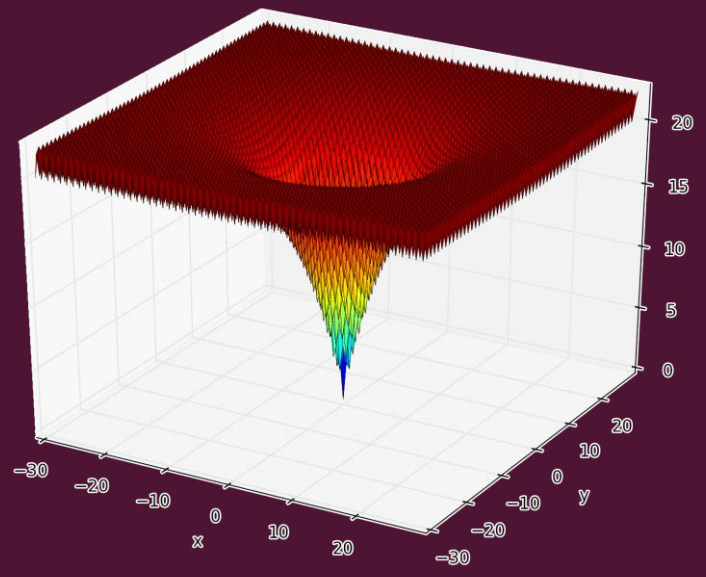
تمارش شماره دوم

تهیه کنندگان:
امیرحسین وطنی باف
حسین دماوندی

زیرنظر استاد محترم:
دکتر امیرفرید امینیان مدرس



یافتن مینیمم تابع Ackley

به کمک الگوریتم های تکاملی

مقدمات و تعریف توابع جانبی

```
[1]: import random
     import numpy as np
     import math
     import matplotlib.pyplot as plt
```

install requirements:

```
pip install numpy
```

برای اجرای این برنامه نیاز به نصب کتابخانه numpy خواهید داشت.

همچنین پیشنهاد می‌شود سورس کد برنامه را در محیط Jupyter Notebook یا Google Colab اجرا فرمایید

مقدمات و تعریف توابع جانبی

تعریف متغیر سراسری sx برای سایز مسئله

```
[25]: sx = 100
```

کدگذاری

CODING

مرحله اول در هر الگوریتم تکاملی کدگذاری یا همان coding می‌باشد.
در این مرحله می‌بایست تعیین کنیم که در هر کروموزوم چه اطلاعاتی باید ذخیره شود.

در مسئله یافتن مقدار مینیمم تابع Ackley هر نقطه از یک x,y بوجود آمده که هر کدام از این نقطات (موجودات) دارای ارتفاع h می‌باشند.

هر نقطه از صفحه و ویژگی های آنرا داخل یک DS ذخیره میکنیم.

در حل این مسئله ما تصمیم گرفتیم کلاس point را ایجاد کنیم و هر نقطه را یک شی در نظر بگیریم. (در اسلاید بعدی قابل مشاهده است)

CODING

کدگذاری

[26]:
```
class point:

    def __init__(self, x , y):
        self.x = x
        self.y = y
        self.h = fitness_value(x, y)
        self.xbin = to_binary(x)
        self.ybin = to_binary(y)
        self.x_dec_place = to_binary_NoneSigin(x)
        self.y_dec_place = to_binary_NoneSigin(y)
```

همانطور که قابل مشاهده است در این بلاک کلاس point با اشیاء مختلفی تعریف شده.

برای راحتی کار ما در همین قسمت، مرحله ارزیابی یا همان evaluation را نیز انجام میدهیم و برای هر موجود محاسبه انتفاع نیز با استفاده از تابع fitness_value (که در اسلاید های آینده قابل مشاهده است) انجام میپذیرد.

باتوجه به اینکه در الگوریتم های تکاملی غالبا رشته باینری است تابعی برای تبدیل به باینری با عنوان to_binary در نظر گرفته شده و همچنین برای تبدیل قسمت اعشاری نیز یک تابع دیگر با نام to_binary_NoneSigin در نظر گرفتیم.

کدگذاری

to_binary(num)

برای هر value تعداد 6 بیت در نظر میگیریم و یک بیت را به بیت علامت اختصاص میدهیم (طبق تعریف تابع در بازه 30,30- می‌باشد و در 5 بیت قابل نمایش دادن است)

[29]:
```
def to_binary(num):
    x = num
    if x < 0 :
        x = x * -1

    x_bin = "{0:b}".format(int(x))
    x_str = ['0','0','0','0','0']

    j = 0
    for i in range(len(x_str) - len(x_bin) , 5):
        x_str[i] = x_bin[j]
        j +=1

    if num >= 0 :
        x_str.insert(0,'0')
    else :
        x_str.insert(0,'1')

    return x_str
```

کدگذاری

to_binary_NoneSigin(x)

برای هر value قسمت اعشار آن را تا چهار رقم اعشار در 14 بیت اعشاری در x_str قرار میدهد

[30]:
```
def to_binary_NoneSigin(x):

    x = round(x,4)
    x2 = str(x)

    temp = ''
    for i in range(len(x2)):
        if x2[i] == '.':
            index = i
            break

    for i in range(index + 1 , len(x2)):
        temp = temp + x2[i]

    while len(temp) < 4:
        temp = '0' + str(temp)

    x_bin = "{0:b}".format(int(temp))

    x_str = ['0','0','0','0','0','0','0','0','0','0','0','0','0','0']

    j = 0
    for i in range(len(x_str) - len(x_bin) , 14):
        x_str[i] = x_bin[j]
        j +=1

    return x_str
```

to_decimal(xbin, x_dec_place) to_decimal_z(xbin) to_decimal_place(x_dec_place)

[31]:
```
def to_decimal(xbin, x_dec_place) :

    x = to_decimal_z(xbin)
    x += to_decimal_place(x_dec_place)

    if xbin[0] == '1':
        x *= -1

    return x
```

تابع اصلی

با توجه به نیاز چند تابع نیز برای تبدیل به اعداد دهدهی درنظر گرفته شده است

[32]:
```
def to_decimal_z(xbin) :

    x=''
    for i in range(1,len(xbin)):
        x = x[:len(xbin)] + xbin[i] + x[len(xbin):]

    x = int(x, 2)

    return x
```

قسمت صحیح

[33]:
```
def to_decimal_place(x_dec_place) :

    x=''
    for i in range(0,len(x_dec_place)):
        x = x[:len(x_dec_place)] + x_dec_place[i] + x[len(x_dec_place):]

    x = int(x, 2)

    return x * (10 ** -4)
```

قسمت اعشاری

ارزیابی

fitness_value(x, y)

[703]:
```
def fitness_value(x, y):

    value = -20.0 * np.exp(-0.2 * np.sqrt(0.5 * (x**2 + y**2))) - np.exp(0.5 * (np.cos(2 * np.pi * x) + np.cos(2 * np.pi * y))) + np.e + 20

    return value
```

هر موجود توسط این تابع ارزیابی میگردد

بدلیل اینکه قصد ما یافتن مینیمم است، این تابع برای موجودات بهتر مقدار کمتری را بر میگرداند

مراحل یک الگوریتم تکاملی

- تولید جمعیت اولیه
- محاسبه برازندگی (ارزیابی جمعیت ورودی)
- انتخاب برای تولید مثل
- باز ترکیب والدین (تولید مثل)
- جهش فرزندان تولید شده
- محاسبه برازندگی (ارزیابی جمعیت فرزندان)
- انتخاب برای جایگزینی
- بررسی شرط توقف

1 - تولید جمعیت اولیه

first_population(size)

روش تصادفی - پوشش یکنواخت فضا

```
[34]: def first_population(size):
          popu = []
          for i in range(size):
              x = random.uniform(-30 , 30)
              y = random.uniform(-30 , 30)
              popu.append(point(x,y))

          return popu
```

در مرحله اول یعنی تولید جمعیت اولیه از **روش تصادفی** (تولید مقادیر تصادفی در بازه -30 تا 30 برای هر ژن)

این تابع در ازای تعداد size دریافتی جمعیت تصادفی ایجاد می‌کند.

1 - تولید جمعیت اولیه

first_population(size)

روش هیوریستیک - پوشش بخش های مهم فضا

```
[34]: def first_population(size):
          popu = []
          for i in range(size):
              x = random.uniform(-15 , 15)
              y = random.uniform(-15 , 15)
              popu.append(point(x,y))

          return popu
```

همچنین می‌توانیم از روش هیوریستیک استفاده کنیم، به این گونه که نقاط تصادفی را در جایی که فکر می‌کنیم جواب است تولید کنید

در قسمت نتیجه گیری به تفاوت خروجی های این دو روش میپردازیم

2 - تابع برازش

fitness_value(x, y)

همانطور که قبلا در بخش ارزیابی معرفی شد این اقدام با استفاده از تابع fitness_value(x, y) انجام میپزیرد

[703]:
```
def fitness_value(x, y):

    value = -20.0 * np.exp(-0.2 * np.sqrt(0.5 * (x**2 + y**2))) - np.exp(0.5 * (np.cos(2 * np.pi * x) + np.cos(2 * np.pi * y))) + np.e + 20

    return value
```

3 - عملگر انتخاب select_parents(population,size)

روش نسبی - fitness بیشتر ؛ شانس انتخاب بیشتر

```
[35]: def select_parents(population,size):

          All_Parent = []
          sort_population = []
          weights = []
          sort_population = sorted(population, key=lambda p: p.h, reverse=True)

          for i in range(len(sort_population)):
              weights.append(i+1)

          All_Parent = random.choices(sort_population, weights = weights , k = size)

          return All_Parent
```

ابتدا جمعیت را بر اساس انتفاع مرتب کرده و برای هر کدام یک وزن (شانس) با توجه به جایگاهش داده می‌شود؛ و با توجه به وزن هر موجود شانس بیشتر برای انتخاب دارد

3 - عملگر انتخاب select_parents_computation(population,size)

روش مسابقه ای

```
[13]: def select_parents_computation(population,size):

    All_Parent = []
    sort_population = []


    for i in range(0,size) :
        temp_population = population.copy()
        random_population = []
        for j in range(0,len(population)//5) :
            index = random.randint(0,len(population)-1-j)
            random_population.append(temp_population[index])
            temp_population.pop(index)

        sort_population = sorted(random_population, key=lambda p: p.h, reverse=False)
        All_Parent.append(sort_population[0])


    return All_Parent
```

ابتدا انتخاب تصادفی از بین موجودات داریم و سپس بر اساس انتفاع آن ها را مرتب کرده و بهترین را انتخاب میکنیم

4 - عملگر های الگوریتم ژنتیک

cross_over(parents)

بر روی value که یک عدد اعشاری می‌باشد (قسمت صحیح و اعشاری توسط دو تابع اسلاید بعدی) ترکیب یک نقطه ای (One point) انجام میدهیم.

[19]:
```
def cross_over(parents) :
    childeren = []
    for i in range(0,len(parents)-1) :

#         xbin_child1 = cross_one_point_z(parents[i].xbin,parents[i+1].xbin)
        xbin_child1 = cross_uniform_z(parents[i].xbin,parents[i+1].xbin)
        xbin_child2 = xbin_child1[::-1]
#         ybin_child1 = cross_one_point_z(parents[i].ybin,parents[i+1].ybin)
        ybin_child1 = cross_uniform_z(parents[i].ybin,parents[i+1].ybin)
        ybin_child2 = ybin_child1[::-1]

#         x_dec_place_child1 = cross_one_point_palce(parents[i].x_dec_place,parents[i+1].x_dec_place)
        x_dec_place_child1 = cross_uniform_palce(parents[i].x_dec_place,parents[i+1].x_dec_place)
        x_dec_place_child2 = x_dec_place_child1[::-1]
#         y_dec_place_child1 = cross_one_point_palce(parents[i].y_dec_place,parents[i+1].y_dec_place)
        y_dec_place_child1 = cross_uniform_palce(parents[i].y_dec_place,parents[i+1].y_dec_place)
        y_dec_place_child2 = y_dec_place_child1[::-1]

        x_child1 = to_decimal(xbin_child1,x_dec_place_child1)
        y_child1 = to_decimal(ybin_child1,y_dec_place_child1)
        x_child2 = to_decimal(xbin_child2,x_dec_place_child2)
        y_child2 = to_decimal(ybin_child2,y_dec_place_child2)

        childeren.append(point(x_child1,y_child1))
        childeren.append(point(x_child2,y_child2))

        i += 1


    return childeren
```

4 – عملگر های الگوریتم ژنتیک

select_parents(population,size)
cross_one_point_z(parent1,parent2)

روش اول: ترکیب یک نقطه ای (One point)

[36]:
```
def cross_one_point_z(parent1, parent2) :
    index = random.choice([1,2,3,4,5])
    child = []

    for i in range(0,len(parent1)) :

        if i < index :
            child.append(parent1[i])
        else :
            child.append(parent2[i])

    return child
```

[37]:
```
def cross_one_point_palce(parent1, parent2) :

    index = random.choice([0,1,2,3,4,5,6,7,8,9,10,11,12,13])
    child = []

    for i in range(0,len(parent1)) :
        if i < index :
            child.append(parent1[i])
        else :
            child.append(parent2[i])

    return child
```

4 – عملگر های الگوریتم ژنتیک

select_parents(population,size)
cross_one_point_z(parent1,parent2)

روش دوم: ترکیب یکنواخت (Uniform)

[16]:
```
def cross_uniform_z(parent1, parent2) :
    child = []

    for i in range(0,len(parent1)) :
        if random.choice(['1', '2']) == '1' :
            child.append(parent1[i])
        else :
            child.append(parent2[i])

    return child
```

[18]:
```
def cross_uniform_palce(parent1, parent2) :
    child = []

    for i in range(0,len(parent1)) :
        if random.choice(['1', '2']) == '1' :
            child.append(parent1[i])
        else :
            child.append(parent2[i])

    return child
```

5 – عملگر های الگوریتم ژنتیک

mutate(childeren)

جهش (mutation) – جهش ژنتیکی

یکی از کروموزوم هارا بصورت تصادفی انتخاب کرده و یکی از ژن های آن را (اینبار هم با احتمال 50 درصد) تغییر می‌دهیم

[39]:
```
def mutate(childeren):
    mutatant = childeren

    for i in range(0,len(mutatant)) :
        index = random.randint(0,len(mutatant[i].xbin)-1)
        if random.choice(['0', '1']) == '1' :
            if random.random() < 0.2:
                mutatant[i].xbin[index] = random.choice(['0', '1'])
        else :
            if random.random() < 0.2:
                mutatant[i].ybin[index] = random.choice(['0', '1'])


    for i in range(0,len(mutatant)) :
        index = random.randint(0,len(mutatant[i].x_dec_place)-1)
        if random.choice(['0', '1']) == '1' :
            if random.random() < 0.2:
                mutatant[i].x_dec_place[index] = random.choice(['0', '1'])
        else :
            if random.random() < 0.2:
                mutatant[i].y_dec_place[index] = random.choice(['0', '1'])

    return mutatant
```

6 – جایگزینی نسل

replacement(population, parents, childeren)

```
[40]: def replacement(population, parents, childeren) :
          population_copy = population.copy()
          parents_copy = parents.copy()
          childeren_copy = childeren.copy()

          newlistChild = []
          newlistParent = []
          newlistChild = sorted(childeren_copy, key=lambda ch: ch.h, reverse=False)      #STEP 6
          newlistParent = sorted(parents_copy, key=lambda p: p.h, reverse=True)

          for i in range(0,int((len(parents_copy)*30/100)+0.5)) :
              newlistParent[i] = newlistChild[i]

          newpopulation = newlistParent
          sort_pop = sorted(population_copy, key=lambda p: p.h, reverse=True)

          for i in range(0,len(newpopulation)) :
              sort_pop[i] = newpopulation[i]

          return sort_pop
```

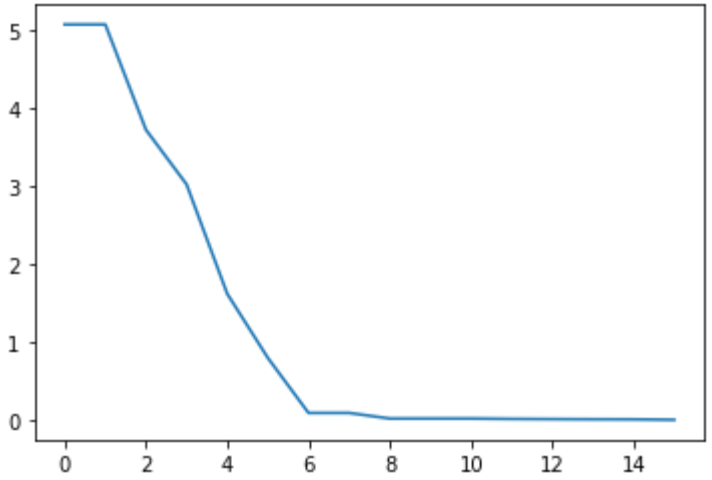
توسط این تابع از بین والدین یکسری را انتخاب میکنیم و به نسل بعدی انتقال می‌دهیم (نسبت 30 – 70) به عبارت دیگر بخش بزرگی از والدین و بخش کمتری از فرزندان به دلیل جلوگیری از همگرا شدن سریع (در الگوریتم های تکاملی میبایست سرعت همگرایی کم باشد زیرا ممکن از موجودی که میتواند بهترین نقطه را پیدا کند از بین برود)

main()

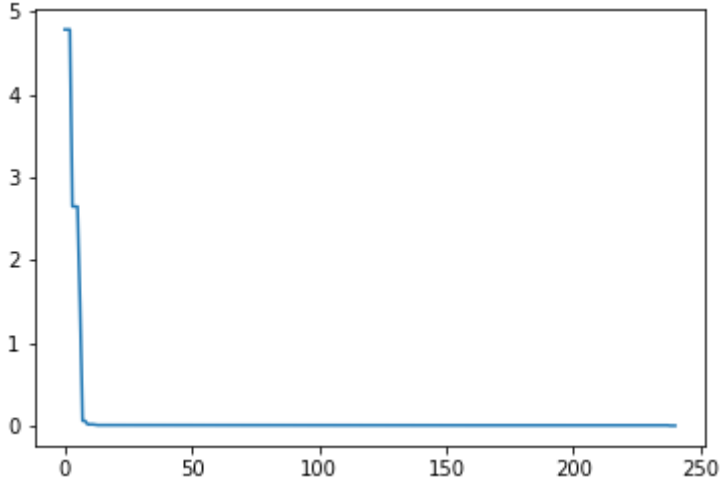
تابع اصلی

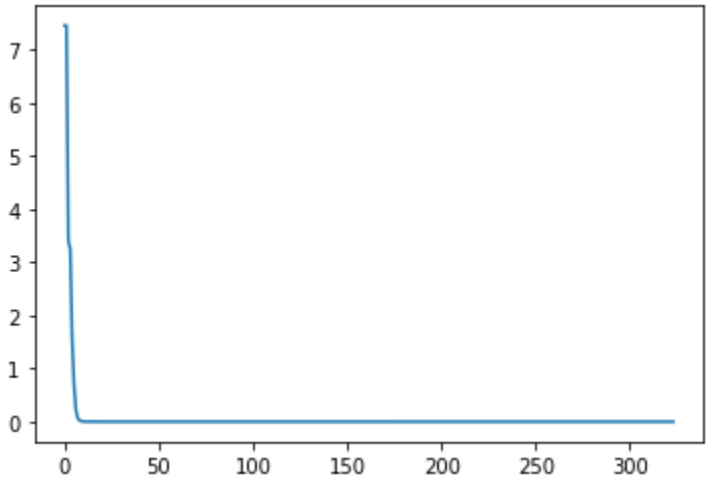
[146]:
```
def main():
    draw_fitness = []    #for draw result
    cnt=0                # for count number of repeat best in generations
    Best=100             # maximum of ackley
    gen=0                # for count number of generations
    target=0             # minimum of ackley
    firstpopulation = first_population(sx)   #STEP 1
    current_population = firstpopulation    #STEP 2
    while True :
        if gen > 1000 :
            break
        if cnt == 800 :
            break
        parents = select_parents(current_population,sx//2)   #STEP 3
        childeren = cross_over(parents)      #STEP 4
        childeren = mutate(childeren)        #STEP 5
        next_population = replacement(current_population, parents, childeren)  #STEP 6
        BestNew = GetBest(current_population)      #STEP 7

        if Best == BestNew :
            cnt += 1
        elif Best > BestNew :
            cnt=0
            Best = BestNew
        else :
            cnt = 0

        print('gen :', gen)
        print("Best (in gen ",gen,") :",Best)
        print("********************")
        draw_fitness.append(Best)
        if Best == target :
            break

        current_population = next_population
        gen += 1
    hist = draw_fitness
    plt.plot(hist)
```

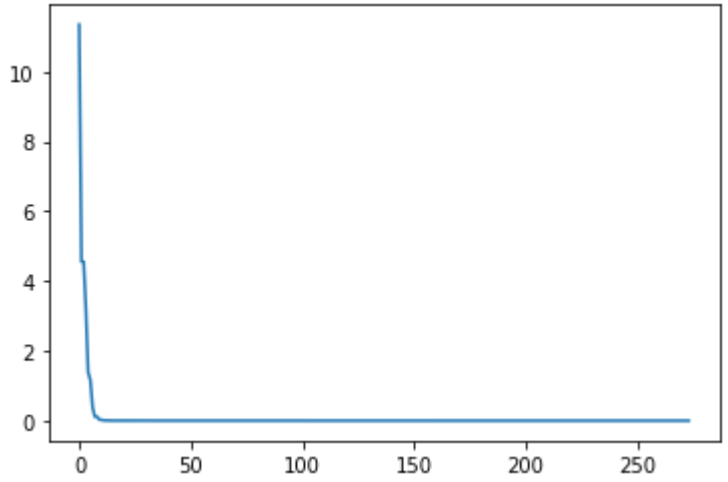
نمودار رسیدن به انتفاع ایده آل در چند اجرای متفاوت



همانطور که مشاهده می‌شود در نسل های متمادی مقدار برازش ما
به مقدار ایده آل 0 نزدیک و نزدیک تر می‌شود تا در نهاید به آن برسد





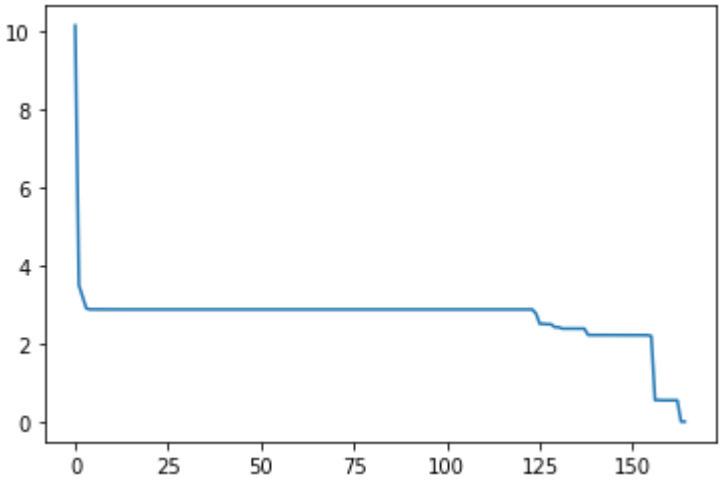


نتیجه گیری

تفاوت در پاسخ تولید جمعیت اولیه به روش های هیوریستیک و تصادفی

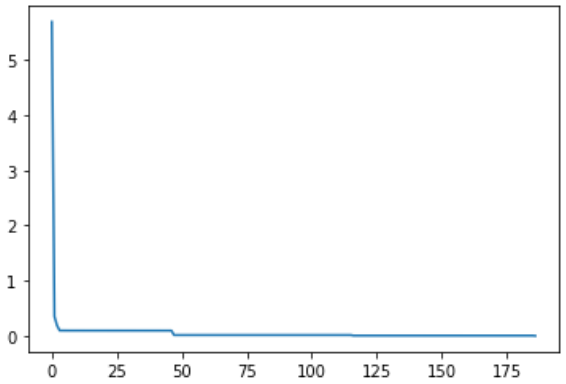
روش تصادفی:

مشاهده می‌شود که با استفاده از این روش اغلب موارد بین 40 تا 150 نسل طول میکشد که به پاسخ ایده آل برسیم



روش هیوریستیک:

باتوجه به اینکه در این روش در بازه نزدیک تری از نقطه ایده آل جمعیت ایجاد شده بود، انتظار میرفت که سریعتر به پاسخ برسیم اما بدلیل اینکه ما در پیاده سازی خود و استراکچر مورد استفاده تعداد بیت ها را برای بازه -30 تا 30 در نظر گرفته بودیم دیرتر از روش تصادفی به پاسخ می رسیدیم. اما نکته قابل توجه این است که همگرایی به سمت پاسخ درست تر زودتر انجام میشود



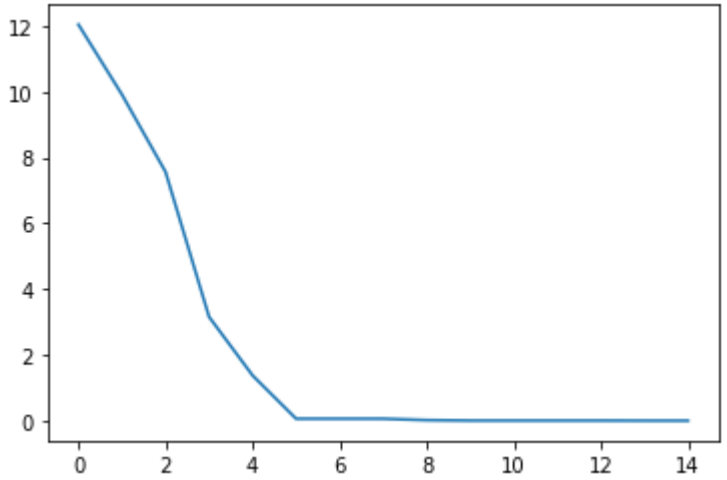
مستندات در فایل های پوشه first population قابل مشاهده است

نتیجه گیری

تفاوت در پاسخ انتخاب پدر و مادر به روش های نسبی و مسابقه ای

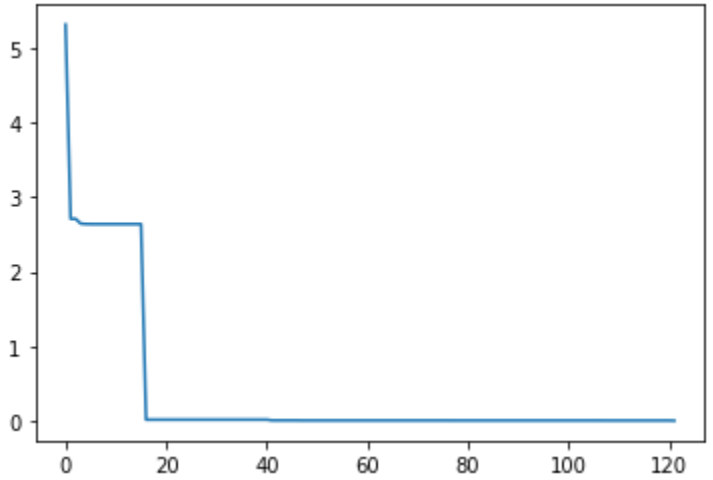
روش نسبی:

در این روش ما با توجه به انتفاع به هر موجود شانسی را انتصاب میدهیم (مدت زمان رسیدن به پاسخ معمولا زیر 100 نسل)



روش مسابقه ای:

بنظر می‌رسد که به دلیل اینکه در روش مسابقه ای ابتدا بصورت تصادفی انتخاب میکنیم و سپس بهترین را انتخاب میکنیم گاهی اوقات سریعتر از نسبی پاسخ میگیریم اما در بیشتر مواقع اینطور نیست(مدت زمان رسیدن به پاسخ بین 90 تا 200 نسل)



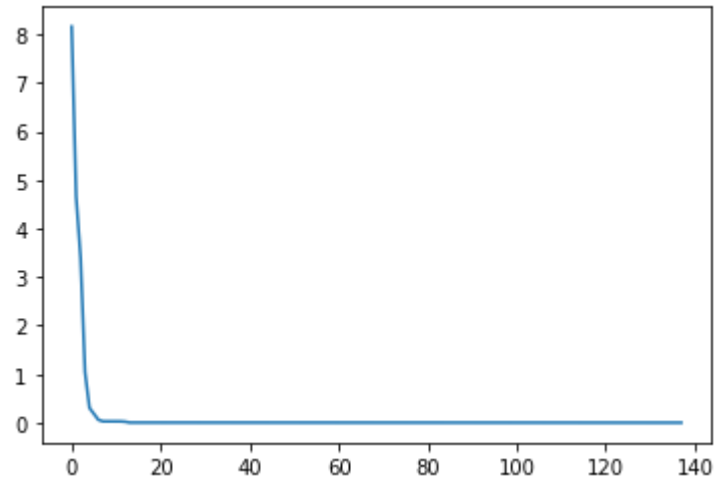
مستندات در فایل های پوشه select parents قابل مشاهده است

نتیجه گیری

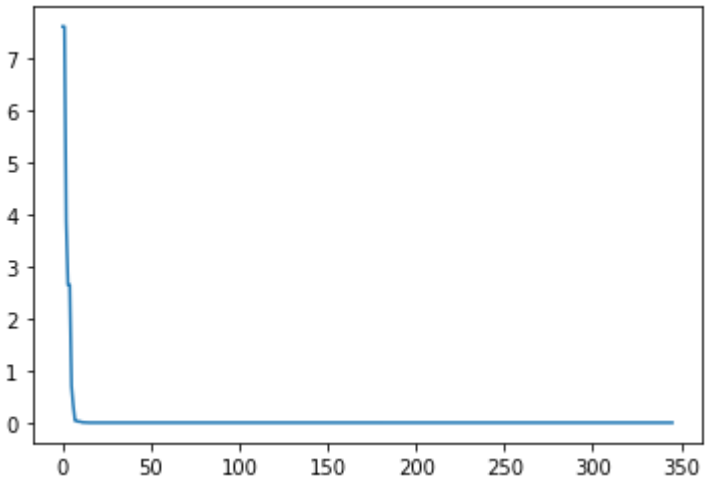
تفاوت در پاسخ تولید مثل به روش های وان پوینت و یونیفرم

زمان رسیدن به پاسخ در روش یونیفرم نسبت به وان پوینت بیشتر است. دلیل این اتفاق میتواند تنوع زیاد در فرزندانی که بوجود می‌آیند باشد و ممکنه فرزندانی بوجود آیند که از جواب بسیار دور تر باشند

روش وان پوینت:



روش یونیفرم:



مستندات در فایل های پوشه cross over قابل مشاهده است