



## **Evaluarea performanțelor algoritmului Particle Swarm Optimization pentru minimizarea unei funcții**

### **Profesor:**

Ş. L. Dr. Ing. Tiberius Dumitriu

### **Studenți:**

Curpăn Robert-Gabriel  
Istrate Sebastian  
Paraschiv Ștefan  
Scînteie Gabriel-Alexandru

## **Cuprins:**

1. Enunțarea temei
2. Arhitectura generală
3. Funcționalitate
4. Rolul fiecărui membru al echipei
5. Complexitatea algoritmului
6. Demonstrația corectitudinii
7. Testarea algoritmului
8. Explicații suplimentare
9. Concluzie

## 1. Enunțarea temei

Tema proiectului este utilizarea algoritmului **Particle Swarm Optimization**(PSO) în forma sa standard pentru a găsi minimul funcției  $f: R^5 \rightarrow R$ ,  $f(x) = \sum_{i=1}^5 x(i)^2$  cu o precizie aleasă de utilizator prin intermediul numărului maxim de iterații pe care să le realizeze algoritmul.

## 2. Arhitectura generală

Arhitectura aplicației este compusă dintr-un modul ce asigură funcționalitatea algoritmului și un modul de test ce verifică funcționarea corectă a algoritmului în conformitate cu precizia dorită.

Arhitectura internă a modulului principal este în strânsă legătură cu principiul algoritmului, având clasa **Particle** care conține informații legate de poziția particulei în spațiul problemei, cât și despre valoarea funcției în acel punct, viteza acesteia pe fiecare dimensiune, viteza maximă admisă pe fiecare dimensiune și costul cel mai bun pe care l-a avut particula până în acel moment de timp.

```
class Particle:
    def __init__(self):
        self.position = [random.uniform(inputLowerLimits[i], inputUpperLimits[i]) for i in
                        range(inputDimension)] # Setam pozitia initiala a particulei
        self.cost = f(copy(self.position)) # Setam valoarea functiei pentru particula
        self.speed = [0 for _ in range(inputDimension)] # Setam viteza particulei pe fiecare dimensiune
        self.maxSpeeds = [alfa * (inputUpperLimits[i] - inputLowerLimits[i]) for i in
                          range(inputDimension)] # Setam viteza maxima pe fiecare dimensiune
        self.personalOptim = copy(self.position)
```

Tot în acest modul se află și funcția care implementează algoritmul PSO, aceasta primind ca parametrii variabilele ce influențează probabilitatea de succes precum numărul de iterații, numărul de particule, valoarea parametrului de inerție, coeficientul de încredere în sine și coeficientul de încredere în vecini.

```
def particleSwarmOptimization(f, noIterations, noParticles, w, c1, c2):
```

### 3. Funcționalitate

Algoritmul de Particle Swarm Optimization este o metodă de calcul inspirată din natură, atât din observarea mișcării particulelor, cât și din observarea organizării roiurilor de păsări migratoare. Acesta caută minimul unei funcții încercând să îmbunătățească o soluție candidat. El rezolvă problema având o populație de soluții candidate, numite particule, care se mișcă prin spațiul de căutare. Mișcarea fiecărei particule este influențată de cea mai bună poziție pe care a avut-o particula, dar este, de asemenea, ghidată către cele mai bune poziții cunoscute de către întreaga populație, care sunt actualizate pe măsură ce poziții mai bune sunt găsite de alte particule. Conform acestei euristică se așteaptă ca algoritmul să conveargă spre puncte de minim ale funcției.

```
swarm = initializeSwarm()
socialOptim = min(list(map(lambda x: x.position, swarm)), key=f)

for iteration in range(noIterations):
    for index in range(len(swarm)):
        r1 = random.random()
        r2 = random.random()

        # particle este o referință către swarm[index] deoarece în Python obiectele se transmit prin referință
        particle = swarm[index]

        for i in range(inputDimension):
            particle.speed[i] = limit(
                w * particle.speed[i] + \
                c1 * r1 * (particle.personalOptim[i] - particle.position[i]) + \
                c2 * r2 * (socialOptim[i] - particle.position[i]),
                -particle.maxSpeeds[i], particle.maxSpeeds[i])

            particle.position[i] = limit(particle.position[i] + particle.speed[i], inputLowerLimits[i],
                                         inputUpperLimits[i])

        particle.cost = f(particle.position)

        if f(particle.position) < f(particle.personalOptim):
            particle.personalOptim = copy(particle.position)
        if particle.cost < f(socialOptim):
            socialOptim = copy(particle.position)

# socialOptim = f(min(list(map(lambda x: x.position, swarm)), key=f))
return socialOptim, f(socialOptim)
```

Rezultatele cele mai bune ale algoritmului au avut o precizie de peste 25 de zecimale și s-au atins pentru următorii parametrii de intrare:

- $w = 0.4$
- $c1 = 1$
- $c2 = 2$
- $\text{noParticles} = 30$
- $\text{noIterations} = 200$

## 4. Rolul fiecărui membru al echipei

- Curpan Robert:
  - corectitudinea unui bloc din algoritm
  - documentatie: complexitatea algoritmului, demonstrația corectitudinii
- Istrate Sebastian
  - testarea automată a algoritmului
  - documentatie: explicații suplimentare, concluzii
- Paraschiv Stefan
  - calculul complexitatii
  - documentatie: testarea algoritmului
- Scîntei Gabriel-Alexandru
  - implementarea algoritmului
  - documentație: enunțarea temei, arhitectura generală, funcționalitate

## 5. Complexitatea algoritmului

Având în vedere că studiem un algoritm de natură evolutivă, nu există complexitate în caz favorabil sau nefavorabil. Vom enunța o singură complexitate, valabilă în cazul general.

Vom analiza ordinul de creștere a câtorva dintre funcțiile folosite în algoritm, iar la final vom calcula ordinul de creștere a întregului algoritm și vom enunța complexitatea.

$d$  = dimensiunea vectorului de intrare (numărul de parametri ai funcției)

$p$  = nr de particule

$g$  = nr de generații (iterații) ale algoritmului evolutiv

```

# Complexitate : O(1+d*3)
# Functie obiectiv
def f(x):
    sum = 0
    for elem in x:
        sum += elem * elem
    return sum

# Structuri de date

#O(d + 1 + d * 4 + d + d * 3 + d) = O(9*d)
class Particle:
    def __init__(self):
        self.position = [random.uniform(inputLowerLimits[i], inputUpperLimits[i]) for i in
                        range(inputDimension)] # Setam pozitia initiala a particulei
        self.cost = f(copy(self.position)) # Setam valoarea functiei pentru particula
        self.speed = [0 for _ in range(inputDimension)]
        # self.speed = [round(0.05 * random.uniform(inputLowerLimits[i], inputUpperLimits[i]), 3) for i in
        #               range(inputDimension)] # Setam viteza pe fiecare dimensiune
        self.maxSpeeds = [alfa * (inputUpperLimits[i] - inputLowerLimits[i]) for i in
                          range(inputDimension)] # Setam viteza maxima pe fiecare dimensiune
        self.personalOptim = copy(self.position)

O(2)
def limit(value, minim, maxim):
    return min(max(value, minim), maxim)

O(p * 9d)
def initializeSwarm():
    return [Particle() for _ in range(noParticles)]

```

```

## Algoritm

if __name__ == "__main__":
    swarm = initializeSwarm()
    socialOptim = min(list(map(lambda x: x.position, swarm)), key=f)

for iteration in range(iterationMax):
    for index in range(len(swarm)):
        r1 = random.random()
        r2 = random.random()

        # particle este o referinta catre swarm[index] deoarece in Python obiectele se transmit prin referinta
        particle = swarm[index]

        for i in range(inputDimension):
            particle.speed[i] = limit(
                w * particle.speed[i] + \
                c1 .position[i]),
                -particle.maxSpeeds[i], particle.maxSpeeds[i])* r1 * (particle.personalOptim[i] - particle.position[i]) + \
                c2 * r2 * (socialOptim[i] - particle
    
```

OPERATIA DOMINANTA

```

            particle.position[i] = limit(particle.position[i] + particle.speed[i], inputLowerLimits[i],
                                           inputUpperLimits[i])

            particle.cost = f(particle.position)

            if f(particle.position) < f(particle.personalOptim):
                particle.personalOptim = copy(particle.position)
            if particle.cost < f(socialOptim):
                socialOptim = copy(particle.position)
    
```

Complexitatea finală este:  $O(g * p * [3 + 2d + 1 + 3d] + 2 + 2d) \Rightarrow O(g * p * d)$

Obs:  $d =$  nr de parametri ai funcției. În cazul acestui algoritm,  $d = 5 =$  constant (se studiază o anumită funcție de 5 parametri), deci este neglijabil în cazul complexității

$\Rightarrow O(g * p)$

## 6. Demonstratia corectitudinii

Operația dominantă este cea care se executa sub (while  $i < inputDimension$ ) - se executa de aproximativ  $g * p * d$  ori.

## Demonstratie cotaștiruire a unui bloc important din algoritm

$P \equiv P_0$

iteration = 0 A<sub>1</sub>

P<sub>1</sub> while iteration < iterationMax:

P<sub>2</sub> & C<sub>1</sub> index = 0 A<sub>2</sub>

P<sub>2</sub> while index < len(swarm):

P<sub>2</sub> & C<sub>2</sub> k<sub>1</sub> = random, random() A<sub>3</sub>

P<sub>3</sub> k<sub>2</sub> = random, random() A<sub>4</sub>

P<sub>4</sub> particle = swarm[index] A<sub>5</sub>

P<sub>5</sub> i = 0 A<sub>6</sub>

P<sub>6</sub> while i < inputDimension:

P<sub>6</sub> & C<sub>3</sub> particle · speed[i] = ...; A<sub>7</sub>

P<sub>7</sub> particle · position[i] = ...; A<sub>8</sub>

P<sub>8</sub> L += 1 A<sub>9</sub>

P<sub>9</sub> particle · cost = f(particle · position) A<sub>10</sub>

P<sub>11</sub> if f(particle · position) < f(particle · personalOptim):

P<sub>12</sub> & C<sub>4</sub> particle · personalOptim = copy(particle · position) A<sub>11</sub>

P<sub>12</sub> & C<sub>5</sub> if particle · cost < f(socialOptim):

P<sub>13</sub> & C<sub>5</sub> socialOptim = copy(particle · position) A<sub>12</sub>

P<sub>14</sub> index += 1 A<sub>13</sub>

P<sub>14</sub> iteration += 1 A<sub>14</sub>

Q  $\equiv P_{15}$

### 1) Precondiții

- Iteration Max  $\geq 1$
- Len(swarm)  $\geq 1$
- inputDimension  $\geq 1$
- Swarm = Array < Particle >

### 2) Postcondiții

- Social Optimum = poziția primei cărei a trecut una sau mai multe particule de-a lungul celei "iterationMax" iterații, pt. căre  $f(position) = \min$

$$f(Social\ Optimum) = \min \text{ pt } \{Swarm[0], \dots, Swarm[l] \text{ } | \text{ } l = 0, \dots, \overline{IterationMax} - 1\}$$

In contextul problemelor noastre, Social Optimum va conține coordonatele punctului în care funcția pe care o studiem ia valoarea minimă.

### 3) Adnotare program

### 4) Verificarea validității condițiilor din bucle

- Bucle 1 :  $iteration = 0$   
 $iteration < iterationMax$   
 $iteration \geq 1$  }  $\Rightarrow$  Bucle se exec. măcar o dată  $\Rightarrow$  bucle și validate

- Bucle 2 :  $index = 0$   
 $index < len(swarm)$   
 $len(swarm) \geq 1$  }  $\Rightarrow$  Bucle se exec. măcar o dată  $\Rightarrow$  bucle și validate.

Analog pt. bucla 3.

### 5) Determinare invariante bucle

$J_1$ :  $f(\text{socialOptim}) = \min \text{ pt } \{ \text{swarm}^m[0], \dots, \text{swarm}^m[\text{len}(\text{swarm})-1] \}, \forall m \in \{0, \dots, \text{iteration}\}$

$J_2$ :  $f(\text{socialOptim}) = \min \text{ pt } \{ \text{swarm}^m[0], \dots, \text{swarm}^m[\text{index}] \}, \forall m \in \{0, \text{iteration}\}$

Pt. bucla 3 nu avem un invariant deoarece nu e relevantă pt determinarea postcondiției (nu se iterează prin particule proprie-zise; în acestă buclă doar se fac mutări actualizări abrupte uneia din particule).

### 6) Demonstrația invariantei prin inducție

$\bullet J_2$

$P(n)$ :  $f(\text{socialOptim}) = \min \text{ pt } \{ \text{swarm}^m[0], \dots, \text{swarm}^m[n-1] \}, \forall m \in \{0, \text{iteration}\}$

Dacă în iteratărea curentă nu se găsește un nou minim, atunci demonstrația s-a încheiat (condiția e respectată din start). Altfel, dacă în iteratărea curentă se găsește un nou minim, atunci vom demonstra prin inducție că:

$f(\text{socialOptim}) = \min \text{ pt } \{ \text{swarm}^{\text{iteration}}[0], \dots, \text{swarm}^{\text{iteration}}[n-1] \}$

I)  $P(1)$ : avem un singur element  $\text{swarm}[0] \Rightarrow \text{socialOptim} = \text{swarm}[0].\text{position} =$   
 $\Rightarrow f(\text{socialOptim}) = \min$ .

II)  $P(k)$ :  $f(\text{socialOptim}) = \min \text{ pt } \{ \text{swarm}[0], \dots, \text{swarm}[k-1] \}$

$P(k+1)$ :  $f(\text{socialOptim}) = \min \text{ pt } \{ \text{swarm}[0], \dots, \text{swarm}[k] \}$

Dacă  $f(\text{socialOptim}) \leq f(\text{swarm}[k].\text{position}) \Rightarrow$  nu e nevoie să se actualizeze  $\text{socialOptim}$  că  $f(\text{socialOptim})$  e doar minim pt. particulele  $0 \dots k$ .

Dacă  $f(\text{Social Optimal}) > f(\text{vector}[h]. \text{position}) \Rightarrow$  se intră în al doilea if și se face actualizarea Social Optimal = vector[ $h$ ]. position  $\Rightarrow$  se găsește un nou minim  $\Rightarrow$   
 $\Rightarrow f(\text{Social Optimal}) = \text{MINIM pt. partic. } o \dots h$ .

Din I) și II)  $\stackrel{\text{P.i.M}}{\Rightarrow} P(n) : A$

• J1

$P(n) : f(\text{Social Optimal}) = \text{MINIM pt. } \left\{ \underbrace{\text{vector}^m[0], \dots, \text{vector}^m[\text{len}(\text{vector}) - 1]}_{+}, \forall m \in \{0, \text{iterație}\} \right\}$

I)  $P(1) : f(\text{Social Optimal}) = \text{MINIM după prima iterare} \rightarrow$  acest lucru e sigur că de  
bucă 2 pt că am demonstrat invariantele.

II)  $P(h) : f(\text{Social Optimal}) = \text{MINIM pt. } \left\{ \underbrace{\text{vector}^m[0], \dots, \text{vector}^m[\text{len}(\text{vector}) - 1]}_{+}, \forall m \in \{0, h\} \right\}$

$P(h+1) : f(\text{Social Optimal}) = \text{MINIM pt. } \left\{ \underbrace{\text{vector}^m[0], \dots, \text{vector}^m[\text{len}(\text{vector}) - 1]}_{+}, \forall m \in \{0, h+1\} \right\}$

Invariantele celei de-a două bucle (pe care le-am demonstrat) ne obligă să la  
finalul buclei, dacă există un nou minim global în iterarea curentă ( $h+1$ ), atunci  
Social Optimal va fi actualizat cu acea valoare. Atât, va rămâne valoarea veche că  
era deja minimul pt. iterările  $o \dots h$ , devenind astfel minim pt. iterările  
 $o \dots h+1$ .

Nodat, e valabilă și demonstrată și condiția acestui invariante.

7) Demonstrarea faptului că buclele se termină

Funcția de terminare a primei bucle: ~~F(0)~~  $F(\text{iteration}) = \text{iterationMax} - 1 - \text{iteration}$   
↳ iteratia crește tot timpul

$$\begin{aligned} F(\text{iteration}) > F(\text{iteration}+1) &\Rightarrow \text{iterationMax} - 1 - \text{iteration} > \text{iterationMax} - 2 - \text{iteration} \\ &\Rightarrow 1 > 0 \quad \text{(A)} \end{aligned}$$

$\Rightarrow$  F-descrez.      }  $\Rightarrow$  bucla se termină.  
 $F(\text{iterationMax}-1) = 0$

Analog pt celelalte 2 bucle.

## 7. Testarea algoritmului

Pentru a verifica corectitudinea algoritmului se realizează 25 de teste, din care, în medie, 16 teste sunt cu rezultat pozitiv, iar 9 cu rezultat negativ. Testele derivă dintr-un caz general, anume când inerția este 0.4, sunt 30 de particule, coeficientul de încredere în sine este 1 , coeficientul de încredere în vecini este 2 și sunt 100 de iterații. Acest caz se testează cu alte 5 intrări și pentru a face testarea completa se schimbă cale un parametru pe rand din cazul general cu parametrul corespunzător din setul de date cu care este comparat. Testele care nu trec au parametrii care nu conduc la o precizie destul de buna.

Cele 5 seturi de date și setul de date general:

```
data_array = [Data(30, 10, 1, 1, 0.4),
              Data(50, 20, 2, 2, 0.6),
              Data(80, 30, 4, 4, 0.7),
              Data(100, 40, 6, 6, 0.9),
              Data(200, 60, 5, 5, 0.5)]

default = Data(100, 30, 1, 2, 0.4)

for data in data_array:
    test(data.w, default.c1, default.c2, default.noParticles, default.iterationMax, precision)
    test(default.w, data.c1, default.c2, default.noParticles, default.iterationMax, precision)
    test(default.w, default.c1, data.c2, default.noParticles, default.iterationMax, precision)
    test(default.w, default.c1, default.c2, data.noParticles, default.iterationMax, precision)
    test(default.w, default.c1, default.c2, default.noParticles, data.iterationMax, precision)
```

Rezultatele testării:

```
Succes: w=0.4 c1=1 c2=2 noParticles=30 iterationMax=100 calc = 9.65425e-20
Succes: w=0.4 c1=1 c2=2 noParticles=30 iterationMax=100 calc = 2.57856e-20
Succes: w=0.4 c1=1 c2=1 noParticles=30 iterationMax=100 calc = 1.3145404073305658e-08
Succes: w=0.4 c1=1 c2=2 noParticles=30 iterationMax= 30 calc = 9.991238057992199e-05
Succes: w=0.6 c1=1 c2=2 noParticles=30 iterationMax=100 calc = 1.66243366384002e-11
Succes: w=0.4 c1=2 c2=2 noParticles=30 iterationMax=100 calc = 8.7225595429e-14
Succes: w=0.4 c1=1 c2=2 noParticles=30 iterationMax=100 calc = 2.27756e-19
Succes: w=0.4 c1=1 c2=2 noParticles=30 iterationMax= 50 calc = 2.8541005961171423e-09
Succes: w=0.7 c1=1 c2=2 noParticles=30 iterationMax=100 calc = 1.648598936414582e-07
Succes: w=0.4 c1=1 c2=2 noParticles=30 iterationMax=100 calc = 4.1616e-21
Succes: w=0.4 c1=1 c2=2 noParticles=30 iterationMax=100 calc = 2.682876e-17
Succes: w=0.4 c1=1 c2=2 noParticles=40 iterationMax=100 calc = 7.374e-22
Succes: w=0.4 c1=1 c2=2 noParticles=30 iterationMax=100 calc = 2.2593e-21
Succes: w=0.5 c1=1 c2=2 noParticles=30 iterationMax=100 calc = 2.4837298e-18
Succes: w=0.4 c1=1 c2=2 noParticles=60 iterationMax=100 calc = 2.5e-24
Succes: w=0.4 c1=1 c2=2 noParticles=30 iterationMax=200 calc = 0.0
Failed: w=0.4 c1=1 c2=2 noParticles=10 iterationMax=100 calc = 0.06394
Failed: w=0.4 c1=1 c2=2 noParticles=20 iterationMax=100 calc = 0.00016
Failed: w=0.4 c1=4 c2=2 noParticles=30 iterationMax=100 calc = 0.00229
Failed: w=0.4 c1=1 c2=4 noParticles=30 iterationMax=100 calc = 0.01286
Failed: w=0.9 c1=1 c2=2 noParticles=30 iterationMax=100 calc = 1.19562
Failed: w=0.4 c1=6 c2=2 noParticles=30 iterationMax=100 calc = 0.37532
Failed: w=0.4 c1=1 c2=6 noParticles=30 iterationMax=100 calc = 0.43708
Failed: w=0.4 c1=5 c2=2 noParticles=30 iterationMax=100 calc = 0.4506
Failed: w=0.4 c1=1 c2=5 noParticles=30 iterationMax=100 calc = 0.02531
-----
Successful tests: 16 / 25
```

## 8. Explicații suplimentare

Evaluarea corectitudinii algoritmului a fost realizată în două moduri. Prima data a fost construit algoritmul și s-a demonstrat experimental că poate acoperi anumite clase de cazuri. Testarea a fost făcută prin construirea a 25 teste cu parametri diferiți care să încerce să acopere cât mai multe postcondiții. Pentru a arăta că acoperă întreaga plajă de valori a precondițiilor, s-a urmărit și demonstrația formală prin evidențierea blocului principal cu proprietăți invariante.

In cazul teoretic, s-a ajuns la concluzia că operația dominantă care determină clasa de complexitate este cea de calculare a vitezei particulelor, care este imbricată în 3 bucle for. Având în vedere că  $d = 5$  este considerat o constantă, ea va fi eliminată din calculul complexității generale.

Algoritmul un depinde de dimensiunile problemei, deci avem complexitate generală. Nu este necesar să calculăm clase de complexitate care depind de marginile inferioară și superioară.

## **9. Concluzii**

Cu ajutorul tehniciilor de verificare a corectitudinii unui algoritm, proiectul a reușit să demonstreze buna funcționare a optimizării roiului de particule (eng. Particle Swarm Optimization) pentru minimizarea funcției date. Prima dată s-a urmărit testarea pe cale experimentală cu ajutorul a 25 teste. În acest caz algoritmul a acoperit o plajă extinsă de valori dar nu a putut acoperi toate cazurile posibile. De aceea, a doua modalitate de demonstrare a fost făcută formal prin adnotări, precondiții și postcondiții și a fost explicat cum, prin succesiunea unui număr finit de pași, precondițiile duc la postcondiții.