Wydział Inżynierii Mechanicznej i Robotyki

Katedra Robotyki i Mechatroniki

Przetwarzanie sygnałów i identyfikacja w sterowaniu urządzeń mechatronicznych,

Przetwarzanie sygnałów i identyfikacja w monitorowaniu urządzeń mechatronicznych

Temat: Klasyczne algorytmy optymalizacyjne

Cel ćwiczenia: Implementacja klasycznych algorytmów optymalizacyjnych w zagadnieniu znajdowania minimum lokalnego dwuwymiarowej funkcji celu

Zagadnienia: Algorytm losowy, grid search, gradientowy, 1+1

Prowadzący: Ziemowit Dworakowski, zdw@agh.edu.pl, http://galaxy.agh.edu.pl/~zdw/

Informacje wstępne
Dotyczące wszystkich zajęć prowadzonych przez dr. Dworakowskiego

Podstawą do realizacji zadań jest dostarczony przez prowadzącego zbiór funkcji:

- *op_f_RandomSampling.m* (funkcja realizująca optymalizację losową funkcji 2 zmiennych z wizualizacją działania oraz krzywą konwergencji)

- Rodzina funkcji 2D zawartych w folderze *FunctionsForOptimization* zawierających jedno minimum lokalne, kilka minimów lokalnych oraz dużo minimów lokalnych.

Poszczególne programy przygotowywać należy jako osobne pliki i zachować na kolejne zajęcia - bowiem będą niezbędne do porównania z algorytmem genetycznym.

Wszystkie modyfikowalne parametry kodu (ilość iteracji, stałe, wartości kroku, ilości startów algorytmów, zakresy losowania wartości) należy umieszczać na początku kodu i wyraźnie opisywać.

Podstawą do realizacji ćwiczeń jest wiedza z wykładu (na wykładach znajdziecie Państwo komplet wiedzy niezbędnej do zrozumienia zadań oraz zaplanowania sposobu ich wykonywania)

W instrukcji znajdują się "Zadania na 3.0" - oznaczone kolorem czerwonym, "Zadania na 4.0" oznaczone kolorem pomarańczowym oraz "Zadania na 5.0" oznaczone kolorem zielonym. Wykonanie zadań na daną ocenę oznacza uzyskanie oceny warunkowej (np. zrobienie wszystkich zadań na 3.0 oraz na 4.0 skutkuje uzyskaniem warunkowego "4.0" - o ile na kolejnych zajęciach student zda i obroni komplet podstawowych zadań z instrukcji (do 5.0 włącznie).

W przypadku jednokrotnej nieobecności lub jednokrotnego braku uzyskania oceny 3.0 na laboratorium student powinien rozwiązać komplet zadań podstawowych z problematycznej instrukcji oraz jedno wskazane przez prowadzącego zadanie rozszerzające (w kolorze niebieskim) - i zdać rezultaty w formie raportu.

Przypadki nieobjęte powyższymi zasadami (nieobecność na wielu zajęciach, Brak uzyskania oceny 3.0 na wielu zajęciach) traktowane będą indywidualnie i wymagać będą kontaktu z prowadzącym w celu ustalenia wymaganego zakresu wiedzy i formy zaliczenia laboratorium.

Każde spotkanie zaczynać się będzie od krótkiego kolokwium wejściowego obejmującego materiał z wykładów. Wszystkie testy muszą być zaliczone aby można było uzyskać zaliczenie z całości przedmiotu. Do każdego testu będzie zorganizowany jeden termin poprawkowy (zwykle podczas następnych zajęć). Termin poprawkowy będzie trudniejszy niż pierwszy termin testu.

Zakres wiedzy wymaganej na ćwiczeniach:

- Znajomość pojęć związane z optymalizacją (optymalizacja, funkcja celu, minimum lokalne lub globalne, krzywa konwergencji, zasada działania algorytmu losowego, grid search, gradientowego, gradientowego wielostartowego, algorytmu 1+1)

Własna funkcja

W części zadań do wykonania ćwiczenia niezbędne będzie rozwiązanie problemu dla ściśle określonej funkcji - innej niż funkcje wykorzystywane przez pozostałe osoby w grupie. Taka funkcja nazywana jest w tej instrukcji **własną** funkcją. Jeśli prowadzący na zajęciach nie określi inaczej, numer domyślnej "własnej" funkcji powstaje przez określenie reszty z dzielenia sumy ilości liter nazwiska i imienia przez 8.

Zadanie wstępne: uruchomienie i przetestowanie podstawowego kodu

Proszę uruchomić kod `of_f_randomSampling` dostępny w przekazanym przez prowadzącego archiwum oraz w załączniku do niniejszej instrukcji. Kod ten służy do optymalizacji funkcji 2 zmiennych. Funkcja dana jest jako "***of_2D_oneminimum_8'"*** (dwuwymiarowa, posiadająca jedno minimum). Po uruchomieniu kodu zobaczymy wykres funkcji oraz "punkty próbne" powstające jako efekt działania algorytmu przeszukiwania losowego.

Proszę przejrzeć kod zwracając uwagę na komentarze oraz zrozumieć jego zasadę działania. Kluczowe z punktu widzenia dalszych ćwiczeń jest zrozumienie w jaki sposób współrzędne punktu próbnego są generowane oraz testowane: zawartość funkcji ***while*** powinna być na tym etapie dobrze i szczegółowo zrozumiana.

Zakończenie działania kodu powoduje wygenerowanie krzywej konwergencji pokazującej historię poszukiwania minimum krzywa składa się z dwóch komponentów: aktualny test (wartość uzyskana w konkretnej iteracji algorytmu) oraz historyczne minimum (najlepsza wartość odnaleziona przez algorytm w czasie swojego działania).

Proszę uruchomić i przetestować kod dla kilku wybranych funkcji. Proszę sprawdzić "wygląd" i "poziom trudności" zadania optymalizacji dla różnych funkcji z kategorii "Manyminima" oraz "Fewminima". Na tym etapie NIE korzystamy jeszcze z funkcji mających w nazwie "adaptive", funkcji mających w nazwie "rand" oraz funkcji więcej niż dwóch zmiennych.

Podstawowe algorytmy optymalizacyjne

Przejdźmy następnie ponownie do zadania optymalizacji funkcji o jednym minimum (**'of_2D_oneminimum_2')** i w oparciu o tą funkcję przygotujmy algorytm *grid search*.

Algorytm będzie musiał sprawdzić wszystkie punkty w węzłach siatki rozpiętej na obu parametrach, najprościej więc będzie zastąpić pętlę **while** zagnieżdżoną instrukcją **for:**

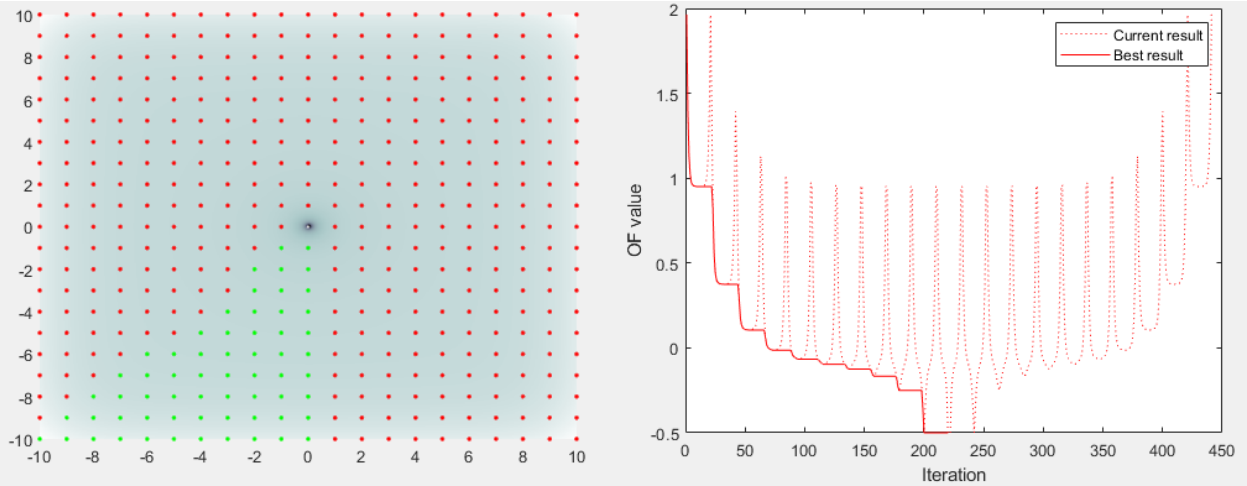
```
for NewX = MaxRangeX(1):step_x:MaxRangeX(2)
    for NewY = MaxRangeY(1):step_x:MaxRangeY(2)

        ...
    end
end
```

tutaj **step_x** oraz **step_y** to oczywiście rozmiar "oczka siatki" a rozmiar obszaru przeszukiwania definiowany jest przez współrzędne zawarte w wektorach **MaxRangeX** oraz **MaxRangeY.**

Oczywiście skoro generujemy wartości *NewX* i *NewY* na początku pętli nie będziemy musieli już ich losować wewnątrz.

Jeśli uzyskany obraz wygląda podobnie do przedstawionego na rys. 1, to algorytm działa dobrze



Rys 1 - przykład działania algorytmu Grid Search

Zadanie 1.1: Proszę przygotować, uruchomić, skonfigurować i przetestować algorytm *grid search* do optymalizacji **własnej*** funkcji 2 zmiennych posiadającej kilka minimów lokalnych (typ **2D fewminima**) - po wykonaniu zadania proszę je zapisać jako osobny program.

Następnie wykonamy zadanie optymalizacji z wykorzystaniem metody gradientowej. Punktem wyjścia będzie ponownie funkcja *of_f_randomSampling*. Tym razem współrzędne kolejnego punktu próbnego nie będą wyznaczane losowo, ale będą powstawały w wyniku przesunięcia się o krok w kierunku największego spadku gradientu.

Będziemy chcieli zacząć optymalizację od punktu losowego, wygenerowanego np. tak:

```
NewX = InitialRangeX(1) +  rand()*(InitialRangeX(2) - InitialRangeX(1));
NewY = InitialRangeY(1) +  rand()*(InitialRangeY(2) - InitialRangeY(1));
```

a następnie przejść do pętli **while** i wewnątrz tej funkcji, w każdym jej obiegu wyznaczyć gradient w zdefiniowanym punkcie oraz zaproponować nowy punkt do sprawdzenia w kolejnej iteracji. Aby wyznaczyć gradient, będziemy potrzebowali wyznaczenia wartości funkcji celu w punkcie próbnym oraz w dwóch punktach przesuniętych wzdłuż osi X i Y o niewielką wartość:

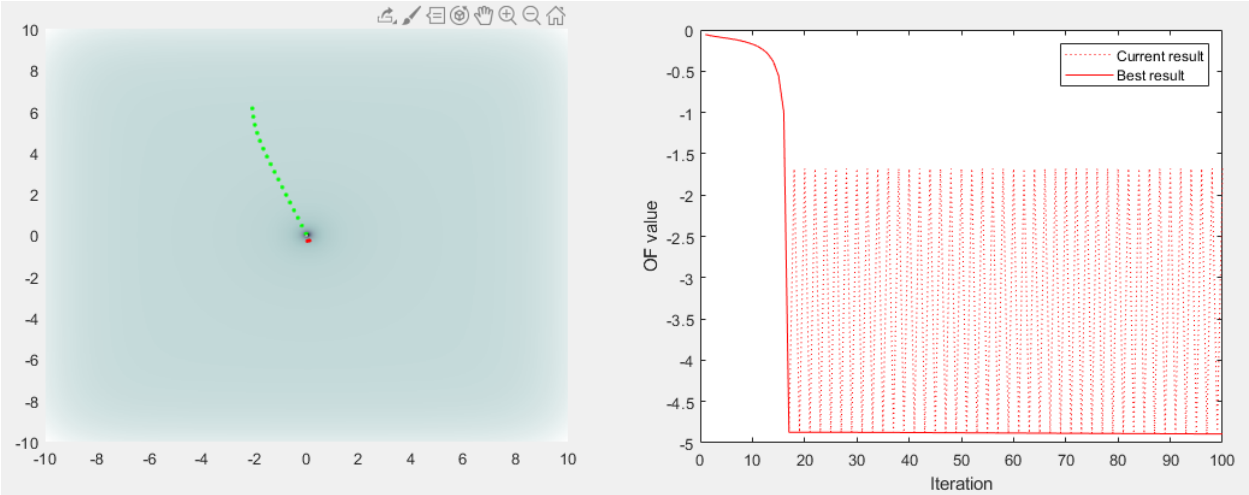
```
CurrentValue =  FunctionForOptimization([NewX,NewY]);
CV_dx =  FunctionForOptimization([NewX+g_step,NewY]);
CV_dy =  FunctionForOptimization([NewX,NewY+g_step]);
```

Następnie kolejny punkt do sprawdzenia należy wyznaczyć poprzez przemnożenie współrzędnych poprzedniego punktu przez znormalizowany gradient przemnożony przez stałą **Step**:

```
CV = CurrentValue;  % Aby kod zmieścił się na stronie instrukcji ;)
NewX = NewX + Step*(CV-CV_dx)/sqrt((CV-CV_dx)^2 + (CV-CV_dy)^2)
NewY = NewY + Step*(CV-CV_dy)/sqrt((CV-CV_dx)^2 + (CV-CV_dy)^2)
```

Proszę zwrócić uwagę na znajdujące się w kodzie metaparametry **step_g** oraz **Step**. Pierwszy z nich odpowiada za krok wyznaczania gradientu (na potrzeby niniejszych zadań zazwyczaj wartość 0.01 będzie odpowiednia). Drugi z nich określa jak daleko w stronę malejącego gradientu funkcja "przeskoczy". Proszę sprawdzić co się dzieje przy zmianach wartości tego parametru (np. w zakresie 0.1 do 3) - w szczególności w kontekście szybkości poprawy rezultatów oraz pojawiających się oscylacji po odnalezieniu okolicy minimum lokalnego.

Jeśli uzyskany obraz wygląda podobnie do przedstawionego na rys. 2, to algorytm działa dobrze



Rys 2 - przykład działania algorytmu gradientowego

Zadanie 1.2: Proszę przygotować, uruchomić, skonfigurować i przetestować algorytm gradientowy do optymalizacji **własnej*** funkcji (typ **2D fewminima**) - po wykonaniu zadania proszę je zapisać jako osobny program.

Ostatnim z algorytmów do zaimplementowania w ramach niniejszego ćwiczenia jest algorytm 1+1. Aby go zastosować ponownie zaczniemy od funkcji *of_f_randomSampling* działającej na zadaniu **'of_2D_oneminimum_2'** i ponownie przed uruchomieniem pętli **while** potrzebować będziemy losowo wygenerowanego punktu. Będziemy również potrzebować punktu do przechowywania aktualnie najlepszego rozwiązania. W tym celu można wykorzystać znajdujący się już w kodzie fragment:

```
%% Storing of a best solution

    CurrentMin = 50000;
    ResultX = 1;
    ResultY = 1;
```

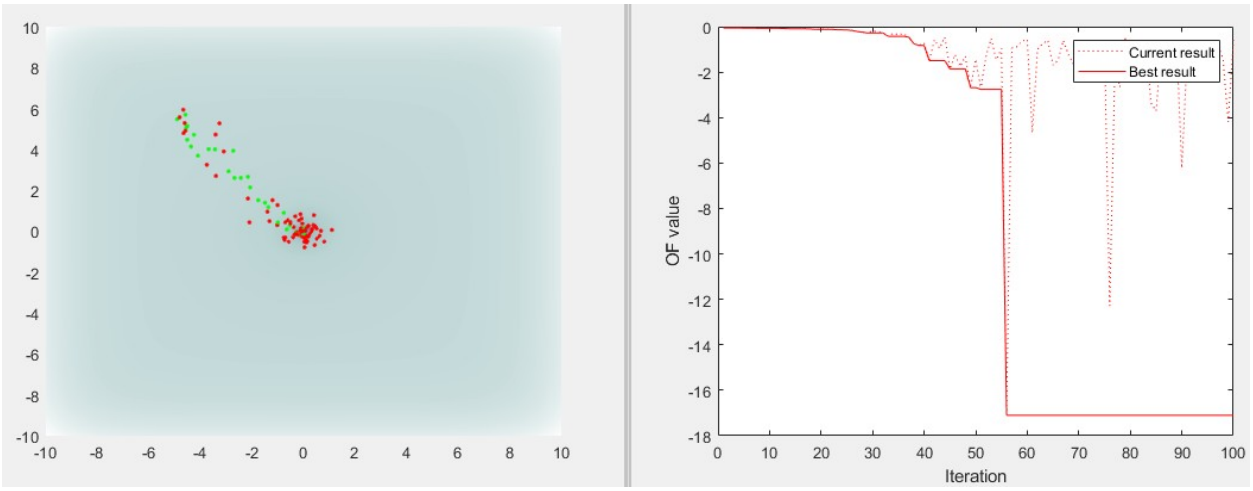
I dokładnie tak jak w przypadku podstawowego algorytmu losowego, sprawdzać będziemy wartość funkcji w punkcie próbnym (**CurrentValue = ...**) a następnie sprawdzać i ew. nadpisywać aktualnie najlepsze znalezione rozwiązanie (blok kodu zaczynający się od **if(CurrentValue < CurrentMin)**).

Po zakończeniu tego bloku, gdy aktualne rozwiązanie zostało odrzucone lub zapisane do "CurrentMin" generujemy nowy punkt w oparciu o historycznie najlepsze znalezione rozwiązanie:

NewX = ResultX+Step*randn();
NewY = ResultY+Step*randn();

Ono będzie sprawdzane oczywiście dopiero w kolejnej iteracji pętli. W kodzie znajduje się jeden metaparametr **Step**. Proszę sprawdzić co się dzieje przy zmianie jego wartości (np. w zakresie 0.1 do 4)

Jeśli uzyskany obraz wygląda podobnie do przedstawionego na rys. 3, to algorytm działa dobrze



Rys 3 - przykład działania algorytmu 1+1

Zadanie 1.3: Proszę przygotować, uruchomić, skonfigurować i przetestować algorytm 1+1 do optymalizacji **własnej*** funkcji (typ **2D fewminima**) - po wykonaniu zadania proszę je zapisać jako osobny program.

Wielostartowość

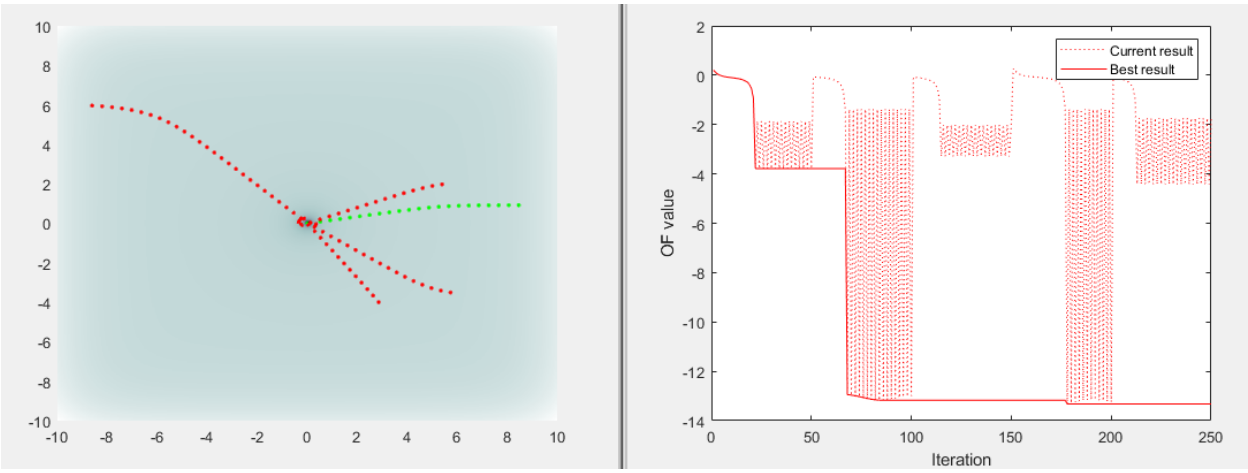
Algorytmy wrażliwe na minima lokalne wymagają często wielokrotnego uruchamiania, zapamiętywania wyników a następnie zwracania wyniku najlepszego w całej serii uruchomień. Przykładem algorytmu nadającego się bardzo dobrze do takiego podejścia jest algorytm gradientowy. Zaimplementujmy algorytm gradientowy wielostartowy poprzez objęcie zastosowanej pętli **while** za pomocą pętli **for** zarządzającej kolejnymi startami:

for starts = 1:Starts

end

gdzie **Starts** to metaparametr określający ilość startów. Należy pamiętać o tym, że przechowywanie historycznie najlepszego wyniku powinno się odbywać na zewnątrz tej pętli (tj. wartości **ResultX, ResultY** oraz **CurrentMin** nie powinny być resetowane wewnątrz pętli for). Jeśli algorytm gradientowy wielostartowy po uruchomieniu i przetestowaniu na funkcji **'of_2D_oneminimum_2'** wygląda jak na rys. 4, to prawdopodobnie działa dobrze. Jeśli masz problem z uzyskaniem podobnego wyglądu krzywej konwergencji lub Twój algorytm zatrzymuje się po pierwszym starcie i nie rusza dalej - najprawdopodobniej problem wynika z iteratora *iter*

którego używaliśmy poprzednio do rysowania krzywych konwergencji oraz sprawdzania warunku stopu pętli. Tym razem potrzebować będziemy **dwóch** iteratorów - jednego, który zadba o warunek stopu i będzie resetowany przy każdym starcie oraz drugiego, który będzie śledził sumę wykonań pełnego algorytmu - i jego użyjemy do wygenerowania krzywych konwergencji.



Rys 4 - przykład działania algorytmu gradientowego, wielostartowego

Zadanie 1.4: Proszę przygotować, uruchomić, skonfigurować i przetestować algorytm gradientowy wielostartowy do optymalizacji **własnej*** funkcji (typ **2D fewminima**). Algorytm powinien mieć do dyspozycji 400 sprawdzeń funkcji celu (np. 5 x 26 iteracji) - po wykonaniu zadania proszę je zapisać jako osobny program.

Zmienny krok w algorytmach, eksploracja i eksploatacja

Gdy algorytm optymalizacyjny testuje obszary daleko od minimum, duży krok pozwala zazwyczaj na szybką poprawę rezultatów. Z drugiej strony, w pobliżu minimum lokalnego wyłącznie mały krok algorytmu pozwala na poprawę rezultatów. W związku z tym metaparametr **Step** stosowany w algorytmie gradientowym i 1+1 w powyższych kodach zazwyczaj powinien być zmieniany wraz ze zmianą numeru iteracji w ten sposób, aby na początku był relatywnie duży a na końcu działania algorytmu bardzo mały. Dobrym pomysłem jest zastosowanie funkcji sigmoidalnej do zarządzania zmianami kroku algorytmu (Rys 5)

W naszym przypadku zastosować można następujący fragment kodu wewnątrz pętli *while*:

```
Step(iter) = InitialStep * (1/(1+exp((iter-(MaxSteps/P1))/P2)));
```

Wraz z odpowiednimi metaparametrami:

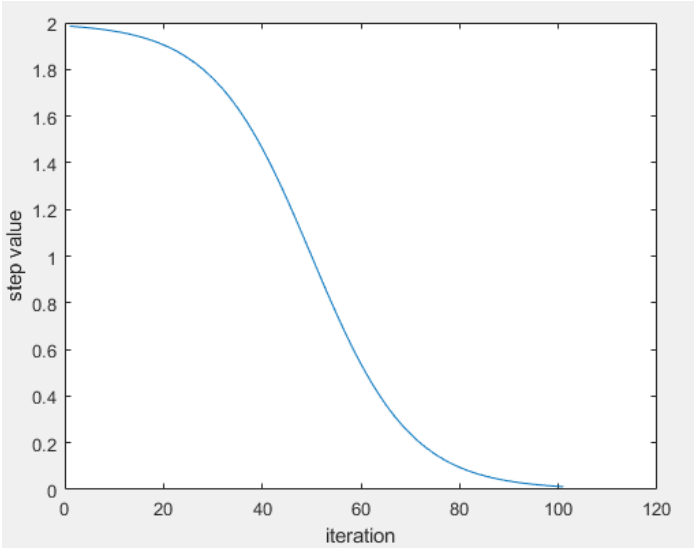
```
InitialStep = 2;      % Exploration/exploitation balance parameters:
P1 = 2;
P2 = 10;
```

Na koniec można wyświetlić wartość *Step* w funkcji iteracji:

```
figure(4);
plot(Step)
```

```
xlabel('iteration');
ylabel('mutation step value');
```

Uwaga! "Step" stał się teraz wektorem przechowującym historyczne wartości kroku. Jeśli korzystałeś wcześniej w kodzie z z pola "**`Step`**", teraz prawdopodobnie musisz to miejsce zmienić na ostatnie pole wektora, czyli "**`Step(iter)`**".



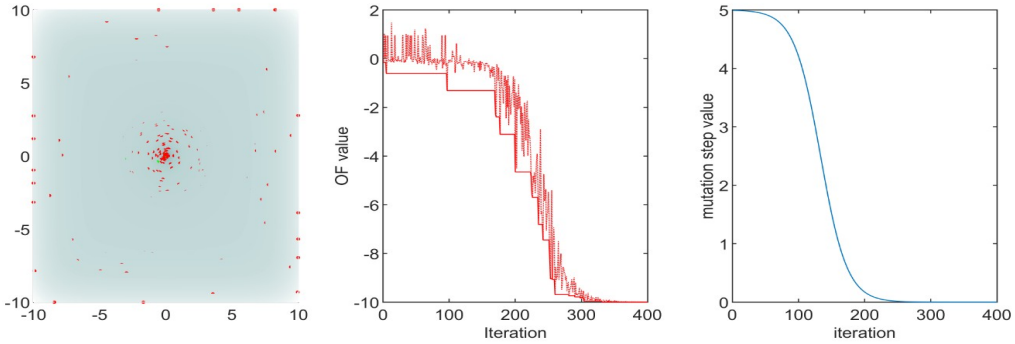
Rys 5 - przykład funkcji sigmoidalnej przystosowanej do zarządzania krokiem algorytmu optymalizacyjnego

Porównywanie algorytmów w uczciwy i obiektywny sposób

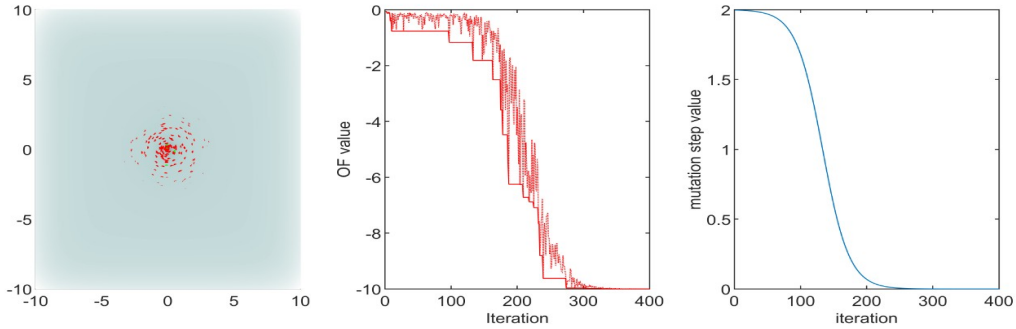
Porównywanie ze sobą dwóch algorytmów optymalizacyjnych powinno być zawsze uczciwe - tzn. **oba algorytmy powinny mieć możliwość sprawdzić funkcję celu taką samą ilość razy**. Oznacza to, że jeśli np. algorytm gradientowy ma 500 iteracji a algorytm gradientowy wielostartowy startuje dziesięciokrotnie, każdy start algorytmu wielostartowego będzie zawierał wyłącznie 50 iteracji! Gdyby później miało nastąpić porównanie tych algorytmów z algorytmem 1+1 ten ostatni otrzymałby 1500 iteracji - bowiem w każdej iteracji algorytm gradientowy wykonuje 3 sprawdzenia funkcji celu (do wyznaczenia gradientu) podczas gdy 1+1 wykonuje tylko jeden test. Gdy później nastąpi porównanie powyższych metod z algorytmem genetycznym działającym na 20-osobniczej populacji, algorytm ten otrzyma 75 generacji (75 x 20 = 1500).

Zadanie 1.5: Proszę wyposażyć algorytm gradientowy wielostartowy w zmienny krok, skonfigurować go i przetestować rozwiązanie w optymalizacji **własnej*** funkcji (typ **2D fewminima**). Proszę porównać otrzymywane wyniki z "podstawową" wersją algorytmu. Po wykonaniu zadania proszę je zapisać jako osobny program. Rozwiązanie należy skonfigurować tak, aby algorytm gradientowy wielostartowy wykonał w sumie ok. 400 sprawdzeń funkcji celu (np. 5 startów x 26 iteracji)

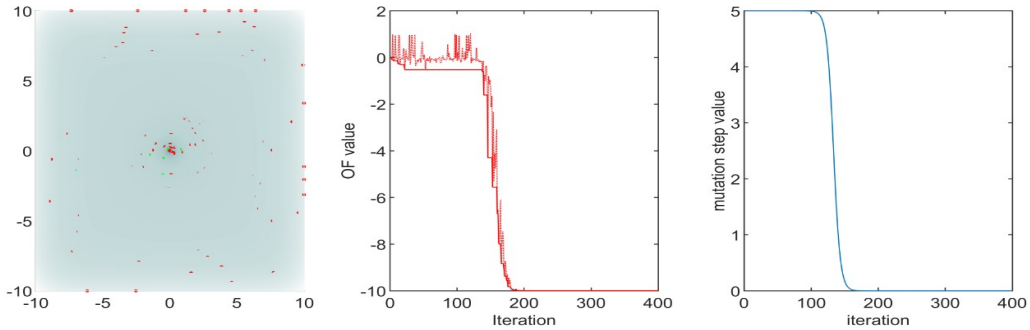
Uwaga! Algorytm gradientowy wielostartowy powinien zmieniać wielkość kroku cyklicznie, w każdym starcie - tzn. każdy start powinien się zaczynać od dużego kroku i kończyć małym).



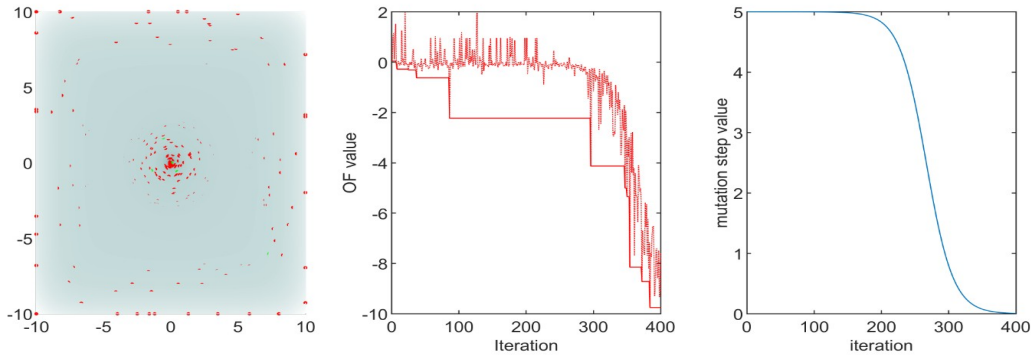
a) Dobra (wzorcowa) konfiguracja



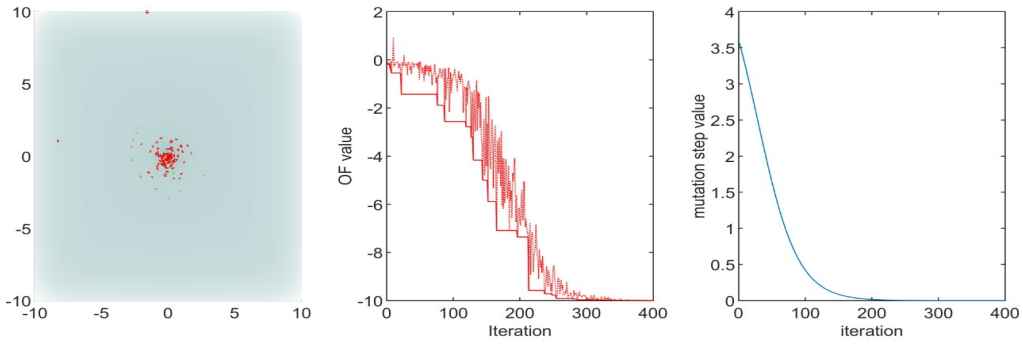
b) Zbyt mały początkowy krok: Nie występuje eksploracja, dla problemu o wielu minimach prawdopodobne jest utknięcie w minimum lokalnym



c) Zbyt gwałtowne przejście od eksploracji do eksploatacji: Ostatnie 150 iteracji nie powoduje już żadnego efektu, krok zmian jest zerowy. Należy "złagodzić" sigmoid.



d) Brak eksploatacji. Do samego końca następuje poprawa rezultatu i występuje duża zmienność (duża różnica między osobnikiem najlepszym a typowym). Należy przesunąć sigmoid w lewo lub go "zaostrzyć".



e) Zbyt krótka eksploracja: algorytm od razu przechodzi do pogłębiania pierwszego znalezionego minimum nie dając sobie czasu na więcej prób (znalezienie kilku obiecujących minimów i porównanie ich ze sobą). Należy wydłużyć fazę eksploracji.

Rys 6 - Przykłady właściwej i niewłaściwej konfiguracji algorytmu 1+1

Zadanie 1.6: Proszę wyposażyć algorytm 1+1 w zmienny krok, skonfigurować metaparametry i wykorzystać go do optymalizacji **własnej*** funkcji (typ **2D fewminima**). Algorytm powinien posiadać do dyspozycji 400 sprawdzeń funkcji celu (tj. 400 iteracji). W początkowym okresie działania (eksploracja) punkty powinny pokryć cały widoczny obszar a zmienność algorytmu obserwowana na krzywej konwergencji powinna być duża. W końcowym okresie punkty powinny zbiegać się na bardzo niewielkim obszarze a zmienność obserwowana na krzywej konwergencji powinna spadać do 0. Wyniki powinny wyglądać podobnie do tych przedstawionych na Rys. 6 Po wykonaniu zadania proszę je zapisać jako osobny program.

Zadanie 1.7: Proszę skonfigurować algorytm 1+1 do optymalizacji **własnej*** funkcji (typ **2D manyminima**). Algorytm powinien posiadać do dyspozycji 800 sprawdzeń funkcji celu.

Sprawdzanie powtarzalności rozwiązań

Należy pamiętać, że algorytmy optymalizacyjne zazwyczaj działają w sposób niedeterministyczny - a to oznacza konieczność zbierania wyników statystycznych i porównywania ze sobą nie tyle pojedynczych startów, ale parametrów statystycznych opisujących typowe wyniki, powtarzalność oraz spodziewany rozkład tych wyników w przestrzeni. Aby zebrać statystykę z wielu uruchomień algorytmu, najprościej jest wyłączyć w odpowiednim kodzie wizualizację (brak wyświetlania wykresów i punktów) oraz zamknąć funkcję optymalizującą wewnątrz pętli *for* działającej przez tyle iteracji, ile wyników dla metody chcemy zebrać. Należy pamiętać, aby punkt startowy metody oraz najlepszy wynik były reinicjalizowane wewnątrz tej pętli:

```
for repetition = 1:10

%% Reinitialization of a starting point and ending condition
    CurrentMin = 50000;
    ResultX = 1;
    ResultY = 1;
    EndingCondition = 0;
    iter = 0;

    while(EndingCondition == 0);
```

```
        % Here we have a main optimization loop
        ...

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

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

Zadanie 1.8: Proszę porównać algorytmy z zadań 1.5, 1.6 i 1.7. Niech każda z metod zostanie uruchomiona co najmniej 20 razy. Proszę porównać i przedyskutować wyniki oraz zapisać je jako plik excel lub jako strukturę matlaba: do wykorzystania w kolejnych instrukcjach.

Zadania rozszerzające, problemowe:

Zadanie 1.9: Proszę podjąć próbę optymalizacji metaparametrów opracowanych rozwiązań w celu rozwiązywania **własnej** funkcji (typ **2D fewminima**) (zadanie optymalizacyjne 2 rzędu) - Czy do optymalizacji metaparametrów można zastosować metodę grid search? Podpowiedź: testowane algorytmy są niedeterministyczne - to ma istotne znaczenie)

Zadanie 1.10: Proszę wygenerować histogram rozwiązań dla algorytmu 1+1 w rozwiązywaniu **własnej** funkcji (typ 2D **manyminima)**. Proszę podjąć próbę skonfigurowania kroku algorytmu **(Step)** na podstawie wygenerowanych histogramów (jeden histogram dla jednej wartości kroku) oraz pokazać histogram wynikowy (po optymalizacji metaparametru).

Zadanie 1.11: Proszę wykonać opracowanie statystyczne wyników działania dobrze skonfigurowanego algorytmu 1+1 oraz dobrze skonfigurowanego algorytmu gradientowego wielostartowego w zadaniu rozwiązywania **własnej** funkcji (typ **2D fewminima**) - proszę zbadać powtarzalność wyników oraz ich rozkład - i użyć tej informacji aby spróbować poprawić wartości metaparametrów. Proszę sprawdzić, czy taka poprawa jest statystycznie znacząca.

Zadanie 1.12: Proszę przygotować algorytm gradientowy z momentem, proszę przetestować go w zadaniu rozwiązywania **własnej** funkcji (typ 2D, fewminima) oraz dodatkowo funkcji **of_2D_fewminima_5** Czy zastosowanie momentu pozwala podnieść skuteczność algorytmu?

Zadanie 1.13: Proszę zmodyfikować kod algorytmu do optymalizacji 1+1 oraz gradientowego wielostartowego, aby pozwolił na optymalizowanie funkcji **4D** a następnie go skonfigurować w tym celu. (Należy zwrócić uwagę, że wizualizacja w tym wypadku traci sens - jedyny sposób konfiguracji metaparametrów musi się odbywać poprzez krzywą konwergencji lub ew. histogram

Zadanie 1.14: Proszę przetestować działanie algorytmu gradientowego oraz algorytmu 1+1 w zadaniu optymalizacji funkcji zaszumianej zmiennym w czasie szumem losowym - proszę wykorzystać funkcje zawierające w nazwie **rand -** Czy algorytm gradientowy ma sens w swojej podstawowej konfiguracji? Czy można zmienić jego konfigurację tak, aby był w stanie nawiązać walkę z algorytmem 1+1?
Podpowiedź: Rozwiązanie znajduje się już w metaparametrach. Zmiana jednego z nich powinna pozwolić na poprawę wyniku.

Załącznik: kody źródłowe

of_f_randomSampling

```
% A simple random optimization algorithm. It tries new locations until it
% runs out of time. Delay serves as a way of slowing FunctionPlot.
% It requires a function for optimization (any function from folder
% "FunctionsForOptimization"

clear all
addpath FunctionsForOptimization

%% Optimization task:
FunctionForOptimization = str2func('nof_2D_oneminimum_2');

%% Adjustable parameters:
MaxRangeX = [-10 10];  % Range of parameters for optimization
MaxRangeY = [-10 10];

MaxSteps = 100;          % How many iterations do we perform?
FunctionPlot = 1;        % change to 0 If you want to get rid of the underlying function plot
PointPlot = 1;           % Change to 0 if you want to get rid of the visualization
ConvergenceColor = 'r'; % Change color of the convergence curve here
%close all                % Comment this out if you want to have many convergence curves plotted

ViewVect = [0,90];             % Initial viewpoint
Delay =0.001;                  % Inter-loop delay  - to slow down the visualization
FunctionPlotQuality = 0.05;    % Quality of function interpolation. Lower for a quicker run

%% Map initialization
InitialRangeX = MaxRangeX;      % This is the range from which we can draw points.
InitialRangeY = MaxRangeY;

%% Map visualization (this code is not used for problem solving)
TimePercent = 0;
if(FunctionPlot == 1)
    figure(1);
    clf
        vectX = [MaxRangeX(1):FunctionPlotQuality:MaxRangeX(2)];
        vectY = [MaxRangeY(1):FunctionPlotQuality:MaxRangeY(2)];
        [X,Y] = meshgrid(vectX,vectY);    indx = 1;  indy = 1;
        for x = vectX
            indy = 1;
            for y = vectY
                Val(indx,indy) = FunctionForOptimization([x,y]);
                indy = indy + 1;
            end
            indx = indx + 1;
        end
        mesh(X,Y,Val);    surf(X,Y,Val,'LineStyle','none');
        view(ViewVect);   colormap(bone);    hold on
else end

%% Storing of a best solution
    CurrentMin = 50000;
    ResultX = 1;
    ResultY = 1;

%% The main optimization loop
    EndingCondition = 0;
    iter = 0;
    tic;

    while(EndingCondition == 0);
        iter = iter + 1;
```

```
        % Random selection of a candidate for optimum:
        NewX = InitialRangeX(1) +  rand()*(InitialRangeX(2) - InitialRangeX(1));
        NewY = InitialRangeY(1) +  rand()*(InitialRangeY(2) - InitialRangeY(1));
        % Check for constraints (they could be different than the range
        % from which we draw our solutions)
        NewX = min(MaxRangeX(2),max(NewX,MaxRangeX(1)));
        NewY = min(MaxRangeY(2),max(NewY,MaxRangeY(1)));

% If you'd like to provide function as a 2D image or use here any other objective function,
% following line needs to be modified. The 0 passed to the function denotes the fact,
% that the function is constant in time.
        CurrentValue =  FunctionForOptimization([NewX,NewY]);

        if(CurrentValue < CurrentMin)
            CurrentMin = CurrentValue;  % Storing of a historically best result
            ResultX = NewX;
            ResultY = NewY;
            % FunctionPlot (green, if we have a new minimum):
            if(PointPlot == 1)
                figure(1);   plot3(NewY, NewX, CurrentValue,'.g'); hold on
            end
        else
            % FunctionPlot (red, if we don't have a new minimum):
            if(PointPlot == 1)
                figure(1);   plot3(NewY, NewX, CurrentValue,'.r'); hold on
            end
        end

        % Command-window stuff for monitoring of algorithm's progress:
        SimTime = toc;
        clc
        fprintf('\nCurrent best:  %f',CurrentMin);
        fprintf('\nCurrent:       %f',CurrentValue);
        fprintf('\n\n\n');
        fprintf('\nIteration:     %d',iter);
        fprintf('\nTime:          %d',SimTime);

        BestHistory(iter) = CurrentMin;         % Here we store our historically best result
        CurrentHistory(iter) = CurrentValue;    % Here we store our currently investigated result

    if(iter >= MaxSteps)
        EndingCondition = 1;    % To stop the while loop from running
    else

    end
        % If we'd like to slow down the simulation - this line is where it
        % is done:
        pause(Delay);
    end

    figure(2);
    plot(BestHistory,'Color',ConvergenceColor); hold on
    plot(CurrentHistory,'Color',ConvergenceColor,'LineStyle',':'); hold on
     xlabel('Iteration number');
     ylabel('Objective function value');
```