Wydział Inżynierii Mechanicznej i Robotyki

Katedra Robotyki i Mechatroniki

Podstawy Sztucznej Inteligencji I Uczenia Głębokiego

Inżynieria Mechatroniczna

Instrukcja 1:

Wprowdzenie do laboratoriów
oraz
Podstawowe metody optymalizacji

Nauczysz się: Jak napisać od podstaw i skonfigurować podstawowe algorytmy optymalizacyjne: grid search, 1+1 oraz gradientowy. Nauczysz się podstawowego ustawiania metaparametrów oraz zarządzania wynikami testów. Te algorytmy posłużą jako podstawa dla bardziej zaawansowanych rozwiązań decyzyjnych w przyszłości (m.in. nauki klasyfikatorów i regresorów)

Dodatkowe materiały:

- Wykład nr 1 [*obowiązkowy*]

Ta instrukcja została przetłumaczona przez ChatGPT na podstawie angielskiego oryginału – materiałów z przedmiotu *Basic of Artificial Intelligence and Deep Learning*

Koordynator przedmiotu:
Krzysztof Holak, holak@agh.edu.pl

Autor instrukcji:
Ziemowit Dworakowski, zdw@agh.edu.pl

Podstawowe informacje dotyczące organizacji instrukcji laboratoryjnych

Struktura laboratorium

Laboratoria są przygotowane w oparciu o metodę Odwróconej Klasy. Oznacza to, że zalecanym sposobem realizacji zajęć jest wstępne wykonanie części zadań w domu, a następnie dokończenie pracy w trakcie zajęć. Im więcej wykonasz w domu, tym więcej wiedzy zdobędziesz i (miejmy nadzieję) tym łatwiej będzie Ci ukończyć cały kurs.
Każda instrukcja obejmuje wyjaśnienie kroków potrzebnych do rozwiązania problemu (na białym tle), a następnie serię zadań do wykonania przez Ciebie. Zadania są podzielone na trzy kategorie: „Na ocenę 3.0" – oznaczone na czerwono, „Na ocenę 4.0" – oznaczone na pomarańczowo, „Na ocenę 5.0" – oznaczone na zielono.

Ukończenie określonego zestawu zadań podczas laboratorium skutkuje uzyskaniem oceny warunkowej. Przykładowo, jeśli podczas instrukcji 1 ukończysz wszystkie zadania oznaczone na czerwono i pomarańczowo, otrzymasz ocenę 4.0, pod warunkiem, że podczas kolejnego laboratorium obronisz komplet ćwiczeń z pierwszej instrukcji, w tym również te oznaczone na zielono. Innymi słowy: im więcej zadań zrobisz podczas zajęć, tym wyższą otrzymasz ocenę, ale wcześniej czy później i tak konieczne będzie wykonanie wszystkich zadań.

Kolory oznaczają także przewidywany poziom trudności zadań: Czerwone – proste, możliwe do wykonania poprzez ścisłe podążanie za instrukcją. Pomarańczowe – wymagają pewnej implementacji kodu lub podejmowania decyzji. Zielone – obejmują rzeczywiste problemy, często wymagają interakcji z prowadzącym. Z tego powodu zalecane podejście zakłada rozwiązanie czerwonych zadań w całości w domu przed zajęciami oraz podjęcie próby rozwiązania pomarańczowych. Zielone najlepiej zostawić na same zajęcia.

Poprawianie nieobecności lub braku zaliczenia instrukcji

Proces uzyskania zaliczenia z instrukcji laboratoryjnej może być zakłócony na trzy sposoby:

- Możesz być nieobecny (nie będziesz wówczas miał okazji uzyskać „oceny warunkowej")
- Możesz nie poradzić sobie z zadaniami „na 3.0"
- Możesz nie poradzić sobie z obroną kompletu zadań na kolejnym spotkaniu.

W takich przypadkach konieczne jest przygotowanie raportu laboratoryjnego, zawierającego wszystkie wymagane zadania (na 3.0, 4.0 i 5.0) oraz dodatkowe zadanie wybrane przez prowadzącego – z puli niebieskich zadań na końcu instrukcji. Raport należy obronić. Szczegóły dotyczące wymagań raportu są dostępne pod tym linkiem: http://galaxy.agh.edu.pl/~zdw/Documents/RaportTechniczny.pdf

Uwaga: Możliwość zaliczenia laboratorium w ten sposób jest ograniczona do trzech przypadków. Dalsze korekty wymagają konsultacji z koordynatorem przedmiotu, który określi zakres pracy wymagany do uzyskania zaliczenia lub zaleci ponowne podejście do przedmiotu w kolejnym roku.

Testy i poprawki testów

Niektóre laboratoria rozpoczną się testem sprawdzającym wiedzę. Harmonogram testów jest dostępny w notatkach z pierwszego wykładu. Każdy test można poprawiać dwukrotnie pod koniec kursu. Jeśli jednak nie zaliczysz więcej niż dwóch testów, poprawa będzie uwzględniała materiał z całości kursu.

Wprowadzenie do zadań optymalizacyjnych

Punkt startowy dla ćwiczeń wykonywanych w ramach laboratoriów to biblioteka funkcji udostępniona przez prowadzącego:

- ***op_f_RandomSampling.m*** – funkcja optymalizująca funkcję dwuwymiarową metodą losową, z wizualizacją działania i krzywą zbieżności.
- Zbiór funkcji w katalogu ***FunctionsForOptimization*** o różnych cechach: z jednym minimum lokalnym, kilkoma minimami lokalnymi lub wieloma minimami lokalnymi.

Wszystkie programy powinny być przygotowywane jako **osobne skrypty** i przechowywane do wykorzystania w przyszłych zajęciach.

Wszystkie regulowane parametry w kodach (np. wartości stałych, liczba iteracji, kroki, zakresy losowania itp.) powinny być umieszczone na początku kodu i opatrzone czytelnym komentarzem.

Indywidualne funkcje i zadania

W większości ćwiczeń będziesz rozwiązywać problem zdefiniowany za pomocą konkretnej funkcji celu. Ta funkcja nazywana będzie w instrukcji Twoją **Indywidualną Funkcją** lub **Indywidualnym Zadaniem**. Każdy student może mieć przypisaną inną funkcję, jeśli prowadzący nie poda innej metody – numer Twojej funkcji można obliczyć jako resztę z dzielenia sumy liter w Twoim imieniu i nazwisku przez 8. Przykładowo, *Jane Doe* będzie pracowała z funkcją nr 7 a *John Smith* będzie pracował z funkcją nr **1**

Zadanie początkowe: uruchomienie i testowanie kodu podstawowego

Uruchom kod ***of_f_randomSampling*** dostępny w bibliotece kursu. Kod optymalizuje funkcję celu o dwóch parametrach – i w tym przykładzie posłużył do zoptymalizowania funkcji danej plikiem ***of_2D_oneminimum_2***. Ta funkcja jest dwuwymiarowa (dwuparametryczna – 2D), posiada jedno minimum (nie posiada minimów lokalnych), i jest drugą funkcją w bibliotece kursu o tych właściwościach. Po uruchomieniu kodu zobaczysz **wykres funkcji** oraz **punkty testowe**, w których wartość funkcji była testowana metodą losową.

Zapoznaj się z kodem i jego komentarzami, aby zrozumieć zasadę działania, zwłaszcza jak generowane są współrzędne punktów testowych i jak działa pętla ***while***.

Prawidłowe działanie kodu kończy się wyświetleniem **krzywej zbieżności**, przedstawiającej historię poszukiwania minimum. Krzywa zbieżności składa się z dwóch komponentów: wartości testowanej w każdej iteracji oraz historycznie osiągniętego minimum (czyli wartości reprezentującej najlepsze znalezione rozwiązanie).

Spróbuj uruchomić kod i przetestować jego działanie na przykładzie kilku różnych funkcji celu. Sprawdź działanie dla funkcji z kategorii *Manyminima* oraz *Fewminima*. Teraz nie będziemy potrzebować funkcji z dopiskami *adaptive* oraz *rand*.

Podstawowe algorytmy optymalizacyjne

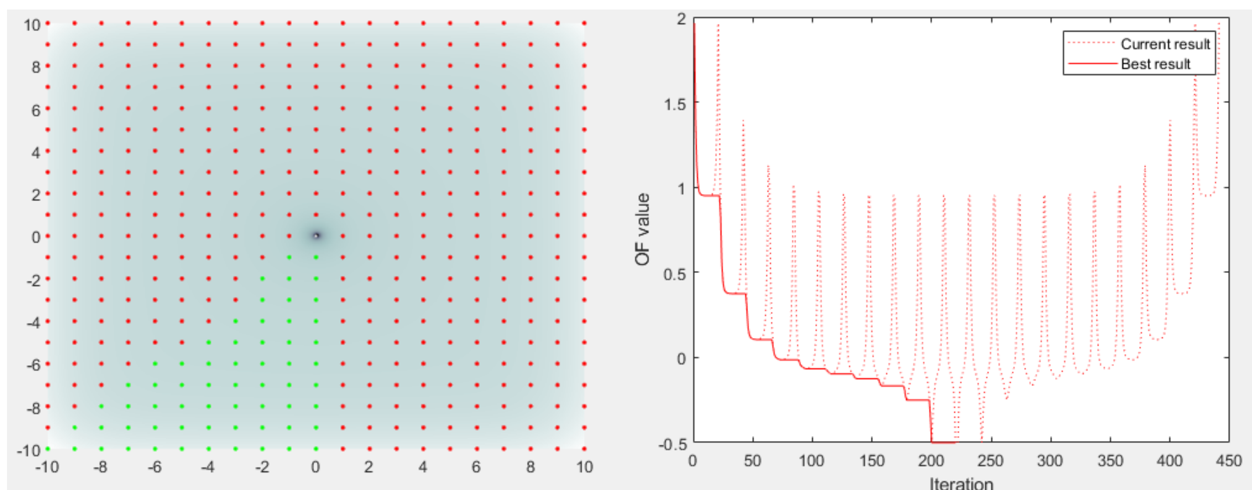
Wróćmy do optymalizacji funkcji z jednym minimum lokalnym (*of_2D_oneminimum_2*) i na tej podstawie zbudujmy algorytm przeszukiwania siatkowego (*grid search*).

Algorytm będzie sprawdzał wszystkie punkty w węzłach siatki, zdefiniowanej przez wartości rozmieszczone równomiernie na obu osiach przestrzeni parametrów. Najlepszym podejściem do tego będzie zastąpienie pętli *while* dwiema zagnieżdżonymi pętlami *for*:

```
    for NewX = Range(1,1):step_x:Range(1,2)
        for NewY = Range(2,1):step_y:Range(2,2)

        iter = iter + 1;
        Point = [NewX,NewY];

          ... % Here comes the rest of what was previously inside while loop
    end
end
```

Tutaj *step_x* i *step_y* to oczywiście odległości między węzłami siatki, podczas gdy rozmiar przeszukiwanego obszaru jest określony przez wartości *MaxRangeX* i *MaxRangeY*. Oczywiście, ponieważ generujemy wartości *x* i *y* parametru *Point* w pętli *for*, nie musimy ich później losować.

Jeśli uzyskany obraz wygląda podobnie do tego przedstawionego na rys. 1, oznacza to, że algorytm działa poprawnie.



Rys 1 – Przykład działania metody GridSearch

Zadanie 1.1: Skonfigurujmy i przetestujmy algorytm przeszukiwania siatkowego (*grid search*) w celu optymalizacji Twojej **indywidualnej funkcji** 2D, która posiada kilka minimów lokalnych (*2D, fewminima*). Uruchom tę metodę trzykrotnie, zapisując wyniki w Tabeli 1. Po wykonaniu tego kroku zapisz kod jako osobny skrypt.

Teraz zaimplementujemy optymalizację opartą na metodzie gradientowej. Punktem wyjściowym ponownie będzie skrypt *of_f_randomSampling*. Tym razem współrzędne kolejnego punktu testowego nie będą generowane losowo, lecz wybierane w kierunku najszybszego spadku gradientu (*steepest gradient descent*).

Rozpocznijmy optymalizację od losowo wybranego punktu, wygenerowanego przed rozpoczęciem pętli *while*, np. w taki sposób:

```
Point = Range(:,1)' + rand(1,dimensions).*(Range(:,2)-Range(:,1))';
```

Teraz przejdziemy do pętli *while*, w której będziemy obliczać gradient funkcji celu (*OF*) wokół punktu testowego i na tej podstawie wybierać nowy punkt testowy dla kolejnej iteracji pętli. Aby obliczyć gradient, należy wyznaczyć wartość funkcji *OF* w naszym punkcie oraz w punktach położonych w niewielkiej odległości (*g_step*) wzdłuż wszystkich osi. Warto zwrócić uwagę na metaparametr *dimensions*, który określa, w ilu kierunkach chcemy obliczać pochodne cząstkowe funkcji *OF*. W naszym przypadku *dimensions* powinno być ustawione na 2. Gdy będziesz przygotowywać funkcję do współpracy z większą ilością wymiarów, metaparametr ten będzie oczywiście wymagał zmiany.

```
 CurrentValue = FunctionForOptimization(Point);
 for d = 1:dimensions   % "dimensions" says how long is our parameter vector
     TestPoint = Point;
     TestPoint(d) = TestPoint(d) + g_step;
     CV_d(d) = FunctionForOptimization(TestPoint);
 end

 Grad = CV_d - CurrentValue;

% If we want to use just direction of the gradient - this is the way to go:
if(max(abs(Grad))==0)
  Grad = 0
else
 Grad = Grad/max(abs(Grad));
end
```

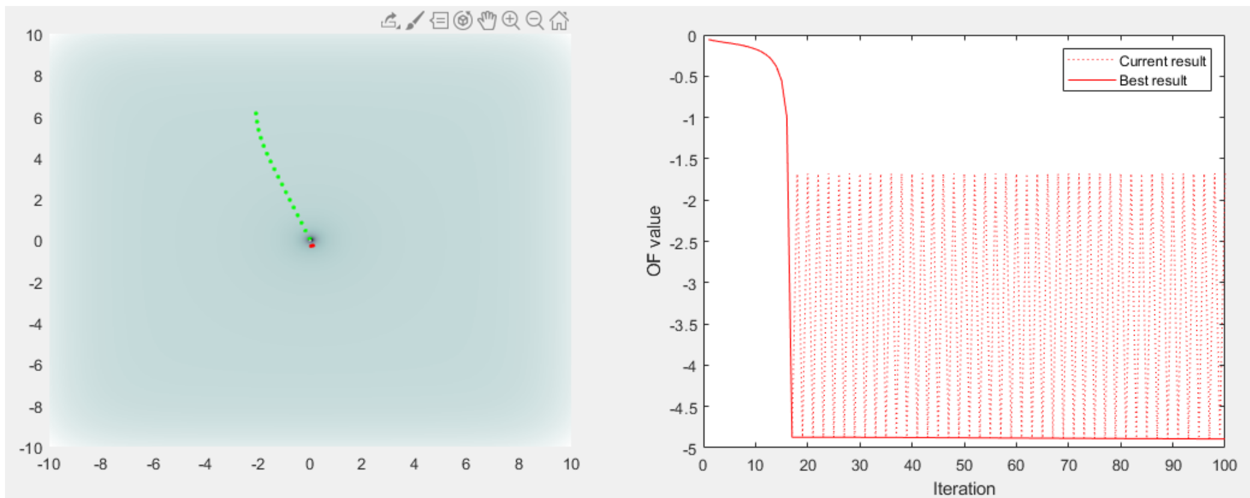
Następnie wybierzemy kolejny punkt do następnej iteracji pętli, mnożąc poprzednie współrzędne przez znormalizowany gradient oraz stałą *Step*:

```
Point = Point - Step*Grad;
```

Chciałbym zwrócić Twoją uwagę na metaparametry w tym kodzie. Mamy tutaj *step_g* oraz *Step*. Pierwszy z nich określa, jak blisko siebie znajdują się punkty używane do obliczania gradientu. W naszych zadaniach wartość *0.001* zazwyczaj będzie dobrym wyborem. Drugi metaparametr określa, jak daleko od swojego poprzednika zostanie wybrany każdy kolejny punkt testowy.

Sprawdź, co się stanie, gdy *Step* przyjmie różne wartości z zakresu *(0.1, 3)*. Zwróć szczególną uwagę na liczbę kroków potrzebnych do osiągnięcia minimum oraz na szerokość oscylacji, które można zaobserwować w pobliżu minimum.

Jeśli wyniki wyglądają jak na rys. 2, oznacza to, że algorytm działa poprawnie.



Rys 2 – Przykład działania algorytmu gradientowego

Zadanie 1.2: Przygotuj, skonfiguruj (wybierając odpowiednią wartość *Step*) i przetestuj algorytm gradientowy w celu optymalizacji Twojej **indywidualnej funkcji** 2D, która posiada kilka minimów lokalnych (*2D, fewminima*). Uruchom metodę trzykrotnie, zapisując wyniki w Tabeli 1. Po wykonaniu tego kroku zapisz kod jako osobny skrypt.

Ostatnim algorytmem, który dziś zaimplementujemy, jest metoda *1+1*. Ponownie rozpoczniemy od funkcji *of_f_randomSampling*, rozwiązując zadanie *of_2D_oneminimum_2*. Podobnie jak wcześniej, przed rozpoczęciem pętli *while* będziemy potrzebować losowo wygenerowanego rozwiązania.

Dodatkowo konieczne będzie miejsce do przechowywania współrzędnych oraz wartości aktualnie najlepszego wyniku. Zwróć uwagę, że w kodzie mamy już odpowiedni fragment do tego celu:

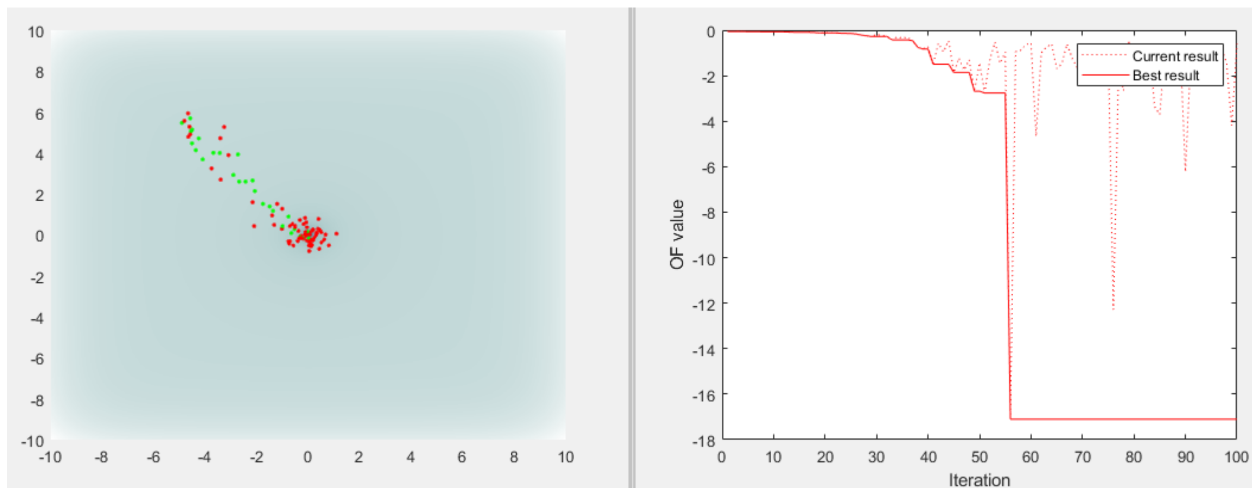
```
CurrentMin = Inf;
Result = [10,10];
```

Podobnie jak w przypadku podstawowego algorytmu losowego, sprawdzimy wartość w punkcie testowym (*CurrentValue = ...*), a następnie porównamy ją z aktualnie najlepszym zapisanym rozwiązaniem i, jeśli zajdzie taka potrzeba, zaktualizujemy je (fragment kodu zaczynający się od *if(CurrentValue < CurrentMin)*). Po tym kroku, niezależnie od tego, czy bieżące rozwiązanie zostało zachowane, czy odrzucone, wygenerujemy nowy punkt do przetestowania w kolejnej iteracji pętli *while*. Nowy punkt zostanie uzyskany poprzez dodanie do najlepszego zapisanego rozwiązania niewielkiej losowej wartości:

```
Point = Result + Step*randn(size(Point));
```

Obecnie mamy jeden metaparametr metody: wartość *Step*. Sprawdź, co się stanie, gdy jego wartość zostanie zmieniona w zakresie od *0.1* do *4*.

Zwróć uwagę na wpływ tego parametru na szybkość zbieżności oraz stabilność algorytmu. Jeśli uzyskane wyniki wyglądają podobnie do tych przedstawionych na rys. 3, oznacza to, że algorytm działa poprawnie.



Rys 3 – Przykład działania metody 1+1

Zadanie 1.3: Przygotuj, skonfiguruj (wybierając odpowiednią wartość *Step*) i przetestuj metodę *1+1* w celu optymalizacji Twojej **indywidualnej funkcji** 2D z kilkoma minimami lokalnymi (*2D, fewminima*). Uruchom metodę trzykrotnie, zapisując wyniki w Tabeli 1. Po wykonaniu tego kroku zapisz kod jako osobny skrypt.

Algorytm gradientowy, wielostartowy

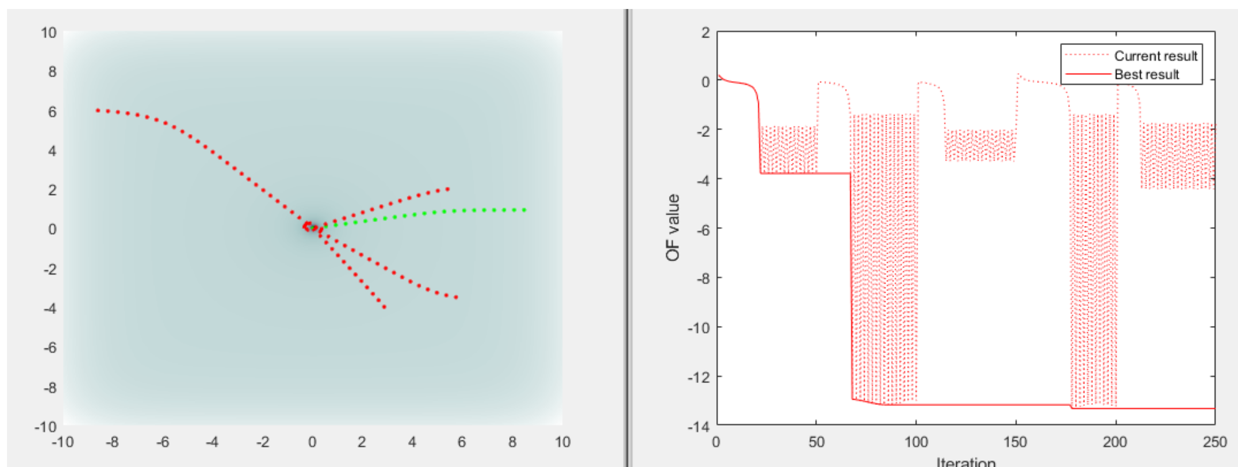
Algorytmy wrażliwe na minima lokalne często wymagają wielu uruchomień z losowych punktów początkowych, przechowywania wyników każdego przebiegu i ostatecznie wyboru najlepszego znalezionego rozwiązania spośród wszystkich wykonanych prób. Przykładem algorytmu, który szczególnie korzysta z takiego podejścia, jest oczywiście algorytm gradientowy. Dodamy tę funkcjonalność, umieszczając naszą wcześniejszą pętlę *while* algorytmu gradientowego wewnątrz następującego fragmentu kodu:

```
for starts = 1:Starts

   ....

end
```

Gdzie *Starts* jest metaparametrem określającym, ile razy metoda zostanie uruchomiona. Zwróć uwagę, że inicjalizacja historycznie najlepszego rozwiązania powinna być wykonana poza tą pętlą (czyli *Result* i *CurrentMin* nie powinny być resetowane po każdym starcie – w końcu chcemy zachować najlepszą wartość spośród wszystkich prób). Natomiast generowanie punktu początkowego, ustawianie początkowej wartości *Step* i resetowanie *EndingCondition* powinno być wykonywane osobno dla każdego startu.

Jeśli uzyskane wyniki wyglądają podobnie do tych przedstawionych na rys. 4, oznacza to, że algorytm działa poprawnie.

Jeśli masz trudności z rysowaniem krzywych zbieżności lub algorytm nagle zatrzymuje się po pierwszym starcie, zwróć uwagę na fakt, że dotychczas miałeś tylko jedną zmienną licznikową zarówno do zapisywania krzywych zbieżności, jak i sprawdzania warunku zatrzymania. Teraz potrzebujesz dwóch liczników**:** jeden do śledzenia całkowitej liczby iteracji metody (do rysowania krzywych zbieżności) i drugi (np. *iter2*) do śledzenia liczby iteracji dla jednego startu metody (do sprawdzania warunku zakończenia *EndingCondition*), który powinien być resetowany dla każdego nowego startu.



Rys 4 – Przykład działania algorytmu gradientowego, wielostartowego

Zadanie 1.4: Przygotuj, skonfiguruj (wybierając odpowiednią wartość *Step*) i przetestuj algorytm gradientowy z wielokrotnym startem (*multistart gradient algorithm*) do optymalizacji Twojej **indywidualnej funkcji** 2D z kilkoma minimami lokalnymi (*2D, fewminima*).

Algorytm powinien wykonać **około 400** obliczeń funkcji celu, np. **5 startów x 26 iteracji**. Uruchom metodę trzykrotnie, zapisując wyniki w Tabeli 1. Po wykonaniu tego kroku zapisz kod jako osobny skrypt.

Uwaga: 5 startów x 26 iteracji daje w rzeczywistości 390 obliczeń funkcji celu a nie 130. Czy potrafisz wyjaśnić dlaczego?

Adaptacyjny krok w algorytmach optymalizacyjnych

Jeśli krok optymalizacji jest zbyt duży, algorytm ma tendencję do bardzo wolnej poprawy lub nawet całkowitego braku poprawy. Z tego powodu warto zmniejszyć rozmiar kroku, jeśli przez pewną liczbę generacji nie obserwujemy poprawy. Policzmy nieudane iteracje naszego rozwiązania gradientowego:

```
if(CurrentValue < CurrentMin)
     % ...
     % Here is our piece of code for storing best results, etc…
     % ...
     NoImprove = 0;
else
     % FunctionPlot (red, if we don't have a new minimum):
     if(PointPlot == 1)
          figure(1);   plot3(Point(1),Point(2), CurrentValue,'.r'); hold on
     end
     NoImprove = NoImprove + 1;

end
```

Oczywiście przed rozpoczęciem pętli musimy zainicjalizować licznik *NoImprove*. Teraz możemy także zmniejszyć wartość *Step*, jeśli zauważymy, że nie udało się osiągnąć poprawy np. 3 razy z rzędu:

```
        if(NoImprove > 2)  % The '2' is the adaptation trigger here
            NoImprove = 0;
            Step = Step * 0.6;  % The '6' is the reduction factor here
        end
```

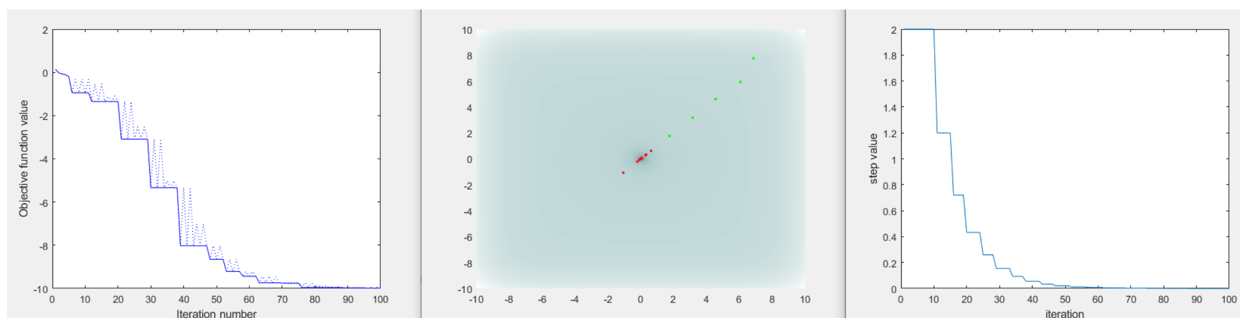
Zwróć uwagę na wyróżnione wartości. Na razie pozostawiamy je w kodzie w takiej postaci – jednak są to metaparametry i będziemy musieli później określić ich wartości. Dobrze byłoby również śledzić, jak zmienia się nasza wartość *Step*:

```
StepHistory(iter) = Step;
```

Tę historię możemy później wyświetlić i zobaczyć, jak algorytm dostosowywał się do optymalizowanej funkcji:

```
figure(4);
plot(StepHistory)
xlabel('iteration');
ylabel('step value');
```

Po uruchomieniu tego kodu powinieneś zobaczyć coś podobnego do rys. 5. Zauważ, że dzięki adaptacyjnemu krokowi możemy łatwo rozpocząć od większej początkowej wartości *Step* niż wcześniej i nie napotkamy problemu oscylacji w pobliżu minimum lokalnego!



Rys 5 – Krzywa zbieżności, wykres punktowy i historia zmian kroku optymalizacji

Zadanie 1.5: Przygotuj algorytm gradientowy wielostartowy oraz podstawowy algorytm gradientowy, wyposaż je w adaptacyjny krok i przetestuj je w optymalizacji Twojej **indywidualnej funkcji** 2D z kilkoma minimami lokalnymi (*2D, fewminima*).
Przetestuj każde rozwiązanie trzykrotnie i uzupełnij odpowiednie wiersze w Tabeli 1. Następnie zapisz kod jako osobny skrypt.

Po tym etapie przygotuj również adaptacyjne rozwiązanie dla metody *1+1* i sprawdź jego wydajność.

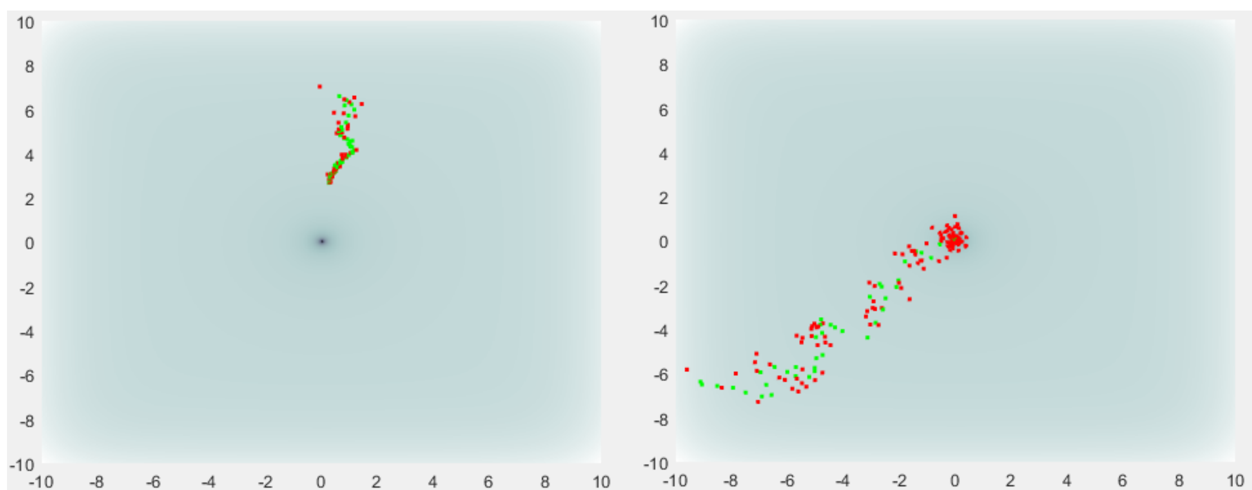
Note 1: Jeśli zauważysz, że Twoje rozwiązanie 1+1 "zatrzymuje się zbyt wcześnie", nie panikuj. Na tym etapie jest to oczekiwane.

Note 2: Jeśli zauważysz, że Twoje rozwiązanie gradientowe nie zmniejsza wartości kroku tak, jak się spodziewałeś, również nie panikuj – ale spróbuj wyjaśnić, dlaczego tak się dzieje i co można zrobić, aby temu zapobiec?

Konfiguracja metaparametrów rozwiązania adaptacyjnego

Podczas wykonywania zadania 1.5 prawdopodobnie napotkałeś istotne problemy z obiema metodami: optymalizacja metodą *1+1* zatrzymała się zbyt wcześnie, a w rozwiązaniu opartym na gradientach krok nie zmniejszał się wystarczająco. Jest to spowodowane tym, że „niepowodzenia” powinny być traktowane inaczej w zależności od zastosowanego algorytmu i rozwiązywanego problemu optymalizacyjnego. Pamiętasz metaparametry wyróżnione we wcześniejszym kodzie? Przyjęliśmy, że 3 niepowodzenia wystarczą do zmniejszenia kroku – jednak rozwiązanie gradientowe powinno poprawiać się w każdej iteracji, podczas gdy metoda *1+1* w najlepszym przypadku osiąga sukces w 50% prób.
Im więcej wymiarów ma problem i im bliżej jesteśmy minimum, tym mniejsze jest prawdopodobieństwo poprawy z iteracji na iterację. Z tego powodu próg powinien być **wyższy** dla metody *1+1* i **niższy** dla rozwiązania gradientowego.

Jeśli ustawię próg na **9** dla metody *1+1*, metoda konsekwentnie osiąga optimum dla mojej funkcji (zob. rys. 6). Jeśli ustawię próg na **> 0** dla rozwiązania opartego na gradientach, powinno ono również działać znacznie lepiej.

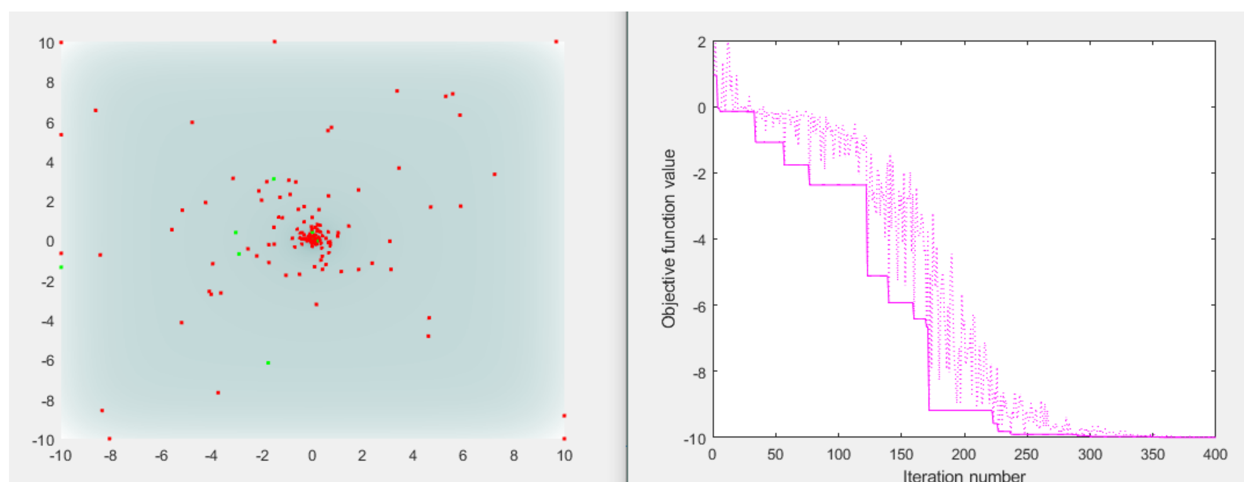


Rys 6 – Rezultaty dla zbyt małego progu do redukcji kroku adaptacji (z lewej: NoImprove > 2) i poprawnego kroku (z prawej, NoImprove > 8)

Kolejnym problemem, który musimy rozwiązać, jest współczynnik redukcji kroku (*reduction coefficient*). Parametr ten musi być dostosowany zarówno do konkretnej metody, jak i do konkretnego problemu. Celem jest zmniejszanie kroku wystarczająco wolno, aby umożliwić płynne przejście od eksploracji do eksploatacji, ale jednocześnie wystarczająco szybko, aby na końcu zapewnić dobrą eksploatację. Warto również pamiętać, że możemy pracować z wartością początkowego kroku – określając poziom, od którego rozpoczynamy adaptację.
Naszym celem jest uzyskanie przebiegu podobnego do przedstawionego na rys. 7:

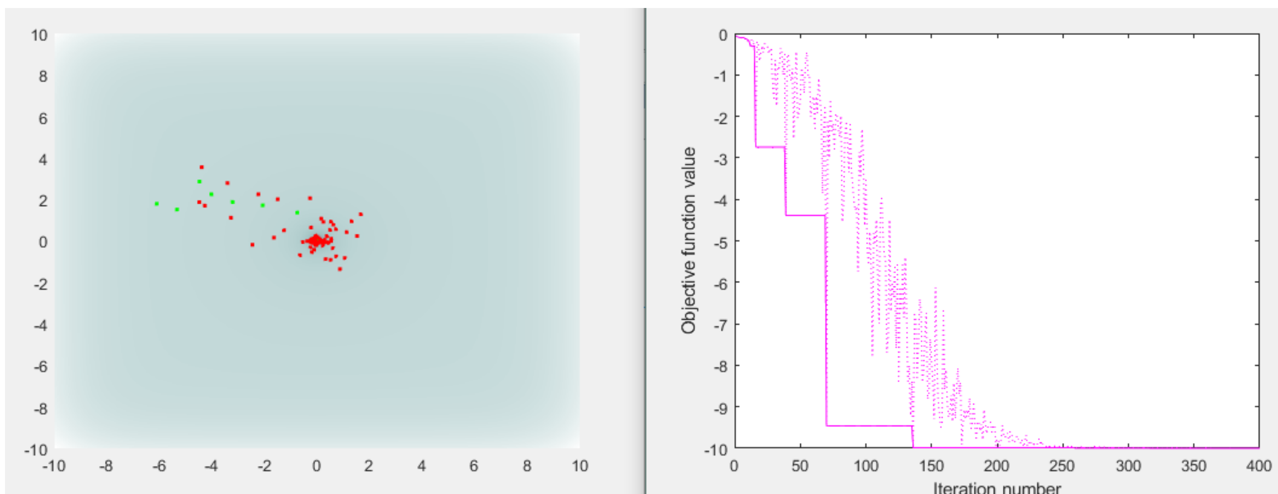
- **Krok początkowy jest na tyle duży,** że umożliwia dobrą eksplorację (punkty początkowo rozłożone w całym zakresie przeszukiwania).
- **Przejście od eksploracji do eksploatacji jest stopniowe i płynne.**
- **Krok oraz zmienność rozwiązań dążą do zera na końcu procesu.**

Przykładowe problemy konfiguracyjne przedstawiono na rys. 8.

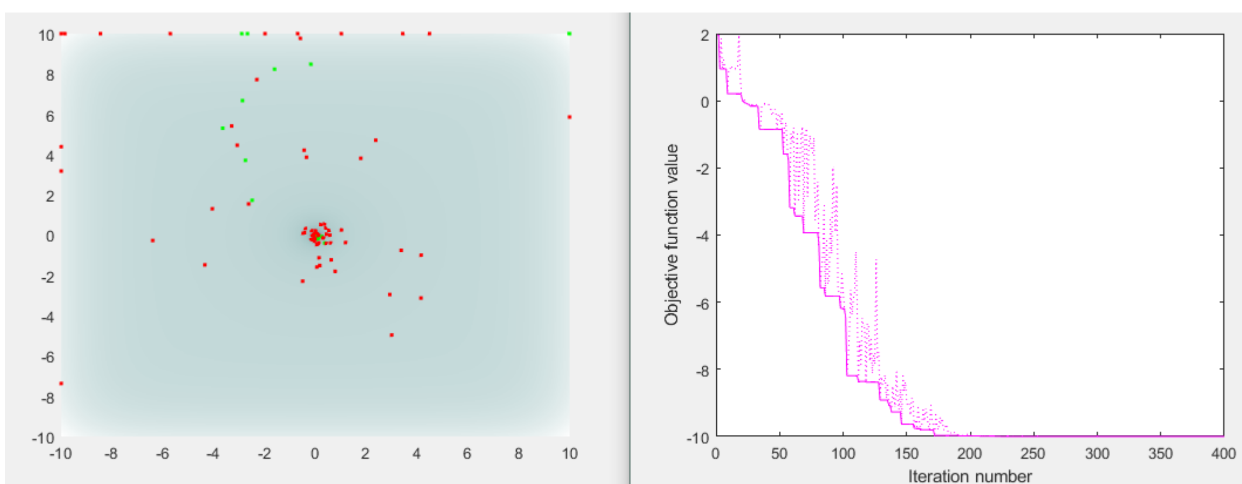


Rys 7 – Poprawna konfiguracja rozwiązania 1+1

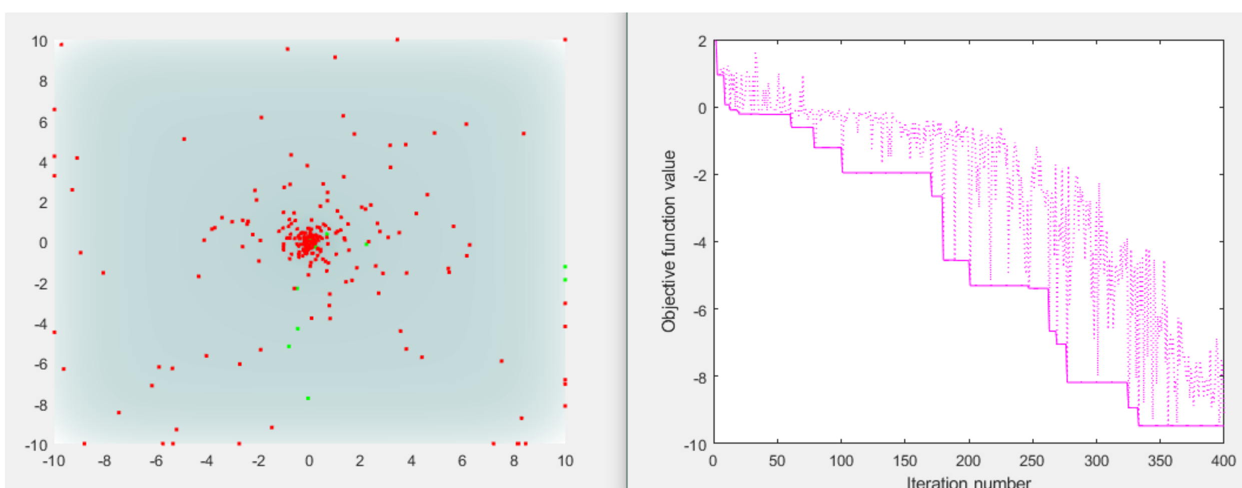
Zadanie 1.6: Please provide your 1+1 algorithm with adaptive step, configure metaparameters (Initial Step, Adaptation trigger and Reduction factor) for optimization of your **individual*** 2D function that has few local minima (**2D, fewminima** type). The method should use 400 objective function checks (i.e. 400 iterations). Consult Fig 8 for common mistakes. Run the configured solution 3 times and save the results in Table 1. Please save the code as a separate script.



(a) – Za mały krok początkowy – wyłącznie niewielki zakres przestrzeni jest testowany



(b) – Za duży współczynnik redukcji – zmiana z eksploracji do eksploatacji jest zbyt gwałtowna – koniec optymalizacji zawiera "martwe iteracje" – algorytm nie poprawia swojego działania od 200 iteracji.



(c) – Zbyt mały współczynnik redukcji – zmiana z eksploracji na eksploatację jest zbyt wolna – algorytm nie eksploatuje poprawnie minimum (zmienność na końcu jest za duża)

Rys 8 – Przykłady błędów w konfiguracji algorytmu 1+1

Uczciwe i obiektywne porównywanie metod optymalizacyjnych

Porównanie różnych metod powinno zawsze być rzetelne i nie powinno faworyzować żadnego rozwiązania. Z tego powodu wszystkie metody optymalizacji powinny mieć taką samą liczbę obliczeń funkcji celu – ponieważ to właśnie liczba tych obliczeń jest zwykle głównym ograniczeniem obliczeniowym.
Oznacza to, że jeśli algorytm gradientowy ma 500 iteracji, a algorytm gradientowy z wielokrotnym startem (*multistart gradient algorithm*) wykonuje 10 startów, to każdy z tych startów powinien mieć tylko 50 iteracji, aby był porównywalny z metodą podstawową.

Jeśli później do porównania dodamy metodę *1+1*, to powinna ona wykonać 1500 iteracji (zakładając porównanie w przestrzeni 2D). W przestrzeni 2D dla każdego kroku metody gradientowej funkcja celu musi być obliczona trzykrotnie, więc dla 1 iteracji metody gradientowej metoda *1+1* powinna mieć pozwolenie na trzy kroki.

Powtarzalność rozwiązań

Algorytmy optymalizacji zazwyczaj działają w sposób niedeterministyczny – każdy przebieg może dawać nieco inny wynik. Z tego powodu konieczna jest statystyczna analiza powtarzalności metody oraz ocena prawdopodobieństwa trafienia w minimum lokalne.
Aby szybko uzyskać statystyki z wielu uruchomień:

- Warto wyłączyć wizualizację punktów oraz wartości funkcji celu.
- Dobrze jest opakować główny kod w pętlę *for*, która wykona tyle powtórzeń, ile potrzebujemy do uzyskania statystyk.
- Generowanie punktu startowego oraz reinicjalizacja najlepszego wyniku powinny być umieszczone wewnątrz tej pętli:

```
for repetition = 1:10

%% Reinitialization of a starting point and ending condition
    CurrentMin = Inf;
    Result = [0,0];
    EndingCondition = 0;
    iter = 0;

    while(EndingCondition == 0);
       % Here we have a main optimization loop
       ...


    end
    % Here we store best result from each run:
    Results(repetition) = BestHistory(end)

end
% Here we store statistics from our run:
mean(Results)
std(Results)
```

Zadanie 1.7: Skonfiguruj i przetestuj swój algorytm *1+1* z adaptacyjnym krokiem, algorytm gradientowy z pojedynczym startem oraz algorytm gradientowy z wielokrotnym startem (*multistart gradient algorithm*) z adaptacyjnym krokiem w celu optymalizacji Twojej indywidualnej funkcji 2D z wieloma minimami lokalnymi (*2D, manyminima*).

Skonfiguruj parametry adaptacji w taki sposób, aby metody konsekwentnie eksploatowały minimum, do którego zostały przyciągnięte. Użyj 800 obliczeń funkcji celu dla każdej metody. Przetestuj metody statystycznie, wykonując 20 powtórzeń każdej metody, oblicz średnie wyniki i zapisz je w Tabeli 1.

Zapisz wszystkie kody do dalszego wykorzystania.

Rozbudowa algorytmów do optymalizacji problemów wielowymiarowych

W praktycznych sytuacjach rzadko optymalizujemy problemy, które mają tylko dwa parametry do ustawienia. Teraz nadszedł czas, aby rozszerzyć nasze rozwiązania na problemy wielowymiarowe. Na szczęście metody są już w większości przygotowane do pracy w przestrzeniach wielowymiarowych. Wystarczy:

- Wyłączyć wizualizację mapy (*FunctionPlot = 0*, *PointPlot = 0*), ponieważ wyświetlanie jedynie przekroju naszej przestrzeni wielowymiarowej nie ma większego sensu.
- Dostosować liczbę wymiarów oraz macierz *Range* odpowiednio do nowego problemu.
- Wygenerować punkt początkowy dla Twojej metody o właściwym rozmiarze – tzn. zawierający tyle współrzędnych ile wymiarów ma problem optymalizacyjny.

Zadanie 1.8: Skonfiguruj i przetestuj swój:
- Algorytm *1+1* z adaptacyjnym krokiem,
- Algorytm gradientowy z pojedynczym startem i adaptacyjnym krokiem,
- Algorytm gradientowy z wielokrotnym startem i adaptacyjnym krokiem,
w celu optymalizacji Twojej **indywidualnej funkcji wielowymiarowej**.
Użyj 1200 obliczeń funkcji celu dla każdej metody. Przetestuj metody statystycznie, wykonując 20 powtórzeń każdej metody, oblicz średnie wyniki i zapisz je w Tabeli 1.
Zapisz wszystkie kody do dalszego wykorzystania.

Tabela 1: Zagregowane wyniki algorytmów optymalizacyjnych

	2D Fewminima			2D Manyminima, Statystyka z 20 startów		Multidimensional, Statystyka z 20 startów	
	Test1	**Test2**	**Test3**	**mean**	**std**	**mean**	**std**
Grid search				-	-	-	-
1+1				-	-	-	-
Gradient				-	-	-	-
Gradient wielostartowy				-	-	-	-
Gradient, adaptacyjny krok							
Gradient wielostartowy, adaptacyjny krok							
1+1, adaptacyjny krok							

Additional tasks:

Zadanie 1.9: Spróbuj zoptymalizować metaparametry opracowanych rozwiązań podczas rozwiązywania Twojego indywidualnego zadania **typu 2D, fewminima**. (Jest to zadanie optymalizacji drugiego rzędu, w którym optymalizujemy parametry algorytmu optymalizacji). Czy możemy tutaj użyć metody przeszukiwania siatkowego (*grid search*)?

***Podpowiedź**: Testowane algorytmy są niedeterministyczne. Jest to tutaj istotny czynnik! Jeśli chcesz użyć którejkolwiek z poznanych metod optymalizacji, każde sprawdzenie funkcji celu (OF check) zwróci wartość pochodzącą z pewnego rozkładu statystycznego. Aby uzyskać lepszą wiedzę na temat „jakości", którą należy przypisać konkretnym wartościom metaparametrów, powinieneś uśrednić wyniki wielu testów.*

Zadanie 1.10:
Przygotuj statystyczny raport dotyczący wyników dobrze skonfigurowanego algorytmu *1+1* oraz dobrze skonfigurowanego algorytmu gradientowego z wielokrotnym startem (*multistart gradient algorithm*) w rozwiązaniu Twojego indywidualnego zadania typu 2D, fewminima. Sprawdź rozkład wyników dla obu metod. Wykorzystaj te dane do ewentualnej poprawy wartości metaparametrów. Na koniec sprawdź, czy wprowadzona poprawa jest statystycznie istotna.

Zadanie 1.11: Zaimplementuj algorytm gradientowego spadku z momentem (*gradient descent with momentum*). Przetestuj go w rozwiązaniu Twojego indywidualnego zadania typu 2D, fewminima. Przetestuj go również w rozwiązaniu funkcji *of_2D_fewminima_5*. Czy moment pomaga pod względem szybkości zbieżności? Czy poprawia dokładność wyników?

Zadanie 1.12: Przetestuj algorytm gradientowy oraz algorytm *1+1* w zadaniu, w którym do obliczeń funkcji celu dodany jest szum (wyniki dwóch kolejnych testów w tym samym punkcie mogą się różnić). Użyj funkcji *of_2D_oneminimum_1_rand*. Czy algorytm gradientowy działa w tych warunkach? Co powinniśmy zrobić, aby umożliwić mu konkurowanie z metodą *1+1*?

Podpowiedź: Masz już wszystkie metaparametry, które należy zmienić w tym przypadku – nie musisz modyfikować kodu. Zmiana jednego z nich powinna pomóc.