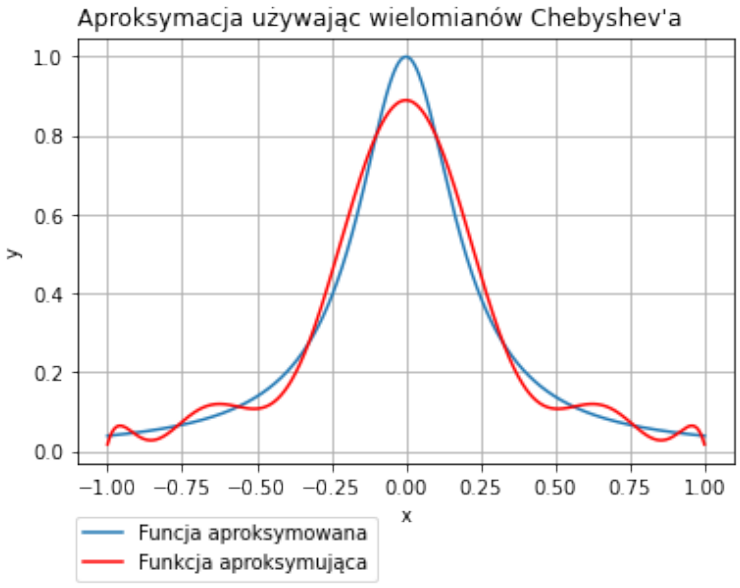
$$Q(x) = -35.9x^{10} + 98.5x^8 - 100x^6 + 46.5x^4 - 9.89x^2 + 0.899$$

3.3 Aproksymacja przy użyciu wielomianów Czebyszewa'a

Możemy postąpić podobnie jak w poprzednim punkcie przyjmując za ϕ_k n-ty wielomian Czebyszewa. Wielomiany te również można obliczyć przy pomocy wzoru rekurencyjnego:
$T_{n+1}(x) = 2xT_n(x) - Tn - 1(x)$ (n=2,3,..n)
$T_0 = 1$
$T_1 = x$
Wielomiany Czebyszewa, podobnie jak wielomiany Legendre'a są ortogonalne na przedziale [1-,1], przy czym $< \phi_i, \phi_i >= \frac{\pi}{2}$, gdy $i \neq 0$ oraz $< \phi_0, \phi_0 >= \pi$. Dodatkowo należy uwzględnić funkcję wagową wynoszącą $w(x) = \frac{1}{\sqrt{1-x^2}}$
Korzystając z powyższych zależności można napisać funkcję:

```
def approx_4(func, deg):

@lru_cache(maxsize=None)
def Chebyshev(n):
    if n == 0:
        return 1

    if n == 1:
        return x

    return 2 * x * Chebyshev(n-1) -  Chebyshev(n-2)

def c_k(k, f):
    factor = math.pi
    if k!=0:
        factor /= 2

    w = 1/sqrt(1 - x**2)

    return 1/factor * quad(lambdify(x, w * f * Chebyshev(k)), -1, 1)[0]


res = 0
for i in range(deg + 1):
    res += c_k(i, func) *  Chebyshev(i)
```

Pozwala ona na otrzymanie następującej aproksymacji:



Wielomian aproksymujący (z dokładnością do 3 miejsc znaczących ma postać) wygląda następująco:

$$Q(x) = -27.5x^{10} + 79.1x^8 - 84.5x^6 + 41.4x^4 - 9.34x^2 + 0.899$$

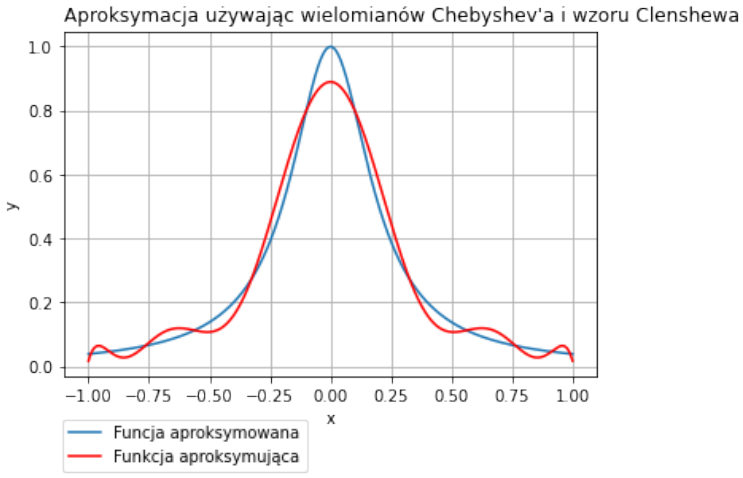
3.4 Aproksymacja przy użyciu wielomianów Czebyszewa'a i wzoru Clenshawa

Wzór Clenshawa pozwala na zmniejszenie liczby mnożeń i dodawań podczas obliczania wielomianu aproksymacyjnego przy pomocy wielomianów Czebyszewa. Wzór ten umożliwia alternatywną metodę obliczenia wielomianu aproksymacyjnego po wyliczeniu współczynników c_k. Metoda sprowadza się do obliczenia zmiennych pomocniczych y_1 i y_2 przy użyciu formuły rekurencyjnej:
$y_{m+2} = y_{m+1} = 0$
$y_k(x) = 2xy_{k+1} - y_{k+2} + c_k$
Wówczas wielomian interpolacyjny ma postać: $p = -y_2 + xy_1 + c_0$

Poniżej znajduje się kod funkcji realizującej opisaną metodę:

```
def approx_4b(func, deg):

def Chebyshev(n):
    if n == 0:
        return 1

    if n == 1:
        return x

    return 2 * x * Chebyshev(n-1) -  Chebyshev(n-2)

def c_k(k, f):
    factor = math.pi
    if k!=0:
        factor /= 2

    w = 1/sqrt(1 - x**2)

    return 1/factor * quad(lambdify(x, w * f * Chebyshev(k)), -1, 1)[0]

@lru_cache(maxsize=None)
def y_k(k, f):
    if k == deg + 1 or k == deg+2:
        return 0
    return 2*x*y_k(k+1, f) - y_k(k+2, f) + c_k(k, f)


res = -1* y_k(2, func) + y_k(1, func) * x + c_k(0, func)

return simplify(res), lambdify(x, res)
```

Dzięki metodzie uzyskano następującą aproksymację:



Wielomian aproksymujący (z dokładnością do 3 miejsc znaczących ma postać) ma postać:

$$Q(x) = -27.5x^{10} + 79.1x^8 - 84.5x^6 + 41.4x^4 - 9.34x^2 + 0.899$$

3.5 Wnioski

- Wszystkie otrzymane aproksymacje były dobrym przybliżeniem badanej funkcji (porównując to na przykład do interpolacji z efektem Rungego).
- Wykorzystanie zmiennych pomocniczych do obliczenia aproksymacji punktowej nie wpłynęło znacząco na wynik.
- Aproksymacja wielomianami w bazie naturalnej i wielomianami Legendre'a dała praktycznie taki sam rezultat.
- Aproksymacja punktowa miała nieco odmienny rezultat od aproksymacji wielomianami Legendre'a i wielomianami w bazie naturalnej, co widać na końcach przedziału.
- Interpolacja wielomianami Czebyszewa poskutkowała najbardziej odmiennym wzorem funkcji aproksymującej. Również wykres był nieco inny, jednak wciąż był on zbliżony do wykresów uzyskanych dla pozostałych metod.
- Użycie wzoru Clenshawa nie miało wpływu na wynik aproksymacji wielomianami Czebyszewa.