Wstęp do metod numerycznych

14. Uwagi o minimalizacji globalnej

P. F. Góra

http://th-www.if.uj.edu.pl/zfs/gora/

2018

Konieczność minimalizacji globalnej

W wielu przypadkach o minimalizowanych funkcjach *wiemy*, że posiadają tylko jedno minimum. W innych sytuacjach faktycznie zadowalamy się znalezieniem *jakiegoś* minimum. Jednak w bardzo wielu przypadkach o dużym znaczeniu praktycznym albo chcemy znaleźć minimum globalne, albo, jeśli istnieje wiele minimów lokalnych, chcemy znaleźć jak najwięcej z nich i poznać ich wzajemne relacje.

Rozważane dotąd algorytmy poszukiwania minimów szukają minimów lokalnych, a więc są nieprzydatne w problemie minimalizacji globalnej. Co najwyżej znalezione innymi metodami minimum globalne może zostać użyte jako punkt startowy dla metody gradientów sprzężonych, zmiennej metryki lub Powella, które wówczas powinny szybko znaleźć jego lepsze przybliżenie.

Z uwagi na swoje bardzo duże znaczenie praktyczne, przedstawione tu pokrótce algorytmy — a także ich warianty, modyfikacje, uogólnienia i algorytmy należące do całkiem innych kategorii — są przedmiotem bardzo intensywnych analiz i badań naukowych. Niektóre algorytmy dobrze działają dla pewnych klas problemów, podczas gdy dla innych zawodzą. *State of the art* w tej dziedzinie zmienia się niemalże z roku na rok.

Ponieważ nie wiemy*, gdzie znajduje się poszukiwane minimum globalne (bądź alternatywne minima lokalne), algorytmy globalnej optymalizacji bardzo często zawierają element *losowości*.

*Gdybyśmy wiedzieli, nie musielibyśmy szukać ☺.

Uwagi o metodach zachłannych

Wydaje się, że strategia zachłanna, polegająca na (możliwie gęstym) "przeskanowaniu" całego obszaru, w którym spodziewamy się znaleźć minimum, może być pomocna. To nieprawda, *zwłaszcza w wielu wymiarach*. Przypuśćmy, że w celu "przeskanowania" jednego wymiaru musimy wykonać N kroków (na przykład sto). Wówczas do "przeskanowania" dwu wymiarów musimy wykonać N^2 kroków (na przykład dziesięć tysięcy), trzech — N^3 kroków (na przykład milion), czterech — N^4 (na przykład sto milionów), i tak dalej. Szybko staje się to niezwykle kosztowne. Strategie zachłanne nie sprawdzają się przy poszukiwaniu minimów globalnych funkcji (bardzo) wielu zmiennych. Niewiele lepiej działa też strategia polegająca na losowaniu wielu punktów początkowych i startowaniu z nich znanych metod lokalnych. Potrzebne jest jakieś "sprytne", "inteligentne" rozwiązanie.

Metoda Monte Carlo — motywacja

Zaletą omawianych dotąd metod poszukiwania minimów lokalnych jest to, że zawsze wykonują one "dobre" kroki, w stronę malejących wartości funkcji. Jest to jednak zarazem wadą, jeśli poszukujemy minimum globalnego: Algorytm lokalny, gdy raz wejdzie do basenu atrakcji jakiegoś minimum lokalnego, nie może z niego wyjść. Opłaca się zatem od czasu do czasu wykonać krok w złą stronę, "pod górę". Widzieliśmy to przy okazji omawiania metody *stochastic gradient descent*.

Ponieważ nie wiemy, gdzie leży poszukiwane minimum globalne, pozwalamy algorytmowi eksplorować dostępną przestrzeń stanów w sposób *losowy*. Również *losowo* zezwalamy na wykonanie "złych" kroków, podczas gdy kroki "dobre" są (w oryginalnej wersji algorytmu) zawsze akceptowane. Kroki "bardzo złe", w wyniku których wartość *minimalizowanej* funkcji rośnie znacznie, powinny być akceptowane rzadziej, niż kroki "trochę złe", w wyniku których wartość funkcji rośnie niewiele.

Algorytm Monte Carlo

Niech $f : \mathbb{R}^N \to \mathbb{R}$ będzie funkcją, której minimum poszukujemy. Startujemy z pewnego punktu $\mathbf{x}_0$, obliczamy $f_0 = f(\mathbf{x}_0)$. $\mathbf{x}_{\text{min}} = \mathbf{x}_0$, $f_{\text{min}} = f_0$. $\sigma > 0$, $T > 0$ są parametrami. Następnie wykonujemy M kroków iteracji:

1. Obliczamy $\widetilde{\mathbf{x}} = \mathbf{x}_k + \sigma\boldsymbol{\xi}_k$, gdzie $\boldsymbol{\xi}_k$ jest N-wymiarową gaussowską zmienną losową.
2. Obliczamy $\tilde{f} = f(\widetilde{\mathbf{x}})$.
3. Jeżeli $\tilde{f} < f_k$, akceptujemy nowe położenie ($\mathbf{x}_{k+1} = \widetilde{\mathbf{x}}$, $f_{k+1} = \tilde{f}$).
4. Jeżeli $\tilde{f} < f_{\text{min}}$, $\mathbf{x}_{\text{min}} = \mathbf{x}_{k+1}$, $f_{\text{min}} = \tilde{f}$.
5. Jeżeli $\tilde{f} > f_k$,
   (a) Losujemy zmienną losową z o rozkładzie równomiernym na przedziale $[0, 1]$.
   (b) Jeżeli
   $$z < \exp\left(-\frac{\tilde{f} - f_k}{T}\right) \tag{1}$$
   akceptujemy nowe położenie. Jeżeli (1) nie zachodzi, nie akceptujemy nowego położenia.
6. `GOTO 1`

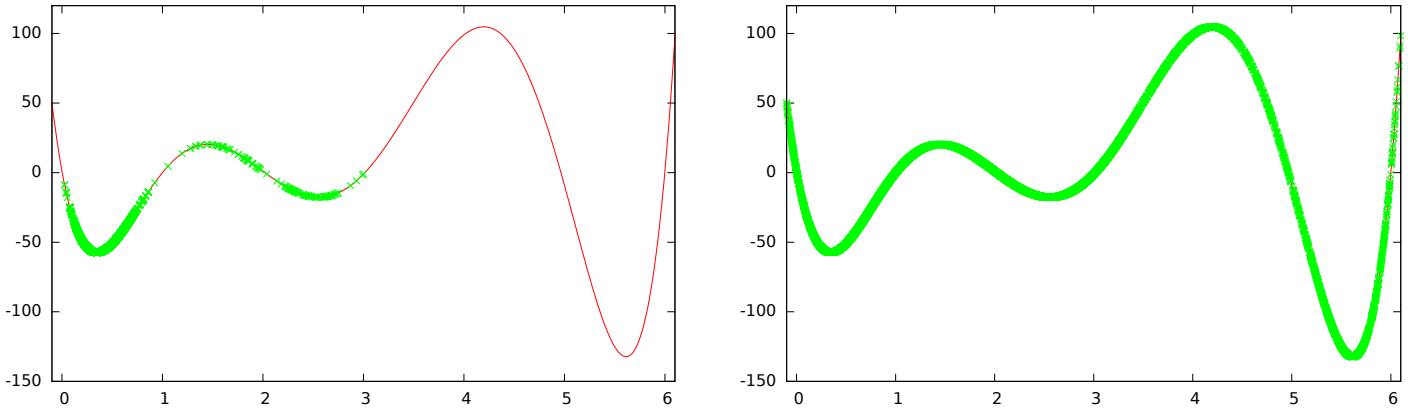
Uwagi

- Nie ma naturalnego kryterium stopu. Wykonujemy tyle kroków, M, ile wstępnie ustaliliśmy.
- Zamiast brać duże M, *znacznie lepiej* jest kilkakrotnie (wielokrotnie) *wylosować* punkt początkowy z podanego zakresu (z podanej kostki wielowymiarowej), a następnie wykonać niewielką (kilkaset, kilka tysięcy) liczbę iteracji rozpoczynając z każdego wylosowanego punktu początkowego.
- **Dla dużej liczby wymiarów nie ma gwarancji,** że wylosowane punkty początkowe same z siebie znajdą się w basenach atrakcji wszystkich interesujących minimów.
- Zamiast brać zmienne gaussowskie ξ_k i boltzmannowskie prawdopodobieństwo (1), można losować kroki z innego rozkładu i inaczej określić prawdopodobieństwo akceptacji stanu o *wyższej* wartości funkcji. Dyskusja tego punktu dalece wykracza poza ramy tego wykładu.

Przyjęcie kombinacji Gauss-Boltzmann oznacza poszukiwanie minimum układu znajdującego się w stanie równowagi termodynamicznej.

- Zakładając, że stosujemy kombinację Gauss-Boltzmann, jak w algorytmie powyżej, zauważmy, że $T \to 0$ oznacza, że konfiguracje o wyższej wartości funkcji nie będą akceptowane. Na odwrót, $T \to \infty$ oznacza, że każda konfiguracja zostanie zaakceptowana. W praktyce, większe wartości T oznaczają, że algorytmowi łatwiej iść "pod górę", za cenę rezygnacji z trzymania się w pobliżu lokalnego minimum; efektywnie sprowadza się to do równomiernego skanowania całego dostępnego zakresu, czego chcieliśmy uniknąć!
- Parametry σ, T — w braku lepszych pomysłów — dobieramy metodą prób i błędów, kierując się wcześniejszym doświadczeniem.
- Jeśli bariera oddzielająca dwa baseny atrakcji jest wysoka, czas potrzebny do jej pokonania może być bardzo duży: *Mean First Passgae Time* $\sim \exp(\text{wysokość\_bariery}/T)$.

Przykład



8192 kroki metody Monte Carlo w zastosowaniu do pewnej funkcji. $\sigma = 0.125$, lewy panel $T = 8$, prawy panel $T = 128$. Minimum globalne zostało znalezione, ale za cenę bardzo dokładnego przeskalowania całego przedziału. Na lewym panelu, po wykonaniu takiej samej liczy kroków, minimum globalne *nie* zostało znalezione.

Metoda Monte Carlo i problemy dyskretne

Wielką zaletą metody Monte Carlo jest to, że można jej używać do minimalizacji problemów dyskretnych, to jest takich, w których zmienne mogą przybierać tylko pewne z góry określone wartości. Powiedzmy, niech układ będzie opisywany przez N zmiennych dyskretnych $\{x_1, x_2, \ldots, x_N\}$, z których każda może przyjmować tylko wartości $\{d_1, d_2, ..., d_s\}$. W kroku Monte Carlo zmieniamy wartość jednej ze zmiennych $\{x_i\}$, losując dla niej (z zadanego rozkładu prawdopodobieństwa) nową wartość dopuszczalną. Obliczamy nową wartośc funkcji dopasowania. Jeśli jest ona mniejsza (lepsza) od dotychczasowej, akceptujemy nową konfigurację. Jeśli jest większa, nową konfigurację akceptujemy z prawdopodobieństwem zależnym od starej i nowej wartości funkcji dopasowania — na przykład według kryterium bolztmannowskiego. Następnie procedurę powtarzamy dla innej zmiennej (można je wybierać sekwencyjnie, a można też losować, którą zmienną

będziemy modyfikować). Można też jednocześnie modyfikować całą grupę zmiennych, a dopiero później obliczać funkcję dopasowania.

Niestety, taki wariant metody Monte Carlo może — zwłaszcza w przypadku problemów bardzo wielo wymiarowych — wpaść w pułapkę minimum lokalnego. Dzieje się tak wtedy, gdy przejście z minimum lokalnego do jakiegoś innego, lepszego minimum, wymaga przekonfigurowania *wielu* zmiennych, co łączy się ze wzrostem wartości minimalizowanej funkcji.

Algorytmy genetyczne

W algorytmie Monte Carlo zmiany zawsze mają charakter lokalny, przez co proces wspinania się na barierę oddzielającą jedno minimum od drugiego może być bardzo powolny. Trzeba znaleźć sposób dokonywania znacznych zmian w sposób losowy, ale jednocześnie sensowny.

Algorytmy genetyczne inspirowane są podstawową zasadą ewolucyjną: *survival of the fittest* (najlepiej dostosowani przeżywają). W kontekście algorytmów genetycznych często, zamiast o minimalizacji, mówi się o maksymalizacji *funkcji dopasowania* (fitness function), zwanej także funkcją celu; bezpiecznie jest zatem w tym kontekście mówić o optymalizacji.

W algorytmie genetycznym mamy do czynienia z całą populacją osobników. Jeden osobnik reprezentuje jakiś stan optymalizowanego układu. Liczba osobników (liczność populacji) jest ustalona i jest znacznie mniejsza, niż liczba wszystkich możliwych stanów.

Chromosomy

Każdy osobnik reprezentowany jest przez *chromosom*. Chromosom stanowi zdyskretyzowany — na ogół, choć niekoniecznie, binarny — zapis stanu układu. Zakładamy, że istnieje jednoznaczny sposób przypisania danemu chromosomowi określonej wartości funkcji dopasowania badanego układu (co nie musi być proste). Zakładamy, że każdy chromosom dopuszczony przez reguły zapisu odpowiada dopuszczalnemu stanowi badanego układu i że nie ma dopuszczalnych stanów, które nie byłyby reprezentowane przez chromosomy.

Chromosom może **na przykład** reprezentować:

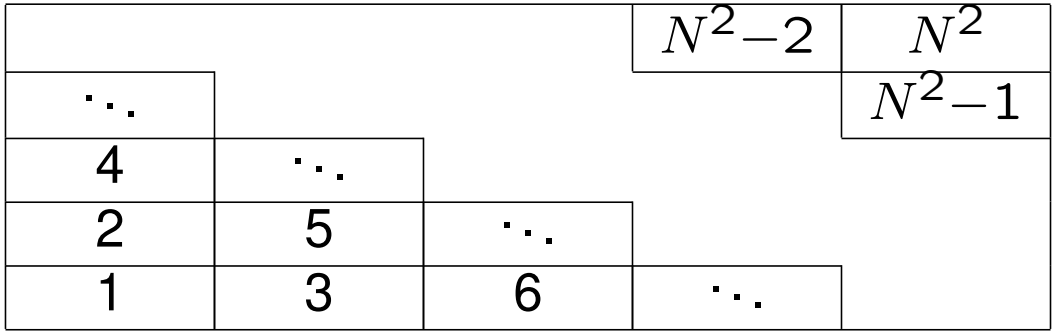
- Numer przedziału, plakietki, N-wymiarowej hiperkostki, w środku którego/której obliczamy wartość minimalizowanej funkcji;
- stan poszczególnych spinów (momentów magnetycznych) w badanym krysztale;

- decyzje poszczególnych agentów w agentowym modelu rynku;
- stan poszczególnych węzłów w badanej sieci społecznej;
- stan poszczególnych węzłów w modelu rozprzestrzeniania się epidemii;
- ustawienia poszczególnych bramek lub przełączników w badanym złożonym układzie elektronicznym;
- wybór poszczególnych krawędzi w problemie komiwojażera;
- wybór gałęzi w drzewie decyzyjnym
- i bardzo wiele innych.

Największa trudność w zastosowaniu algorytmów genetycznych często sprowadza się do znalezienia (wymyślenia?) odpowiedniej reprezentacji "chromosomowej" dla badanego problemu.

Przykład

Jeśli kostkę dwuwymiarową dzielimy na siatkę o N oczkach wzdłuż każdego boku, poszczególne plakietki możemy *na przykład* numerować diagonalnie:



Poszczególne chromosomy będą teraz odpowiadać binarnie zapisanym numerom plakietek. Wartością funkcji dopasowania chromosomu jest wartość optymalizowanej funkcji w środku plakietki o danym numerze.

Selekcja

W układach biologicznych osobniki najlepiej dopasowane mają największe szanse wydania potomstwa. W algorytmie genetycznym obliczamy wartość funkcji dopasowania dla wszystkich członków populacji, po czym przydzielamy im prawdopodobieństwa na zasadzie "kto bardziej dopasowany, ten lepszy". Zakładając, że dyskutowana optymalizacja jest w tym wypadku maksymalizacją i że wartości funkcji dopasowania są nieujemne[†], każdemu osobnikowi (chromosomowi z aktualnej populacji) przypisujemy wycinek koła proporcjonalny do jego wartości funkcji dopasowania:

$$p_i = \frac{f_i}{\sum_j f_j} \tag{2}$$

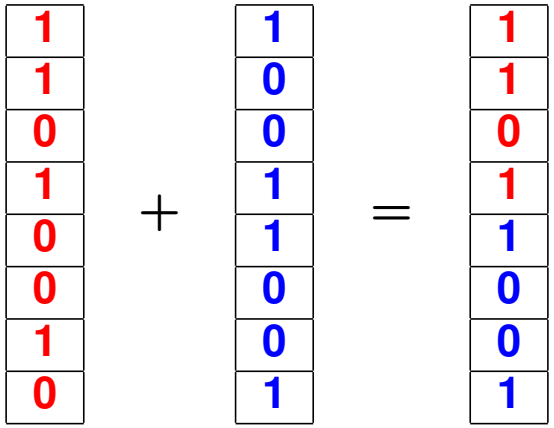
[†]W przypadku wartości ujemnych, można je zawsze przesunąć, tak, aby najmniejsza wartość równała się zero. W przypadku minimalizacji, bierzemy wartości przeciwne.

gdzie f_j oznacza wartość funkcji dopasowania j-tego członka populacji. Metoda ta zwana jest "metodą ruletki". Jest oczywiste, że zamiast wycinka koła, wystarczy brać elementy odcinka $[0, 2\pi]$, czyli, po przeskalowaniu, odcinka $[0, 1]$. Jest to metoda najbardziej popularna, choć istnieją też inne, znacznie bardziej wyrafinowane warianty.

Ustaliwszy prawdopodobieństwa, losujemy dwoje "rodziców", których potomek stanie się członkiem populacji w następnej iteracji, zwanej w tym kontekście *epoką*.

Operatory genetyczne

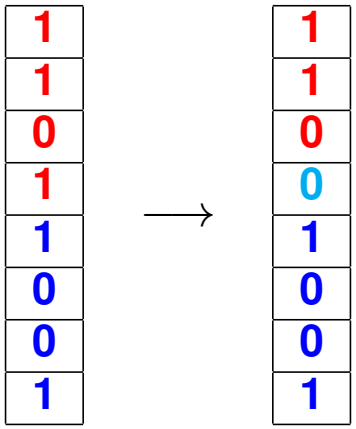
Operator krzyżowania: Po wyborze dwóch chromosomów rodzicielskich, tworzymy z nich potomka, biorąc pierwszą połówkę od jednego rodzica, drugą od drugiego; wariantowo możliwe jest losowanie za każdym razem pozycji, w której nastąpi krzyżowanie (dla ośmiobitowego chromosomu nie tylko 4-4, ale na przykład 5-3, 2-6 itp):



(3)

Po utworzeniu jednego potomka, rodzice wracają do puli, z której odbywa się losowanie. Tworzymy łącznie tylu potomków (tyle razy losujemy parę rodzicielską), ile wynosi zadana z góry liczność populacji.

Operator mutacji: Po utworzeniu każdego potomka, z *niewielkim* prawdopodobieństwem poddajemy go mutacji: Jeśli mutacja ma zajść, losujemy pozycję, która ma zostać zmutowana, i odwracamy na niej bit (lub losujemy inną, niż dotychczasowa, dopuszczalną wartość w notacji niebinarnej). Podkreślam, że prawdopodobieństwo mutacji jest *niewielkie*.



(4)

Start i stop

Na początku populację startową, zdecydowanie mniej liczną, niż liczba potencjalnie dostępnych stanów, dobieramy losowo, z rozkładu równomiernego na całej przestrzeni stanów.

W każdym kroku iteracji, po wygenerowani tylu potomków, ilu ma liczyć populacja, obliczamy dla nich funkcję dopasowania i rozpoczynamy kolejną *epokę*.

Iteracje kończymy gdy wyczerpaliśmy z góry założoną liczbę epok lub gdy wszyscy członkowie populacji (chromosomy) znajdą się w niewielkim otoczeniu jakiegoś stanu, lub gdy funkcja dopasowania przestanie się zmieniać na przestrzeni kilku epok.

Paralelizacja

Ze względu na to, że losowanie jakiejś pary rodzicielskiej jest niezależne od wyników losowania innych (poprzednich) par, algorytmy genetyczne latwo jest zrównoleglić. Mianowicie, po obliczeniu funkcji dopasowania i prawdopodobieństw wyboru poszczególnych chromosomów, każdemu procesorowi (wątkowi) przydzielamy pewną liczbę potomków do wygenerowania, tak, aby suma wszystkich była równa zadanej liczności populacji. Gdy już wszyscy potomkowie zostali wygenerowani, scalamy wątki, obliczamy funkcję dopasowania i prawdopodobieństwa wyboru dla członków populacji potomnej, po czym przechodzimy do kolejnej epoki. Zapamiętujemy chromosom, który w ciągu wszystkich epok dał najlepszą wartość funkcji dopasowania.

Particle Swarm Optimization

Kolejna klasa algorytmów służy do minimalizacji funkcji zmiennych zmieniających się w sposób ciągły. Jest ona także motywowana biologicznie, w tym wypadku obserwacją zachowań stad ptaków lub ławic ryb.

W algorytmie mamy do czynienia z niezbyt licznym stadem (ang. *swarm*, dosłownie "rój") osobników. Każdy indywidualny osobnik podlega nieskomplikowanym regułom, ponieważ jednak osobniki w pewien sposób oddziałują ze sobą, całe stado wykazuje się pewną "inteligencją" (*swarm intelligence*).

Mamy funkcję $f : \mathbb{R}^N \to \mathbb{R}$. Każdemu osobnikowi przydzielamy położenie $\mathbf{x}_i \in \mathbb{R}^N$ i prędkość $\mathbf{v}_i \in \mathbb{R}^N$. Początkowe położenia losujemy z rozkładu jednostajnego na całym badanym obszarze. Początkowe prędkości także losujemy, dbając wszakże o to, aby nie były one zbyt duże. Dla każdego członka stada zapamiętujemy najmniejszą znalezioną *przez niego* wartość funkcji dopasowania, $f_{\mathrm{min},i}$, i odpowiadające jej położenie, $\mathbf{x}_{\mathrm{min},i}$. Inicjalizujemy je jako wartości w położeniach początkowych. Zapamiętujemy też *globalną* najmniejszą wartość funkcji dopasowania, f_{glob} i odpowiadające jej położenie, $\mathbf{x}_{\mathrm{glob}}$. $0 < \omega < 1$, $a > 0$, $b > 0$ są parametrami.

Pozwalamy członkom stada eksplorować dostępną przestrzeń. Każdy członek stada z jednej strony ma tendencję do zachowania swojej dotychczasowej prędkości, z drugiej do orientowania się w stronę najlepszej znalezionej *przez siebie* wartości, z trzeciej — do orientowania się w stronę najlepszej znalezionej wartości *globalnej*.

Sekwencyjnie dla wszystkich członków stada wykonujemy następujące operacje:

- Losujemy dwie liczby losowe p, q z rozkładu jednostajnego na przedziale $[0, 1]$.
- $\mathbf{v}_{i+1} = \omega \mathbf{v}_i + a \cdot p \cdot (\mathbf{x}_{\text{min},i} - \mathbf{x}_i) + b \cdot q \cdot (\mathbf{x}_{\text{glob}} - \mathbf{x}_i)$
- $\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{v}_{i+1}$
- Obliczamy $f_{i+1} = f(\mathbf{x}_{i+1})$.
- Jeżeli $f_{i+1} < f_{\text{min},i}$, to $f_{\text{min},i} = f_{i+1}$ oraz $\mathbf{x}_{\text{min},i} = \mathbf{x}_{i+1}$.
- Jeżeli $f_{i+1} < f_{\text{glob}}$, to $f_{\text{glob}} = f_{i+1}$ oraz $\mathbf{x}_{\text{glob}} = \mathbf{x}_{i+1}$.

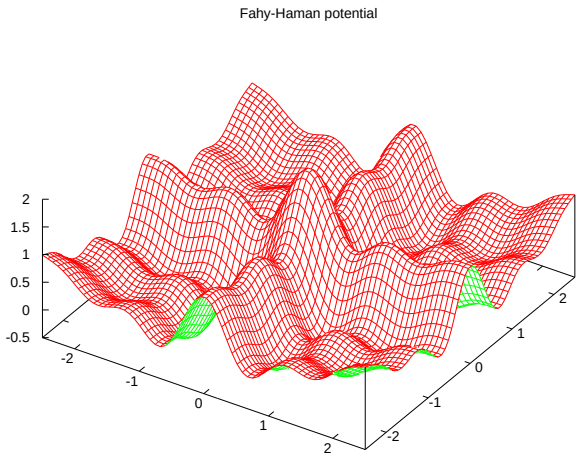
Po wykonaniu całego *sweepu* przechodzimy do następnej iteracji.

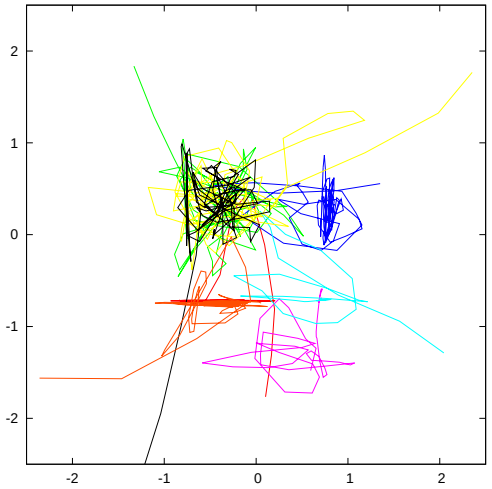
Uwagi

- Tendencja do podążania w stronę optimum znalezionego przez *danego* członka stada odpowiada tendencji do podążania "w dół". Tendencja do podążania w stronę minimum globalnego odpowiada za inteligencję roju.
- Wadą algorytmu jest to, że nie zwraca on uwagi na kształt i prędkość zmienności optymalizowanej funkcji, ograniczając się do porównań mniejszy-większy.
- Zaletą tego podejścia jest brak konieczności obliczania gradientów. Algorytmu można zatem użyć do minimalizacji funkcji ciągłych, ale niekoniecznie różniczkowalnych.
- Dobór wartości parametrów ω, a, b sam jest przedmiotem intensywnych badań. Czasami dobiera się je eksperymentalnie, sprawdzając, jak przebiega minimalizacja problemów w jakimś sensie podobnych do naszego, ale od niego prostszych (metaoptymalizacja).

- Koniecznie należy zachować warunek $0 < \omega < 1$. Dla $\omega > 1$ prędkości wybuchają (co można pokazać analitycznie).
- Dla układów o znacznej wymiarowości, członkowie stada miewają tendencję do grupowania się wokół jakiegoś minimum, niekoniecznie globalnego, pozostawiając znaczne fragmenty przestrzeni niewyeksplorowane. Aby tego uniknąć, dzielimy całe stado nad pod-stada, których członkowie widzą się wzajemnie, ale nie widzą pozostałych (widzą minimum globalne dla swojego pod-stada, nie widzą minimów znalezionych przez inne pod-stada). Aby to zrealizować w architekturze równoległej, każdemu procesorowi (wątkowi) przydzielamy własne pod-stado. Na końcu scalamy wątki i wybieramy najlepszą osiągniętą wartość.

Przykład — potencjał Fahy-Hamana



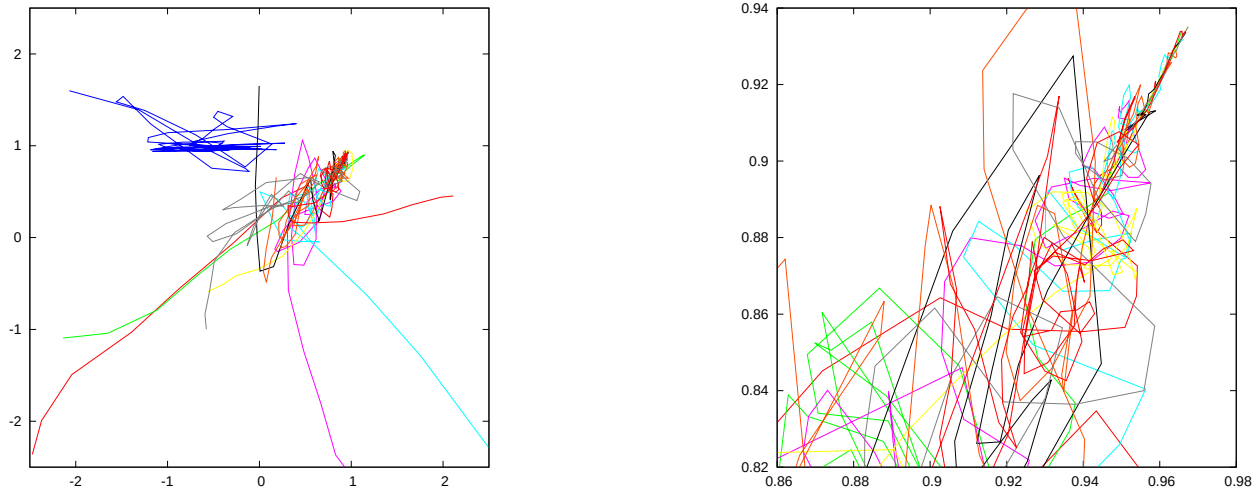


Potencjał Fahy-Hamana

$$V(x,y) = \frac{\sin(2\pi x)}{2\pi x} + \frac{\sin(2\pi y)}{2\pi y} + \frac{(x^2+y^2)^2}{16\pi^2} \tag{5}$$

i kilka trajektorii uzyskanych w *Particle Swarm Optimization*. Widać jak trajektorie grupują się wokół czterech (równoważnych) minimów globalnych.

Przykład — funkcja Rosenbrocka



Kilka trajektorii uzyskanych w *Particle Swarm Optimization* zastosowanej do funkcji Rosenbrocka. Prawy panel pokazuje szczegóły okolic znalezionego kandydata na minimum. Widać jak także w tym algorytmie trajektorie powoli podążają wzdłuż dna doliny w stronę minimum. Prawdziwe minimum znajduje się w punkcie $(1, 1)$.