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: Algorytm genetyczny

Cel ćwiczenia: Implementacja algorytmu genetycznego do w zagadnieniu znajdowania minimum lokalnego dwuwymiarowej funkcji celu

Zagadnienia: Algorytm genetyczny, mutacja, selekcja, krzyżowanie, eksploracja, eksploatacja

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

Przygotowanie podstawowego algorytmu genetycznego

Okazuje się, że podstawowy algorytm genetyczny został już zaimplementowany. Jest to metoda 1+1 przygotowana w ramach poprzedniej instrukcji. Tutaj populacja wynosiła 2 osobniki z wyłączną sukcesją elitarną (tzn. z dwuosobniczej populacji osobnik najlepszy przechodził bez zmian do populacji potomnej - a populacja uzupełniana była osobnikiem wygenerowanym na podstawie osobnika elitarnego). Rozbudujemy ten kod, aby umożliwić działanie klasycznego algorytmu genetycznego, wyposażonego w selekcję **n-najlepszych**.

W tym celu uruchomimy nasz kod podstawowy (random sampling) działający na funkcji `nof_2D_oneminimum_2` i dopiszemy do niego nowe metaparametry:

Zbiór metaparametrów na początku kodu:

```
P_size = 20;        %  Population size
n = 10;             %  Parameter n for n best succession
Step = 0.1          %  Mutation range
```

Zainicjalizujemy populację przed uruchomieniem pętli *while*:

```
for k = 1:P_size
      Population(k).OF = Inf;
      Population(k).Parameters(1) = InitialRangeX(1) + ...
      rand()*(InitialRangeX(2) - InitialRangeX(1));
      Population(k).Parameters(2) = InitialRangeY(1) + ...
      rand()*(InitialRangeY(2) - InitialRangeY(1));
end
```

Zwróćmy uwagę, że geny osobników (tzn. ich współrzędne X i Y) oraz odpowiadająca im funkcja przystosowania przechowywane będą teraz w strukturze.

Teraz zmodyfikujemy samą pętlę *while*. W celu zwiększenia przejrzystości instrukcji zostanie tutaj przepisana pełna treść nowej pętli *while* (tzn. na tym etapie można ją w całości wyczyścić i wypełnić krok po kroku poniższym kodem).

Zaczniemy od inkrementacji ilości iteracji:

```
iter = iter + 1;
```

W każdej iteracji pętli *while*, zamiast oceniać wyłącznie jedno rozwiązanie, ocenimy wszystkie rozwiązania znajdujące się aktualnie w populacji. Wyniki tej oceny zapiszemy w polu OF struktury Population:

```
for k = 1:P_size
     Population(k).OF = FunctionForOptimization(Population(k).Parameters);
end
```

Chcemy przechować n-najlepszych rozwiązań, populację wypada więc posortować ze względu na wartość OF:

```
[~,Indices] = sortrows([Population(:).OF]');
```

Teraz wektor **Indices** przechowuje informację o tym, który z osobników populacji ma jaką pozycję w rankingu. Aby odnaleźć osobnika najlepszego, wystarczy wywołać:

```
Population(Indices(1))
```

Aby zaprezentować aktualny stan populacji wyświetlimy ponownie wykres funkcji wraz z nałożonymi wszystkimi osobnikami z populacji:

```
 if(FunctionPlot == 1)
figure(1);
clf
surf(X,Y,Val,'LineStyle','none');
view(ViewVect)
colormap(bone)
hold on
else
end

if(PointPlot == 1)
    for k = 1:1:P_size
    plot3([Population(k).Parameters(1)],...
    [Population(k).Parameters(2)],[Population(k).OF],'.r'); hold on
    end
end
```

Zapis najlepszego osobnika oraz zarazem aktualnego najlepszego rozwiązania odbywa się analogicznie jak wcześniej. Dodatkowo, zapiszemy również genom najlepszego osobnika na wypadek, gdybyśmy chcieli go potem odtworzyć:

```
BestHistory(iter) = Population(Indices(1)).OF;
CurrentHistory(iter) = Population(Indices(floor(P_size/2))).OF;
BestIndividualGenome(iter) = Population(Indices(1));
```

I możemy przejść do budowy populacji potomnej. Aby wypełnić populację potomną będziemy losowali dwóch osobników rodzicielskich z n najlepszych osobników populacji rodzicielskiej, wykorzystywali jednego z nich jako "matrycę" do budowy potomka i dokopiowywali do niego jedną współrzędną drugiego z rodziców. Następnie wynik zmodyfikujemy (zmutujemy) i zapiszemy go w populacji potomnej:

```
for k = 1:1:P_size
   ind1 = randi(n);
   ind2 = randi(n);
   NewPopulation(k) = Population(Indices(ind1));
   NewPopulation(k).Parameters(1) = Population(Indices(ind2)).Parameters(1);
   NewPopulation(k).Parameters = NewPopulation(k).Parameters + ...
     Step*randn(size(NewPopulation(k).Parameters));
   NewPopulation(k).OF = Inf;
   NewPopulation(k).Parameters = ...
     min(MaxRangeX(2),max(NewPopulation(k).Parameters,MaxRangeX(1)));
end
```

Pozostaje już jedynie podmienić populację bazową na potomną:

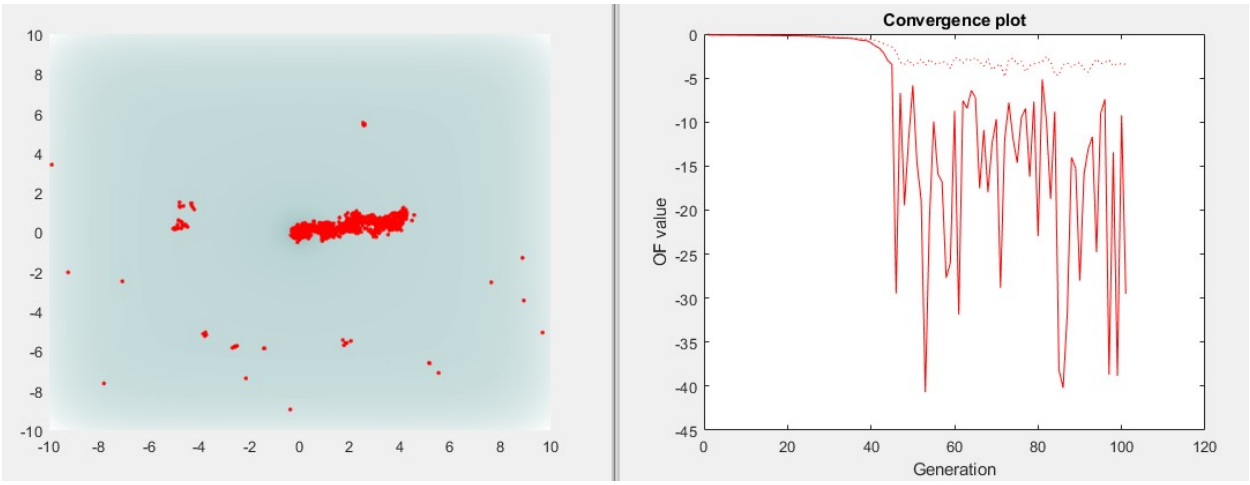
```
Population = NewPopulation;
```

I zakończyć pętlę:

```
SimTime = toc;
clc
fprintf('\nCurrent best: %f',BestHistory(end));
fprintf('\nIteration: %d',iter);
```

```
fprintf('\nTime: %d',SimTime);
if(iter > MaxSteps)
    EndingCondition = 1;
else
end
pause(Delay);
```

Jeśli uruchomiony kod zakończy się wynikiem analogicznym jak na rys. 1 - wszystko działa dobrze.



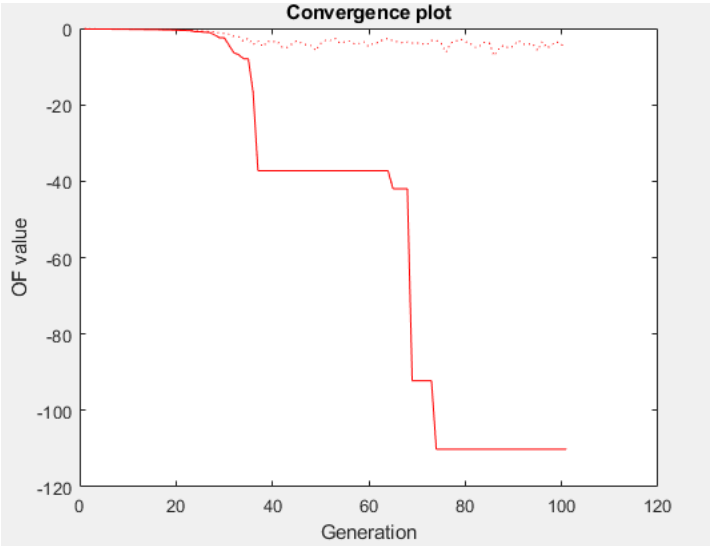
Rys 1 - przykład działania algorytmu genetycznego

Zadanie 2.1: Proszę przygotować, uruchomić, skonfigurować podstawowy algorytm genetyczny (AG) do optymalizacji **własnej*** funkcji 2 zmiennych posiadającej wiele minimów lokalnych (typ **2D manyminima**) - po wykonaniu zadania proszę je zapisać jako osobny program.

Sukcesja elitarna polega na przekazywaniu osobnika najlepszego (lub kilku najlepszych) bez zmian do populacji potomnej. Proszę zmodyfikować kod podstawowego AG w ten sposób, aby osobnik elitarny był kopiowany z populacji rozdzicielskiej:

```
NewPopulation(1) = Population(Indices(1));
```

Oczywiście wymagać to będzie zmiany początku pętli wypełniającej populację potomną - nie będzie zaczynała działać od indeksu 1 ale od indeksu 2. Dobrze zaimplementowana sukcesja elitarna pozwoli na uzyskanie krzywej konwergencji wyglądającej jak na wykresie 2.



Rys 2 - Krzywa konwergencji AG z sukcesją elitarną - widoczny brak pogarszania wyniku.

Uwaga! Aby porównać ze sobą dwa AG o różnych konfiguracjach, wystarczy uruchomić je jeden po drugim nie zamykając okna krzywej konwergencji. Kolejne uruchomienia narysują kolejne krzywe które mogą być łatwo ze sobą porównane. Oczywiście aby zachować przejrzystość wykresu warto zmienić kolor krzywej konwergencji dla kolejnego uruchomienia algorytmu. Brak zamykania okna uzyskujemy poprzez zakomentowanie frazy *close all* w kodzie programu. Zmiana koloru wykresu odbywa się poprzez zmienną `ConvergenceColor`

Zadanie 2.2: Porównaj działanie algorytmu zawierającego sukcesję elitarną z algorytmem nie zawierającym sukcesji elitarnej w zadaniu optymalizacji **własnej*** funkcji 2 zmiennych posiadającej wiele minimów lokalnych (typ **2D manyminima**). Przedstaw wyniki działania przynajmniej trzech uruchomień algorytmu bez sukcesji elitarnej kolorem czerwonym oraz przynajmniej trzech uruchomień algorytmu zawierającego sukcesję elitarną kolorem niebieskim. Czy dodanie elity pozwoliło na polepszenie średniego wyniku? Czy pozwoliło na zwiększenie powtarzalności?

Analogicznie jak w przypadku algorytmu 1+1 balans między eksploracją i eksploatacją ma bardzo istotne znaczenie w konfiguracji AG. Również analogicznie jak w przypadku algorytmu 1+1 właściwe wartości metaparametrów można dobierać na podstawie oceny krzywej konwergencji.

Proszę zaimplementować zmianę kroku mutacji zgodnie z krzywą sigmoidalną. W tym celu na początku kodu powinny pojawić się nowe metaparametry:

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

A wewnątrz pętli while wyznaczenie nowej wartości kroku mutacji:

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

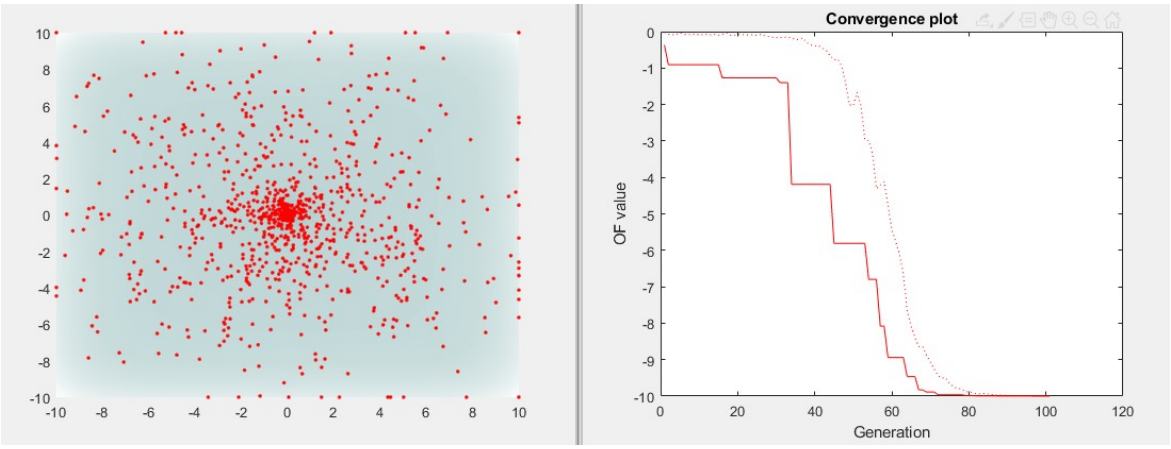
Oczywiście podczas mutacji nie będziemy się już odwoływać do *Step* (stałej) ale do *Step(iter)* (zmiennej). Na koniec można wyświetlić wartość *Step* w funkcji iteracji:

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

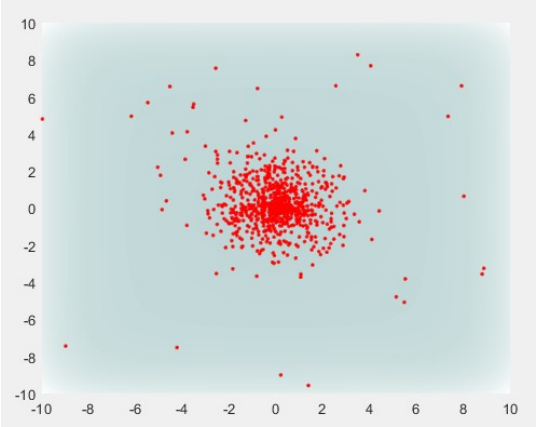
Pierwszy z metaparametrów (InitialStep) odpowiada za początkowy zakres mutacji. Powinien być na tyle duży, aby algorytm "rozprzestrzenił się" po całej rozważanej przestrzeni ale na tyle mały, aby warunek granicy obszaru nie uruchamiał się zbyt często (nie chcemy, aby w początkowym okresie działania algorytmu większość osobników potomnych lądowała na granicy zakresu). P1 odpowiada za punkt zmiany znaku drugiej pochodnej sigmoidu - tzn. punkt w którym wykres kroku osiąga połowę swojej wysokości. Ustawienie wartości na "2" (czyli w równaniu sigmoidu podzielenie ilości iteracji przez 2) będzie oznaczało, że punkt ten przypada w połowie iteracji. P2 oznacza jak stromo nachylony jest sigmoid - tzn. jak wyraźny jest podział między eksploracją i eksploatacją.

Parametry powinny być tak dobrane, aby w początkowym etapie algorytm przeczesywał całą dostępną przestrzeń i zachowywał dużą różnorodność a następnie stopniowo przechodził do fazy eksploatacyjnej i w końcówce działania pogłębiał wyłącznie jedno najlepsze minimum. Ostatnie

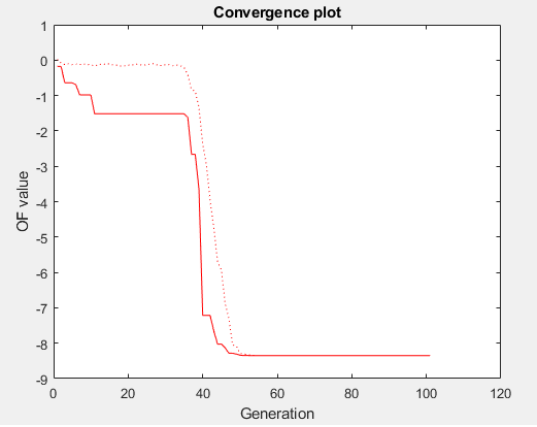
iteracje powinny spowodować zmniejszenie się różnorodności algorytmu do prawie 0 (wartość osobnika najlepszego i typowego na krzywej konwergencji powinny być prawie takie same). Przykład KK dla dobrze skonfigurowanego AG przedstawia rys. 3.Przykładowe wykresy oznaczające błędną konfigurację widoczne są na Rys. 4.



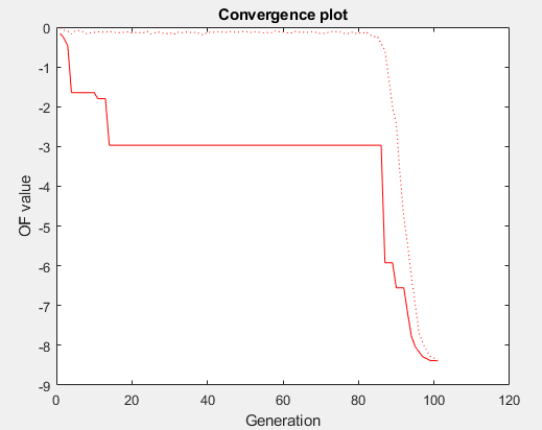
Rys 3 - Krzywa konwergencji dobrze skonfigurowanego AG



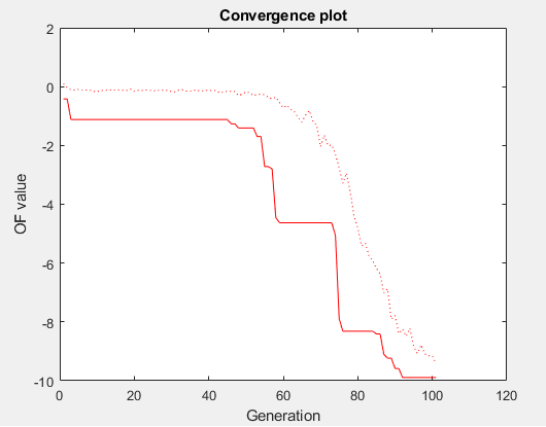
a) Za słaba eksploracja (punkty nie "próbkują" całej dostępnej przestrzeni



b) Zbyt gwałtowne przejście od eksploracji do eksploatacji, "martwe iteracje" - algorytm od połowy iteracji nie robi już nic...



c) Zbyt długa eksploracja i zbyt gwałtowne przejście do eksploatacji



d) Zbyt słaba eksploatacja - algorytm do końca ma duże zróżnicowanie

Rys 4 - Błędne konfiguracje Algorytmu Genetycznego

Zadanie 2.3: Dobierz wartości odpowiadające za balans między eksploracją i eksploatacją w ten sposób, aby w zadaniu optymalizacji **własnej*** funkcji 2 zmiennych posiadającej wiele minimów lokalnych (typ **2D manyminima**) uzyskać poprawnie wyglądającą krzywą konwergencji. Niech Twój algorytm korzysta z ok. 800 sprawdzeń funkcji celu. Powtórz zadanie dla funkcji typu **2D fewminima,** tym razem używając 400 sprawdzeń funkcji celu. Zapisz skrypty jako osobne programy.

Zadanie 2.4: Porównaj działanie:
- A. Gradientowego wielostartowego w konfiguracji z zadania 1.5
- A. 1+1 w konfiguracjach z zadań 1.6 i 1.7
- Algorytmu genetycznego z zadania 2.3

W zadaniach optymalizacji **własnej*** funkcji 2 zmiennych posiadającej wiele minimów lokalnych (typ **2D manyminima**) oraz optymalizacji własnej funkcji 2 zmiennych posiadającej kilka minimów lokalnych (typ **2D fewminima**). Wyznacz średni wynik oraz jego odchylenie standardowe dla 10 prób dla każdego z powyższych algorytmów. Możesz skorzystać z wyników wyznaczonych w ramach 1 instrukcji, jeśli zostały zaakceptowane przez prowadzącego. Zapisz reprezentatywne krzywe konwergencji dla każdej konfiguracji (powinieneś zaprezentować 6 krzywych: dla 3 metod i 2 funkcji).

W praktyce konfiguracja algorytmów odbywa się za pomocą iteracyjnego procesu testów zachodzącego według następującego schematu:

1. Dobierz "rozsądną" konfigurację, niech krzywa konwergencji wygląda poprawnie
2. Uruchom algorytm wielokrotnie
3. zaobserwuj zachowanie wielu krzywych konwergencji jednocześnie. Czy metoda jest powtarzalna? Czy problemy występują w części eksploracyjnej? Czy algorytm jest wrażliwy na minima lokalne? Czy algorytm ma kłopoty z eksploatacją?
4. Zaproponuj zmianę mającą na celu rozwiązanie problemów zidentyfikowanych w (3) a następnie wróć do punktu (2). Powtarzaj wielokrotnie, aż nie będziesz już w stanie poprawić rozwiązań.

Aby proces ten był możliwy w naszym przypadku zastosujemy ponownie przedstawioną w instrukcji 1 modyfikację polegającą na wyłączeniu wizualizacji i umożliwieniu metodzie efektywnego i szybkiego dostarczania danych statystycznych. Następnie zmodyfikujemy kod, aby był w stanie obsłużyć funkcje wielu zmiennych. W tym celu zmodyfikować musimy zaledwie trzy fragmenty naszego kodu:

1) Na stałe zakomentować lub wyłączyć trzeba będzie całą wizualizację postępów (tj. "wyświetlanie kropek" lub wykresu funkcji. Mówimy tu o funkcjach wielu zmiennych, nie będziemy więc mogli wygodnie podejrzeć ich wyglądu. Jedynym możliwym do obejrzenia wykresem będzie wynikowa krzywa konwergencji.

2) W miejscu, gdzie generujemy populację początkową musimy zamiast dwóch elementów wektora Parameters przygotować ich ilość odpowiadającą ilości wymiarów problemu. Proszę zwrócić uwagę, że w mutacji oraz wywołaniu funkcji celu NIE musimy już niczego modyfikować - oba elementy działają tutaj w sposób uniwersalny.

Zadanie 2.5: Skonfiguruj AG do rozwiązywania problemu **własnej*** funkcji typu **multidimensional** (wielu zmiennych). Wykorzystaj domyślny zestaw metaparametrów. Zastosuj 1000 sprawdzeń funkcji celu:

```
P_size = 20;
n = 10;
InitialStep = 2;
P1 = 2;
P2 = 10;
```

Sprawdź powtarzalność AG z takimi parametrami, zidentyfikuj problemy, popraw konfigurację i ponownie ją przetestuj. W przypadku braku powtarzalności oprócz wpływania na długość fazy eksploracyjnej przetestuj też wpływ zmiany wielkości populacji i nacisku selektywnego. Czy udało się uzyskać poprawę powtarzalności i jakości rozwiązania? Powtórz czynność kilkukrotnie (zmiana parametrów i testy) zapisując poszczególne zestawy krzywych konwergencji oraz uzyskiwane przez algorytm wyniki. Bądź przygotowany na wyjaśnienie podjętych decyzji.

Zadania dodatkowe i rozszerzające:

Zadanie 2.6: Jak na działanie AG wpływa niedeterminizm testu wartości optymalizowanej funkcji? Przetestuj działanie swojego algorytmu w optymalizacji funkcji **jakiej_rand**, porównaj to działanie z dobrze skonfigurowanym algorytmem 1+1, sprawdź czy da się poprawić otrzymany rezultat poprzez lepszą konfigurację metaparametrów AG. Zwróć uwagę, że w optymalizacji funkcji niederministycznej pomóc powinien mniejszy nacisk selektywny.

Zadanie 2.7: W niektórych przypadkach zadanie optymalizacji zmienia się w trakcie jej trwania (tzw. problemy adaptacyjne). W naszych kodach przetestować tą klasę problemu można za pomocą funkcji **_Adaptive -** w tym celu oprócz podmiany nazwy funkcji trzeba również przekazać do funkcji aktualny "procent czasu pozostałego" - czyli zamiast wywołania funkcji tak:

```
Population(k).OF = FunctionForOptimization(Population(k).Parameters);
```

należy to zrobić tak:

```
Population(k).OF = FunctionForOptimization(Population(k).Parameters,iter/MaxSteps);
```

W zadaniach tego typu istotny jest nie tyle wynik optymalizacji na końcu, ale raczej zdolność podążania za zmianami położenia minimum przez cały czas pracy algorytmu. Jedną z możliwych modyfikacji kodu jest sprawienie, aby osobniki potomne tworzone były według następującego schematu: Połowa populacji posiada mały zakres mutacji (umożliwiający eksploatację) a połowa populacji posiada duży zakres mutacji (pozwalający na eksplorację). Należy skonfigurować te dwa parametry (co oznacza "mały" i "duży"?) tak, aby zminimalizować ŚREDNI błąd przez cały czas optymalizacji (tzn. kryterium w tym przypadku jest wartość *mean(BestHistory)*

Zadanie 2.8: Poczynając od finalnego rezultatu zadania 2.5 wykonaj optymalizację metaparametrów AG z zastosowaniem algorytmu 1+1 (niech punkt startowy metody to punkt przyjęty jako najlepszy w zadaniu 2.5. Czy udało się poprawić wyniki metody w sposób statystycznie istotny? wykonaj czynność ponownie - tym razem dla innej funkcji wielu zmiennych. Czy tym razem poprawa po zastosowaniu metody 1+1 była większa czy może podobnego rzędu co poprzednio?

Zadanie 2.9*: Jednym z bardzo skutecznych rozwiązań w obszarze algorytmów ewolucyjnych jest tzw. algorytm memetyczny. W jego przypadku, wewnątrz standardowej pętli algorytmu genetycznego znajduje się algorytm gradientowy wykonujący kilka kroków optymalizacji dla każdego osobnika. Innymi słowy procedura będzie wyglądała następująco:

1) Oceń przystosowanie osobników
2) Wygeneruj populację potomną
3) Dla każdego osobnika populacji potomnej uruchom kilka iteracji algorytmu gradientowego o stałym kroku (ale zależnym od kroku mutacji dla danej iteracji AG).
4) Wróć do punktu (1)

Czy taki algorytm - po odpowiednim skonfigurowaniu, tj. np. wybraniu ilości kroków gradientu dla każdego osobnika, po wyposażeniu go w taką samą ilość "sprawdzeń funkcji celu" jest statystycznie odmienny od AG w optymalizacji Twojej **własnej** funkcji typu **multidimensional?**

Uwaga! Pamiętaj, aby zachować uczciwe warunki