MATEMATYKA

Metody Numeryczne

Projekt Indywidualny C, D i E

Wykonał:

Oleg Łyżwiński 305158

Warszawa 2024

Poniżej przedstawiono rozwiązanie zadań C, D i E projektu indywidualnego z przedmiotu Metody Numeryczne. Do rozwiązań wykorzystano język Python. Zastosowano następujące biblioteki: numpy, matplotlib, math, scipy oraz sklearn.

Zadanie C 8

Korzystając z biblioteki sympy wyznaczono wartości pierwszej oraz drugiej pochodnej:

```
# Wyznaczenie f'()
def f_prime_x():
    x = Symbol('x')
    y = x - exp(-x)
    y_prime = y.diff(x)
    y_prime = lambdify(x, y_prime, 'numpy')
    return y_prime

# Wyznaczenie f''()
def f_prime_prime_x():
    x = Symbol('x')
    y = x - exp(-x)
    y_prime = y.diff(x)
    y_prime_prime = y_prime.diff(x)
    y_prime_prime = lambdify(x, y_prime_prime, 'numpy')
    return y_prime_prime
```

Metoda siecznych:

```
def secant_method(x0, x1, solution, precizion):
    iteration = 0
    tab_error = []
    tab_fx = []
    while True:
        # Wyznaczenie punktu przecięcia siecznej z osią OX
        x_new = x1 - (f(x1) * (x1 - x0)) / (f(x1) - f(x0))

        tab_fx.append(abs(f(x_new)))

        error = abs(x_new - solution)
        if error != 0.0:
            tab_error.append(error)

        # Sprawdzenie dokładności przybliżenia
        if abs(f(x_new)) < precizion:
            return x_new, tab_error, tab_fx

        if f(x_new)*f(x0) > 0:
            x0 = x_new
        else:
            x1 = x_new
        iteration += 1
```

Metoda Newton’a:

```
def newton_method(x0, x1, solution, precizion):
    # Jako pierwsze przybliżenie przyjmujemy koniec w którym
    # f(x) i f''(x) mają ten sam znak
    if f(x0)*f_prime_prime(x0)>=0:
        x0 = x0
    elif f(x1)*f_prime_prime(x1)>=0:
        x0 = x1
    else:
        print(":(")

    iteration = 0
    tab_error = []
    tab_fx = []
    while True:
        tab_fx.append(abs(f(x0)))
        # Wyznaczenie wartości błędu
        error = abs(abs(x0)-solution)
        if error != 0.0:
            tab_error.append(error)

        # Sprawdzenia warunku wyjścia
        if abs(f(x0)) < precizion or x1 < x0:
            return x0, tab_error, tab_fx

        x0 = x0 - f(x0) / f_prime(x0)

        iteration += 1
```

Wyznaczenie rozwiązania równania metodą Brent’a:

```
solution = root_scalar(f, bracket=[0, 2], method='brentq')
```

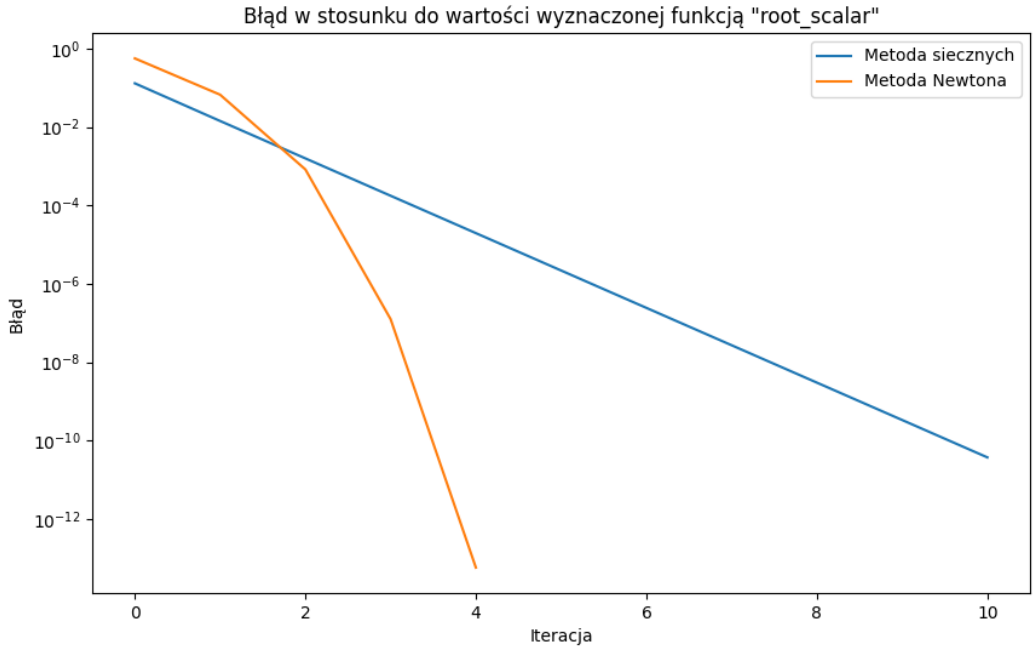
```
# Wartość funkcji w punkcie x
def f(x):
    return x - math.exp(-x)
```

Uzyskano następujące wyniki dla zadanej dokładności 10^{-10}:

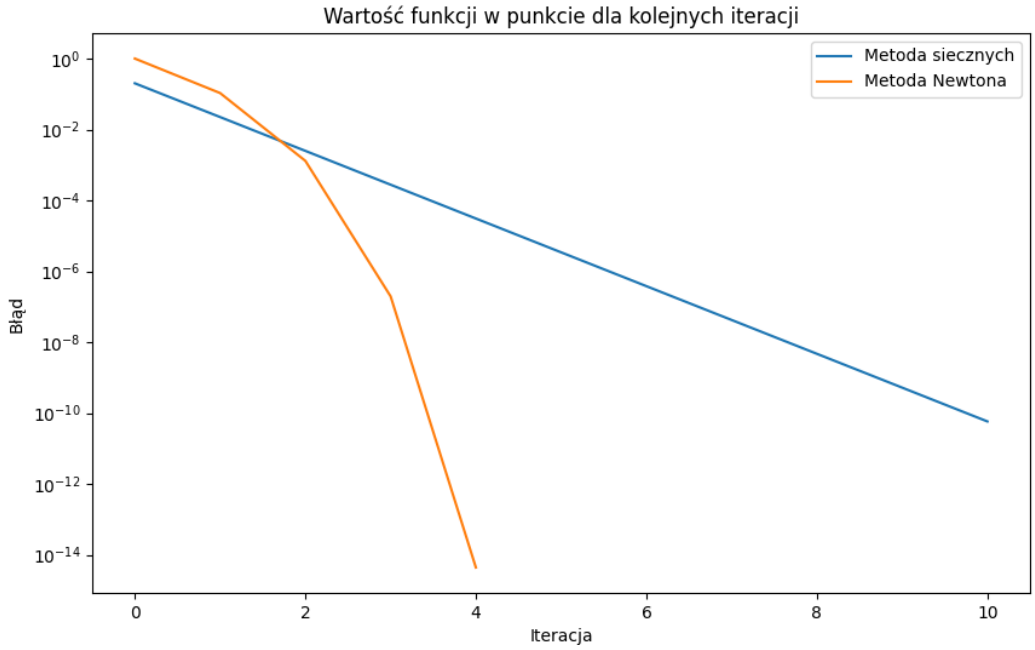
```
Rozwiązanie równania x = exp(-x): 0.5671432904098384
Rozwiązanie metodą siecznych:     0.5671432904470005
Rozwiązanie metodą Newtona:       0.5671432904097811
```

Na podstawie uzyskanych wyników: wynik uzyskany metodą siecznych osiągnął dokładność 10^{-10}, a metodą Newtona 10^{-14} w stosunku do rozwiązania metodą Brent’a zawartą w funkcji root_scalar.

Wartość błędu wyznaczano jako różnicę wyznaczonej wartości x w danej iteracji, a wartości wzorcowej x wyznaczonej metodą Brent'a:



Wykreślono również charakterystykę f(x) dla x wyznaczonych w kolejnych iteracjach pętli poszczególnych algorytmów:



Na podstawie powyższych charakterystyk możemy zaobserwować, że metoda Newtona jest znacząco szybsza od metody siecznych, ponieważ już po 4 iteracjach osiągnęła dokładność na poziomie 10^{-14}. Poprawa precyzji w metodzie siecznych ma charakter liniowy i do osiągnięcia pożądanej dokładności 10^{-10} potrzebuje 10 iteracji. Dokładność w przypadku metody siecznych z każdą iteracją wzrasta o 10^{-1}.

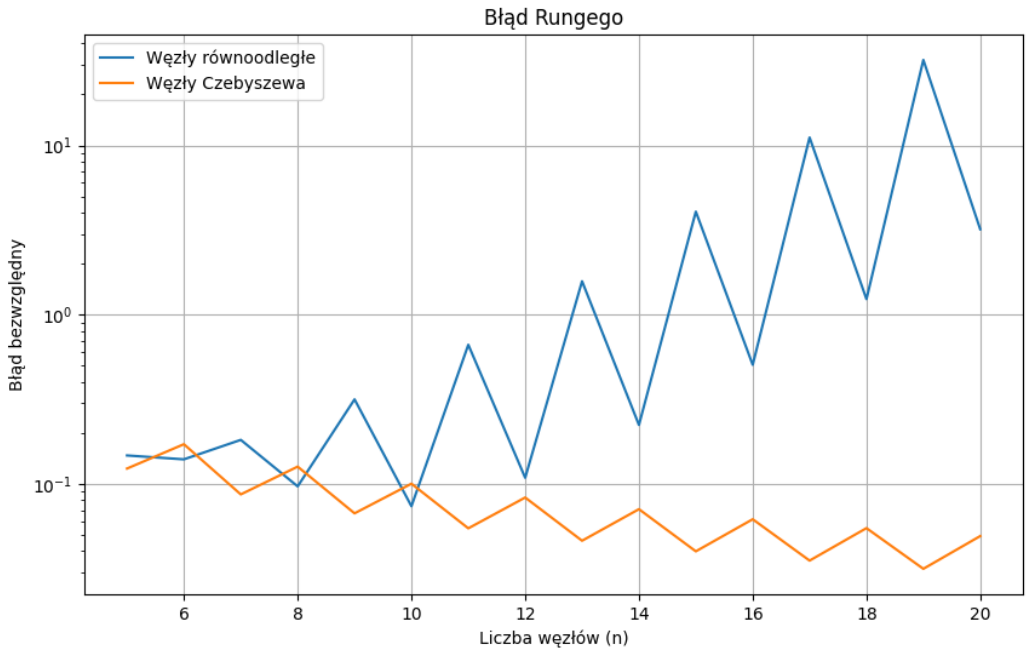
```
[0.13101869283952428, 0.014337042067054329, 0.0015915862887008192, 0.00017695859628463761, 1.9678294498559623e-05, 2.1883233745079167e-06,
 2.433528206724489e-07, 2.7062049268167243e-08, 3.009392068875627e-09, 3.346116717040104e-10, 3.716216223637048e-11, 4.084177440688563e-12
, 4.057865155004947e-13]
```

Zad D 9

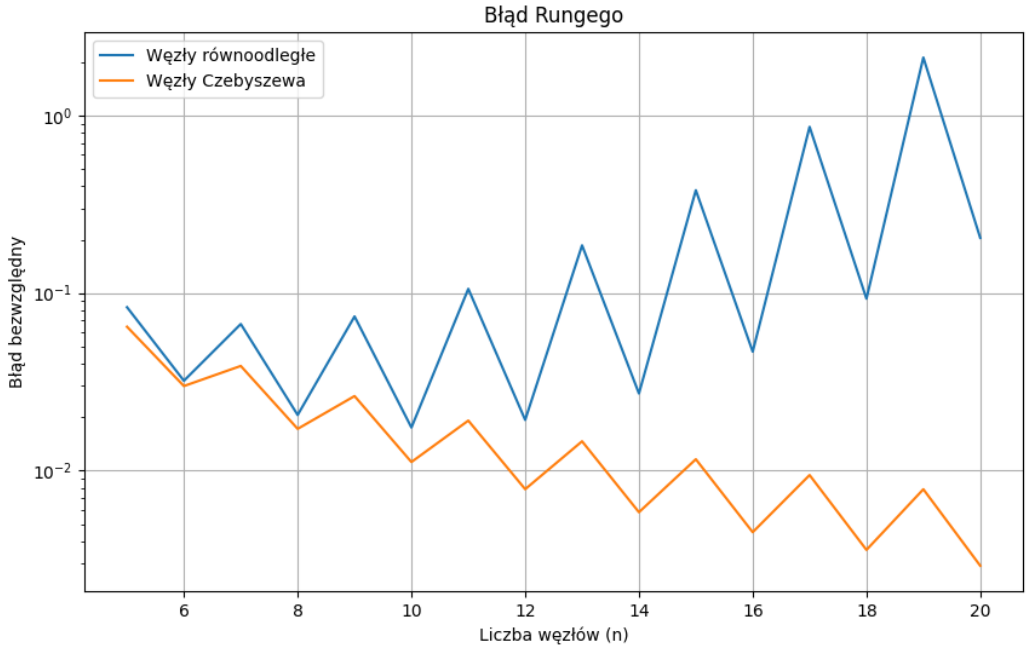
```python
def f(x):
    return np.abs(x)

def runge_phenomenon(n_values):
    errors_equidistant = []
    errors_chebyshev = []

    for n in n_values:
        # Węzły równoodległe
        x_equidistant = np.linspace(-1, 1, n)
        y_equidistant = f(x_equidistant)

        # Interpolacja wielomianowa
        interpol_poly_e = np.polyfit(x_equidistant, y_equidistant, n - 1)
        x_values_e = np.linspace(-1, 1, 1000)
        y_values_e = np.polyval(interpol_poly_e, x_values_e)
        true_values_e = f(x_values_e)
        error_e = np.max(np.abs(y_values_e - true_values_e))
        errors_equidistant.append(error_e)

        # Węzły Czebyszewa
        x_chebyshev = np.cos(np.pi * (2 * np.arange(1, n + 1) - 1) / (2 * n))
        y_chebyshev = f(x_chebyshev)
        # Interpolacja wielomianowa
        interpol_poly = np.polyfit(x_chebyshev, y_chebyshev, n - 1)
        x_values = np.linspace(-1, 1, 1000)
        y_values = np.polyval(interpol_poly, x_values)
        true_values = f(x_values)
        error_c = np.max(np.abs(y_values - true_values))
        errors_chebyshev.append(error_c)

        '''# Rysowanie wykresu···

    return errors_equidistant, errors_chebyshev

n_values = np.arange(5, 23, 1)

errors_equidistant, errors_chebyshev = runge_phenomenon(n_values)
```

Uzyskano następujące charakterystyki błędu bezwzględnego:

- Dla maksymalnego błędu bezwzględnego:

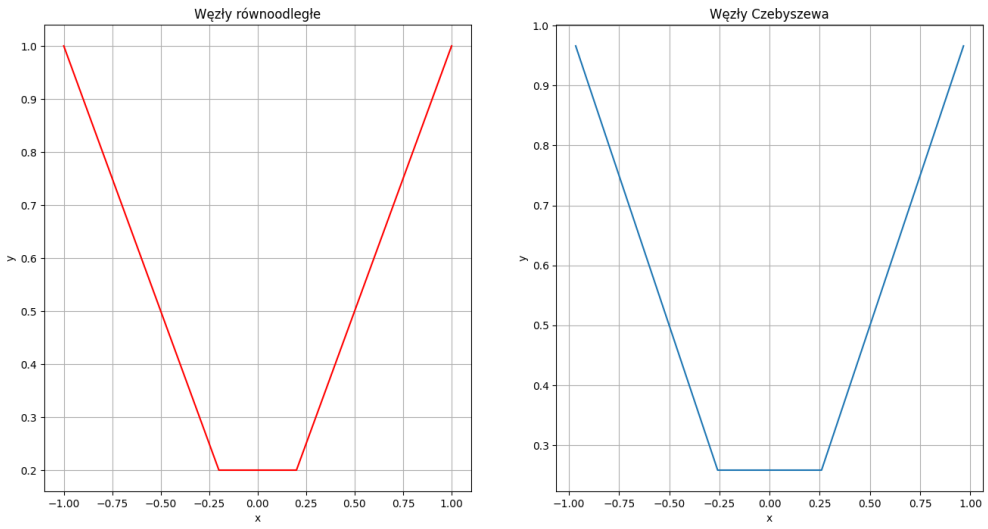


- Dla średniego błędu bezwzględnego:

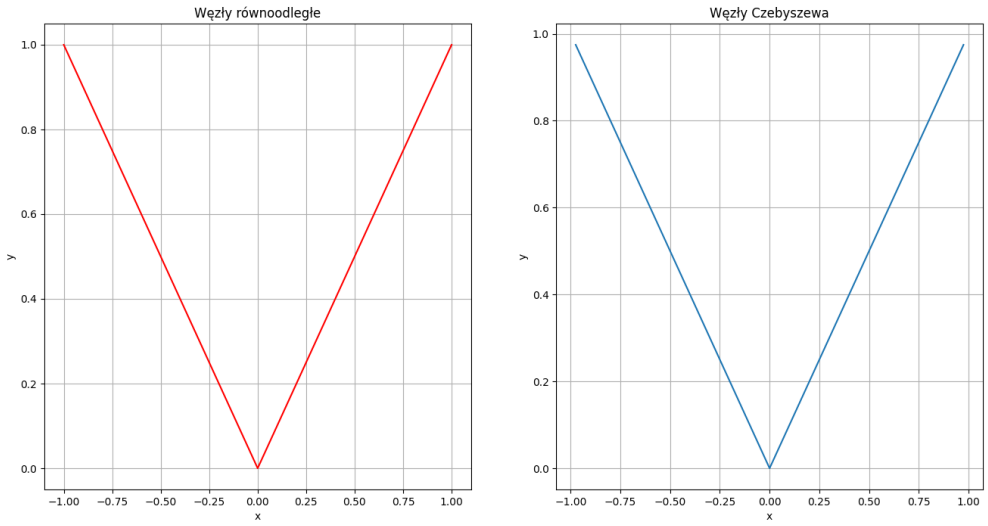


Na powyższej charakterystyce możemy zaobserwować, że błąd skacze w zależności od tego czy liczba węzłów była parzysta czy nieparzysta. Wynika to z rozkładu punktów, które w przypadku parzystej liczby punktów, nie zawierają punktu (0, 0). Poniżej przedstawiono funkcje otrzymane przez interpolację wielomianową:

- Dla parzystej liczby węzłów (n=6):



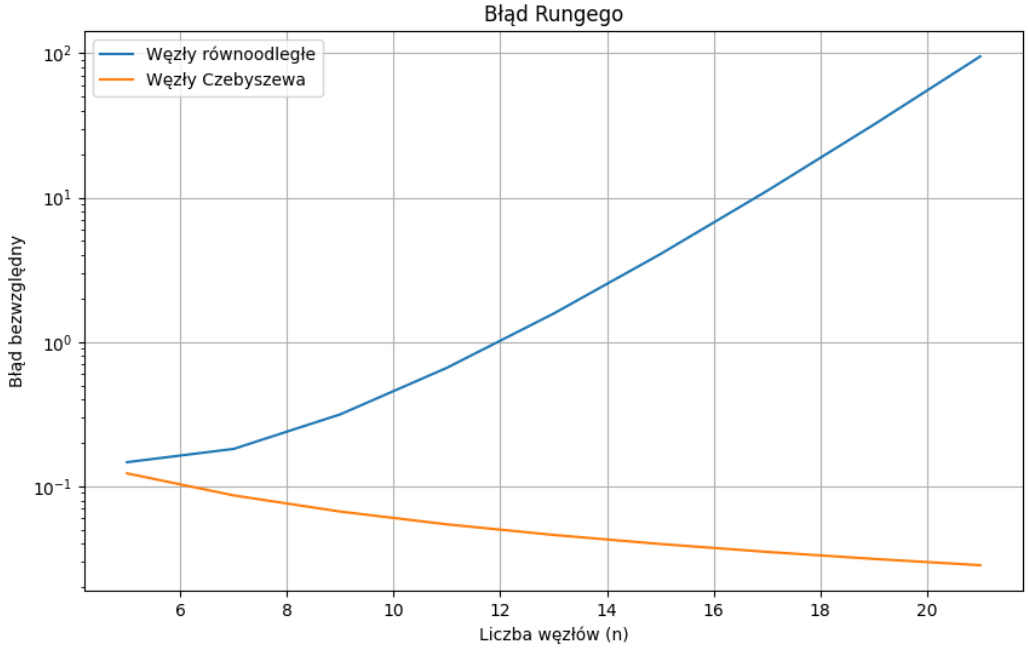
- Dla nieparzystej liczby węzłów (n=7):



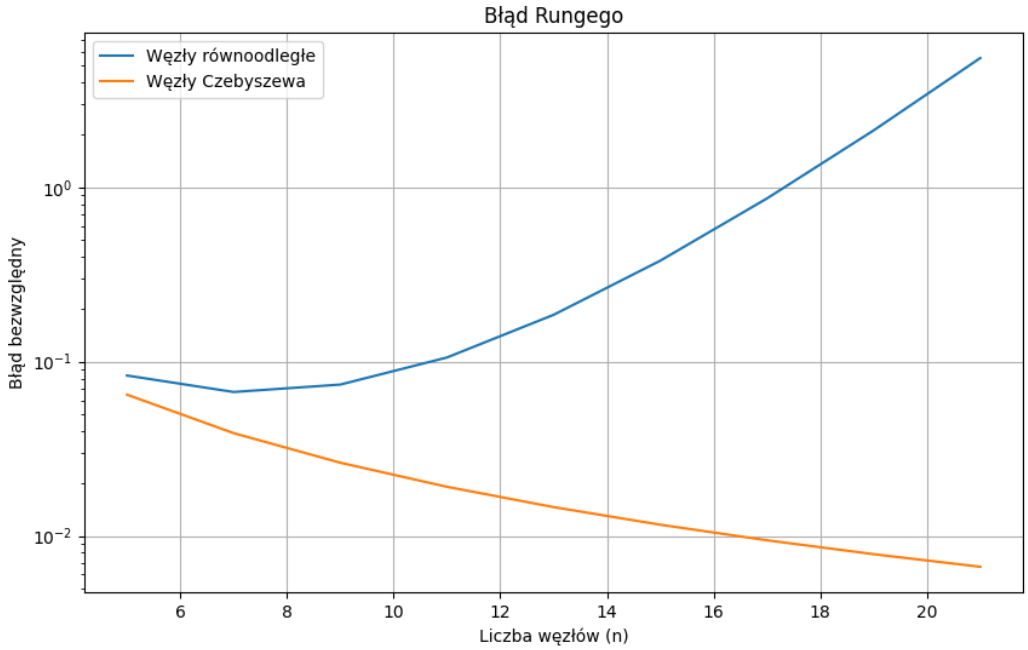
Na powyższych charakterystykach można zaobserwować, że dla nieparzystej liczby węzłów uwzględniany jest punkt (0, 0), co sprawia, że błąd dla nieparzystej liczby węzłów jest mniejszy. Możemy również zaobserwować, że maksymalny błąd bezwzględny dla parzystej liczby węzłów (dla n ∈ {6, 8 i 10}) w przypadku węzłów Czebyszewa daje większy błąd maksymalny większy niż dla węzłów równorozłożonych. Wynika to z charakteru rozkładu Czebyszewa, w którym większość węzłów jest na krańcach przedziału, co za tym idzie są dalej od punktu (0, 0) niż w przypadku węzłów równorozłożonych. Natomiast średni błąd względny dla węzłów Czebyszewa zawsze przyjmuje mniejszą wartość, niż dla węzłów równoodległych.

Wykteślo więc charakterystykę wyłącznie dla niepażystej ilości węzłów:

- Dla maksymalnego błędu bezwzględnego:



- Dla średniego błędu bezwzględnego:

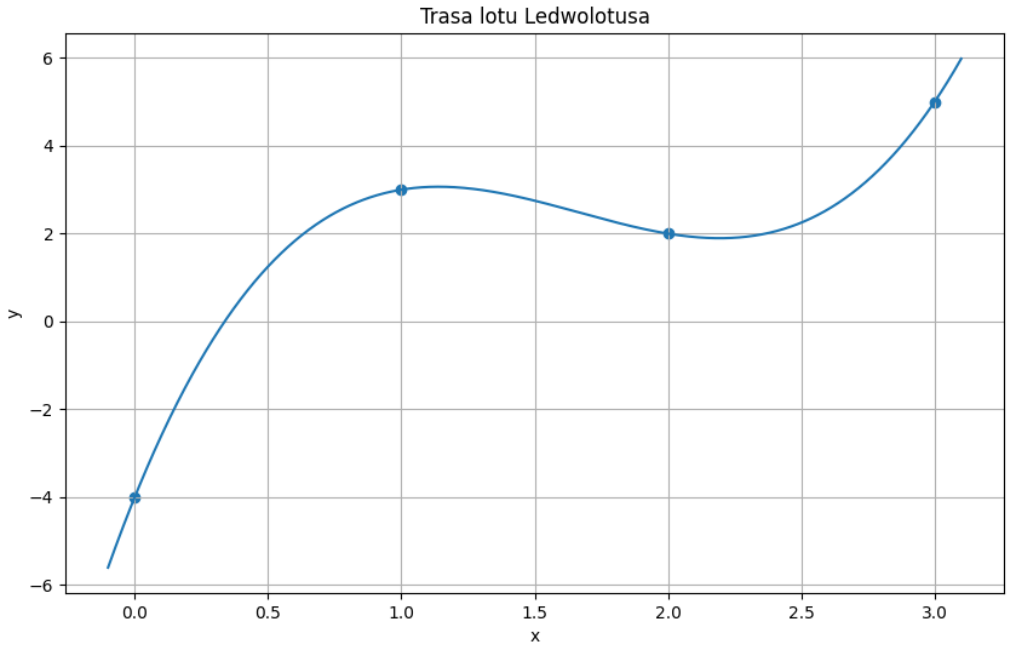


Uzyskane wykresy jednoznacznie wskazują na znaczącą przewagę wykorzystania węzłów Czebyszewa, dla których błąd bezwzględny wraz z wzrostem liczby węzłów maleje. Odwrotna sytuacja ma miejsce dla węzłów równoodległych, ponieważ w przypadku wzrostu ich liczby średni błąd bezwzględny interpolacji wielomianowej rośnie.

Zad D 10

```python
# Punkty interpolacji
survivor_points = [(0, -4), (1, 3), (3, 5), (2, 2)]

# Wyzanczenie funkcji bazowych Lagrange'a
def lagrange_base_f(i, x, points):
        x_i = [p[0] for p in points]
        return np.prod(x - np.delete(x_i, i)) / np.prod(x_i[i] - np.delete(x_i, i))

# Interpolacja Lagrange'a
def lagrange_interpolation(points):
    # Deklaracja przedziału dla którego wyznaczymy trasę Ledwolotusa
    X = np.linspace(min([p[0] for p in points])-0.1, max([p[0] for p in points])+0.1, 1000)
    Y = []

    # Wyznacznie wartości y dla poszczególnych x
    #P(x) = L0(x)*y + L1(x)*y + L2(x)*y + L3(x)*y
    for x in X:
        y = 0
        for i, p in enumerate(points):
            y += p[1] * lagrange_base_f(i, x, points)
        Y.append(y)

    return X, Y

X, Y = lagrange_interpolation(survivor_points)
```

Odtworzona trasa Ledwolotusa przebiegała w następujący sposób:



Zad D 11

```python
# Funkcja interpolacji Newtona
def newton_coeffs(x, y):
    n = len(x)
    coeffs = y.copy()
    for j in range(1, n):
        for i in range(n-1, j-1, -1):
            coeffs[i] = (coeffs[i] - coeffs[i-1]) / (x[i] - x[i-j])
            print(coeffs)
    return coeffs
```

```python
# Dane pomiarowe
times = [8, 9, 10, 11]              # Godziny pomiarów
temperatures = [20, 24, 26, 20]     # Temperatury dla poszczególnych godzin
print("\n")

# Obliczenie współczynników wielomianu interpolacyjnego
coeffs = newton_coeffs(times, temperatures)
```

```python
# Wyznaczenie wartosci na podstawie wielomianu interpolacyjnego
def interpolated_temperature(t, times):
    result = coeffs[0]
    for i in range(1, len(coeffs)):
        pom = coeffs[i]
        for j in range(i):
            pom *= (t - times[j])
        result += pom
    return result
```

```python
# Interpolacja temperatury o godzinie 10:30
hour_1030 = 10.5
temperature_1030 = interpolated_temperature(hour_1030, times)
```

```python
def float_to_time(f_h):
    X_h=[]
    for float_hour in f_h:
        hour = int(float_hour)
        minute = int((float_hour - hour) * 60)
        X_h.append(f"{hour}:{minute:02d}")
    return X_h
```

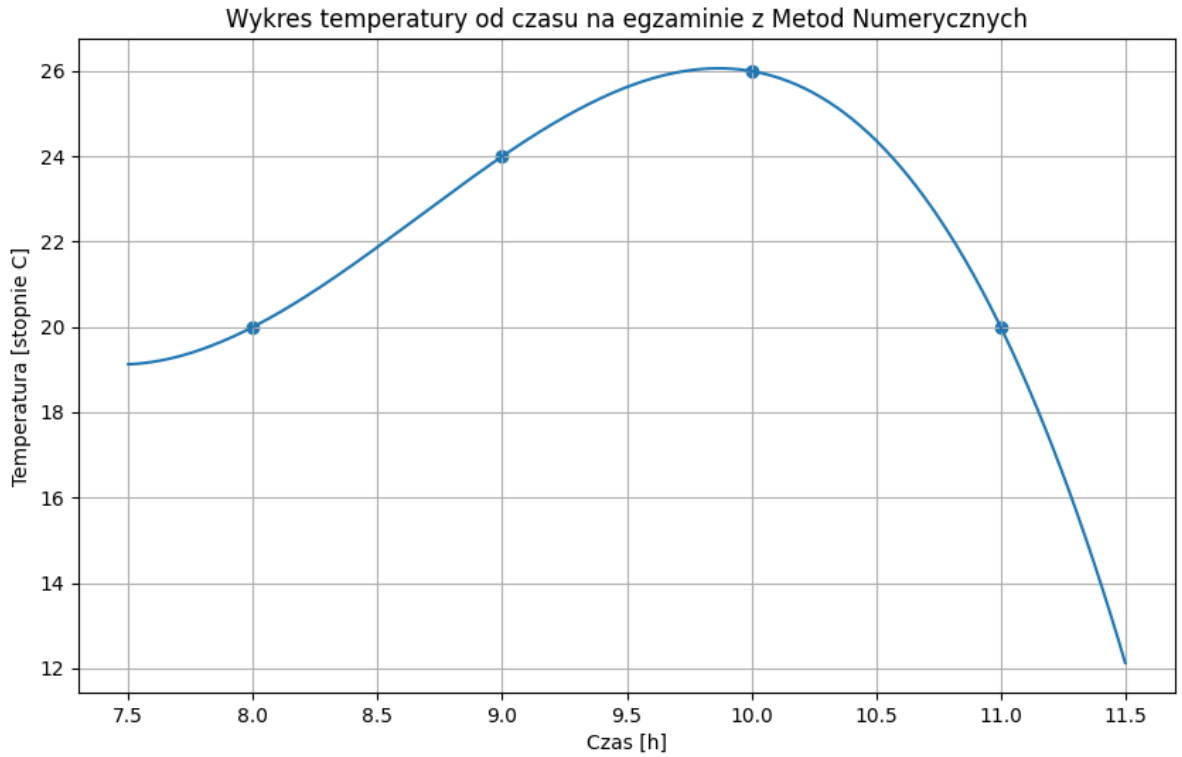
Temperatura w sali egzaminacyjnej o godzinie 10:30 wynosiła:

```
Temperatura o godzinie 10:30: 24.38 stopni Celsjusza
```

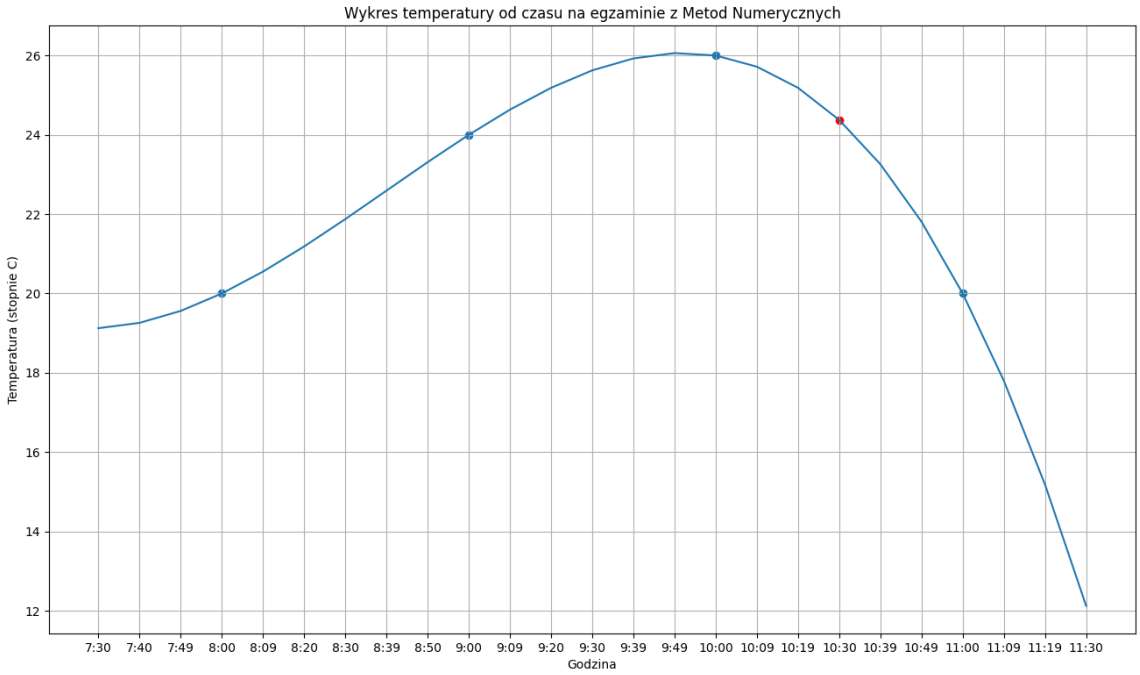
Algorytm kolejno od końca wyznacza kolejne współczynniki, używając do wyznaczenia kolejnych współczynników, tych z poprzednich iteracji (oczywiście pierwszy współczynnik przepisywany jest od razu):

```
[20, 24, 26, -6.0]
[20, 24, 2.0, -6.0]
[20, 4.0, 2.0, -6.0]
[20, 4.0, 2.0, -4.0]
[20, 4.0, -1.0, -4.0]
[20, 4.0, -1.0, -1.0]
```

Na podstawie wyznaczonego wielomianu Newtona wyznaczono Interpolację wartości temperatury.



Podanie godziny 10:30 jako 10.5 nie było wystarczająco satysfakcjonujące, dokonano więc zamiany godzin, przy pomocy prostej funkcji uzyskując następującą charakterystykę:



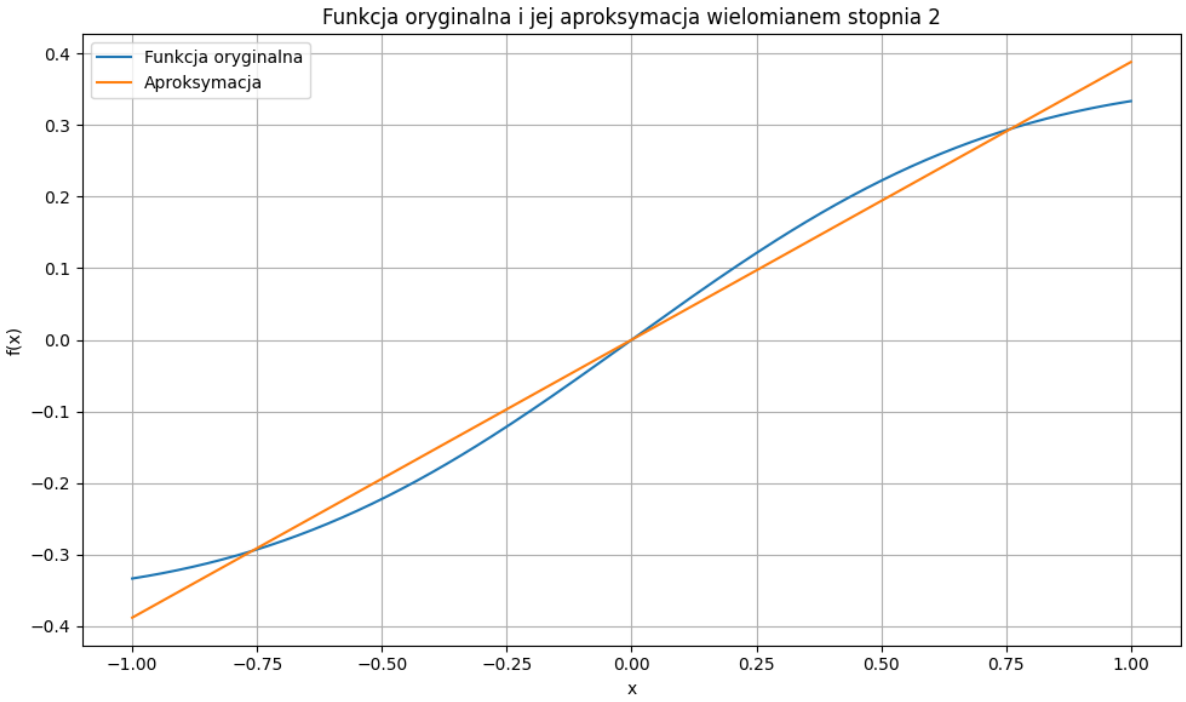
Zad E 12

```python
# Funkcja
def f(x):
    return x / (x**2 + 2)

# Wielomian aproksymacyjny - 2. stopnia
def approximation(x, a, b, c):
    return a * x**2 + b * x + c

# Przedział i krok
x_values = np.arange(-1.0, 1.01, 0.01)
y_values = f(x_values)

# Wyznaczenie współczynników wielomianu 2 stopnia
polynomial_coeffs = np.polyfit(x_values, y_values, 2)

# Obliczenie wartości aproksymacji wielomianowej
approx_values = approximation(x_values, *polynomial_coeffs)

# Błąd bezwzględny aproksymacji
absolute_error = np.abs(f(x_values) - approx_values)

# Błąd względny aproksymacji
relative_error = np.abs((f(x_values) - approx_values))/np.abs(f(x_values))*100

# Nie dzielimy przez 0
relative_error[100] = 'nan'

# Maksymalny błąd aproksymacji
max_error = np.max(absolute_error)
```

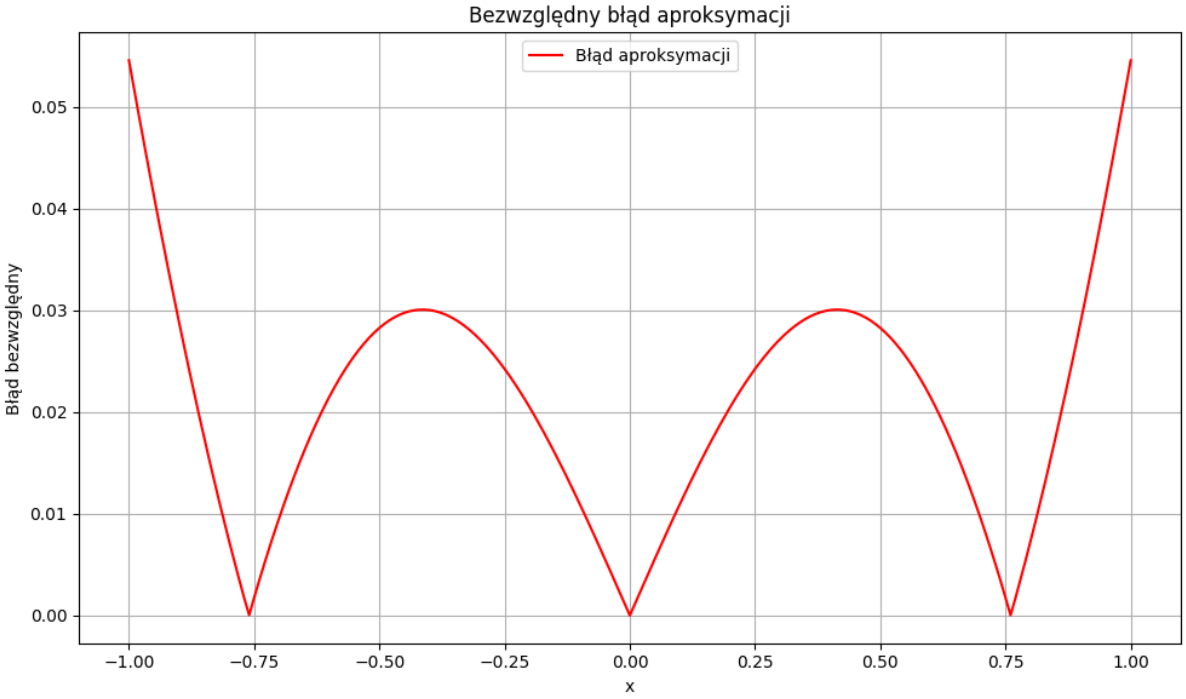
Maksymalny bezwzględny błąd aproksymacji wynosił:

```
Maksymalny błąd aproksymacji: 0.054580421688088
```

Poniżej przedstawiono wykres oryginalnej funkcji oraz jej aproksymacji

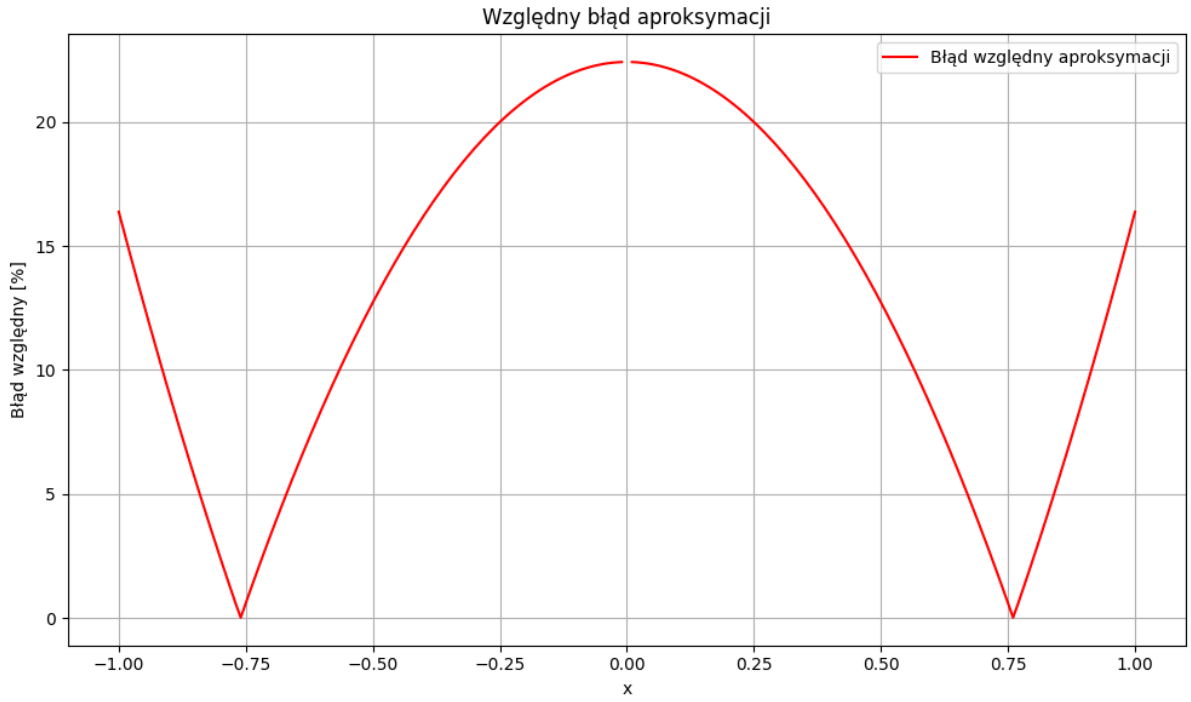


Poniżej przedstawiono charakterystykę błędu bezwzględnego aproksymacji wielomianem drugiego stopnia:



Na podstawie charakterystyki błędu bezwzględnego możemy zaobserwować, że błąd bezwzględny dla wartości od -0,9 do 0,9 nie przekracza 0,3. Największą wartość błędu bezwzględnego funkcja osiąga na granicach przedziału.

Wyznaczono również błąd względny aproksymacji (oczywiście pominięto punkt (0, 0)):



Błąd bezwzględny daje mylne wrażenie, że na granicach przedziału popełniany błąd jest największy. Jednak gry weźmiemy pod uwagę wartości funkcji i wyznaczymy błąd względny maksymalny błąd osiągamy dla punktu -0,01 oraz 0,01 i wynosi on:

```
Maksymalny błąd względny: 22.41%
```

Wynika to oczywiście z małej wartości funkcji w tych punktach. Najniższy błąd względny otrzymano natomiast w otoczeniu dwóch punktów przecięcia wykresu funkcji i jego aproksymacji.

Zad E 13

```python
# Dane
xi = np.array([0.4, 0.8, 1.2, 1.6, 2.0, 2.3])
yi = np.array([750, 1000, 1400, 2000, 2700, 3750])

'''plt.figure(figsize=(10, 6))···

# Funkcja eksponencjalna
def exponential_f(x, a0, a1):
    return a0 * np.exp(a1 * x)

# Dopasowanie modelu eksponencjalnego metodą najmniejszych kwadratów
params_exp, covariance_exp = curve_fit(exponential_f, xi, yi)
a0_exp, a1_exp = params_exp

# Przygotowanie danych do regresji liniowej
x_lin = xi.reshape(-1, 1)
# Zastosowanie ln do przekształcenia danych
y_lin = np.log(yi)

'''plt.figure(figsize=(10, 6))···

# Dopasowanie modelu liniowego
linear_regression = LinearRegression()
linear_regression.fit(x_lin, y_lin)

# Wyciągnięcie współczynników regresji liniowej
a1_lin = linear_regression.coef_[0]
a0_lin = np.exp(linear_regression.intercept_)

# Obliczenie przybliżeń punktów początkowych dla obu modeli
predict_exp_values = exponential_f(xi, a0_exp, a1_exp)
predict_lin_values = exponential_f(xi, a0_lin, a1_lin)

# Porównanie błędu R^2
r2_exp = r2_score(yi, predict_exp_values)
r2_lin = r2_score(yi, predict_lin_values)
```

Uzyskane parametry modelu ekspotencjalnego:

```
Parametry modelu eksponencjalnego:
a0: 489.51158925311313
a1: 0.8763237240936685

Parametry modelu liniowego:
a0: 519.4969133545007
a1: 0.8417242672173104
```

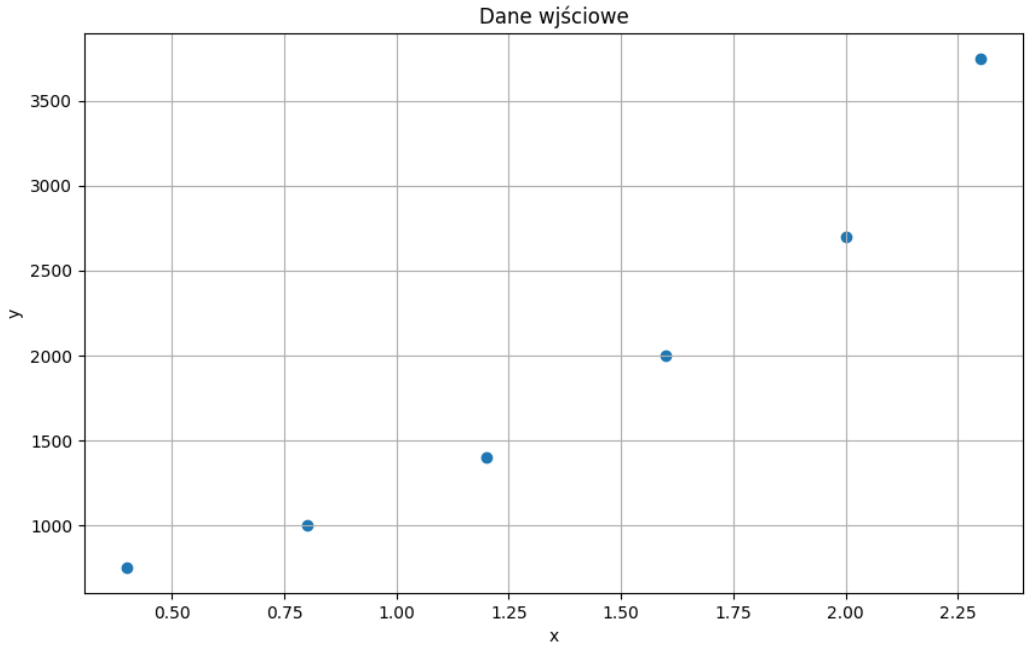
Dopasowanie wyznaczonych modeli oceniono na podstawie błędu R^2:

```
Współczynniki R^2:
Model eksponencjalny: 0.9961827941737817
Model liniowy:        0.994831868988449
```

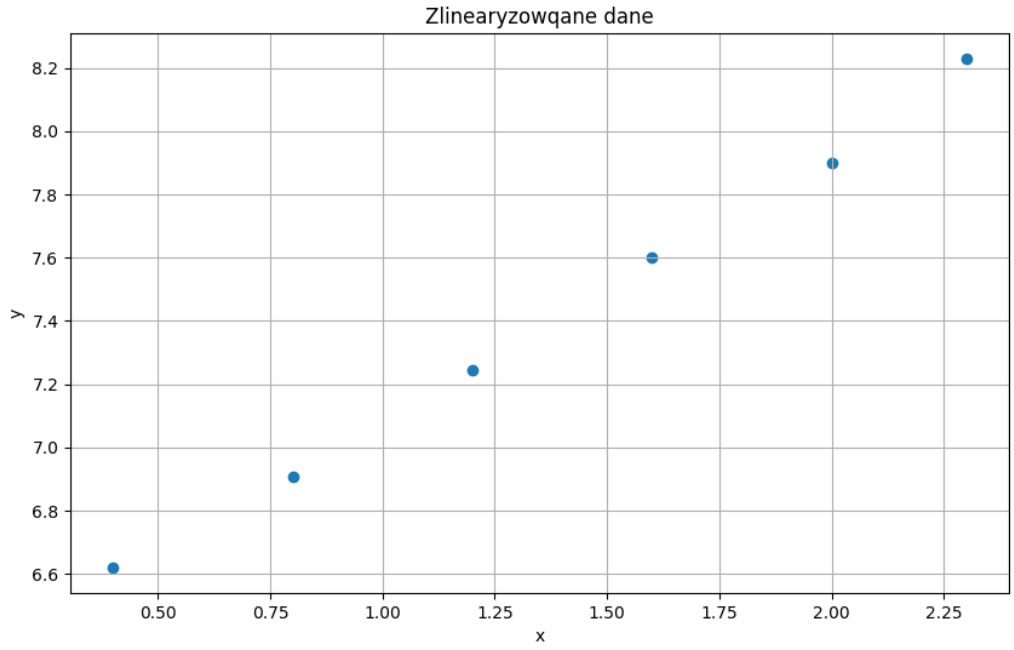
Możemy zaobserwować, że dopasowanie modelu bez wcześniejszej linearyzacji dało lepszy efekt, jednak wartość dopasowania dla modelu zlinearyzowanego była niższa jedynie o:

```
0.0013509251853327 52
```

Wejściowe Dane:



Dane po linearyzacji:



Uzyskane dopasowania dla obu modeli: