1 遗传算法的代码与分析

1.1 引言

1.1.1 什么是遗传算法

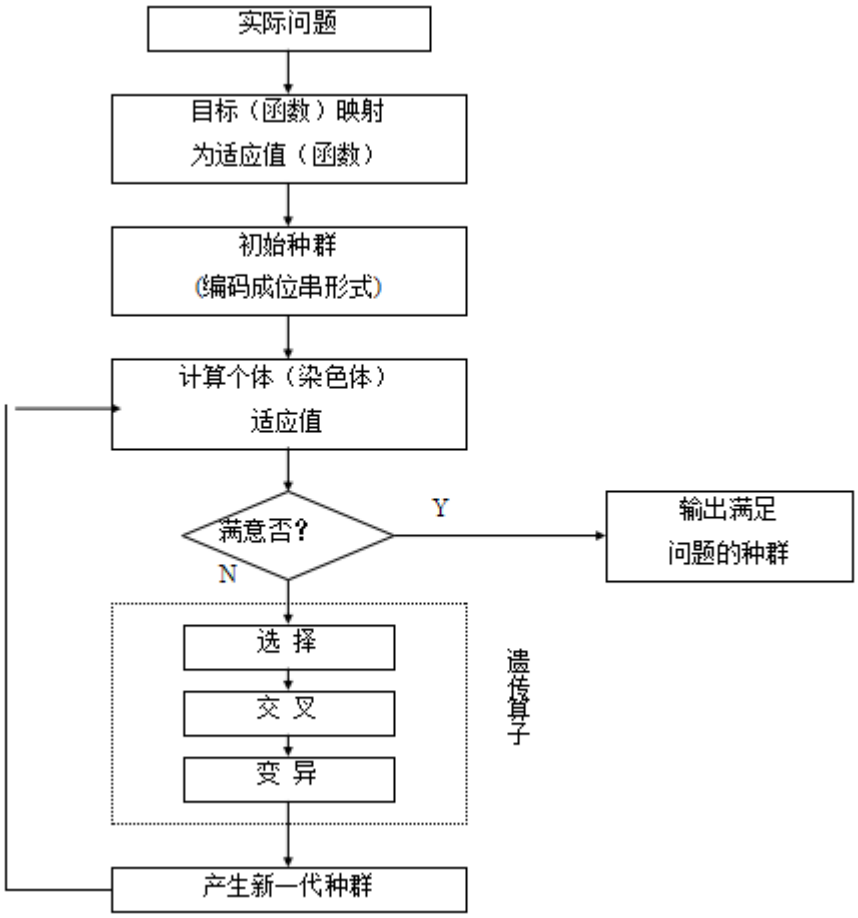
遗传算法是模拟达尔文生物进化论的自然选择和遗传学机理的生物进化过程的计算模型，是一种通过模拟自然进化过程搜索最优解的方法。

采用概率化的寻优方法，不需要确定的规则就能自动获取和指导优化的搜索空间，自适应地调整搜索方向。

1.1.2 遗传算法的执行过程

遗传算法首先在初代种群产生之后，按照适者生存和优胜劣汰的原理，逐代演化产生出越来越好的近似解，在每一代，根据问题域中个体的适应度大小选择个体，并借助于自然遗传学的遗传算子进行组合交叉和变异，产生出代表新的解集的种群。

这个过程将导致种群像自然进化一样的后生代种群比前代更加适应于环境，末代种群中的最优个体经过解码，可以作为问题近似最优解。



1.2 遗传算法代码分析

题目中 Ackley 函数的参数 d 表示特征值x_i的数量，考虑到需要画出 3d 演化图，因此选取 d 为 2，因此有特征值x和y。

1.2.1 设置初始种群

首先生成 200 个随机的x, y，将x, y坐标对带入要求解的函数$Ackley(x, y)$中，题目要求函数最小值，根据适者生存，定义使得函数值$Ackley(x, y)$越小的x, y越适合环境，这些适应环境的x, y对更有可能被保留下来，而保留下来的点经过繁殖产生新的点，根据上述规则进化，最后留下的大部分点都是适应环境的点，即在函数最低点附近。

```
DNA_SIZE = 26  # 个体编码长度

POPULATION_SIZE = 200  # 种群大小
GENERATION_NUMBER = 200  # 世代数目
CROSS_RATE = 0.8  # 交叉率
VARIATION_RATE = 0.01  # 变异率

X_RANGE = [-32.768, 32.768]  # X 范围
Y_RANGE = [-32.768, 32.768]  # Y 范围

# 生成随机种群矩阵，这里 DNA_SIZE * 2 是因为种群矩阵要拆分为 x 和 y 矩阵，单条 DNA（染色体、个体）长度为 24
# 若视 x 和 y 为等位基因，x 和 y 组成染色体对，共同影响个体，这里巧妙地与遗传信息对应起来
population_matrix = np.random.randint(2, size=(POPULATION_SIZE, DNA_SIZE * 2))
```

1.2.2 编码与解码操作

在遗传算法中，将可能解编码为一个向量表示，常见做法是将十进制的数编码为二进制，上述过程称为编码。将可能解编码后的二进制串叫做染色体，染色体即可能解的二进制编码表示。

在本程序中，由于生成的初始种群即二进制数，因此省略了编码过程。

与编码相对的操作称为解码，目标是求一个逆映射将二进制串映射到x, y的定义域内，为将二进制串映射到指定范围，首先将二进制串按权展开，将二进制数转化为十进制数，并将转换后的实数压缩到[0,1]之间的一个小数，再将上述结果映射到x, y的定义域区间。

```
# 解码 DNA 个体

# @param population_matrix 种群矩阵
# @return population_x_vector, population_y_vector 种群 x 向量，种群 y 向
```

```
量
def decodingDNA(population_matrix):
    x_matrix = population_matrix[:, 1::2]  # 矩阵分割，行不变，抽取奇数列作为 x 矩阵
    y_matrix = population_matrix[:, 0::2]  # 矩阵分割，行不变，抽取偶数列作为 y 矩阵
    # 解码向量，用于二进制转十进制，其值为[2^23 2^22 ... 2^1 2^0]，对位相乘累加，二进制转十进制的基础方法
    decoding_vector = 2 ** np.arange(DNA_SIZE)[::-1]
    # 种群 x 向量，由二进制转换成十进制并映射到 x 区间
    population_x_vector = x_matrix.dot(decoding_vector) / (2 ** DNA_SIZE - 1)\
                          * (X_RANGE[1] - X_RANGE[0]) + X_RANGE[0]
    # 种群 y 向量，由二进制转换成十进制并映射到 y 区间
    population_y_vector = y_matrix.dot(decoding_vector) / (2 ** DNA_SIZE - 1)\
                          * (Y_RANGE[1] - Y_RANGE[0]) + Y_RANGE[0]
    return population_x_vector, population_y_vector
```

1.2.3 适应度与选择操作

在得到种群后，需要根据适者生存规则将优秀个体保存下来，同时淘汰掉那些不适应环境的个体。在题目中，即保留靠近最低点的个体。而在做个体选择之前，需要计算所有个体对环境的适应度，在本程序中使用 Ackley 函数本身进行适应度的计算，即直接用可能解对应的函数值的大小进行评估。

```
# 问题函数 阿克莱

# @param x x 坐标
# @param y y 坐标
# @return z 函数值
def ackleyFunc(x, y):
    a = 20
    b = 0.2
    c = 2 * np.pi
    d = 2
    sum1 = x * x + y * y
    sum2 = np.cos(c * x) + np.cos(c * y)
    term1 = -a * np.exp(-b * np.sqrt(1 / d * sum1))
    term2 = -np.exp(1 / d * sum2)
    z = term1 + term2 + a + np.exp(1)
    return z

# 获取适应度向量
# @param population_matrix 种群矩阵
# @return fitness_vector 适应度向量
```

```
def getFitnessVector(population_matrix):
    population_x_vector, population_y_vector =
decodingDNA(population_matrix)  # 获取种群 x 和 y 向量
    fitness_vector = ackleyFunc(population_x_vector,
population_y_vector)  # 获取适应度向量
    # print(fitness_vector)
    fitness_vector = np.max(fitness_vector) - fitness_vector + 1e-3 #
保证适应度大于 0 并且原适应度越小 得到的值越大
    return fitness_vector
```

根据适者生存规则，在求最小值问题上，函数值越小的可能解对应的适应度应该越大，同时适应度也不能为负值，将最大预测值减去适应度，这样能够保证适应度大于等于 0，同时适应度作为选择函数的一个概率，如果适应度为 0，其对应的概率也为 0，表示该个体不可能在选择中保留下来，这不符合算法思想，遗传算法不绝对否定谁也不绝对肯定谁，所以最后加上了一个很小的正数。

在计算所有个体的适应度后，即可根据适应度大小选择个体，选择则是根据新个体的适应度进行，但同时不意味着完全以适应度高低为导向，因为单纯选择适应度高的个体将可能导致算法快速收敛到局部最优解而非全局最优解。作为折中，遗传算法依据原则：适应度越高，被选择的机会越高，而适应度低的，被选择的机会就低。

```
# 自然选择

# @param population_matrix 种群矩阵
# @param fitness_vector 适应度向量
# @return population_matrix[index_array] 选择后的种群
def naturalSelection(population_matrix, fitness_vector):
    index_array = np.random.choice(np.arange(POPULATION_SIZE),  # 被选
取的索引数组
                                   size=POPULATION_SIZE,  # 选取数量
                                   replace=True,  # 允许重复选取
                                   p=fitness_vector / fitness_vector.sum())
# 数组每个元素的获取概率
    # print(population_matrix[index_array])
    return population_matrix[index_array]
```

1.2.4 交叉与变异操作

交叉和变异是遗传算法中的核心模块。通过选择操作后，得到了一组尚可的基因，通过繁殖后代，即交叉与变异过程来产生新的可能更好的基因。

交叉是指各个个体是由父亲和母亲两个个体繁殖产生，子代个体的 DNA 获得了一半父亲的 DNA，一半母亲的 DNA。

通过交叉子代获得了一半来自父亲一半来自母亲的 DNA，但是子代自身可能发生变异，使得其 DNA 即不来自父亲，也不来自母亲，在某个位置上发生随机改变，通常就是改变 DNA 的一个二进制位。

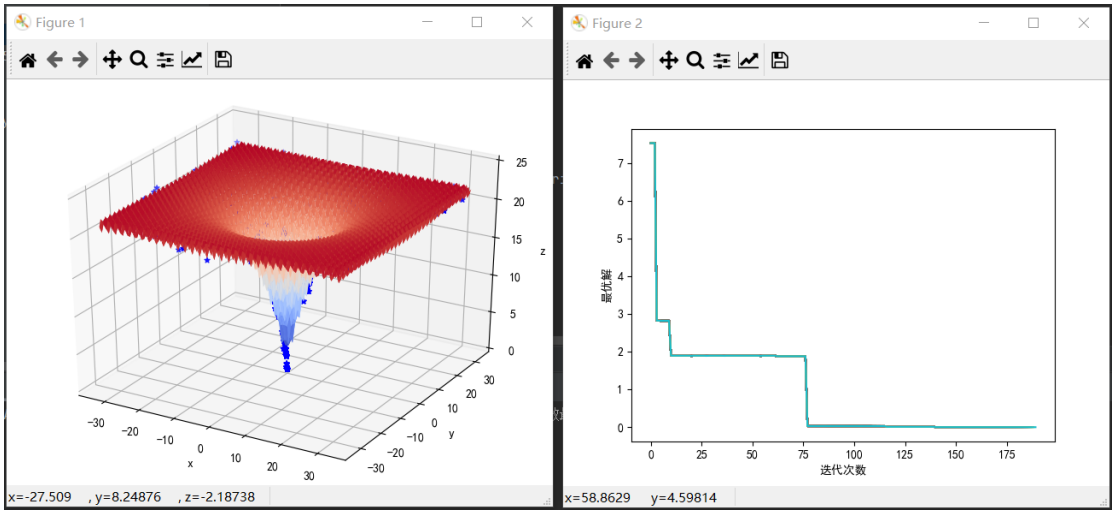
```
# DNA 交叉

# @param child_DNA 孩子 DNA
# @param population_matrix 种群矩阵
def DNACross(child_DNA, population_matrix):
    # 概率发生 DNA 交叉
    if np.random.rand() < CROSS_RATE:
        mother_DNA =
population_matrix[np.random.randint(POPULATION_SIZE)]  # 种群中随机选择
一个个体作为母亲
        cross_position = np.random.randint(DNA_SIZE * 2)  # 随机选取交叉
位置
        child_DNA[cross_position] = mother_DNA[cross_position]  # 孩子获
得交叉位置处母亲基因


# DNA 变异
# @param child_DNA 孩子 DNA
def DNAVariation(child_DNA):
    # 概率发生 DNA 变异
    if np.random.rand() < VARIATION_RATE:
        variation_position = np.random.randint(DNA_SIZE * 2)  # 随机选取
变异位置
        child_DNA[variation_position] = child_DNA[variation_position]
^ 1  # 异或门反转二进制位


# 更新种群
# @param population_matrix 种群矩阵
# @return new_population_matrix 更新后的种群矩阵
def updatePopulation(population_matrix):
    new_population_matrix = []  # 声明新的空种群
    # 遍历种群所有个体
    for father_DNA in population_matrix:
        child_DNA = father_DNA  # 孩子先得到父亲的全部 DNA（染色体）
        DNACross(child_DNA, population_matrix)  # DNA 交叉
        DNAVariation(child_DNA)  # DNA 变异
        new_population_matrix.append(child_DNA)  # 添加到新种群中
    new_population_matrix = np.array(new_population_matrix)  # 转化数组
    return new_population_matrix
```

```
# 迭代过程
for i in range(GENERATION_NUMBER):
    population_x_vector, population_y_vector =
decodingDNA(population_matrix)  # 获取种群 x 和 y 向量
    # 核心模块
    population_matrix = updatePopulation(population_matrix)  # 更新种群
    fitness_vector = getFitnessVector(population_matrix)  # 获取适应度向
量
    population_matrix = naturalSelection(population_matrix,
fitness_vector)  # 自然选择
```

2 结果展示与理解

2.1 结果展示

通过 Python 绘图程序，分别将结果绘制成了种群迭代图（动图）与迭代数与最优解的折线图（动图），如下图所示：



输出的函数最优值结果为：

```
E:\ProgramData\Anaconda3\python.exe G:/BaiduNetdiskDownload/遗传算法/遗传算法解决函数最小值问题/ga/ga1.py
最佳适应度为： 0.0011497216793673317
最优基因型为： [0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1
 1 1 1 1 0 1 0 0 0 1 1 0 0 0]
最优基因型十进制表示为： (-0.0010112304838187924, -0.000278808597904856)
最优函数值为： 0.0029962121557152166

Process finished with exit code 0
```

2.2 遗传算法的难点及编程心路历程

2.2.1 遗传算法的难点

在编写遗传算法求解二元函数的最小值程序中，我认为最难的部分是理解交叉、变异、选择的原理。

需要注意的是：

1. 在繁殖时选取父亲与母亲各一半的 DNA，这里的一半并不是真正的一半，这个位置可以是染色体的任意位置，是随机产生的。

2. 繁殖后代并不能保证每个后代个体的基因都比上一代优秀，这时需要继续通过选择过程来让满足条件的个体保留下来，从而完成进化，不断迭代上述过程，种群中的个体逐步完成进化。

3. 本程序中的个体编码长度取的是 26，编码长度越长，一般来说能表示得越精确。

4. 本程序中的迭代次数选取的是 200，理论上迭代次数越大，得到的结果就越准确，但我认为迭代次数要在一个合理的范围内选取，不能太高。实际上没有结论证明迭代次数超过多少代后得到的解是最优的，在连续迭代了 n 次后，最佳个体都没有变化，那么可以认为遗传算法已经收敛了，或者陷入了局部最优解了，这个时候得到的解就可以认为是最优解了。

5. 本程序中求解的问题为二元函数的最小值，在适应度的选取上，需要注意不能像求解函数最大值一样，把函数值越高直接映射为适应度越高，而是要通过一个用最大适应度减去当前适应度的转化，并加上一个极小的正数，最后得到的适应度作为选择概率的参考，要求需要是大于 0 的正数。

2.2.2 编程中的细节

由于本程序的结果是绘制两幅动态图，在迭代绘图的过程中遇到了许多 bug，例如：无法把两幅图分别绘制在不同 figure 上、polt()函数的 label 不能显示中文等细节问题，但通过参考博客内容都解决了。

3 源程序附录及 github 地址

3.1 源程序

采用 Python 语言编写：

```
import numpy as np

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

```
DNA_SIZE = 26  # 个体编码长度
POPULATION_SIZE = 200  # 种群大小
GENERATION_NUMBER = 200  # 世代数目
CROSS_RATE = 0.8  # 交叉率
VARIATION_RATE = 0.01  # 变异率
X_RANGE = [-32.768, 32.768]  # X 范围
Y_RANGE = [-32.768, 32.768]  # Y 范围


# 问题函数 阿克莱
# @param x x 坐标
# @param y y 坐标
# @return z 函数值
def ackleyFunc(x, y):
    a = 20
    b = 0.2
    c = 2 * np.pi
    d = 2
    sum1 = x * x + y * y
    sum2 = np.cos(c * x) + np.cos(c * y)
    term1 = -a * np.exp(-b * np.sqrt(1 / d * sum1))
    term2 = -np.exp(1 / d * sum2)
    z = term1 + term2 + a + np.exp(1)
    return z


# 初始化图
# @param ax 3D 图像
def init3DGraph(ax):
    x_sequence = np.linspace(*X_RANGE, 100)  # 创建 x 等差数列
    y_sequence = np.linspace(*Y_RANGE, 100)  # 创建 y 等差数列
    x_matrix, y_matrix = np.meshgrid(x_sequence, y_sequence)  # 生成 x
和 y 的坐标矩阵
    z_matrix = ackleyFunc(x_matrix, y_matrix)  # 生成 z 坐标矩阵
    # 创建曲面图,行跨度为 1，列跨度为 1，设置颜色映射
    ax.plot_surface(x_matrix, y_matrix, z_matrix, rstride=1,
cstride=1, cmap=plt.get_cmap('coolwarm'))
    ax.set_zlim(0, 25)  # 自定义 z 轴范围
    ax.set_xlabel('x')  # 设置 x 坐标轴标题
    ax.set_ylabel('y')  # 设置 y 坐标轴标题
    ax.set_zlabel('z')  # 设置 z 坐标轴标题
    plt.pause(2)  # 暂停 2 秒
    plt.show()  # 显示图
```

```
# 解码 DNA 个体
# @param population_matrix 种群矩阵
# @return population_x_vector, population_y_vector 种群 x 向量，种群 y 向量
def decodingDNA(population_matrix):
    x_matrix = population_matrix[:, 1::2]  # 矩阵分割，行不变，抽取奇数列作为 x 矩阵
    y_matrix = population_matrix[:, 0::2]  # 矩阵分割，行不变，抽取偶数列作为 y 矩阵
    # 解码向量，用于二进制转十进制，其值为[2^23 2^22 ... 2^1 2^0]，对位相乘累加，二进制转十进制的基础方法
    decoding_vector = 2 ** np.arange(DNA_SIZE)[::-1]
    # 种群 x 向量，由二进制转换成十进制并映射到 x 区间
    population_x_vector = x_matrix.dot(decoding_vector) / (2 ** DNA_SIZE - 1)\
                          * (X_RANGE[1] - X_RANGE[0]) + X_RANGE[0]
    # 种群 y 向量，由二进制转换成十进制并映射到 y 区间
    population_y_vector = y_matrix.dot(decoding_vector) / (2 ** DNA_SIZE - 1)\
                          * (Y_RANGE[1] - Y_RANGE[0]) + Y_RANGE[0]
    return population_x_vector, population_y_vector


# DNA 交叉
# @param child_DNA 孩子 DNA
# @param population_matrix 种群矩阵
def DNACross(child_DNA, population_matrix):
    # 概率发生 DNA 交叉
    if np.random.rand() < CROSS_RATE:
        mother_DNA = population_matrix[np.random.randint(POPULATION_SIZE)]  # 种群中随机选择一个个体作为母亲
        cross_position = np.random.randint(DNA_SIZE * 2)  # 随机选取交叉位置
        child_DNA[cross_position] = mother_DNA[cross_position]  # 孩子获得交叉位置处母亲基因


# DNA 变异
# @param child_DNA 孩子 DNA
def DNAVariation(child_DNA):
    # 概率发生 DNA 变异
    if np.random.rand() < VARIATION_RATE:
```

```
        variation_position = np.random.randint(DNA_SIZE * 2)  # 随机选取
变异位置
        child_DNA[variation_position] = child_DNA[variation_position]
^ 1  # 异或门反转二进制位


# 更新种群
# @param population_matrix 种群矩阵
# @return new_population_matrix 更新后的种群矩阵
def updatePopulation(population_matrix):
    new_population_matrix = []  # 声明新的空种群
    # 遍历种群所有个体
    for father_DNA in population_matrix:
        child_DNA = father_DNA  # 孩子先得到父亲的全部 DNA（染色体）
        DNACross(child_DNA, population_matrix)  # DNA 交叉
        DNAVariation(child_DNA)  # DNA 变异
        new_population_matrix.append(child_DNA)  # 添加到新种群中
    new_population_matrix = np.array(new_population_matrix)  # 转化数组
    return new_population_matrix


# 获取适应度向量
# @param population_matrix 种群矩阵
# @return fitness_vector 适应度向量
def getFitnessVector(population_matrix):
    population_x_vector, population_y_vector =
decodingDNA(population_matrix)  # 获取种群 x 和 y 向量
    fitness_vector = ackleyFunc(population_x_vector,
population_y_vector)  # 获取适应度向量
    # print(fitness_vector)
    fitness_vector = np.max(fitness_vector) - fitness_vector + 1e-3 #
保证适应度大于 0 并且原适应度越小 得到的值越大 #fitness_vector -
np.min(fitness_vector) + 1e-3  # 适应度修正，保证适应度大于 0
    return fitness_vector


# 自然选择
# @param population_matrix 种群矩阵
# @param fitness_vector 适应度向量
# @return population_matrix[index_array] 选择后的种群
def naturalSelection(population_matrix, fitness_vector):
    index_array = np.random.choice(np.arange(POPULATION_SIZE),  # 被选
取的索引数组
                                   size=POPULATION_SIZE,  # 选取数量
```

```
                                replace=True,  # 允许重复选取
                                p=fitness_vector / fitness_vector.sum())
# 数组每个元素的获取概率
    # print(population_matrix[index_array])
    return population_matrix[index_array]

# 获取当前迭代最优解
def getOptSol(population_matrix):
    fitness_vector = getFitnessVector(population_matrix)  # 获取适应度向
量
    optimal_fitness_index =
np.argmax(fitness_vector)#np.argmax(fitness_vector)  # 获取最大适应度索
引

    return ackleyFunc(population_x_vector[optimal_fitness_index],
population_y_vector[optimal_fitness_index])

# 打印结果
# @param population_matrix 种群矩阵
def printResult(population_matrix):
    fitness_vector = getFitnessVector(population_matrix)  # 获取适应度向
量
    optimal_fitness_index =
np.argmax(fitness_vector)#np.argmax(fitness_vector)  # 获取最大适应度索
引
    print('最佳适应度为: ', fitness_vector[optimal_fitness_index])
    print('最优基因型为: ', population_matrix[optimal_fitness_index])
    population_x_vector, population_y_vector =
decodingDNA(population_matrix)  # 获取种群 x 和 y 向量
    print('最优基因型十进制表示为: ',
          (population_x_vector[optimal_fitness_index],
population_y_vector[optimal_fitness_index]))
    print('最优函数值为: ',
          ackleyFunc(population_x_vector[optimal_fitness_index],
population_y_vector[optimal_fitness_index]))



if __name__ == '__main__':
    fig = plt.figure()  # 创建空图像
    ax = Axes3D(fig)  # 创建 3D 图像
    plt.ion()  # 切换到交互模式绘制动态图像
    init3DGraph(ax)  # 初始化图
    # 生成随机种群矩阵，这里 DNA_SIZE * 2 是因为种群矩阵要拆分为 x 和 y 矩阵，单条
```

```
DNA（染色体、个体）长度为 24
    # 若视 x 和 y 为等位基因，x 和 y 组成染色体对，共同影响个体，这里巧妙地与遗传信息对应起来
    population_matrix = np.random.randint(2, size=(POPULATION_SIZE, DNA_SIZE * 2))

    #画折线图
    # plt.axis([0, 100, 0, 1])
    # plt.ion()
    optSol = [] #最优解
    iter = [] #迭代次数
    # 迭代 50 世代
    for i in range(GENERATION_NUMBER):
        population_x_vector, population_y_vector = decodingDNA(population_matrix)  # 获取种群 x 和 y 向量
        # 绘制散点图，设置颜色和标记风格
        ax.scatter(population_x_vector,
                   population_y_vector,
                   ackleyFunc(population_x_vector, population_y_vector),
                   c='b',
                   marker='*')
        plt.show()  # 显示图
        plt.pause(0.01)  # 暂停 0.1 秒
        #绘制迭代-最优解图
        iter.append(i)
        optSol.append(getOptSol(population_matrix))
        # print(optSol)
        # print(iter)
        plt.figure(2)
        plt.plot(iter, optSol)
        # 解决中文显示问题
        plt.rcParams['font.sans-serif'] = ['SimHei']
        plt.rcParams['axes.unicode_minus'] = False
        plt.xlabel('迭代次数')
        plt.ylabel('最优解')
        plt.show()
        #核心模块
        population_matrix = updatePopulation(population_matrix)  # 更新种群
        fitness_vector = getFitnessVector(population_matrix)  # 获取适应度向量
        population_matrix = naturalSelection(population_matrix, fitness_vector)  # 自然选择
```

```
        if(i==GENERATION_NUMBER-1):
            iter.append(i+1)
            optSol.append(getOptSol(population_matrix))

    # print(optSol)
    # print(iter)
    # plt.figure(2)
    # plt.plot(iter, optSol)
    # plt.xlabel('Group')
    # plt.ylabel('Count')
    # plt.show()

    printResult(population_matrix)  # 打印结果
```

3.2 github 地址

本程序的 github 地址为:

https://github.com/StrontiaForWork/machineLearning/homeWork