Optymalizacja: Metoda optymalizacji rojem cząstek (PSO) i prostych poszukiwań losowych

Zespół

Damian Kolaska
Tomasz Słojewski
Łukasz Reszka

Problem

Celem zadania jest zbadanie dwóch algorytmów przeszukiwania pod kątem możliwości zrównoleglenia.

- proste poszukiwanie losowe
- optymalizacja rojem cząstek (PSO)

Proste przeszukiwanie losowe

Metoda polega na **wielokrotnym losowaniu punktów** w przestrzeni poszukiwań. W każdej iteracji losujemy *n* punktów, z których wybieramy ten, dla którego funkcja celu przyjmuje najlepszą wartość (w naszych zadaniach - najmniejszą).

Współrzędne punktów są losowane z rozkładem jednostajnym. Metoda bardzo dobrze się zrównolegla z racji na niezależność poszczególnych losowań.

Optymalizacja rojem cząstek

Algorytm polega na stworzeniu pewnej ilości ***cząstek***, które znajdują się początkowo w losowych położeniach. Każda z nich ma szereg atrybutów:

- położenie,
- prędkość,
- aktualną jakość,
- swoje najlepsze położenie,
- zbiór sąsiadów,
- najlepsze odnalezione przez sąsiadów położenie.

W naszym rozwiązaniu zastosowaliśmy **sąsiedztwo globalne**, aby nie spowalniać algorytmu poprzez wyszukiwanie sąsiadów i sprawdzania za każdym razem, który z nich ma najlepsze położenie.

Prędkość każdej z *cząstek* jest **kombinacją trzech wartości**:

- poprzedniej prędkości cząstki,
- kierunku do najlepszej swojej pozycji,
- kierunku do najlepszej globalnie pozycji.

$v = c_1 r_1 v + c_2 r_2 (pbest - x) + c_3 r_3 (gbest - x)$, gdzie:

- v - prędkość *cząsteczki*
- x - położenie *cząsteczki*
- $pbest$ - najlepsze znalezione przez *cząsteczkę* położenie
- $gbest$ - najlepsze znalezione przez zbiór sąsiadów położenie
- c_1, c_2, c_3 - współczynniki określające wpływ poszczególnych elementów
- r_1, r_2, r_3 - wartości losowe z rozkładu jednostajnego $U(0, 1)$

Dodatkowo, w celu polepszenia własności eksploracyjnych algorytmu w dalszych etapach działania, wartość c_1 zmniejsza się w czasie: $c_{1(n)} = 0,992c_{1(n-1)}$. Ostateczny wzór na prędkość *cząstki* w *n*-tej iteracji ma więc postać:

$$v_{(n)} = c_{1(n)}r_1v_{(n-1)} + c_2r_2(pbest–x) + c_3r_3(gbest–x)$$

Cząstka w każdej iteracji:

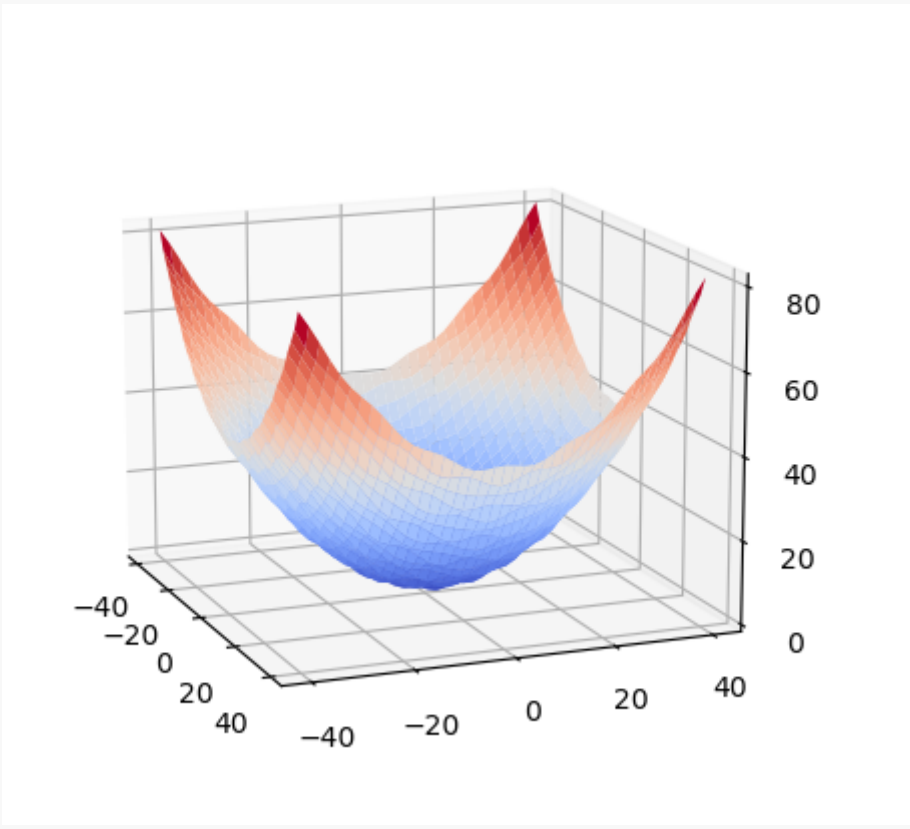
1. porusza się o wartość prędkości,
2. sprawdza, czy nie znalazła się w swojej najlepszej pozycji lub w pozycji najlepszej globalnie,
3. aktualizuje swoją prędkość.

Jeśli prędkość ma taką wartość, że doprowadza ona do wyjścia *cząstki* **poza dziedzinę**, to stosujemy ***odbijanie***, tzn. *cząstka* jest cofana od granicy dziedziny o taką wartość, o jaką ją przekroczyła.

Funkcje celu

Zadanie 1.

$min_x(f(x) = \frac{1}{40} * \sum_{i=1}^{n}(x_i^2) + 1 - \prod_{i=1}^{n} cos(\frac{x_i}{i}))$

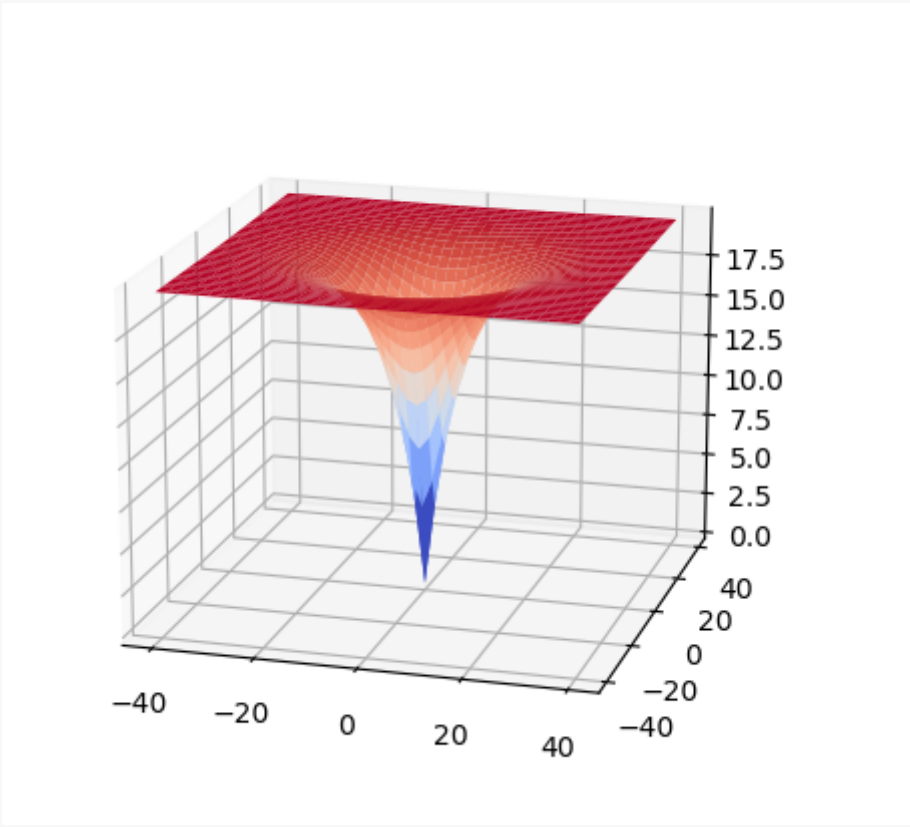


Rysunek 1. Wykres funkcji z zadania 1.

Zadanie 2.

$$min_x(f(x) = -20exp(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}) - exp(\frac{1}{n}\sum_{i=1}^{n} cos(2\pi x_i)) + 20 + e)$$

> W treści zadania przy pierwszym eksponencie nie było znaku minus. Jednak wtedy funkcja w punkcie (0, 0) posiadała maksimum, a nie minimum.



Rysunek 2. Wykres funkcji z zadania 2.

Implementacja

Repozytorium

Kod został umieszczony w repozytorium na platformie GitHub.
https://github.com/GRO4T/PORR_particle_swarm_optimization_in_OpenMP

Struktura katalogów

- doc - folder poświęcony na dokumentację projektu
- include - folder na pliki nagłówkowe
- src - folder na pliki źródłowe
- test_scripts - folder na skrypty testowe

Kompilacja rozwiązania jest opisana w pliku README.md.

Pliki źródłowe

- programs/
  - plotter.cpp - źródła programu plotter (patrz. *Pliki binarne*)
  - runner.cpp - źródła programu runner (patrz. *Pliki binarne*)
- plots.cpp - funkcje pomocnicze do generowania wykresów

- random_search.cpp - implementacja algorytmu prostego przeszukiwania losowego
- swarm_search.cpp - implementacja algorytmu optymalizacji rojem cząstek
- test_functions.cpp - implementacje testowanych funkcji celu

Pliki binarne

Po kompilacji w odpowiednim katalogu pojawią się dwa programy.

runner

Program służący do uruchamiania algorytmów bez środowiska graficznego.

Użycie

```
$ ./runner -h
usage: runner [-h | --help] [-n | --dimension] DIMENSION [-t | --threads] THREADS
[-i | --iterations] ITERATIONS [-p | --particle] PARTICLE_COUNT [-s | --search]
SEARCH_ALGORITHM [-f | --obj_func] OBJ_FUNC
[-m | --time] TIME [-l | --threshold] THRESHOLD
        dimension - number of dimensions of test function
        threads - number of threads
        iter - finish after X iterations
        search - search algorithm (random or swarm)
        obj_func - id of test function (1 or 2)
        time - finish after X seconds
        threshold - finish after best result reaches X
```

Przykładowe wykonanie

```
$ ./runner -n 50 -t 16 -i 10000 -s swarm
---------------------------------------
n: 50
threads: 16
iterations: 10000
search_algorithm: swarm
objective_func_id: 1
---------------------------------------
Search: swarm
Result:
{
    "value": -5.36871e+07
    "position": [-2.3873, -3.4641, 0.00195812, -0.0325867, 0.00396919, 0.358742,
33.9093, 0.290891, 0.0054658, -0.468637, -0.248944, 4.58262, -13.6362, -3.17518,
0.843873, -7.43172, -1.60979, -1.11461, 7.82605, 1.53818, -24.5503, 2.25529,
4.14486, -9.64952, 25.9706, 18.8753, 7.55107, 0.297182, 17.6965, -3.62814,
-32.9565, 26.0179, 19.5599, -7.37969, 20.8117, -37.4399, -0.94533, 0.205607,
4.79949, 115547, 29.1306, 1.55907, 1.01852, 4.64064, -10.3429, 1.61431, -35.8724,
11.6137, -0.386012, 2.09328, ]
    "exec_time_in_nanos": 1043979417
    "exec_time": 1.04398
}
```

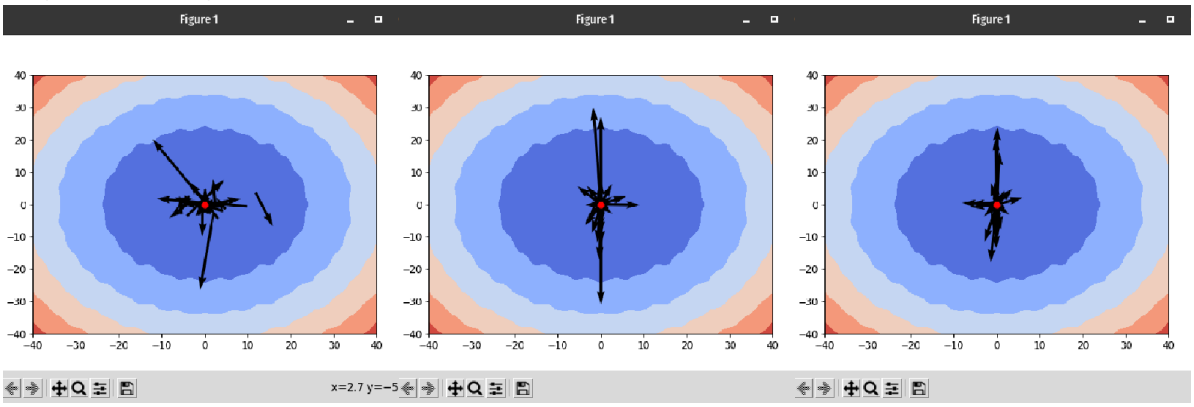
plotter

Program służący do graficznego obrazowania działania algorytmów.

Użycie

```
$ ./plotter -h
usage: plotter [-h | --help] [-n | --dimension] DIMENSION
[-i | --iterations] ITERATIONS [-s | --search] SEARCH_ALGORITHM [-f | --obj_func]
OBJ_FUNC
```

Przykładowe użycie

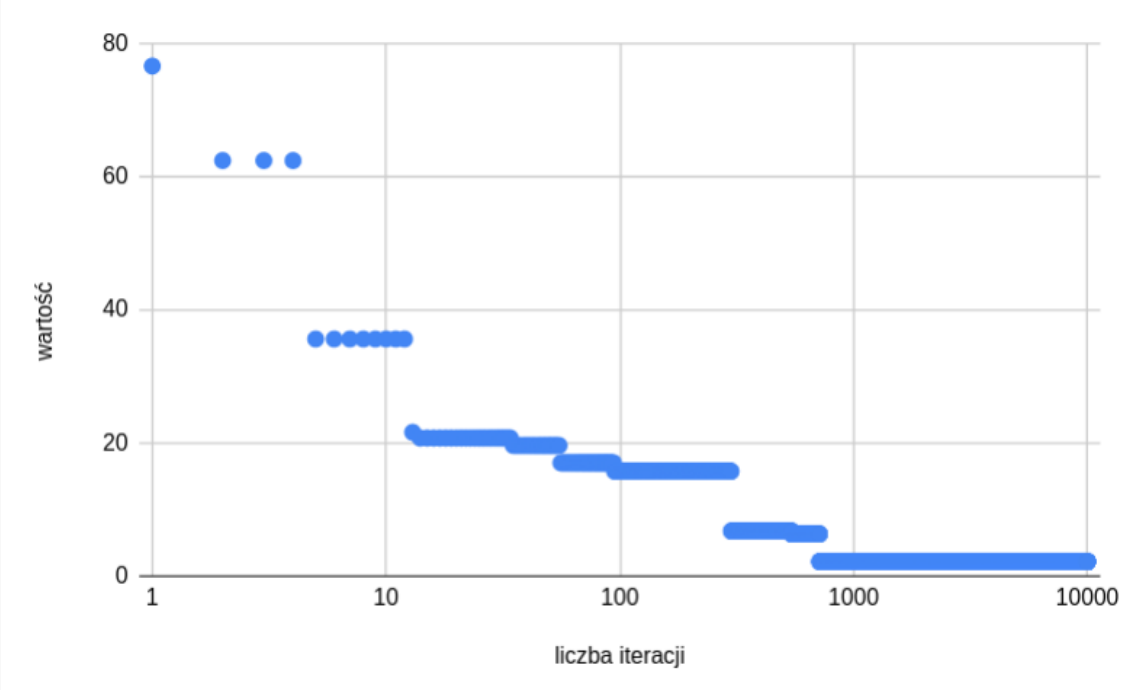


Eksperymenty

Algorytmy zostały zaimplementowane w języku C++.
Do tworzenia wykresów używamy biblioteki matplotlib.

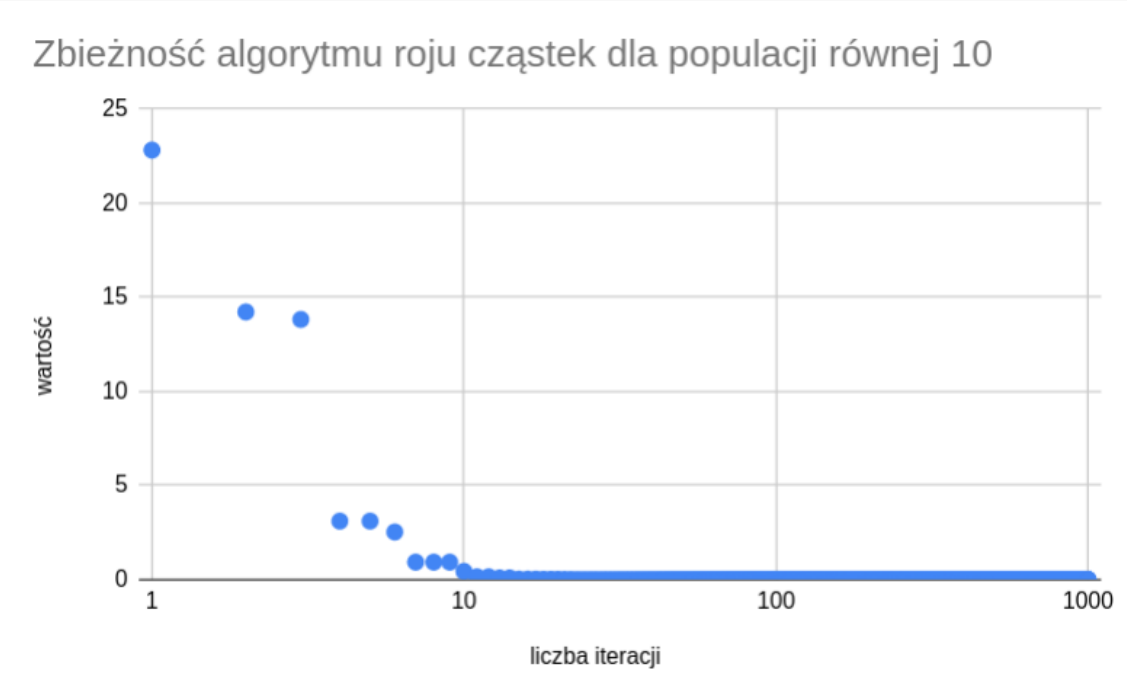
Reprezentacja graficzna zbieżności algorytmów (n=2)

Proste przeszukiwanie losowe



Rysunek 3. Wykres zbieżności algorytmu prostego przeszukiwania losowego.

Optymalizacja rojem cząstek



Rysunek 4. Wykres zbieżności algorytmu optymalizacji rojem cząstek.

Najlepsze rozwiązanie po określonym czasie dla różnych wymiarów problemu

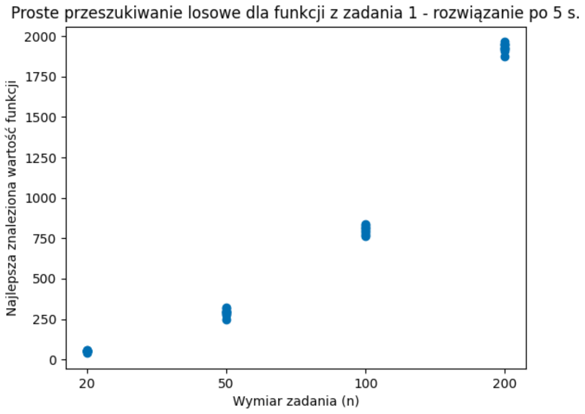
W tym eksperymencie uruchamialiśmy algorytmy z zadanym czasem t. Każdy algorytm był uruchamiany 10-krotnie w danej konfiguracji.

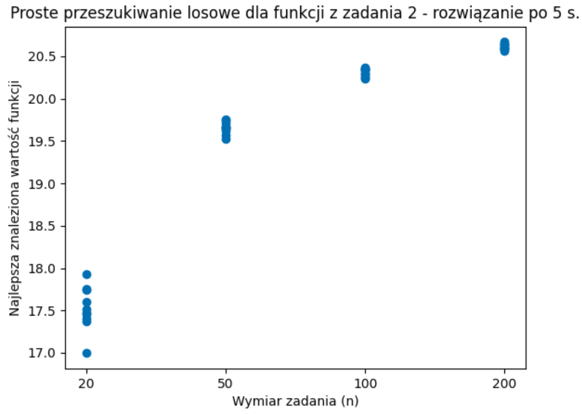
W pierwszych wierszach mamy wykresy dla algorytmu przeszukiwania losowego.
Następny wiersz pokazuje wyniki dla algorytmu roju cząstek dla populacji równej 100.
Trzeci wiersz wykresów przedstawia wyniki działania algorytmu roju cząstek dla wielkości roju równej $100 * n$, gdzie n oznacza wymiarowość zadania.

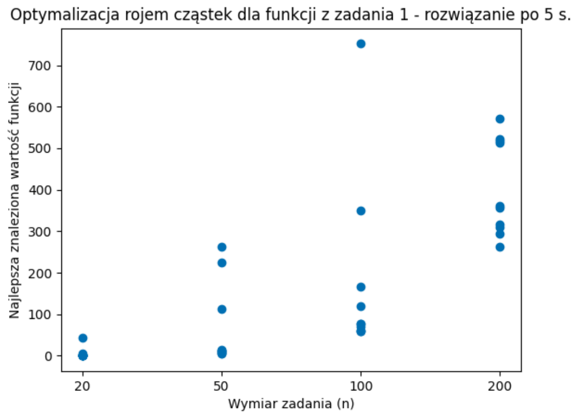
Znalezione rozwiązanie po 5 sekundach

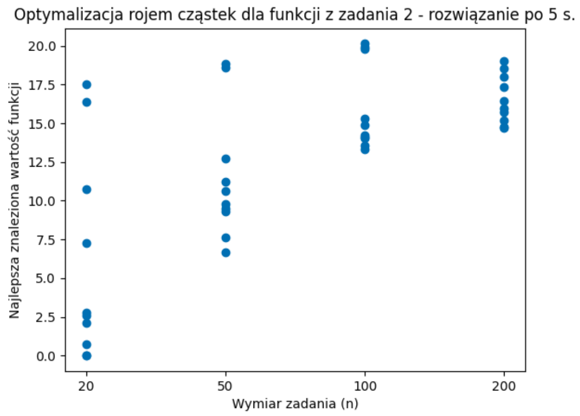
Widzimy, że już po 5 sekundach jesteśmy w stanie znaleźć wyniki bardzo zbliżone do minimum globalnego dla niewielkiej wymiarowości zadania.
Jedynie w przypadku prostego przeszukiwania losowego dla funkcji z zadania 2. nie zbliżyliśmy się do minimum globalnego funkcji.
Już na tym etapie widzimy znaczną różnicę między wynikami dla populacji równej 100, a populacji

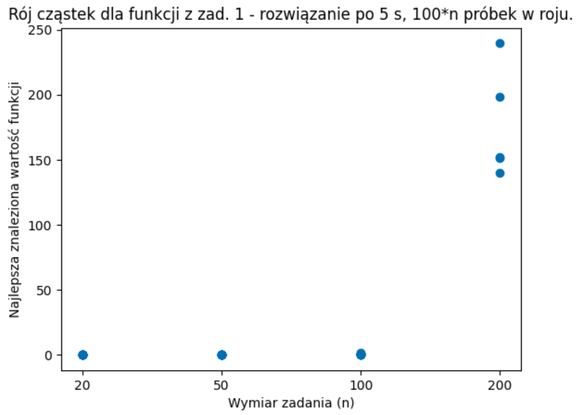
równej $100 * n$.

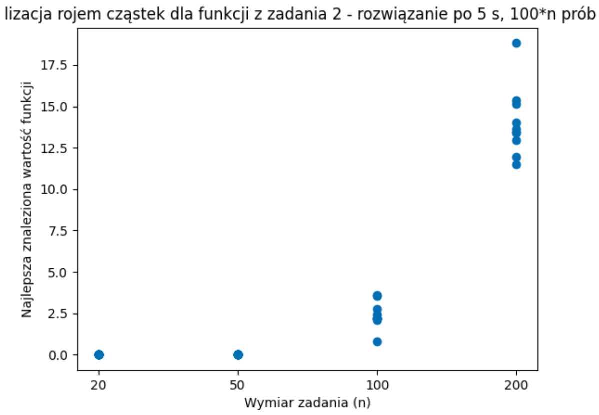








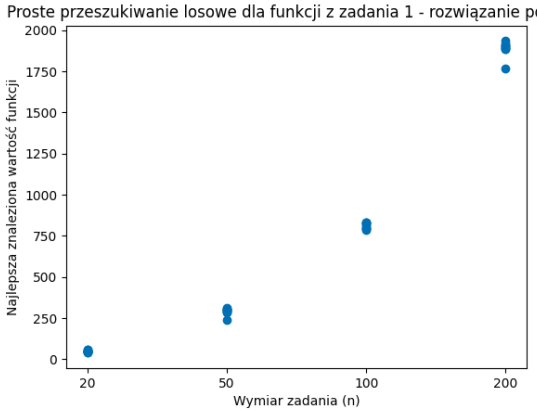




Znalezione rozwiązanie po 10 sekundach

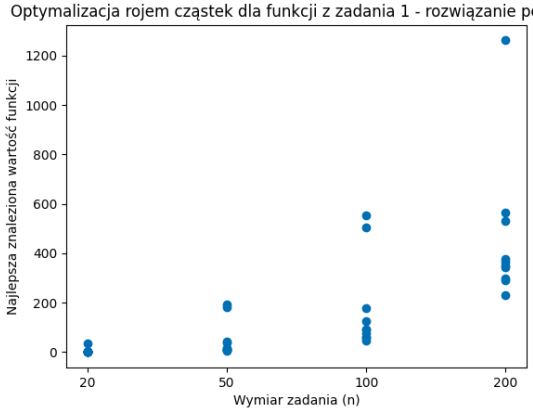
Po 10 sekundach algorytm optymalizacji roju cząstek z populacją $100 * n$ znacznie zwiększa jakość swoich wyników.
Dla roju cząstek z populacją 100 i dla wyszukiwania losowego nie odnotowujemy znacznej poprawy wyników.

Proste przeszukiwanie losowe dla funkcji z zadania 1 - rozwiązanie po 10 s.

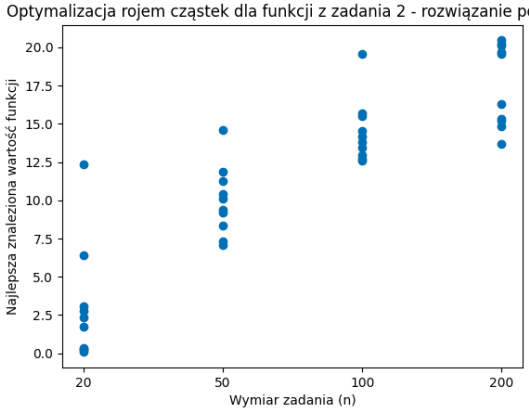


Proste przeszukiwanie losowe dla funkcji z zadania 2 - rozwiązanie po 10 s.

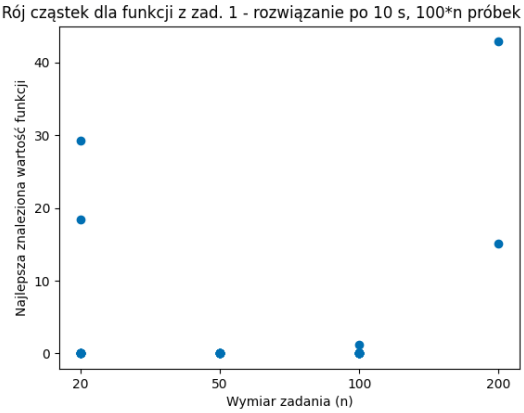
Optymalizacja rojem cząstek dla funkcji z zadania 1 - rozwiązanie po 10 s.



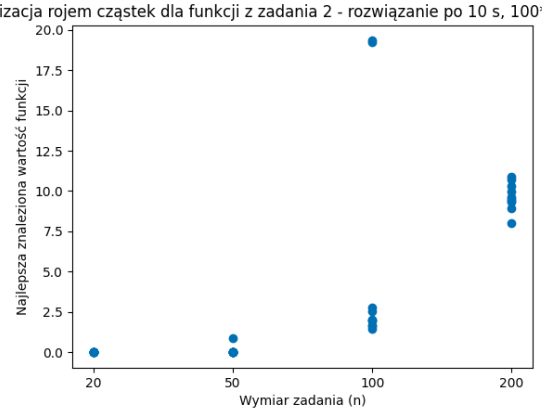
Optymalizacja rojem cząstek dla funkcji z zadania 2 - rozwiązanie po 10 s.



Rój cząstek dla funkcji z zad. 1 - rozwiązanie po 10 s, 100*n próbek w roju.

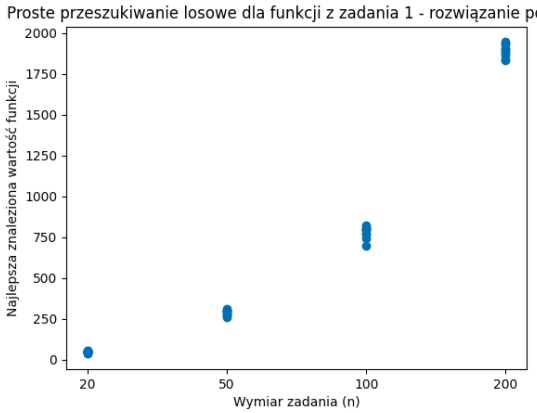


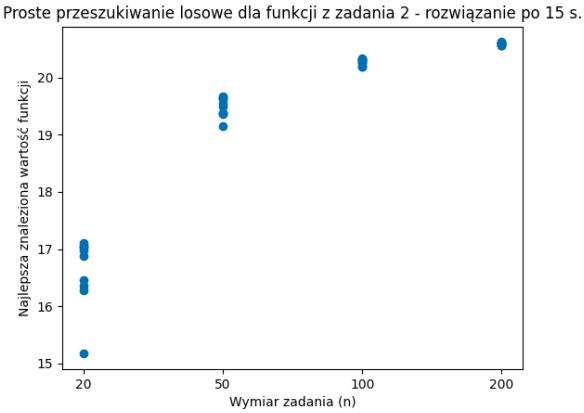
izacja rojem cząstek dla funkcji z zadania 2 - rozwiązanie po 10 s, 100*n prób

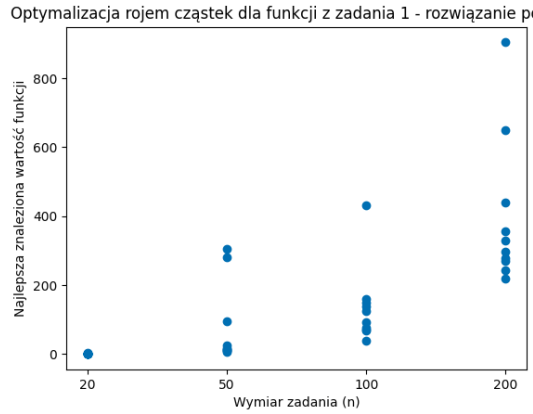


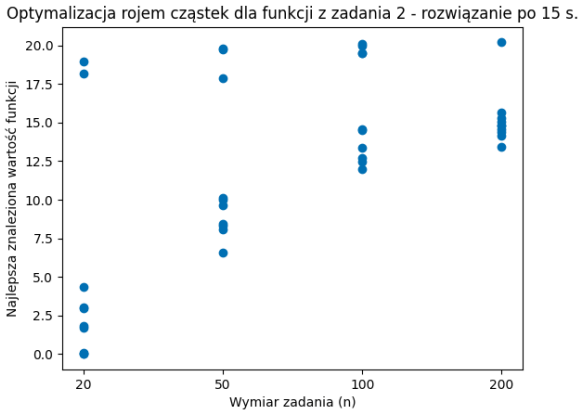
Znalezione rozwiązanie po 15 sekundach

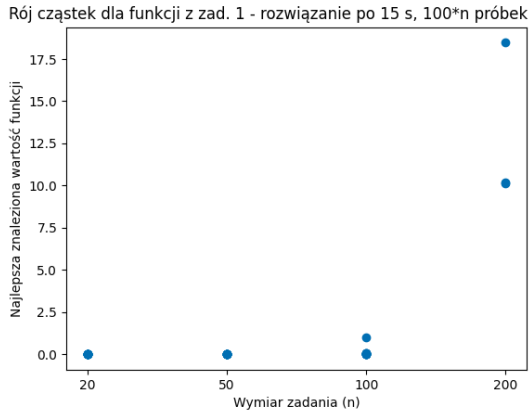
Widzimy, że kolejne 5 sekund uruchomienia algorytmów znaczną poprawę tylko w przypadku roju cząstek o populacji $100 * n$ dla zadania nr 1.

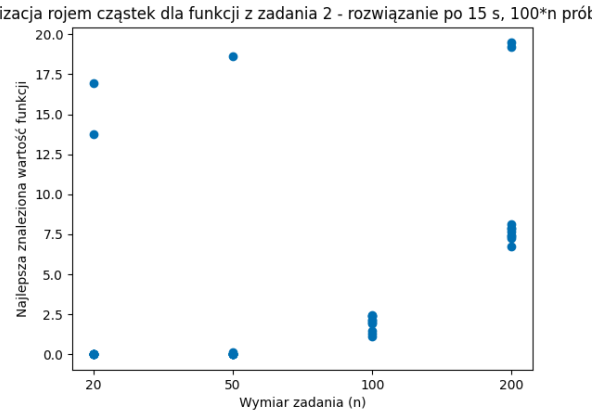






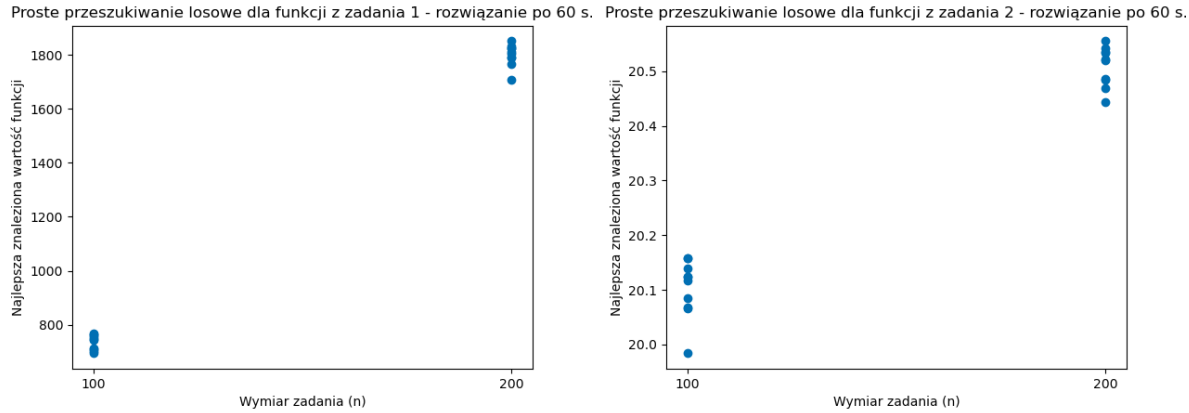


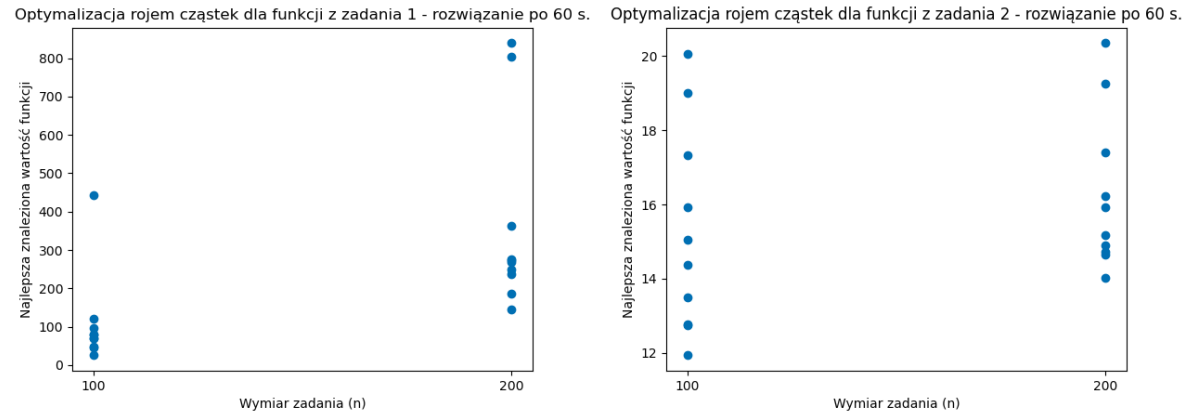


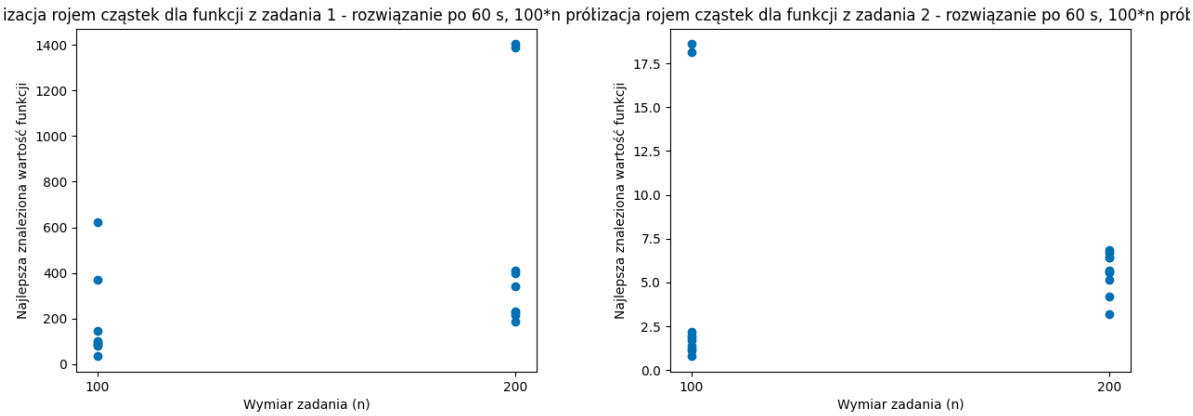


Znalezione rozwiązanie po 60 sekundach

Jako ostatni z testów postanowiliśmy uruchomić algorytmy ze zwiększonym budżetem czasowym, ale jedynie dla wymiarów problemu 100 oraz 200. Widzimy, że wartości są znikomo lepsze niż w poprzednich testach.

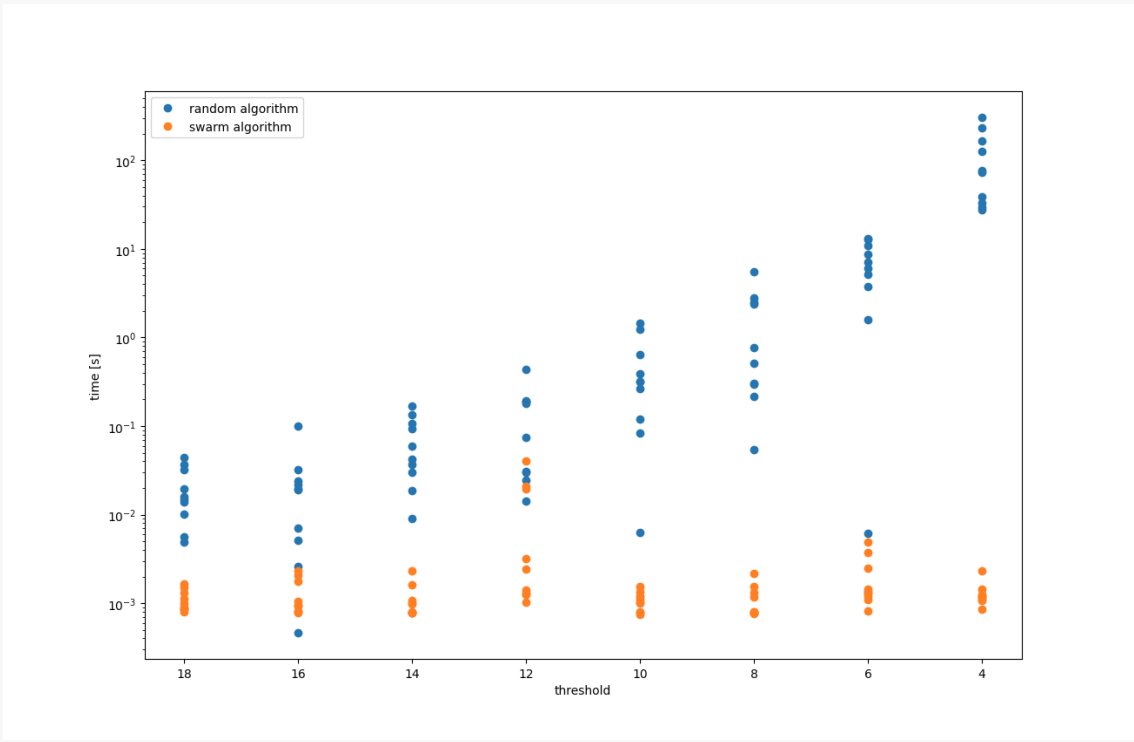




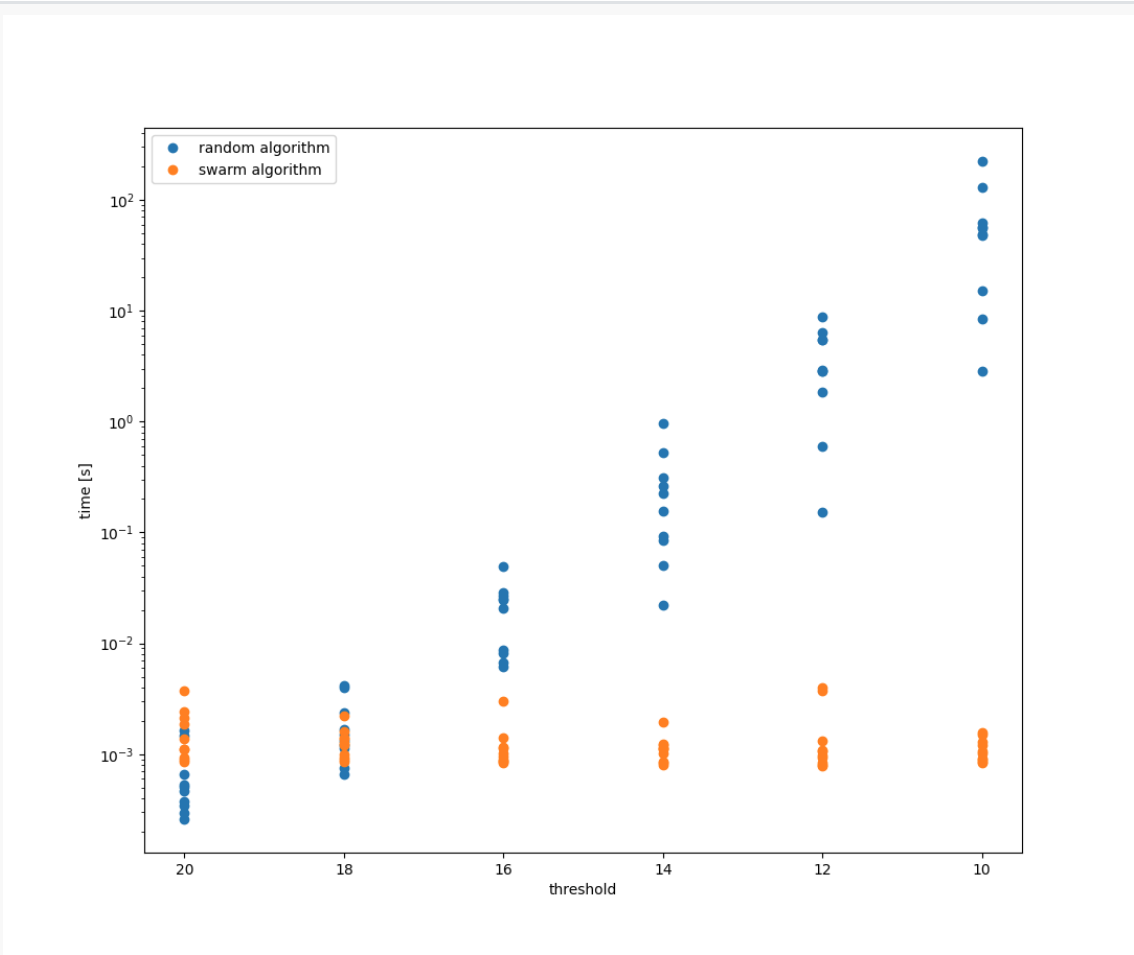


Najszybsza zbieżność

Z racji, że z góry znaliśmy poszukiwaną wartość optymalną, dlatego też postanowiliśmy przeprowadzić testy pokazujące jak szybko każdy z algorytmów będzie zmierzał do optimum globalnego. Testy te polegają na zmierzeniu czasu działania programu aż do osiągnięcia minimum z pewną dokładnością.



Rysunek 17. Wykres pokazujący czasy osiągnięcia dokładności wyniku. Funkcją testową wyła funkcja nr 1, wymiarowość n = 10. Dla każdego z algorytmów i progów przeprowadzono 10 prób.



Rysunek 18. Wykres pokazujący czasy osiągnięcia dokładności wyniku. Funkcją testową wyła funkcja nr 2, wymiarowość n = 10. Dla każdego z algorytmów i progów przeprowadzono 10 prób.

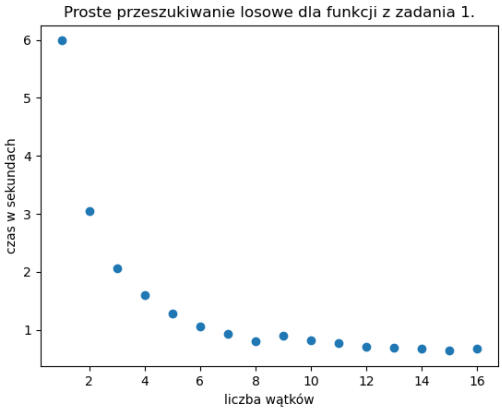
Dla dużego progu widzimy, że czas działania obu algorytmów jest bardzo krótki i zbliżony do siebie.
Dla mniejszego progu czas działania algorytmu przeszukiwania losowego rośnie znacząco, zaś czas działania algorytmu roju cząstek jest podobny.
Wynika to z tego, że wydłużenie czasu działania algorytmu roju cząstek moglibyśmy zaobserwować przy o wiele mniejszym progu, jednakże dla takich wartości działanie algorrytmu przeszukiwania losowego byłoby nieakceptowalnie długie.

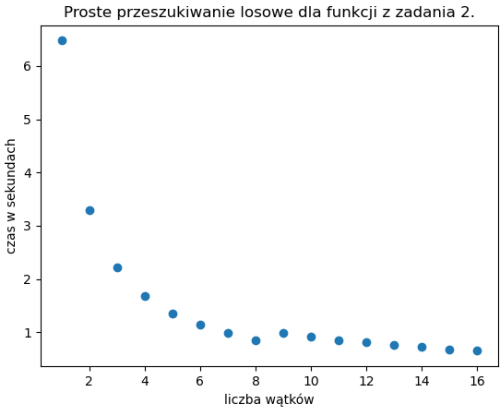
Przyspieszenie obliczeń przy zrównolegleniu

W tym eksperymencie testowaliśmy oba algorytmy pod kątem przyspieszenia obliczeń przy zrównolegleniu.
Eksperyment polegał na uruchamianiu algorytmu ze zwiększającą się liczbą wątków przy stałej liczbie iteracji i wymiarze problemu.

Proste przeszukiwanie losowe

wymiar problemu: 50
liczba iteracji: 1000000



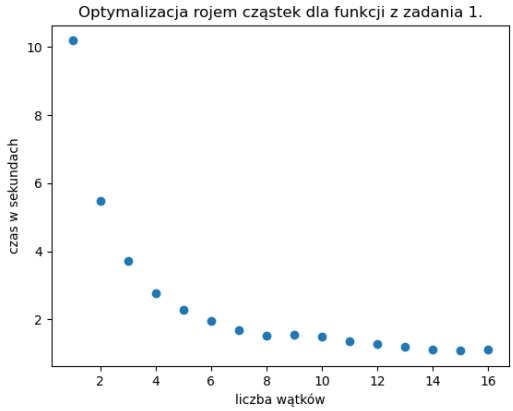


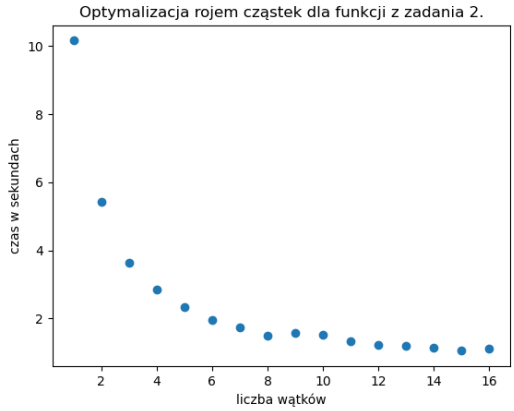
Współczynniki przyspieszenia

	p=2	p=3	p=4	p=5	p=6	p=7	p=8
zad 1	1.968	2.892	3.756	4.658	5.625	6.442	7.398
zad 2	1.964	2.908	3.868	4.803	5.671	6.572	7.553

Optymalizacja rojem cząstek

wymiar problemu: 50
liczba iteracji: 10000





Współczynniki przyspieszenia

	p=2	p=3	p=4	p=5	p=6	p=7	p=8
zad 1	1.858	2.734	3.686	4.455	5.184	6.044	6.738
zad 2	1.870	2.788	3.550	4.351	5.195	5.876	6.762

Nierówność S(n, p) <= p jest spełniona.

Wnioski

Zrównoleglenie obu algorytmów okazało się skuteczne. Uzyskane współczynniki przyspieszenia są niewiele mniejsze od wartości p - ilości wątków.
Zrównoleglenie algorytmu przeszukiwania losowego okazało się trochę skuteczniejsze niż algorytmu roju cząstek. Jest to spowodowane naturą algorytmu przeszukiwania losowego, który pozwala na całkowite uniezależnienie od siebie kolejnych losowań.
Jednakże współczynniki przyspieszenia w przypadku algorytmu roju cząstek są tylko nieznacznie mniejsze i są na zadowalającym poziomie.

Porównanie dwóch algorytmów pokazuje, że algorytm roju cząstek jest znacznie skuteczniejszy dla obu zadań. Jednakże ważne jest dobranie odpowiedniej wielkości populacji, ponieważ ma ona duży wpływ na osiągnięte wyniki.