Metaheurystyki — zadanie 5

Algorytm roju cząstek (PSO)

GRUPA 3 — piątek 10:15

Bartosz Kołaciński
251554

Nikodem Nowak
251598

6 stycznia 2026

Użyte technologie	Python 3.13
Użyte biblioteki	numpy, scipy, pandas, matplotlib.pyplot

Spis treści

1 Opis zasad działania algorytmu

1.1 Opis problemu

Problem polega na znalezieniu minimum globalnego funkcji dwóch zmiennych $f(x, y)$. Jest to klasyczny problem optymalizacji ciągłej, gdzie celem jest znalezienie takiego punktu (x^*, y^*) w zadanej dziedzinie, dla którego wartość funkcji jest najmniejsza.

W ramach zadania wybrano dwie funkcje testowe:

- **Funkcja Himmelblau** — posiada 4 minima globalne o wartości 0
- **Funkcja Beale** — posiada 1 minimum globalne o wartości 0

1.2 Opis algorytmu PSO

Algorytm roju cząstek (ang. *Particle Swarm Optimization*, PSO) to metaheurystyka inspirowana zachowaniem stadnym ptaków i ławic ryb. Każda cząstka w roju reprezentuje potencjalne rozwiązanie problemu i porusza się w przestrzeni poszukiwań, kierując się własnym doświadczeniem oraz informacją od pozostałych cząstek.

1.2.1 Pseudokod algorytmu

Algorithm 1 Algorytm roju cząstek (PSO)

Inicjalizacja:
Zainicjuj pozycje cząstek x_i losowo w przestrzeni poszukiwań
Zainicjuj prędkości cząstek $v_i \leftarrow 0$
Dla każdej cząstki: $p_{best,i} \leftarrow x_i$ (najlepsza pozycja lokalna)
$g_{best} \leftarrow$ najlepsza pozycja spośród wszystkich cząstek (globalna)
for $t = 1$ **to** T (liczba iteracji) **do**
  for każda cząstka $i = 1, \ldots, N$ **do**
    Aktualizacja prędkości:
    $v_i \leftarrow w \cdot v_i + c_1 \cdot r_1 \cdot (p_{best,i} - x_i) + c_2 \cdot r_2 \cdot (g_{best} - x_i)$
    Aktualizacja pozycji:
    $x_i \leftarrow x_i + v_i$
    Ogranicz x_i do granic przestrzeni poszukiwań
    Ewaluacja:
    Oblicz wartość funkcji celu $f(x_i)$
    if $f(x_i) < f(p_{best,i})$ **then**
      $p_{best,i} \leftarrow x_i$
    end if
    if $f(x_i) < f(g_{best})$ **then**
      $g_{best} \leftarrow x_i$
    end if
  end for
end for
return g_{best} (najlepsza znaleziona pozycja)

1.2.2 Składniki algorytmu

- **Pozycja cząstki (x_i)** — aktualna lokalizacja cząstki w przestrzeni poszukiwań, reprezentująca potencjalne rozwiązanie problemu.
- **Prędkość cząstki (v_i)** — wektor określający kierunek i szybkość ruchu cząstki w następnej iteracji.
- **Najlepsza pozycja lokalna (p_{best})** — najlepsza pozycja znaleziona przez daną cząstkę w całej historii jej ruchu. Reprezentuje "pamięć" cząstki.

- **Najlepsza pozycja globalna** (g_{best}) — najlepsza pozycja znaleziona przez którąkolwiek cząstkę w całym roju. Reprezentuje "wiedzę zbiorową" roju.
- **Składowa inercyjna** ($w \cdot v_i$) — utrzymuje dotychczasowy kierunek ruchu cząstki. Wysoka wartość w sprzyja eksploracji, niska — eksploatacji.
- **Składowa kognitywna** ($c_1 \cdot r_1 \cdot (p_{best} - x)$) — przyciąga cząstkę do jej własnej najlepszej pozycji. Reprezentuje "indywidualne doświadczenie".
- **Składowa socjalna** ($c_2 \cdot r_2 \cdot (g_{best} - x)$) — przyciąga cząstkę do najlepszej pozycji w roju. Reprezentuje "wpływ społeczny".

1.2.3 Parametry algorytmu

Parametr	**Opis**
N (swarm_size)	Liczba cząstek w roju
T (iterations)	Liczba iteracji algorytmu
w	Współczynnik inercji — kontroluje wpływ poprzedniej prędkości
c_1	Współczynnik kognitywny — siła przyciągania do własnego najlepszego
c_2	Współczynnik socjalny — siła przyciągania do globalnego najlepszego
r_1, r_2	Liczby losowe z przedziału $[0, 1]$ — wprowadzają element stochastyczny

2 Opis implementacji rozwiązania

2.1 Inicjalizacja pozycji cząstek

Pozycje początkowe cząstek są generowane za pomocą metody Latin Hypercube Sampling (LHS), która zapewnia równomierne pokrycie przestrzeni poszukiwań.

```
from scipy.stats import qmc

sampler = qmc.LatinHypercube(d=self._dim)
samples = sampler.random(n=swarm_size)

self.positions = qmc.scale(samples, lower_bounds, upper_bounds)

self.velocities = np.zeros((swarm_size, self._dim))

self.p_best_positions = self.positions.copy()
self.p_best_fitness = np.full(swarm_size, -np.inf)

self.g_best_position = np.zeros(self._dim)
self.g_best_fitness = -float("inf")

```

Kod 1: Inicjalizacja pozycji metodą LHS

2.2 Aktualizacja prędkości

Prędkość każdej cząstki jest aktualizowana zgodnie ze wzorem PSO, uwzględniając składową inercyjną, kognitywną i socjalną.

```
def _update_velocity(self) -> None:
    if self._use_randomness:
        r1 = np.random.rand(*self.positions.shape)
        r2 = np.random.rand(*self.positions.shape)
    else:
        r1 = 1.0
        r2 = 1.0

    self.velocities = (
        self._w * self.velocities
        + self._c1 * r1 * (self.p_best_positions - self.positions)
        + self._c2 * r2 * (self.g_best_position - self.positions)
    )

```

Kod 2: Aktualizacja prędkości cząstek

2.3 Aktualizacja pozycji

Pozycje cząstek są aktualizowane przez dodanie wektora prędkości, a następnie ograniczane do granic przestrzeni poszukiwań.

```
def _update_position(self) -> None:
    self.positions += self.velocities
    np.clip(
        self.positions,
        self._bounds[:, 0],
        self._bounds[:, 1],
        out=self.positions,
    )

```

Kod 3: Aktualizacja pozycji z ograniczeniem do granic

2.4 Ewaluacja i aktualizacja najlepszych pozycji

Po każdej aktualizacji pozycji, obliczana jest wartość funkcji celu i aktualizowane są najlepsze pozycje lokalne oraz globalna.

```python
def _evaluate(self) -> None:
    current_fitness = self._func(self.positions)

    mask = current_fitness > self.p_best_fitness
    self.p_best_positions[mask] = self.positions[mask]
    self.p_best_fitness[mask] = current_fitness[mask]

    current_best_idx = np.argmax(self.p_best_fitness)
    if self.p_best_fitness[current_best_idx] > self.g_best_fitness:
        self.g_best_fitness = self.p_best_fitness[current_best_idx]
        self.g_best_position = self.p_best_positions[current_best_idx].copy()

```

Kod 4: Ewaluacja i aktualizacja p_best oraz g_best

2.5 Główna pętla algorytmu

```python
def run(self, iterations: int) -> tuple[NDArray, float]:
    self._record_convergence()

    for _ in range(iterations):
        self._update_velocity()
        self._update_position()
        self._evaluate()
        self._record_convergence()

    return self.g_best_position, self.g_best_fitness

```

Kod 5: Główna pętla algorytmu PSO

3 Opis wybranych funkcji testowych

3.1 Funkcja Himmelblau

Funkcja Himmelblau jest popularną funkcją testową w optymalizacji, charakteryzującą się czterema minimami globalnymi.

Wzór:

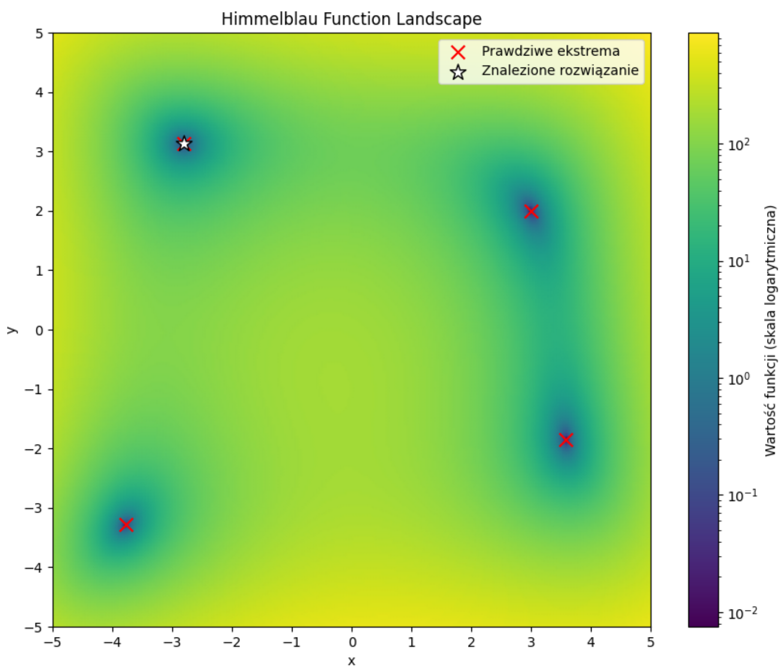
$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2 \tag{1}$$

Dziedzina: $-5 \leq x, y \leq 5$
Minima globalne: Funkcja posiada 4 minima globalne o wartości $f = 0$:

- $(3.0, 2.0)$
- $(-2.805118, 3.131312)$
- $(-3.779310, -3.283186)$
- $(3.584428, -1.848126)$

```
def himmelblau_function_batch(args: NDArray) -> NDArray:
    xs = args[:, 0]
    ys = args[:, 1]
    part_1 = xs * xs + ys - 11
    part_2 = xs + ys * ys - 7
    return part_1 * part_1 + part_2 * part_2
```

Kod 6: Implementacja funkcji Himmelblau



Rysunek 1: Mapa cieplna funkcji Himmelblau. Ciemniejsze obszary oznaczają niższe wartości funkcji. Czerwone krzyżyki wskazują położenie czterech minimów globalnych.

3.2 Funkcja Beale

Funkcja Beale jest funkcją testową z jednym minimum globalnym, charakteryzującą się płaskim dnem doliny, co utrudnia precyzyjne znalezienie optimum.

Wzór:

$$f(x, y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2 \tag{2}$$

Dziedzina: $-4.5 \leq x, y \leq 4.5$

Minimum globalne: $f(3, 0.5) = 0$

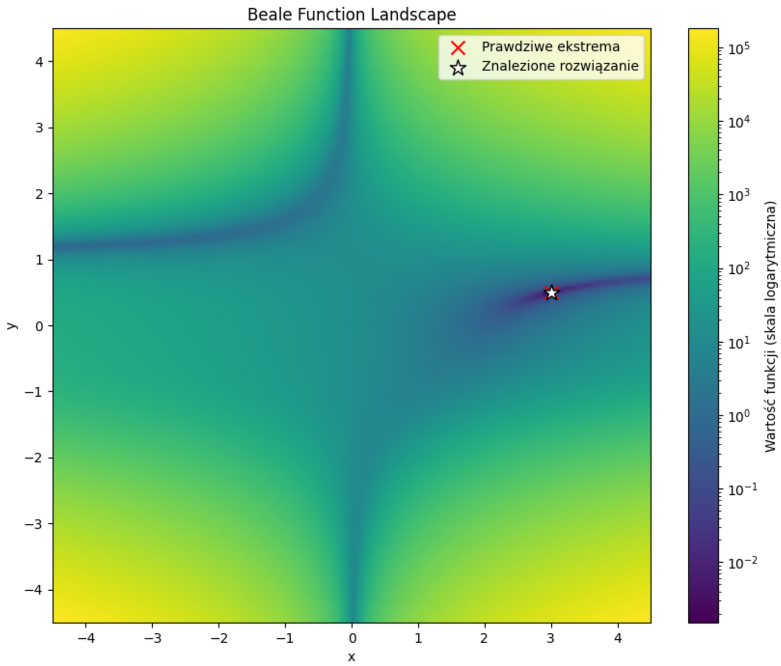
```python
def beale_function_batch(args: NDArray) -> NDArray:
    xs = args[:, 0]
    ys = args[:, 1]

    prod_xy = xs * ys
    prod_xy2 = prod_xy * ys
    prod_xy3 = prod_xy2 * ys

    part_1 = 1.5 - xs + prod_xy
    part_2 = 2.25 - xs + prod_xy2
    part_3 = 2.625 - xs + prod_xy3
    return part_1 * part_1 + part_2 * part_2 + part_3 * part_3

```

Kod 7: Implementacja funkcji Beale



Rysunek 2: Mapa cieplna funkcji Beale. Czerwony krzyżyk wskazuje położenie jedynego minimum globalnego w punkcie $(3, 0.5)$.

4 Instrukcja uruchomienia programu

Program uruchamia się poprzez wywołanie pliku `run_pso.py` w katalogu `src/`:

```
python run_pso.py
```

Skrypt ten wykonuje pojedyncze uruchomienie algorytmu PSO dla obu funkcji testowych z domyślnymi parametrami i generuje mapy cieplne z zaznaczonymi znalezionymi rozwiązaniami.

W celu wygenerowania wszystkich eksperymentów opisanych w dalszej części sprawozdania, należy uruchomić skrypt:

```
python all_experiments.py
```

Skrypt ten wykonuje:

1. Badanie wpływu każdego parametru (w, c1, c2, swarm_size, iterations)
2. Analizę zbieżności dla różnych konfiguracji
3. Wizualizację najlepszych znalezionych rozwiązań
4. Zapis wyników do plików CSV i wykresów PNG

Plan eksperymentalny:

- Każda konfiguracja parametrów jest uruchamiana **15 razy** (N_RUNS = 15)
- Dla każdego parametru badane są różne wartości przy stałych pozostałych parametrach bazowych
- Parametry bazowe: swarm_size=50, iterations=100, w=0.7, c1=1.5, c2=1.5

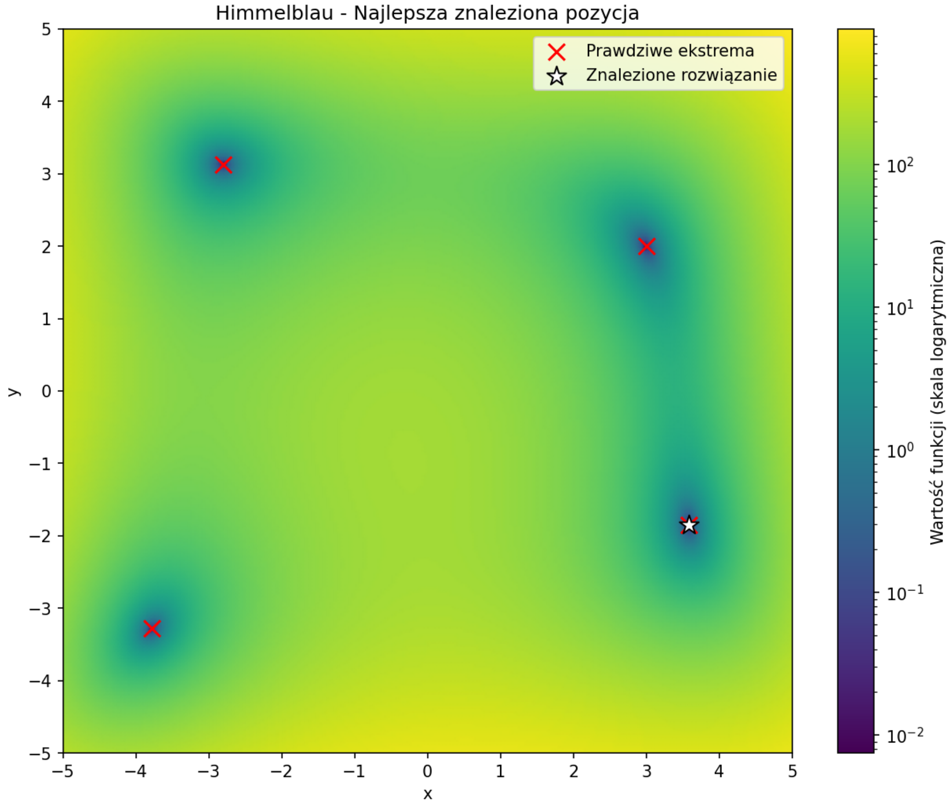
Wyniki zapisywane są w katalogach:

- `../results/Himmelblau/` — pliki CSV z wynikami dla funkcji Himmelblau
- `../results/Beale/` — pliki CSV z wynikami dla funkcji Beale
- `../plots/Himmelblau/` — wykresy dla funkcji Himmelblau
- `../plots/Beale/` — wykresy dla funkcji Beale

5 Eksperymenty i wyniki

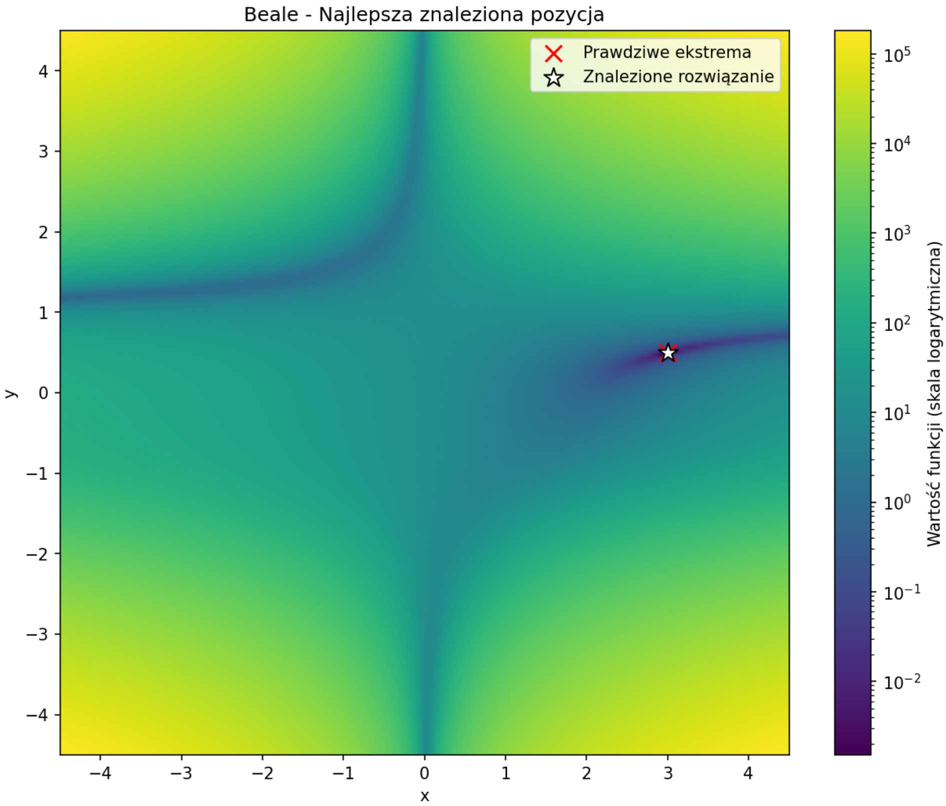
Przeprowadzono serię eksperymentów dla dwóch funkcji testowych. Dla każdej konfiguracji algorytm był uruchamiany 15 razy w celu zmniejszenia wpływu losowości.

5.1 Najlepsze znalezione rozwiązania



Rysunek 3: Najlepsza znaleziona pozycja dla funkcji Himmelblau. Gwiazdka oznacza znalezione rozwiązanie, które niemal idealnie pokrywa się z jednym z czterech minimów globalnych.

Dla funkcji Himmelblau algorytm PSO skutecznie znajduje jedno z czterech minimów globalnych. Najlepszy uzyskany wynik to $f \approx 1.13 \times 10^{-25}$, co jest praktycznie równe wartości teoretycznej 0.

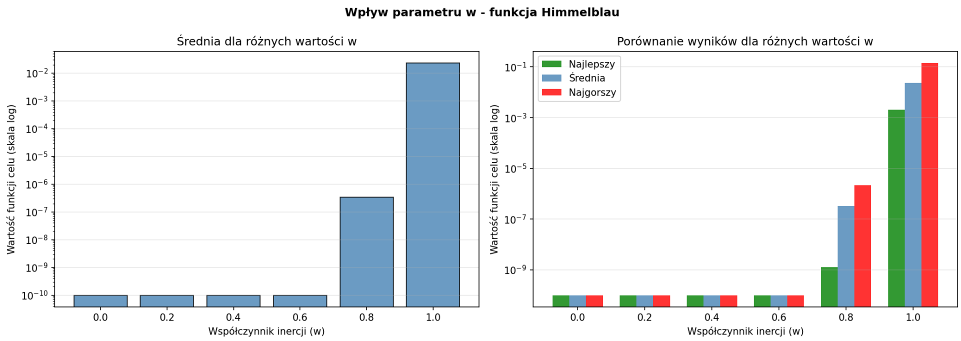


Rysunek 4: Najlepsza znaleziona pozycja dla funkcji Beale. Algorytm skutecznie odnajduje minimum globalne w punkcie $(3, 0.5)$.

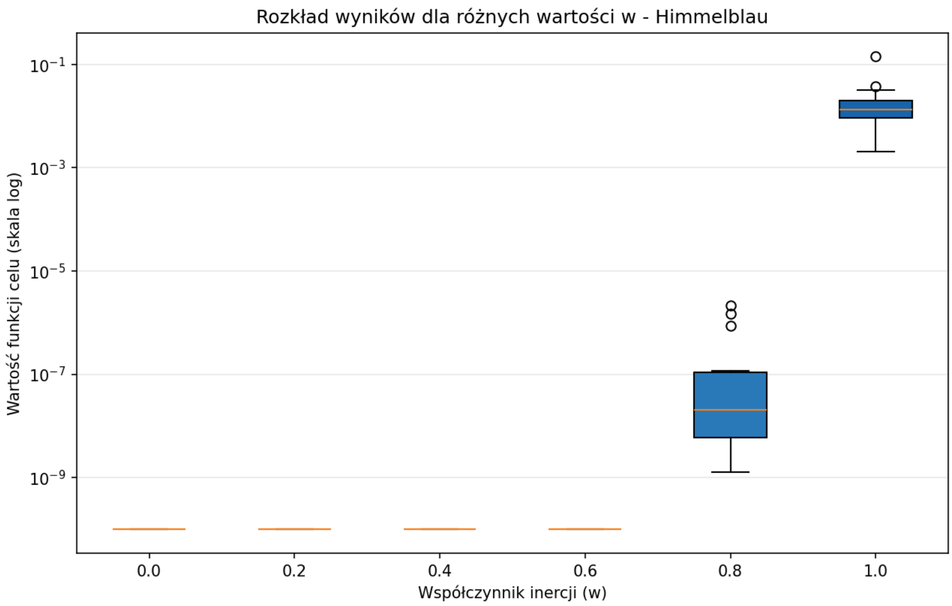
Dla funkcji Beale najlepszy uzyskany wynik to $f \approx 4.13 \times 10^{-23}$, co również jest bardzo bliskie wartości teoretycznej 0.

5.2 Wpływ współczynnika inercji (w)

Badano wartości: $w \in \{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$.



Rysunek 5: Wpływ współczynnika inercji na średnią wartość funkcji Himmelblau.



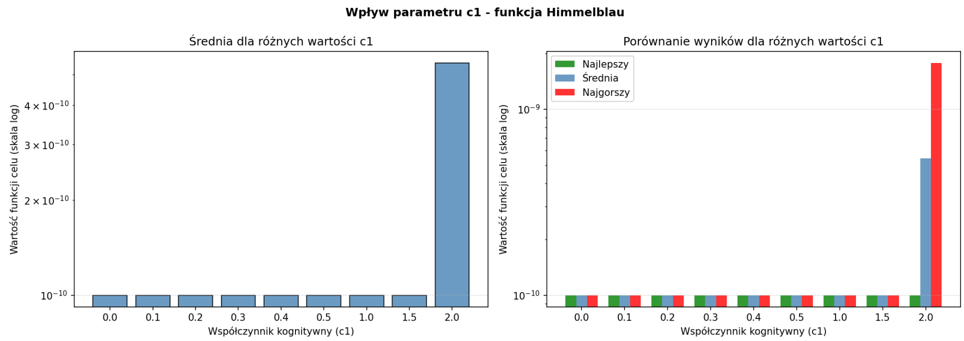
Rysunek 6: Rozkład wyników dla różnych wartości w (Himmelblau).

Obserwacje:

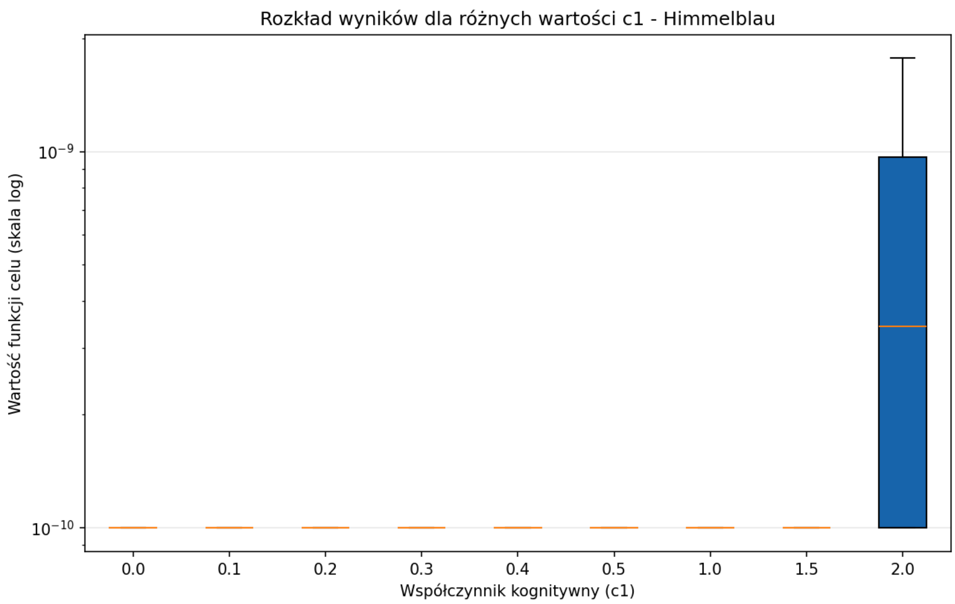
- Dla funkcji Himmelblau najlepsze wyniki uzyskano dla $w = 0.2$, gdzie średni wynik wynosi $\approx 3.68 \times 10^{-31}$.
- Zbyt wysoka wartość w (np. 1.0) powoduje, że cząstki "przelatują" przez minima, co pogarsza zbieżność.
- Zbyt niska wartość w (0.0) eliminuje składową inercyjną, co może prowadzić do przedwczesnej zbieżności.
- Optymalne wartości to $w \in [0.2, 0.6]$.

5.3 Wpływ współczynnika kognitywnego (c_1)

Badano wartości: $c_1 \in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 1.0, 1.5, 2.0\}$.



Rysunek 7: Wpływ współczynnika kognitywnego na średnią wartość funkcji Himmelblau.



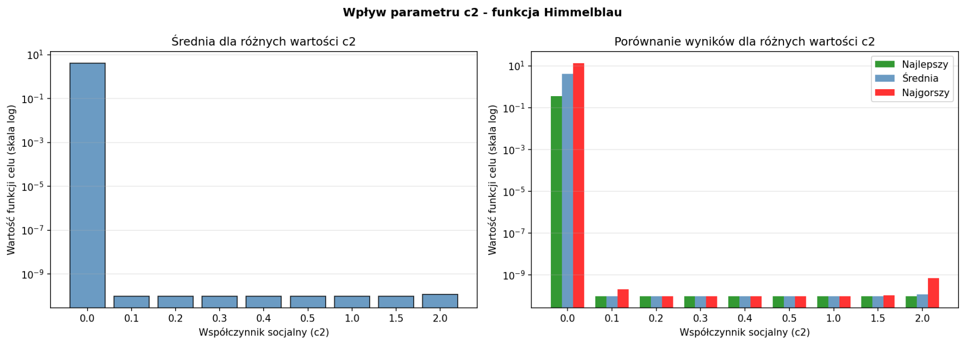
Rysunek 8: Rozkład wyników dla różnych wartości c_1 (Himmelblau).

Obserwacje:

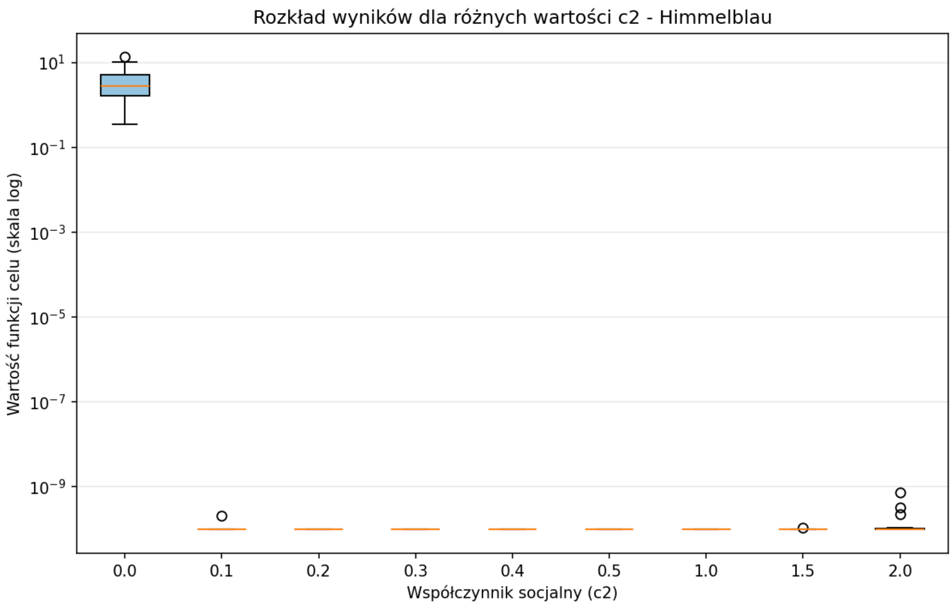
- Najlepsze wyniki dla Himmelblau uzyskano przy $c_1 = 0.3$ (średnia $\approx 3.35 \times 10^{-13}$).
- Wartość $c_1 = 0.0$ oznacza brak składowej kognitywnej — cząstki nie pamiętają swoich najlepszych pozycji.
- Zbyt wysokie wartości c_1 (np. 2.0) mogą powodować oscylacje wokół najlepszych pozycji lokalnych.
- Umiarkowane wartości $c_1 \in [0.3, 1.5]$ zapewniają dobrą równowagę.

5.4 Wpływ współczynnika socjalnego (c_2)

Badano wartości: $c_2 \in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 1.0, 1.5, 2.0\}$.



Rysunek 9: Wpływ współczynnika socjalnego na średnią wartość funkcji Himmelblau.



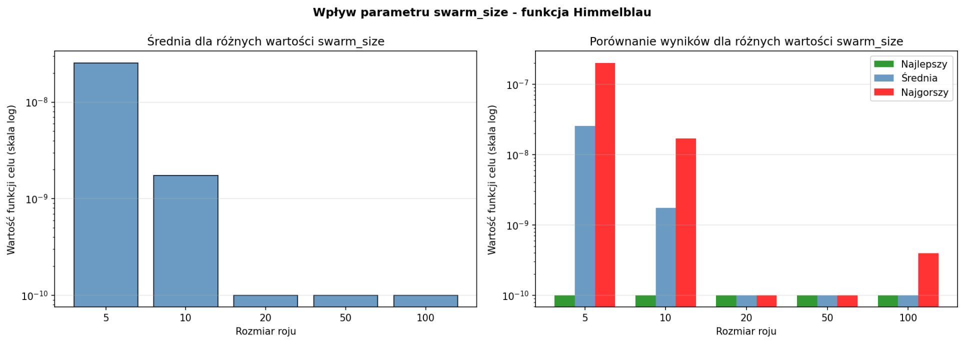
Rysunek 10: Rozkład wyników dla różnych wartości c_2 (Himmelblau).

Obserwacje:

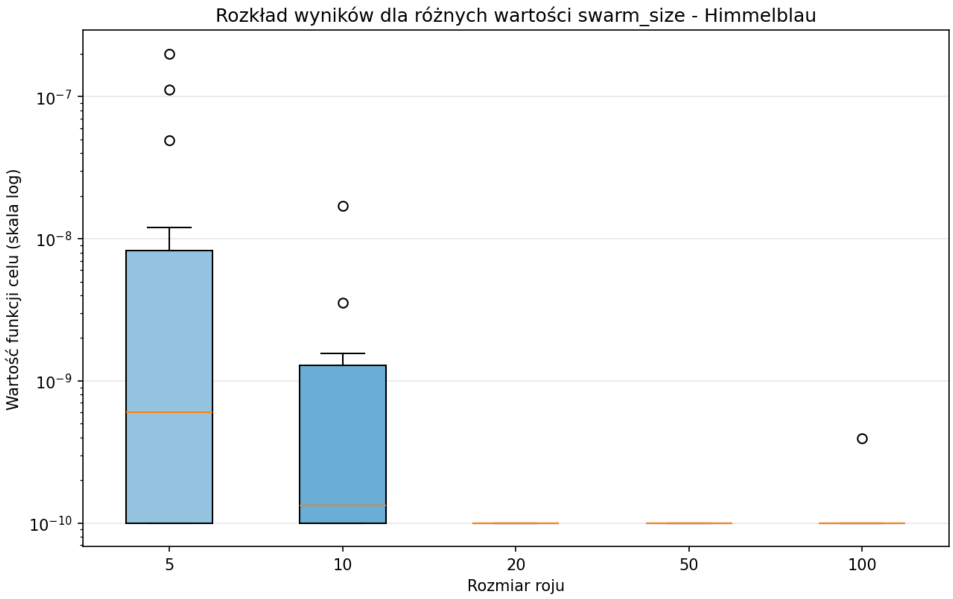
- Najlepsze wyniki dla Himmelblau uzyskano przy $c_2 = 0.5$ (średnia $\approx 1.16 \times 10^{-14}$).
- Wartość $c_2 = 0.0$ eliminuje wpływ społeczny — każda cząstka działa niezależnie.
- Zbyt wysokie c_2 powoduje zbyt szybką zbieżność wszystkich cząstek do jednego punktu (przedwczesna konwergencja).
- Optymalne wartości to $c_2 \in [0.3, 1.5]$.

5.5 Wpływ rozmiaru roju (swarm_size)

Badano wartości: swarm_size $\in \{5, 10, 20, 50, 100\}$.



Rysunek 11: Wpływ rozmiaru roju na średnią wartość funkcji Himmelblau.



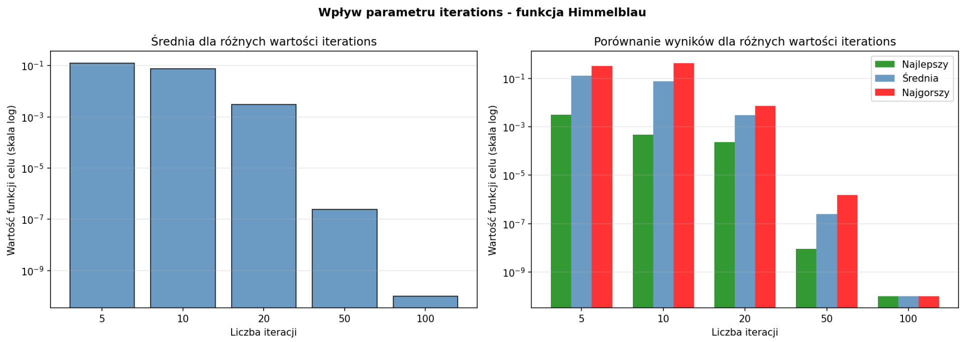
Rysunek 12: Rozkład wyników dla różnych rozmiarów roju (Himmelblau).

Obserwacje:

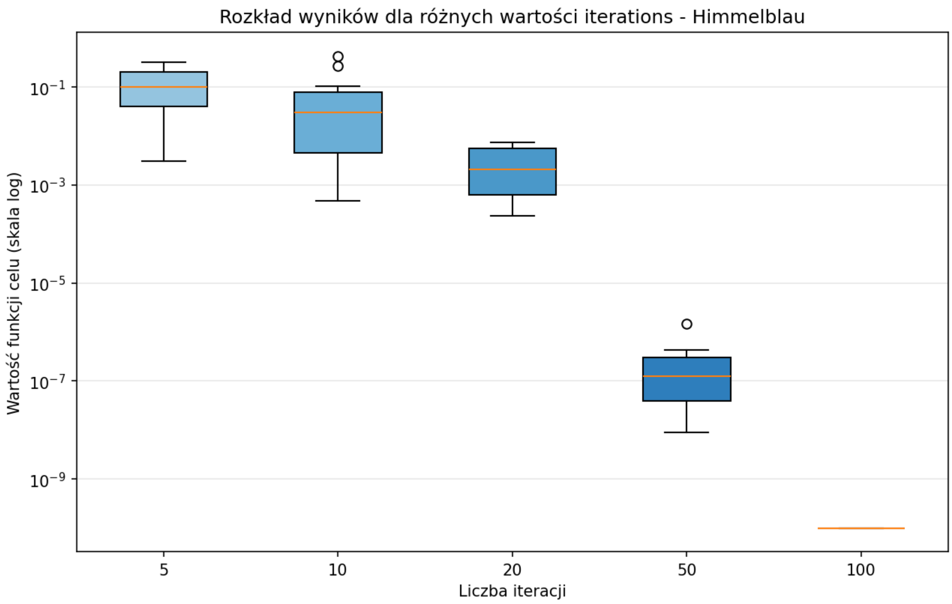
- Większy rój generalnie daje lepsze wyniki — więcej cząstek oznacza lepsze pokrycie przestrzeni poszukiwań.
- Dla swarm_size = 100 uzyskano najlepsze średnie wyniki ($\approx 4.43 \times 10^{-13}$).
- Mały rój (5–10 cząstek) może nie być wystarczający do efektywnej eksploracji.
- Zwiększanie rozmiaru roju powyżej 50–100 daje malejące korzyści przy rosnącym koszcie obliczeniowym.

5.6 Wpływ liczby iteracji

Badano wartości: iterations $\in \{5, 10, 20, 50, 100\}$.



Rysunek 13: Wpływ liczby iteracji na średnią wartość funkcji Himmelblau.

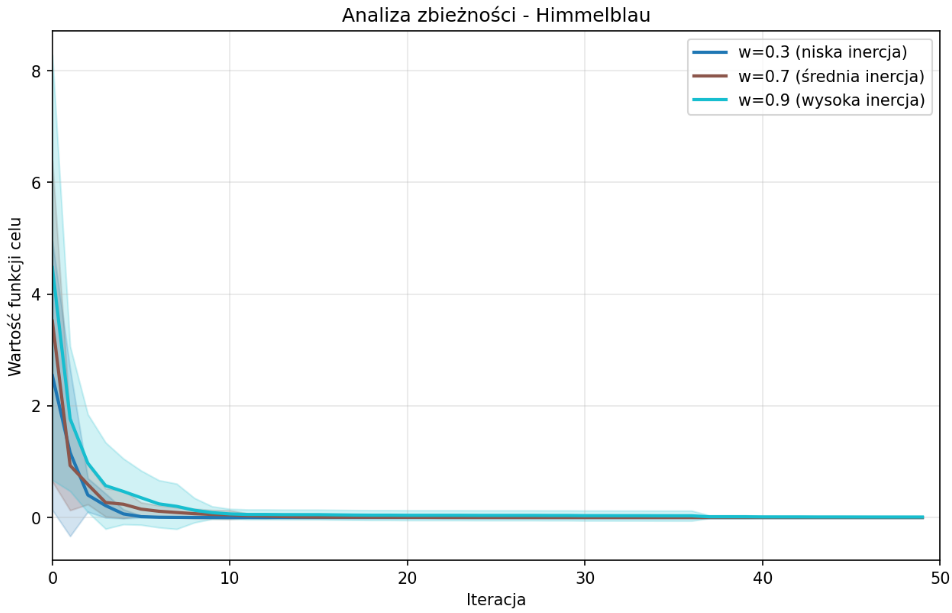


Rysunek 14: Rozkład wyników dla różnej liczby iteracji (Himmelblau).

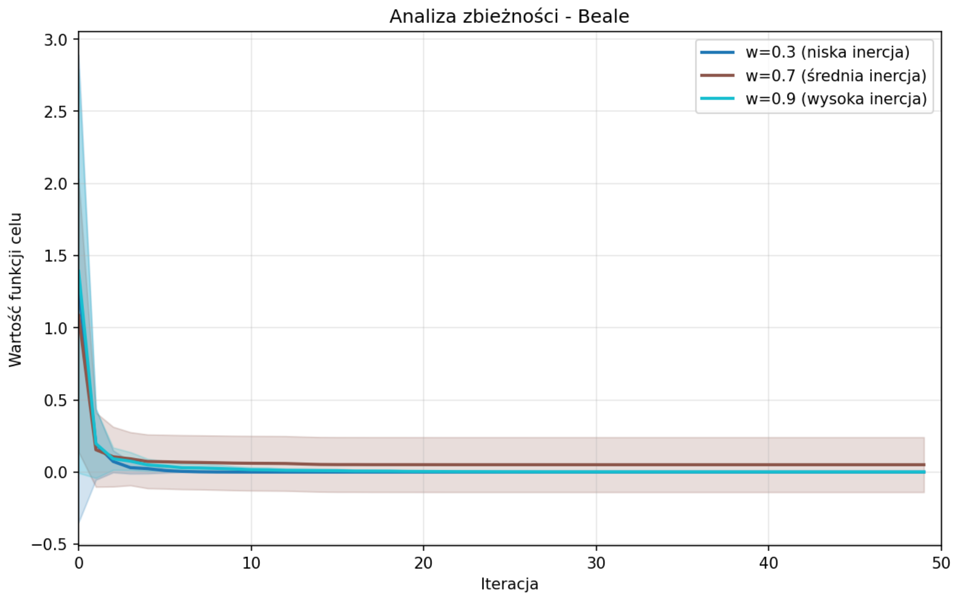
Obserwacje:

- Więcej iteracji pozwala na dokładniejsze zbliżenie się do minimum.
- Przy 100 iteracjach uzyskano najlepsze wyniki ($\approx 5.61 \times 10^{-12}$).
- Algorytm szybko osiąga dobre przybliżenie — już po 20–50 iteracjach wyniki są bliskie optimum.
- Dla prostych funkcji jak Himmelblau, 50–100 iteracji jest zazwyczaj wystarczające.

5.7 Analiza zbieżności



Rysunek 15: Analiza zbieżności dla funkcji Himmelblau przy różnych wartościach współczynnika inercji. Kolorowe obramowanie pokazuje odchylenie standardowe z 15 uruchomień.



Rysunek 16: Analiza zbieżności dla funkcji Beale przy różnych wartościach współczynnika inercji.

Obserwacje:

- Algorytm PSO wykazuje szybką zbieżność — większość poprawy następuje w pierwszych 20–50 iteracjach.
- Niska inercja ($w = 0.3$) prowadzi do szybszej początkowej zbieżności, ale może utknąć w lokalnym minimum. Dla tej wartości osiągnięto najlepszy wynik, ale ta wartość może być zła dla bardziej skomplikowanych funkcji.

- Wysoka inercja ($w = 0.9$) pozwala na dłuższą eksplorację, ale zbieżność jest wolniejsza.
- Średnia inercja ($w = 0.7$) stanowi dobry kompromis między eksploracją a eksploatacją.

6 Analiza wyników

Na podstawie przeprowadzonych eksperymentów można odpowiedzieć na kluczowe pytania dotyczące algorytmu PSO:

6.1 Jak wybrane parametry wpływają na jakość wyników?

- **Współczynnik inercji (w):** Jest to najważniejszy parametr algorytmu. Decyduje o tym, czy cząstki będą bardziej eksplorować przestrzeń (wysokie w), czy skupią się na eksploatacji znalezionych rozwiązań (niskie w). W naszych eksperymentach najlepsze wyniki uzyskano dla $w \in [0.2, 0.8]$. Przy zbyt niskiej wartości algorytm zbyt szybko zbiegał do pierwszego znalezionego minimum, a przy zbyt wysokiej — cząstki poruszały się chaotycznie.
- **Współczynniki c_1 i c_2:** Parametr c_1 określa, jak bardzo cząstka ufa swojemu dotychczasowemu doświadczeniu (komponent poznawczy), a c_2 — jak bardzo podąża za najlepszym rozwiązaniem całego roju (komponent społeczny). W naszej implementacji użyliśmy domyślnych wartości $c_1 = c_2 = 1.5$, które zapewniają zrównoważony wpływ obu komponentów na ruch cząstek. Takie ustawienie pozwala cząstkom zarówno eksplorować okolice własnych najlepszych pozycji, jak i zbiegać do globalnie najlepszego rozwiązania.
- **Rozmiar roju:** Im więcej cząstek, tym lepiej przeszukiwana jest przestrzeń rozwiązań, ale rośnie też czas obliczeń. W praktyce 50–100 cząstek okazało się wystarczające dla testowanych funkcji.
- **Liczba iteracji:** Algorytm PSO szybko znajduje dobre przybliżenie rozwiązania. Dla dwuwymiarowych funkcji testowych 50–100 iteracji w zupełności wystarczało do osiągnięcia satysfakcjonujących wyników.

6.2 Czy algorytm jest stabilny?

Analiza odchylenia standardowego pokazuje, że:

- Dla funkcji Himmelblau algorytm jest bardzo stabilny — odchylenie standardowe jest niskie ($\approx 4.89 \times 10^{-20}$).
- Dla funkcji Beale stabilność jest nieco niższa (odch. std. ≈ 0.19), co wynika z trudniejszego krajobrazu funkcji.
- Stabilność wzrasta wraz z rozmiarem roju i liczbą iteracji.

6.3 Czy zaobserwowano szybkie czy wolne zbieganie do rozwiązania?

Algorytm PSO wykazuje **szybką zbieżność**:

- Większość poprawy następuje w pierwszych 20–50 iteracjach.
- Po osiągnięciu okolic minimum, dalsza poprawa jest stopniowa.
- Szybkość zbieżności zależy od współczynnika inercji — niższa inercja przyspiesza początkową zbieżność.

6.4 Czy pojawia się ryzyko utknięcia w minimum lokalnym?

- Dla funkcji Himmelblau (4 minima globalne) algorytm zawsze znajduje jedno z nich — nie ma minimów lokalnych.
- Dla funkcji Beale ryzyko utknięcia jest niskie, ale istnieje — w niektórych uruchomieniach wynik był gorszy (stąd wyższe odch. std.).
- Większy rój i odpowiedni dobór parametrów minimalizują ryzyko utknięcia.

6.5 Czy trudność obu funkcji była różna?

Tak, funkcje różnią się trudnością:

- **Himmelblau** — łatwiejsza, ponieważ ma 4 minima globalne (większa szansa trafienia).
- **Beale** — trudniejsza, ma płaskie dno doliny prowadzącej do minimum, co utrudnia precyzyjne zbliżenie.
- Średnie wyniki dla Beale są gorsze (≈ 0.05) niż dla Himmelblau ($\approx 10^{-20}$).

7 Wnioski

Na podstawie przeprowadzonych eksperymentów sformułowano następujące wnioski:

- **Skuteczność algorytmu:** PSO skutecznie znajduje minima globalne dla obu funkcji testowych. Dla Himmelblau uzyskano wyniki rzędu 10^{-25}, dla Beale — 10^{-23}.
- **Współczynnik inercji (w):** Optymalne wartości to $w \in [0.2, 0.8]$. Najlepsze wyniki dla Himmelblau uzyskano przy $w = 0.2$, dla Beale przy $w = 0.8$.
- **Współczynniki kognitywny i socjalny:** Umiarkowane wartości $c_1, c_2 \in [0.3, 1.5]$ dają najlepsze rezultaty. Zbyt wysokie wartości prowadzą do oscylacji.
- **Rozmiar roju:** Większy rój (50–100 cząstek) poprawia jakość wyników, ale zwiększa czas obliczeń.
- **Liczba iteracji:** Algorytm szybko zbliża się do rozwiązania. 50–100 iteracji jest zazwyczaj wystarczające.
- **Szybkość zbieżności:** PSO charakteryzuje się szybką zbieżnością — większość poprawy następuje w pierwszych 20–50 iteracjach.
- **Stabilność:** Algorytm jest stabilny, szczególnie dla funkcji Himmelblau. Dla trudniejszych funkcji (Beale) wariancja wyników jest wyższa.

7.1 Rekomendowane parametry

Na podstawie eksperymentów sugerujemy następujące wartości parametrów:

Parametr	**Zalecana wartość**
Rozmiar roju (swarm_size)	50–100
Współczynnik inercji (w)	0.7
Współczynnik kognitywny (c_1)	0.3–1.5
Współczynnik socjalny (c_2)	0.3–1.5
Liczba iteracji	50–100