Projekt – metody optymalizacji	Data złożenia projektu: 16.05.2021
Numer grupy projektowej: 02	Imię i nazwisko I: Karol Matoga Imię i nazwisko II: Dariusz Nowak

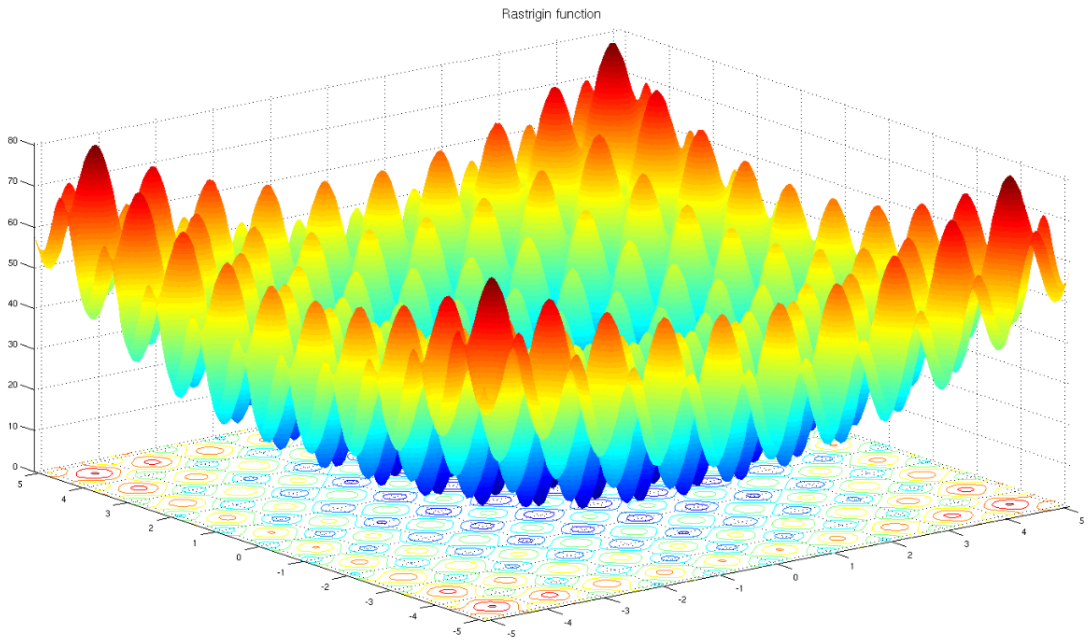
Tytuł projektu

1. Opis optymalizowanych funkcji

Optymalizowaliśmy następujące funkcje:

- Funkcje Rastrigina
- Funkcje Ackley
- Funkcję Sfery

Funkcja Rastrigina



Rys 1. *Wykres funkcji Rastrigina (dla N=2)*

Dziedzina: -5.12 <= x_i <= 5.12

Minima lokalne:

$f(-\ 0.00029737,\ 0.00528922)\ =\ 0.0056$

$f(0.0020469,\ 0.00033317)\ =\ 0.0009$

$f(0.00204161, 0.00033944)\ =\ 0.0008$

Minimum globalne: $f(0,\ldots,0) = 0$

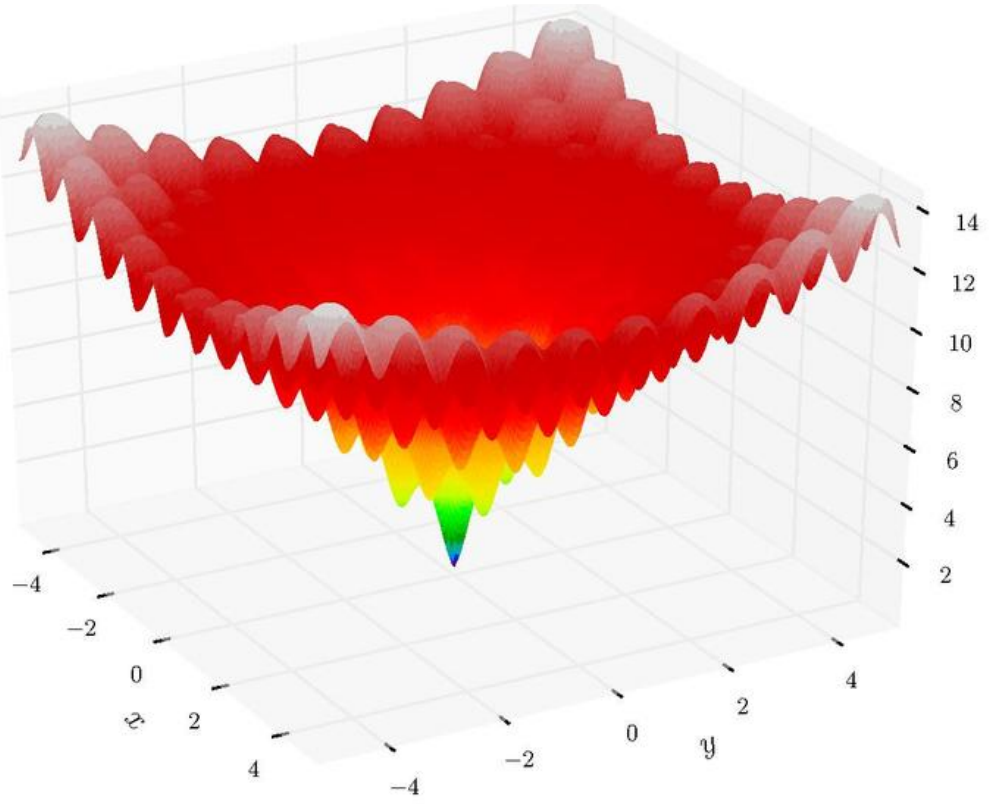
Funkcja Rastrigin jest funkcją niewypukłą używaną jako test wydajności dla algorytmów optymalizacji. Jest to nieliniowa funkcja multimodalna. Znalezienie minimum tej funkcji jest dość trudnym problemem ze względu na dużą przestrzeń wyszukiwania i dużą liczbę minimów lokalnych

Na przestrzeni wektorowej **n** opisana jest wzorem:

$$f(\mathbf{x}) = An + \sum_{i=1}^{n} \left[x_i^2 - A\cos(2\pi x_i) \right]$$

gdzie A = 10 oraz x_i należy do [-5.12, 5.12].

Funkcja Ackleya



Rys 2. *Wykres funkcji Ackley'a*

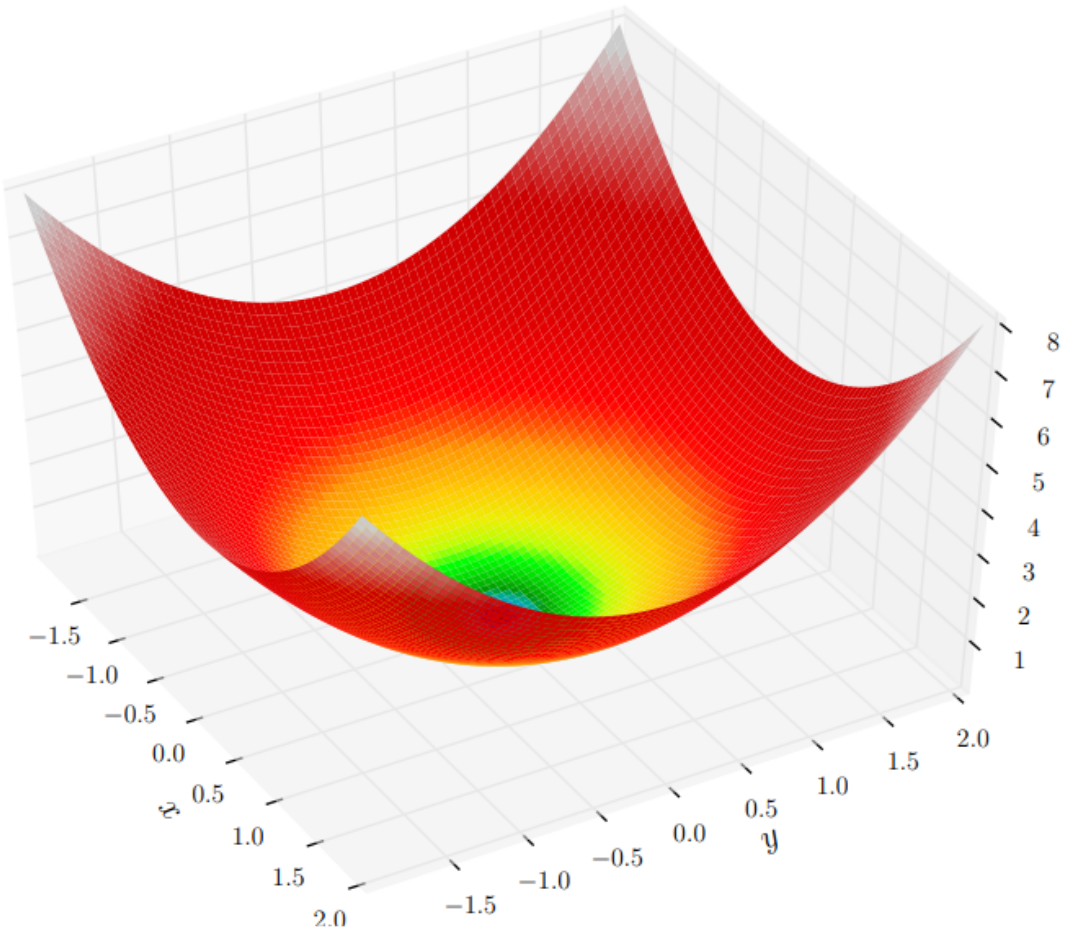
Dziedzina: -5 <= *x, y* <= 5
Minima lokalne:
$f(-\ 0.13052397\ \ 0.12817194) = 1.2403$
$f(-\ 0.21820178\ \ 0.04745513) = 1.5590$
$f(-\ 0.72352724\ \ 0.07104326) = 3.2272$
Minimum globalne: $f(0,0) = 0$

Funkcja Ackleya jest funkcją niewypukłą używaną jako problem testu wydajności dla algorytmów optymalizacji.

W przestrzeni wektorowej dwuwymiarowej jest zdefiniowana przy pomocy wzoru:

$$f(x,y) = -20\exp\left[-0.2\sqrt{0.5\left(x^2+y^2\right)}\right] - \exp\left[0.5\left(\cos 2\pi x + \cos 2\pi y\right)\right] + e + 20$$

Funkcja Sfery



Rys 2. *Wykres funkcji Sfery (dla N=2)*

Dziedzina: $-\infty <= x_i <= \infty,\ 1 <= i <= n$

Minima lokalne:
$f(0.225,\ 0.153,\ 0.341,\ 0.431, 0.283,\ 0.261,\ 0.271,\ 0.414,\ 0.222,\ 0.127) = 0.8375$
$f(0.273,\ 0.272,\ 0.263,\ 0.355,\ 0.181,\ 0.310,\ 0.142,\ 0.212,\ 0.396,\ 0.321) = 0.7996$
$f(\ 0.191,\ 0.467,\ 0.123,\ 0.289,\ 0.398,\ 0.096,\ 0.295,\ 0.156,\ 0.120,\ 0.065) = 0.6527$
Minimum globalne: $f(x_1,\ldots, x_n) = f(0,\ldots, 0) = 0$

Funkcja Sfery jest funkcją unimodalną. Opisana jest za pomocą formuły:

$$f(\boldsymbol{x}) = \sum_{i=1}^{n} x_i^2$$

2. Wykorzystana implementacja wraz z opisem parametrów

Nasz projekt został zrobiony przy użyciu języka Python (środowisko: jupyter notebook uruchomiony z konsoli anaconda, biblioteka: pyswarm ()). Do optymalizacji wybranych przez nas funkcji wykorzystano algorytm optymalizacji rojem cząstek. Ideą algorytmu PSO jest przeszukiwanie przestrzeni rozwiązań danej funkcji za pomocą tzw. roju cząstek. Do każdej cząstki przypisana jest jej pozycja oraz wektor prędkości w jakim się porusza. Każda z cząstek zapamiętuje swoje najlepsze rozwiązanie (rozwiązanie lokalne) oraz najlepsze rozwiązanie z całego roju (rozwiązanie globalne). Prędkość ruchu każdej z cząstek jest zależny od najlepszego położenia globalnego i lokalnego rozwiązania, a także od prędkości w poprzednich krokach. Poniższy wzór opisuje prędkość danej cząstki:

$$v = wv + c_1 r_l (l - x) + c_2 r_g (g - x)$$

Gdzie:

- *v* - prędkość cząstki
- *w* - współczynnik bezwładności, określa wpływ prędkości w poprzednim kroku
- c_1 - współczynnik dążenia do najlepszego lokalnego rozwiązania (im większy, tym cząstka będzie miała większą skłonność do latania wokół swojej najlepszej pozycji)
- c_2 - współczynnik dążenia do najlepszego globalnego rozwiązania (im większy, tym cząstki chętniej będą grupowały się w pobliżu najlepszego globalnego rozwiązania)

- *l* - położenie najlepszego lokalnego rozwiązania
- *g* - położenie najlepszego globalnego rozwiązania
- *x* - położenie cząstki
- r_l, r_g - losowe wartości z przedziału <0,1>

Schemat działania można podzielić na dwa etapy (ogólnie jak w metodach optymalizacji) :

I. Inicjalizacja algorytmu:
   1) Losowanie początkowych pozycji cząstek
   2) Zapisanie aktualnych pozycji cząstek, jako ich najlepszych lokalnych rozwiązań.
   3) Jeśli w jakiejś cząstce jej najlepsze rozwiązanie lokalne jest lepsze od najlepszego globalnego to - zapisz je jako najlepsze rozwiązanie globalne.
   4) Wylosowanie prędkości początkowych cząstek.

II. Pętla optymalizacyjna:
   1) Aktualizacja prędkości cząstek (w oparciu o wiedzę własną, sąsiadów, roju i losowość)
   2) Aktualizacja położenia cząstek (w oparciu o prędkość)
   3) Jeżeli w jakiejś cząstce jej aktualne rozwiązanie jest lepsze od jej najlepszego lokalnego - zaktualizuj jej najlepsze rozwiązanie lokalne.
   4) Jeśli w jakiejś cząstce jej najlepsze rozwiązanie lokalne jest lepsze od najlepszego globalnego to - zaktualizuj najlepsze rozwiązanie globalne.

3. Wyniki, ich dyskusja oraz wnioski

Funkcja Sfery

```
In [85]: iterations = 100
for i in range(iterations):
    # aktualizacja najlepszego minimum lokalnego jakie znalazła czastka
    my_swarm.current_cost = sphere(my_swarm.position) # obliczenie obecnego kosztu kosztu (wartosc funkcji celu)
    my_swarm.pbest_cost = sphere(my_swarm.pbest_pos)  # obliczenie najlepszej lokalnej  pozycji
    my_swarm.pbest_pos, my_swarm.pbest_cost = P.compute_pbest(my_swarm) # przypisanie

    # aktualizacja najlepszego globalnego
    # obliczenie gbest jest zależne od naszej topologii (w naszym wypadku gwiazda)
    if np.min(my_swarm.pbest_cost) < my_swarm.best_cost:
        my_swarm.best_pos, my_swarm.best_cost = my_topology.compute_gbest(my_swarm)


    if i%20==0:
        print('Iteracja: {} | my_swarm.best_cost: {:.4f}'.format(i+1, my_swarm.best_cost))

    # aktualizacja wektorow pozycji i predkosci, zalezne od topologii
    my_swarm.velocity = my_topology.compute_velocity(my_swarm)
    my_swarm.position = my_topology.compute_position(my_swarm)

print('Najlepszy znaleziony koszt: {:.4f}'.format(my_swarm.best_cost))
print('Najlepsza znaleziona pozycja: {}'.format(my_swarm.best_pos))
```

```
Iteracja: 1 | my_swarm.best_cost: 1.5199
Iteracja: 21 | my_swarm.best_cost: 0.8392
Iteracja: 41 | my_swarm.best_cost: 0.8375
Iteracja: 61 | my_swarm.best_cost: 0.8375
Iteracja: 81 | my_swarm.best_cost: 0.8375
Najlepszy znaleziony koszt: 0.8375
Najlepsza znaleziona pozycja: [0.22582631 0.15362977 0.34160872 0.4312552  0.28310125 0.26129921
 0.27148397 0.41479346 0.2229798  0.12778015]
```

Rys 3. *Implementacja oraz wyniki z pierwszego uruchomienia optymalizacji funkcji sfery*

Pierwsze uruchomienie optymalizacji (100 iteracji)

Wartość funkcji celu (y): **0.8375**
Znalezione rozwiązania (x): **0.22582631**, **0.15362977**, **0.34160872**, **0.4312552**, **0.28310125**, **0.26129921**, **0.27148397**, **0.41479346**, **0.2229798**, **0.12778015**

Drugie uruchomienie optymalizacji (100 iteracji)

Wartość funkcji celu (y): **0.7996**
Znalezione rozwiązania (x): **0.27356104**, **0.27284337**, **0.26323798**, **0.35504425**, **0.18161761**, **0.31052417**, **0.14228952**, **0.21253046**, **0.39616645**, **0.32125183**

Trzecie uruchomienie optymalizacji (100 iteracji)

Wartość funkcji celu (y): **0.6527**
Znalezione rozwiązania (x): **0.19175509**, **0.46715089**, **0.12381516**, **0.28961174**, **0.39823832**, **0.09680466**, **0.29505187**, **0.15672704**, **0.12089779**, **0.06586591**

Aby uzyskać następujące wyniki zaimportowano predefiniowaną funkcję sfery:

```
In [3]: from pyswarms.utils.functions.single_obj import sphere
```

Następnie zdefiniowano topologię, opcje uruchomienia (c1, c2 oraz w) i na samym końcu stworzono instancję klasy Swarm, która zawiera funkcjonalności algorytmu roju cząstek:

```
In [101]: my_topology = Star()

In [102]: my_options = {'c1': 0.6, 'c2': 0.3, 'w': 0.4}

In [103]: my_swarm = P.create_swarm(n_particles=50, dimensions=10, options=my_options)
```

Funkcja Rastrigina

```
In [133]: iterations = 100
for i in range(iterations):
    # aktualizacja najlepszego minimum lokalnego jakie znalazła czastka
    my_swarm_rastrigin.current_cost = rastrigin(my_swarm_rastrigin.position) # obliczenie obecnego kosztu kosztu (wartosc funkcji
    my_swarm_rastrigin.pbest_cost = rastrigin(my_swarm_rastrigin.pbest_pos)  # obliczenie najlepszej lokalnej  pozycji
    my_swarm_rastrigin.pbest_pos, my_swarm_rastrigin.pbest_cost = P.compute_pbest(my_swarm_rastrigin) # przypisanie

    # aktualizacja najlepszego globalnego
    # obliczenie gbest jest zależne od naszej topologii (w naszym wypadku gwiazda)
    if np.min(my_swarm_rastrigin.pbest_cost) < my_swarm_rastrigin.best_cost:
        my_swarm_rastrigin.best_pos, my_swarm_rastrigin.best_cost = my_topology_rastrigin.compute_gbest(my_swarm_rastrigin)


    if i%20==0:
        print('Iteracja: {} | my_swarm.best_cost: {:.4f}'.format(i+1, my_swarm_rastrigin.best_cost))

    # aktualizacja wektorow pozycji i predkosci, zalezne od topologii
    my_swarm_rastrigin.velocity = my_topology_rastrigin.compute_velocity(my_swarm_rastrigin)
    my_swarm_rastrigin.position = my_topology_rastrigin.compute_position(my_swarm_rastrigin)

print('Najlepszy znaleziony koszt: {:.4f}'.format(my_swarm_rastrigin.best_cost))
print('Najlepsza znaleziona pozycja: {}'.format(my_swarm_rastrigin.best_pos))
```

```
Iteracja: 1 | my_swarm.best_cost: 6.0220
Iteracja: 21 | my_swarm.best_cost: 0.0515
Iteracja: 41 | my_swarm.best_cost: 0.0057
Iteracja: 61 | my_swarm.best_cost: 0.0057
Iteracja: 81 | my_swarm.best_cost: 0.0057
Najlepszy znaleziony koszt: 0.0056
Najlepsza znaleziona pozycja: [-0.00029737  0.00528922]
```

Rys 4. *Implementacja oraz wyniki z pierwszego uruchomienia optymalizacji funkcji rastrigina*

Pierwsze uruchomienie optymalizacji (100 iteracji)

Wartość funkcji celu (y): **0.0056**
Znalezione rozwiązania (x): **-0.00029737**, **0.00528922**

Drugie uruchomienie optymalizacji (100 iteracji)

Wartość funkcji celu (y): **0.0009**
Znalezione rozwiązania (x): **0.0020469 0.00033317**

Trzecie uruchomienie optymalizacji (100 iteracji)

Wartość funkcji celu (y): **0.0008**
Znalezione rozwiązania (x): **0.00204161 0.00033944**

Aby uzyskać następujące wyniki zaimportowano predefiniowaną funkcję rastrigina:

```
In [12]: from pyswarms.utils.functions.single_obj import rastrigin
```

Następnie zdefiniowano topologię, opcje uruchomienia (c1, c2 oraz w) i na samym końcu stworzono instancję klasy Swarm, która zawiera funkcjonalności algorytmu roju cząstek:

```
In [129]: my_topology_rastrigin = Star()

In [130]: max_bound = 5.12 * np.ones(2)
          min_bound = - max_bound
          bounds = (min_bound, max_bound)

In [131]: my_options_rastrigin = {'c1': 0.6, 'c2': 0.3, 'w': 0.4}

In [132]: my_swarm_rastrigin = P.create_swarm(n_particles=50, dimensions=2, options=my_options, bounds=bounds)
```

Co ważne trzeba było zdefiniować dodatkowe ograniczenia na dziedzinę (bounds) ponieważ funkcja Rastrigina posiada ograniczenia co do dziedziny [-5.12, 5.12].

Funkcja Ackley'a

```
In [73]: iterations = 100
for i in range(iterations):
    # aktualizacja najlepszego minimum lokalnego jakie znalazła czastka
    my_swarm_Ackley.current_cost = ackley(my_swarm_Ackley.position) # obliczenie obecnego kosztu kosztu (wartosc funkcji celu)
    my_swarm_Ackley.pbest_cost = ackley(my_swarm_Ackley.pbest_pos)  # obliczenie najlepszej lokalnej  pozycji
    my_swarm_Ackley.pbest_pos, my_swarm_Ackley.pbest_cost = P.compute_pbest(my_swarm_Ackley) # przypisanie

    # aktualizacja najlepszego globalnego
    # obliczenie gbest jest zależne od naszej topologii (w naszym wypadku gwiazda)
    if np.min(my_swarm_Ackley.pbest_cost) < my_swarm_Ackley.best_cost:
        my_swarm_Ackley.best_pos, my_swarm_Ackley.best_cost = my_topology_Ackley.compute_gbest(my_swarm_Ackley)


    if i%20==0:
        print('Iteracja: {} | my_swarm.best_cost: {:.4f}'.format(i+1, my_swarm_Ackley.best_cost))

    # aktualizacja wektorow pozycji i predkosci, zalezne od topologii
    my_swarm_rosenbrock.velocity = my_topology_Ackley.compute_velocity(my_swarm_Ackley)
    my_swarm_rosenbrock.position = my_topology_Ackley.compute_position(my_swarm_Ackley)

print('Najlepszy znaleziony koszt: {:.4f}'.format(my_swarm_Ackley.best_cost))
print('Najlepsza znaleziona pozycja: {}'.format(my_swarm_Ackley.best_pos))
```

```
Iteracja: 1 | my_swarm.best_cost: 1.2403
Iteracja: 21 | my_swarm.best_cost: 1.2403
Iteracja: 41 | my_swarm.best_cost: 1.2403
Iteracja: 61 | my_swarm.best_cost: 1.2403
Iteracja: 81 | my_swarm.best_cost: 1.2403
Najlepszy znaleziony koszt: 1.2403
Najlepsza znaleziona pozycja: [-0.13052397  0.12817194]
```

Rys 5. *Implementacja oraz wyniki z pierwszego uruchomienia optymalizacji funkcji Ackley'a*

Pierwsze uruchomienie optymalizacji (100 iteracji)

Wartość funkcji celu (y): **1.2403**
Znalezione rozwiązania (x): **-0.13052397**, **0.12817194**

Drugie uruchomienie optymalizacji (100 iteracji)

Wartość funkcji celu (y): **1.5590**

Znalezione rozwiązania (x): **-0.21820178**, **0.04745513**

Trzecie uruchomienie optymalizacji (100 iteracji)

Wartość funkcji celu (y): **3.2272**
Znalezione rozwiązania (x): **-0.72352724**, **0.07104326]**
Aby uzyskać następujące wyniki zaimportowano predefiniowaną funkcję Ackley'a:

```
In [48]: from pyswarms.utils.functions.single_obj import ackley
```

Następnie zdefiniowano topologię, opcje uruchomienia (c1, c2 oraz w) i na samym końcu stworzono instancję klasy Swarm, która zawiera funkcjonalności algorytmu roju cząstek:

```
In [139]: my_topology_Ackley = Star()

In [140]: max_bound_Ackley = 5 * np.ones(2)
          min_bound_Ackley = - max_bound_Ackley
          bounds_Ackley = (min_bound_Ackley, max_bound_Ackley)

In [141]: my_options_Ackley = {'c1': 0.9, 'c2': 0.3, 'w': 0.4}

In [142]: my_swarm_Ackley = P.create_swarm(n_particles=50, dimensions=2, options=my_options_Ackley, bounds=bounds_Ackley)
```

Co ważne trzeba było zdefiniować dodatkowe ograniczenia na dziedzinę (bounds) ponieważ funkcja Rastrigina posiada ograniczenia co do dziedziny [-5, 5].

Warto zauważyć, że wraz ze zmianą parametru c1 oraz c2 rosła dokładność oszacowania minimum globalnego.

Za każdym uruchomieniem optymalizacji otrzymywano różne wyniki. Dzieje się tak ponieważ funkcja tworząca Swarm'a (**pyswarms.backend.generators.create_swarm**) nie otrzymywała argumentu **init_pos**, a w związku z tym pozycja początkowa była generowana losowo.

PSO należy do algorytmów stochastycznych (prym wiedzie losowość). Zazwyczaj algorytmy stochastyczne dają lepsze wyniki od deterministycznych. Aby poprawić działanie algorytmów stochastycznych takich jak PSO należy wykonać wiele iteracji danych obliczeń pod daną metodę aby zminimalizować losowość i otrzymać dokładniejszy wynik.