Radzikowski Kacper
Żelazowski Michał

Dokumentacja końcowa

Implementacja algorytmu ewolucji różnicowej oraz algorytmu wspinaczkowego z wykonaniem testów dla funkcji sferycznej oraz funkcji Goldsteina-Price'a.

Niniejszą dokumentację postanowiliśmy podzielić na dwie sekcje w których z osobna każdy z algorytmów został opisany wraz z przeprowadzonymi testami optmalizacji jego użycia dla określonych funkcji. Na końcu dokumentu znajduje się porównanie algorytmów.

1. Implementacja algorytmu różnicowego

Implementacja algorytmu

Ustalone parametry uruchomieniowe algorytmu to :
- wielkość populacji S
- waga parametru F określającą wpływ losowej różnicy
- stała operacji krzyżowania CR

Ogólny schemat działania algorytmu

$inicjuj\ P(0) \leftarrow \{x_1, x_2, \ldots, x_3\}$
$t \leftarrow 0$
$while\ !stop$
    $for(i\ \varepsilon\ 1 : \mu)$
        $x_j \leftarrow select(P(t))$
        $x_k, x_l \leftarrow sample(P(t))$
        $y \leftarrow x_j + F(x_k - x_l)$
        $z \leftarrow crossover(x_i, y)$
        $x_i \leftarrow tournament(x_i, z)$
    $t \leftarrow t + 1$

Zasada działania algorytmu:
- (1) wygeneruj populacje początkową zgodnie z ograniczeniami lub bez (została zaimplementowana możliwość generowania nowej populacji dla każdego uruchomienia oraz podania własnej) oraz zapisz do historii
- (2) zapisz do modelu pozycje potomstwa w historii
- (3) dopóki nie został osiągnięty warunek stopu (ilość iteracji lub alternatywnie brak wzrostu jakości najgorszego potomka z populacji) wykonuj
    - dla każdego kolejnego potomka:
        - (a) wylosuj zgodnie z rozkładem jednostajnym element z populacji (X_j)
        - (b) wylosuj bez zwracania dwa elementy z populacji(X_k i X_l) i oblicz ich różnicę (utwórz wektor różnicowy)
        - (c) do potomka X_j dodaj przeskalowany wektor różnicowy (X_k - X_l): otrzymany zostaje mutant $Y = X_j + F*(X_k - X_l)$, gdzie F to współczynnik skalowania
        - (d) dokonaj operacji krzyżowania potomka X_i z mutantem Y (wynik zawsze różny od rodzica X_i)
        - (e) sprawdź, czy otrzyamany potomek spełnia założenia o ograniczeniach, jeśli nie, to powtórz operacje od zaczynając od operacji (b)
        - (f) porównaj otrzymanego potomka z potomkiem X_i, gdy lepszy, to zaktualizuj model dla danej pozycji w populacji i w przypadku gdy dany punkt nie występuje w historii, dopisz go do historii.

Ocena jakości rozwiązania związana jest z znalezieniem punktu z wartością minimalną rozpatrywanej funkcji (dokładnie punktu z jak najmniejszego sąsiedztwa punktu wartości minimalnej)

Został zaimplementowany algorytm ewolucji różnicowej określany jako DE/rand/1/bin, co oznacza

że selekcja wybiera losowo osobnika z populacji, którego podda mutacji wektorem różnicowym. Mutacja tak wylosowanego punktu odbywa się za pomocą jednej pary różnicowanych punktów przeskalowanych wartością parametru algorytmu F.

Algorytm ten posiada zaimplementowany model w postaci tablicy, który przechowuje S wartości całkowitoliczbowych, które stanowią pozycję członków aktualnej populacji w historii. Historia jest to lista, która przechowuje wszystkie wygenerowane punkty, czyli punkty startowe algorytmu oraz wszystkie zwrócone przez funkcję *tournament*.

Zaimplementowany algorytm został wyposażony w obsługę ograniczeń zakresu zmiennych funkcji w postaci mechanizmu, który wymusza ponowną generację punktu w przypadku wygenerowania punktu spoza określonego zakresu. Metoda ta została wybrana dla tego algorytmu ze względu na przewidywane najlepsze rezultaty. Metoda ta nie zaburza nam mechanizmu tworzenia punktu w takim zakresie jak np. rzutowanie na granice zakresu. W przypadku rzutowania proces wariacji doprowadził by do „przyklejenia się" populacji do granicy. Użycie mechanizmu odbicia spowodowało by rozlegulowanie chmury punktów. Użycie mechanizmu ponownej generacji w tym przypadku nie dotyczy selekcji (w rozumieniu wyboru pierwszego punktu przy tworzeniu mutanta) a tylko powtórzeniu operacji wariacji.

Zatrzymanie algorytmu.
Postanowiłem wykonać dwa mechanizmy zatrzymania algorytmu, co pozwoliło nam lepiej przeanalizować działanie algorytmu dla podanych funkcji.

Zatrzymanie po wykonaniu n iteracji umożliwi nam zaprezentowanie zależności pomiędzy jakością otrzymanego wyniku przy ustalonej ilości iteracji.

Zatrzymanie w związku z „stagnacją" populacji. Przyjąłem, inaczej niż zostało to stwierdzone w dokumentacji wstępnej, że wyznacznikiem stagnacji populacji będzie maksymalna poprawa jakości najgorszego osobnika populacji (nie musi być to porównanie potomków w linii przez rodziców niemutantów do potomka) na przestrzeni n iteracji. Jeśli poprawa przekroczy pewien ustalony próg, początek „okna" rozpatrywanych iteracji przesunie się do aktualnej iteracji. Nieosiągnięcie poprawy jakości w zakresie rozpatrywanego okna iteracji kończy algorytm. Metoda ta pozwoli nam prezentować wyniki szybkości osiągania wyniku (czyli stagnacji blisko wartości poszukiwanej) oraz moment wystąpienia pogorszenia osiąganej wartości.

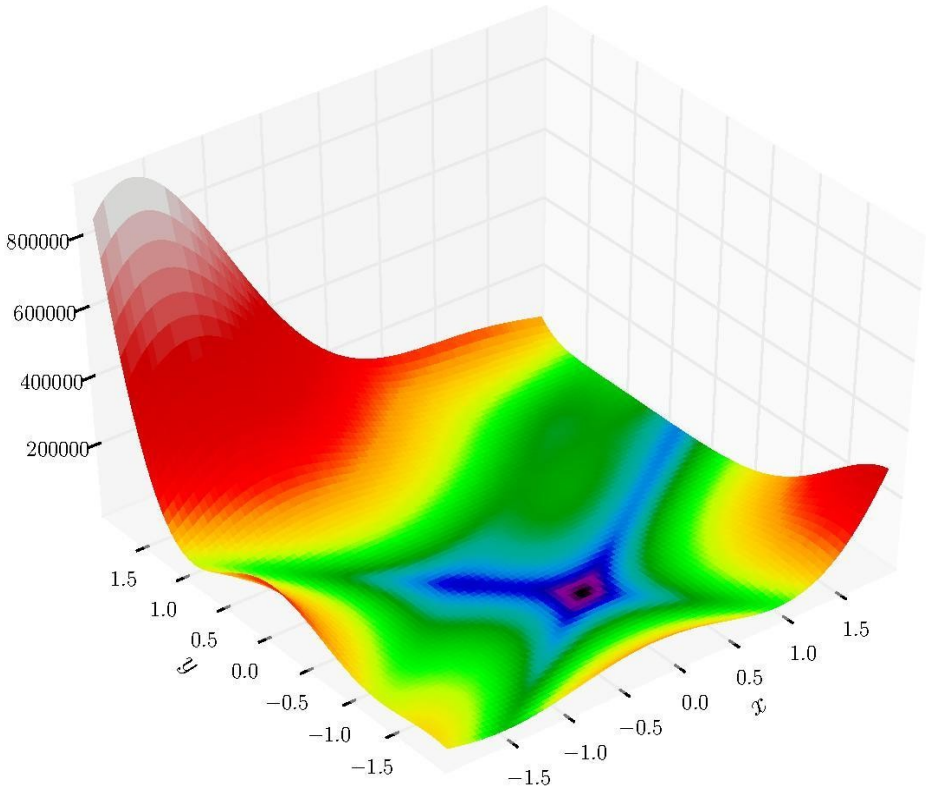
Testowanie
Procedura testowania zakłada powtórzenie tych samych zestawów operacji dla wykonania algorytmu dla funkcji Goldsteina-Price'a oraz funkcji sferycznej.

W przypadku funkcji goldsteina korzystamy a jej określoności w przedziale [-2,2][-2,2] oraz wiedzy że $f_{min[-2,2][-2,2]} = f(0,-1) = 3$, funkcja sferyczna jest posiada nieograniczoną dziedzinę, dlatego algorytm generujący punkty korzysta z rozkładu normalnego.

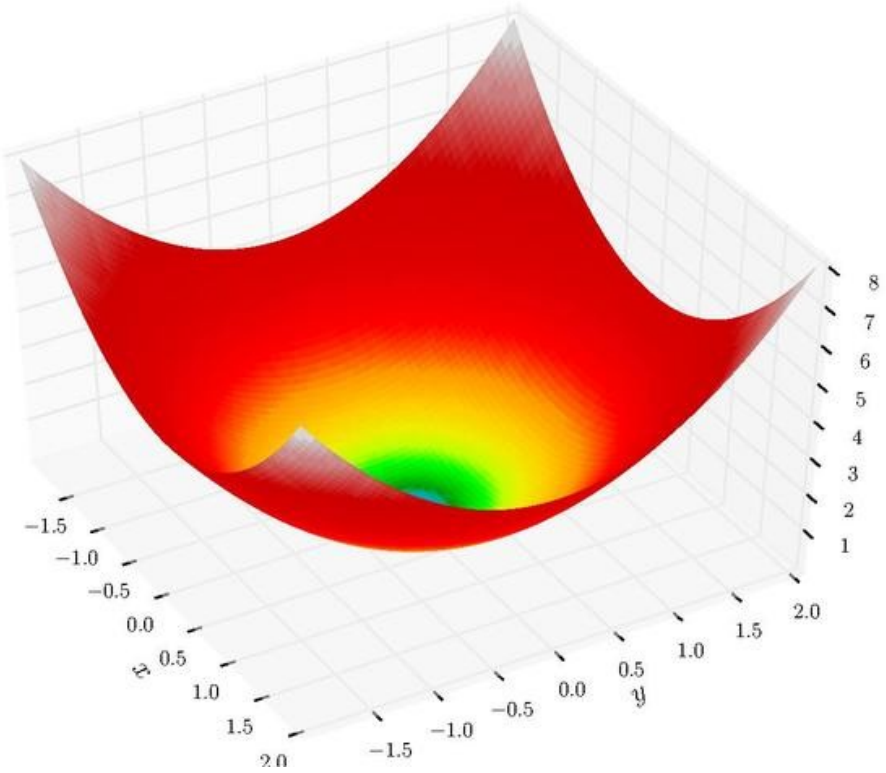
Operacje testowania będą polegały na wykonaniu 10 przebiegów dla każdej z testowanych zależności. Wynikiem testowania będzie wykres zależności wraz z wnioskami.

Przypomnienie ustawień algortymu:
S – wielkość populacji
F – waga wpływu losowej różnicy

CR – stała operacji krzyżowania



1: Wykres funkcji Goldsteinn-Price



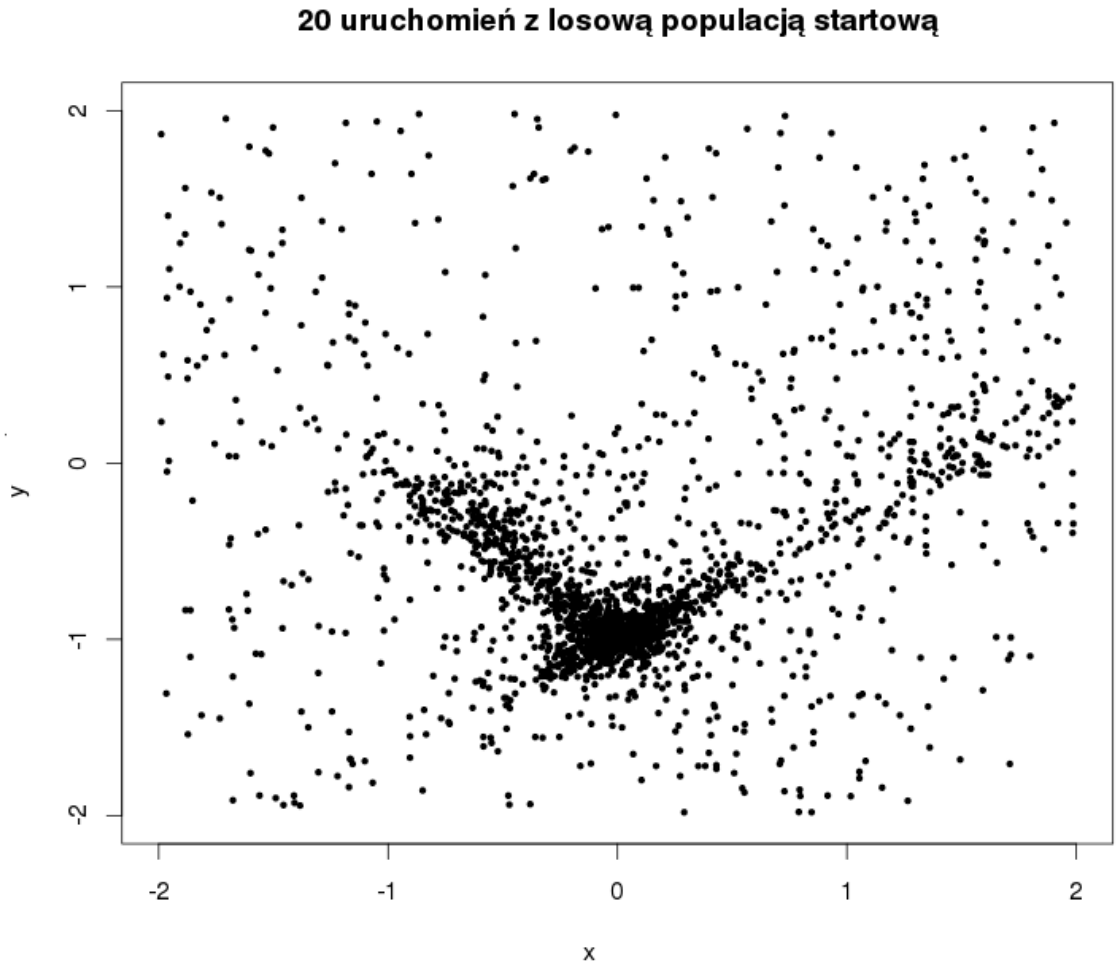
Ilustracja 2: Wykres funkcji sferycznej

Wynik
Wynikiem działania algorytmu jest jakość punktu będącego wartośćią średnią z ostatniej populacji wykonania algorytmu.

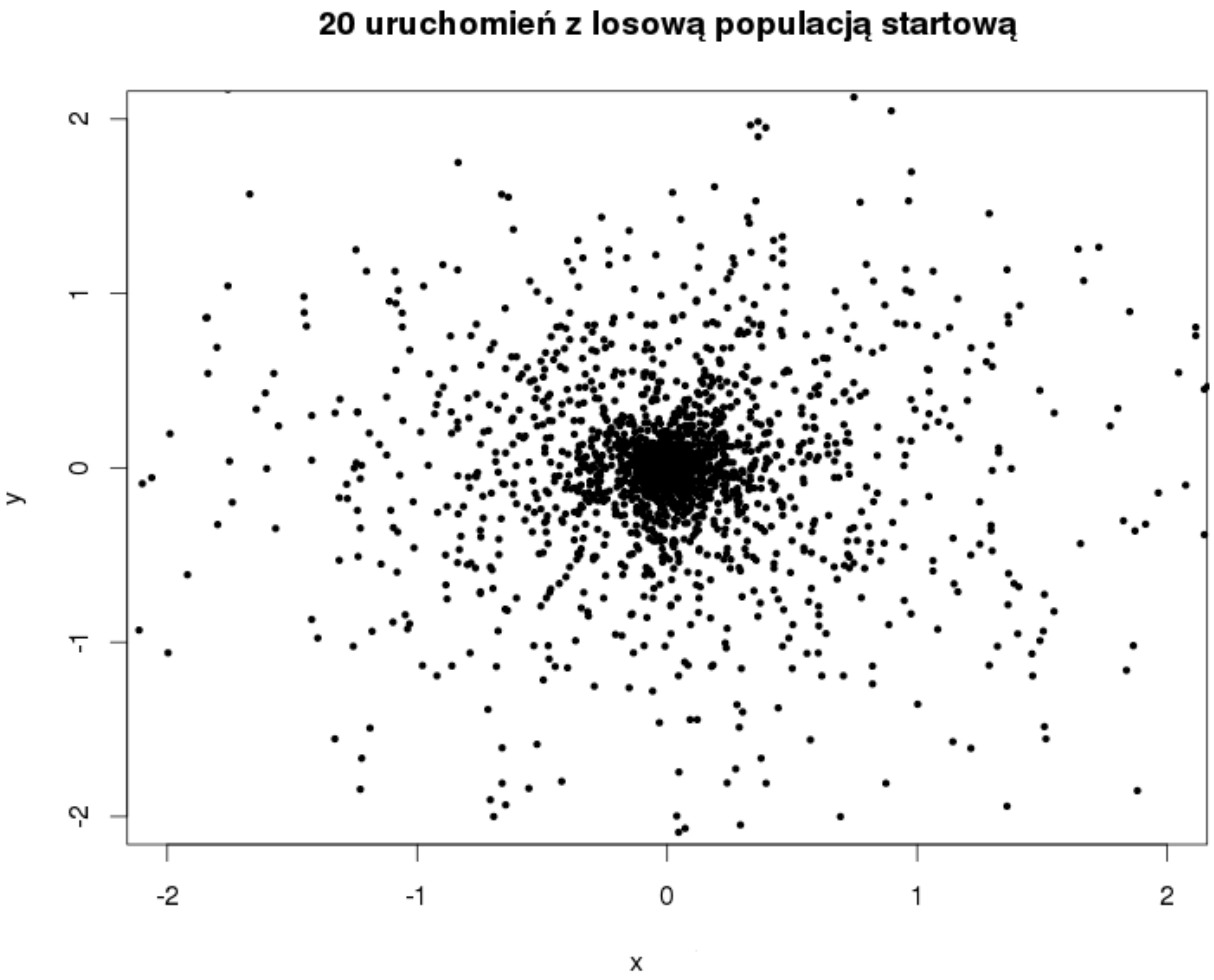
Test #1
test ten będzie polegał na wykonaniu 20 iteracji algorytmu na wcześniej dobranych ustawieniach testowych (S = 15, F = 0.9, CR = 0.5), losując za każdym razem nową populację startową oraz stosując zatrzymanie algorytmu w związku z „stagnacją”.

Funkcja Goldsteina-Price'a

Ilustracja 3: Wykres funkcji Goldsteina-Price'a



Porównując powyższy wykres z wykresem 3D przedstawiającym wartości funkcji, możemy stwierdzić, że algorytm przebywa głównie w 'dołkach' czyli obszarach o mniejszych wartościach funkcji. W punkcie (0,-1) możemy zaobserwować czarną plamę punktów. Jest to związane z tym że potomkowie z populacji oscylują wokół punktu minimum globalnego, ale nie osiągają jego wartości. Jednak dzięki parametrowi F < 1, populacja z każdą iteracją zacieśnia się. Oczywiście w ogólnym przypadku istniała by możliwość, że kilka punktów populacji znajdzie się w jakimś minimum lokalnym i nie będzie w stanie z niego wyjść ale specyfika tej funkcji (a jeszcze bardzie funkcji sferycznej) jak i dobrane parametry nie pozwalają na to.
Funkcja sferyczna



Ilustracja 4: Wykres funkcji sferycznej
Analizując powyższy wykres i porównując do wykresu 3D przedstawiającego wartości funkcji dla pewnego przedziału wartości również możemy zauważyć że algorytm szybko 'przemieszcza' populacje w kierunku małych wartości funkcji.

Algorytm osiągał warunek stopu (brak zmiany jakości w 20 iteracjach o 0.000001) w przypadku funkcji sferycznej średnio po 65 iteracjach, dla funkcji Goldesteina-Price'a średnio po 95 iteracjach przy osiąganiu za każdym razem wyniku w przybliżeniu równym 3.

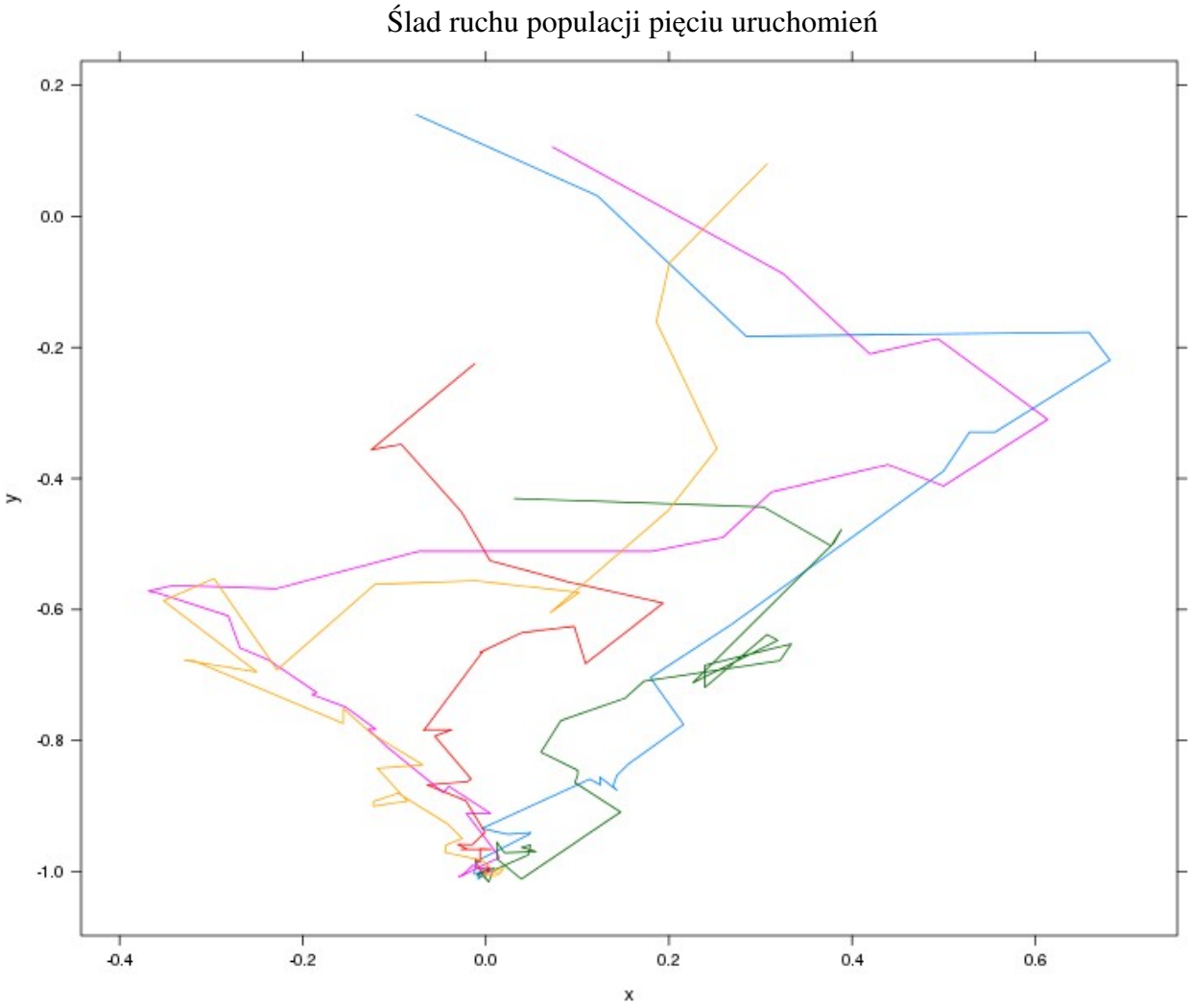
Algorytm ma tendencję do 'przebywania' w pobliżu wartości o największej jakości (tu najmniejszych). Wykazuje ciągłą oscylację punktów populacji wokół pewnego punktu w przypadku osadzenia się populacji w jednym miejscu.

Test 2.

Test ten ma za zadanie pokazać drogę chmury punktów populacji w czasie wykonania algorytmu, oraz jej zależność od parametrów.
Wyznacznikiem położenia chmury punktów jest wartość średnia z punktów należących do populacji.

Dla takich samych ustawień algorytmu, tzn (S = 15, F = 0.9, CR = 0.5) oraz ustawienia warunku zakończenia na ilość iteracji (n = 200) i losowy początek punktu otrzymałem takie wykresy:

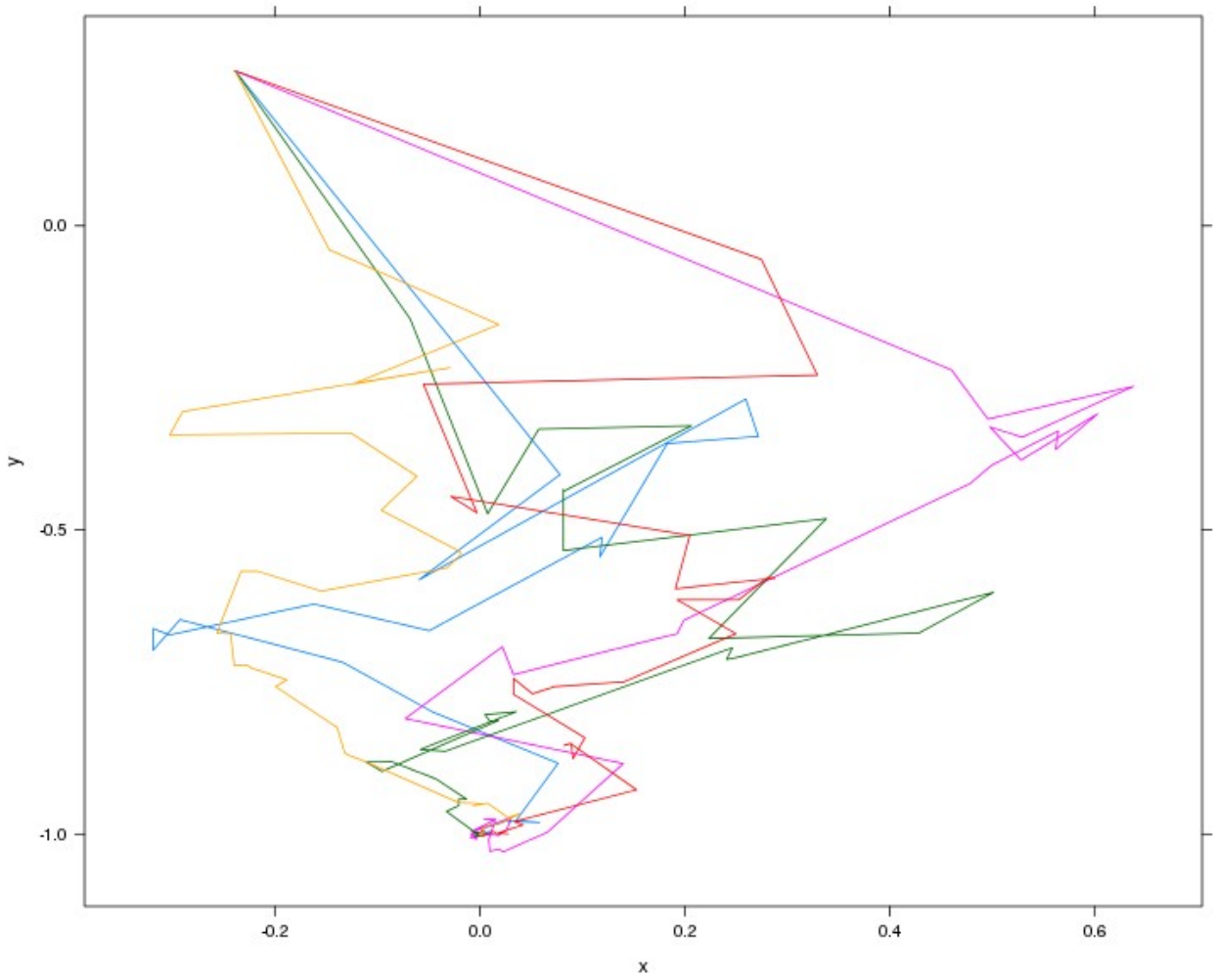
Funkcja Goldsteina-Price'a



Ilustracja 5: Wykres funkcji Goldsteina-Price'a

Na powyższym wykresie widać ze algorytm w podążaniu do celu w miejscach gdzie pochodna ma dużą wartość jest bardziej „zdecydowany”. Początkowy skok jakości populacji jest duży ponieważ pierwsi potomkowie są losowej jakości. Widzimy, też że algorytm potrafi czasem się cofać od rozwiązania (najlepiej widać to na przykładzie „zielonego' uruchomienia). Blisko wartości minimalnej można zaobserwować kłębienie się linii ruchu populacji co świadczy o tym że jest ona ciągle w ruchu.

Ślad ruchu populacji – wspólna populacja startowa – pięć uruchomień

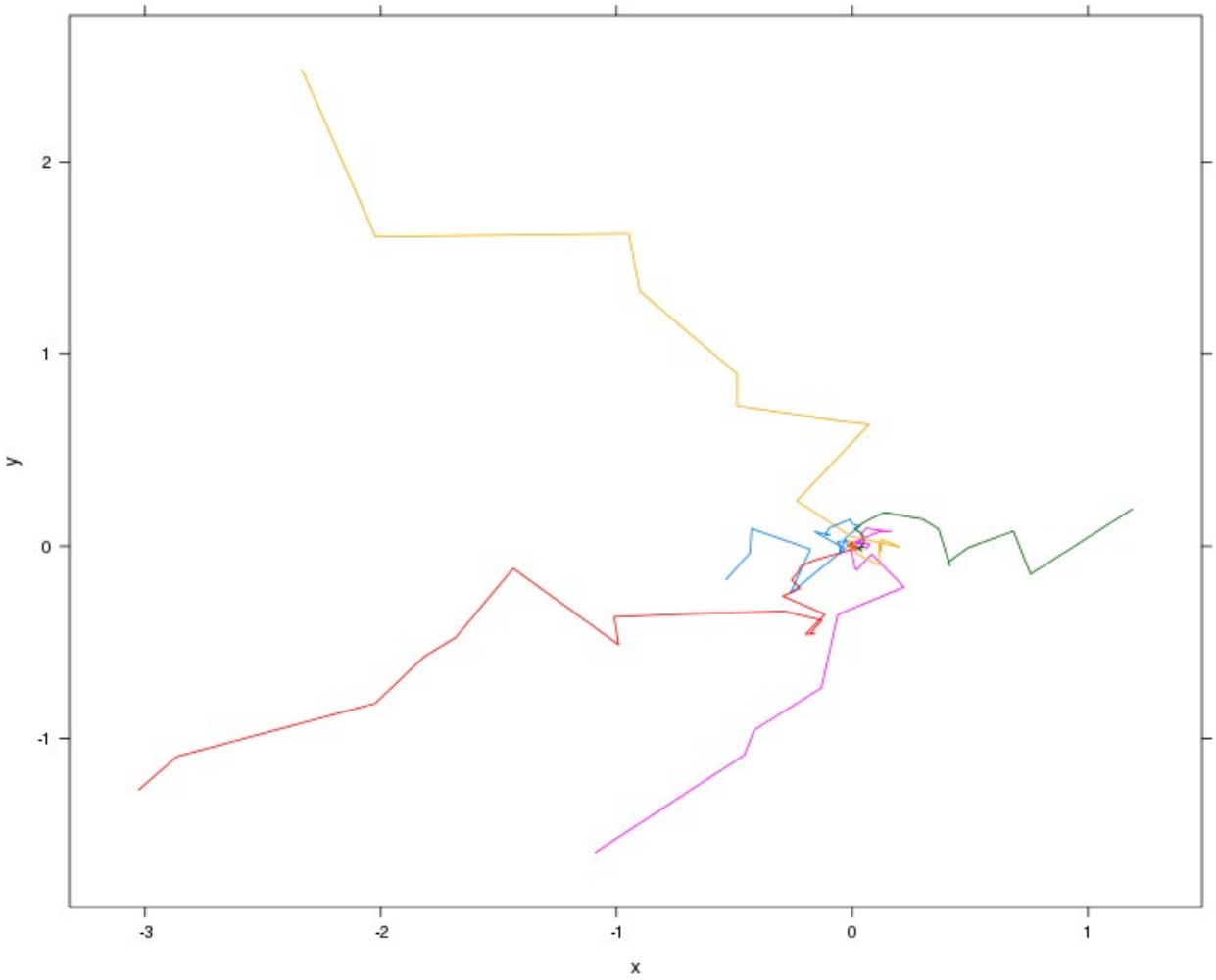


Ilustracja 6: Wykres funkcji Goldsteina-Price'a - wspólna populacja początkowa

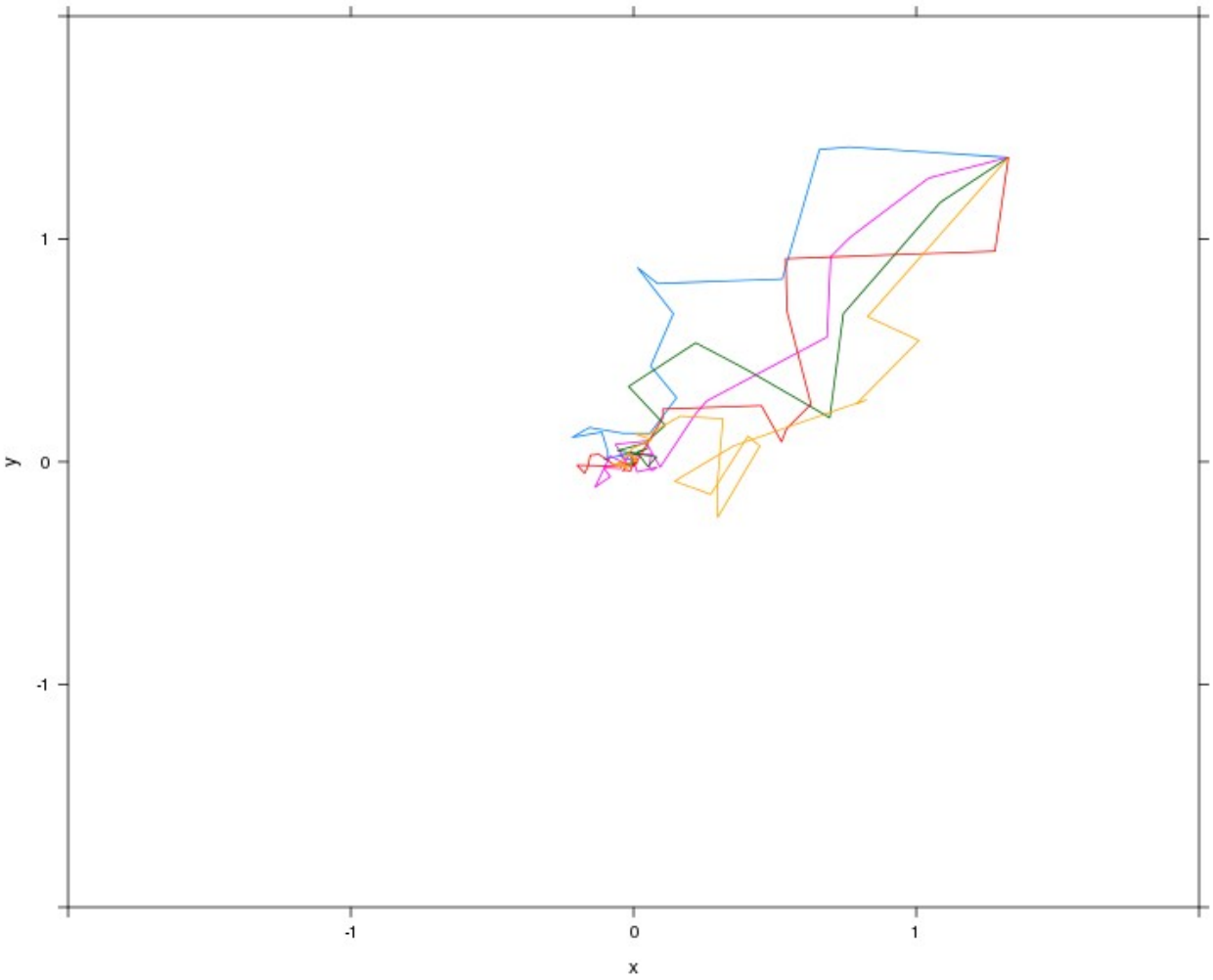
Ten wykres przedstawia rozbieżność algorytmu spowodowaną niedeterminizmem procesu selekcji oraz wariacji. Każde uruchomienie algorytmu było zainicjowane tą samą populacją. W każdym przypadku pierwsza iteracja związana była z dużym skokiem jakościowym.

Funkcja sferyczna

Ślad ruchu populacji pięciu uruchomień



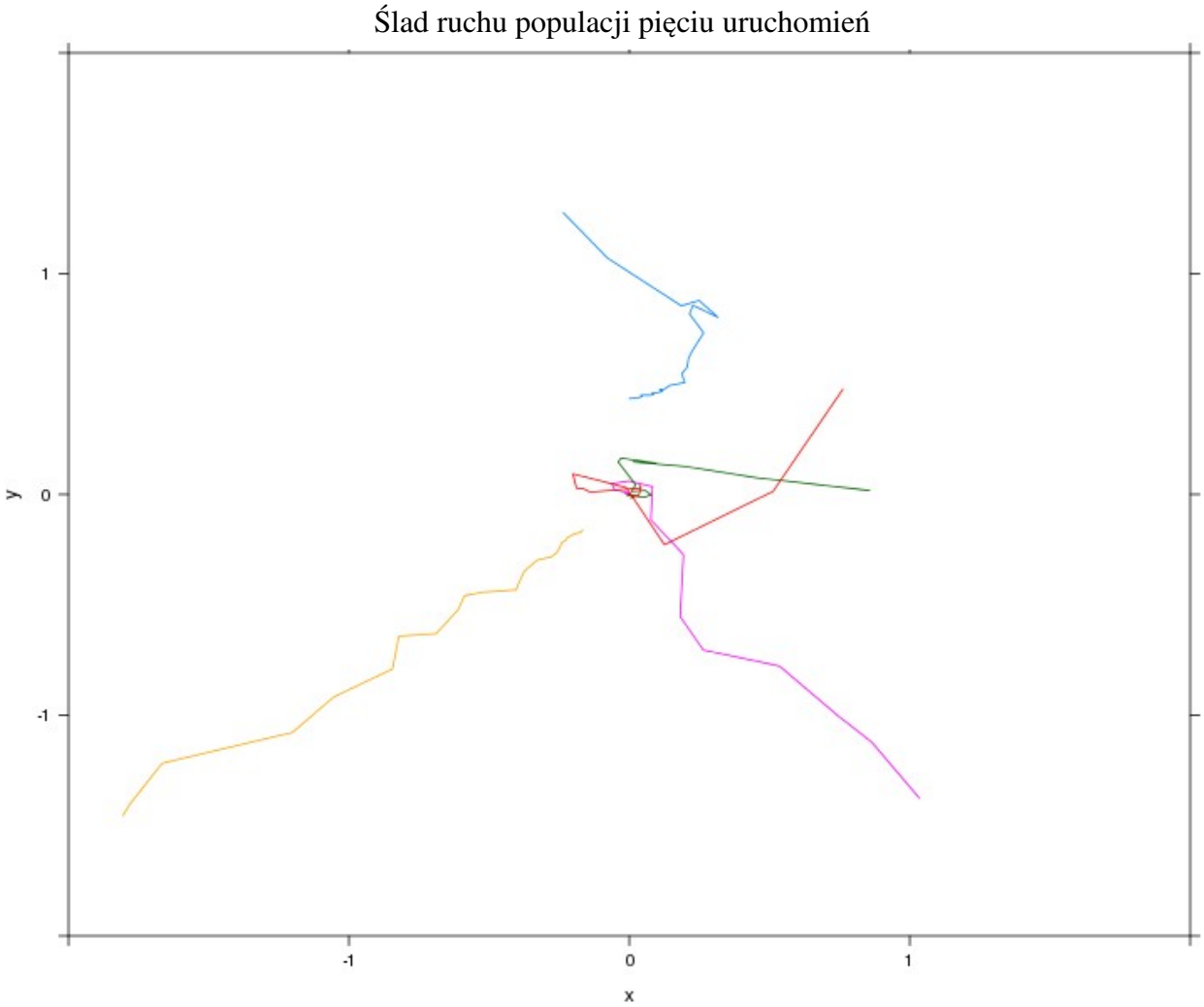
Dla funkcji sferycznej algorytm jest znacznie bardziej „zdecydowany” w podążaniu do celu, im dalej od wartości minimalnej, tym skoki jakościowe poszczególnych iteracji populacji są większe. W pewnej odległości od punktu (0,0) możemy zaobserwować, że algorytm wykazuje większą tendencję do ruchów w innych kierunkach niż kierunek do wartości minimalnej.



Ten wykres przedstawia sytuację uruchomienia algorytmu z tego samego punktu. Widać, że w dużej odległości od punktu minimalnego dla tej funkcji uruchomienia algorytmu zachowują się mniej więcej tak samo. Algorytm ten bardzo szybko przemieszcza się w przypadku dużych różnic wartości na przestrzeni na której rozpościera się jego populacja.

Zmiana wartości współczynnika wpływu losowej różnicy F (F = 0.2)

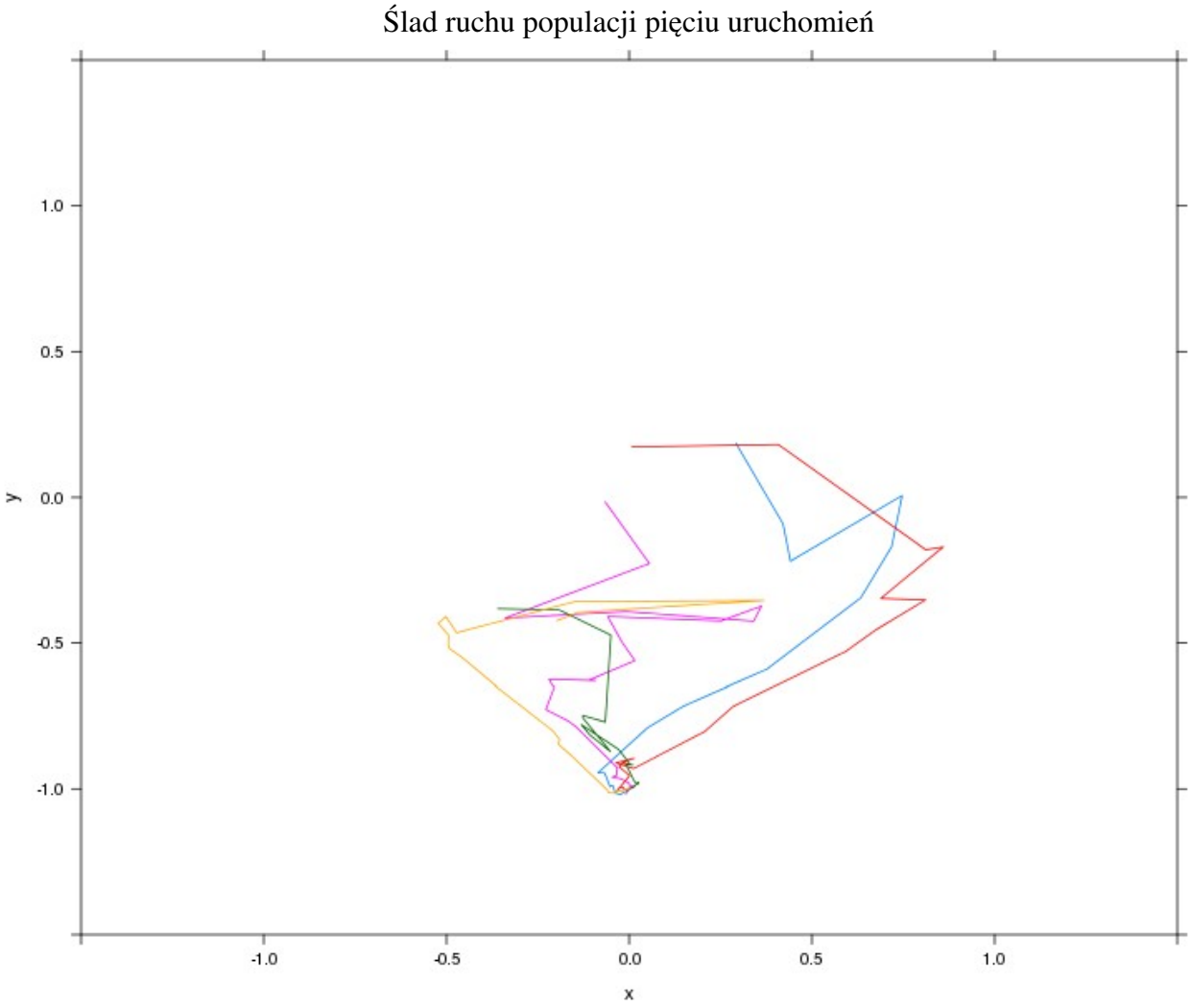
Funkcja sferyczna



Ilustracja 7: Wykres funkcji sferycznej - wartość F = 0.2

Zmniejszenie wpływy wektora różnicowego na mutację osobnika wybranego w procesie mutacji sprawiło, że algorytm porusza się znacznie wolniej, co przy ustalonych 200 iteracjach testu powoduje, że nie jest w stanie dojść do szukanego punktu, choć ma możliwość jego odnalezienia. Zmiana ta wpłynęła również na mniejsze odstępstwa w podążaniu do celu od kierunku tego celu. Dopiero w bardzo blisko punktu minimalnego algorytm wykazuje większe rozbieżności w kierunku.

Funkcja Goldsteina-Price'a

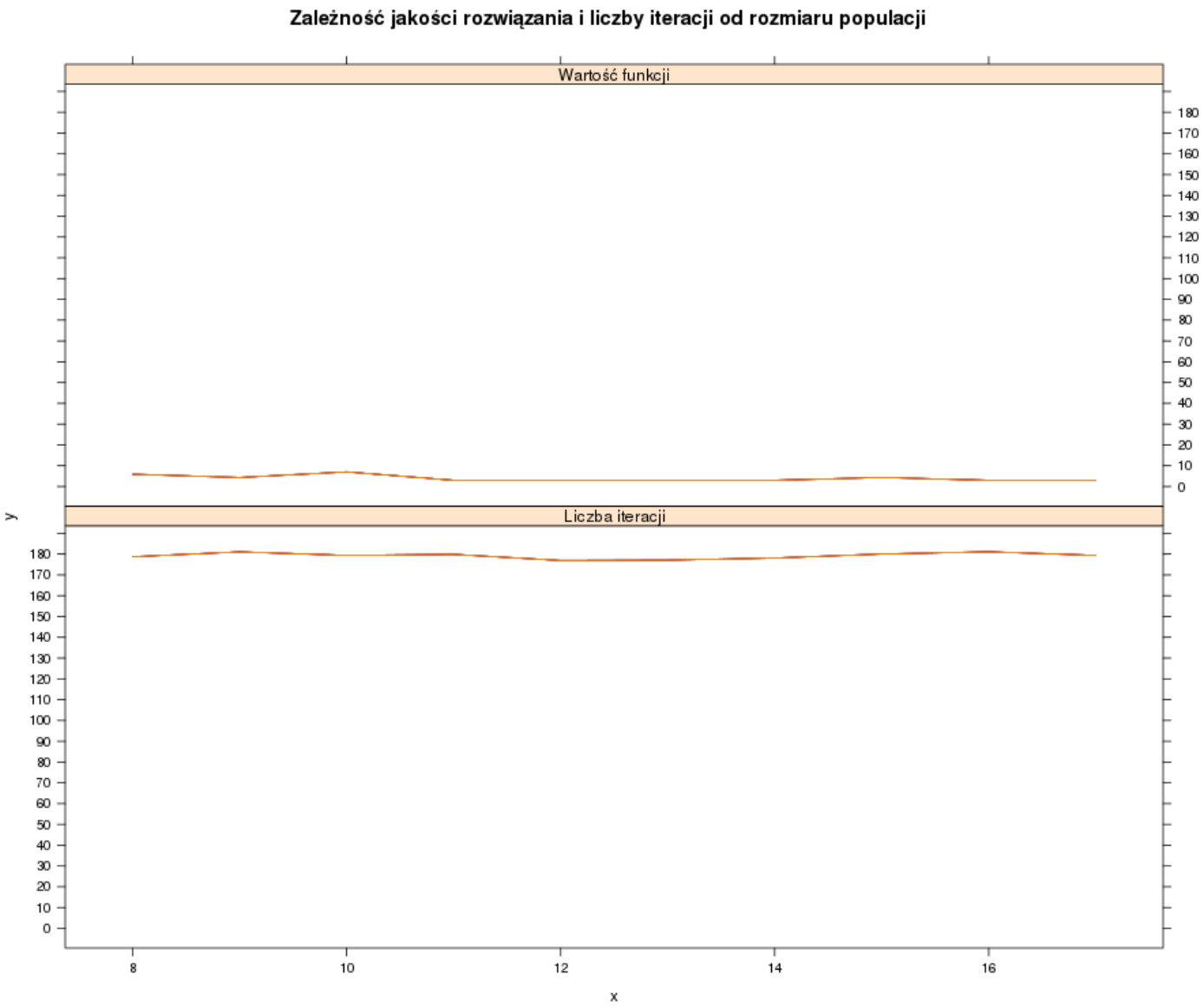


Ilustracja 8: Wykres funkcji Goldsteina-Price'a - wartość F = 0.2

W przypadku funkcji Goldsteina-Price'a nie można zaobserwować tak łatwo ukierunkowania się algorytmu w podążaniu do celu. Sądzę, że po zmniejszeniu wartości F z 0.9 na 0.2 algorytm jest bardziej czuły na zmiany wartości funkcji, zwiększyłą się jego cecha eksploatacyjna a zmniejszyła eksploracyjna. Z tego powodu też po wejściu do „dolin" uruchomienia żółty niebieski i czerwony posuwają się nimi nie dokonując żadnych prób wyjścia z nich.

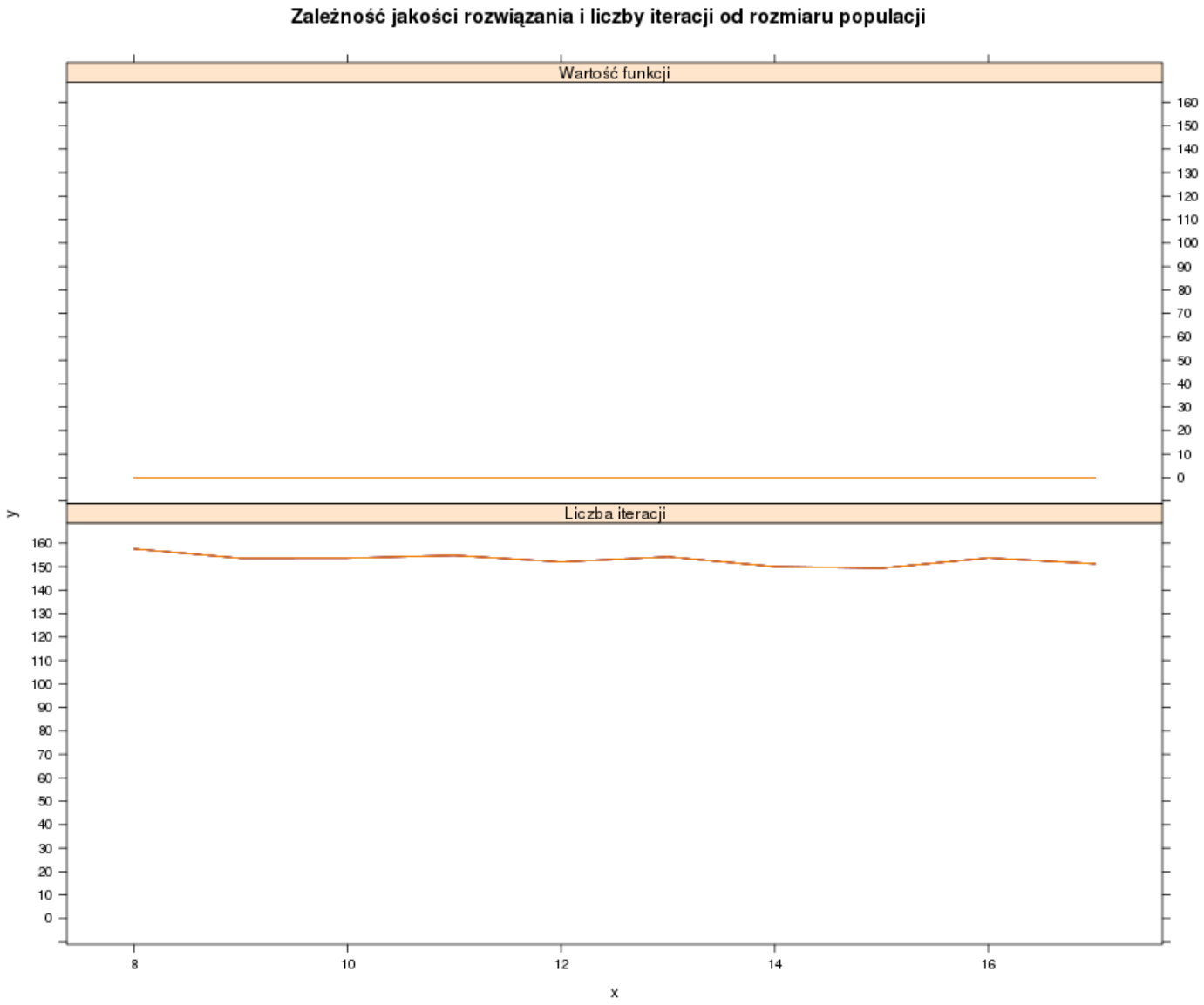
Zmniejszenie wartości współczynnika F ma wpływa na spowolnienie algorytmu, zwiększa jeko eksploatatywność kosztem eksploracyjności, algorytm w większym stopniu bierze pod uwagę zmianę wartości funkcji na przestrzeni na której znajduje się populacja.

Test 3
Zależność szybkości osiąganego wyniku i jego jakości od zmiany parametrów. Ten test będzie polegał na wyznaczeniu wpływu zmiany poszczególnych parametrów na jakość wyniku oraz ilość iteracji algorytmu. Warunkiem stopu będzie stagnacja jakości najgorszego elementu populacji. Wynik jakości to wartość funkcji w punkcie będącym średnią punktów populacji. Ilość iteracji w których wymagana jest poprawa o wartość 0.00001 to 20 iteracji.
Sprawdzenie każdego ustawienia parametru będzie związane z wykonaniem 20 iteracji algorymtu z danymi ustawieniami oraz z losową populacją początkową.



Wykres funkcji Goldsteina-Price'a

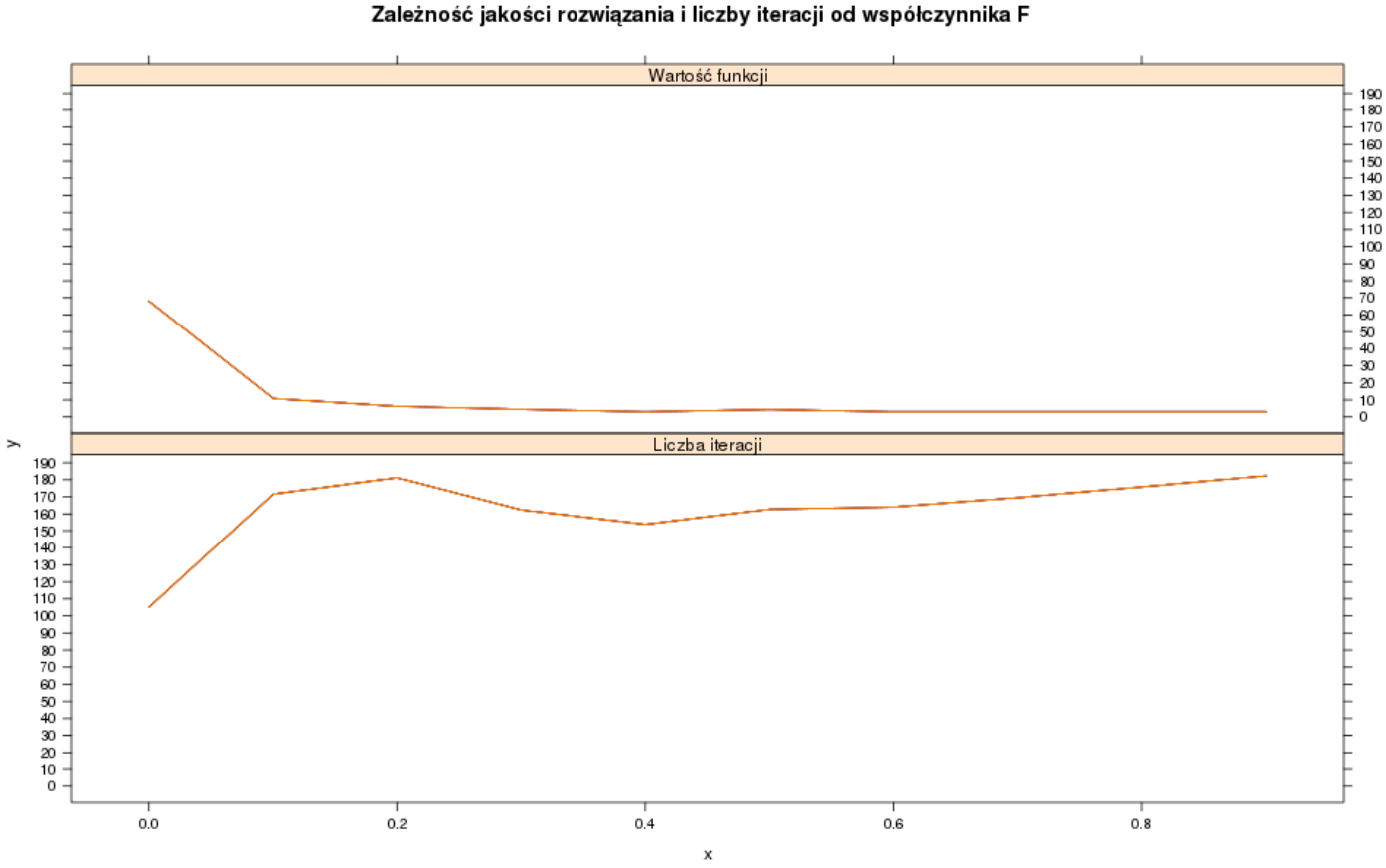
Wykres ten pokazuje zależność jakości rozwiązania i liczby iteracji od wielkości populacji. Jakość rozwiązania dla mało licznych populacji jest dość mała, gdyż powoduje to że średnia jest liczona z małej ilości członków populacji co wpływa na jej dokładność. Liczba iteracji utrzymuje się na stałym poziomie. Można zauważyć, że dobrym ustawieniem liczby populacji jest zakres od 11 do 14. Populacja nie jest za duża i dość szybko się liczy a wynik jest bardzo dobry przy jednocześnie względnie niskiej ilości iteracji (choć różnice zapewne można by tą pominąć).



Wykres funkcji sferycznej

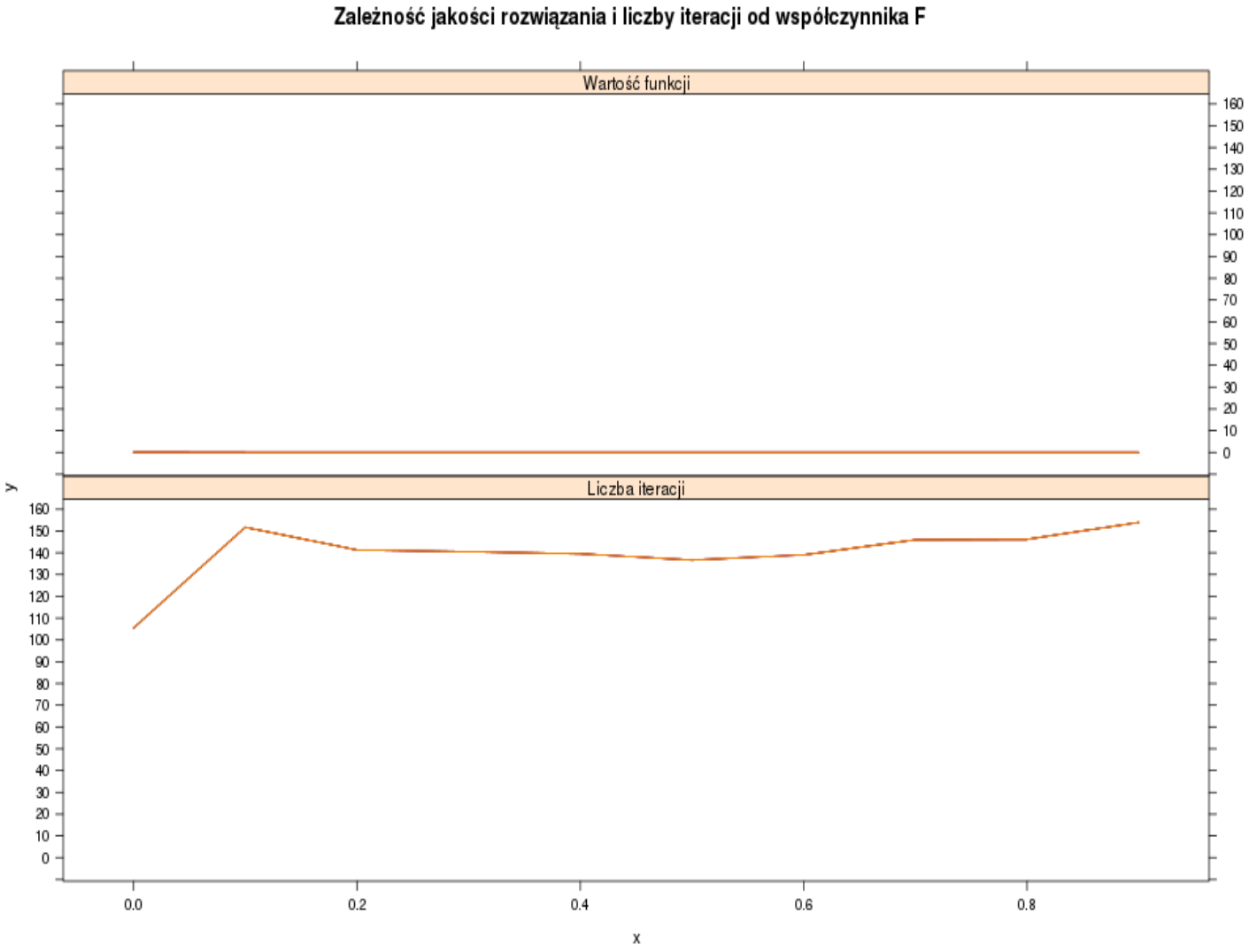
Dla tej funkcji rozmiar populacji nie ma wpływu na otrzymaną jakość rozwiązania, co jest dość oczywiste ze względu na jej specyfikę (algorytm zawsze będzie podążał w kierunku rozwiązania). Najmniejsza ilość iteracji występuje dla populacji o wielkości 14,15 elementowej.

Badanie wpływu współczynnika F (wpływ losowej różnicy). Parametr S = 15, CR = 0.5



Wykres funkcji Goldsteina-Price'a

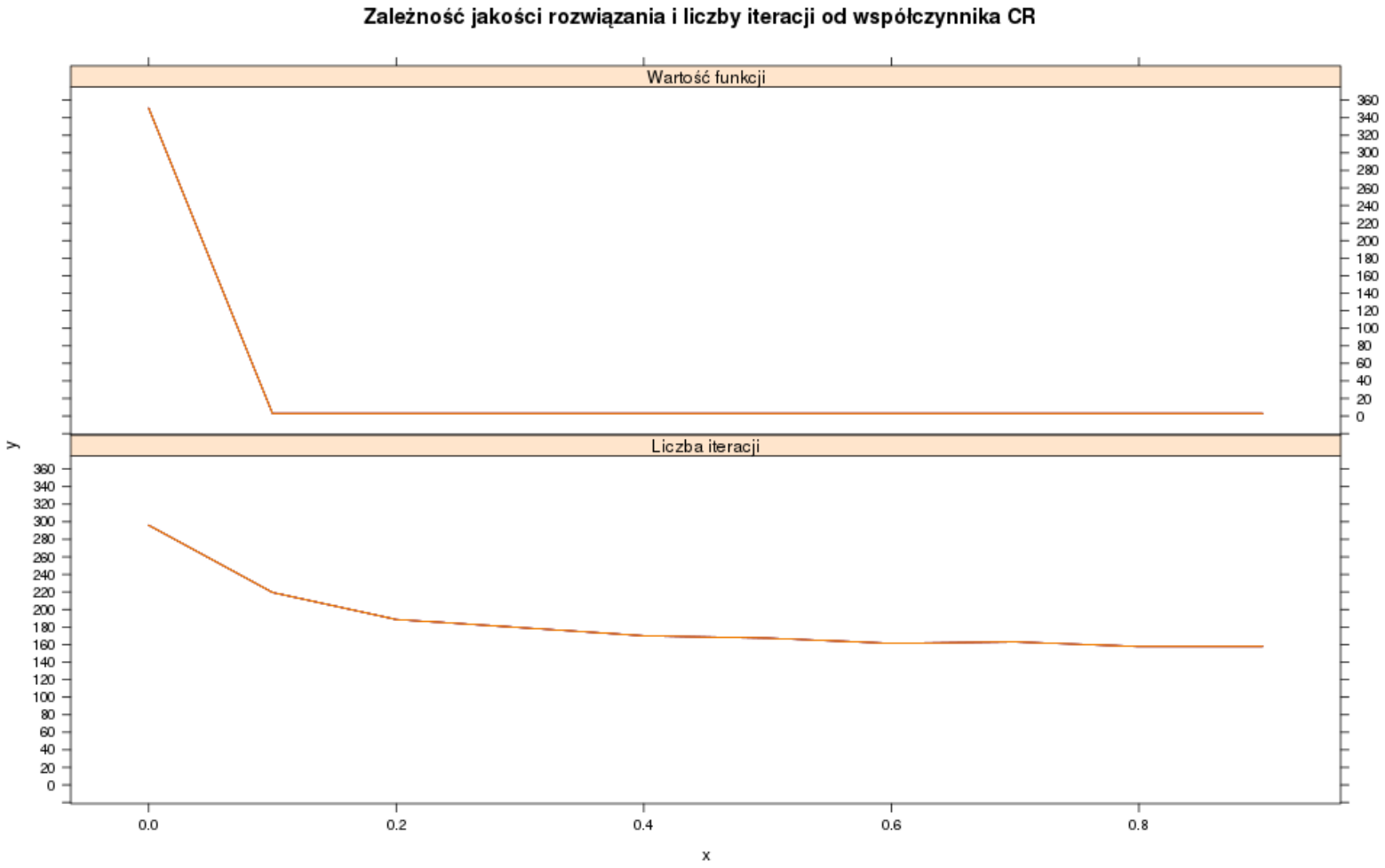
Parametr F = 0 powoduje, że algorytm nie dokonuje mutacji wektorem różnicowym, a tylko krzyżuje dwa osobniki potomstwa. Z wykresu możemy wyczytać że dla podanej funkcji warto ustalać współczynnik F nie mniejszy niż 0.6, gdyż wtedy nie zauważamy pogorszenia wyników, jednak ustalanie zbyt dużej wartości parametru F powoduje wzrost liczby iteracji prowadzących do otrzymania zadowalającego wyniku. Niska wartość parametru F (od 0 do około 0.2) powoduje niską jakość wyniku. Ciekawym efektem jest wysoka ilość iteracji dla wartości parametru f od 0,1 do około 0.25



Wykres funkcji sferycznej

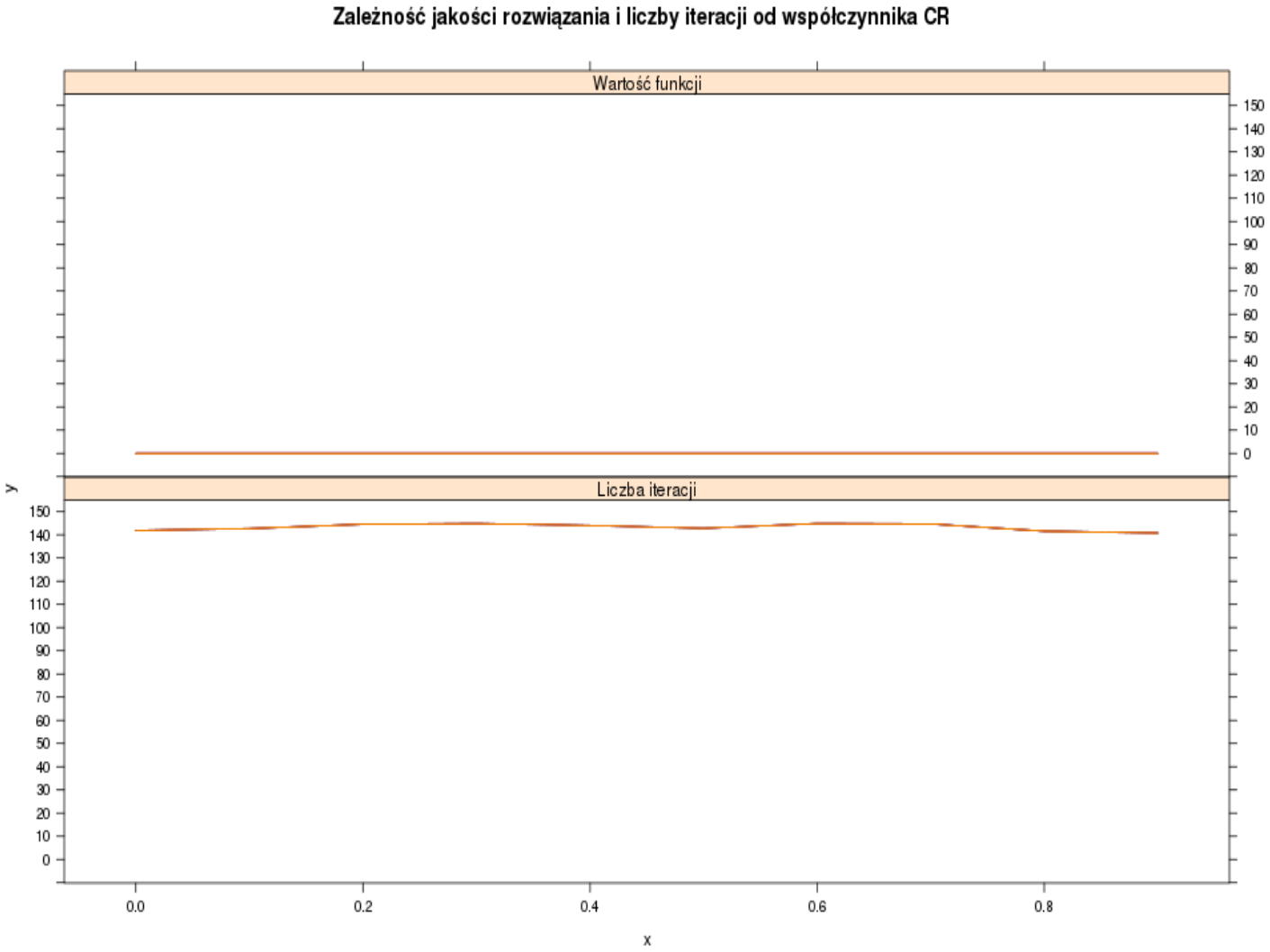
Dla tej funkcji zaobserwowałem taką samą zależność ilości iteracji od współczynnika F. Współczynnik ten nie ma wpływu na jakość rozwiązania przy ustawieniu innych ustawień S = 15, CR = 0.5. Oznacza to że do jej optymalizacji można by również użyć zwykłego algorytmu ewolucyjnego (równoważność F = 0)

Badanie wpływu współczynnika krzyżowania CR (parametr S = 15, F = 0.7)



Wykres Goldsteina-Price'a

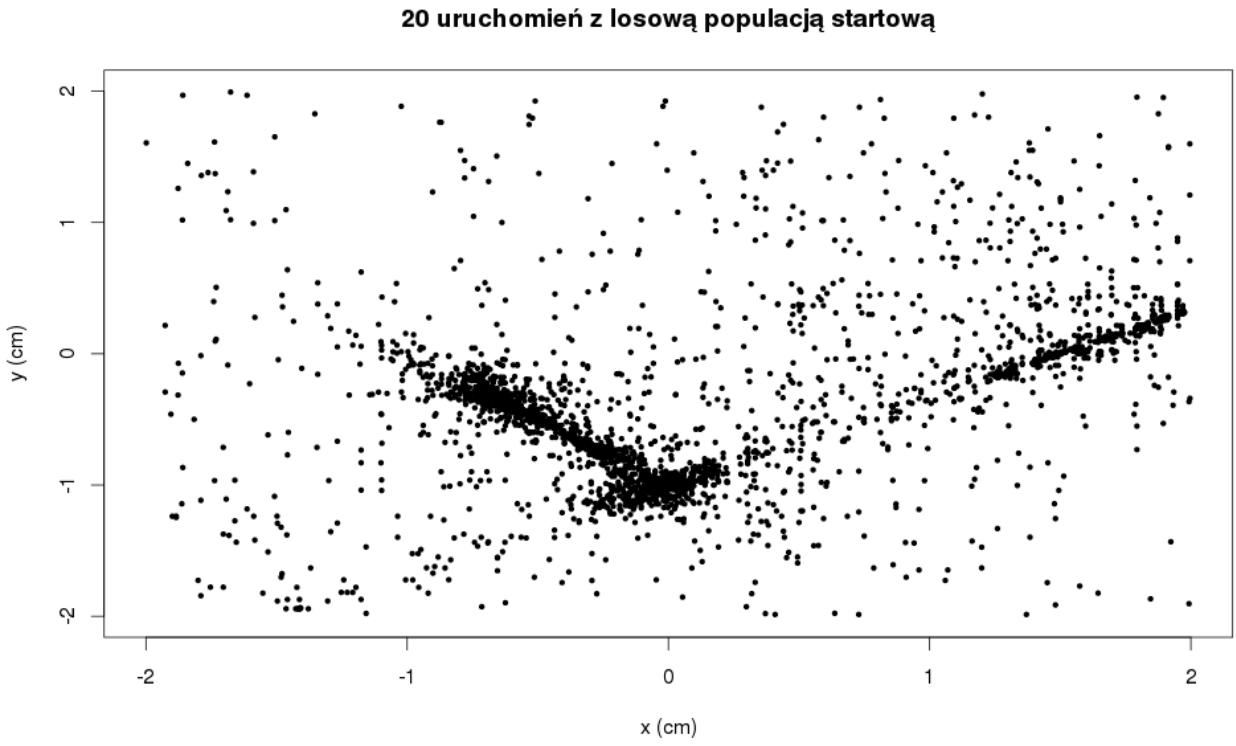
Z wykresu możemy wyczytać że ustawienie zbyt niskiej wartości CR jest nieoptymalne, powoduje dużą ilość iteracji oraz niską jakość rozwiązania. W przypadku gdy parametr CR wynosi 0 zawsze jeden element potomka pochodzi z mutanta. Wtedy też potomek będzie się zawsze znajdował albo oddalony zgodnie z osią x albo y od rodzica.



Wykres funkcji sferycznej

Wykres ten pokazuje, że funkcja sferyczna jest bardzo niewymagająca dla algorytmu ewolucji różnicowej, jakość jest stała równa w przybliżeniu maksymalnej a ilość iteracji utrzymuje się mniej więcej stałym poziomie, choć dla większych wartości parametru CR można zauważyć jej mały spadek. Ilość iteracji się zmniejszyła w stosunku do pierwszych prób testu 3 ze względu na ustawienie parametru F zgodnie z wynikami symulacji, czyli na wartość 0.7

Ciekawość zachęciła mnie do wykonania wykresu rozkładu generowanych punktów dla ustawień: S = 15, F= 0.7, CR = 0.0, wynikiem tego jest:

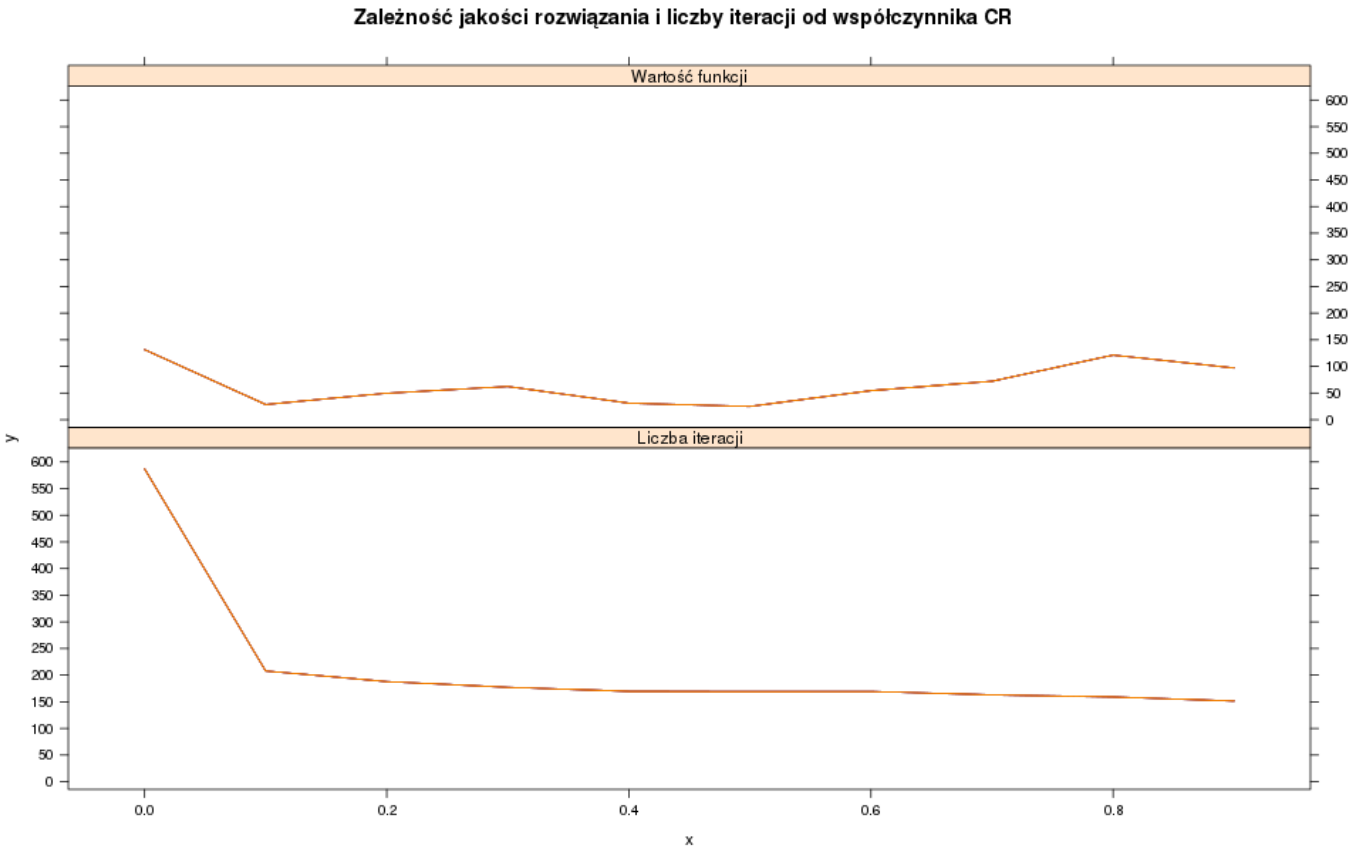


Ilustracja 9: Wykres funkcji Goldsteina-Price'a

Przy takich parametrach ustawień algorytm będąc w dolinach nie wykazuje chęci przemieszczania się w kierunku wartości minimalnej. Nie jest on w stanie wykonywać efektywnych przesunięć punktów populacji „w skos", co ogranicza jego możliwości przesunięcia się będąc ograniczonym w ruchach poprzez warunek stopu jakim jest stagnacja wyniku rozumiana jako brak poprawy w ciągu n iteracji. Wykonanie testu z ustaloną ilością iteracji na 2000 dla uruchomienia z tymi samymi parametrami dała te same rezultaty. Nie należy zatem nigdy ustawiać parametru CR na bardzo niską wartość.

Podsumowując testowanie algorytmu ewolucji różnicowej dla funkcji sferycznej i Goldsteina-Price'a możemy stwierdzić, że optymalnymi ustawieniami dla funkcji Goldsteina-Proce'a są parametry S = {14,15} F = <0.6;0.7> CR = 0.5

Aby dobrać wartość parametru CR musiałem wykonać jeszcze jeden test:



Wykres funkcji Goldsteina-Price'a

Test ten został wykonany dla ustawień S = 8 i F = 0.1 czyli dla takich, które powodują gorsze wyniki na wyjściu. Pozwoliły one lepiej ocenić jakość w zależność od współczynnika CR. Najlepsza jakość została osiągnięta dla wartości parametru CR 0.1 oraz 0.5, jednak ze względu na większą ilość iteracji dla wartości 0.1 oraz znaczne pogorszenie wyników przy mniejszych wartościach proponowałbym ustawienie wartości Cr na 0.5

Dobrane parametry są dobre również dla funkcji sferycznej.

Program testujący

Na potrzeby testowania algorytmu ewolucji różnicowej został utworzony program w języku R składający się z plików:
- ewolucja.R
- metody_funkcji_goldesteina.R
- metody_funkcji_sferycznej.R
- testy.R
- warunki_stopu.R

Właściwym programem implementującym algorytm są pliki: „ewolucja.R”, „metody_funkcji_goldesteina.R”, „metody_funkcji_sferycznej.R”, „warunki_stopu.R”. Plik „testy.R” zawiera uruchamiane testy oraz kod rysujący wykresy funkcji. Aby użyć w algorytmu dla konkretnej funkcji należy załadować po załadowaniu pliku „ewolucja.R” i „warunki_stopu.R” odpowiedni plik „metody(...).R”. Aby dodać inną funkcję do testowania należy utworzyć własny plik „metody(...).R” i utworzyć w nim funkcje o tych samych nazwach i typach parametrów co funkcje zawarte w plikach „metody(...).R”.

2. Algorytm wspinaczkowy

Implementacja algorytmu

```
H ← init(s 0)
x ← selBest(H)
while !stop
        Y ← N (x)
        y ← selBest(Y )
        if q(y) > q(x)
                x ← y
        H ← H ∪ Y
```

Ogólna zasada działania:

Inicjacja

Pierwszym krokiem algorytmu jest zainicjowanie algorytmu punktem początkowym. Dalsze kroki algorytmu wykonywane są iteracyjnie
Generacja jest kolejnym krokiem algorytmu i polega na wybraniu wszystkich sąsiadów rozpatrywanego punktu. Sąsiedzi wybierani są z otoczenia określonego przez promień sąsiedztwa – jest on parametrem.

Selekcja

Kolejnym krokiem algorytmu jest wybranie najlepszego(w sensie funkcji celu), z wygenerowanych w poprzednim kroku sąsiadów punktu roboczego.
Jeżeli najlepszy sąsiad okaże się być lepszym od punktu roboczego, to on sam staje się punktem roboczym i rozpoczynana jest kolejna iteracja algorytmu

Zatrzymanie algorytmu

Algorytm zakończy działanie w momencie przekroczenia maksymalnej ilości iteracji lub w momencie gdy selekcja zwróci punkt roboczy, co oznaczać będzie, że wśród sąsiedztwa punktu roboczego, nie istnieje punkt lepszy niż on sam.

Parametry algorytmu

Parametrem wywołania algorytmu będzie promień sąsiedztwa, oraz maksymalna ilość iteracji algorytmu.

Testowanie algorytmu

Testowanie algorytmu wspinaczkowego zostało przeprowadzone na 2 sposoby.
Pierwszy z nich miał na celu zademonstrowanie zachłannego charakteru algorytmu.

Zostało to zrealizowane poprzez wielokrotne uruchomienie algorytmu wspinaczkowego dla zestawu różnych promieni sąsiedztwa.

Promień sąsiedztwa	Ilosc uruchomień
0.02	100
0.04	100
0.05	100
0.1	100
0.2	50
0.4	20
0.5	20

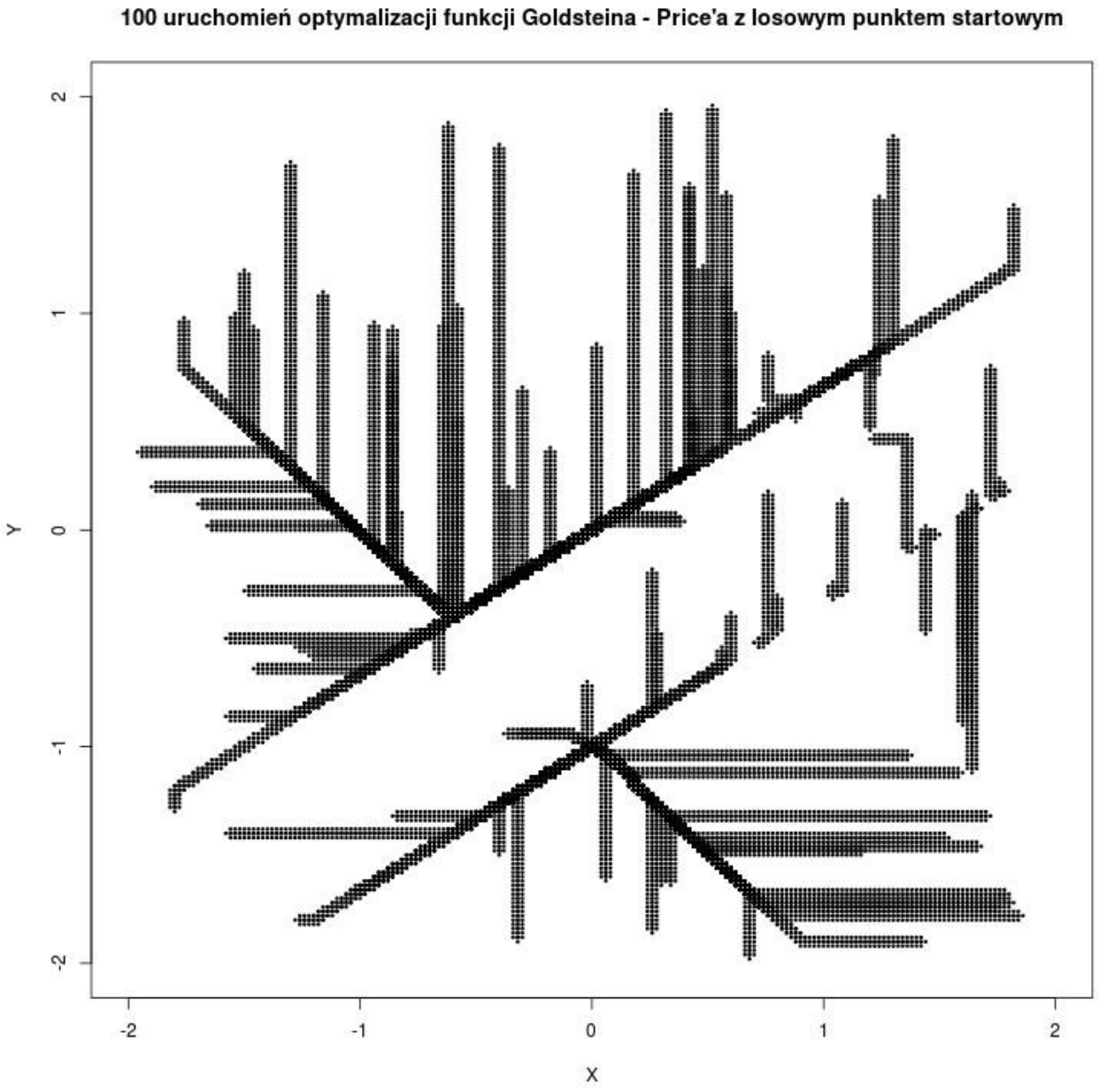
Podczas każdego uruchomienia dla zadanego promienia, punkty, które zostały zapisane do okna historii, zostały naniesione na wykres. Następnie wszystkie wykresy w obrębie danego promienia zostały nałożone na siebie, w celu pokazania miejsc, w których algorytm generował najwięcej punktów. Są to oczywiście miejsca, w których znajdują się maksima lokalne funkcji, mogące w szczególności być maksimami globalnymi.

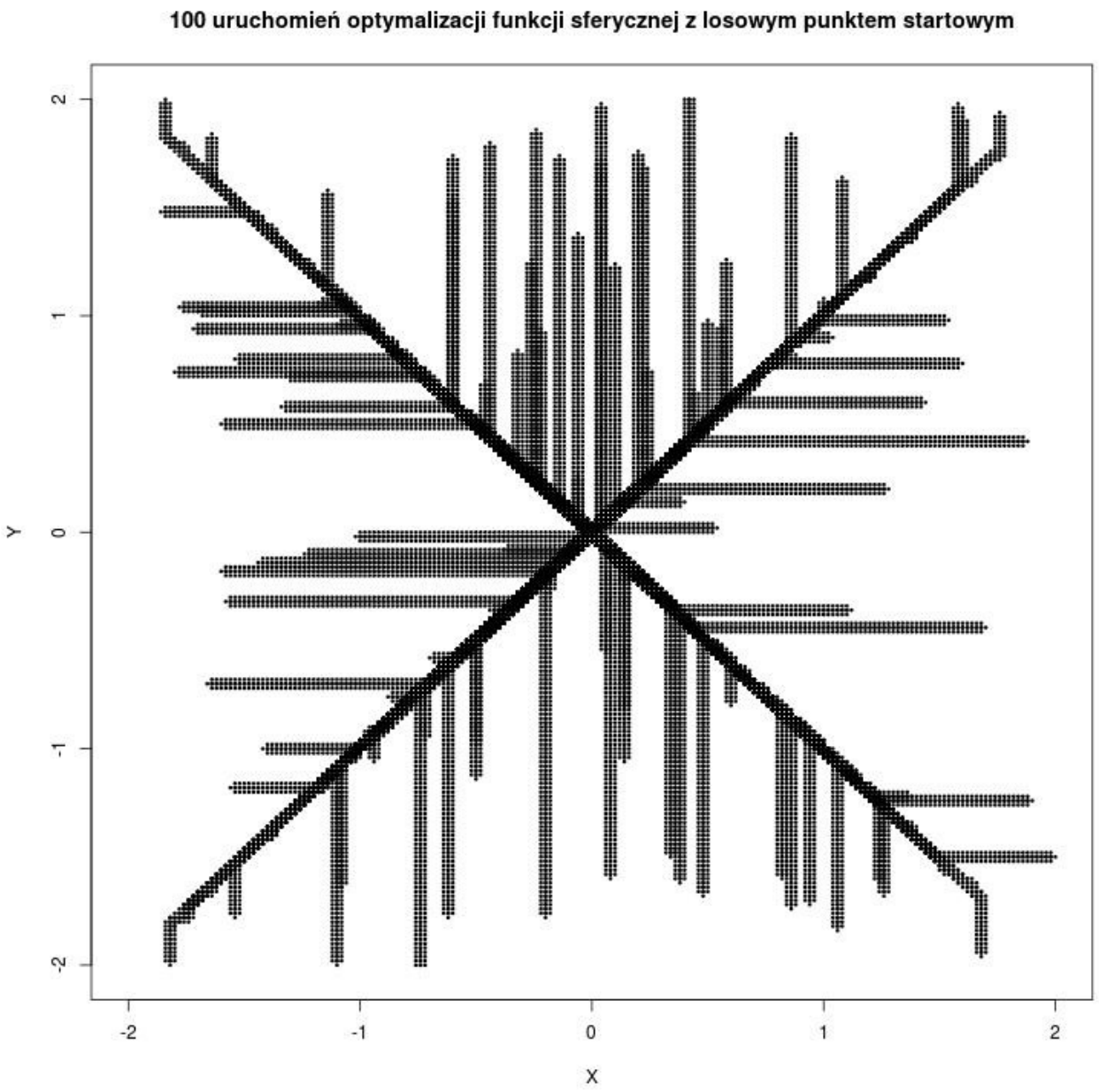
Dodatkowo, wartość parametru maksymalnej ilość iteracji algorytmu wspinaczkowego została ustalona na tak dużą, aby każde uruchomienie algorytmu zakończyło się “naturalnie”, tzn. Poprzez brak lepszego punktu wśród sąsiadów punktu roboczego, zamiast przez przekroczenie liczby iteracji algorytmu.

Rezultaty zostały zaprezentowane na poniższych wykresach. Ilości uruchomień algorytmu zostały dobrane metodą “prób i błędów” tak, aby zbyt duża ilość punktów nie przysłoniła wykresu.

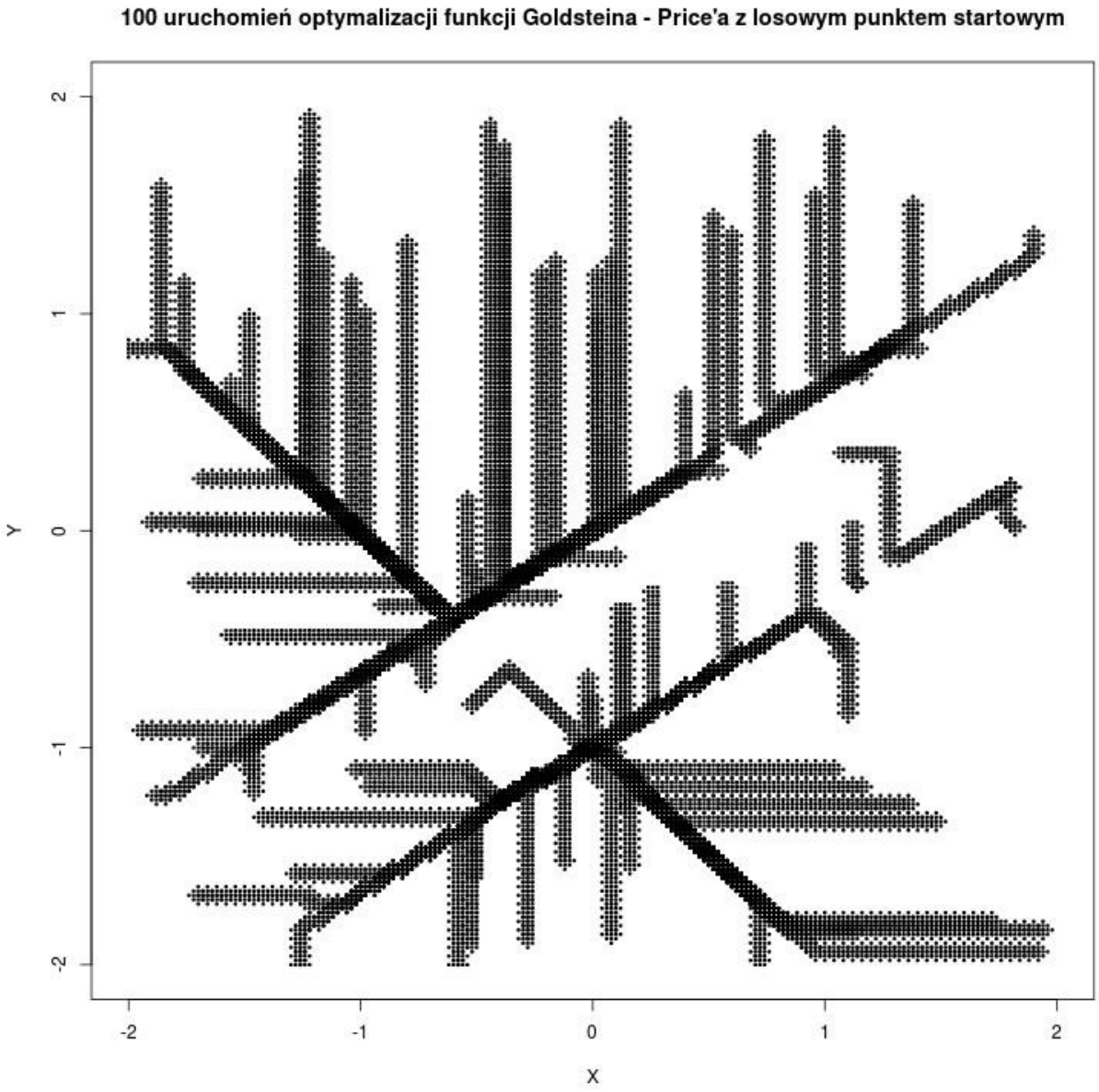
Na wykresach można zaobserwować charakterystyczne koła. Jest to spowodowane wybraniem przez algorytm sąsiadów punktu roboczego, w celu znalezienia potencjalnie lepszego punktu.
Im większy promień sąsiedztwa, tym “koła” stają się większe.

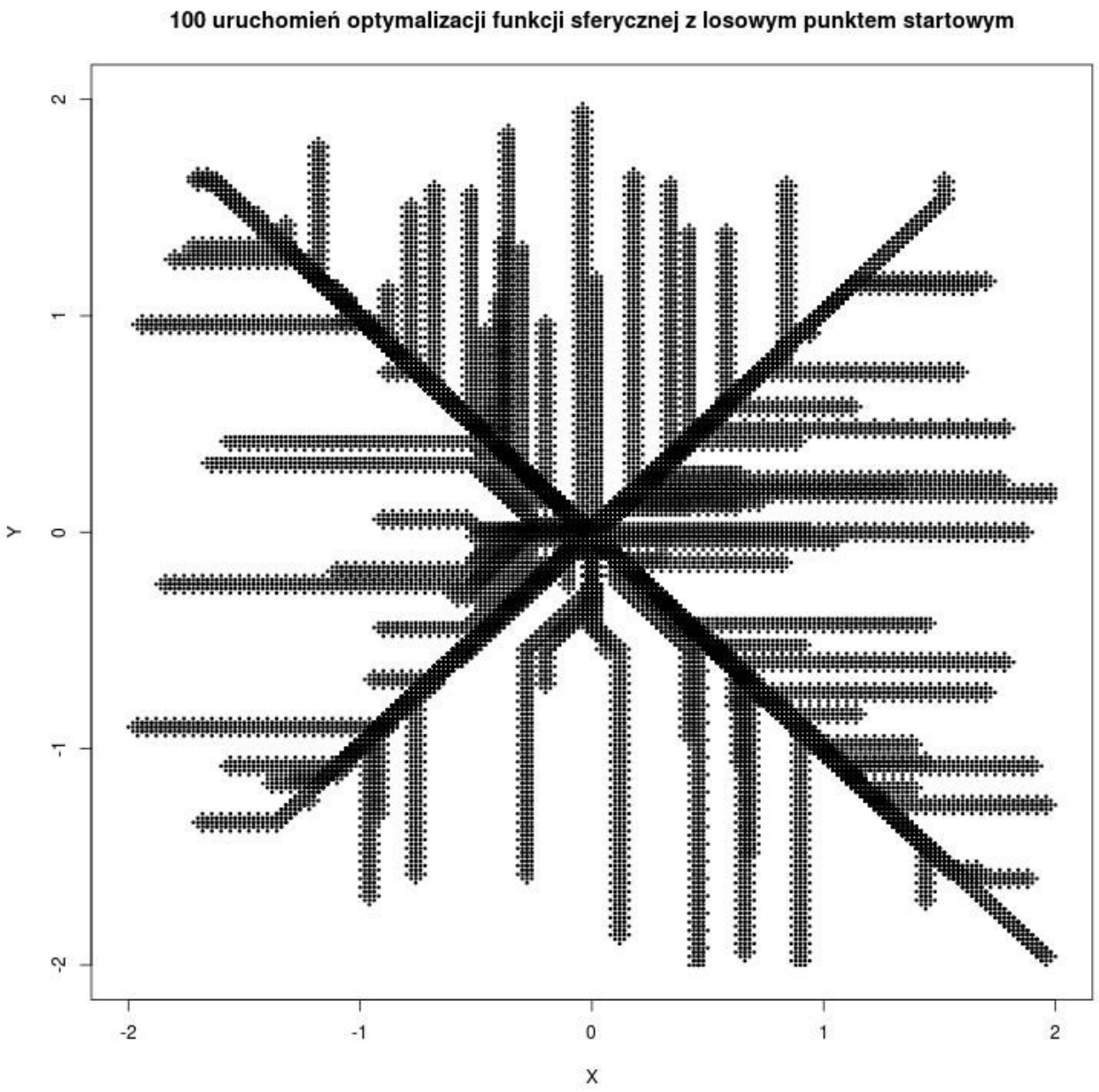
- promień sąsiedztwa 0.02



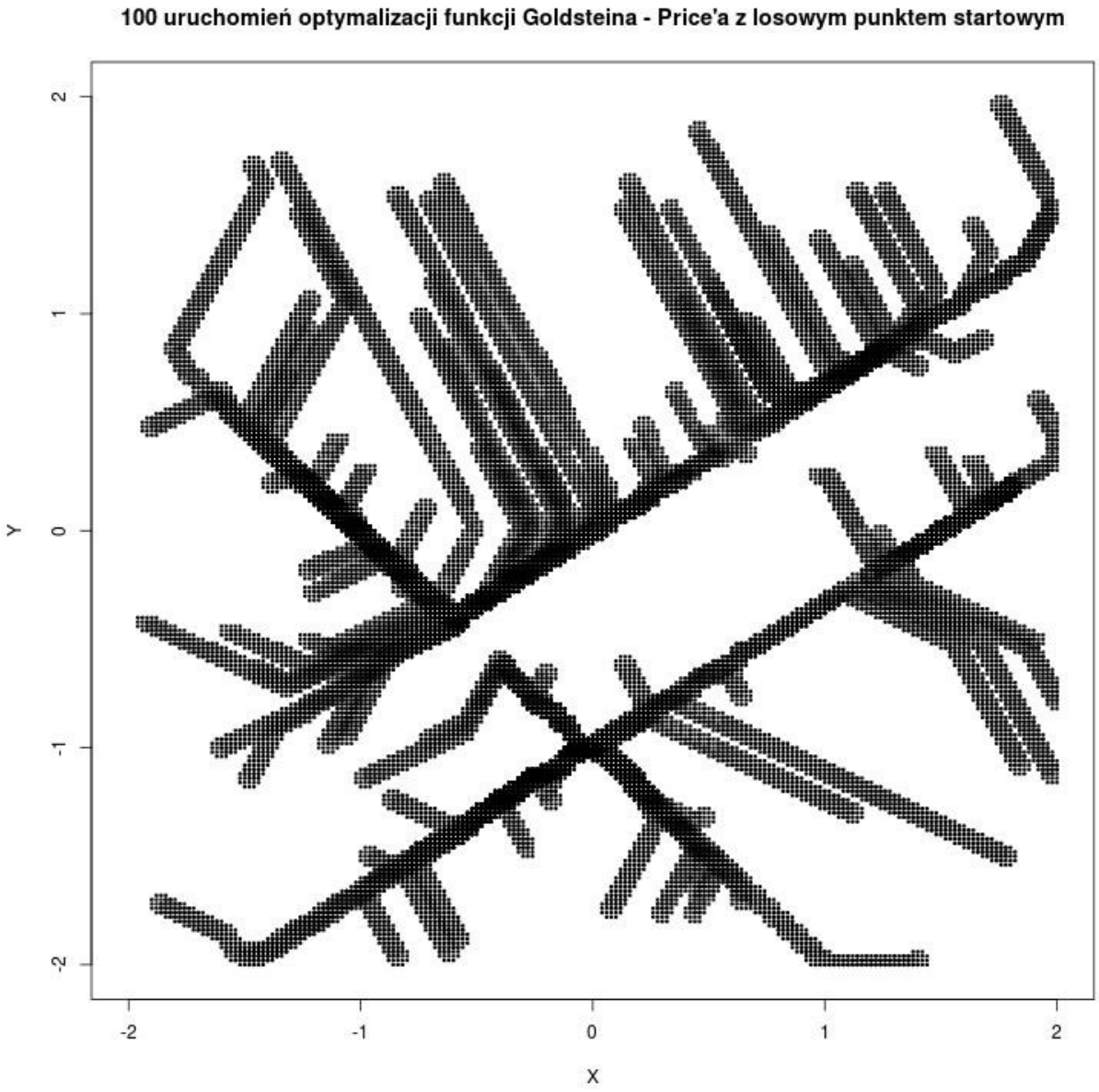

100 uruchomień optymalizacji funkcji sferycznej z losowym punktem startowym
Y
X
2
1
0
-1
-2
-2
-1
0
1
2

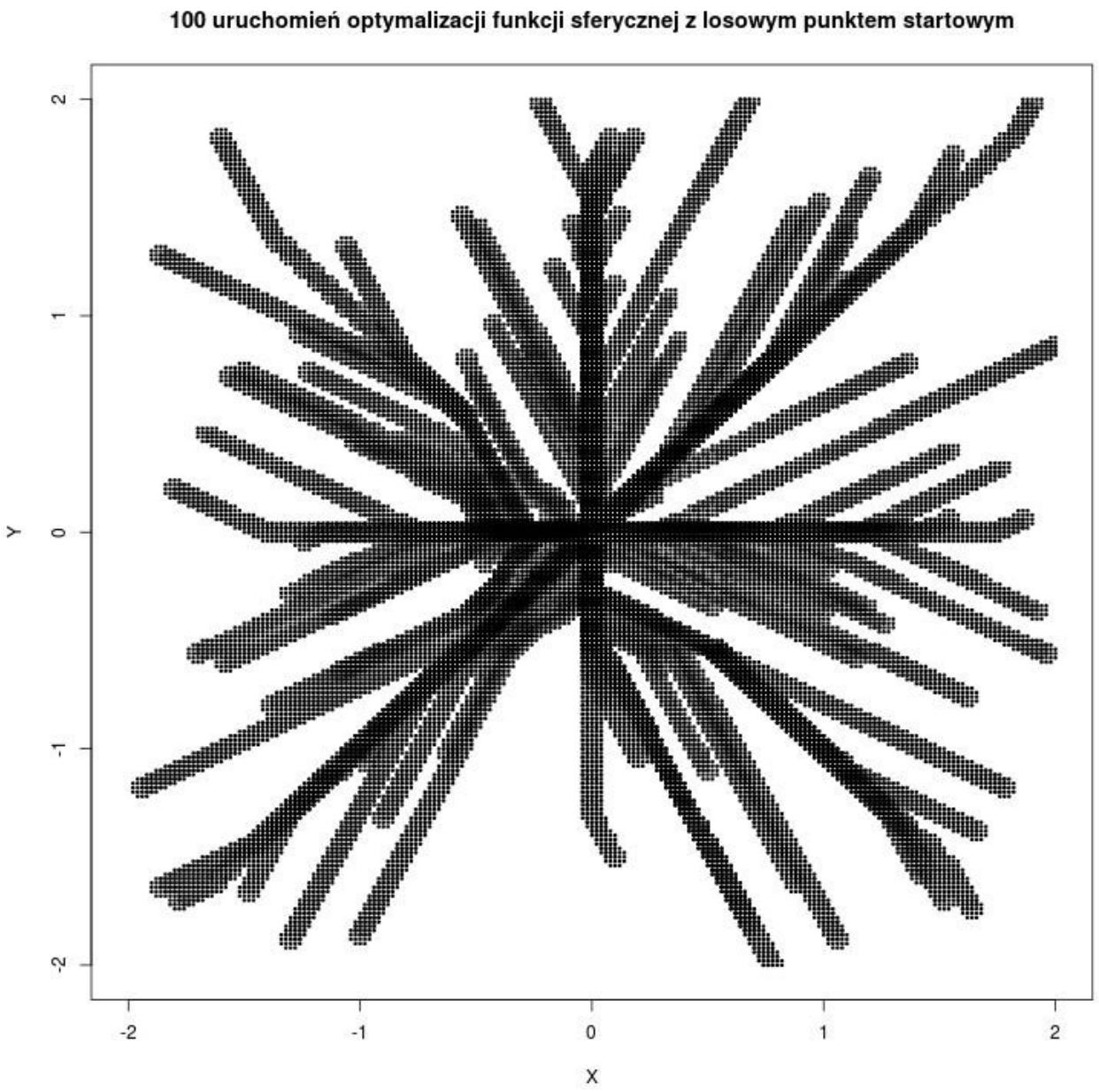
- promień sąsiedztwa 0.04



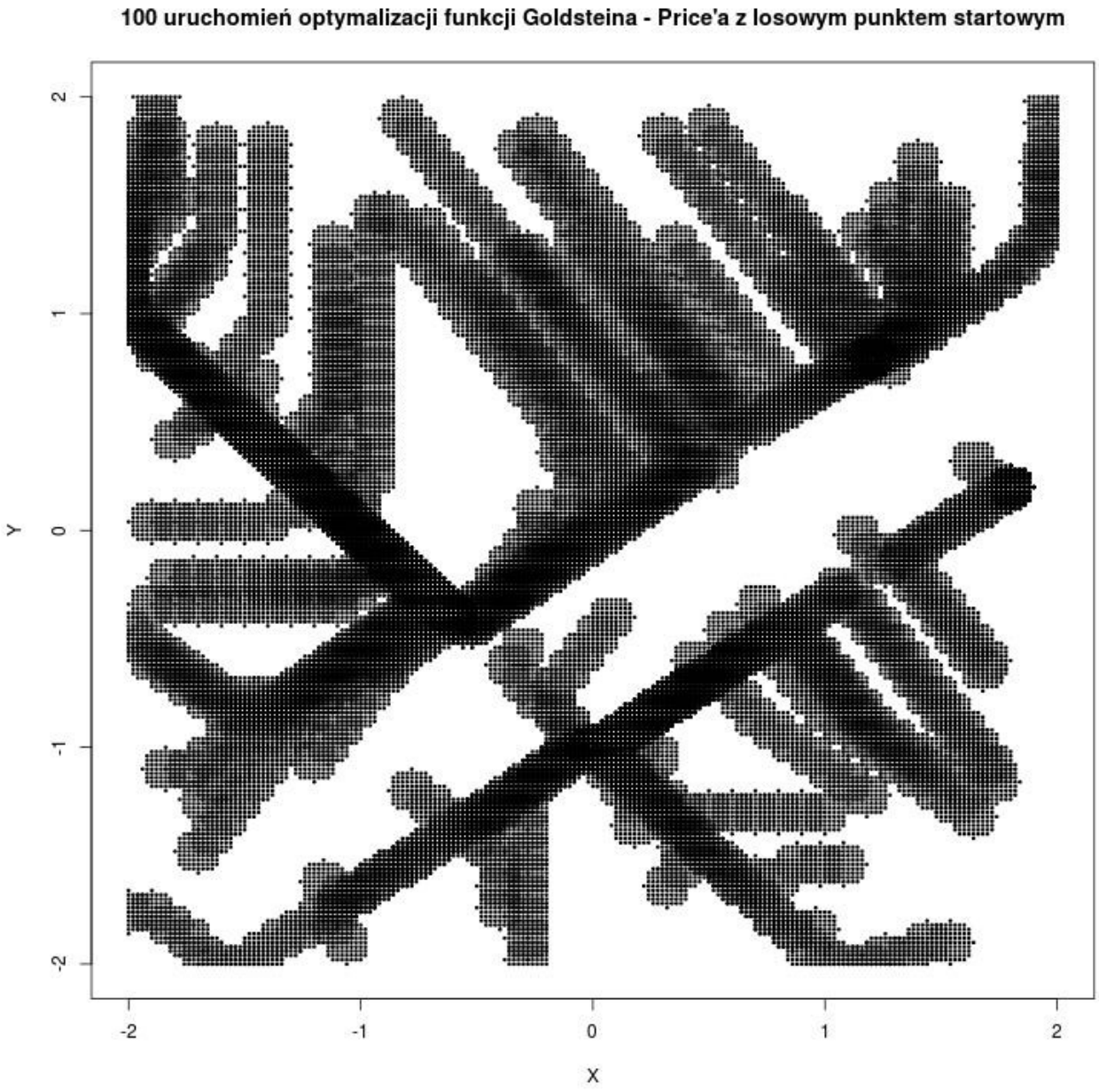

100 uruchomień optymalizacji funkcji sferycznej z losowym punktem startowym
Y
X
2
1
0
-1
-2
-2
-1
0
1
2

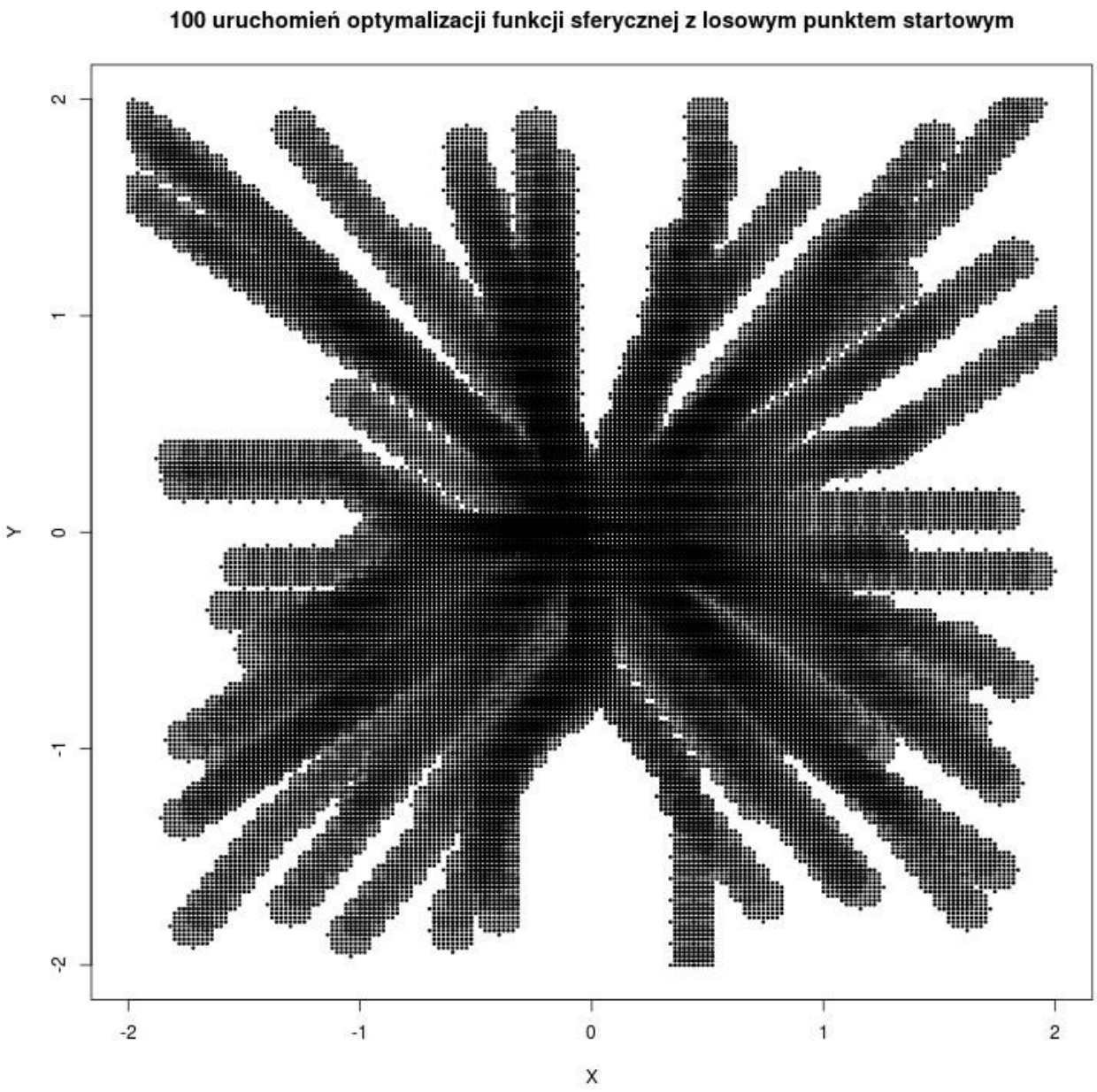
- promień sąsiedztwa 0.05



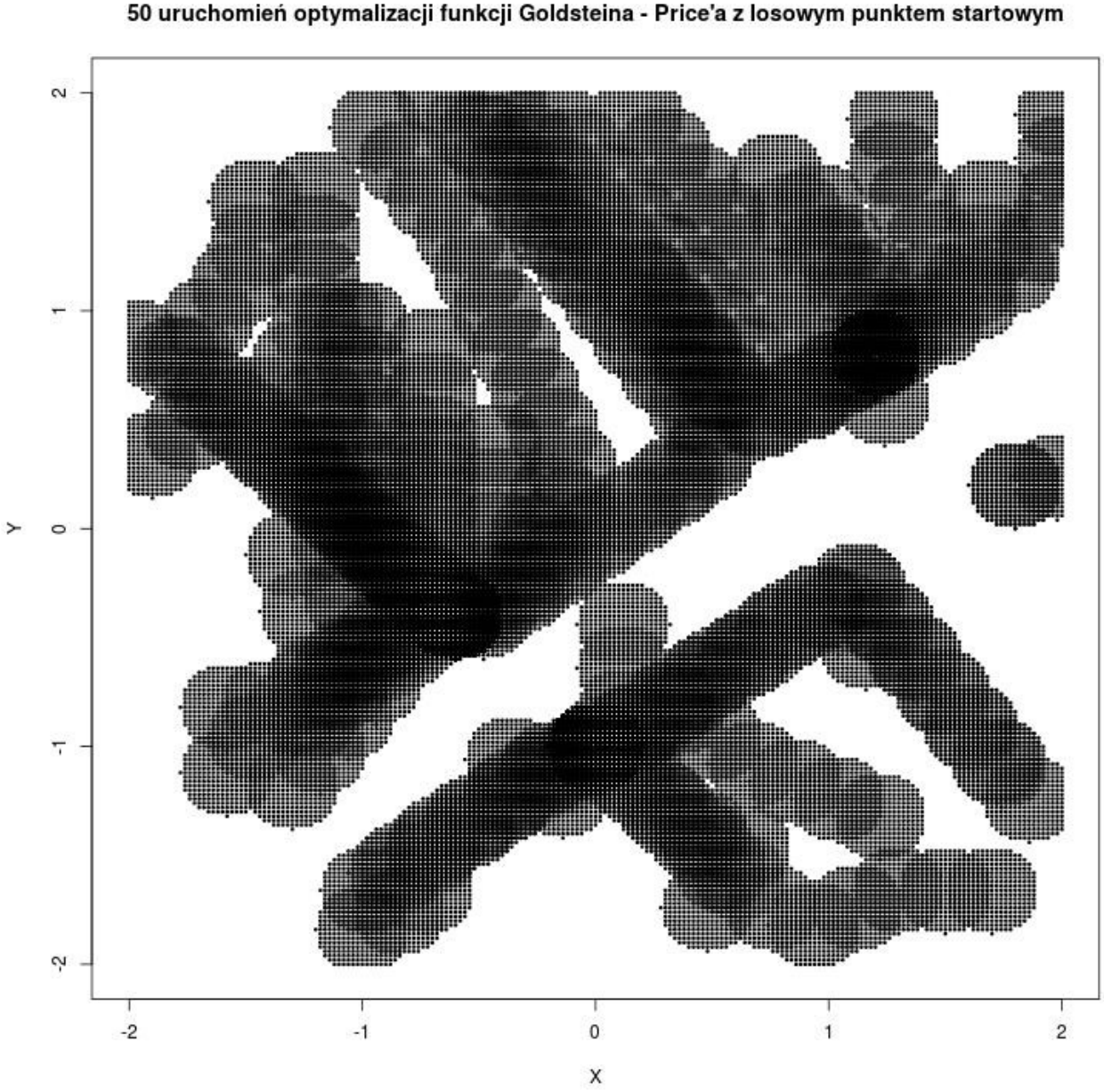

100 uruchomień optymalizacji funkcji sferycznej z losowym punktem startowym
Y
X
2
1
0
-1
-2
-2
-1
0
1
2

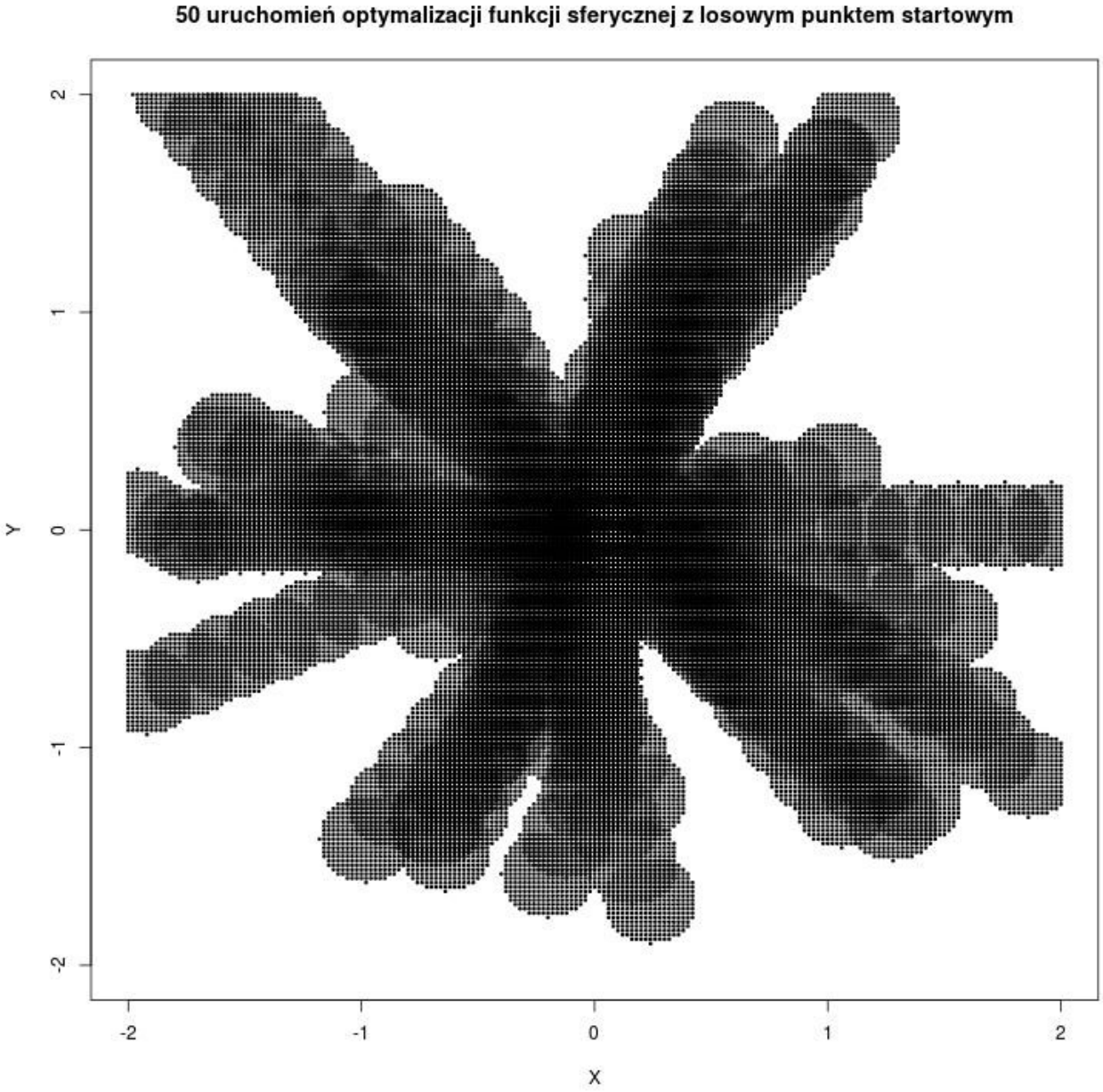
- promień sąsiedztwa 0.1



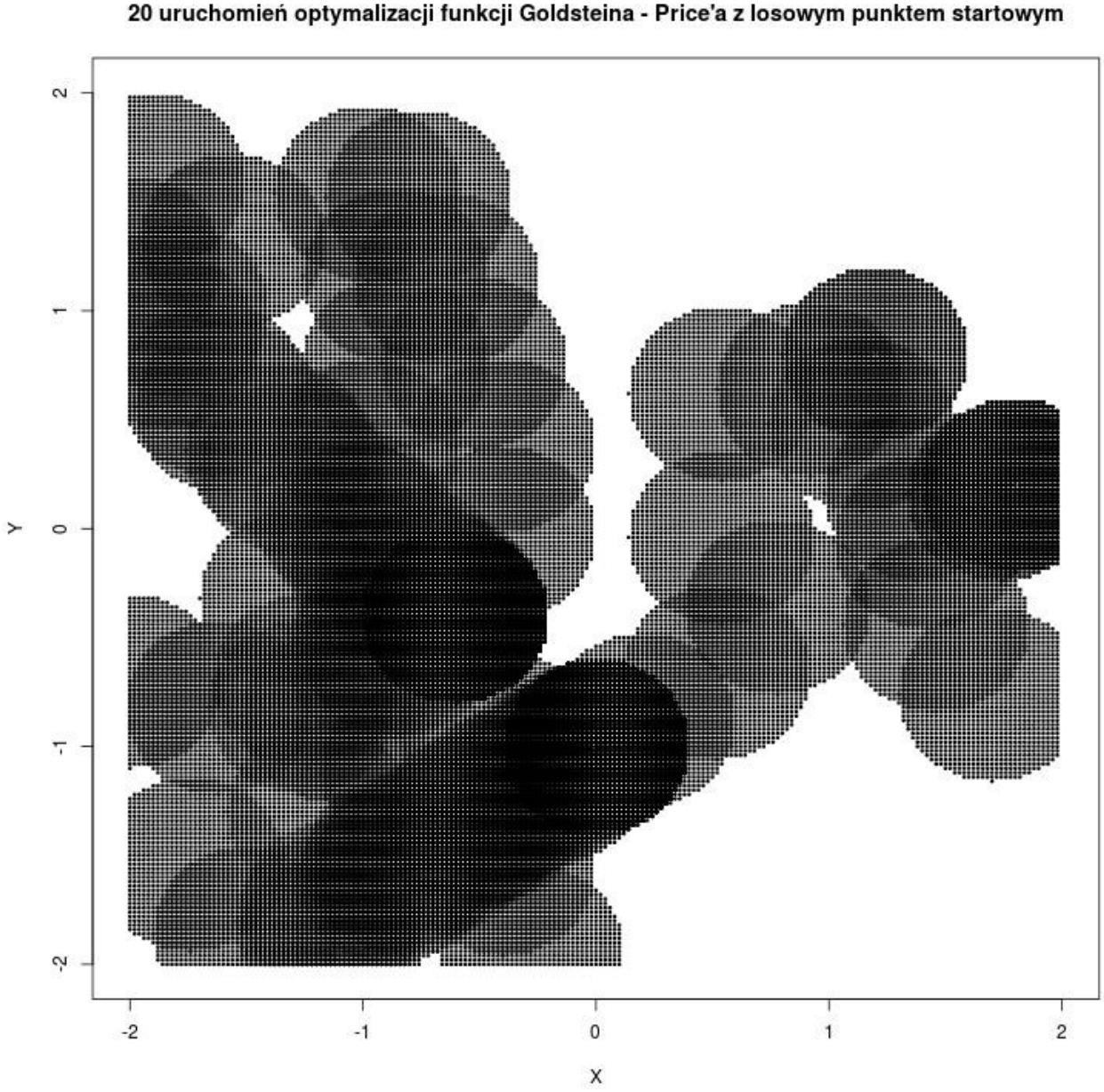

100 uruchomień optymalizacji funkcji sferycznej z losowym punktem startowym
Y
X
2
1
0
-1
-2
-2
-1
0
1
2

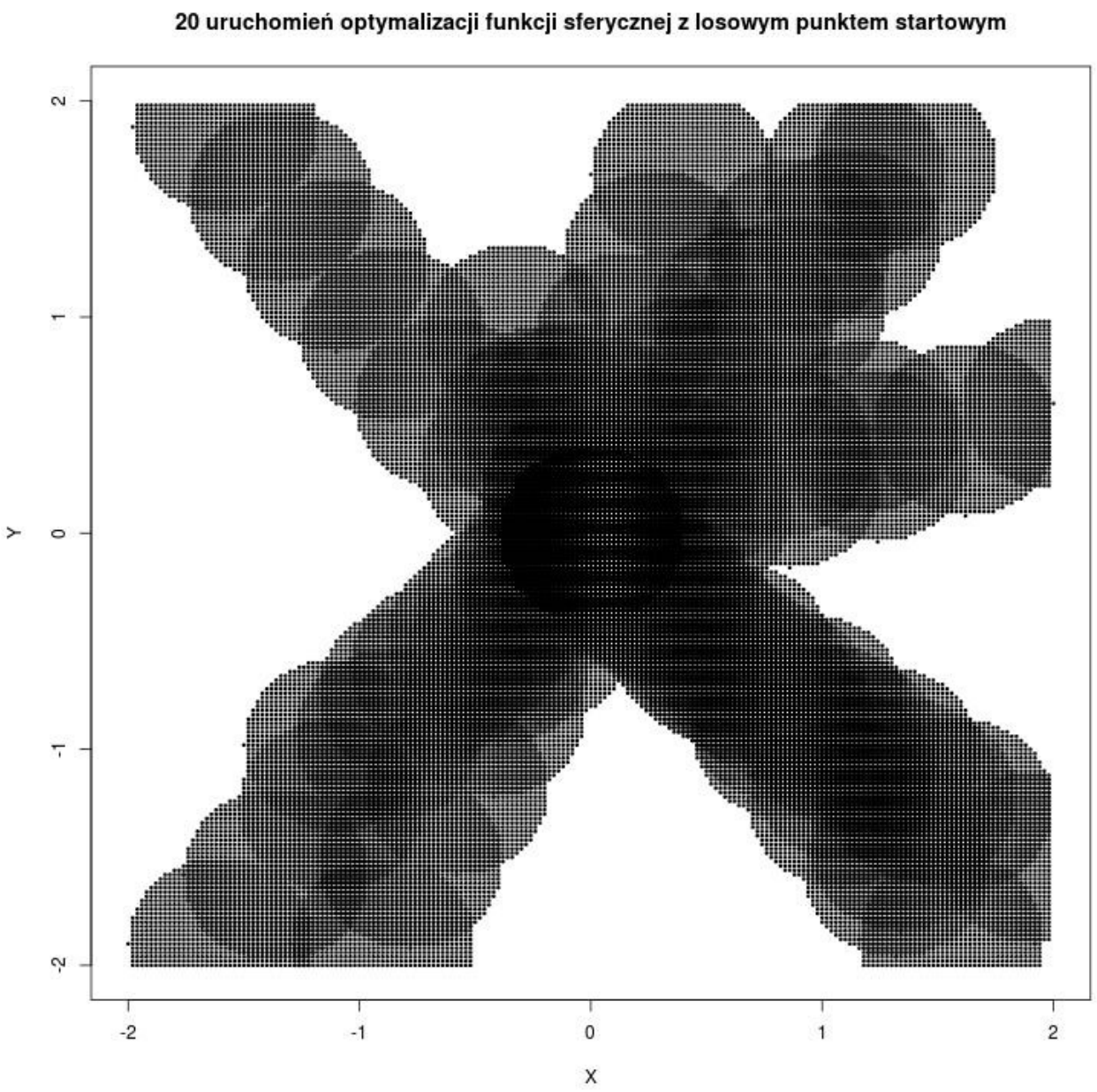
- promień sąsiedztwa 0.2



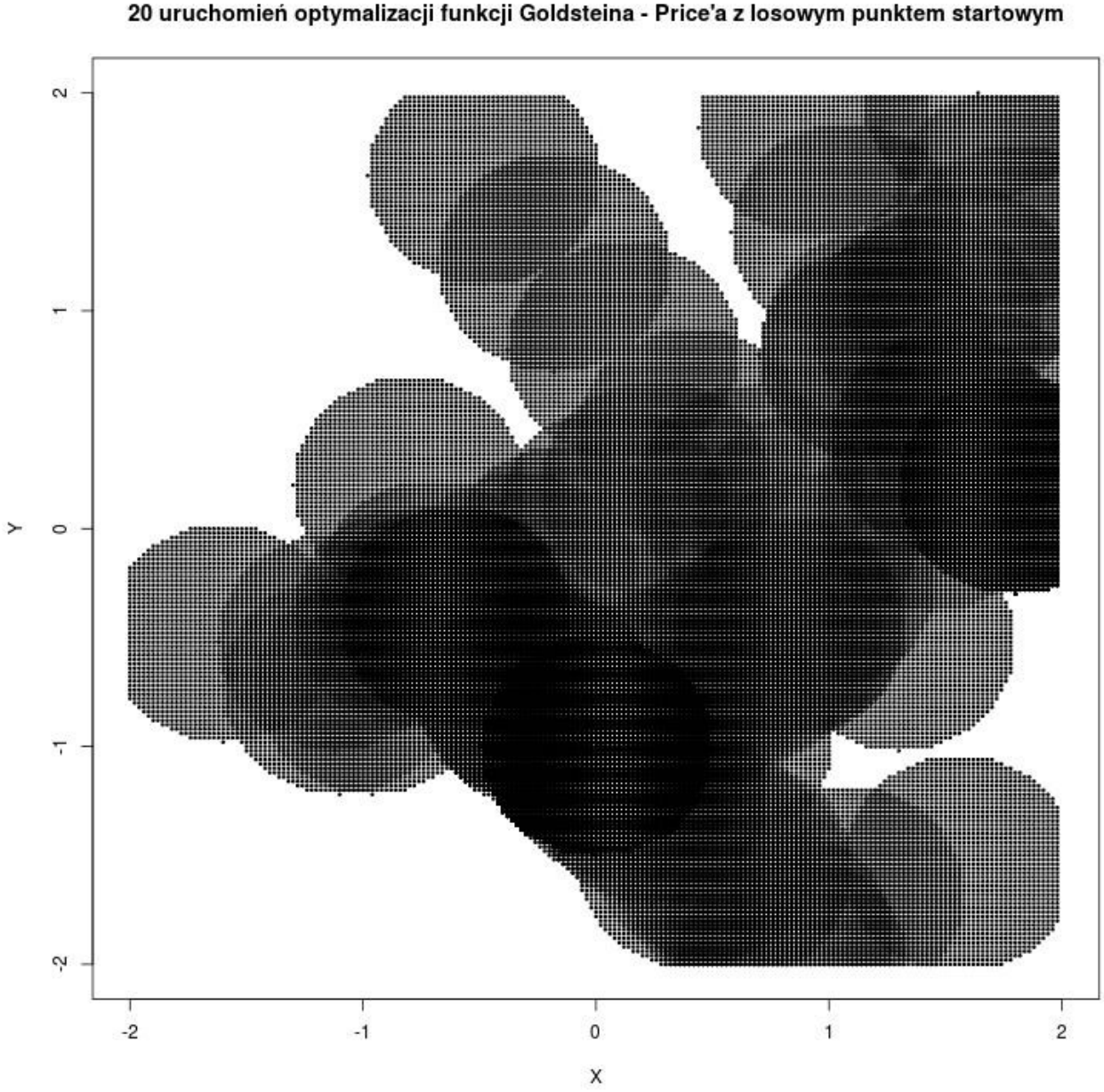

50 uruchomień optymalizacji funkcji sferycznej z losowym punktem startowym
Y
X
2
1
0
-1
-2
-2
-1
0
1
2

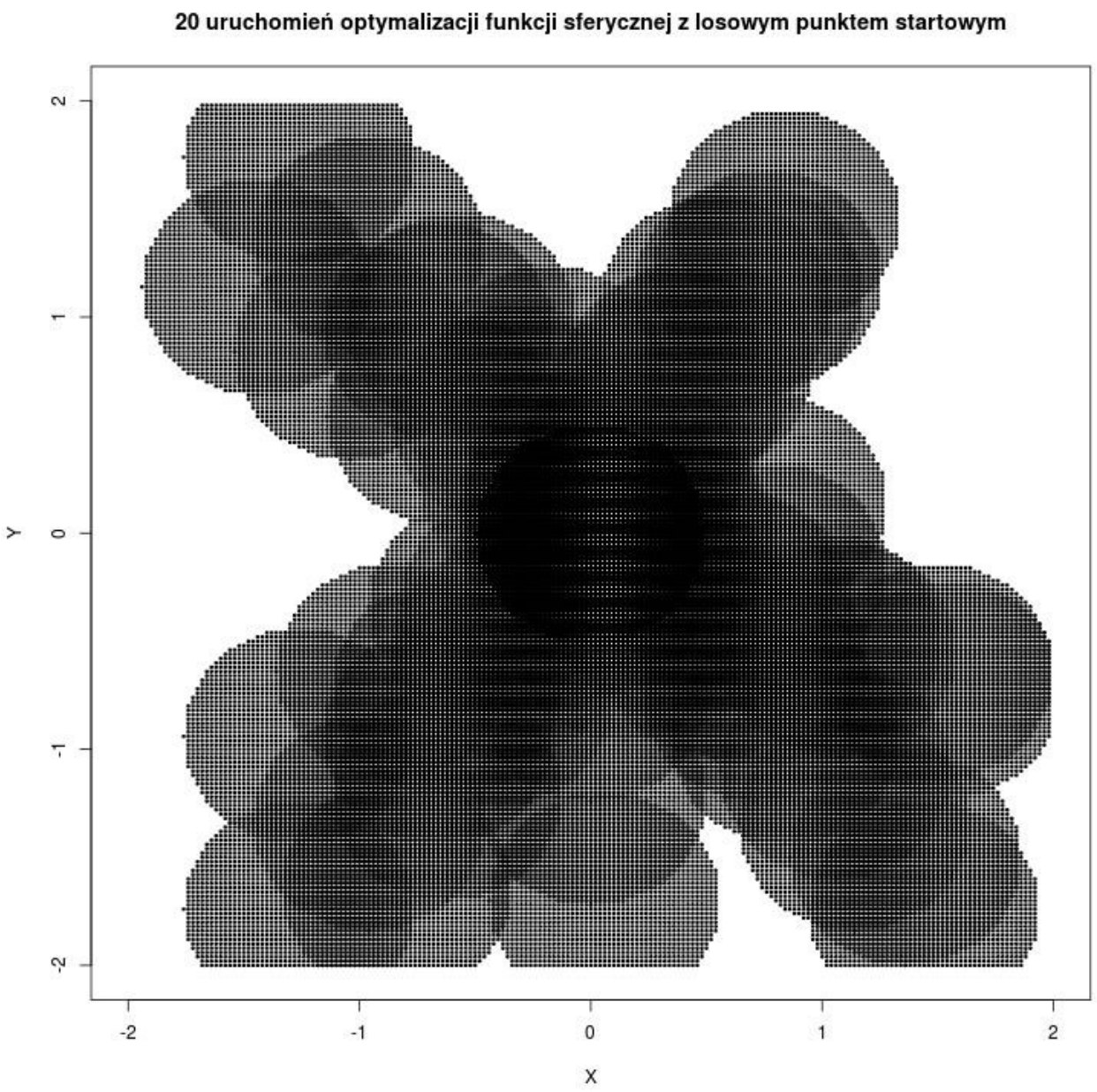
- promień sąsiedztwa 0.4




20 uruchomień optymalizacji funkcji sferycznej z losowym punktem startowym
Y
X
2
1
0
-1
-2
-2
-1
0
1
2

- promień sąsiedztwa 0.5




20 uruchomień optymalizacji funkcji sferycznej z losowym punktem startowym
Y
X
2
1
0
-1
-2
-2
-1
0
1
2

Wnioski

Na powyższych wykresach możemy zaobserwować wyraźne zagęszczenie generowanych przez algorytm punktów, w okolicach minimów lokalnych funkcji.

Dla funkcji sferycznej istnieje jedno minimum lokalne, będące jednocześnie globalnym, punkt (0, 0). W tym przypadku algorytm zawsze bez problemu znajdował minimum, niezależnie od punktu startowego algorytmu

Dla funkcji Golsteina – Price'a, minimum globalne znajduje się w punkcie (0, -1).
Algorytm nie zawsze kończył swoje działanie w minimum globalnym.
Jest to spowodowane istnieniem minimów lokalnych funkcji, możliwych do zaobserwowania na wykresach.
Przykładami takich minimów mogą być punkty

- (1, 0.8)
- (-0.5, -0.4)

Analizując powyższe wykresy, możemy stwierdzić, że algorytm wspinaczkowy jest algorytmem zachłannym.
Podczas każdej iteracji wyszukuje najlepszy przyszły punkt, który może wybrać do kolejnego obiegu. W przypadku braku lepszego punktu, kończy swoje działanie.

Drugi rodzaj testu, polegał na wyznaczeniu zależności

- średniego czasu działania algorytmu
- prawdopodobieństwa sukcesu algorytmu

od promienia sąsiedztwa dla zadanej maksymalnej ilości iteracji.

Zostało to zrealizowane poprzez uruchomienie algorytmu z następującymi parametrami

Promień sąsiedztwa	0.02	0.04	0.05	0.1	0.2	0.4	0.5
Maksymalna ilość iteracji	5	10	20	50	100	500	1000

Test został przeprowadzony w każdej możliwej kombinacj powyższych parametrów wykonania(49) dla każdej z funkcji, co daje w sumie 98 testów.

Aby uzyskać w miarę wiarygodne, średnie wyniki, każdy test został uruchomiony 1000 razy, po czym wyniki z uruchomień zostały uśrednione.

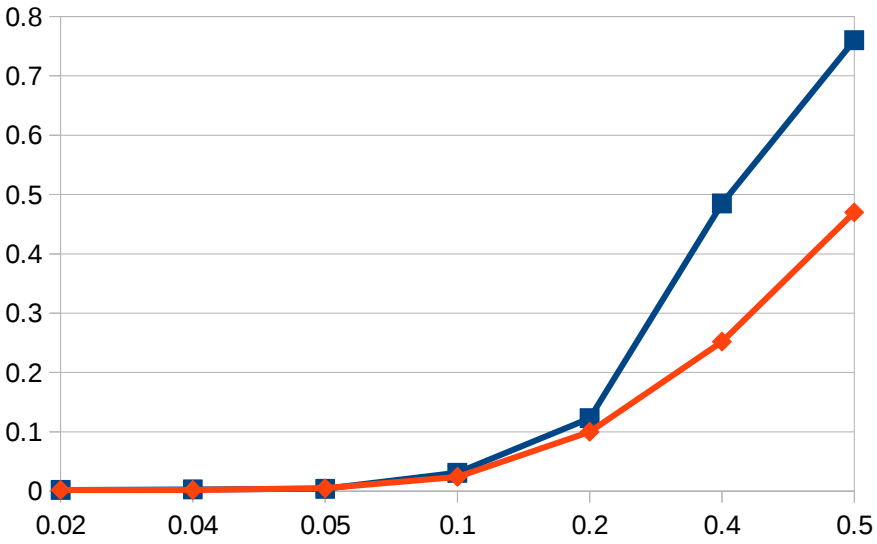
Zatrzymanie uruchomienia programu następowało w przypadku znalezienia minimum funkcji, lub przekroczenia maksymalnej ilości iteracji.

Wykresy zostały przedstawione poniżej.

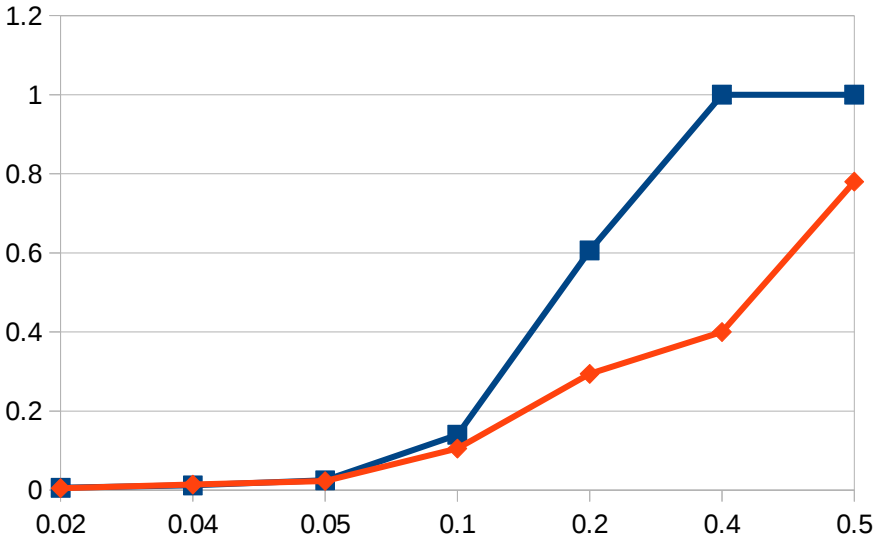
Czerwona linia symbolizuje prawdopodobieństwo znalezienia minimum globalnego funkcji

Goldsteina – Price'a, w zalezności od promienia sąsiedztwa, przy określonej maksymalnej liczbie iteracji algorytmu. Analogicznie niebieska linia symbolizuje sytuację w przypadku optymalizacji funkcji sferycznej.

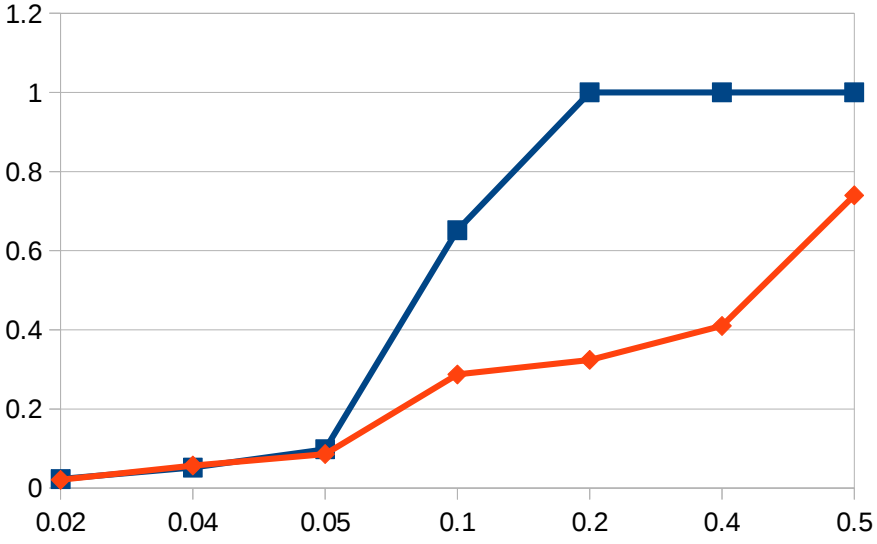
maksymalna ilość iteracji: 5



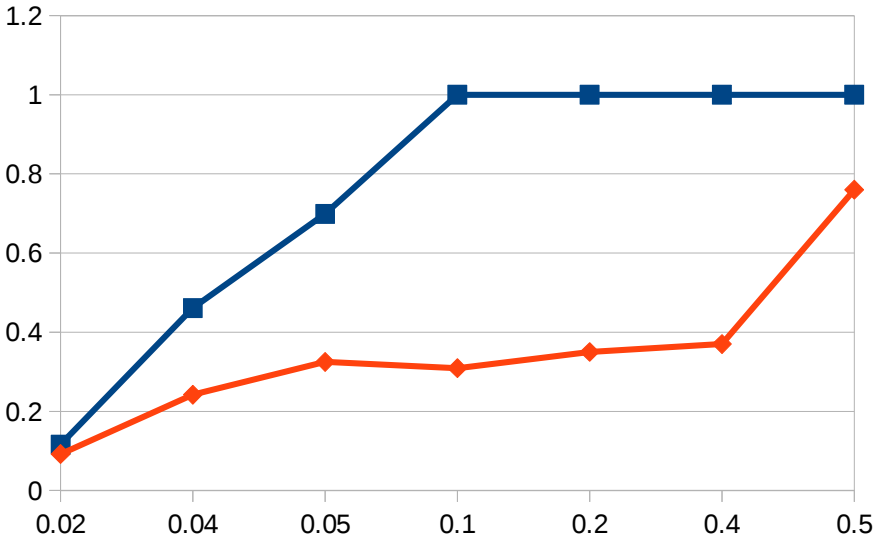
maksymalna ilość iteracji: 10



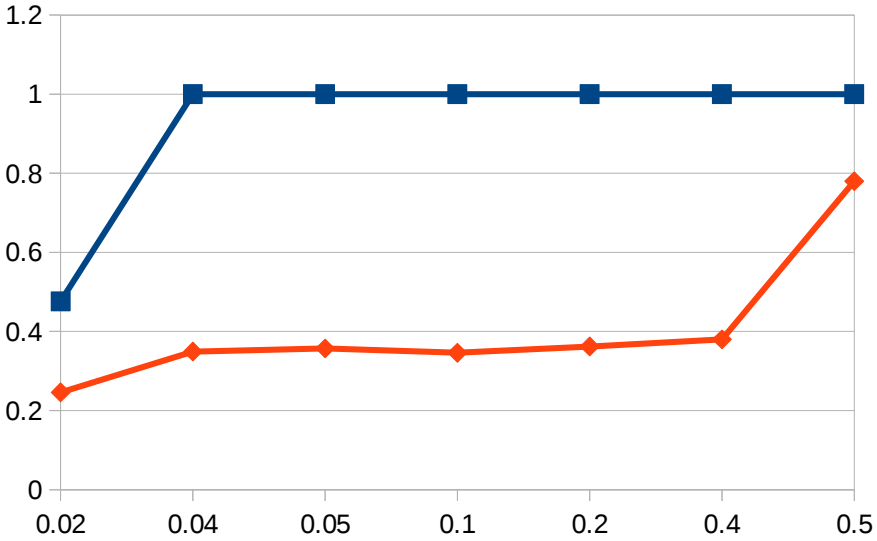
maksymalna ilość iteracji: 20



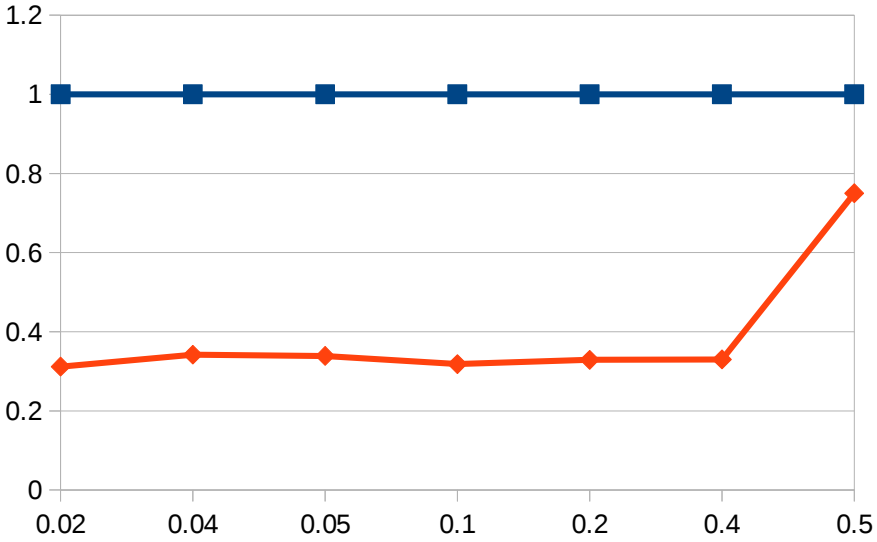
maksymalna ilość iteracji: 50



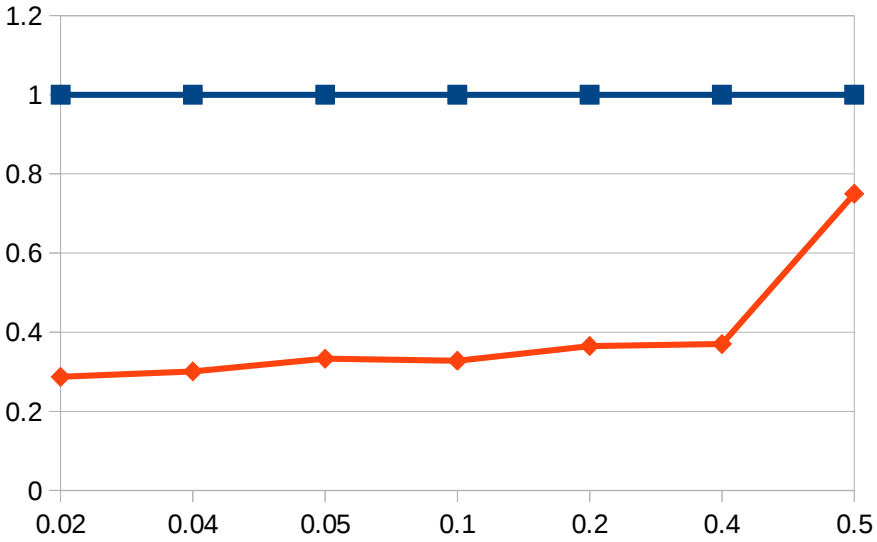
maksymalna liczba iteracji: 100



maksymalna liczba iteracji: 500



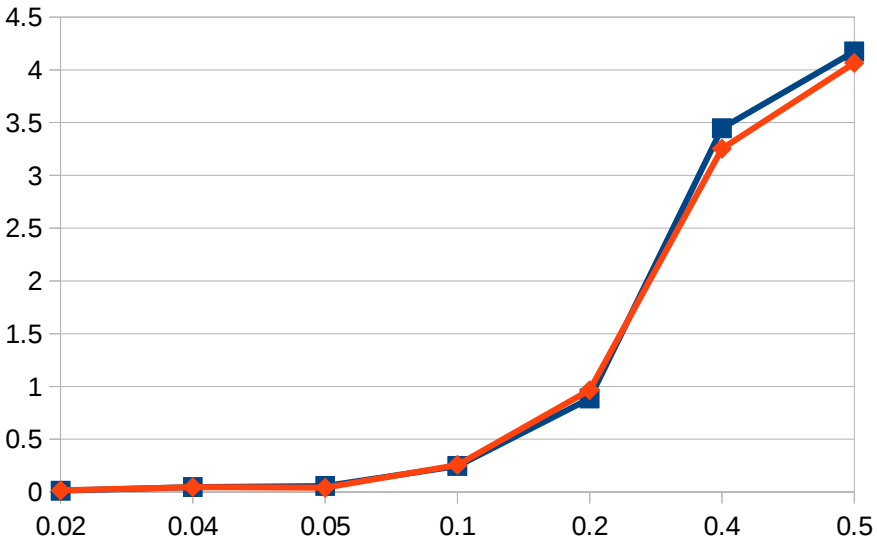
maksymalna ilość iteracji: 1000



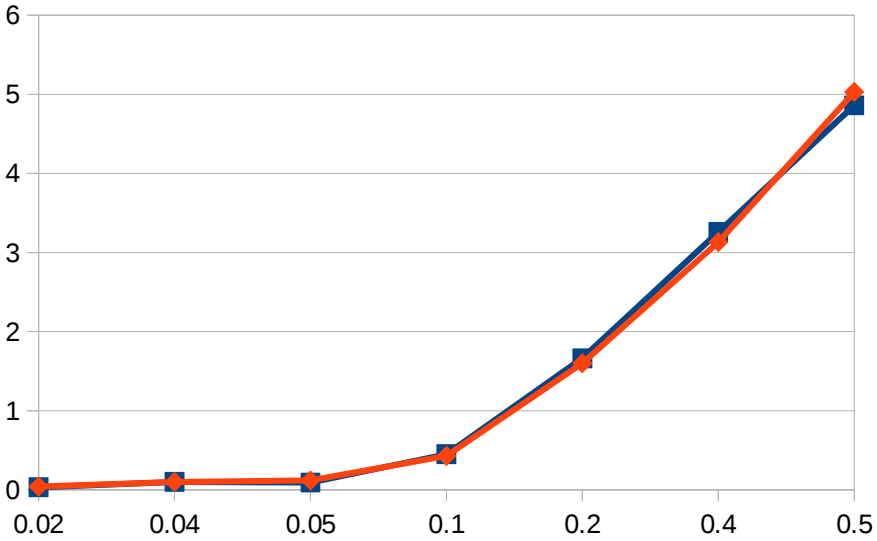
Test szybkości działania

Czerwona linia symbolizuje średni czas działania algorytmu optymalizacji funkcji Goldsteina – Price'a, w zależności od promienia sąsiedztwa, przy określonej maksymalnej liczbie iteracji algorytmu. Analogicznie niebieska linia symbolizuje sytuację w przypadku optymalizacji funkcji sferycznej. Wartość wyrażona jest w sekundach, i nie uwzględnia czy minimum globalne zostało rzeczywiście znalezione.

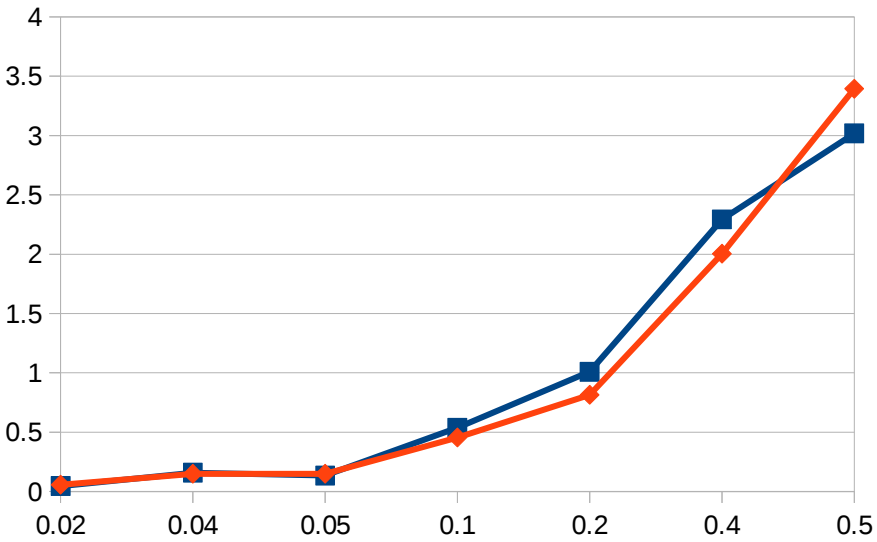
maksymalna ilość iteracji: 5



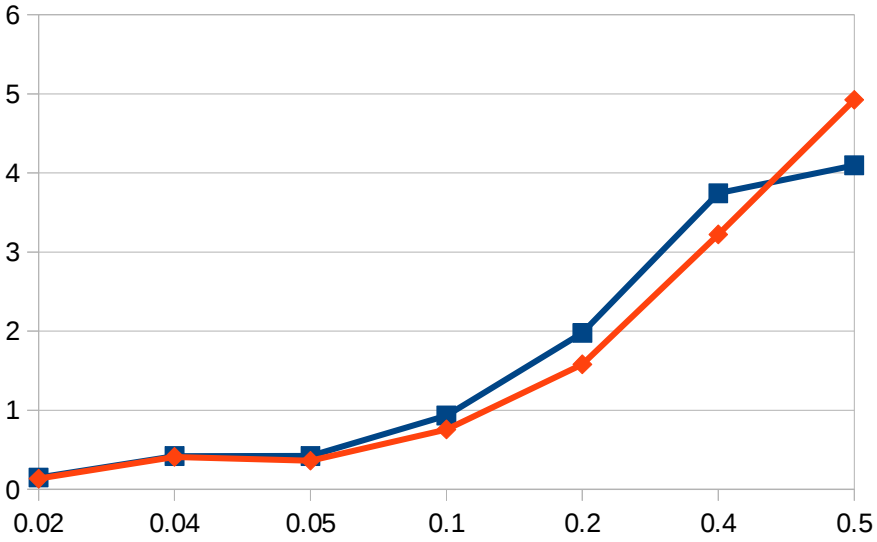
maksymalna ilość iteracji: 10



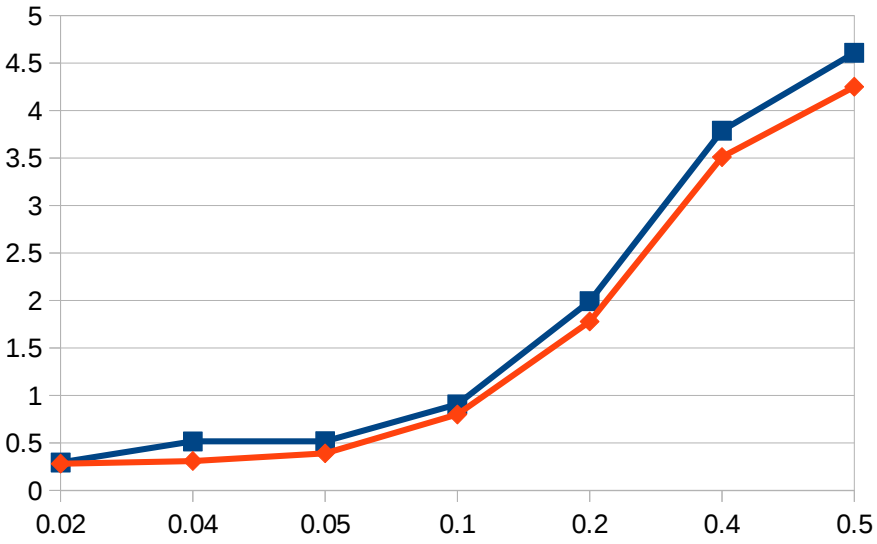
maksymalna ilość iteracji: 20



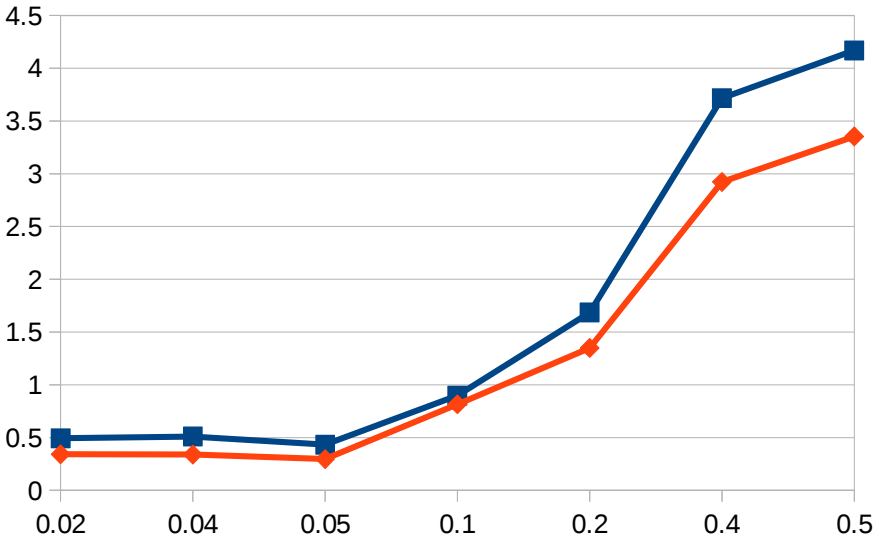
maksymalna ilość iteracji: 50



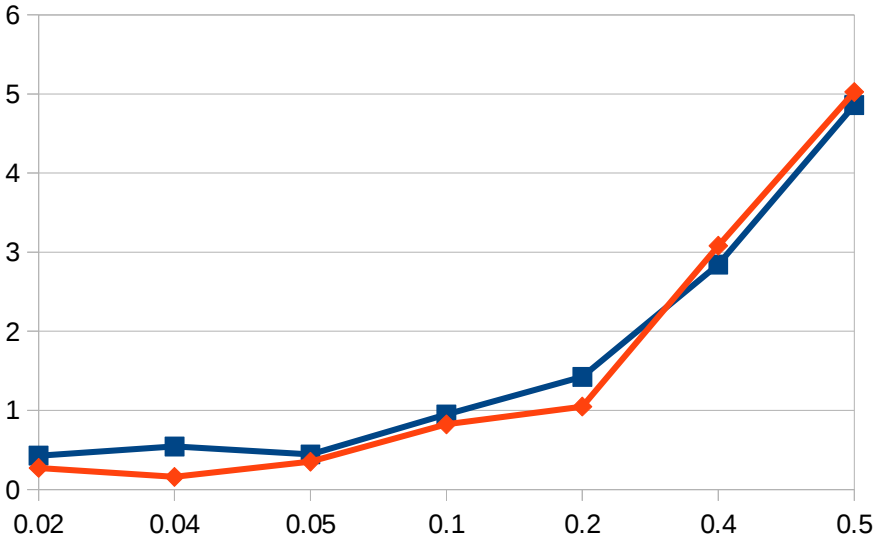
maksymalna ilość iteracji: 100



maksymalna ilość iteracji: 500



maksymalna ilość iteracji: 1000



Wnioski

Po przeprowadzeniu powyższych testów, potwierdziło się moje początkowe przekonanie o naiwności algorytmu wspinaczkowego.
Oczywiście, z uwagi na swoją prostotę i łatwość implementacji, jego użycie może byc wygodne, ale raczej tylko do funkcji o niewielkiej (a najlepiej zerowej) ilości minimów lokalnych, różnych od globalnego.

Tendencja do eksploatacji, jest dobrze widoczna przy analizie prawdopodobieństwa sukcesu algorytmu, w przypadku optymalizacji *funkcji Goldsteina – Price'a*, posiadającej kilka minimów lokalnych.

Dodatkowo powodzenie algorytmu, jest w dużej mierze zależne od wylosowanego punktu początkowego.
Oczywiście, możemy manipulować promieniem sąsiedztwa algorytmu, co pozwoli na większe szanse wyjścia ze zbioru przyciągania danego minimum, jednakże należy liczyć się z dłuższym czasem działania wymaganym przez analizę większej ilości punktów.

Czas działania, może jednak być redukowany poprzez obniżenie maksymalnej ilośći iteracji algorytmu, jednakże wtedy algorytm nawet jeśli będzie zmierzał w kierunku minimum globalnego, może zakończyć swoje działanie, przed jego osiągnięciem.

W przypadku funkcji o jednym minimum, algorytm zawsze je znajduje, o ile nie przekroczy zadanej ilości iteracji.

Program testujący

Na potrzeby projektu, stworzony został program ułatwiający testowanie i analizę otrzymanych wyników.

Całość kodu podzielona jest na 2 pliki
- *evaluators.R*
- *hillclimbing.R*

Pierwszy z nich zawiera kod funkcji podlegających optymalizacji, czyli w naszym przypadku, *funkcja sferyczna* i *funkcja Goldsteina – Price'a.*
Taki podział programu, ułatwia potencjalne przyszłe testy innych funkcji.

Głównym modułem aplikacji jest drugi z plików.
Znajduje się tam cały kod algorytmu, bazujący na otrzymanym szkielecie metody heurystycznej.

Algorytm można wywoływać na 2 sposoby

1. przeprowadzenie pojedyńczego testu
   a) należy wywołać funkcję *runTests()* z następującymi parametrami (wszystkie są opcjonalne):
      - *run_num [= 50]*, oznacza ilość uruchomień algorytmu, których wyniki zostaną następnie nałożone na jeden wykres
      - *width [= 743]*, oznacza szerokość obrazka z wykresem
      - *height [= 743]*, wysokość obrazka
      - *start [= F]*, oznacza punkt startowy algorytmu
        - należy podawać wartość w formie c(x, y)
        - wartość F oznacza punkt losowy
      - *rate [= 0.02]*, oznacza częstość próbkowania przestrzeni punktów, czyli odległość między najbliższymi punktami w przestrzeni
      - *dimension [= 2]*, oogranicza przestrzeń punktów
        - algorytm będzie działał tylko w dziedzinie "kwadratowej" ograniczonej przez punkty *(-dimension, -dimension) i (dimension, dimension)*
      - *alg_iter [= 1000]*, oznacza maksymalną ilość iteracji algorytmu wspinaczkowego
      - *neigh [= 0.5]*, oznacza promień sąsiedztwa, jest to parametr algorytmu
      - *maxym [= F]*, oznacza "kierunek" optymalizacji
        - wartość F spowoduje minimalizację funkcji, wartość T maksymalizację

   b) Rezultatem takiego wywołania algorytmu będą pary plików, po jednej dla każdej z funkcji.
      Będą one nazywane w formacie:
      - *[neigh]_[alg_iter]_[run_num]_[metoda].jpg*
      - *[neigh]_[alg_iter]_[run_num]_[metoda].txt*

      W pierwszym będzie znajdował się wykres uzyskany przez naniesienie *run_num* rezultatów na jeden wykres.
      W drugim pliku będą znajdować się zmierzone parametry testu
      - średni czas działania algorytmu dla zadanych parametrów
      - prawdopodobieństwo uzyskania minimum globalnego

Poprawność wyników testu znajdowana jest poprzez porównanie współrzędnych otrzymanego punktu końcowego, ze współrzędnymi minimum globalnego, umiejscowionego w tablicy *results.*

c) przykładowe wywołanie
   - *runTests(neigh=0.2, run_num=50, alg_iter=100)*

2. przeprowadzenie serii testów
   a) należy wywołac funkcję *perform()* z następującymi parametrami
      - *args*, jest to lista wektorów parametrów początkowych
      - każdy wektor składa się z parametrów
        (nie są to parametry typu keyword, i są wymagane)
        - *neigh*, oznacza promień sąsiedztwa
        - *alg_iter*, oznacza maksymalną ilość iteracji algorytmu

   b) rezultatem będzie wywołanie funkcji *runTests()* dla każdego wektora parametrów początkowych, zadanego w *args*

   c) przykładowe wywołanie
      - *perform(list(c(0.05, 200), (0.05, 500), c(0.05, 1000)))*

Porównanie algorytmów

Algorytm ewolucji różnicowej okazał się być bardzo dobrym algorytmem do rozwiązania problemu znalezienia minimum globalnego funkcji sferycznej i funkcji Goldsteina-Price'a. Nie było problemem dobranie takich parametrów uruchomieniowych algorytmu, aby za każdym uruchomieniem otrzymywać oczekiwany wynik. Inna sytuacja wystąpiła w przypadku algorytmu wspinaczkowego, jest to algorytm zachłanny. Dla funkcji sferycznej było możliwe dobranie takich parametrów uruchomieniowych, które by pozwoliły za każdym razem otrzymywać oczekiwany wynik.
Jednakże dla funkcji Goldsteina-Price'a nie dało się dobrać takich parametrów.
Testy wykazywały, że przy odpowiednio dużym promieniu sąsiedztwa w porównaniu do rozmiarów przestrzeni, prawdopodobieństwo znalezienia szukanego punktu zwiększa się, jednakże czas przeszukiwania stawał się wtedy zbyt długi.

Porównując oba algorytmy możemy jednoznacznie stwierdzić wyższość algorytmu ewolucji różnicowej nad algorytmem wspinaczkowym.