Algorytm Mrówkowy

Link do repozytorium na github

https://github.com/kingasmi/algorytm_mrowkowy

1. Teoria

1.1 Wstęp

Algorytm mrówkowy, znany również jako optymalizacja mrówkowa (ang. ant colony optimization, ACO), jest metaheurystyczną metodą rozwiązywania problemów optymalizacyjnych, która naśladuje zachowanie kolonii mrówek w poszukiwaniu najkrótszej trasy do źródła pokarmu.

Algorytm mrówkowy opiera się na działaniu wielu wirtualnych mrówek, które poruszają się po grafie reprezentującym problem optymalizacyjny. Każda mrówka wybiera kolejne kroki na podstawie informacji lokalnych i globalnych. Informacje lokalne to feromony pozostawiane przez inne mrówki na odwiedzanych ścieżkach, a globalne to heurystyka, która określa atrakcyjność danego kierunku na podstawie pewnych heurystycznych informacji.

W początkowej fazie algorytmu mrówki poruszają się losowo po grafie, a w miarę upływu czasu wybierają coraz bardziej optymalne trasy na podstawie informacji pozostawianych przez siebie i inne mrówki. Feromony są aktualizowane na podstawie jakości wybranych rozwiązań, co prowadzi do wzmacniania atrakcyjności lepszych ścieżek.

Algorytm mrówkowy znalazł zastosowanie w wielu dziedzinach, takich jak problem komiwojażera, układanie planów, routing w sieciach telekomunikacyjnych, projektowanie układów elektronicznych i wiele innych problemów optymalizacyjnych. Dzięki swojej zdolności do znajdowania zbliżonych do

optymalnych rozwiązań w czasie rzeczywistym, algorytmy mrówkowe są często wykorzystywane w sytuacjach, w których tradycyjne metody optymalizacyjne mogą być nieefektywne lub niewystarczające.

1.2 Schemat działania algorytmu mrówkowego

1. Inicjalizacja:
    - Tworzenie grafu lub planszy, na której odbywać się będzie poszukiwanie rozwiązania.
    - Losowe rozmieszczenie mrówek w początkowych punktach.

2. Przesuwanie się mrówek:
    - Każda mrówka porusza się po grafie/planszy, odwiedzając kolejne wierzchołki/komórki.
    - Przy wyborze kolejnego ruchu, mrówka kieruje się informacjami lokalnymi i globalnymi.
3. Informacje lokalne:
    - Mrówka odczuwa informacje lokalne, które mogą obejmować:
        - Feromony pozostawione przez inne mrówki na odwiedzonych ścieżkach.
        - Heurystyki opisujące atrakcyjność danego ruchu, np. odległość do celu.

4. Informacje globalne:
    - Mrówka uwzględnia informacje globalne, które mogą zawierać:
        - Ogólne właściwości grafu/planszy, np. odległości między wierzchołkami.
        - Ogólne kryteria optymalizacyjne.

5. Aktualizacja feromonów:
    - Gdy mrówka przechodzi przez krawędź, pozostawia tam feromony.

- Feromony na krawędziach są stopniowo aktualizowane, biorąc pod uwagę jakość rozwiązania.

6. Powtarzanie procesu:
   - Proces przesuwania się mrówek i aktualizacji feromonów jest powtarzany przez określoną liczbę iteracji.
   - Może być również zdefiniowany warunek zakończenia, np. czas trwania, brak znaczących zmian.
7. Wybór rozwiązania:
   - Po zakończeniu iteracji, można wybrać najlepsze znalezione rozwiązanie.
   - Najlepsze rozwiązanie może być wybrane na podstawie ilości feromonów na odwiedzonych krawędziach lub innych kryteriów.

Obliczanie prawdopodobieństwa przejście z wierzchołka *i* do wierzchołka *j* przez mrówkę *k*

$$P_{ij}^{k} = \frac{(\tau_{ij})^{\alpha} * (\eta_{ij})^{\beta}}{\sum_{l \epsilon J_i^k} (\tau_{il})^{\alpha} * (\eta_{il})^{\beta}}$$

gdzie:

- P_{ij}^{k} to prawdopodobieństwo przejścia z wierzchołka *i* do wierzchołka *j*
- τ_{ij} to stężenie feromonu na krawędzi między wierzchołkami *i* i *j*
- η_{ij} to atrakcyjność przejścia z wierzchołka *i* do *j*
- α *i* β to parametry, które kontrolują wpływ intensywności feromonu i atrakcyjności przejścia
- J_i^k to zbiór wierzchołków dostępnych dla mrówki *k* z wierzchołka *i*

Aktualizacja fermonu na ścieżce:

$$\tau_{ij} = (1 - \rho)\tau_{ij} + \Delta\tau_{ij}^{k}$$

Gdzie:

- τ_{ij} to intensywność feromonu na ścieżce między *i*-tym a *j*-tym wierzchołkiem.
- $\Delta\tau_{ij}^{k}$ to ilość feromonu, którą *k*-ta mrówka zdeponowała na ścieżce między *i*-tym a *j*-tym wierzchołkiem.
- ρ to współczynnik parowania feromonu, który jest zazwyczaj ustalany na wartość pomiędzy 0 a 1.

Obliczanie $\Delta\tau_{ij}^{k}$ dla każdej mrówki po przejściu ścieżki

$$\Delta\tau_{ij}^{k} = \frac{Q}{L_k}$$

Gdzie:

- Q to parametr określający ilość fermonów pozostawionych przez mrówkę na ścieżce
- L_k to długość ścieżki przebytej przez mrówkę *k*

2. Rozwiązanie

2.1 Importowanie potrzebnych bibliotek

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.animation as animation
```

2.2 Zdefiniowanie funkcji celu

Funkcja"sphere_function" oblicza wartość funkcji celu dla podanych wartości x i y. Funkcja ta jest reprezentacją funkcji sfery, która jest zdefiniowana jako suma kwadratów wartości x i y. Zwraca wynik tej sumy.

Funkcja "rastrigin_function" również oblicza warto±¢ funkcji celu dla podanych wartości x i y. Ta funkcja reprezentuje funkcję Rastrigina, która jest zdefiniowana jako suma kilku składników. Pierwszy składnik to 20, a pozostałe składniki zawierają kwadraty wartości x i y, a także obliczenia kosinusów na podstawie tych wartości. Funkcja zwraca wynik sumy tych składników.

```
def sphere_function(x, y):
    """Funkcja celu - sfera"""
    return x**2 + y**2

def rastrigin_function(x, y):
    """Funkcja celu - Rastrigin"""
    return 20 + x**2 - 10 * np.cos(2 * np.pi * x) + y**2 - 10 * np.cos(2 *
np.pi * y)
```

2.3 Ustalamy parametry optymalizacji

Klasa *Ant* reprezentuje mrówkę. Konstruktor klasy inicjalizuje obiekt *Ant* z początkowymi współrzędnymi *x* i *y*. Dodatkowo, oblicza wartość funkcji sfery dla tych współrzędnych i przechowuje ją w zmiennej *z*.

```
class Ant:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.z = sphere_function(x, y)
```

2.4 Funkcja Update

Funkcja *update_ant* aktualizuje współrzędne mrówki *ant*. Tworzy nowe wartości *new_x* i *new_y*, które są zaktualizowanymi współrzędnymi *x* i *y* mrówki, dodając do nich losową wartość z zakresu *[-step_size, step_size]*. Następnie oblicza nową wartość funkcji sfery dla zaktualizowanych współrzędnych (*new_x*, *new_y*) i porównuje ją z poprzednią wartością *z* mrówki (*ant.z*). Jeśli nowa wartość jest mniejsza, to aktualizuje współrzędne *x*, *y* i *z* mrówki.

```
def update_ant(ant, step_size):
    delta_x = np.random.uniform(-step_size, step_size)
    delta_y = np.random.uniform(-step_size, step_size)
    new_x = ant.x + delta_x
    new_y = ant.y + delta_y
    new_z = rastrigin_function(new_x, new_y)

    if new_z < ant.z:
        ant.x = new_x
        ant.y = new_y
        ant.z = new_z
```

2.5 Wykresy dla Funkcji celu - sfera

Funkcja *plot_animation* tworzy animację wizualizując działanie algorytmu mrówkowego dla funkcji sfery. Tworzone są dwa wykresy: jeden w trzech wymiarach (*fig_3d*) i drugi w dwóch wymiarach (*fig_2d*).

W trzech wymiarach, najpierw tworzona jest siatka *X* i *Y* dla zakresów *x_range* i *y_range*. Następnie obliczane są wartości *Z* funkcji sfery dla każdego punktu siatki. Wykres trójwymiarowy przedstawia powierzchnię funkcji sfery, a także wyświetla punkty reprezentujące współrzędne mrówek (*ant_points*) oraz punkt najlepszego

wyniku (*best_point*). Dodatkowo, na wykresie wypisywane są informacje o iteracji i najlepszym wyniku.

W dwóch wymiarach, tworzony jest wykres konturowy funkcji sfery, a także wyświetlane są punkty mrówek i punkt najlepszego wyniku. Również wypisywane są informacje o iteracji i najlepszym wyniku.

Funkcja *update(frame)* jest funkcją wywoływaną w każdej iteracji animacji. Aktualizuje współrzędne mrówek, wykresy i teksty informacyjne.

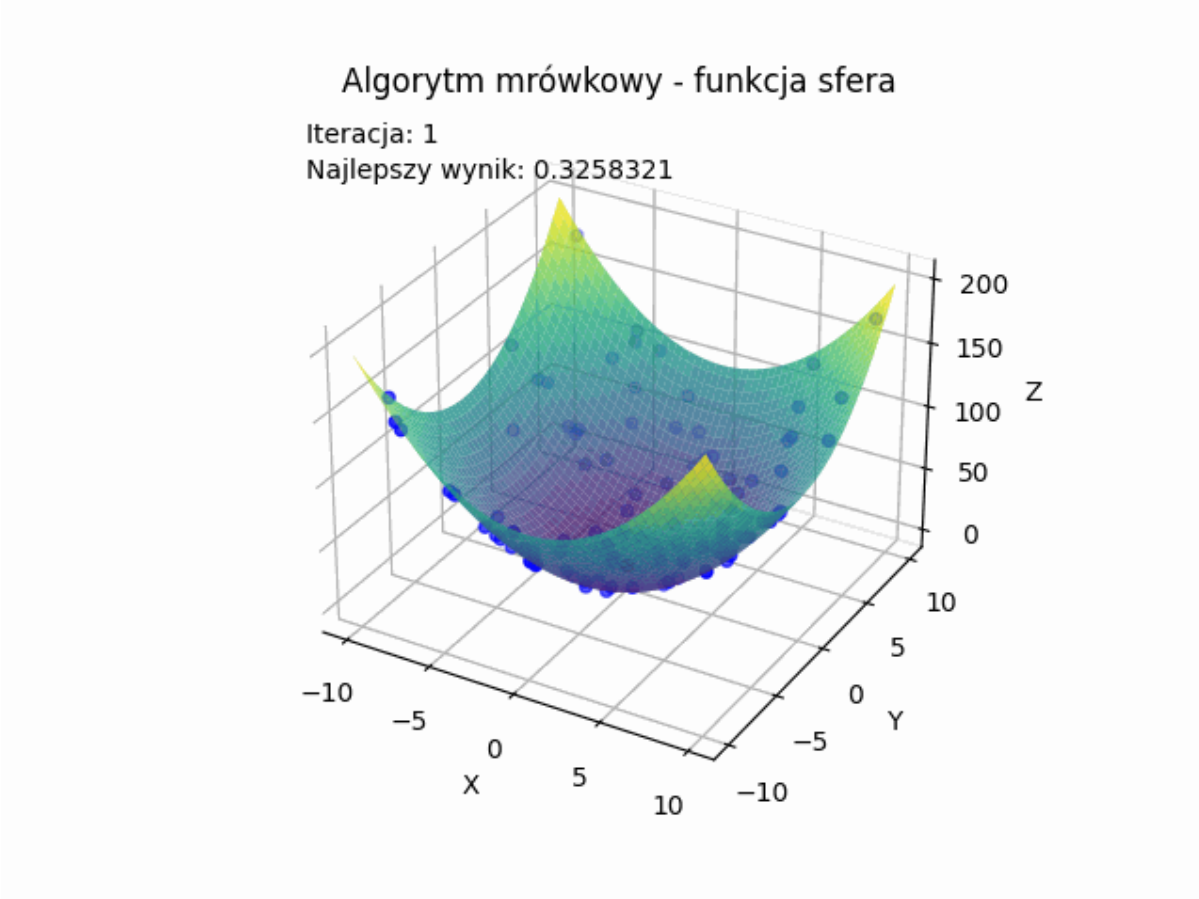
Na koniec, tworzone są animacje (*anim* i *anim_2d*) i zapisywane do plików "sphere_3d.gif" i "sphere.gif". Następnie animacje są wyświetlane przy użyciu *plt.show()*.

Na początku kodu tworzona jest populacja mrówek (*ants*) i następnie uruchamiana jest animacja dla określonej liczby kroków (*num_steps*) i rozmiaru kroku (*step_size*).

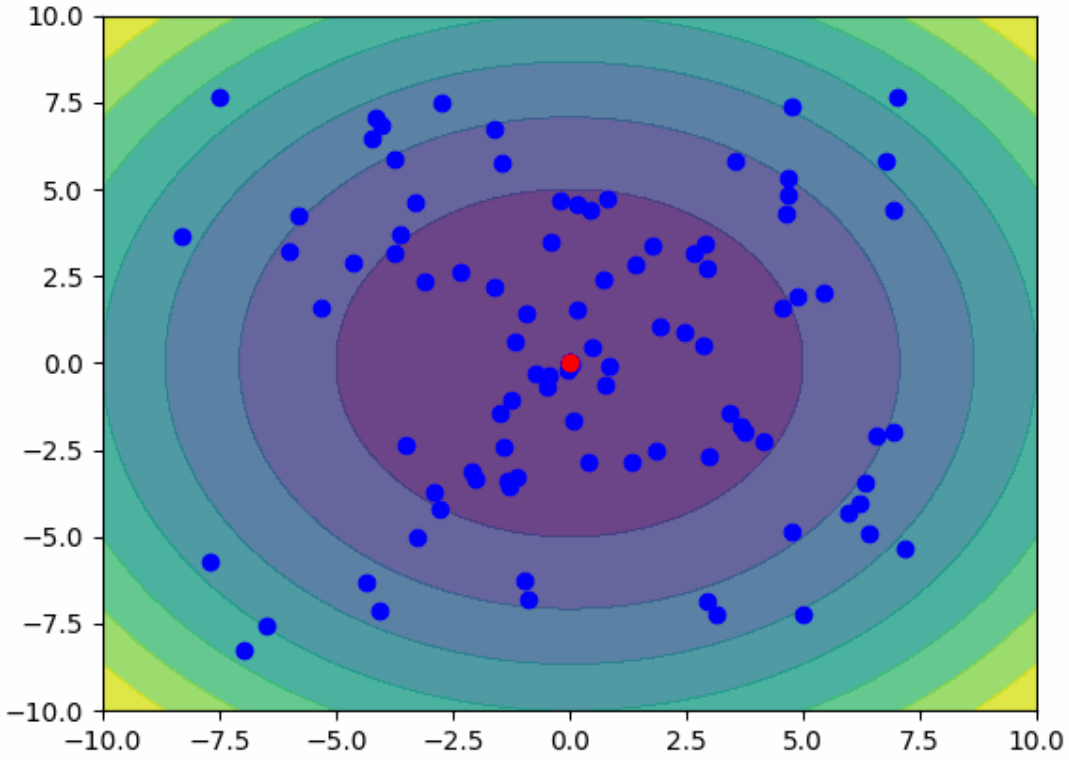
```
def plot_animation(ants, num_steps, step_size):
    fig_3d = plt.figure()
    ax_3d = fig_3d.add_subplot(111, projection='3d')

    fig_2d = plt.figure()
    ax_2d = fig_2d.add_subplot(111)

    x_range = np.linspace(-10, 10, 200)  # Zwiększenie liczby próbek x
    y_range = np.linspace(-10, 10, 200)  # Zwiększenie liczby próbek y
    X, Y = np.meshgrid(x_range, y_range)
    Z = sphere_function(X, Y)

    ax_3d.plot_surface(X, Y, Z, cmap='viridis', alpha=0.8)

     ant_points = ax_3d.scatter([ant.x for ant in ants], [ant.y for ant in
ants], [ant.z for ant in ants], color='blue')
    best_ant = min(ants, key=lambda ant: ant.z)
          best_point = ax_3d.scatter(best_ant.x, best_ant.y, best_ant.z,
color='red')
    iter_text_3d = ax_3d.text2D(0.05, 0.95, "", transform=ax_3d.transAxes)
    best_text_3d = ax_3d.text2D(0.05, 0.90, "", transform=ax_3d.transAxes)
```

```
    ax_3d.set_xlabel('X')
    ax_3d.set_ylabel('Y')
    ax_3d.set_zlabel('Z')
    ax_3d.set_title('Algorytm mrówkowy - funkcja sfera')

    iter_text_2d = ax_2d.text(0.05, 0.95, "", transform=ax_2d.transAxes)
    best_text_2d = ax_2d.text(0.05, 0.90, "", transform=ax_2d.transAxes)

    def update(frame):
        for ant in ants:
            update_ant(ant, step_size)

          ant_points._offsets3d = ([ant.x for ant in ants], [ant.y for ant in
ants], [ant.z for ant in ants])
        best_ant = min(ants, key=lambda ant: ant.z)
        best_point._offsets3d = ([best_ant.x], [best_ant.y], [best_ant.z])
        iter_text_3d.set_text(f"Iteracja: {frame+1}")
        best_text_3d.set_text(f"Najlepszy wynik: {best_ant.z:.7f}")

        ax_2d.clear()
        ax_2d.contourf(X, Y, Z, cmap='viridis', alpha=0.8)
        ax_2d.plot([ant.x for ant in ants], [ant.y for ant in ants], 'bo')
        ax_2d.plot(best_ant.x, best_ant.y, 'ro')
        iter_text_2d.set_text(f"Iteracja: {frame+1}")
        best_text_2d.set_text(f"Najlepszy wynik: {best_ant.z:.7f}")

         anim  =  animation.FuncAnimation(fig_3d,  update,  frames=num_steps,
interval=200, repeat=False)
       anim_2d  =  animation.FuncAnimation(fig_2d,  update,  frames=num_steps,
interval=200, repeat=False)

    # Zapisywanie animacji do plików
    anim.save('sphere_3d.gif', writer='imagemagick')
    anim_2d.save('sphere.gif', writer='imagemagick')

    plt.show()

# Tworzenie populacji mrówek
num_ants = 100
ants = [Ant(np.random.uniform(-10, 10), np.random.uniform(-10, 10)) for _ in
range(num_ants)]
```

```
# Uruchomienie animacji
num_steps = 100
step_size = 0.1
plot_animation(ants, num_steps, step_size)
```



Rysunek nr. 1 Wykres funkcji celu - strefa na wykresie 3D



Rysunek nr. 2 Wykres funkcji celu - strefa na wykresie 2D

2.6 Wykresy dla Funkcji celu - Rastrigin

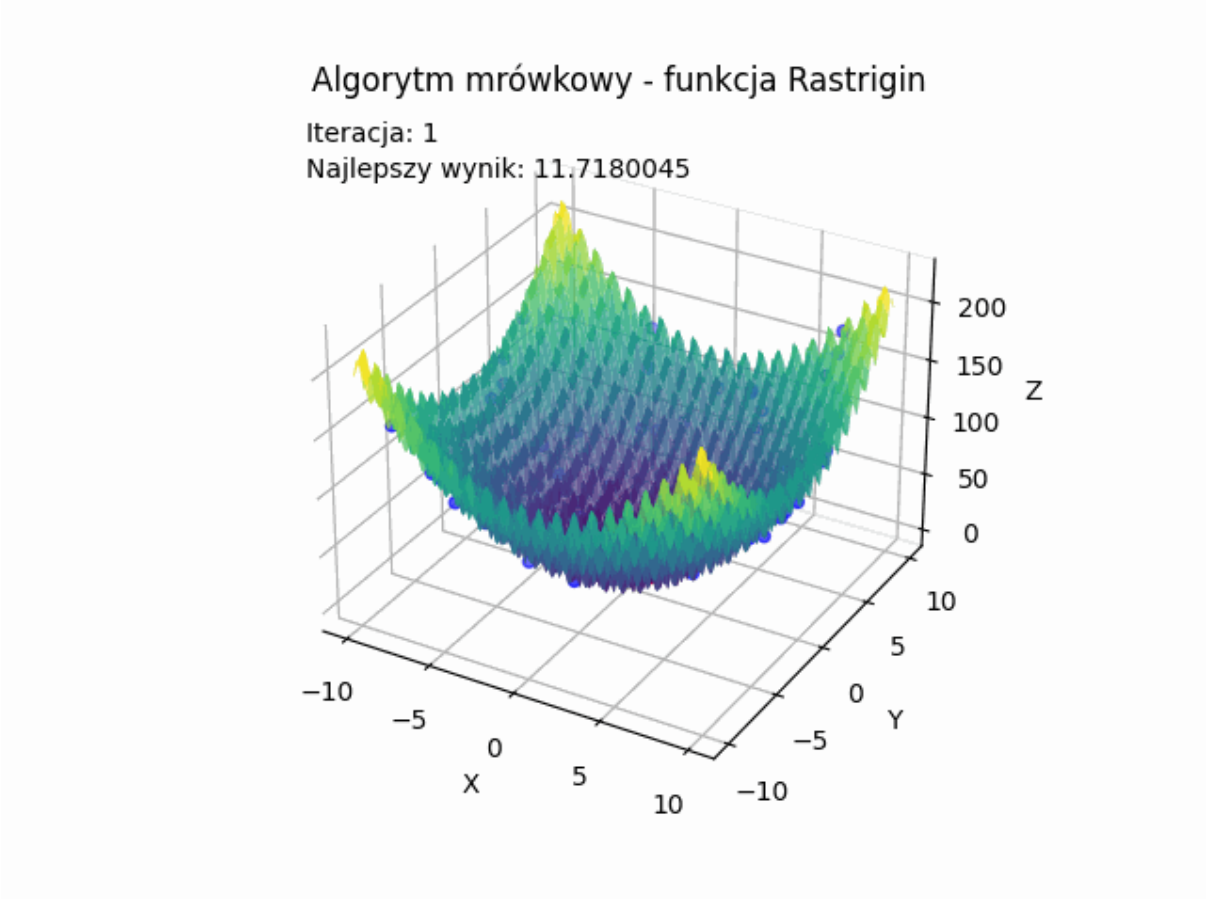
Funkcja *plot_animation* służy do generowania animacji wykresów 2D i 3D dla algorytmu mrówkowego, korzystając z danych populacji mrówek, liczby kroków i rozmiaru kroku. Tworzone są dwa wykresy: jeden w trzech wymiarach (*fig_3d*) i drugi w dwóch wymiarach (*fig_2d*).

Na koniec, funkcje *animation.FuncAnimation* tworzą animacje na podstawie figur i funkcji update dla wykresów 3D i 2D. Animacje są zapisywane do plików GIF (rastrigin_3d.gif i rastrigin.gif). Następnie wyświetlane są wykresy.

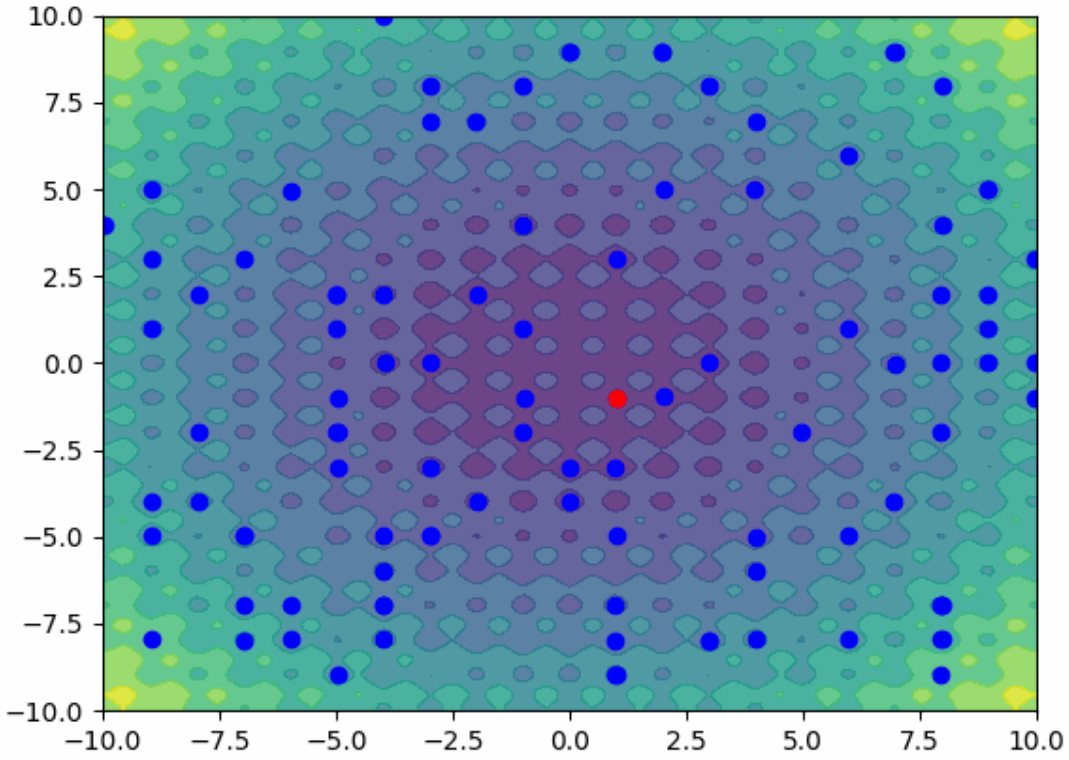
```
def plot_animation(ants, num_steps, step_size):
    fig_3d = plt.figure()
```

```
    ax_3d = fig_3d.add_subplot(111, projection='3d')

    fig_2d = plt.figure()
    ax_2d = fig_2d.add_subplot(111)

    x_range = np.linspace(-10, 10, 200)  # Zwiększenie liczby próbek x
    y_range = np.linspace(-10, 10, 200)  # Zwiększenie liczby próbek y
    X, Y = np.meshgrid(x_range, y_range)
    Z = rastrigin_function(X, Y)

    ax_3d.plot_surface(X, Y, Z, cmap='viridis', alpha=0.8)

    ant_points = ax_3d.scatter([ant.x for ant in ants], [ant.y for ant in
ants], [ant.z for ant in ants], color='blue')
    best_ant = min(ants, key=lambda ant: ant.z)
    best_point = ax_3d.scatter(best_ant.x, best_ant.y, best_ant.z,
color='red')
    iter_text_3d = ax_3d.text2D(0.05, 0.95, "", transform=ax_3d.transAxes)
    best_text_3d = ax_3d.text2D(0.05, 0.90, "", transform=ax_3d.transAxes)

    ax_3d.set_xlabel('X')
    ax_3d.set_ylabel('Y')
    ax_3d.set_zlabel('Z')
    ax_3d.set_title('Algorytm mrówkowy - funkcja Rastrigin')

    iter_text_2d = ax_2d.text(0.05, 0.95, "", transform=ax_2d.transAxes)
    best_text_2d = ax_2d.text(0.05, 0.90, "", transform=ax_2d.transAxes)

    def update(frame):
        for ant in ants:
            update_ant(ant, step_size)

        ant_points._offsets3d = ([ant.x for ant in ants], [ant.y for ant in
ants], [ant.z for ant in ants])
        best_ant = min(ants, key=lambda ant: ant.z)
        best_point._offsets3d = ([best_ant.x], [best_ant.y], [best_ant.z])
        iter_text_3d.set_text(f"Iteracja: {frame+1}")
        best_text_3d.set_text(f"Najlepszy wynik: {best_ant.z:.7f}")

        ax_2d.clear()
        ax_2d.contourf(X, Y, Z, cmap='viridis', alpha=0.8)
        ax_2d.plot([ant.x for ant in ants], [ant.y for ant in ants], 'bo')
```

```
        ax_2d.plot(best_ant.x, best_ant.y, 'ro')
        iter_text_2d.set_text(f"Iteracja: {frame+1}")
        best_text_2d.set_text(f"Najlepszy wynik: {best_ant.z:.7f}")

    anim = animation.FuncAnimation(fig_3d, update, frames=num_steps,
interval=200, repeat=False)
    anim_2d = animation.FuncAnimation(fig_2d, update, frames=num_steps,
interval=200, repeat=False)

    # Zapisywanie animacji do plików
    anim.save('rastrigin_3d.gif', writer='imagemagick')
    anim_2d.save('rastrigin.gif', writer='imagemagick')

    plt.show()


# Tworzenie populacji mrówek
num_ants = 100
ants = [Ant(np.random.uniform(-10, 10), np.random.uniform(-10, 10)) for _ in
range(num_ants)]


# Uruchomienie animacji
num_steps = 100
step_size = 0.1
plot_animation(ants, num_steps, step_size)
```



Rysunek nr. 3 Wykres funkcji celu - Rastrigin na wykresie 3D



Rysunek nr. 4 Wykres funkcji celu - Rastrigin na wykresie 2D

2.7 Wykresy przedstawiają zmianę najlepszego wyniku dla każdej funkcji w kolejnych iteracjach

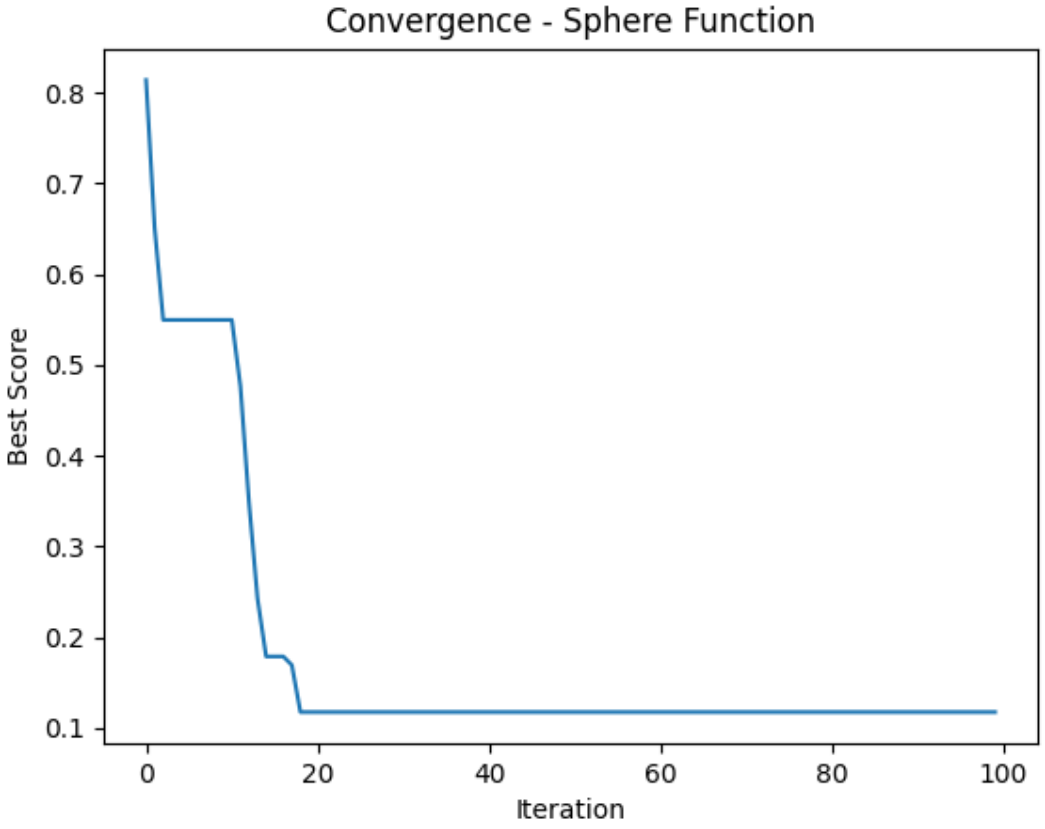
```
for it in range(n_iterations):
    # Każda mrówka wykonuje ruch
    for i in range(n_ants):
        # Losowo wybiera kierunek
        direction = np.random.uniform(-1, 1, 2)
        direction /= np.linalg.norm(direction) # Normalize to unit vector

        # Zaktualizuj pozycje
        ants[i] += step_size * direction

        # Kontrola granicy
        ants[i] = np.clip(ants[i], -10, 10)
```

```
        # Zaktualizuj najlepsze rozwiązanie dla funkcji celu - sfera
        score_sphere = sphere_function(ants[i][0], ants[i][1])
        if score_sphere < best_score_sphere:
            best_score_sphere = score_sphere
            best_ant_sphere = ants[i].copy()

        # Zaktualizuj najlepsze rozwiązanie dla funkcji celu - Rastrigin
        score_rastrigin = rastrigin_function(ants[i][0], ants[i][1])
        if score_rastrigin < best_score_rastrigin:
            best_score_rastrigin = score_rastrigin
            best_ant_rastrigin = ants[i].copy()

    best_scores_sphere.append(best_score_sphere)
    best_scores_rastrigin.append(best_score_rastrigin)
```

- Wykresu funkcji celu - sfera

Tworzymy nowe okno dla wykresu funkcji sfera za pomocą plt.figure(). Następnie rysujemy wykres najlepszego wyniku dla każdej iteracji. Dodajemy etykiety osi i tytuł wykresu. Na koniec używamy plt.savefig() do zapisania wykresu do pliku "sphere.png".

```
# Wykres zbieżności funkcji celu - sfera
plt.figure()
plt.plot(best_scores_sphere)
plt.xlabel('Iteration')
plt.ylabel('Best Score')
plt.title('Convergence - Sphere Function')
plt.savefig('sphere.png')
```

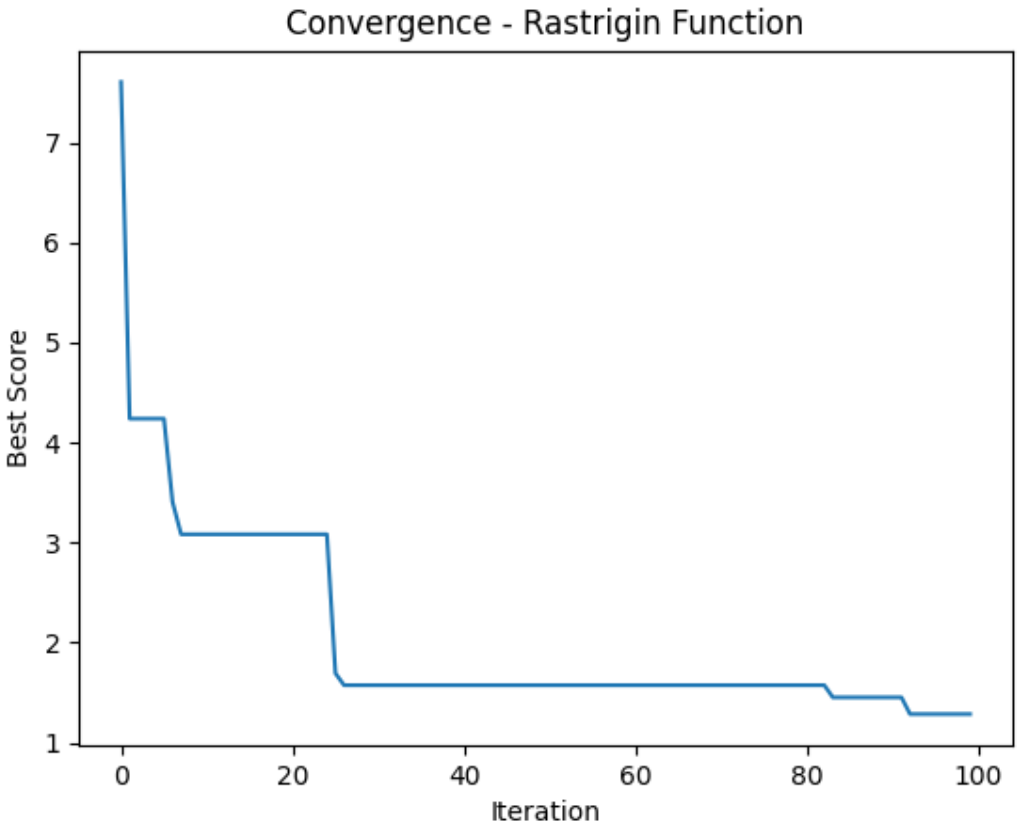


Rysunek nr. 5 Wykres najlepszego wyniku funkcji celu - strefu na

- Wykresu funkcji celu - Rastrigina

Analogicznie tworzymy nowe okno dla wykresu funkcji Rastrigina i zapisujemy go do pliku "rastrigin.png" przy użyciu plt.savefig().

```
# Wykres zbieżności funkcji celu - Rastrigin
plt.figure()
plt.plot(best_scores_rastrigin)
plt.xlabel('Iteration')
plt.ylabel('Best Score')
plt.title('Convergence - Rastrigin Function')
plt.savefig('rastrigin.png')
plt.show(
```



Rysunek nr. 6 Wykres najlepszego wyniku funkcji celu - Rastrigina na