Systemy Inteligentne

Sprawozdanie 2

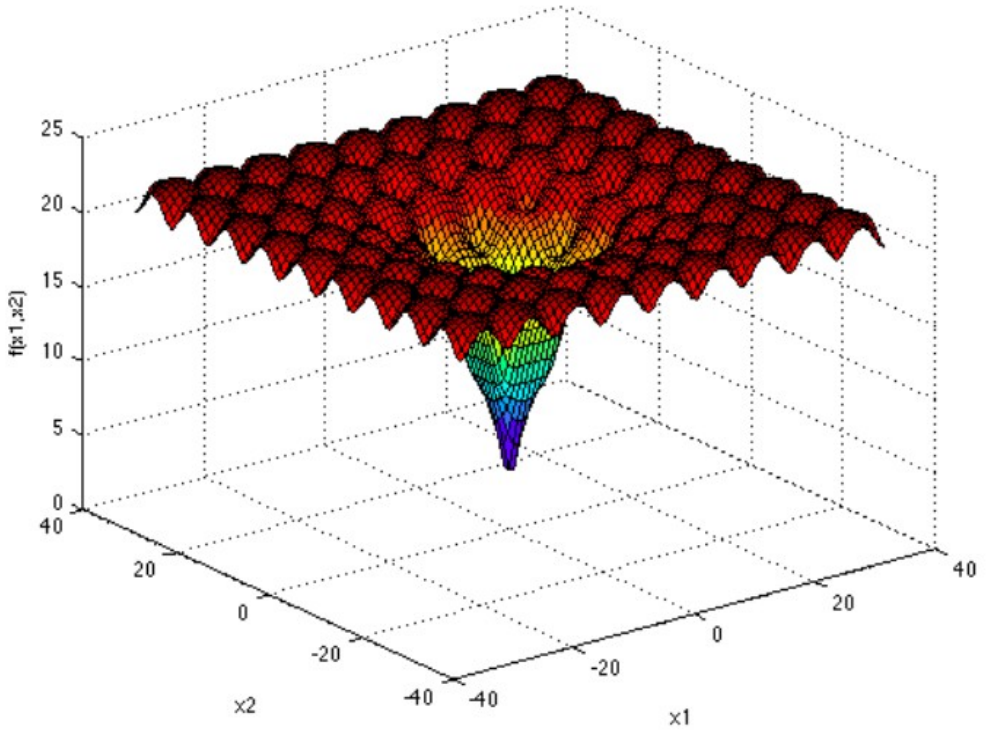
Algorytm optymalizacji stadnej cząsteczek (PSO)

Laboratorium czwartek godzina 8:00

Paweł Lurka
Paweł Chmielarski
Informatyka IV rok 2019/2020, IO1

1. Opis funkcji testowych

• Funkcja Ackleya



$$f(\mathbf{x}) = -a\exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d}x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + \exp(1)$$

Funkcja Ackleya często znajduje zastosowanie w testowaniu algorytmów optymalizacyjnych. Jest to funkcja ciągła, multimodalna. Na powyższym wykresie przedstawiono funkcje w jej dwuwymiarowej formie, która to charakteryzuje się wieloma optimami lokalnymi na obszarze zewnętrznym oraz optimum globalnym w centrum.

d - liczba wymiarów funkcji
a, b, c - parametry stałe, którym zazwyczaj przypisuje się wartości:

$$a=20 \qquad b=0.2 \qquad c=2\pi$$

- Dziedzina funkcji
  Zwykle do badań przyjmuje się dziedzinę z zakresu:
  $x_i \in [-32, 32]$ dla wszystkich $i=1,\ldots,d$
  Jednak może być ona ograniczona do mniejszego zakresu

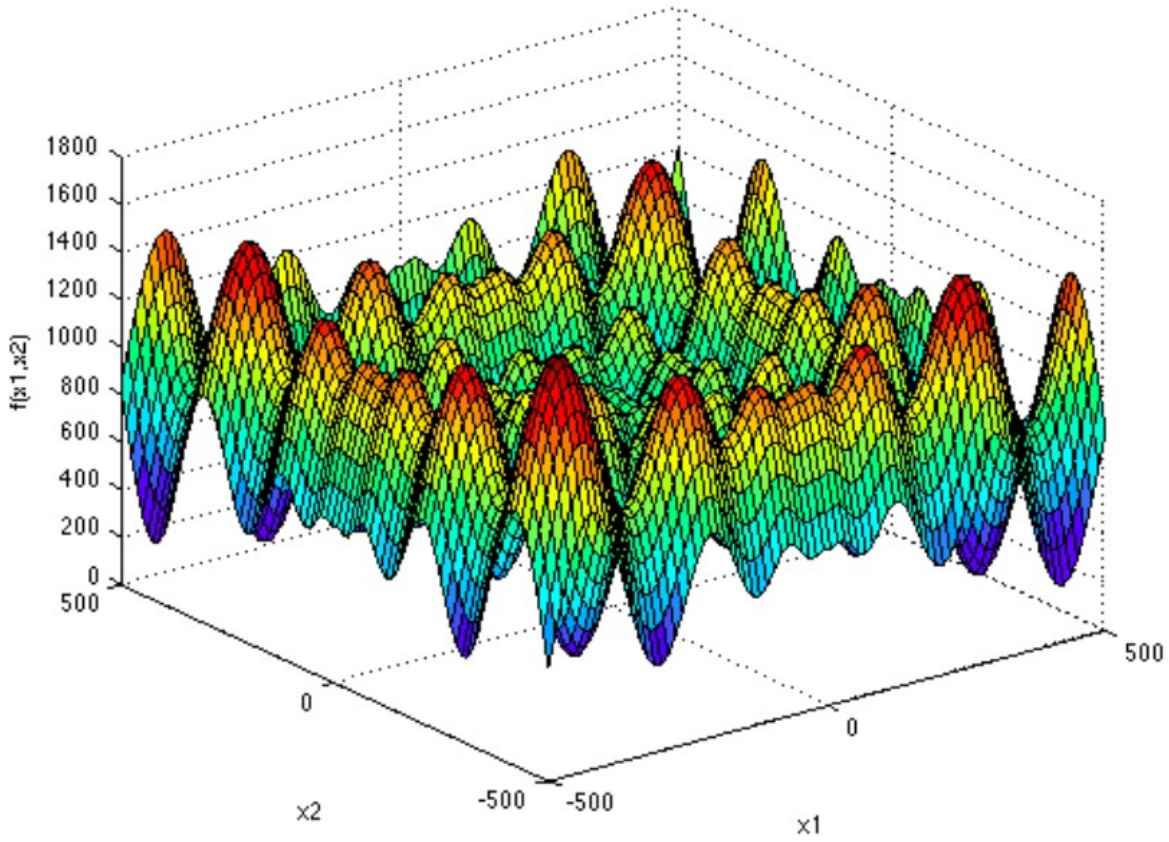
- Minimum globalne
  Funkcja posiada jedne minimum globalne:
  $f(x_1,\ldots,x_d)=0$ w punkcie $x_{1,\ldots},x_d=(0,\ldots,0)$

- W celu przeprowadzenia badań funkcje zaimplementowano w języku Python z wykorzystaniem biblioteki **numpy**

```
from numpy import abs, cos, exp, pi, sin, sqrt, sum

def ackley( x, a=20, b=0.2, c=2*pi ):
    x = np.asarray_chkfinite(x)  # ValueError if any NaN or Inf
    n = len(x)
    s1 = sum( x**2 )
    s2 = sum( cos( c * x ))
    return -a*exp( -b*sqrt( s1 / n )) - exp( s2 / n ) + a + exp(1)
```

- Funkcja Schwefela



$$f(\mathbf{x}) = 418.9829d - \sum_{i=1}^{d} x_i \sin(\sqrt{|x_i|})$$

Funkcja Schwefera jest funkcją złożoną, ciągłą, multimodalną która posiada wiele minimów lokalnych. Powyżej przedstawiono funkcje w przestrzeni 2 wymiarowej, jednak na potrzeby niniejszej pracy będzie ona badana dla trzech wymiarów.

d - liczba wymiarów funkcji

- Dziedzina funkcji
  Zwykle do badań przyjmuje się dziedzinę z zakresu:
  $x_i \in [-500, 500]$ dla wszystkich $i = 1, \ldots, d$
  Jednak może być ona ograniczona do mniejszego zakresu

- Minimum globalne
  Funkcja posiada jedne minimum globalne:
  $f(x_1, \ldots, x_d) = 0$ w punkcie $x_{1,\ldots}, x_d = (420.9687, \ldots, 420.9687)$

- W celu przeprowadzenia badań funkcje zaimplementowano w języku Python z wykorzystaniem biblioteki **numpy**

```
def schwefel( x ):
    x = np.asarray_chkfinite(x)
    n = len(x)
    return 418.9829*n - sum( x * sin( sqrt( abs( x ))))
```

1. Implementacja i modyfikacja

W niniejszej implementacji mimo obecności wielu gotowych bibliotek do optymalizacji stadnej cząsteczek (np. PsoPy) wykorzystano jedynie bibliotekę random w celu generowania liczb pseudolosowych, a reszta zaimplementowana została na podstawie wytycznych z zadania.

Klasa Particle

Pierwszym krokiem było utworzenie klasy pojedynczej cząsteczki i dodanie odpowiednich pól, oraz nadanie jej początkowej prędkości oraz pozycji za pomocą generatora liczb pseudolosowych.

```
class Particle:
    def __init__(self, bounds):
        self.position_i = []  # pozycja cząsteczki
        self.velocity_i = []  # prędkość cząsteczki
        self.pos_best_i = []  # najlepsza pozycja cząsteczki
        self.err_best_i = -1  # najmniejsza znaleziona wartość funkcji
        self.err_i = -1  # aktualna wartość funkcji

        for i in range(0, num_dimensions):
            self.velocity_i.append(uniform(-1, 1))
            self.position_i.append(randint(bounds[i][0], bounds[i][1]))
```

Następnie należało utworzyć metody klasy Particle, z których każda odpowiada za działania wykonywane przez pojedynczą cząsteczkę.

```python
# sprawdzenie wartości funkcji w pozycji cząsteczki
def evaluate(self, costFunc):
    self.err_i = costFunc(self.position_i)

    # sprawdzenie czy aktualna pozycja jest najlepszą znalezioną przez cząsteczkę
    if self.err_i < self.err_best_i or self.err_best_i == -1:
        self.pos_best_i = self.position_i.copy()
        self.err_best_i = self.err_i
```

Najpierw sprawdzana jest wartość funkcji w pozycji cząsteczki i w razie potrzeby aktualizowana najlepsza pozycja znaleziona do tej pory przez cząsteczkę.

```python
# aktualizacja nowej prędkości cząsteczki
def update_velocity(self, pos_best_g, i):
    w = 0.8 * i/(i+1)  # wartośc bezwładności
                       # (w tej modyfikacji wartość bezwładności zmniejsza się wraz z liczbą iteracji)
    c1 = 1  # parametr kognitywny
    c2 = 2  # parametr społeczny

    for i in range(0, num_dimensions):
        r1 = random()
        r2 = random()

        vel_cognitive = c1 * r1 * (self.pos_best_i[i] - self.position_i[i])
        vel_social = c2 * r2 * (pos_best_g[i] - self.position_i[i])
        self.velocity_i[i] = comp_factor * (w * self.velocity_i[i] + vel_cognitive + vel_social)
        # dodano współczynnik ścisku (comp_factor)
```

Na powyższym zdjęciu widać, że w metodzie aktualizacji prędkości zostały wprowadzone dwie modyfikacje względem podstawowego algorytmu PSO.

Pierwsza z nich to zmienna wartość bezwładności, która w tym przypadku zmniejsza się z każdą kolejną iteracją algorytmu, co powoduje, że cząsteczki zatrzymują się szybciej im więcej iteracji algorytmu wykonamy. W efekcie oznacza to dla nas, że im więcej iteracji zostanie wykonane tym bliższe prawdy będą otrzymane wyniki.

Druga to wprowadzenie tak zwanego współczynnika ścisku. Jest to wartość, przez którą przemnażana jest cała otrzymana w poprzednich krokach prędkość. Zazwyczaj jest to wartość znajdująca się między 0 a 1 zaproponowana przez Maurice Clerca. Im mniejszy współczynnik ścisku tym wolniej cząstki poruszają się i dokładniejsze rozwiązania znajdują. Jeśli optimum funkcji okaże się być daleko od początkowej pozycji cząsteczki, a współczynnik ścisku spowolni ją za bardzo, może ona nigdy nie dotrzeć do optimum.

```
# aktualizacja pozycji cząsteczki na podstawie aktualizacji prędkości
def update_position(self, bounds):
    for i in range(0, num_dimensions):
        self.position_i[i] = self.position_i[i] + self.velocity_i[i]

        # dostosowanie pozycji cząsteczki jeśli jest większa niż dziedzina
        if self.position_i[i] > bounds[i][1]:
            self.position_i[i] = bounds[i][1]

        # dostosowanie pozycji cząsteczki jeśli jest mniejsza niż dziedzina
        if self.position_i[i] < bounds[i][0]:
            # jeśli cząstka wyjdzie poza dziedzine zostaje ustawiona na granicy podanej dziedziny
            self.position_i[i] = bounds[i][0]
```

Na koniec należy zaktualizować pozycję cząsteczki dodając do poprzedniej pozycji wartość wektora prędkości.

Metoda Minimize

Następnym etapem jest napisanie metody minimalizującej funkcję, korzystającej z klasy Particle opisanej powyżej.

```
def minimize(costFunc, nd, bounds, num_particles, maxiter, comp_fac, verbose=False):

    global num_dimensions, comp_factor
    num_dimensions = nd
    comp_factor = comp_fac

    err_best_g = -1  # najmniejsza wartość funkcji znaleziona w grupie
    pos_best_g = []  # wartości dla których funkcja przyjęła powyższą wartość
```

Metoda ta przyjmuje jako parametry następujące wartości:

- costFunc – funkcja, dla której będzie szukane minimum
- nd – liczba wymiarów w której rozpatrywana będzie funkcja
- bound – dziedzina funkcji
- num_particles – liczba cząsteczek do utworzenia
- maxiter – liczba iteracji, po której metoda skończy się wykonywać
- comp_fac – współczynnik ścisku
- verbose – zmienna typu Boolean określająca czy wyniki każdej iteracji mają być wyświetlane na bieżąco (False – wyświetli tylko wynik końcowy)

Następnie tworzone są zmienne globalne, z których korzysta także klasa Particle oraz inicjowane są pola przechowujące informacje o najlepszym znalezionym punkcie.

Kolejny etap to utworzenie roju cząsteczek i dodanie ich do kolekcji

```
# stworzenie roju
swarm = []
for i in range(0, num_particles):
    swarm.append(Particle(bounds)) # dodaje nową cząsteczkę do roju
```

Po wykonaniu powyższego metoda przechodzi do optymalizacji

```
# pętla optymalizacyjna
i = 0
while i < maxiter:
    if verbose: print(f'iter: {i:>4d}, best solution: {err_best_g:10.6f}')

    # przejście po wszystkich cząsteczkach w roju i sprawdzenie wartości funkcji
    for j in range(0, num_particles):
        swarm[j].evaluate(costFunc)

        # sprawdzenie czy aktualna cząsteczka znalazła najmniejszą wartość (globalnie)
        if swarm[j].err_i < err_best_g or err_best_g == -1:
            pos_best_g = list(swarm[j].position_i)
            err_best_g = float(swarm[j].err_i)

    # przejście po wszystkich cząsteczkach w roju w celu aktualizacji pozycji i prędkości
    for j in range(0, num_particles):
        swarm[j].update_velocity(pos_best_g, i)
        swarm[j].update_position(bounds)
    i += 1
```

A na koniec wyświetla wyniki na ekranie

```
# wyświetlenie wyników
print('\nFINAL SOLUTION:')
print(f'   pozycja> {pos_best_g}')
print(f'   optimum globalne> {err_best_g}\n')

pass
```

3. Strojenie parametrów i eksperymenty

W celu osiągnięcia najlepszej wydajności zaimplementowanego algorytmu należało odpowiednio dopasować zestaw znajdujących się w nim parametrów. Parametry wymagające strojenia:

- liczba cząsteczek c
- liczba iteracji i
- współczynnik kognitywny ϕ_1
- współczynnik społeczny ϕ_2
- współczynnik bezwładności (inercji) ω
- współczynnik ścisku χ

Podstawowy zestaw parametrów dobrano na podstawie publikacji [1], której jednym z autorów jest twórca algorytmu PSO - James Kennedy. W pracy tej następująco określono sugerowane wartości parametrów:

$$\phi_1=\phi_2=2$$

oraz zaznaczono że $\phi_1+\phi_2\leqslant 4$

Wartość współczynnika inercji powinna maleć liniowo wraz z kolejnymi iteracjami od $\omega=0.9$ do $\omega=0.4$. Są to sugerowane parametry dla klasycznej wersji algorytmu – bez współczynnika ścisku.

Dla wersji z współczynnikiem ścisku autorzy publikacji proponują wartości:

$$\chi=0.7842 \qquad \phi_1=\phi_2=1.49618 \qquad \omega=0.7298$$

Każdy zestaw parametrów będzie sprawdzany na obu funkcjach testowych. Kolejne testy będą przeprowadzane dla 3 wariantów liczby cząsteczek oraz iteracji.

wariant	Liczba iteracji	Liczba cząsteczek
1	30	15
2	50	30
3	100	50

A) $\phi_1=\phi_2=2$, stały współczynnik inercji $\omega=0.5$

```
bounds_ackley = [(-32,32),(-32,32)]
bounds_schwefel = [(-500,500),(-500,500),(-500,500)]
numParticles = [15, 30, 50]
maxIter = [30, 50, 100]
verbose = False
compression_factor = 1 # 0.7842

for i in range(0, 3):
    print("Wariant: ", i)
    print("Ackley: ")
    pso.minimize(functions.ackley, 2, bounds_ackley, numParticles[i], maxIter[i], compression_factor, verbose)
    print("Schwefel: ")
    pso.minimize(functions.schwefel, 3, bounds_schwefel, numParticles[i], maxIter[i], compression_factor, verbose)
```

Wariant	Ackley	Schwefel
1	pozycja: ['0.0018', '-0.0016'] optimum globalne: 0.007081	pozycja: ['-301.6407', '-301.4495', '419.7326'] optimum globalne: 237.314771
2	pozycja: ['-0.0004', '-0.0004'] optimum globalne: 0.001526	pozycja: ['420.9663', '-302.5271', '-500.0000'] optimum globalne: 356.832103
3	pozycja: ['0.0000', '-0.0000'] optimum globalne: 0.000000	pozycja: ['420.9687', '420.9688', '420.9688'] optimum globalne: 0.000038

B) $\phi_1=\phi_2=2$, współczynnik ω maleje od 0,9 do 0.4

Wariant	Ackley	Schwefel
1	pozycja: ['-0.0787', '0.0894'] optimum globalne: 0.681123	pozycja: ['420.0149', '433.4082', '-300.9891'] optimum globalne: 138.320021
2	pozycja: ['0.0013', '0.0015'] optimum globalne: 0.005809	pozycja: ['421.1354', '421.0245', '421.0269'] optimum globalne: 0.004362
3	pozycja: ['-0.0000', '-0.0000'] optimum globalne: 0.000055	pozycja: ['420.9688', '420.9700', '420.9706'] optimum globalne: 0.000039

C) $\chi=0.7842$ $\phi_1=\phi_2=1.49618$ $\omega=0.7298$

Wariant	Ackley	Schwefel
1	pozycja: ['0.0000', '0.0063'] optimum globalne: 0.018732	pozycja: ['-302.5819', '-302.5503', '-302.5099'] optimum globalne: 355.315562
2	pozycja: ['-0.0000', '-0.0000'] optimum globalne: 0.000048	pozycja: ['-302.5251', '-302.5246', '420.9688'] optimum globalne: 236.876707
3	pozycja: ['0.0000', '0.0000'] optimum globalne: 0.000000	pozycja: ['420.9687', '420.9688', '420.9688'] optimum globalne: 0.000038

D) $\chi=0.7842$, $\phi_1=\phi_2=1.49618$ ω maleje od 0,9 do 0.4

Wariant	Ackley	Schwefel
1	pozycja: ['0.0004', '0.0008'] optimum globalne: 0.002569	pozycja: ['-302.4712', '420.9706', '-302.5369'] optimum globalne: 236.877092
2	pozycja: ['0.0000', '0.0000'] optimum globalne: 0.000016	pozycja: ['420.9691', '-302.5248', '-302.5247'] optimum globalne: 236.876707
3	pozycja: ['-0.0000', '-0.0000'] optimum globalne: 0.000000	pozycja: ['420.9687', '420.9687', '-302.5249'] optimum globalne: 118.438373

E) $\chi=0.7842$, $\phi_1=1$, $\phi_2=2$, ω maleje od 0,9 do 0,4

W tej konfiguracji cząsteczki silniej kierują się na optimum globalne

Wariant	Ackley	Schwefel
1	pozycja: ['-0.0003', '-0.0000'] optimum globalne: 0.000870	pozycja: ['203.8219', '420.9385', '420.9376'] optimum globalne: 217.139953
2	pozycja: ['0.0000', '-0.0000'] optimum globalne: 0.000011	pozycja: ['420.9688', '-302.5250', '420.9687'] optimum globalne: 118.438373
3	pozycja: ['-0.0000', '0.0000'] optimum globalne: 0.000000	pozycja: ['420.9687', '420.9687', '420.9687'] optimum globalne: 0.000038

F) $\chi=0.7842$, $\phi_1=2$, $\phi_2=1$, ω maleje od 0,9 do 0,4

W tej konfiguracji cząsteczki silniej kierują się na optimum lokalne

Wariant	Ackley	Schwefel
1	pozycja: ['0.0004', '0.0002'] optimum globalne: 0.001248	pozycja: ['423.9008', '-294.5300', '425.1609'] optimum globalne: 129.767245
2	pozycja: ['0.0000', '-0.0000'] optimum globalne: 0.000000	pozycja: ['420.9689', '420.9687', '420.9687'] optimum globalne: 0.000038
3	pozycja: ['-0.0000', '-0.0000'] optimum globalne: 0.000000	pozycja: ['420.9687', '420.9687', '420.9687'] optimum globalne: 0.000038

- Obserwacje i wnioski

Łatwo można zauważyć że im większa liczba cząsteczek i iteracji tym dokładniejszy wynik uzyskujemy.

Dla analizowanej dwuwymiarowej funkcji Ackleya algorytm za każdym razem znalazł przybliżone optimum globalne, a czasem nawet dokładne optimum. Dla tej funkcji najbardziej obiecujące wydają się warianty parametrów E i F.

Trójwymiarowa funkcja Schwefela wykazuje większą złożoność i tutaj wyniki są bardziej nieprzewidywalne. W wielu przypadkach PCA nie znalazło optimum globalnego, ani razu nie udało się dla wariantu 1 (mało cząstek i iteracji). Dla tej funkcji najbardziej optymalne są warianty parametrów B oraz F.

Bibliografia

[1] Poli, R., Kennedy, J., & Blackwell, T. (2007). Particle swarm optimization. Swarm intelligence

[2] The Parameters Selection of PSO Algorithm influencing On performance of Fault Diagnosis, MATEC Web of Conferences 63, 02019 (2016)

[3] Analysis of the PSO parameters for a robots positioning system in SSL, Marcos Aurelio Pchek Laureano1,2[0000−0002−9399−7633] and Flavio Tonidandel, 2019

[4] https://www.sfu.ca/~ssurjano/optimization.html