设计一个遗传算法来求解4个函数在给定范围内的最小值

1. 引入相关的库

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
```

2. 初始化相关数据

- 最大的迭代次数 $maxEpochs$
- 变异的概率和交叉的概率 cmp, mop
- 最开始种群的规模 $firstPopulation$
- 最好的适应度，所有适应度，某一代的适应度 $bestFiT, allFit, oneFit$
- 每个函数的各自范围 $[xLowBound, xHighBound]$

- 交叉和变异的选择；选择了轮盘交叉方法（轮盘赌算法）。

- 精英主义的比例 ： $bestIdx = fitness.index(np.min(fitness))$

```python
max_epochs = 400  # 最大的迭代次数
_cmp = 0.95  # 种群交叉的概率
_mop = 0.05  # 种群变异的概率

fun_one_bound = [-5.12, 5.12]
fun_two_bound = [-2.048, 2.048]
fun_four_bound = [-65.536, 65.536]
fun_six_bound = [-5.12, 5.12]

best_fitness = []  # 每一代的最好的适应度
all_fitness = []  # 所有代所有个体的适应度
one_fitness = []  # 某一代所有个体的适应度

# 初始化最开始的种群规模
first_population = np.random.randint(low=0, high=2, size=(200, 2, 20))

# 精英主义
best_idx=fitness.index(np.min(fitness)) # 找到最小的那个
best_best=parent[best_idx].copy() # 找到最好的那个值，下标，当作是精英
```

```python
def rws_algoritm(first_population, fitness, n):  # 定义轮盘赌算法
    next_population = []  # 定义下一个子代
    sum_ = sum(fitness)  # 获取所有的适应度的和

    p_ = ((sum_-fitness)/sum_) / (len(fitness)-1) # 获得概率

    idx = np.random.choice(np.arange(len(first_population)), size=n, replace=True,
p=p_)

    for i in idx:
        next_population.append(first_population[i])
    return next_population
```

3. 定义出**4**个函数，并利用画图函数绘制图像

- ```python
  def function_one(x1, x2):  # 范围: -5.12<= x <= 5.12
      return x1 ** 2 + x2 ** 2

  def function_two(x1, x2):  # 范围: -2.048<= x <= 2.048
      return 100 * (x1 ** 2 - x2) ** 2 + (1 - x1) ** 2

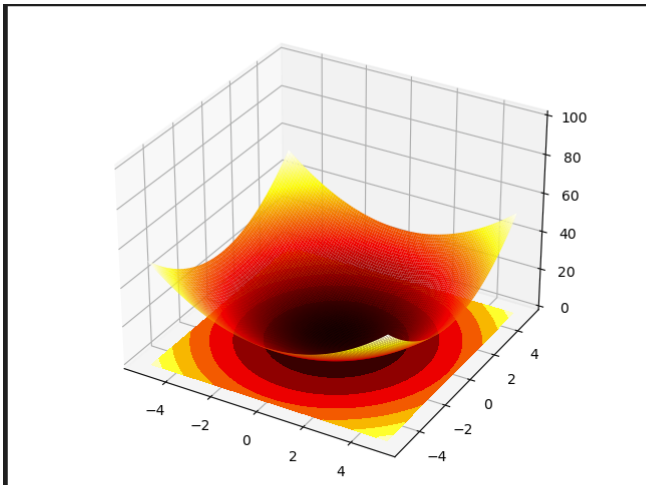
  def function_four(x):  # 范围: -65.536<= x <= 65.536
      aS = np.array(
          [
              [-32, -16, 0, 16, 32,
               -32, -16, 0, 16, 32,
               -32, -16, 0, 16, 32,
               -32, -16, 0, 16, 32,
               -32, -16, 0, 16, 32],
              [-32, -32, -32, -32, -32,
               -16, -16, -16, -16, -16,
               0, 0, 0, 0, 0,
               16, 16, 16, 16, 16,
               32, 32, 32, 32, 32]
          ]
      )
      bS = np.zeros(25)
      for j in range(0, 25):
          bS[j] = np.sum((x.T-aS[:, j])**6)
      return (1/500+np.sum(1/(np.arange(1, 25+1)+bS)))**(-1)

  def function_six(x1,x2):  # 范围: -5.12<= x <= 5.12
      return 20+x1*x1+x2*x2-10*np.cos(2*np.pi*x1)-10*np.cos(2*np.pi*x2)
  ```
- 定义画图函数
  ```

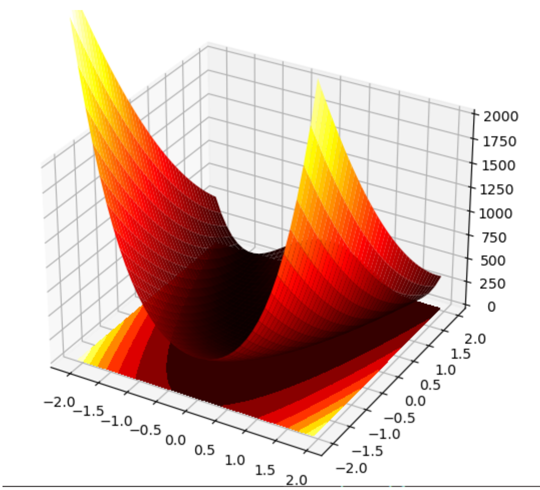
```python
fig = plt.figure()
ax = Axes3D(fig)
X = np.arange(-2.048, 2.048, 0.1) # 这里用第2个函数的范围为例子，其他代入进去即可
Y = np.arange(-2.048, 2.048, 0.1)
X, Y = np.meshgrid(X, Y)
Z = 100 * (X ** 2 - Y) ** 2 + (1 - X) ** 2

ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=plt.cm.hot)
ax.contourf(X, Y, Z, zdir='z', offset=-2, cmap=plt.cm.hot)
ax.set_zlim(0, 200)
plt.show()
```

- 大致图像如下：

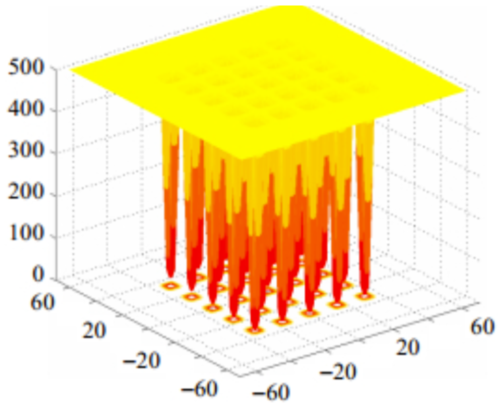
  - DeJong1



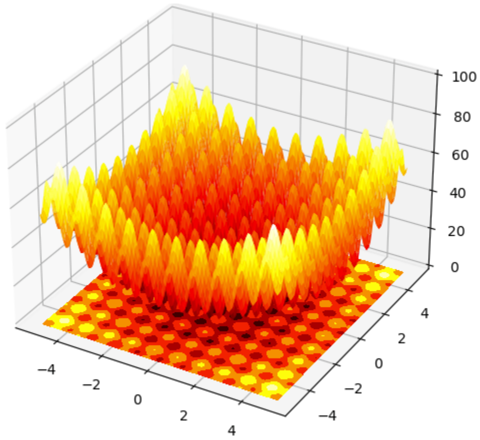
  - DeJong2



  - DeJong4



- DeJong6



4. 定义 **GA** 的相关函数

- 二进制映射：首先传入函数的定义域范围，建立二进制编码到实数的映射

```
def reflect_fuc(each, func_bound):
    x_, y_ = 0, 0
    for i in range(10):
        x_ += each[0][i]*(2**i)
    for i in range(10):
        y_ += each[1][i]*(2**i)
    fin_x = (x_/(2**10-1))*(func_bound[1]-func_bound[0])+func_bound[0]
    fin_y = (y_/(2**10-1))*(func_bound[1]-func_bound[0])+func_bound[0]
    return fin_x, fin_y
```

- 适应度计算：传入初始的种群，函数以及函数的定义域，先通过映射，映射的值带入原函数即是适应度

```
def get_fitness(first_population, function, func_bound):
    each_fitness = []
    for each in first_population:
        x_1, x_2 = reflect_fuc(each, func_bound)  # 映射得到x1,x2
        each_fitness.append(function(x_1, x_2))  # 添加到fitness
    return each_fitness
```

- 交叉变异选择函数：轮盘赌函数

```
def rws_algoritm(first_population, fitness, n):  # 定义轮盘赌算法
    next_population = []  # 定义下一个子代
    sum_ = sum(fitness)  # 获取所有的适应度的和

    p_ = ((sum_-fitness)/sum_) / (len(fitness)-1)  # 获得概率

    idx = np.random.choice(np.arange(len(first_population)), size=n, replace=True, p=p_)

    for i in idx:
        next_population.append(first_population[i])
    return next_population
```

- 变异函数：随机初始化一个数，比他大不发生变化 比他小就发生交叉交换，然后初始化一个值，把他替换掉

```
def mutation(first_population, mop):  # 变异
    next_population = []  # 定义下一代
    for each in first_population:  # 遍历
        res = each  # 子代变成父代(进行更新)
        if np.random.rand() < mop:  # 随机初始化一个数 比他大不发生变化 比他小就发生交叉交换
            # 定义x_mu_place 是发生变异的位置
            x_mu_place = np.random.randint(0, len(each[0]))
            res[0][x_mu_place] = abs(res[0][x_mu_place]-1)

        if np.random.rand() < mop:  # 随机初始化一个数 比他大不发生变化 比他小就发生交叉交换
            # 同理, y_place 也是一样的。
            y_mu_place = np.random.randint(0, len(each[1]))
            res[1][y_mu_place] = abs(res[1][y_mu_place]-1)

        next_population.append(res)
    return next_population
```

- 交叉与变异：在变异的时候加入了交叉元素。我们随机定义一个数，$np.random.rand()$ 。如果这个值比我们自己设置的 cmp （交叉编译概率）来得大，就不会发生；反之，会发生交叉变异。我们设置随机的长度 $randomLength$ ，模拟在一个长度为 n 的染色体上随机选取一段长度 x，然后把两个染色体这个片段进行交换，就可以得到新的子代。

```
def cross_mutation(first_population, cmp):  # 交叉与变异
    next_population = []  # 定义下一代
    for each in first_population:  # 遍历
        res = each  # 子代变成父代
        if np.random.rand() < cmp:  # 随机初始化一个数 比他大不发生变化 比他小就发生交叉交
换
            # 随机从first_population中生成一段，然后进行交换
            random_length = first_population[np.random.randint(
                0, len(first_population))]  # 另一个

            # 定义随机生成的x交叉点和y交叉点
            x_c_m_pos = np.random.randint(0, len(random_length[0]))
            y_c_m_pos = np.random.randint(0, len(random_length[1]))

            for i in range(x_c_m_pos, len(random_length[0])):
                res[0][i] = random_length[0][i]  # 赋值,剪切之后交换
            for j in range(y_c_m_pos, len(random_length[1])):
                res[1][j] = random_length[1][j]  # 赋值,剪切之后交换

        next_population.append(res)
    return next_population
```

- 自然选择：精英主义，获取到最好的值，接下来进行变异，交叉和变异，轮盘选择，再进行fitness的测试，每一个产生的next_作为下一个输入的种群，如此反复即可

```
def nature_selection(function, parent, cop, mop, fun_bound):  # 自然选择

    next_population, fitness, best_best = [], [], []  # 下一代 && 适应度 && 精英

    fitness = get_fitness(parent, function, fun_bound)  # 获取适应度
    best_idx = fitness.index(np.min(fitness))  # 找到最小的那个
    best_best = parent[best_idx].copy()  # 找到最好的那个值，下标，当作是精英

    next_population = mutation(parent, mop)
    next_population = cross_mutation(next_population, cop)
    next_population = rws_algoritm(
        next_population, fitness, len(next_population)-1)

    next_population.append(best_best)  # 添加精英进去

    fitness = get_fitness(next_population, function, fun_bound)  # 再一次获取适应度
```

```python
    return next_population
```

5.主函数的测试与画图

函数一：

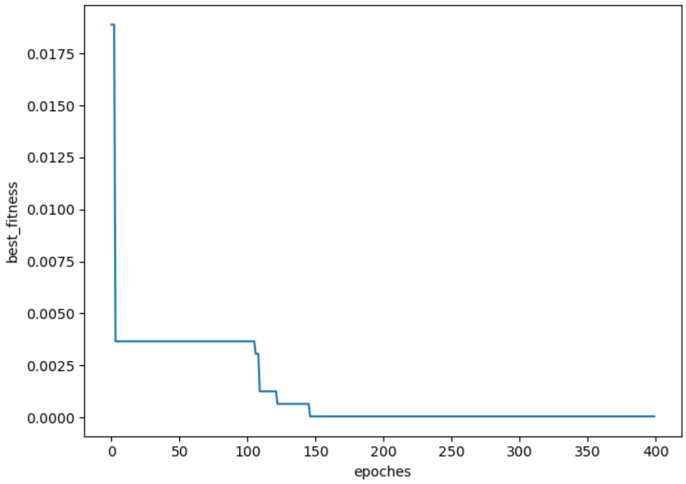
- 测试函数，求出平均适应度，最佳适应度，最坏适应度

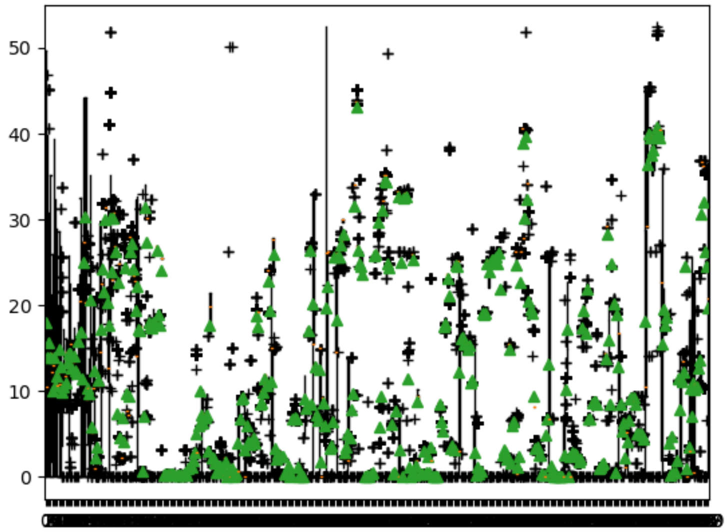
```python
for i in range(max_epochs):    # 最大的迭代次数
    first_population = nature_selection(
        function_one, first_population, _cmp, _mop, fun_one_bound)
    one_fitness = get_fitness(first_population, function_one, fun_one_bound)
    all_fitness.append(one_fitness)
    best_fitness.append(np.min(one_fitness))

print("finish")

print("average:{}".format(sum(best_fitness)/len(best_fitness)))
print("max_element:{}".format(max(best_fitness)))
print("min_element:{}".format(min(best_fitness)))
```

```
finish
average:0.0054290985304754945
max_element:0.028505647908466915
min_element:0.0008516625912898801
```

- 绘制图像

```python
plt.plot(np.arange(max_epoches),best_fitness)
plt.xlabel("epoches")
plt.ylabel("best_fitness")
plt.show()
```



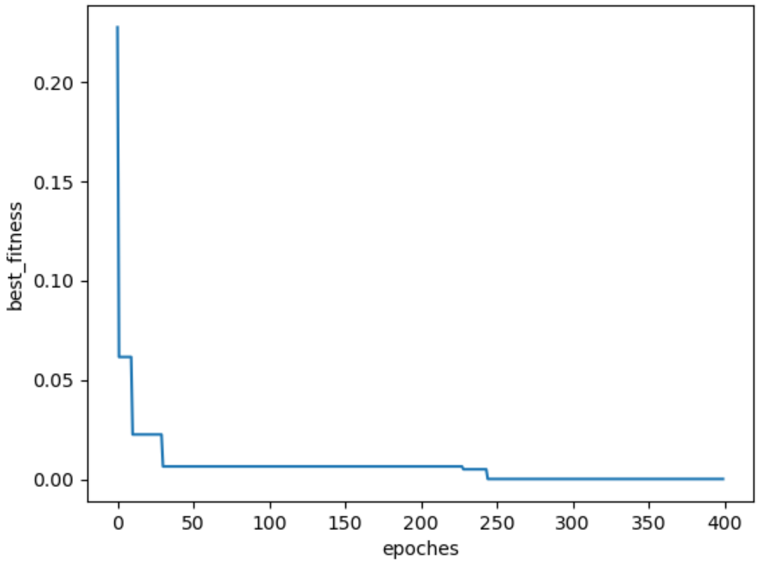


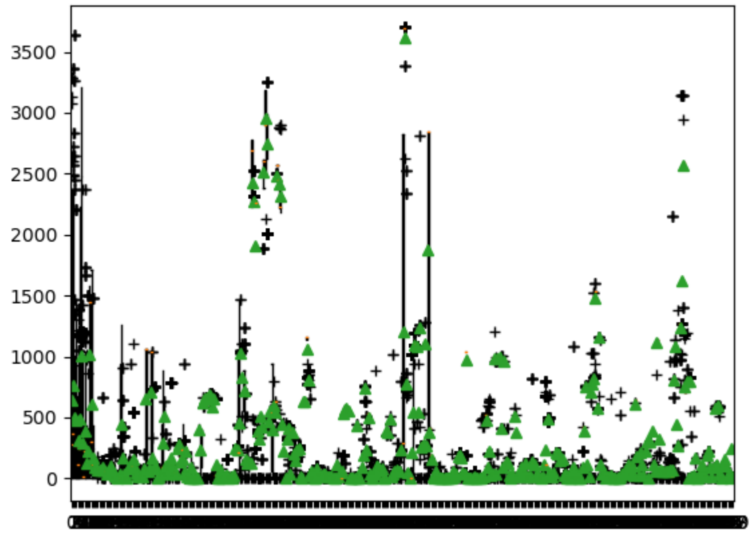

```
average:0.013822483856634643
max_element:0.22989880184877856
min_element:0.0006512713933393305
```

- 结论：大致在 (0,0) 处取得最小值 0

函数二：

```
finish
average:0.006501269053881985
max_element:0.22766862261457968
min_element:0.00010578186596766037
```



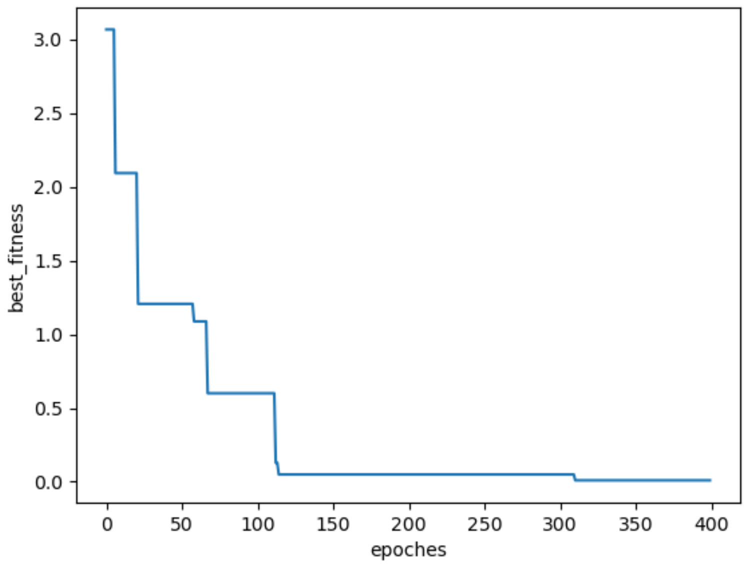


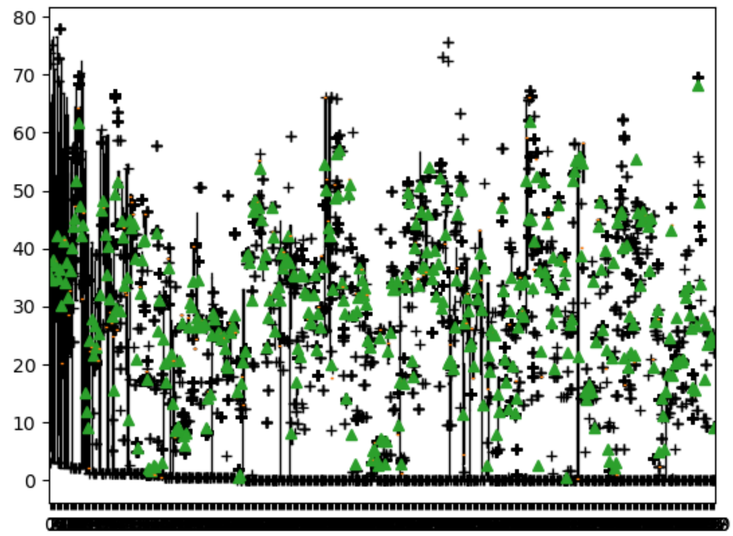
大致在 (1,1) 处取得最小值 0

函数三：

函数四：

```
finish
average:0.3553705047097828
max_element:3.0677520896260067
min_element:0.009938192151459191
```





大致在[0,0,0....,0] 处取得最小值 0