

PARÇACIK SÜRÜ OPTİMİZASYONU (PSO)

TOKAT GAZİOSMANPAŞA ÜNİVERSİTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ
Dr. Öğr. Üyesi Mahir Kaya
TOGU-CENG

Sürü Zekası

Doğada bulunan hayvan ve böcek sürülerinin davranışları uzun yillardır insanların dikkatini çekmiştir. Kuş sürülerinin farklı şekiller alarak havada süzülmesi, karıncaların yiyecek ararken izledikleri yol, balık sürülerinin beraberce yüzmesi ve yaptıkları hareketler bu sürü davranışlarından sadece birkaçıdır. Son yillardaki çalışmalar, yapay yaşam alanı kapsamı altında, bu sürü davranışlarının nasıl modellenebileceği ve sürünen bireyleri arasındaki iletişim mantığı üzerine yapılmaktadır.

Günümüzde problemlerin çözümünde kullanılan ve biyolojik sistemlerden esinlenilerek ortaya konmuş birçok yöntem vardır. Örneğin yapay sinir ağları insan beyninin basitleştirilmiş bir modelidir.



Sürü Zekası

- Bazen tek başına hiçbir iş yapamayan varlıklar, toplu hareket ettiklerinde çok zekice davranışlar sergileyebilmektedir. Bir topluluğa ait bireyler, en iyi bireyin davranışından ya da diğer bireylerin davranışlarından ve kendi deneyimlerinden yararlanarak yorum yapmakta ve bu bilgileri ileride karşılaşacakları problemlerin çözümleri için bir araç olarak kullanmaktadır.
- Örneğin, bir canlı sürüsünü oluşturan bireylerden birisi bir tehlike sezdiğinde bu tehlikeye karşı tepki verir ve bu tepki sürü içinde ilerleyip tüm bireylerin tehlikeye karşı ortak bir davranış sergilemesini sağlar.  Canlıların sürüler halinde bir problemi kendi başlarına çözemeyip ortaklaşa ortaya koydukları gayretle çözme yeteneği sürü zekası olarak adlandırılmaktadır.

Sürü, birbirleriyle etkileşen dağınık yapılı bireyler yiğini anlamında kullanılır. Bireyler insan veya karınca olarak ifade edilebilir.

Sürülerde N adet temsilci bir amaca yönelik davranışı gerçekleştirmek ve hedefe ulaşmak için birlikte çalışmaktadır. Kolaylıkla gözlenebilen bu "kolektif zeka" temsilciler arasında sık tekrarlanan davranışlardan doğmaktadır.

Temsilciler faaliyetlerini idare etmek için basit bireysel kurallar kullanmakta ve grubun kalan kısmıyla etkileşim yolu ile sürü amaçlarına ulaşmaktadır.

Grup faaliyetlerinin toplamından bir çeşit kendini örgütleme doğmaktadır.

En çok bilinen sürü zekası optimizasyon algoritmalarından bazıları; Karınca Koloni Optimizasyonu, Parçacık Sürü Optimizasyonu ,Yapay Balık Sürüsü Algoritması, Bakteriyel Besin Arama Optimizasyon Algoritması, Kurt Kolonisi Algoritması, Yapay Arı Koloni Algoritması



Parçacık Sürü Optimizasyonu

Parçacık sürü optimizasyonu kuş ve balık surulerinden esinlenerek geliştirilmiş populasyon tabanlı arama ve optimizasyon algoritmasıdır.

Parçacık sürü optimizasyonu, ilk olarak Kennedy ve Eberhart tarafından 1995 yılında ortaya atılmıştır.

Diğer evrimsel algoritma ve matematiksel temelli algoritmalarla göre fazla hafıza gerektirmeyen, hesapsal olarak etkili ve uygulanması kolay bir optimizasyon yöntemidir. Ayrıca hızlı yakınsama özelliğine sahiptir. PSO az parametre gerektirmesi, hızlı sonuç bulması ve çok etkili global arama algoritması nedeniyle diğer arama algoritmalarına göre daha iyi performans sağlar.

Parçacık sürü optimizasyonu aslında sürü zekasına dayanan bir algoritmadır. PSO parçacıklar arasındaki sosyal bilgi paylaşımına dayanmaktadır. Her bireye parçacık denir ve parçacıklardan oluşan popülasyona da sürü denir. Her bir parçacık optimize edilmiş problemin birer aday çözümünü temsil etmektedir. Amaç sürüdeki en iyi parçacığın konumunu belirlemektir. Parçacıklar bir sonraki konumunu geçmiş tecrübelerine ve sürüdeki en iyi bireye dayanarak iyileştirmeye çalışırlar.

- PSO algoritmasındaki ilk populasyon rastgele parçacıklardan oluşmaktadır.
- PSO algoritması, her iterasyonda parçacıklarını güncelleyerek optimum çözümü arar.
- Sürü içerisindeki her bir parçacık bir cevabı temsil etmektedir.
- Parçacıkların sayısı PSO algoritmasında üretilen çözüm sayısına eşittir. Her parçacık, en iyi pozisyonu belirlemek için hafızasındaki o andaki ve önceki pozisyonlarını kullanır.
- Parçacığın en iyi pozisyonuna pbest adı verilir. Tüm parçacıkların en iyi pozisyonu gbest olarak adlandırılır. Durdurma kriteri sağlanıyor ise, PSO algoritması durdurularak çözüm olarak gbest parçacık seçilir.



PSO algoritması rastgele üretilen parçacıkların popülasyonu ile başlar ve parçacıkları güncelleyerek en uygun değeri arar.

PSO algoritmasındaki her bir iterasyonda parçacıklar farklı hızlarla en iyi çözüme doğru hareket ederler. Sürüdeki tüm parçacıklar, uygunluk (amaç) fonksiyonuna göre değerlendirilir.

Uygunluk fonksiyonu, çözümün ne kadar iyi olduğunu belirlemek için kullanılan bir değerlendirme fonksiyonudur. Arama alanındaki her bir parçacığın en iyi kendi çözümü (pbest) ve tüm parçacıkların en iyi çözümü (gbest) göre güncellenir.

Bu değerler bulunduktan sonra; parçacığın hızı ve konumu sırasıyla aşağıdaki denklemlerine göre güncellenir.

$$V_i^{(k+1)} = w * V_i^{(k)} + c_1 * r_1 * \left(x_{i,best}^{(k)} - x_i^{(k)} \right) + c_2 * r_2 * \left(x_{gbest}^{(k)} - x_i^{(k)} \right) \quad \text{Denklem 1}$$

$$x_i^{(k+1)} = x_i^{(k)} + V_i^{(k+1)} \quad \text{Denklem 2}$$

Birinci denklemdeki c_1 ve c_2 sabit pozitif hızlanma katsayıları olup ve genellikle $c_1=c_2=2$ olarak seçilir. r_1 ve r_2 katsayıları rastgele sayılardır ve her iterasyonda yenilenir. r_1 ve r_2 katsayıları 0 ile 1 aralığında değişmektedir. ω ise eylemsizlik ağırlığıdır ve genellikle 0.1 ile 1 aralığında değişmektedir.

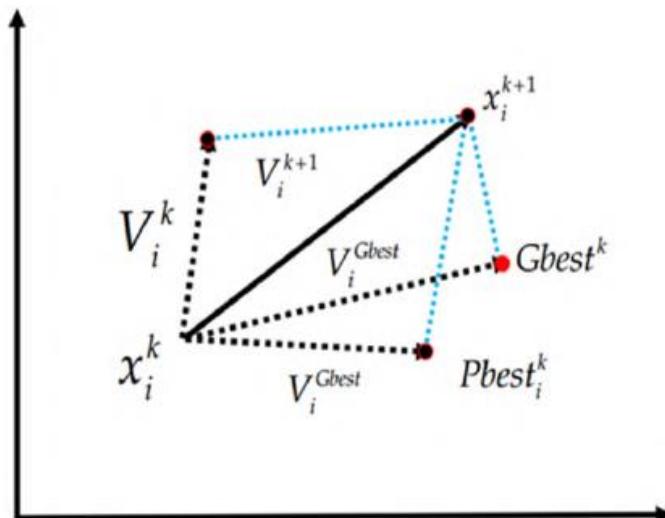
pbest ve gbest değerleri şu şekilde ifade edilebilir.

Dr. Öğr. Üyesi Mahir Kaya
TOGU-CENG

$$Pbest_i^{(k)} = x_{i,best}^{(k)} = \begin{cases} x_{i,best}^{(k-1)} & \text{if } f(x_i^{(k)}) \geq f(x_{i,best}^{(k-1)}) \\ x_i^{(k)} & \text{if } f(x_i^{(k)}) < f(x_{i,best}^{(k-1)}) \end{cases}$$

$$gbest^{(k)} = x_{gbest}^{(k)} = \min \left\{ f(x_{1,best}^{(k)}), f(x_{2,best}^{(k)}), \dots, f(x_{n,best}^{(k)}) \right\}$$

PSO'da parçacıklar (kuşlar), her bir iterasyonda çoklu arama alanındaki konumlarını değiştirirler. PSO'da arama alanındaki parçacıklar hareketi aşağıdaki şekilde (Allaoua et al., 2009) gösterilmiştir.



$X_i^{(k)}$: parçacığın anlık konumu.

$X_i^{(k+1)}$: parçacığın bir sonraki konumu.

$V_i^{(k)}$: parçacığın anlık hızı.

$V_i^{(k+1)}$: parçacığın bir sonraki hızı.

$V_i^{(pbest)}$: Pbest parçacığın hızı.

$V_i^{(gbest)}$: gbest parçacığın hızı.

Parçacık sürüsü optimizasyonu algoritması aşağıdaki adımlara göre çalışır.

Adım 1: Parçacıkların başlangıç konumları (x_i) arama alanında rastgele olarak belirlenir.

Adım 2: Parçacıkları değerlendirmek için sürüdeki tüm parçacıkların uygunluk değerleri hesaplanır.

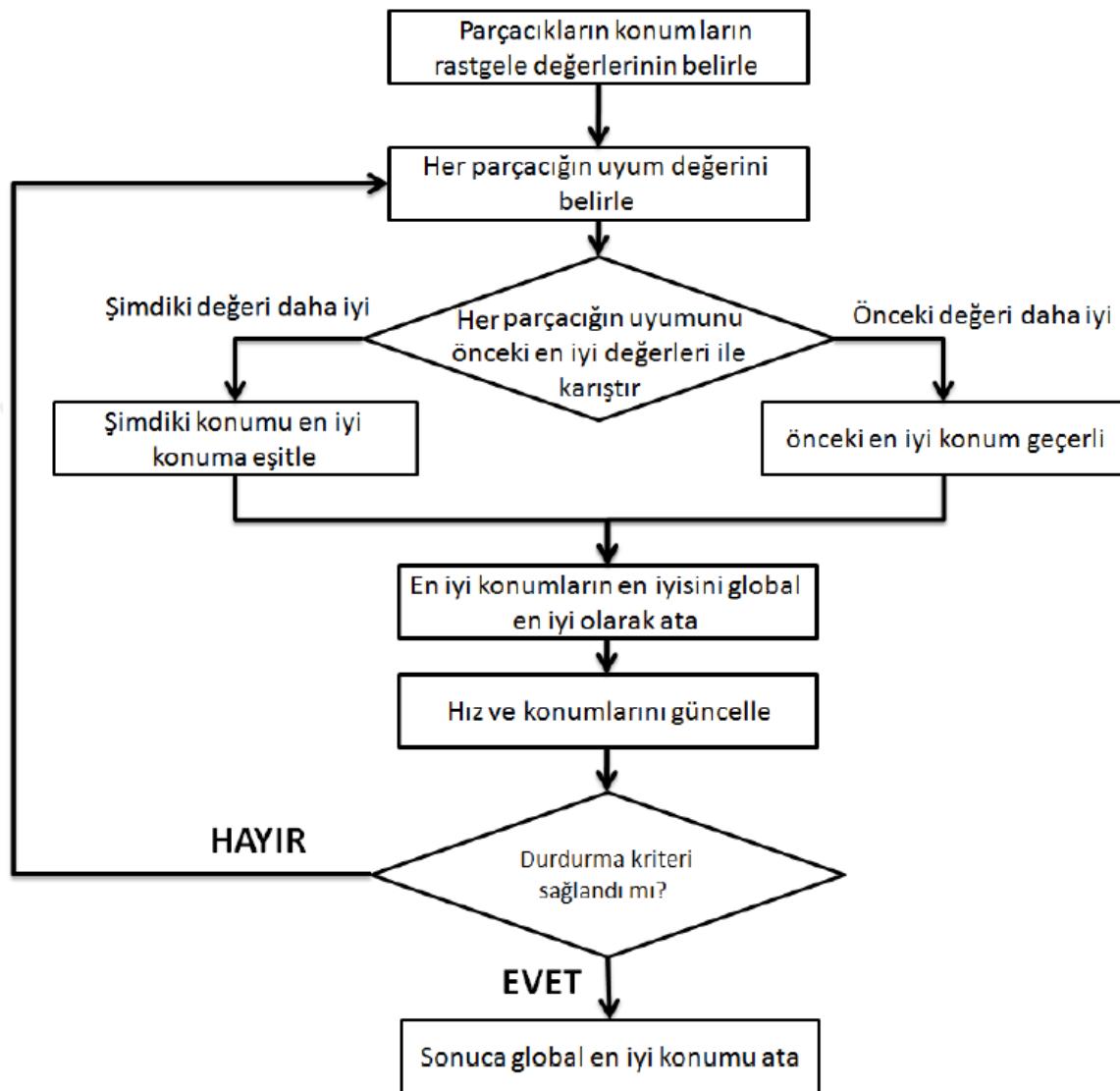
Adım 3: Eğer parçacık konumunun uygunluk değeri kendi yerel en iyi konumunun uygunluk değerinden daha iyi ise, parçacık konumu yerel en iyi konumu Pbest olarak güncellenir.

Adım 4: Yerel en iyi parçacık vektörünün en iyi uygunluk değerine sahip parçacık global en iyi parçacık gbest olarak güncellenir.

Adım 5: Parçacıkların hız ve konumları, denklem 1 ve 2'ye göre güncellenir.

Adım 6: Algoritma, durdurma kriterine ulaşılınca kadar ucunu adımdan itibaren tekrarlanır.

PSO'ya ait temel akış diyagramı



Uygunluk fonksiyonu, algoritmanın her iterasyonunda mevcut popülasyondaki parçacıkların kalitesini değerlendirmek için kullanılır.

Uygunluk fonksiyonu, populasyondaki bir parçacık ne kadar iyi olduğunu değerlendirir.

Maksimum iterasyon sayısına ulaşıldığında veya uygunluk değeri istenilen seviyeye ulaştığında algoritma durdurulabilir.

Evrim sonunda elde edilen global en iyi konum, problemin çözümü olarak alınır.

PSO algoritmasının Pseudo kodu

```
For her parçacık için
    rastgele bir konum değerini ata
End
Do
For her parçacık için uygunluk değerini hesapla
    Eğer parçacık uygunluk değeri, pbest parçacık uygunluk
        değerinden daha iyi ise; bu parçacığı yeni pbest parçacığı
        olarak ata
End
Pbest parçacıklarının en iyisini gbest parçacığı olarak ata
For her parçacık için
    Denklem 1'e göre parçacık hızını hesapla
    Denklem 2'e göre parçacık konumunu güncelle
End
While durdurma kriterine ulaşılıncaya kadar devam et
```

Parçacık Sürüsü Optimizasyonunun Ana

Parametreleri

Parçacık sürü optimizasyonu, problem boyutu, surunun büyüklüğü, eylemsizlik ağırlığı, hızlanma katsayıları, bilişsel ve sosyal katkılarını belirleyen rastgele değerler gibi birçok kontrol parametresinden oluşmaktadır.

Sürünün büyüğü: Sürüdeki parçacıkların sayısıdır. Sürüde çok sayıda parçacığın olması her iterasyonda daha fazla sayıda çözüme ulaşılabilmesini sağlar. Ayrıca fazla sayıdaki parçacık her iterasyonda hesaplama zorluğunu artırır. Daha fazla sayıda parçacığın olması, daha az sayıdaki parçacığa göre iyi çözüme daha az iterasyonda ulaşılmasını sağlar. Sürüde en çok kullanılan parçacık sayısı genellikle 20 ve 40 parçacık arasında değişir. Aslında çoğu problem için sayıyı 10 almak kabul edilebilir çözümler elde etmek için yeterlidir. Bazı özel problemler için 40'tan daha fazla parçacık kullanılması gerekebilir.

İterasyon sayısı: Optimal çözümü elde etmek için gereken iterasyon sayısı probleme bağlı olarak değişmektedir. PSO algoritmasındaki düşük iterasyon sayısı iyi bir çözüm vermek için yeterli olmayıabilir. Yüksek iterasyon sayısı da hesaplama zorluğunu arttırmır.

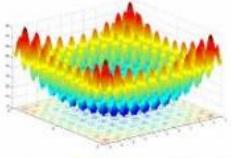
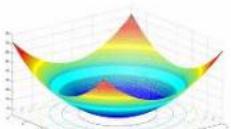
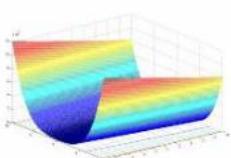
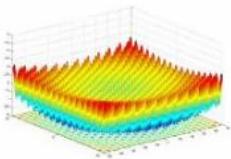
Hızlanma katsayıları: Hızlanma katsayıları c_1 ve c_2 , parçacıkların bilişsel ve sosyal bileşenlerini etkiler. Bilişsel bilesen, yerel arama yeteneğini kontrol eder ve sosyal bilesen, global arama yeteneğini kontrol eder. c_1 , parçacıkların kendi yerel en iyi konumlarına göre hareket etmesini, c_2 ise sürüdeki en iyi parçacığın konumuna göre hareket etmesini sağlar. Eğer hızlanma katsayıları sıfır olarak ayarlanırsa, parçacıklar arama alanındaki mevcut hızlarıyla hareketlerine devam ederler.

Hızlanma katsayılarının düşük değerlerde seçilmesi parçacıkların yavaş yavaş hedef bölgeye doğru hareket etmesini sağlar. Ancak hızlanma katsayılarının yüksek değerlerde seçilmesi, hedefe ulaşmayı hızlandırırken, beklenmedik hareketlerin oluşmasına ve hedef bölgenin es geçilmesine neden olabilir. PSO algoritması üzerinde araştırmacıların yaptığı denemelerde " $c_1=c_2=2$ " olarak almanın iyi sonuçlar verdiği belirtilmiştir.

Parçacık Sürüsü Optimizasyonunun Ana Parametreleri

- **Eylemsizlik Ağırlığı:** PSO algoritmasında eylemsizlik ağırlığı global ve yerel arama sureci arasında denge kurmak için kullanılmaktadır. Büyük eylemsizlik ağırlığı global arama yeteneğini ve küçük eylemsizlik ağırlığı yerel arama yeteneğini artırır. Bu nedenle eylemsizlik ağırlığı, arama alanında yerel ve global araştırma arasında dengeyi sağlar ve en az sayıda iterasyonla optimum çözüme ulaşmayı amaçlar.
- Eylemsizlik ağırlığı eski hız bilgisinin yeni hız bilgisine etkisini ayarlayan parametredir. Başka bir deyişle eski yöne ilişkin hafızanın yeni hızı nasıl etkileyeceğini belirler. Eylemsizlik ağırlığı ω , hız güncelleme denkleminde önceki hız değerine çarpan olarak yazılır. Eğer eylemsizlik ağırlığı birden daha fazla ise parçacıkların hızları zamanla artar, suru ıraksar ve parçacıklar daha iyi bölgelere doğru yön değiştirme kabiliyetini yitirirler. Eğer eylemsizlik ağırlığı sıfırdan daha az ise parçacıklar hızları sıfır oluncaya kadar yavaşlarlar.

Bazı Test Fonksiyonları

Test Fonksiyonları Tekli Nesnesel Optimizasyon				
Adı	Formül	En Küçük Nokta	Arama Aralığı	Grafik
Rastrigin	$f(x) = An + \sum_{i=1}^n [x_i^2 - A \cos(2\pi x_i)]$ $A = 10$	$f(0,0) = 0$	$-5, 12 \leq x, y \leq 5, 12$	
Sphere	$f(x) = \sum_{i=1}^n x_i^2$	$f(0,\dots,0) = 0$	$-\infty \leq x_i \leq \infty, 1 \leq i \leq n$	
Rosenbrock	$f(x) = \sum_{i=1}^{n-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$	$f(1,\dots,1) = 0$	$-\infty \leq x_i \leq \infty, 1 \leq i \leq n$	
Griewank	$f(x) = 1 + \frac{1}{400} \sum_{i=1}^n x_i^2 - \prod_{i=1}^n \cos\left(\frac{x_i}{\sqrt{i}}\right)$	$f(0,\dots,0) = 0$	$-600 \leq x_i \leq 600, 1 \leq i \leq n$	

Python Kod

```
1 import numpy as np  
2 import matplotlib.pyplot as plt
```

Gerekli kütüplaneleri import ediyoruz. numpy sayısal işlemler yapmayı sağlar. matplotlib'i ise grafik çizdirmede kullanacağız.

```
4 def fun(x):  
5     return sum(np.power(x,2))
```

Optimize etmek istediğimiz «sphere» fonksiyonunu tanımlıyoruz.

```
7 alt_sinir = -5  
8 ust_sinir = 5  
9 problem_boyutu = 5  
10 populasyon_boyutu = 10  
11 w = 0.8  
12 c1 = 2  
13 c2 = 2
```

Problem için gerekli parametreleri ayarlıyoruz.

```
15 populasyon = np.random.ranf([populasyon_boyutu,problem_boyutu]) * (ust_sinir-alt_sinir) + alt_sinir
```

`np.random.ranf` fonksiyonun `-1` ve `1` arasında rastgele değerler üretir. Üretilen değerleri, üst sınır ve alt sınır değerleriyle işleme sokarak `-5` ve `5` arasında olmasını sağlıyoruz.

Böylece `-5` ve `5` arasında, `10x5`'lik bir dizi oluşturuldu.

Yani `5` boyutlu bir problem için `10` bireylik bir popülasyon oluşturduk.

```
[[ 1.9233098  4.83348664 -1.19536665  1.33890628  3.75553393]
 [-1.99741024  2.23033211  2.84222985 -4.46153768  4.89855476]
 [ 2.64376672  4.46786319 -4.47502255  4.28493731 -2.13720363]
 [ 2.53746694  0.05199803 -4.34719122 -4.6959648   2.80194307]
 [ 1.77820128 -2.26663735  3.52272236  1.2189377   4.56832483]
 [ 2.72534117 -4.12019712 -0.6629858  -3.79047539  3.18343924]
 [ 2.61636996 -3.63534678 -3.68293378 -1.74972797  4.27707968]
 [-4.08225525 -4.01727973  3.19594031  3.81089701  0.43855357]
 [ 0.37937127  3.07626922 -4.10406105 -2.69759601 -1.95190874]
 [-1.33643569  1.17432522  0.00999566  1.17505743  0.78440131]]
```

```
17     obj = np.zeros(populasyon_boyutu)
18
19     for i in range(populasyon_boyutu):
20         obj[i]=fun(populasyon[i,:])
```

Populasyondaki her bir birey için amaç fonksiyonu değerlerini tutacak bir dizi oluşturup, her bireyin amaç fonksiyonunu hesapladık.

```
23     velocity = np.zeros([populasyon_boyutu,problem_boyutu])
24
25     pBestPos = populasyon
26     pBestVal = obj
27
28     gBestVal = min(obj)
29     idx = np.where(obj==gBestVal)
30     gBestPos = populasyon[idx,:]
```

23. satırda hız değerlerini tutan dizi tanımlandı.

25 ve 26. satırlarda bireylerin pBest pozisyon ve değerlerini tutacak değişkenler oluşturuldu.

28-30. satırlarda ise popülasyondaki gBest değeri tespit ettik.

```
32     objit = list()
33     objit.append(gBestVal)
```

32. Ve 33. satırlarda grafik çizdirmek için her bir iterasyonda gBestVal değerinin ekleneceği boş bir liste oluşturup, ilk bulduğunuz gBestVal değerini listeye ekledik.

```

35     for k in range(40):
36         for i in range(populasyon_boyutu):
37             velocity[i,:] = w*velocity[i,:] + \
38                 c1*np.random.ranf()*(pBestPos[i,:]-populasyon[i,:]) + \
39                 c2*np.random.ranf()*(gBestPos-populasyon[i,:])
40
41         vmax = (ust_sinir - alt_sinir) / 2
42         for i in range(populasyon_boyutu):
43             for j in range(problem_boyutu):
44                 if velocity[i,j]>vmax:
45                     velocity[i,j]=vmax
46                 elif velocity[i,j]<-vmax:
47                     velocity[i,j]=-vmax
48
49         populasyon = populasyon + velocity
50
51         for i in range(populasyon_boyutu):
52             for j in range(problem_boyutu):
53                 if populasyon[i,j]>ust_sinir:
54                     populasyon[i,j]=ust_sinir
55                 elif populasyon[i,j]<alt_sinir:
56                     populasyon[i,j]=alt_sinir
57
58         for i in range(populasyon_boyutu):
59             obj[i]=fun(populasyon[i,:])
60
61         for i in range(populasyon_boyutu):
62             if obj[i]<pBestVal[i]:
63                 pBestVal[i,:]=populasyon[i,:]
64                 pBestVal[i]=obj[i]
65
66             if min(obj)<gBestVal:
67                 gBestVal=min(obj)
68                 idx = np.where(obj==gBestVal)
69                 gBestPos = populasyon[idx,:]
70
71             objit.append(gBestVal)

```

Algoritmamız 40 iterasyon çalışıyor.

Her bir bireyin hız (velocity) güncellemesi.

$$V_i^{(k+1)} = w * V_i^{(k)} + c_1 * r_1 * (x_{i,best}^{(k)} - x_i^{(k)}) + c_2 * r_2 * (x_{gbest}^{(k)} - x_i^{(k)})$$

Hız (velocity) değeri –vmax ve vmax arasına ölçekleniyor.

Konum güncellemesi.

$$x_i^{(k+1)} = x_i^{(k)} + V_i^{(k+1)}$$

Konum değeri alt_sınır ve üst sınır arasına ölçekleniyor.

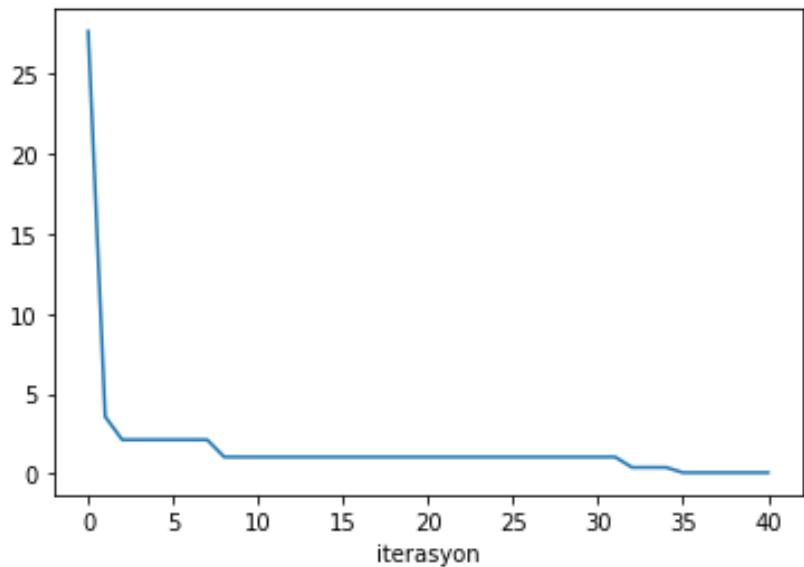
Bireylerin yeni konumlarına göre uygunluk(amaç) fonksiyonu yeniden hesaplanıyor.

Her birey için pBest değeri hesaplanıyor.

Populasyonun gBest değeri hesaplanıyor. Yani popülasyonda en optimum çözümü bulan bireyin pozisyonu ve değeri tutuluyor.

Grafik çizdirmede kullanılmak üzere her bir iterasyonda gBestVal değeri listeye ekleniyor.

```
73 plt.plot(objit)
74 plt.xlabel("iterasyon")
75 plt.show()
```



Tüm iterasyon boyunca «objit» listesinde tuttuğumuz değerlere göre sonuç grafiğini çizdiriyoruz.

Görüldüğü üzere; uygunluk (amaç) fonksiyonumuz iterasyon boyunca azalarak minimize olmuştur.

```
79 print("{:.2f}^2 + {:.2f}^2 + {:.2f}^2 + {:.2f}^2 + {:.2f}^2 = {:.2f}".
80     format(gBestPos[0],gBestPos[1],
81             gBestPos[2],gBestPos[3],
82             gBestPos[4],gBestVal))
```

Toplam iterasyon sonucunda 5 boyutta ele aldığımız sphere fonksiyonunun aşağıdaki karar değişkenleri için o'a yaklaştığını gördük.

$$0.23^2 + -0.00^2 + -0.01^2 + 0.09^2 + 0.04^2 = 0.06$$

Animasyon ve Örnekler

- <https://www.youtube.com/watch?v=OUHAYpWh1Ro>
- <https://www.youtube.com/watch?v=gkGa6WZpcQg>