Sprawozdanie z laboratorium 4.

Hubert Miklas

1 kwietnia 2025

1 Zadanie 1: Przybliżenie wielomianu trzeciego stopnia wielomianem stopnia pierwszego korzystając z metody średniokwadratowej

Aby przybliżyć $f(x) = 1 + x^3$ w przedziale $[0, 1]$ wielomianem trzeciego stopnia $p(x) = a_0 + a_1 x$ stosując metodę ciągłych średnich kwadratów dla $w(x) = 1$ [1]:

Współczynniki są znajdowane przez minimalizowanie całki:

$$E = \int_0^1 [f(x) - p(x)]^2 w(x)\, dx = \int_0^1 [(1 + x^3) - (a_0 + a_1 x)]^2\, dx \tag{1}$$

Aby znaleźć minimum, różniczkujemy względem a_0 i a_1 i ustawiamy pochodne na zero:

$$\frac{\partial E}{\partial a_0} = \int_0^1 -2[(1 + x^3) - (a_0 + a_1 x)]\, dx = 0 \tag{2}$$

$$\frac{\partial E}{\partial a_1} = \int_0^1 -2x[(1 + x^3) - (a_0 + a_1 x)]\, dx = 0 \tag{3}$$

Dostajemy z tego układ równań:

$$\int_0^1 (1 + x^3)\, dx = \int_0^1 (a_0 + a_1 x)\, dx \tag{4}$$

$$\int_0^1 x(1 + x^3)\, dx = \int_0^1 x(a_0 + a_1 x)\, dx \tag{5}$$

Po przekształceniu:

$$a_0 \int_0^1 dx + a_1 \int_0^1 x\, dx = \int_0^1 dx + \int_0^1 x^3\, dx$$
$$a_0 \int_0^1 x\, dx + a_1 \int_0^1 x^2\, dx = \int_0^1 x\, dx + \int_0^1 x^4\, dx$$

Otrzymujemy układ równań:

$$\begin{cases} a_0 + \frac{a_1}{2} = \frac{5}{4} \\ \frac{a_0}{2} + \frac{a_1}{3} = \frac{7}{10} \end{cases} \tag{6}$$

Rozwiązując powyższy układ równań, otrzymujemy:

$$a_0 = \frac{4}{5}, \quad a_1 = \frac{9}{10} \tag{7}$$

Ostatecznie, przybliżenie wielomianem pierwszego stopnia wynosi:

$$p(x) = \frac{8 + 9x}{10} \tag{8}$$

Na wykresie to przybliżenie wygląda następująco:

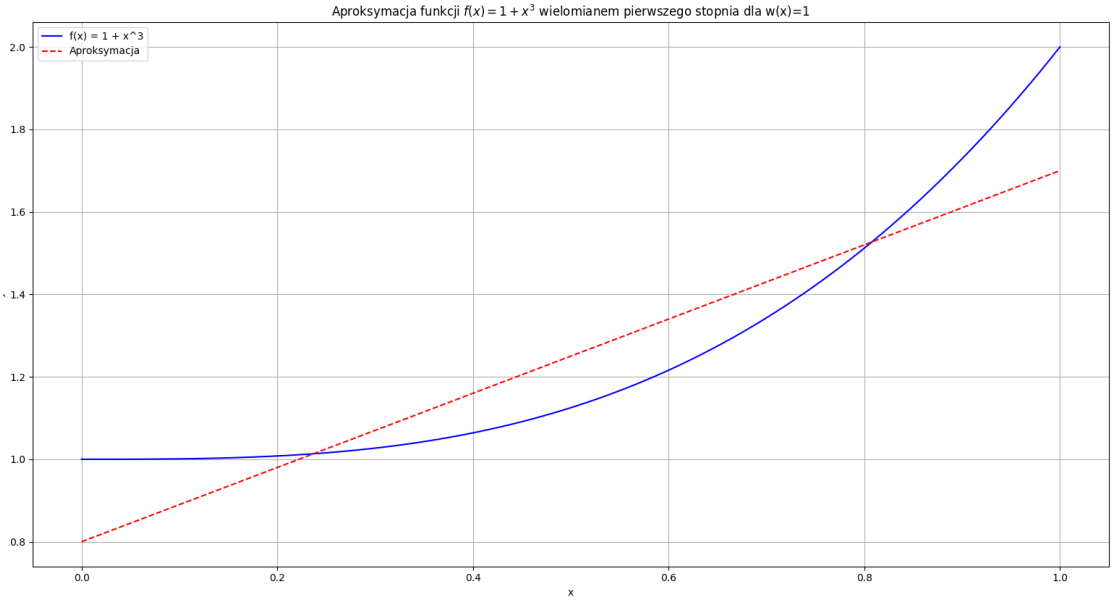

Figure 1: Aproksymacja funckji $f(x) = 1 + x^3$

2 Zadanie 2: Aproksymacja wielomianowa drugiego stopnia przy użyciu wielomianów Legendre'a

Aby przybliżyć $f(x) = 1 + x^3$ za pomocą wielomianów Legendre'a [3] do drugiego stopnia, używamy:

$$p(x) = c_0 P_0(x) + c_1 P_1(x) + c_2 P_2(x) \tag{9}$$

Gdzie wielomiany Legendre'a na przedziale $[0, 1]$ to:

$$\begin{aligned} P_0(x) &= 1, \\ P_1(x) &= 2x - 1, \\ P_2(x) &= 6x^2 - 6x + 1. \end{aligned}$$

Współczynniki wyraża się przez:

$$c_j = \frac{2j+1}{2} \int_0^1 f(x) P_j(x)\, dx \tag{10}$$

Dla $j \in \{0, 1, 2\}$:

$$\begin{aligned} c_0 &= \frac{1}{2} \int_0^1 (1 + x^3)\, dx \\ c_1 &= \frac{3}{2} \int_0^1 (1 + x^3)(2x - 1)\, dx \\ c_2 &= \frac{5}{2} \int_0^1 (1 + x^3)(6x^2 - 6x + 1)\, dx \end{aligned}$$

Po obliczeniu tych całek otrzymujemy wartości:

$$c_0 = \frac{5}{8}, \quad c_1 = \frac{9}{40}, \quad c_2 = \frac{1}{8}.$$

Ostatecznie, przybliżenie wielomianem drugiego stopnia funkcji $f(x) = 1 + x^3$:

$$p(x) = \frac{21}{40} - \frac{3}{10}x + \frac{3}{4}x^2 \tag{11}$$

3 Zadanie domowe 1: Aproksymacja punktowa wielomianem stopnia drugiego

Celem zadania jest aproksymacja funkcji danej w punktach dyskretnych x_i przy użyciu wielomianu drugiego stopnia:

$$F(x) = a_0 + a_1 x + a_2 x^2$$

Zastosowano metodę najmniejszych kwadratów, która polega na minimalizacji błędu:

$$\sum_{i=1}^{n} (f(x_i) - F(x_i))^2 \to \min$$

Rozwiązanie sprowadza się do rozwiązania układu równań normalnych dla współczynników a_0, a_1, a_2. W tym celu skonstruowano macierz Vandermonde'a:

$$A = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 \end{bmatrix} \quad \text{oraz} \quad \mathbf{b} = \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{bmatrix}$$

Następnie rozwiązano układ równań $A^T A\vec{a} = A^T\vec{b}$ metodą najmniejszych kwadratów, otrzymując współczynniki a_0, a_1, a_2.

3.1 Implementacja

Poniżej znajduje się implementacja powyższej metody z demonstracją działania:

```
import numpy as np
import matplotlib.pyplot as plt

def point_square_approximation(xi, fi):
    A = np.vstack([np.ones_like(xi), xi, xi**2]).T
    b = fi
    coeffs = np.linalg.lstsq(A, b, rcond=None)[0]
    return coeffs

def plot_approximation(xi, fi, coeffs, original_func=None):
    a0, a1, a2 = coeffs
    x_dense = np.linspace(min(xi) - 0.5, max(xi) + 0.5, 1000)
    approx_values = a0 + a1 * x_dense + a2 * x_dense**2
    plt.figure(figsize=(10, 6))

    plt.scatter(xi, fi, color='red', s=50, label='Punkty oryginalne')
    plt.plot(x_dense, approx_values, 'b-', label=f'Aproksymacja: {a0:.4f} + {a1:.4f}x + {a2:.4f}x²')

    if original_func:
        original_values = original_func(x_dense)
        plt.plot(x_dense, original_values, 'g--', label='Funkcja oryginalna')

    plt.grid(True, alpha=0.3)
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.title('Aproksymacja kwadratowa')
    plt.legend()
```

```
    approx_at_points = a0 + a1 * xi + a2 * xi**2
    mean_squared_error = np.mean((fi - approx_at_points)**2)
    max_error = np.max(np.abs(fi - approx_at_points))

    plt.figtext(0.5, 0.01, f'Błąd średniokwadratowy: {mean_squared_error:.6f}\nBłąd maksymalny: {max_er
                ha='center', bbox={'facecolor':'lightgray', 'alpha':0.5, 'pad':5})

    plt.tight_layout(rect=[0, 0.05, 1, 1])
    plt.show()

xi = np.array([-1.0, -0.5, 0.0, 0.5, 1.0])
fi = 1 - xi**2

# Definiujemy funkcję oryginalną dla porównania
def original_function(x):
    return 1 - x**2

coeffs = point_square_approximation(xi, fi)
a0, a1, a2 = coeffs
print(f"Coefficients: a0 = {a0:.4f}, a1 = {a1:.4f}, a2 = {a2:.4f}")

plot_approximation(xi, fi, coeffs, original_function)

print("\nExample 2: f(x) = sin(x)")
xi2 = np.linspace(-1, 1, 8)
fi2 = np.sin(np.pi * xi2)

coeffs2 = point_square_approximation(xi2, fi2)
a0, a1, a2 = coeffs2
print(f"Coefficients: a0 = {a0:.4f}, a1 = {a1:.4f}, a2 = {a2:.4f}")

plot_approximation(xi2, fi2, coeffs2, lambda x: np.sin(np.pi * x))

print("\nExample 3: f(x) = e^x")
xi3 = np.linspace(-1, 1, 6)
fi3 = np.exp(xi3)

coeffs3 = point_square_approximation(xi3, fi3)
a0, a1, a2 = coeffs3
print(f"Współczynniki: a0 = {a0:.4f}, a1 = {a1:.4f}, a2 = {a2:.4f}")

plot_approximation(xi3, fi3, coeffs3, lambda x: np.exp(x))
```

4 Zadanie domowe 2: Aproksymacja funkcji $f(x) = 1-x^2$ wielomianem stopnia 3 (Grama)

W zadaniu dokonano aproksymacji funkcji

$$f(x) = 1 - x^2$$

w pięciu punktach:

$$x_i = -1 + 0.5 \cdot i, \quad i = 0, 1, 2, 3, 4$$

czyli:

$$x = [-1.0,\ -0.5,\ 0.0,\ 0.5,\ 1.0]$$

Jako bazę przyjęto funkcje:

$$\varphi_0(x) = 1, \quad \varphi_1(x) = x, \quad \varphi_2(x) = x^2, \quad \varphi_3(x) = x^3$$

Następnie wykonano ortogonalizację metodą Grama-Schmidta [2], uzyskując bazę ortogonalną $\psi_k(x)$, dla $k = 0, 1, 2, 3$, z której zbudowano funkcję aproksymującą:

$$F(x) = \sum_{k=0}^{3} a_k \cdot \psi_k(x)$$

Współczynniki a_k obliczono ze wzoru:

$$a_k = \frac{\sum_{i=1}^{n} f(x_i) \cdot \psi_k(x_i)}{\sum_{i=1}^{n} \psi_k^2(x_i)}$$

4.1 Wyniki

Na wykresie porównano oryginalną funkcję $f(x)$ oraz funkcję aproksymującą $F(x)$.

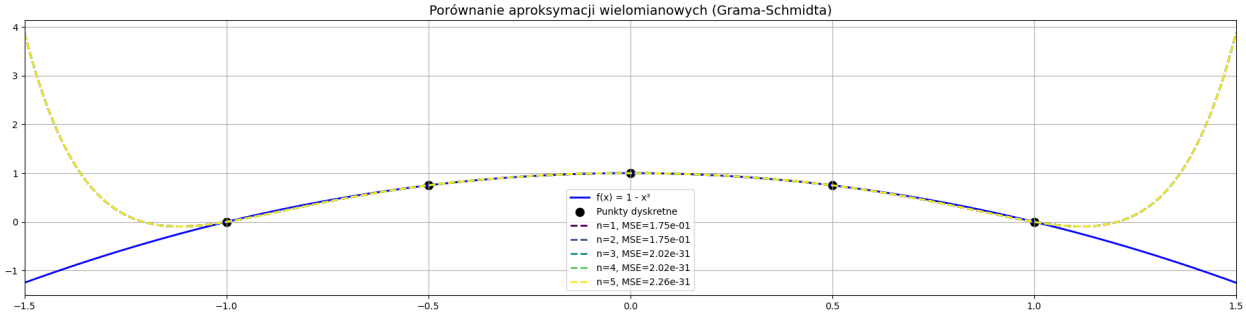

Figure 2: Caption

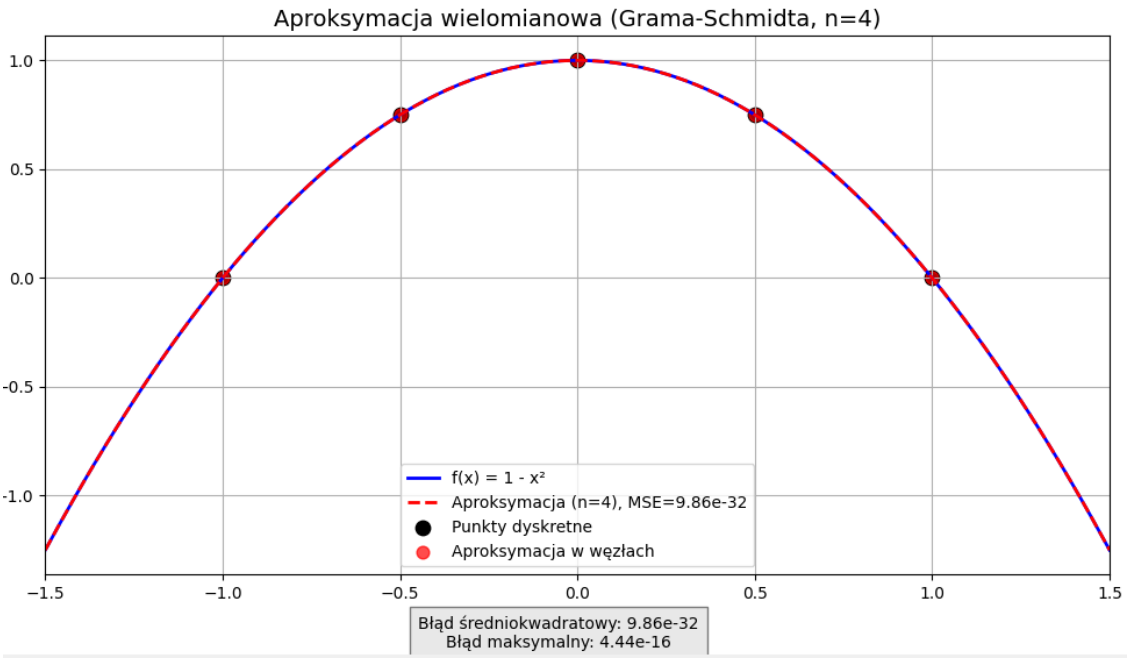

Figure 3: Caption

4.2 Implementacja

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec

xi = np.array([-1 + 0.5 * i for i in range(5)])
fi = 1 - xi**2

def phi(j, x):
    return x**j

def gram_schmidt(xi, n):
    n_points = len(xi)
    Phi = np.zeros((n, n_points))
    Psi = np.zeros_like(Phi)

    for j in range(n):
        Phi[j, :] = phi(j, xi)

    Psi[0, :] = Phi[0, :]
    for k in range(1, n):
        proj = np.zeros(n_points)
        for j in range(k):
            numer = np.dot(Phi[k, :], Psi[j, :])
            denom = np.dot(Psi[j, :], Psi[j, :])
            proj += (numer / denom) * Psi[j, :]
        Psi[k, :] = Phi[k, :] - proj

    return Psi
```

```
def polynomial_approximation(xi, psi_values, degree=10):
    poly_coeffs = []
    for psi in psi_values:
        coeffs = np.polyfit(xi, psi, degree)
        poly_coeffs.append(coeffs)
    return poly_coeffs

def eval_polynomial(coeffs, x):
    return np.polyval(coeffs, x)

def compute_approximation(xi, fi, max_n=5):
    results = []

    for n in range(1, max_n + 1):
        Psi = gram_schmidt(xi, n)
        ak = np.array([np.dot(fi, Psi[k]) / np.dot(Psi[k], Psi[k]) for k in range(n)])
        poly_coeffs = polynomial_approximation(xi, Psi)

        def F(x):
            result = np.zeros_like(x)
            for k in range(n):
                result += ak[k] * eval_polynomial(poly_coeffs[k], x)
            return result

        F_xi = F(xi)
        err = np.mean((fi - F_xi) ** 2)
        results.append((n, Psi, ak, poly_coeffs, F, err))
    return results

def visualize_results(xi, fi, results, original_func):
    n_results = len(results)

    fig = plt.figure(figsize=(15, 10))
    gs = GridSpec(2, 3, figure=fig)

    ax_main = fig.add_subplot(gs[0, :])
    x_vals = np.linspace(-1.5, 1.5, 300)
    f_vals = original_func(x_vals)

    ax_main.plot(x_vals, f_vals, 'b-', label='f(x) = 1 - x²', linewidth=2)
    ax_main.scatter(xi, fi, color='black', s=80, label='Punkty dyskretne')

    colors = plt.cm.viridis(np.linspace(0, 1, n_results))
    for i, (n, _, _, _, F, err) in enumerate(results):
        F_vals = F(x_vals)
        ax_main.plot(x_vals, F_vals, '--', color=colors[i],
                     label=f'n={n}, MSE={err:.2e}', linewidth=2)

    ax_main.set_title("Porównanie aproksymacji wielomianowych (Grama-Schmidta)", fontsize=14)
    ax_main.legend(loc='best')
    ax_main.grid(True)
    ax_main.set_xlim(-1.5, 1.5)
```

```
    for i in range(min(n_results, 5)):
        n, Psi, ak, poly_coeffs, _, _ = results[i]
        if i >= 3:  # Tylko do stopnia 3 włącznie
            continue

        ax = fig.add_subplot(gs[1, i])

        for j in range(n):
            poly_x = np.linspace(-1.5, 1.5, 200)
            poly_y = eval_polynomial(poly_coeffs[j], poly_x)

            norm_factor = np.max(np.abs(poly_y)) if np.max(np.abs(poly_y)) > 0 else 1
            poly_y /= norm_factor

            ax.plot(poly_x, poly_y, label=f'{j}(x)')
            ax.scatter(xi, Psi[j]/np.max(np.abs(Psi[j])) if np.max(np.abs(Psi[j])) > 0 else Psi[j],
                       color='black', s=30)

        ax.set_title(f"Funkcje ortogonalne (n={n})")
        ax.grid(True)
        ax.legend(loc='best')
        ax.set_xlim(-1.5, 1.5)
        ax.set_ylim(-1.2, 1.2)

    ax_err = fig.add_subplot(gs[1, 2])
    degrees = [r[0] for r in results]
    errors = [r[5] for r in results]

    ax_err.semilogy(degrees, errors, 'o-', linewidth=2)
    ax_err.set_title("Błąd średniokwadratowy")
    ax_err.set_xlabel("Stopień wielomianu")
    ax_err.set_ylabel("MSE (skala logarytmiczna)")
    ax_err.grid(True)

    plt.tight_layout()
    return fig

def original_function(x):
    return 1 - x**2

results = compute_approximation(xi, fi, max_n=5)
fig = visualize_results(xi, fi, results, original_function)
plt.show()

Psi = gram_schmidt(xi, 4)

ak = np.array([np.dot(fi, Psi[k]) / np.dot(Psi[k], Psi[k]) for k in range(4)])
print("Coefficients ak:", ak)

print("\nMatrix of scalar multiplication for basis functions:")
for i in range(4):
    for j in range(4):
        dot_prod = np.dot(Psi[i], Psi[j])
        if i != j:
```

```
            print(f"<{i},{j}> = {dot_prod:.2e}", end="\t")
        else:
            print(f"<{i},{i}> = {dot_prod:.4f}", end="\t")
    print()

def F(x):
    x_array = np.atleast_1d(x)
    result = np.zeros_like(x_array, dtype=float)

    for k in range(4):
        poly_coeff = np.polyfit(xi, Psi[k], deg=3)
        result += ak[k] * np.polyval(poly_coeff, x_array)

    return result if len(result) > 1 else result[0]

x_vals = np.linspace(-1.5, 1.5, 300)
f_vals = original_function(x_vals)
F_vals = F(x_vals)

F_xi = F(xi)
mse = np.mean((fi - F_xi) ** 2)
max_err = np.max(np.abs(fi - F_xi))

plt.figure(figsize=(10, 6))
plt.plot(x_vals, f_vals, 'b-', label='f(x) = 1 - x²', linewidth=2)
plt.plot(x_vals, F_vals, 'r--', label=f'Aproksymacja (n=4), MSE={mse:.2e}', linewidth=2)
plt.scatter(xi, fi, color='black', s=80, label='Punkty dyskretne')

plt.scatter(xi, F_xi, color='red', s=60, alpha=0.7, label='Aproksymacja w węzłach')

plt.title("Aproksymacja wielomianowa (Grama-Schmidta, n=4)", fontsize=14)
plt.legend(loc='best')
plt.grid(True)
plt.xlim(-1.5, 1.5)
plt.figtext(0.5, 0.01,
            f'Błąd średniokwadratowy: {mse:.2e}\nBłąd maksymalny: {max_err:.2e}',
            ha='center',
            bbox={'facecolor':'lightgray', 'alpha':0.5, 'pad':5})
plt.tight_layout(rect=[0, 0.05, 1, 0.95])
plt.show()
```

5 Zadanie 3: Aproksymacja funkcji $f(x) = |\sin(x)|$ funkcjami trygonometrycznymi

Celem zadania była aproksymacja funkcji:

$$f(x) = |\sin(x)|$$

w przedziale $x \in [-\pi, \pi]$ przy użyciu szeregu Fouriera [4].

Z uwagi na parzystość funkcji $f(x)$, rozwinięcie zawiera wyłącznie funkcje kosinusowe:

$$f(x) \sim \frac{a_0}{2} + \sum_{n=1}^{N} a_n \cos(nx)$$

Współczynniki a_n obliczono ze wzoru:

$$a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \cos(nx)\, dx$$

gdzie całkowanie przeprowadzono numerycznie. W zadaniu przyjęto $N = 10$ składników szeregu. Wartości współczynników posłużyły do konstrukcji funkcji aproksymującej $F(x)$, której wykres porównano z wykresem oryginalnej funkcji $|\sin(x)|$.

Na wykresie zauważalna jest zbieżność funkcji aproksymującej do funkcji oryginalnej, zwłaszcza dla większych wartości N.

5.1 Wyniki

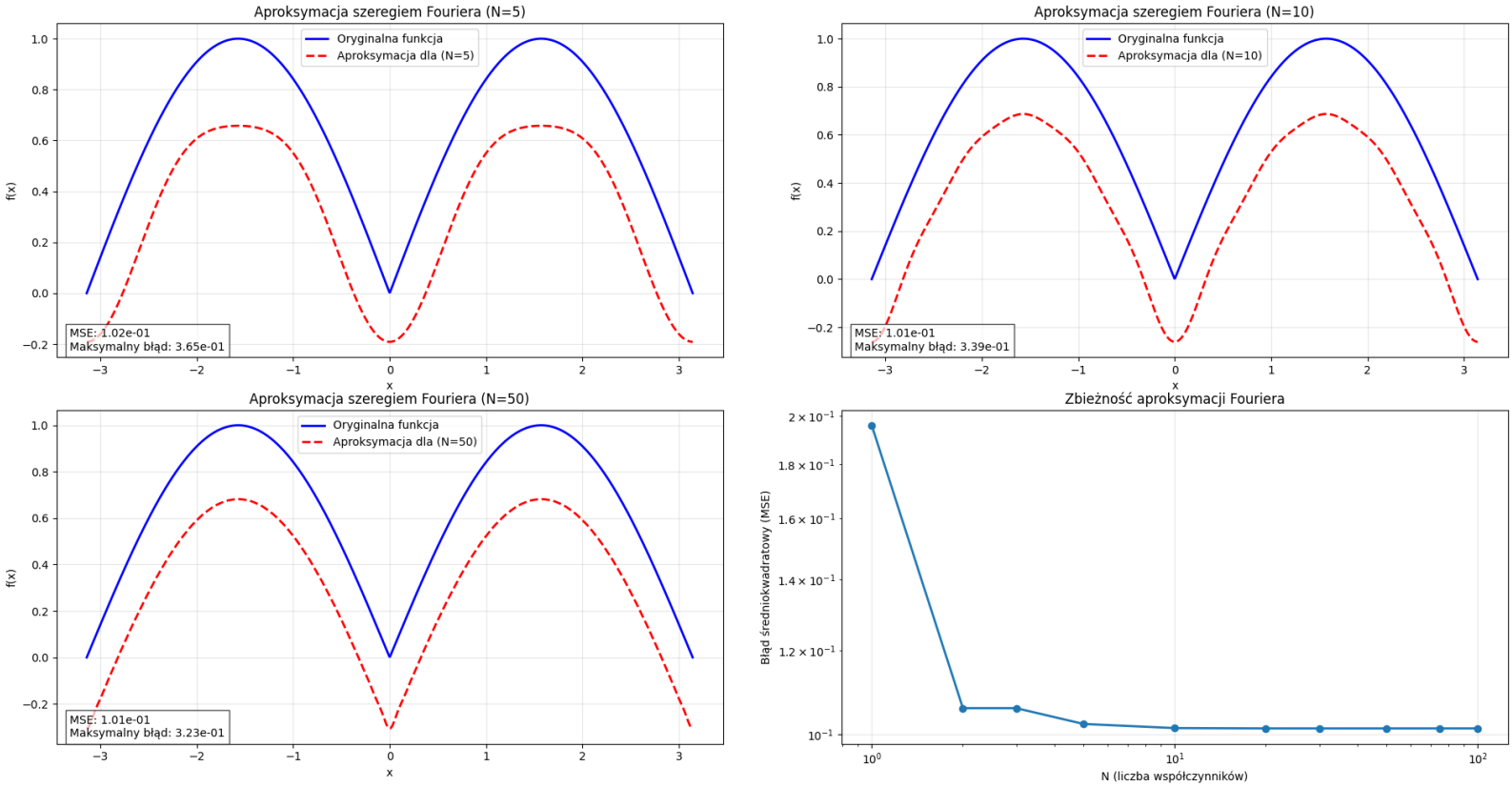

Figure 4: Caption

5.2 Implementacja

```
    import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import quad
from matplotlib.gridspec import GridSpec
import time

def fourier_cosine_coefficient(f, n, a=-np.pi, b=np.pi):
    interval_length = b - a
    if n == 0:
        result, _ = quad(lambda x: f(x), a, b)
```

```
        return result / interval_length
    else:
        result, _ = quad(lambda x: f(x) * np.cos(n * 2 * np.pi * (x - a) / interval_length), a, b)
        return 2 * result / interval_length

def fourier_sine_coefficient(f, n, a=-np.pi, b=np.pi):
    if n == 0:
        return 0
    else:
        interval_length = b - a
        result, _ = quad(lambda x: f(x) * np.sin(n * 2 * np.pi * (x - a) / interval_length), a, b)
        return 2 * result / interval_length

def fourier_coefficients(f, N, a=-np.pi, b=np.pi, include_sine=False):
    a_coeffs = [fourier_cosine_coefficient(f, n, a, b) for n in range(N+1)]

    if include_sine:
        b_coeffs = [fourier_sine_coefficient(f, n, a, b) for n in range(N+1)]
        return a_coeffs, b_coeffs
    else:
        return a_coeffs

def fourier_approximation(x, a_coeffs, b_coeffs=None, a=-np.pi, b=np.pi):
    interval_length = b - a
    x_norm = 2 * np.pi * (x - a) / interval_length

    # Initialize with a_0/2 term
    sum_val = a_coeffs[0] / 2

    # Add cosine terms
    for n in range(1, len(a_coeffs)):
        sum_val += a_coeffs[n] * np.cos(n * x_norm)

    # Add sine terms if provided
    if b_coeffs is not None:
        for n in range(1, len(b_coeffs)):
            sum_val += b_coeffs[n] * np.sin(n * x_norm)

    return sum_val

def calculate_fourier_error(f, a_coeffs, b_coeffs=None, a=-np.pi, b=np.pi, num_points=1000):
    x_vals = np.linspace(a, b, num_points)
    f_vals = f(x_vals)
    F_vals = fourier_approximation(x_vals, a_coeffs, b_coeffs, a, b)

    mse = np.mean((f_vals - F_vals) ** 2)
    max_error = np.max(np.abs(f_vals - F_vals))

    return mse, max_error

def analyze_fourier_convergence(f, max_N, a=-np.pi, b=np.pi, include_sine=False):
    N_values = [1, 2, 3, 5, 10, 20, 30, 50, 75, 100][:np.searchsorted(np.array([1, 2, 3, 5, 10, 20, 30,
    N_values = [n for n in N_values if n <= max_N]
```

```
    errors = []
    computation_times = []

    for N in N_values:
        start_time = time.time()
        if include_sine:
            a_coeffs, b_coeffs = fourier_coefficients(f, N, a, b, include_sine=True)
            mse, _ = calculate_fourier_error(f, a_coeffs, b_coeffs, a, b)
        else:
            a_coeffs = fourier_coefficients(f, N, a, b, include_sine=False)
            mse, _ = calculate_fourier_error(f, a_coeffs, None, a, b)

        end_time = time.time()
        computation_times.append(end_time - start_time)
        errors.append(mse)

    return N_values, errors, computation_times

def plot_fourier_approximation(f, a_coeffs, b_coeffs=None, a=-np.pi, b=np.pi, N=None, ax=None):
    if ax is None:
        fig, ax = plt.subplots(figsize=(10, 6))

    x_vals = np.linspace(a, b, 1000)
    f_vals = f(x_vals)
    F_vals = fourier_approximation(x_vals, a_coeffs, b_coeffs, a, b)

    mse, max_error = calculate_fourier_error(f, a_coeffs, b_coeffs, a, b)

    ax.plot(x_vals, f_vals, 'b-', label=f'Oryginalna funkcja', linewidth=2)
    ax.plot(x_vals, F_vals, 'r--', label=f'Aproksymacja dla (N={N if N else len(a_coeffs)-1})', linewidt

    ax.set_xlabel('x')
    ax.set_ylabel('f(x)')

    title = 'Aproksymacja szeregiem Fouriera'
    if N is not None:
        title += f' (N={N})'
    ax.set_title(title)

    ax.grid(True, alpha=0.3)
    ax.legend()

    # Add error information
    ax.text(0.02, 0.02, f'MSE: {mse:.2e}\nMaksymalny błąd: {max_error:.2e}',
            transform=ax.transAxes, bbox=dict(facecolor='white', alpha=0.7))

    return ax

def plot_fourier_coefficients(a_coeffs, b_coeffs=None, ax=None):
    if ax is None:
        fig, ax = plt.subplots(figsize=(10, 6))

    n_values = np.arange(len(a_coeffs))
```

```
    ax.stem(n_values, np.abs(a_coeffs), 'b', markerfmt='bo', label='|a_n| (Cosine)')

    if b_coeffs is not None:
        ax.stem(n_values, np.abs(b_coeffs), 'r', markerfmt='ro', label='|b_n| (Sine)')

    ax.set_xlabel('n')
    ax.set_ylabel('|Coefficient|')
    ax.set_title('Fourier Coefficient Magnitudes')
    ax.set_yscale('log')
    ax.grid(True, alpha=0.3)
    ax.legend()

    return ax
```

References

[1] Włodzimierz Funika. Aproksymacja. `@misc{key,author={},title={},howpublished={\url{https://home.agh.edu.pl/~funika/mownit/lab4/aproksymacja.pdf}},year={},note={[Accessed07-04-2025]},}`, 2025. [Accessed 07-04-2025].

[2] Wikipedia. Gram–Schmidt process — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Gram\%E2\%80\%93Schmidt\%20process&oldid=1279087342`, 2025. [Online; accessed 07-April-2025].

[3] Wikipedia. Legendre polynomials — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Legendre\%20polynomials&oldid=1283644974`, 2025. [Online; accessed 07-April-2025].

[4] Wikipedia. Transformacja Fouriera — Wikipedia, the free encyclopedia. `http://pl.wikipedia.org/w/index.php?title=Transformacja\%20Fouriera&oldid=75439568`, 2025. [Online; accessed 07-April-2025].