Sprawozdanie z MIO laboratorium 09 - Marcin Knapczyk

Zadanie 1

Proszę, korzystając z algorytmu roju cząstek znaleźć minimum funkcji

$$f(x,y) = 2ln(|x + 0.2| + 0.002) + ln(|y + 0.1| + 0.001) + cos(3x) + 2sin^2(3xy) + sin^2(y) - x^2 - 0.5y^2$$

w przedziale $x, y \in [-1, 1]$,

przy założeniu że rozwiązanie jest reprezentowane przez wektor $[xi, yi]$.

Proszę zbadać:

- funkcjonowanie algorytmu dla c1=0, c2=2
- funkcjonowanie algorytmu dla c1=2, c2=0 oraz kilku przypadków pośrednich.
- funkcjonowanie algorytmu dla c1 = c2 = 2,2
- funkcjonowanie dla różnych wartości w.

Za każdym razem należy podać średnie wyniki (wartość funkcji przystosowania) oraz odchylenie standardowe dla 10 wywołań algorytmu i przedstawić przykładowe przebiegi algorytmu na wykresach (dla jednego z wywołań).

Należy opisać, który wariant sprawdzał się najlepiej.

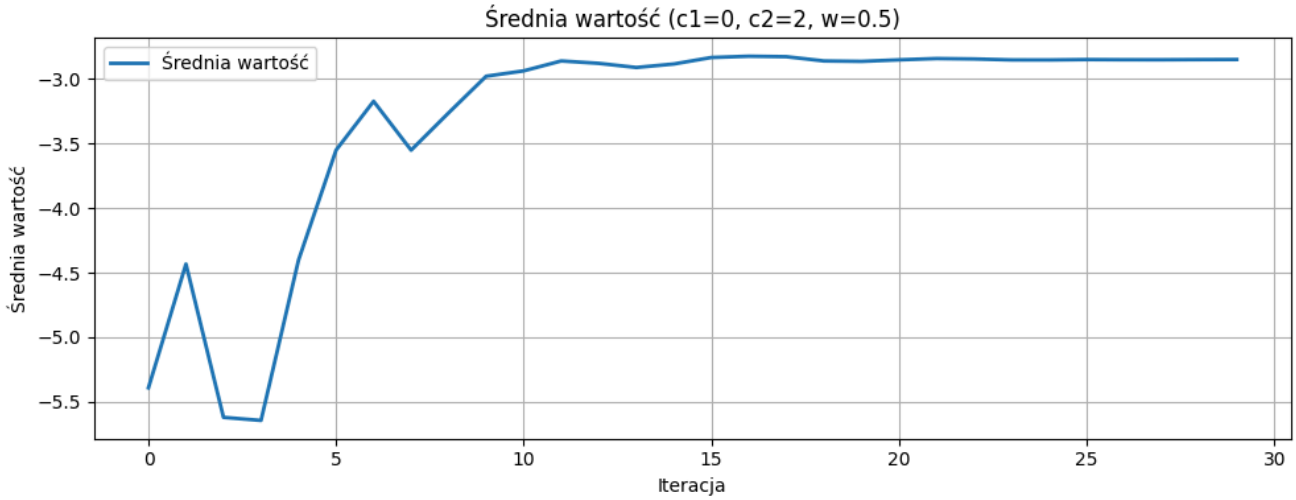
```
!pip install -q plotly
```

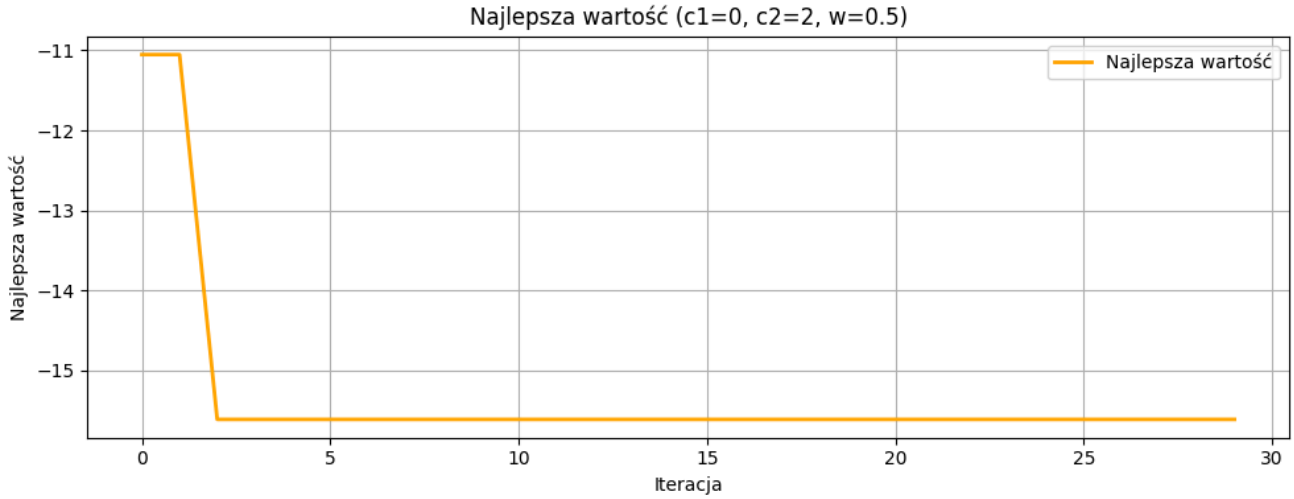
```
import numpy as np
import matplotlib.pyplot as plt
import plotly.graph_objects as go

# funkcja celu
def target_function(pos):
    x, y = pos
    try:
        val = (
            2 * np.log(np.abs(x + 0.2) + 0.002)
            + np.log(np.abs(y + 0.1) + 0.001)
            + np.cos(3 * x)
            + 2 * np.sin(3 * x * y)**2
            + np.sin(y)**2
            - x**2
            - 0.5 * y**2
        )
        return val
    except Exception:
        return np.inf

# algorytm PSO
def pso(cost_func, dim=2, num_particles=30, max_iter=30, w=0.5, c1=1, c2=2):
    particles = np.random.uniform(-1.0, 1.0, (num_particles, dim))
    velocities = np.zeros((num_particles, dim))
    best_positions = np.copy(particles)
    best_fitness = np.array([cost_func(p) for p in particles])
    swarm_best_position = best_positions[np.argmin(best_fitness)]
    swarm_best_fitness = np.min(best_fitness)
    avgs = []
    bests_fitness = []

    for _ in range(max_iter):
```

```
        r1 = np.random.rand(num_particles, dim)
        r2 = np.random.rand(num_particles, dim)
        velocities = (
            w * velocities
            + c1 * r1 * (best_positions - particles)
            + c2 * r2 * (swarm_best_position - particles)
        )
        particles += velocities
        fitness_values = np.array([cost_func(p) for p in particles])
        improved = fitness_values < best_fitness
        best_positions[improved] = particles[improved]
        best_fitness[improved] = fitness_values[improved]
        if np.min(fitness_values) < swarm_best_fitness:
            swarm_best_position = particles[np.argmin(fitness_values)]
            swarm_best_fitness = np.min(fitness_values)
        avgs.append(np.mean(fitness_values))
        bests_fitness.append(swarm_best_fitness)

    return swarm_best_position, swarm_best_fitness, particles, avgs, bests_fitness

# wizualizacja 3D
def plot_plotly_surface(best_pos, best_val, final_particles, title_suffix=""):
    x = np.linspace(-1.0, 1.0, 200)
    y = np.linspace(-1.0, 1.0, 200)
    X, Y = np.meshgrid(x, y)
    Z = np.array([target_function([x_, y_]) for x_, y_ in zip(X.ravel(), Y.ravel())]).reshape(X.shape)

    surface = go.Surface(z=Z, x=X, y=Y, colorscale='Viridis', opacity=0.7, name='f(x, y)')
    particles_scatter = go.Scatter3d(
        x=final_particles[:, 0],
        y=final_particles[:, 1],
        z=[target_function(p) for p in final_particles],
        mode='markers',
        marker=dict(size=3, color='black'),
        name='Cząstki (ostatnia iteracja)'
    )
    best_point = go.Scatter3d(
        x=[best_pos[0]],
        y=[best_pos[1]],
        z=[best_val],
        mode='markers',
        marker=dict(size=6, color='red'),
        name='Najlepsze rozwiązanie'
    )

    fig = go.Figure(data=[surface, particles_scatter, best_point])
    fig.update_layout(
        title=f'Funkcja celu + PSO {title_suffix}',
        scene=dict(
            xaxis_title='x',
            yaxis_title='y',
            zaxis_title='f(x, y)'
        ),
        width=800,
        height=600
    )
    fig.show()

def test_config(c1, c2, w, num_runs=10):
    results = []
    all_runs = []

    print(f"\nTest konfiguracji: c1={c1}, c2={c2}, w={w}")
    for i in range(num_runs):
        solution, fitness, particles, avgs, bests = pso(target_function, c1=c1, c2=c2, w=w)
        results.append(fitness)
        all_runs.append((solution, fitness, particles, avgs, bests))
        print(f"  Uruchomienie {i+1}: fitness = {fitness:.6f}")

    mean_val = np.mean(results)
    std_val = np.std(results)
    print(f"  Średnia wartość: {mean_val:.6f}")
    print(f"  Odchylenie standardowe: {std_val:.6f}")

    # najlepszy przebieg
    best_idx = np.argmin(results)
    best_solution, best_fitness, best_particles, best_avgs, best_bests = all_runs[best_idx]

    # wykresy przebiegów
    plt.figure(figsize=(10, 4))
    plt.plot(best_avgs, label='Średnia wartość', linewidth=2)
    plt.title(f"Średnia wartość (c1={c1}, c2={c2}, w={w})")
    plt.xlabel("Iteracja")
    plt.ylabel("Średnia wartość")
    plt.grid(True)
    plt.legend()
    plt.tight_layout()
```

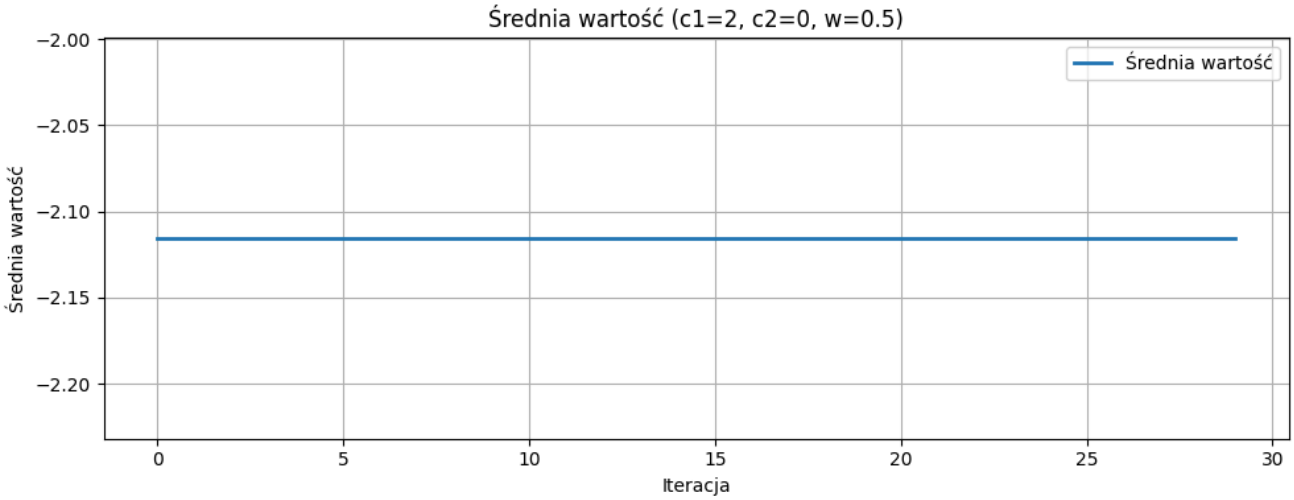
```
    plt.show()

    plt.figure(figsize=(10, 4))
    plt.plot(best_bests, label='Najlepsza wartość', color='orange', linewidth=2)
    plt.title(f"Najlepsza wartość (c1={c1}, c2={c2}, w={w})")
    plt.xlabel("Iteracja")
    plt.ylabel("Najlepsza wartość")
    plt.grid(True)
    plt.legend()
    plt.tight_layout()
    plt.show()

    # wizualizacja 3D
    plot_plotly_surface(best_solution, best_fitness, best_particles, title_suffix=f"(c1={c1}, c2={c2}, w={w})")

# konfiguracje testowe
configs = [
    {"c1": 0, "c2": 2, "w": 0.5},
    {"c1": 2, "c2": 0, "w": 0.5},
    {"c1": 1, "c2": 1, "w": 0.5},
    {"c1": 2.2, "c2": 2.2, "w": 0.5},
    {"c1": 1.5, "c2": 1.5, "w": 0.2},
    {"c1": 1.5, "c2": 1.5, "w": 0.3},
    {"c1": 1.5, "c2": 1.5, "w": 0.4},
    {"c1": 1.5, "c2": 1.5, "w": 0.5},
    {"c1": 1.5, "c2": 1.5, "w": 0.6},
    {"c1": 1.4, "c2": 1.6, "w": 0.2}
]

# testy
for cfg in configs:
    test_config(**cfg)
```

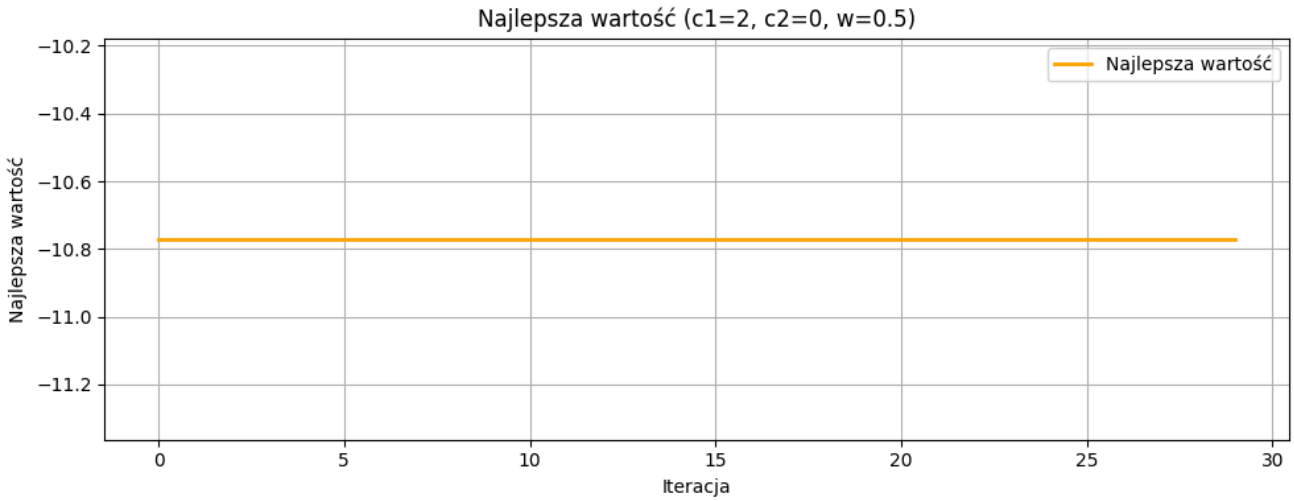
```
Test konfiguracji: c1=0, c2=2, w=0.5
  Uruchomienie 1: fitness = -11.280861
  Uruchomienie 2: fitness = -9.169632
  Uruchomienie 3: fitness = -12.595165
  Uruchomienie 4: fitness = -12.428991
  Uruchomienie 5: fitness = -15.519538
  Uruchomienie 6: fitness = -12.599048
  Uruchomienie 7: fitness = -15.609751
  Uruchomienie 8: fitness = -13.990832
  Uruchomienie 9: fitness = -11.682655
  Uruchomienie 10: fitness = -11.580569
  Średnia wartość: -12.645704
  Odchylenie standardowe: 1.868513
```



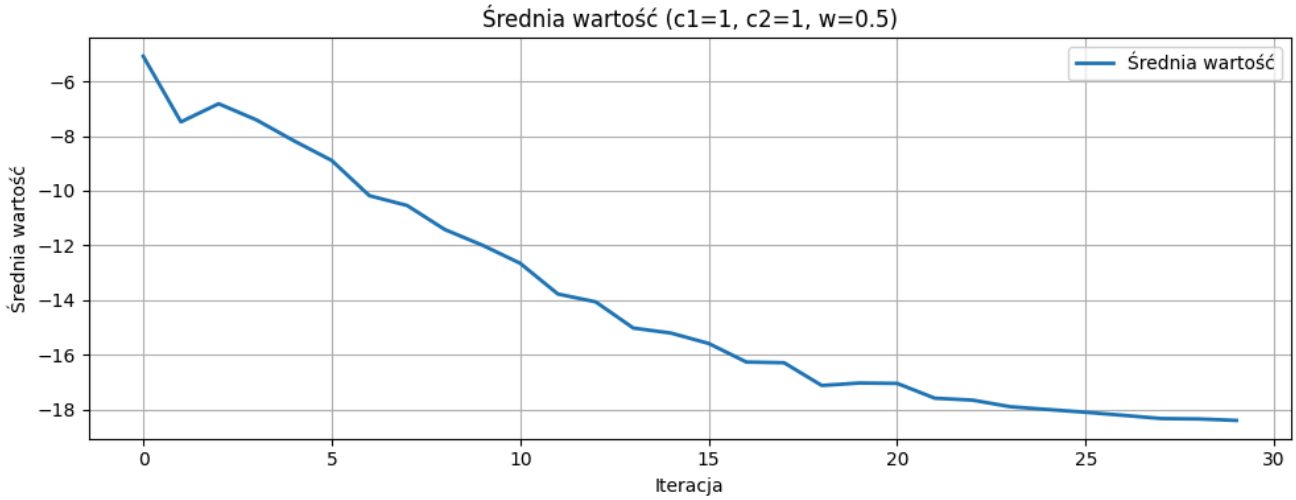


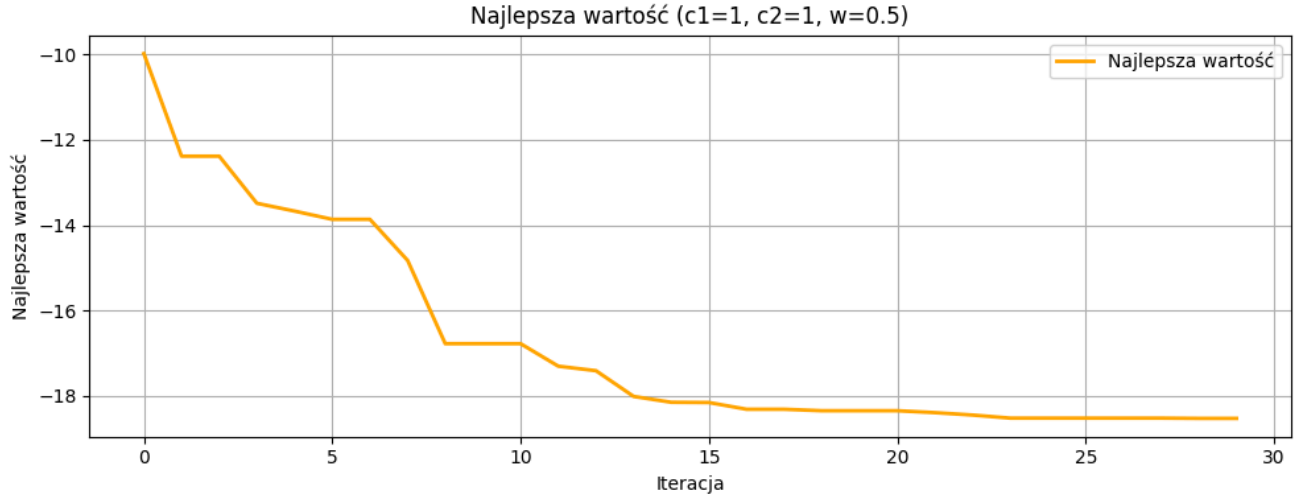

```
Test konfiguracji: c1=2, c2=0, w=0.5
  Uruchomienie 1: fitness = -5.987461
  Uruchomienie 2: fitness = -8.919449
  Uruchomienie 3: fitness = -4.894946
  Uruchomienie 4: fitness = -6.616645
  Uruchomienie 5: fitness = -8.924683
  Uruchomienie 6: fitness = -10.420677
  Uruchomienie 7: fitness = -6.920029
  Uruchomienie 8: fitness = -10.770493
  Uruchomienie 9: fitness = -9.507717
  Uruchomienie 10: fitness = -7.970069
  Średnia wartość: -8.093217
  Odchylenie standardowe: 1.849179
```



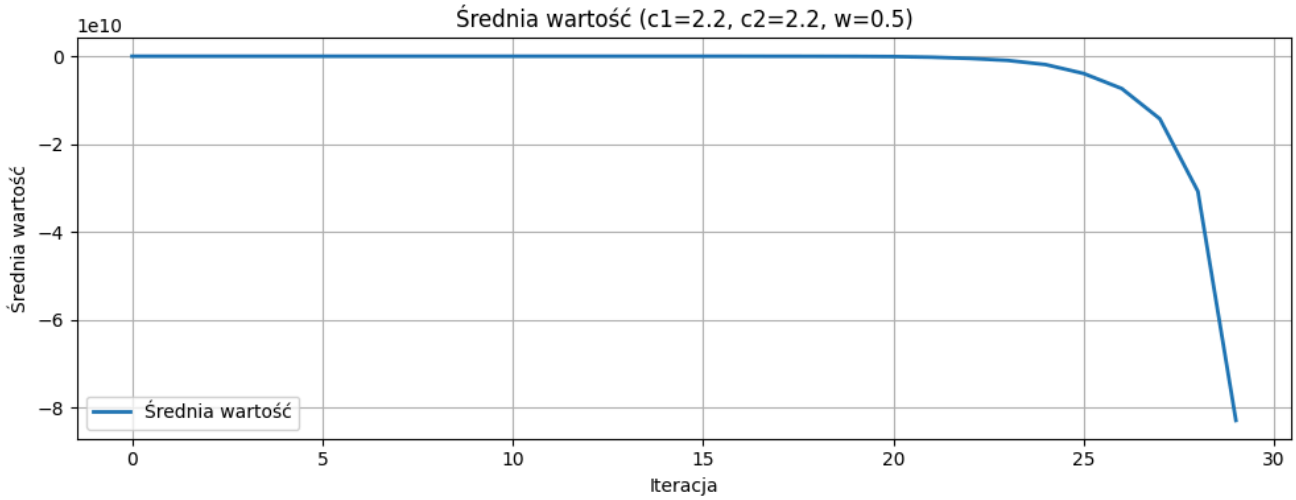


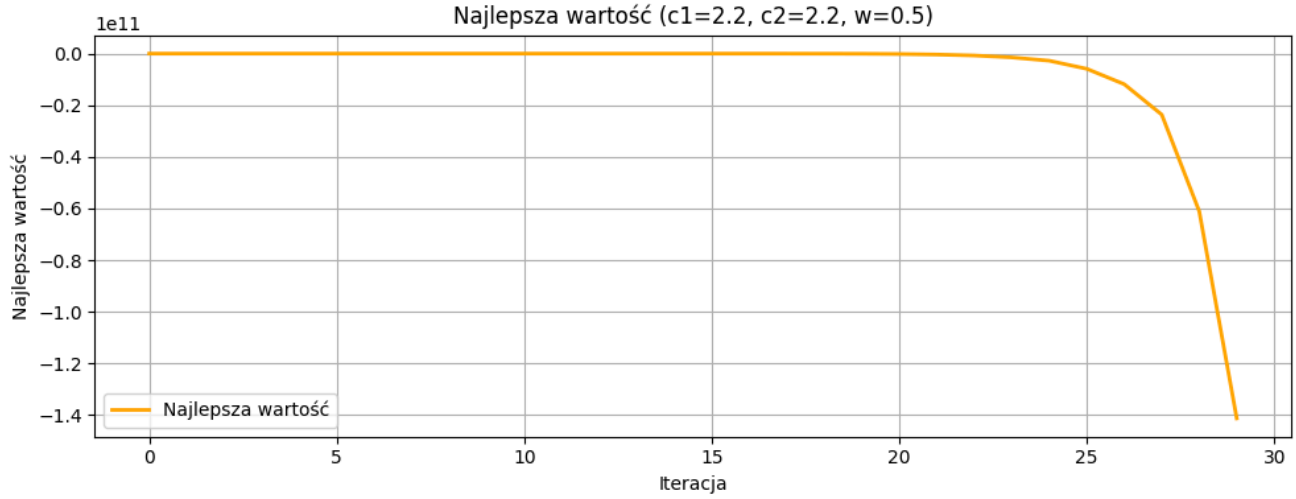
```
Test konfiguracji: c1=1, c2=1, w=0.5
  Uruchomienie 1: fitness = -14.119724
  Uruchomienie 2: fitness = -18.480088
  Uruchomienie 3: fitness = -18.506262
  Uruchomienie 4: fitness = -18.492168
  Uruchomienie 5: fitness = -18.528636
  Uruchomienie 6: fitness = -18.530261
  Uruchomienie 7: fitness = -18.532165
  Uruchomienie 8: fitness = -18.453391
  Uruchomienie 9: fitness = -15.760958
  Uruchomienie 10: fitness = -18.530009
  Średnia wartość: -17.793366
  Odchylenie standardowe: 1.473167
```



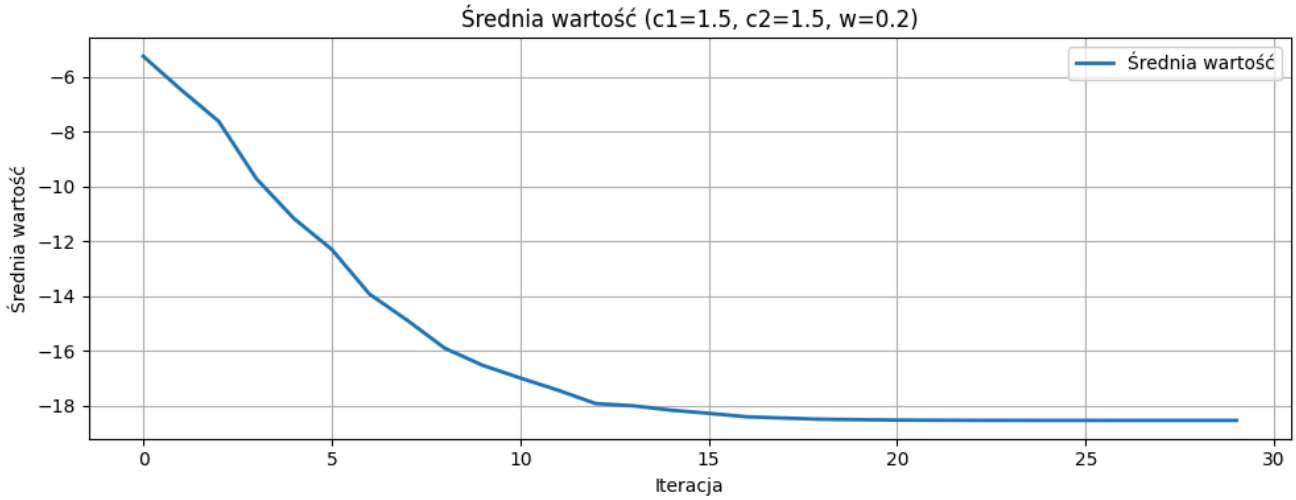


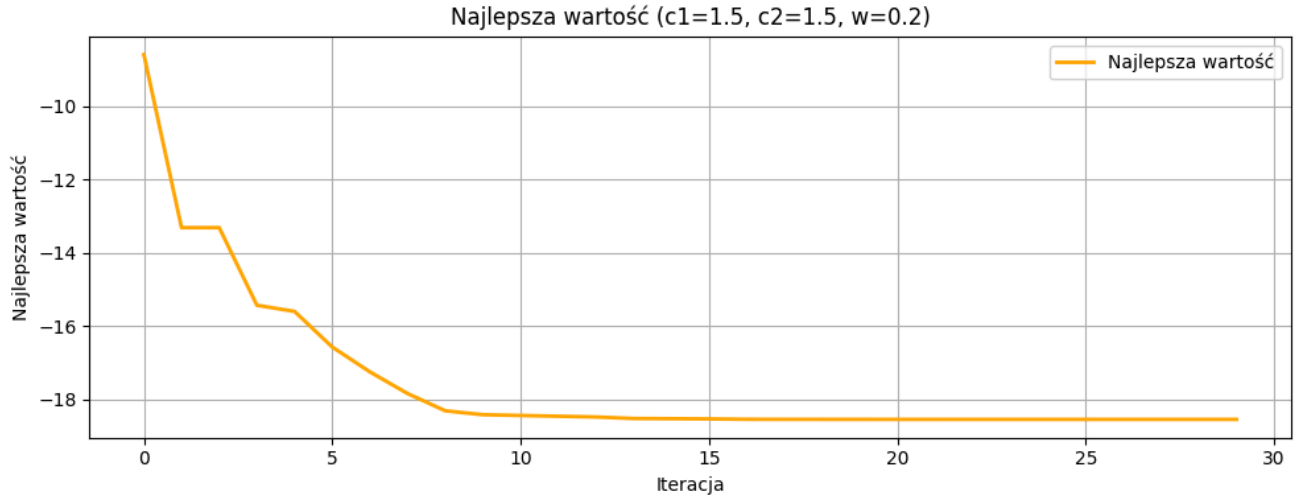

```
Test konfiguracji: c1=2.2, c2=2.2, w=0.5
  Uruchomienie 1: fitness = -612407895.360736
  Uruchomienie 2: fitness = -1318557947.390558
  Uruchomienie 3: fitness = -602640597.117828
  Uruchomienie 4: fitness = -4205908423.346667
  Uruchomienie 5: fitness = -11548984573.182495
  Uruchomienie 6: fitness = -141296926806.297028
  Uruchomienie 7: fitness = -21141344162.625099
  Uruchomienie 8: fitness = -11241428426.250994
  Uruchomienie 9: fitness = -38270562647.243767
  Uruchomienie 10: fitness = -6358877347.216618
  Średnia wartość: -23659763882.603180
  Odchylenie standardowe: 40741772435.836380
```



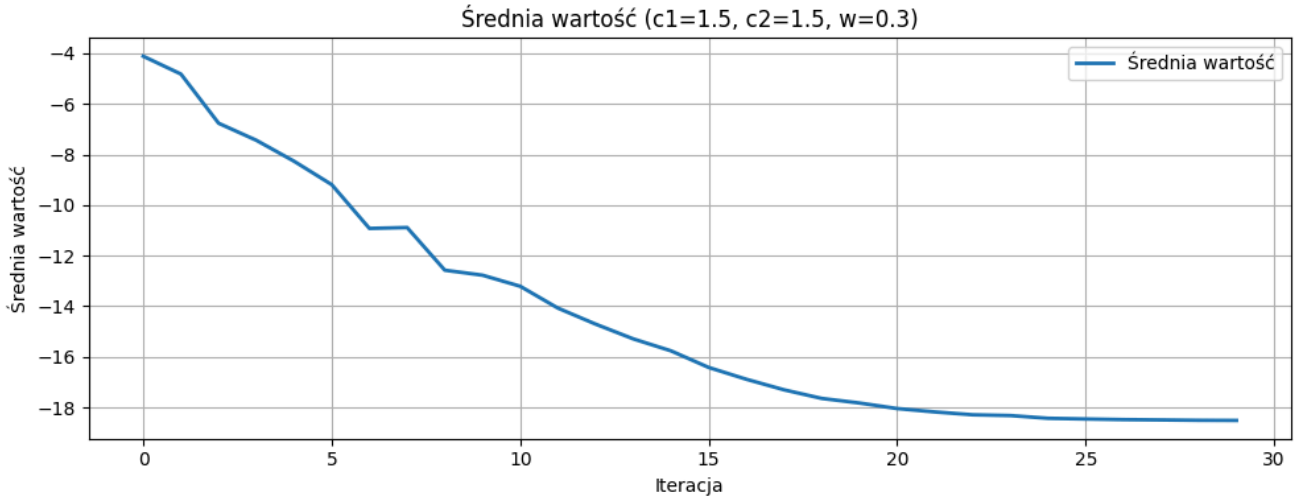


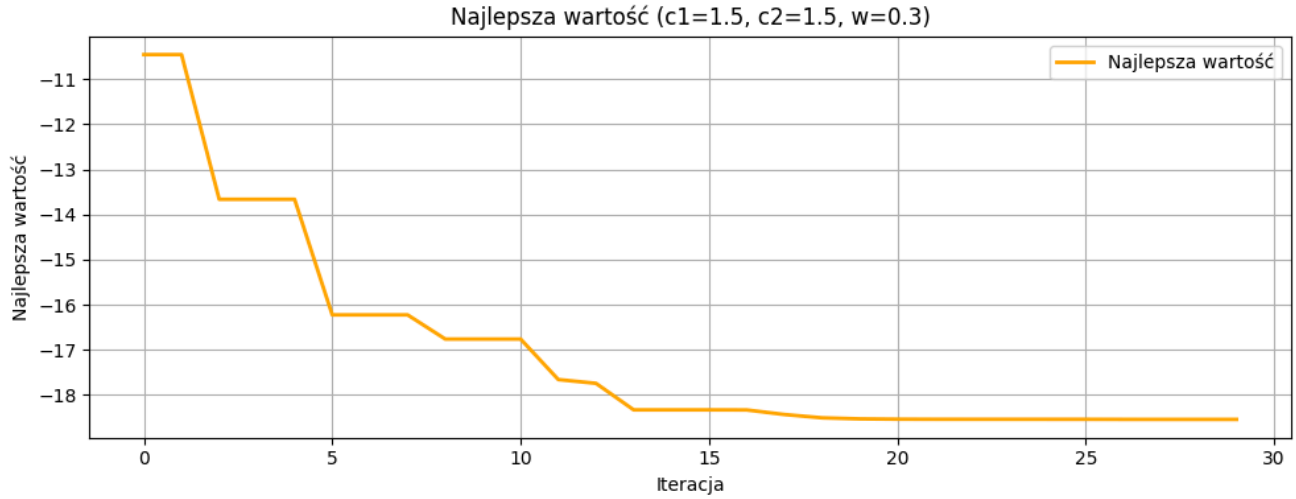

```
Test konfiguracji: c1=1.5, c2=1.5, w=0.2
  Uruchomienie 1: fitness = -18.539477
  Uruchomienie 2: fitness = -18.538918
  Uruchomienie 3: fitness = -18.539456
  Uruchomienie 4: fitness = -18.539222
  Uruchomienie 5: fitness = -18.539471
  Uruchomienie 6: fitness = -18.539421
  Uruchomienie 7: fitness = -18.539385
  Uruchomienie 8: fitness = -18.537218
  Uruchomienie 9: fitness = -18.539289
  Uruchomienie 10: fitness = -18.538579
  Średnia wartość: -18.539044
  Odchylenie standardowe: 0.000668
```



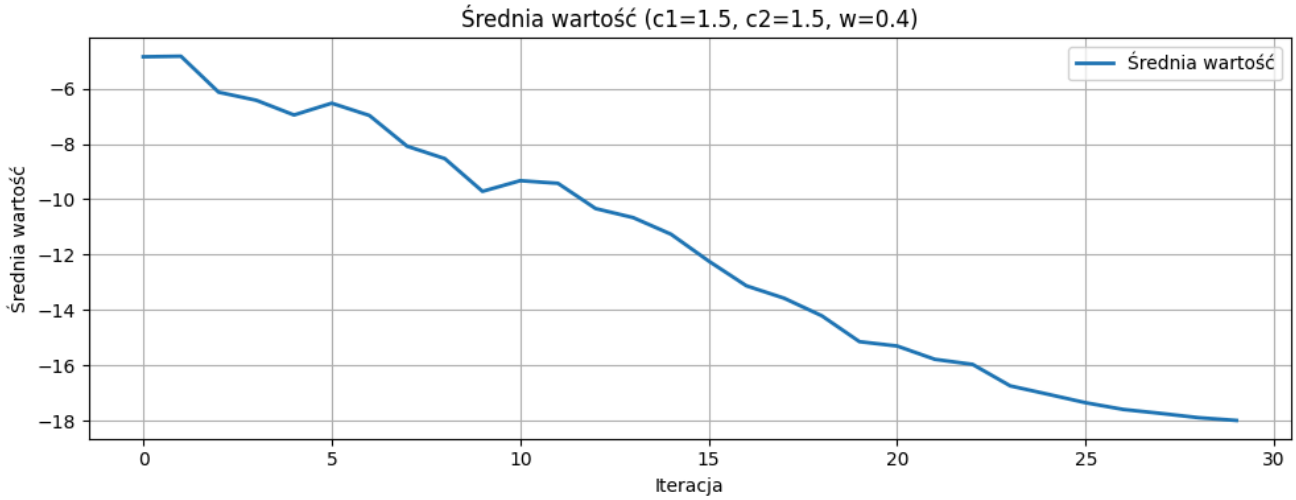


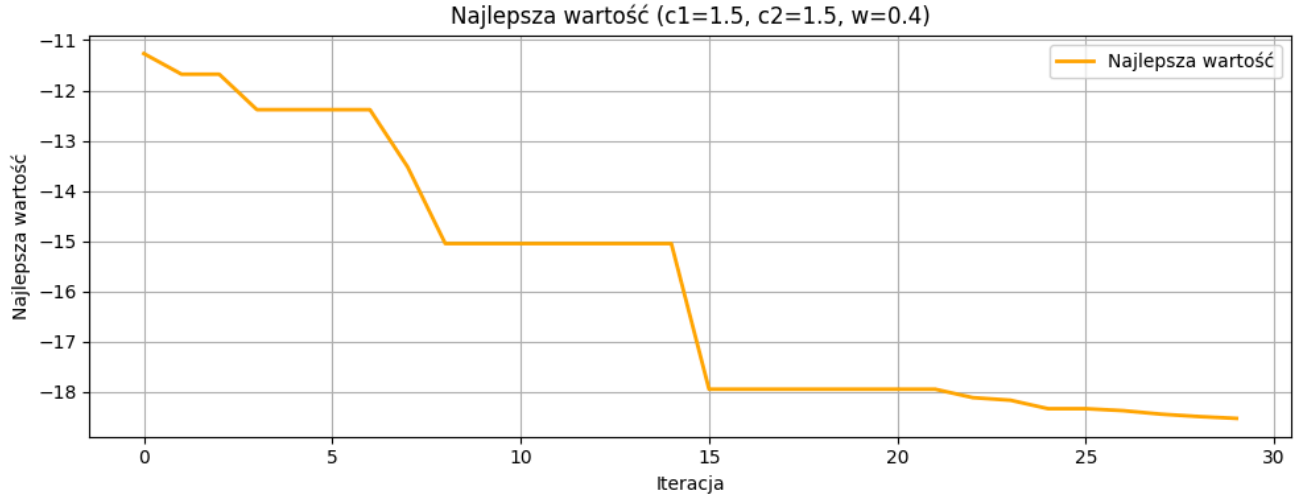

```
Test konfiguracji: c1=1.5, c2=1.5, w=0.3
  Uruchomienie 1: fitness = -18.539066
  Uruchomienie 2: fitness = -18.538628
  Uruchomienie 3: fitness = -18.535466
  Uruchomienie 4: fitness = -18.538363
  Uruchomienie 5: fitness = -18.538443
  Uruchomienie 6: fitness = -18.538343
  Uruchomienie 7: fitness = -18.538439
  Uruchomienie 8: fitness = -18.538791
  Uruchomienie 9: fitness = -18.538112
  Uruchomienie 10: fitness = -18.537847
  Średnia wartość: -18.538150
  Odchylenie standardowe: 0.000950
```



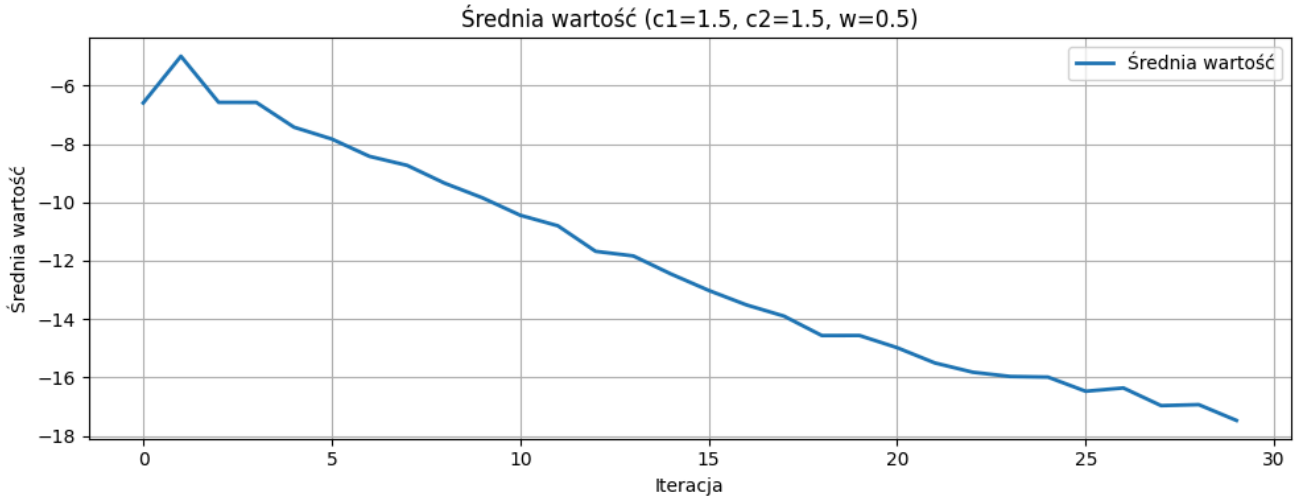


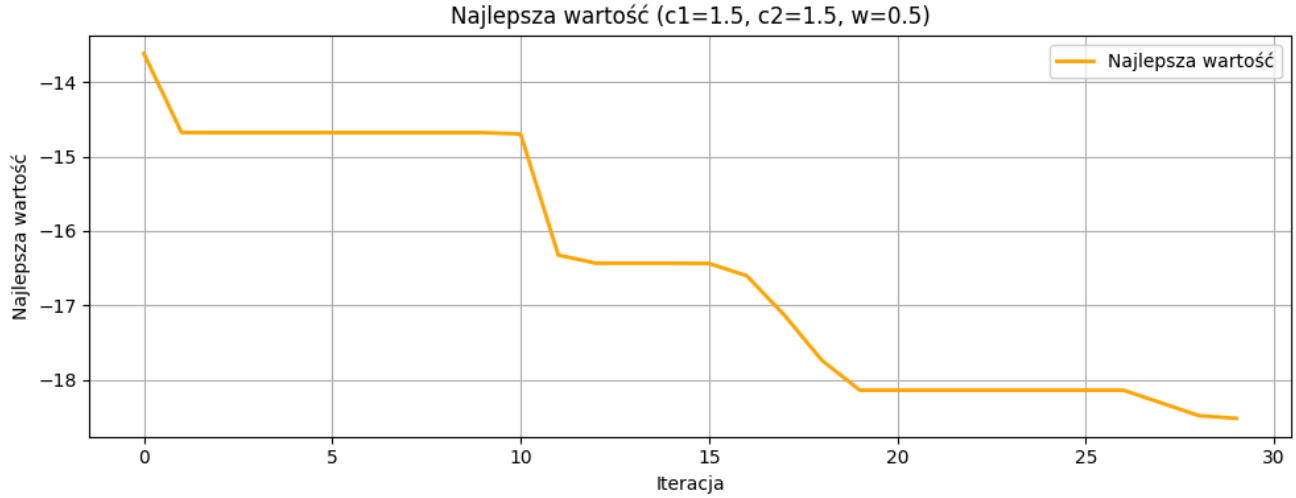
```
Test konfiguracji: c1=1.5, c2=1.5, w=0.4
  Uruchomienie 1: fitness = -18.512758
  Uruchomienie 2: fitness = -18.513733
  Uruchomienie 3: fitness = -18.347488
  Uruchomienie 4: fitness = -18.525904
  Uruchomienie 5: fitness = -18.471846
  Uruchomienie 6: fitness = -18.497997
  Uruchomienie 7: fitness = -18.512868
  Uruchomienie 8: fitness = -18.496429
  Uruchomienie 9: fitness = -18.520131
  Uruchomienie 10: fitness = -18.518182
  Średnia wartość: -18.491734
  Odchylenie standardowe: 0.050309
```



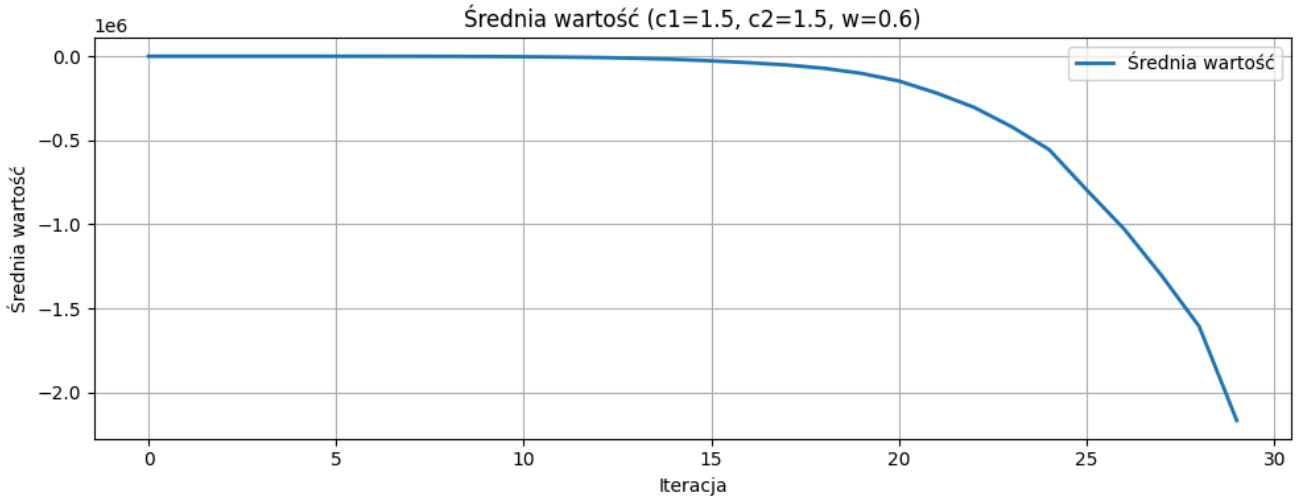


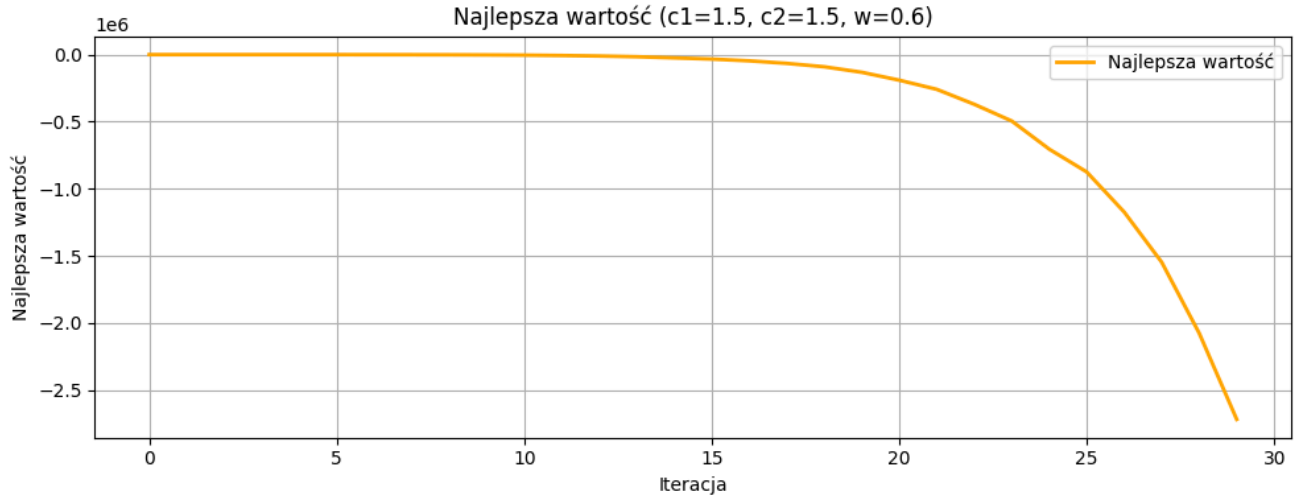

```
Test konfiguracji: c1=1.5, c2=1.5, w=0.5
  Uruchomienie 1: fitness = -18.321739
  Uruchomienie 2: fitness = -18.522471
  Uruchomienie 3: fitness = -18.463997
  Uruchomienie 4: fitness = -18.372472
  Uruchomienie 5: fitness = -18.180454
  Uruchomienie 6: fitness = -18.484909
  Uruchomienie 7: fitness = -18.401814
  Uruchomienie 8: fitness = -18.295806
  Uruchomienie 9: fitness = -18.061819
  Uruchomienie 10: fitness = -18.505909
  Średnia wartość: -18.361139
  Odchylenie standardowe: 0.142383
```



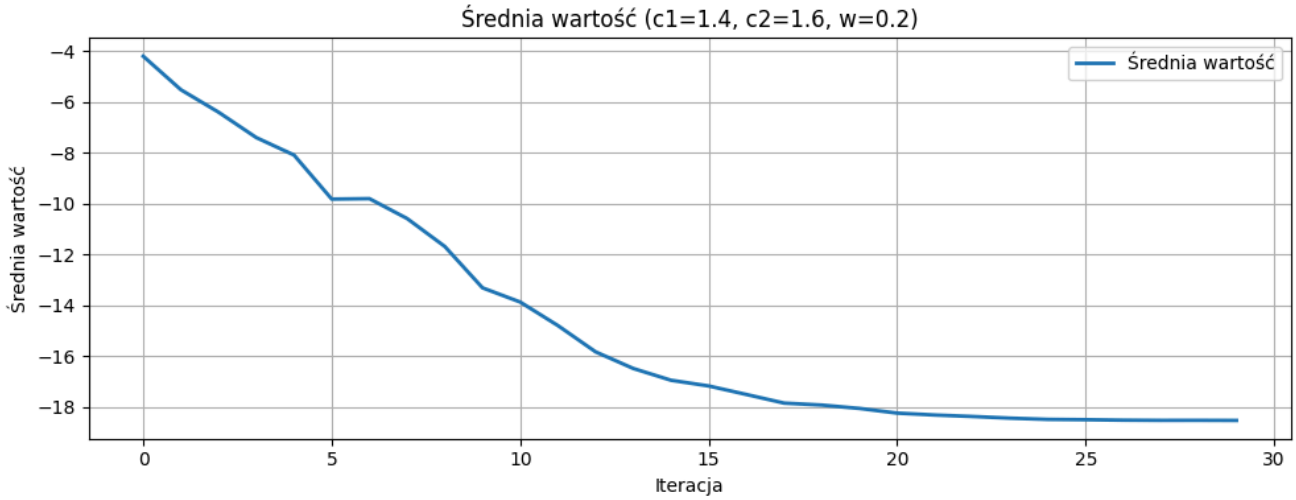


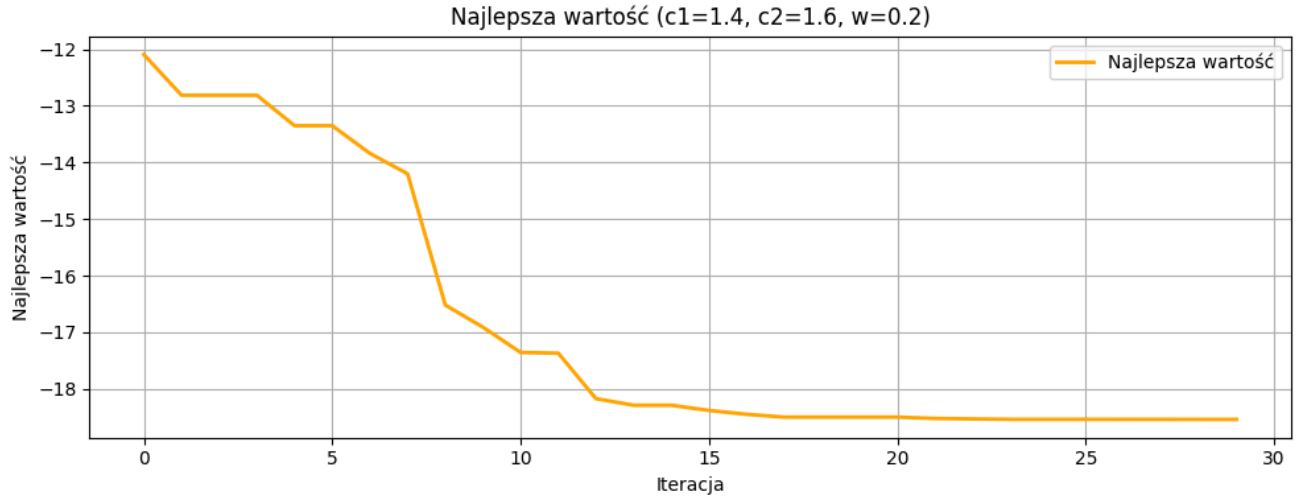

```
Test konfiguracji: c1=1.5, c2=1.5, w=0.6
  Uruchomienie 1: fitness = -2669459.985387
  Uruchomienie 2: fitness = -16.594870
  Uruchomienie 3: fitness = -18.490552
  Uruchomienie 4: fitness = -17.797694
  Uruchomienie 5: fitness = -16.413287
  Uruchomienie 6: fitness = -17.572035
  Uruchomienie 7: fitness = -17.509443
  Uruchomienie 8: fitness = -17.755770
  Uruchomienie 9: fitness = -17.248985
  Uruchomienie 10: fitness = -2719774.828521
  Średnia wartość: -538937.419654
  Odchylenie standardowe: 1077898.710957
```






```
Test konfiguracji: c1=1.4, c2=1.6, w=0.2
  Uruchomienie 1: fitness = -18.539299
  Uruchomienie 2: fitness = -18.538845
  Uruchomienie 3: fitness = -18.539455
  Uruchomienie 4: fitness = -18.539241
  Uruchomienie 5: fitness = -18.539375
  Uruchomienie 6: fitness = -18.538633
  Uruchomienie 7: fitness = -18.539373
  Uruchomienie 8: fitness = -18.539458
  Uruchomienie 9: fitness = -18.539391
  Uruchomienie 10: fitness = -18.539470
  Średnia wartość: -18.539254
  Odchylenie standardowe: 0.000270
```





Wnioski:

- Dla parametrów c1=0 i c2=2 algorytm skupia się na ulepszaniu obecnych rozwiązań, ignorując eksplorację.
- Dla c1=2 i c2=0, algorytm skupia się na eksploracji, ignorując ulepszanie obecnych rozwiązań. Skutkuje to płaskim wykresem przebiegu - najlepsza wartość znaleziona na początku nie jest ekploatowana i nie ulega zmianie
- Dla wartości c1 = c2 = 2.2 algorytm staje się niestabilny
- Dla skrajnych wartości parametrów algorytm daje niepoprawne wyniki
- Wartości c1 i c2 znajdujące się w okolicy wartości 1.5 zdają się dawać najlepsze wyniki
- Wartość w kontroluje wpływ dotychczasowej prędkości cząstki na jej nową prędkość, wpływa więc na balans między eksploracją a eksploatacją
- Niskie wartości parametru w (skupienie na eksploatacji) dają najlepsze wyniki dla tej funkcji celu
- Wysokie wartości parametru w prowadzą do destabilizacji (duże odchylenie standardowe) i błędnych wyników
- Najlepsze wyniki algorytm osiągnął dla parametrów c1=1.4, c2=1.6, w=0.2 i wyniosły:
  - Średnia wartość: -18.539254
  - Odchylenie standardowe: 0.000270

Algorytm najlepiej działa przy dużym skupieniu się na eksploatacji najlepszych rozwiązań, przy jednoczesnym pozostawieniu pola na eksplorację

- Algorytm dobrze poradził sobie ze znalezieniem minimum funkcji

Zadanie 2

Uzyskany najlepszy wynik proszę w miarę możliwości porównać z algorytmem genetycznym – dedykowanym dla optymalizacji tej samej funkcji. Zakładamy użycie takiej samej liczby epok dla obu algorytmów. Należy porównać czas działania obydwu algorytmów.

```
import numpy as np
import random
import matplotlib.pyplot as plt
from time import perf_counter

# funkcja celu
def target_function(pos):
    x, y = pos
    try:
        val = (
            2 * np.log(np.abs(x + 0.2) + 0.002)
            + np.log(np.abs(y + 0.1) + 0.001)
            + np.cos(3 * x)
            + 2 * np.sin(3 * x * y)**2
            + np.sin(y)**2
            - x**2
            - 0.5 * y**2
        )
        return val
    except Exception:
        return np.inf

def run_pso_experiment(runs=10, c1=1.5, c2=1.5, w=0.2, iterations=30):
    best_results = []
    times = []

    for _ in range(runs):
        start = perf_counter()
```

```
        pos, fit, _, _, _ = pso(target_function, dim=2, c1=c1, c2=c2, w=w, max_iter=iterations)
        end = perf_counter()
        best_results.append(fit)
        times.append(end - start)

    best_val = np.min(best_results)
    avg_val = np.mean(best_results)
    avg_time = np.mean(times)

    print("PSO")
    print(f"Best fitness: {best_val:.6f}")
    print(f"Average fitness: {avg_val:.6f}")
    print(f"Average execution time: {avg_time:.4f} s")

    return best_val, avg_val, avg_time


# algorytm genetyczny
class GA_Solution2D:
    def __init__(self, bit_length=16, randomize=True):
        self.bit_length = bit_length
        self.genes = [random.randint(0, 1) for _ in range(bit_length * 2)] if randomize else [0] * (bit_length * 2)

    def decode(self):
        def decode_chunk(chunk):
            number = int("".join(str(b) for b in chunk), 2)
            return -1 + (number / (2**len(chunk) - 1)) * 2

        x_bits = self.genes[:self.bit_length]
        y_bits = self.genes[self.bit_length:]
        return decode_chunk(x_bits), decode_chunk(y_bits)

    def fitness(self):
        x, y = self.decode()
        return target_function([x, y])

    def crossover(self, other):
        point = random.randint(1, len(self.genes) - 1)
        child = GA_Solution2D(bit_length=self.bit_length, randomize=False)
        child.genes = self.genes[:point] + other.genes[point:]
        return child

    def mutate(self, mutation_rate):
        for i in range(len(self.genes)):
            if random.random() < mutation_rate:
                self.genes[i] ^= 1

# wariant selekcja progowa
def run_genetic_algorithm_2d(pop_size=50, iterations=30, mutation_rate=0.05, gamma=30, runs=10):
    best_vals = []
    times = []

    for _ in range(runs):
        population = [GA_Solution2D() for _ in range(pop_size)]
        best_global = None
        best_fitness = float('inf')
        start = perf_counter()

        for _ in range(iterations):
            adaptations = [ind.fitness() for ind in population]
            elite_count = int(gamma / 100 * pop_size)
            elites = [x for _, x in sorted(zip(adaptations, population), key=lambda x: x[0])[:elite_count]]

            children = []
            while len(children) < pop_size:
                p1, p2 = random.choices(elites, k=2)
                child = p1.crossover(p2)
                child.mutate(mutation_rate)
                children.append(child)

            population = children
            local_best = min(population, key=lambda ind: ind.fitness())
            if local_best.fitness() < best_fitness:
                best_fitness = local_best.fitness()
                best_global = local_best

        end = perf_counter()
        best_vals.append(best_fitness)
        times.append(end - start)

    print("\nGENETIC ALGORITHM")
    print(f"Best fitness: {np.min(best_vals):.6f}")
    print(f"Average fitness: {np.mean(best_vals):.6f}")
    print(f"Average execution time: {np.mean(times):.4f} s")

    return np.min(best_vals), np.mean(best_vals), np.mean(times)
```

```python
# wariant selekcja ruletkowa
def run_genetic_algorithm_2d_roulette(pop_size=50, iterations=30, mutation_rate=0.05, runs=10):
    best_vals = []
    times = []

    for _ in range(runs):
        population = [GA_Solution2D() for _ in range(pop_size)]
        best_global = None
        best_fitness = float('inf')
        start = perf_counter()

        for _ in range(iterations):
            adaptations = np.array([ind.fitness() for ind in population])

            epsilon = 1e-10
            fitness_values = 1 / (adaptations - np.min(adaptations) + epsilon)
            probabilities = fitness_values / np.sum(fitness_values)

            children = []
            while len(children) < pop_size:
                p1, p2 = np.random.choice(population, size=2, p=probabilities, replace=True)
                child = p1.crossover(p2)
                child.mutate(mutation_rate)
                children.append(child)

            population = children
            local_best = min(population, key=lambda ind: ind.fitness())
            if local_best.fitness() < best_fitness:
                best_fitness = local_best.fitness()
                best_global = local_best

        end = perf_counter()
        best_vals.append(best_fitness)
        times.append(end - start)

    print("\nGENETIC ALGORITHM (Roulette Selection)")
    print(f"Best fitness: {np.min(best_vals):.6f}")
    print(f"Average fitness: {np.mean(best_vals):.6f}")
    print(f"Average execution time: {np.mean(times):.4f} s")

    return np.min(best_vals), np.mean(best_vals), np.mean(times)


# uruchomienie i porównanie wyników
pso_best, pso_avg, pso_time = run_pso_experiment(runs=10, c1=1.4, c2=1.6, w=0.2, iterations=30)
ga_best, ga_avg, ga_time = run_genetic_algorithm_2d(pop_size=50, iterations=30, mutation_rate=0.05, gamma=20, runs=10)
ga_best_r, ga_avg_r, ga_time_r = run_genetic_algorithm_2d_roulette(pop_size=50, iterations=30, mutation_rate=0.05, runs=10)


print("\n\nPORÓWNANIE")
print(f"PSO: Best = {pso_best:.6f}, Avg = {pso_avg:.6f}, Time = {pso_time:.4f}s")
print(f"GA GAMMA: Best = {ga_best:.6f}, Avg = {ga_avg:.6f}, Time = {ga_time:.4f}s")
print(f"GA ROULETTE: Best = {ga_best_r:.6f}, Avg = {ga_avg_r:.6f}, Time = {ga_time_r:.4f}s")
```

```
PSO
Best fitness: -18.539476
Average fitness: -18.539432
Average execution time: 0.0226 s

GENETIC ALGORITHM
Best fitness: -18.531879
Average fitness: -17.627004
Average execution time: 0.0900 s

GENETIC ALGORITHM (Roulette Selection)
Best fitness: -17.530115
Average fitness: -14.747642
Average execution time: 0.3077 s


PORÓWNANIE
PSO: Best = -18.539476, Avg = -18.539432, Time = 0.0226s
GA GAMMA: Best = -18.531879, Avg = -17.627004, Time = 0.0900s
GA ROULETTE: Best = -17.530115, Avg = -14.747642, Time = 0.3077s
```

Wnioski:

- Zarówno algorytm PSO, jak i algorytm genetyczny dobrze poradził sobie ze znalezieniem minimum funkcji celu
- Czas działania PSO jest dużo krótszy od czasu działania algorytmu genetycznego (kilka razy krótszy)
- Użycie PSO skutkuje otrzymaniem lepszych wyników niż w przypadku użycia GA
- Algorytm genetyczny działa dużo krócej i daje dużo lepsze wyniki dla wariantu z selekcją progową

Zadania dla chętnych

3*.

Proszę obejrzeć jak działa algorytm w trybie animacji dla obydwu wspomnianych w notatniku funkcji.

http://www.alife.pl/files/opt/d/OptiVisJS/OptiVisJS.html?lang=pl