Modelarea și Simularea algoritmului *Simulated Annealing* în limbajele *Python*, *C++* și *Matlab*

Vlad-Mihai Vasilescu
Universitatea Politehnica din București, Facultatea de Electronică
Telecomunicații și Tehnologia Informației
Programul de Masterat Ingineria Informației și a Sistemelor de Calcul

***Abstract*— În această lucrare este prezentat algoritmul *Simulated Annealing*, aplicat pentru cazul concret de optimizare a unei funcții non-convexe. Sunt realizate o serie de simulări folosind limbajele *Python*, *C++* și *Matlab* pentru determinarea soluțiilor eficiente din punct de vedere al timpului de calcul, fiind adusă și o îmbunătățire în cazul utilizării *Python* prin integrarea unei funcții implementate în *C*.**

I. INTRODUCERE

Necesitatea eficientizării algoritmilor de optimizare pentru probleme non-convexe provine din creșterea popularității metodelor ce utilizează diferiți algoritmi pentru rezolvarea subproblemelor din cadrul unor aplicații complexe. Un exemplu în acest sens îl reprezintă antrenarea rețelelor neurale cu constrângeri, utilizată în cadrul metodelor de protecție împotriva atacurilor adversariale. Aceste tipuri de antrenare implică modificarea ponderilor la fiecare *batch* de date astfel încât să conducă atât la o valoare cât mai mică a funcției de cost, cât și la un set de valori care sa respecte anumite reguli impuse inițial. Acestea introduc timpi suplimentari relativ mari în cadrul procesului de antrenare față de cazul uzual, de unde nevoia de algoritmi eficienți pentru găsirea setului optim de parametrii curenți. Astfel, se caută soluții cât mai eficiente pentru aplicarea algoritmilor de optimizare în diferite probleme de interes. În raportul de față este prezentat algoritmul de optimizare *Simulated Annealing* și discutată eficiența de soluționare a unor probleme non-convexe prin diferite implementări ale acestui algoritm.

II. DESCRIEREA ALGORITMULUI

***Simulated Annealing* (SA)** [1] reprezintă un algoritm probabilistic de aproximare a optimului global pentru o funcție de interes. Fiind un algoritm probabilistic, acesta găsește doar o aproximare cât mai bună pentru soluția globală asociată unei probleme de optimizare, nu pe cea exactă – aproximare care în unele cazuri este de preferat în detrimentul utilizării unui algoritm exact, dar mult mai costisitor din punct de vedere computațional. Deși introdus pentru prima oară cu scopul rezolvării problemelor complexe de combinatorică, acesta a fost adaptat de-a lungul timpului și pentru optimizarea problemelor cu suport continuu [2]. Algoritmul se bazează pe conceptul de *răcire controlată* în domeniul metalurgiei pentru creșterea ductilității și reducerea durității materialelor, prin încălzirea urmată de răcirea controlată a acestora, cu scopul ajungerii într-o stare de energie liberă minimă. Astfel, scopul final este de a produce o configurație cu o energie asociată cât mai mică. Algoritmul SA funcționează prin analogie cu descrierea precedentă, pornind de la stări inițiale cât mai haotice (temperatură ridicată), în care pot fi acceptate chiar soluții cu o energie mai mare – cu scopul de explorare a spațiului soluțiilor – urmată de o scădere a temperaturii. Valoarea temperaturii înglobează gradul de libertate pentru soluția curentă, astfel încât la scăderea acesteia spațiul soluțiilor este restrâns probabilistic în jurul soluției globale de energie minimă. În continuare, pentru simplitate, va fi prezentată doar formularea pentru funcții cu suport continuu a algoritmului SA.

Fie $f : \mathbb{R}^N \mapsto \mathbb{R}^M$ funcția obiectiv cu optim global (minim) notat $\boldsymbol{x}^* = (x_1, x_2, \ldots, x_N)$. Se pune problema determinării unei apoximări $\hat{\boldsymbol{x}^*} = (\hat{x_1}, \hat{x_2}, \ldots, \hat{x_3})$ pentru acest optim, aproximare care respectă inegalitatea $\mathcal{L}^p(\boldsymbol{x}^*, \hat{\boldsymbol{x}^*}) < \epsilon$, unde $\mathcal{L}^p$ reprezintă distanța de normă p, p fiind ales pentru a defini tipul de distanță utilizat pentru evaluarea "apropierii" dintre cele 2 puncte multi-dimensionale (pentru $p = 2$ se obține distanța Euclidiană), iar ϵ reprezintă pragul de acceptare a soluției aproximative. În cazul de față se urmărește găsirea minimului global al funcției f, ceea ce permite asocierea valorii funcției f într-un anumit punct cu conceptul de *energie* din algoritmul SA. Algoritmul 1 prezintă pașii urmați de algoritmul SA pentru căutarea minimului unei funcții f.

În cazul acestui algoritm, căutarea se realizează fără cunoștințe a priori despre minimul global, singurul criteriu fiind atingerea numărlui maxim de pași de căutare N (sau atingerea temperaturii minime). De asemenea, un alt criteriu ce poate fi introdus este temperatura minimă permisă la care se poate ajunge. Cele două funcții utilizate în cadrul algoritmului, $g_V : \mathbb{R}^N \mapsto \mathbb{R}^N$ și $g_T : \mathbb{R} \times \mathbb{R} \mapsto \mathbb{R}$ sunt funcții care generează stocastic soluții vecine, respectiv determinist

Algorithm 1 Algoritmul SA

Intrare: $f, \boldsymbol{x_0}, N, T_0, T_{min}, \epsilon$
Ieșire: $\hat{\boldsymbol{x}^*}$

```
T_k ← T_0
x_k ← x_0
x̂* ← x_0
k ← 0
while k ≠ N and T_k ≥ T_min do
    y_k ← g_V(x_k)                          ▷ Generare soluție vecină
    if f(y_k) < f(x_k) then
        x_{k+1} ← y_k
    else
        if exp(−(f(y_k)−f(x_k))/T_k) > U(0,1)^(k) then
            x_{k+1} ← y_k
        else
            x_{k+1} ← x_k
        end if
    end if
    if f(x_{k+1}) < f(x̂*) then
        x̂* ← x̂_{k+1}                        ▷ Modificare optim curent
    end if
    T_{k+1} = g_T(T_0, k)                   ▷ Reducere temperatură
    k = k + 1
end while
```

temperatura la pasul curent, astfel:

$$g_V(x) = x + S, S \sim \mathcal{N}(0,1) \tag{1}$$

$$g_T(T_0, k) = T_0 \cdot 0.997^k \tag{2}$$

unde S reprezintă un vector aleator, de dimensiune N, cu eșantioane extrase dintr-o distribuție *Gaussiana* de medie 0 și varianță 1. După cum se poate observa, funcția de determinare a temepraturii curente prezintă o scădere exponențială în timp, astfel încât odată cu creșterea numărului de iterații temperatura va scădea din ce în ce mai puțin, permițând astfel algoritmului să realizeze o căutare cât mai *fină* în vecinătatea de soluții la care s-a ajuns după iterațiile inițiale.

Algoritmul SA permite astfel optimizarea unor probleme cu multiple minime locale prin mișcări aleatoare chiar și în puncte cu o energie mult mai mare decât ceea ce a vizitat precedent, lucru aproape imposibil pentru un optimizator bazat pe utilizarea gradienților funcției obiectiv în cadrul etapei de modificare a soluției curente, cum este de fel metoda gradientului descendent (*gradient descent* [3]).

III. PLATFORMA COMPUTAȚIONALĂ

Toate experimentele au fost realizate pe o stație de lucru echipată cu un procesor Intel Xeon CPU E5-1680 v3, cu 8 nuclee și o frecvență de bază a procesorului de 3.2 GHz. Stația de lucru dispune, de asemenea, de o memorie RAM de 192 GB.

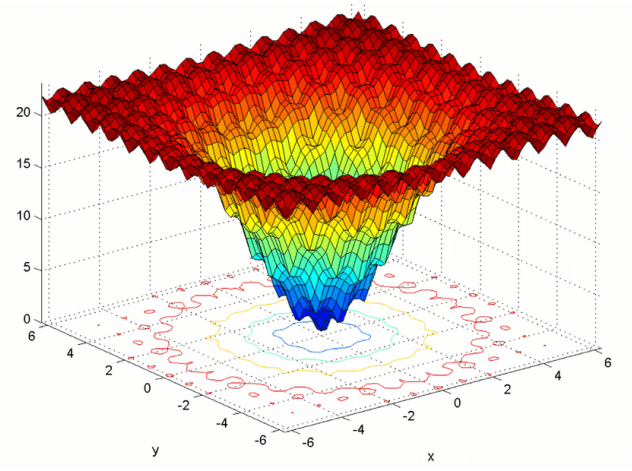

Fig. 1. Graficul funcției *Ackley* bidimensionale [4]

IV. MODELARE ȘI SIMULARE

A. Limbajele utilizate

Implementarea algoritmului *Simulated Annealing*, prezentat anterior, a fost realizată prin intermediul limbajelor *Python*, *C++* și *Matlab*, descrise pe scurt în continuare:

- *Python* reprezintă un limbaj de programare de nivel înalt interpretat – instrucțiunile sunt executate direct de un program adițional denumit *interpretor*, fără nevoia traducerii acestora în cod mașină – utilizat pentru aplicații de uz general; Deși prezintă un grad ridicat de popularitate datorită flexibilității și lizibilității codului, plasându-l pe primele poziții în topul limbajelor utilizate pentru aplicații de Inteligență Artificială, acesta conduce la o eficiență mai scăzută față de alte limbaje compilate, datorită necesității interpretării instrucțiunilor la fiecare rulare a programelor;
- *C++* reprezintă un limbaj de programare de uz general compilat, creat sub forma unei extensii a limbajului C; acesta se remarcă prin performanțe foarte bune ca timp de calcul, dar și prin flexibilitatea utilizării acestuia, datorită multiplelor paradigme de programare pe care le suportă;
- *Matlab (MATrix LABoratory)* reprezintă un limbaj multi-paradigmă pentru calcul numeric științific, utilizat într-o varietate de domenii datorită existenței multiplelor pachete create pentru aplicații specifice, printre care și *Simulink*; acesta se caracterizează prin eficientizarea funcțiilor incorporate, cele mai multe funcții de bază fiind implicit executate pe mai multe fire de execuție, precum și operații de indexare a structurilor multi-dimensionale relativ rapide;

B. Alegerea funcției obiectiv

Algoritmul SA a fost aplicat în cadrul problemei de minimizare a unei funcții obiectiv, în cazul de față funcția

Ackley bidimensională [5] (Figura 1). Forma analitică a acestei funcții este:

$$f(x_0, x_1) = T_1(x_0, x_1) + T_2(x_0, x_1) + 20 + e \quad (3)$$

$$T_1(x_0, x_1) = -20 \cdot exp(-0.2\sqrt{0.5(x_0^2 + x_1^2)}) \quad (4)$$

$$T_2(x_0, x_1) = -exp(0.5(cos(2\pi \cdot x_0) + cos(2\pi \cdot x_1)) \quad (5)$$

Forma analitică a acestei funcții o încadrează în setul funcțiilor non-convexe, având o multitudine de minime locale de gradient 0, fiind utilizată pentru testarea diferiților algoritmi de optimizare.

C. Simulare

Toate functiile cu implementare proprie în *Python* se bazează pe calcule realizate folosind librăria *Numpy*, specifică pentru calcule matematice pentru mulțimi de date multi-dimensionale. Există 3 funcții implementate ce intră în componența programului de simulare: funcția *ackley(...)*, care returnează valoarea funcției pentru un vector de intrare bidimensional, funcția *get_neighbours(...)* prin care sunt generați noi vecini pentru soluția curentă, și funcția *SA(...)* ce înglobează întregul algoritm de optimizare.

Un instrument util pentru analiza amănunțită a performanțelor unui program (sau porțiuni de program) *Python* este reprezentat de *Python Profilers*[1], ce oferă un set de statistici cu privire la timpii de execuție pentru porțiunile individuale de cod. Utilizând acest instrument, s-au analizat timpii pentru funcțiile individuale ce intră în componența simulării algoritmului. S-a observat astfel că un timp de aproximativ 2.3 secunde este pierdut pentru apelurile funcției *ackley()*. S-a urmărit în continuare diminuarea acestui inconvenient prin utilizarea unei alternative mai eficiente pentru calculul valorilor funcției respective.

S-a decis utilizarea librăriei *ctypes*[2] pentru calculul valorilor funcției prin intermediul unei librării în format *.dll*, construită pe baza unui program C ce conține implementarea funcției *Ackley*. Etapele de integrare a aceste funcții în algoritmul scris în *Python* sunt următoarele:

- crearea programului C ce conține funcționalitatea dorită
- compilarea acestui program sub forma unui fișier *.dll* (pentru sistemul de operare *Windows 10*)
- utilizarea librăriei Python *ctypes* pentru încărcarea fișierului creat și transmiterea datelor stocate într-un *numpy.array* prin intermediul unui pointer de același tip cu cel specificat în definiția funcției C
- apelul funcției C prin instrucțiuni *Python* și extragerea rezultatului printr-un pointer

O analiză a programului ce se folosește de funcția C indică o îmbunătățire a timpului necesar pentru calculul valorilor funcției *ackley* în diferite puncte, de la 2.3 secunde la 1.4 secunde. Acest lucru poate fi observat în tabelul I, pentru metodele *Python* și *Python + Ctypes*, pentru valoarea timpului mediu de simulare. Distribuția timpilor de execuție pentru cele 2 tipuri de simulări, pentru un număr de 100 de experimente, poate fi vizualizată în figurile 2 a), b) pentru programul inițial, respectiv programul ce utilizează funcția C.

Metodă	Medie timp	Disp. timp	Dist. medie
Python	6.0659	0.2390	0.0108
Py + ctypes	5.4339	0.1939	0.0117
Scipy	5.2429	0.9199	**1.2E-9**
C++	**0.1079**	**0.0094**	0.0109
Matlab	0.2974	0.1116	0.0774

TABLE I
STATISTICI PENTRU TIMPII DE EXECUȚIE ȘI SOLUȚIILE OBȚINUTE PENTRU UN NUMĂR DE 100 DE SIMULĂRI

Pentru comparație, s-a realizat simularea algoritmului prin intermediul unei funcții deja implementate, înglobată în librăria *SciPy*[3] aferentă limbajului *Python*. Pentru simularea folosind *Scipy* numărul maxim de iterații a fost setat același cu cel din cadrul simulărilor folosind implementările proprii, pentru o comparație cât mai corectă. Distribuția acestor timpi de execuție poate fi vizualizată în figura 2 c).

Implementarea în limbajul *C++* o urmărește îndeaproape pe cea din limbajul *Python*, cu obiectivul unei comparații cât mai corecte pentru timpii de execuție. Distribuția acestor timpi poate fi vizualizată în figura 2 d).

Pentru simularea folosind limbajul *Matlab* a fost utilziată funcția *simulannealbnd(...)* din cadrul *toolbox*-ului *Global Optimization Toolbox*[4]. Aceasta utilizează anumite funcționalități pentru algoritmul de față, dar permite utilizarea unor funcții proprii pentru modul în care temperatura este redusă, precum și pentru modul de generare a vecinilor soluției curente. Astfel, au fost implementate funcțiile *tempHandle(...)* și *generatePoints(...)* pentru a înlocui implementările implicite pentru cele 2 funcționalități, acestea urmărind implementările indicate în ecuațiile 1 și 2.

Performanțele algoritmului din punctul de vedere al găsirii unei soluții cât mai apropiate de cea reală depind puternic de temperatura inițială, modul de generare al vecinilor și de procedura prin care temperatura este redusă de-a lungul iterațiilor. Funcțiile de generare a vecinilor și de reducere a temperaturii utilizate (mai puțin pentru cazul utilizării *SciPy*) sunt cele prezentate în ecuațiile 1 și 2. Temperatura inițială a fost setată la 1000, iar numărul de iterații maxime la aprox.

[1] https://docs.python.org/3/library/profile.html#
[2] https://docs.python.org/3/library/ctypes.html
[3] https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.optimize.anneal.html
[4] https://www.mathworks.com/products/global-optimization.html

4600, corespunzând unei temperaturi finale de aprox. 10^{-3}.

Tabelul I conține media și dispersia timpului de simulare, precum și distanța medie dintre soluțiile obținute și minimul global al funcției *Ackley*, pentru cele 5 tipuri de simulări discutate anterior. Se poate observa faptul că utilizarea limbajului *C++* prezintă cel mai mic timp de simulare (urmat de limbajul *Matlab*), precum și cea mai mică dispersie pentru acest timp. Din punctul de vedere al distanței medii față de minimul real, funcționalitatea din *SciPy* prezintă cea mai mică apropiere, mult superioară față de celelalte implementări ce prezintă performanțe similare din acest punct de vedere. Acest lucru se datorează în principal modului în care este realizată generarea vecinilor și a modului în care temperatura este redusă de-a lungul iterațiilor, precum și faptului că funcția respectivă returnează soluția care a prezentat cea mai mică valoare pentru funcția obiectiv de-a lungul iterațiilor, și nu soluția finală precum celelalte implementări. De remarcat totuși faptul că această metodă prezintă o dispersie pentru timpul de simulare relativ mare față de celelalte metode, fiind, cel mai probabil, puternic dependentă de punctul de start al algoritmului.

Pentru continuarea studiului comparativ au fost realizate experimente cu cele 5 metode de simulare pentru încă 3 funcții obiectiv utilizate pentru testarea algoritmilor de optimizare: *Himmelblau*[6], *Styblinski–Tang* și *Rastrigin*[7]. Acestea pot fi vizualizate în Figura 3. Rezultatele obtinute pentru timpul mediu de simulare (secunde) și dispersia timpului de simulare sunt prezentate în tabelul II. Se observă cu ușurință faptul că simularea implementării în *C++* obține rezultate net superioare față de celelalte metode.

De remarcat pentru cazul optimizării funcției *Himmelblau* faptul că timpul mediu pentru implementarea *Python* este mai redus decât cel în cazul utilizării *Ctypes* pentru evaluarea funcției obiectiv, ceea ce contravine cu așteptările inițiale. Acest lucru poate fi explicat prin simplitatea analitică a funcției respective, care nu reprezintă decât suma a două polinoame de ordinul 4, câte unul pentru fiecare dimensiune a spațiului de intrare.

Astfel, simplitatea funcției respective determină ca timpul efectiv de calcul să fie relativ mai mic sau comparabil cu timpii necesari transmiterii *numpy array*-urilor prin intermediul pointerilor, apelul funcției și extragerea rezultatului prin alt pointer. Deci, în acest caz, utilizarea *Ctypes* nu reprezintă o variantă mai bună decât apelul funcției scrisă în cod *Python*, a cărui durată de apel este mai redusă decât apelul funcției scrisă în *C*. În cazul celorlalte 2 funcții se observă contrariul, anume utilizarea *Ctypes* rezultă în timpi medii de execuție mai mici, în aceste cazuri funcțiile având o complexitate mai mare, timpii de evaluare având de câștigat la utilizarea unei funcții scrise în cod *C*.

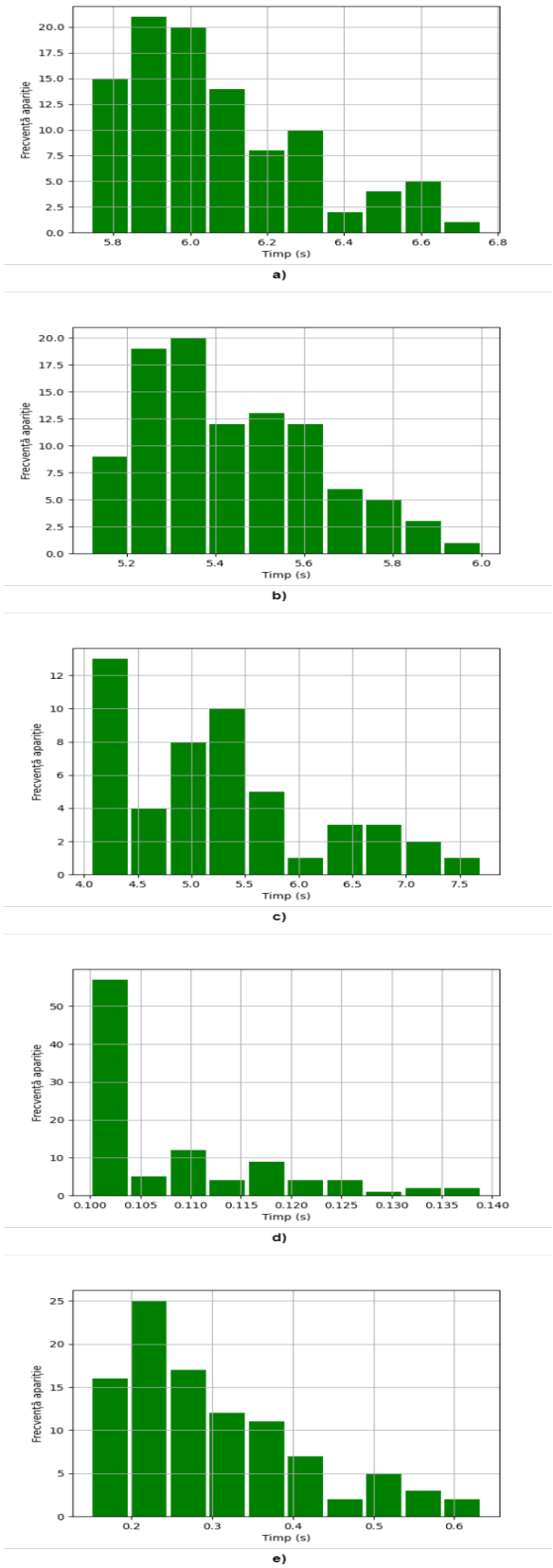

Fig. 2. Distribuția timpilor de execuție pentru optimizarea funcției *Ackley* pentru cele 5 cazuri: a) Implementare *Python*, b) Implementare *Python* folosind *Ctypes*, c) Funcție *Scipy*, d) Implementare *C++*, e) Implementare *Matlab*

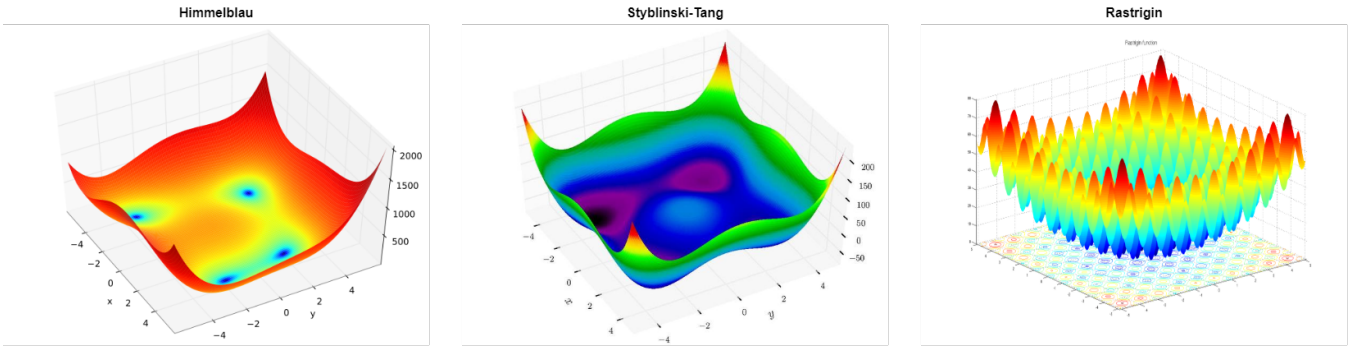

Fig. 3. *
Funcțiile utilizate pentru simulare. Imagini preluate din
https://en.wikipedia.org/wiki/Test_functions_for_optimization

	Himmelblau		**Styblinski-Tang**		**Rastrigin**	
Metodă	Medie	Dispersie	Medie	Dispersie	Medie	Dispersie
Python	4.7096	0.2022	6.5054	1.0236	5.8631	0.9957
Python + Ctypes	5.0265	0.1144	5.2389	0.2877	5.2304	0.1876
SciPy	1.4270	0.1070	1.3992	**0.0895**	1.5414	0.1028
C++	**0.1000**	**0.0107**	**0.0710**	0.1129	**0.0611**	**0.0127**
Matlab	0.2897	0.0858	0.2755	0.0990	0.2886	0.1071

TABLE II
PERFORMANȚELE CA TIMPI DE EXECUȚIE PENTRU CELE 5 TIPURI DE SIMULĂRI APLICATE ÎN OPTIMIZAREA CELOR 3 FUNCȚII, PENTRU UN NUMĂR DE 100 SIMULĂRI

Față de cazul optimizării funcției *Ackley*, se observă timpi medii semnificativ mai mici pentru implementarea folosind *SciPy*. Acest fapt se datorează cu totul formei funcției de optimizare, parametrii inițiali fiind constanți de-a lungul simulărilor pe diferite funcții, întrucât funcția *Ackley* prezintă cea mai "complexă" formă analitică dintre funcțiile prezentate datorită numărului mare de optime locale.

V. CONCLUZII

Simulările prezentate indică faptul că utilizarea unui limbaj compilat precum *C++* rezultă în performanțe net superioare din punctul de vedere al timpilor de execuție pentru algoritmul de optimizare prezentat, afirmație susținută prin intermediul rezultatelor prezentate în tabelele I și II. Evident, dacă se ia în considerare și timpul aferent procesului de compilare nu se mai poate afirma același lucru. Eficiența este măsurată în acest caz prin intermediul timpului de execuție al algoritmului respectiv, nu și prin timpul necesar traducerii codului de limbaj înalt în cod mașină.

O îmbunătățire ce poate fi adusă programelor scrise într-un limbaj relativ încet ca timp de execuție precum *Python* se rezumă la utilizarea unor funcționalități scrise într-un limbaj compilat, mult mai rapid, precum *C/C++*. Astfel, se pot integra diferite secvențe de cod scrise în *C* ce pot accelera execuția programului *Python* rezultat, ținând totuși cont de complexitatea secvenței respective de cod. Integrarea unor astfel de funcționalități poate introduce totuși timpi suplimentari datorați necesității convertirii datelor de un anumit tip *Python* în tipul corespunzător din limbajul *C*, precum și timpii necesari transferului de date prin intermediul diferiților pointeri. S-a constat că pentru o funcție de complexitate redusă utilizarea unui cod echivalent *C* nu aduce îmbunătățiri timpului de execuție, ba din contră, reduce performanțele algoritmului.

REFERENCES

[1] Kirkpatrick, Scott, C. Daniel Gelatt, and Mario P. Vecchi. "Optimization by simulated annealing." science 220.4598 (1983): 671-680.
[2] Dekkers, Anton, and Emile Aarts. "Global optimization and simulated annealing." Mathematical programming 50.1 (1991): 367-393.
[3] Robbins, Herbert E.. "A Stochastic Approximation Method." Annals of Mathematical Statistics 22 (2007): 400-407.
[4] Razavi, F., & Jalali-Farahani, F. (2008, June). Ant colony optimization: a leading algorithm in future optimization of petroleum engineering processes. In International Conference on Artificial Intelligence and Soft Computing (pp. 469-478). Springer, Berlin, Heidelberg.
[5] Back, T. (1996). Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms. Oxford university press.
[6] Himmelblau, D. M. (2018). Applied nonlinear programming. McGraw-Hill.
[7] Rastrigin, L. A. (1974). Systems of extremal control. Nauka.