函数接口及进化算法模板

在“快速入门”章节中我们展示了利用 Geatpy 简单进化算法模板解决多元单峰函数最值的搜索问题。在那时，我们把所有的代码都放在一个脚本文件里。对于解决简单的问题这种方式能够轻松胜任，但非常不利于重构和把 Geatpy 与其他算法或项目进行融合。本章我们将介绍如何通过编写函数接口以及精简的进化算法模板来实现低耦合的编程。

1. 函数接口

函数接口是目标函数和罚函数的统称，一般写在独立的 Python 源文件里，你也可以直接把它们定义在脚本中。目标函数传入种群表现型矩阵 $Phen$，计算各个个体的目标函数值，返回种群的目标函数值列向量 $ObjV$；罚函数传入种群表现型矩阵 $Phen$ 和适应度值列向量 $FitnV$，找出不符合约束条件的个体，并“加以惩罚”，降低其适应度，最后返回新的适应度列向量 $NewFitnV$。

(注：上述变量的命名只是惯用命名，你可以修改其命名。)

另外，有些时候为了方便，我们可以在定义目标函数的同时定义罚函数，即把罚函数的功能和目标函数写在同一个函数内，尤其是在多目标优化问题中，经常这么来实现。

例：设计目标函数和罚函数，分别命名为 aimfuc 和 punishing。其中目标函数实现的是 2 个变量的平方和，罚函数实现的是惩罚值为 0 的变量。

```python
"""目标函数aimfuc.py"""
import numpy as np
def aimfuc(Phen): # 传入种群染色体矩阵解码后的基因表现型矩阵
    x1 = Phen[:, 0] # 从Phen中片取得到x1变量
    x2 = Phen[:, 1]
    ObjV = x1*x1 + x2*x2
    return np.array([ObjV]).T # 为什么要这么复杂?
```

代码标记处的返回语句为什么要这么复杂？这是 numpy 的 array 类型所致的。传入的 Phen 代表种群染色体的表现型，它是一个矩阵。但是在用片取的方法从 Phen 中取出 x_1 变量时，numpy 会自动将取出的结果降维成一维的行向量。x_2 也是如此。在进行 x1*x1 + x2*x2 的计算后，ObjV 也跟着是一维行向量，但我们需要返回的是一个列向量。因此需要通过 np.array([ObjV]).T 语句进行转换。

假设 Phen 的值如下：

$$\text{Phen}=\begin{pmatrix}1 & 2\\ 4 & 0\\ 3 & 4\end{pmatrix}$$

那么调用 aimfuc(Phen) 后，将会得到种群个体对应的目标函数值为：

$$\text{ObjV}=\begin{pmatrix}5\\ 16\\ 25\end{pmatrix}$$

进一步地理解 numpy 的 array 维度，我们执行 print(Phen.shape) 和 print(ObjV.shape) 输出它们的维度信息，可看到结果分别 (3, 2) 和 (3, 1)。说明 Phen 是 3 行 2 列的矩阵 (实际上是数组类型)；ObjV 是 3 行 1 列的列向量。

下面继续编写罚函数 punishing，对值为 0 的变量进行惩罚，使其适应度比当前种群最小值还要小至少 50%：

```python
"""罚函数punishing.py"""
import numpy as np
def punishing(Phen, FitnV):
    idx = np.where(Phen == 0)[0]
    FitnV[idx] = np.min(FitnV) // 2 # 取整除法
    return FitnV
```

假设上一个例子中，种群的适应度等于其目标函数值 ObjV。我们发现种群中第二个个体的表现型中出现了 0，因此我们代入罚函数对其进行惩罚：

```python
"""test.py"""
import numpy as np
from punishing import punishing
from aimfuc import aimfuc
# 创建Phen代表种群的基因表现型矩阵，注意array的维度
Phen = np.array([
    [1, 2],
    [4, 0],
    [3, 4]])
FitnV = aimfuc(Phen)
FitnV = punishing(Phen, FitnV)
print(FitnV) # 输出结果
```

输出结果为：

$$\text{FitnV}=\begin{pmatrix}5\\ 2\\ 25\end{pmatrix}$$

可见种群第二个个体成功地被“惩罚”，其适应度变成了 2。

注意事项：

在编写复杂的罚函数时，要注意一个很容易出错的地方，请看下面的例子：

$$\begin{aligned}&\min f_1(x)=-25(x_1-2)^2-(x_2-2)^2-(x_3-1)^2-(x_4-5)^2-(x_5-1)^2\\&\min f_2(x)=(x_1-1)^2+(x_2-1)^2+(x_3-1)^2+(x_4-1)^2+(x_5-1)^2\end{aligned}$$

$$s.t.\begin{cases}g_1(x)=x_1+x_2-2\geq 0\\ g_2(x)=6-x_1-x_2\geq 0\\ g_3(x)=2+x_1-x_2\geq 0\\ g_4(x)=2-x_1+3x_2\geq 0\\ g_5(x)=4-(x_3-3)^3-x_4\geq 0\\ g_6(x)=(x_5-3)^3+x_4-4\geq 0\end{cases}$$

$$0\leq x_i\leq 10\,(i=1,2,\cdots 5)$$

这是一个双目标优化问题，其中约束条件不再是简单的范围区间，而是多个变量的多个约束不等式。此时我们想让不符合约束条件的解对应的目标函数值 f_1 变大，同时 f_2 变小，于是可以设计以下罚函数：

$$f_i=[p_1f_1(x),p_2f_2(x)]$$

其中，若满足约束条件 $g_i(x)(i=1,2,...,6)$，取 $p_1=p_2=1$，反之，取 $p_1<0$ 和 $p_2>0$ 的随机数。

这里看上去并无问题，但实际上，我们不建议取 $p_1<0$ 的随机数，因为它会改变数值的符号。因为目标函数 f_1 是恒不大于 0 的，如果罚函数遍历所有约束条件来依次让 x_i 不满足约束条件的目标函数值乘上 p_1，那么此时会极有可能出现目标函数值多次被修改的情况，因为 $p_1<0$，所以着意味着这些目标函数值被反复变换正负号，这将导致罚函数反而将目标函数值变得更小的错误。对此有 3 种解决办法：一是在遍历各个约束条件时，记录已被惩罚的个体，并避免相同的个体多次被惩罚。二是重新设计约束条件，使不满足各个约束条件的集合的交集为空集，但这比较难实现。三是重新设计罚函数，把 $p_1<0$ 修改为 $0<p_1<1$。这样，既能让负的目标函数值变大，也能保持其正负号，避免上述问题的出现。

2. 进化算法模板

前面我们已经介绍过进化算法模板的概念和重要性。下面我们从头开始自定义一个遗传算法模板，命名为 mintemp1，并编写测试脚本，调用该遗传算法模板来解决搜索 $y=x_1^2+x_2^2$ 的最小值，其中 x_1 与 x_2 分别是 [-5, 0)U(0, 5] 以及 (2, 10] 的整数。

模板中调用的 Geatpy 内置函数的相关用法可以在”Geatpy 函数” 章节中找到详细的讲解。

```python
"""自定义遗传算法模板mintemp1.py"""
import numpy as np
import geatpy as ga # 导入geatpy库
import time

def mintemp1(AIM_M, AIM_F, PUN_M, PUN_F, ranges, borders, MAXGEN,
    NIND, SUBPOP, GGAP, selectStyle, recombinStyle, recopt, pm,
    maxormin):
    """==========================初始化配置==========================="""
    # 获取目标函数和罚函数
    aimfuc = getattr(AIM_M, AIM_F) # 获得目标函数
    punishing = getattr(PUN_M, PUN_F) # 获得罚函数
    FieldDR = ga.crtfld(ranges, borders) # 初始化区域描述器
    NVAR = ranges.shape[1] # 得到控制变量的个数
    # 定义进化记录器，初始值为nan
    pop_trace = (np.zeros((MAXGEN ,3)) * np.nan).astype('int64')
    # 定义变量记录器，记录控制变量值，初始值为nan
    var_trace = (np.zeros((MAXGEN ,NVAR)) * np.nan).astype('int64')
    """=========================开始遗传算法进化======================="""
    Chrom = ga.crtip(NIND, FieldDR) #
        根据区域描述器FieldDR生成整数型初始种群
    ObjV = aimfuc(Chrom) # 计算种群目标函数值
    start_time = time.time() # 开始计时
    # 开始进化！！
    for gen in range(MAXGEN):
        FitnV = ga.ranking(maxormin * ObjV) # 计算种群适应度
        FitnV = punishing(Chrom, FitnV) # 调用罚函数
        # 记录当代种群最优个体的目标函数值
        pop_trace[gen, 0] = ObjV[np.argmax(FitnV)]
        # 记录当代种群的适应度均值
        pop_trace[gen, 1] = np.sum(FitnV) / FitnV.shape[0]
        # 记录当代种群的最优个体的适应度值
        pop_trace[gen, 2] = np.max(FitnV)
        # 记录当代种群最优个体的变量值
        var_trace[gen, :] = Chrom[np.argmax(FitnV), :]
        # 进行遗传操作！！
        SelCh=ga.selecting(selectStyle,Chrom,FitnV,GGAP,SUBPOP) # 选择
        SelCh=ga.recombin(recombinStyle, SelCh, recopt, SUBPOP) #交叉
        SelCh=ga.mutint(SelCh, FieldDR, pm) # 实值变异
        ObjVSel = aimfuc(SelCh) # 求育种个体的目标函数值
        Chrom,ObjV=ga.reins(Chrom,SelCh,SUBPOP,2,1,maxormin*ObjV,\
        maxormin*ObjVSel)#重插入
    end_time = time.time() # 结束计时
    # 返回进化记录器、变量记录器以及执行时间
    return [pop_trace, var_trace, end_time - start_time]
```

详细解析：

上面我们自定义了一个遗传算法模板 mintemp1, 用于进行带约束的整数变量的目标函数最小化搜索。mintemp1 函数传入了好多参数，下面来一一解析这些参数的含义：

1) AIM_M 和 AIM_F：前者是自定义目标函数接口所在的模块名，后者是该接口的函数名。

2) PUN_M 和 PUN_F：前者是自定义罚函数接口所在的模块名，后者是该接口的函数名。

3) ranges 和 borders：控制变量的范围及是否包含边界。这里设计相关的数据结构，规定 ranges 和 borders 均是 2 行 Nvar 列的矩阵 (Nvar 表示变量的个数)，第一行分别表示第一个变量的范围下界及是否包含下界；第二行分别表示第二个变量的范围上界及是否包含上界。

4) MAXGEN：遗传算法最大进化代数。

5) NIND：种群规模，即种群的个体数。

6) SUBPOP：种群中包含的子种群数量。要求 SUBPOP 必须能被 NIND 整除。

7) GGAP：进化代沟，即子代种群与父代种群个体不相同的概率。

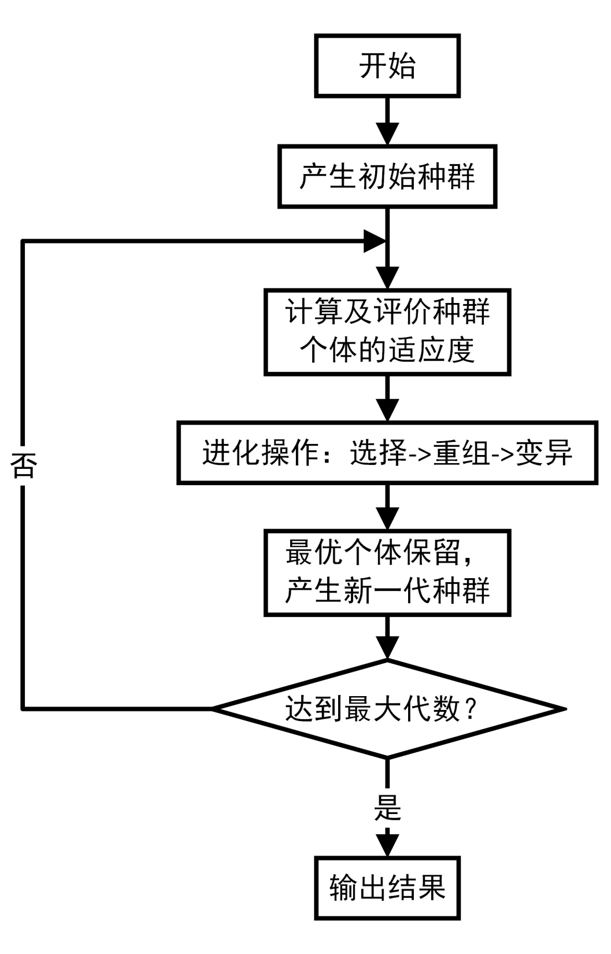
8) selectStyle：低级选择算子的字符串，如’sus’, ’rws’, ’tour’ 等。

9) recombinStyle：低级重组函数的字符串，如’xovsp’, ’xovdp’ 等。

10) recopt 和 Pm：分别代表重组概率和变异概率。

11) maxormin: 最小最大化标记，1 表示是最小化目标,-1 表示是最大化目标。

该算法模板直观地体现了用遗传算法进行目标函数最小化搜索的流程：



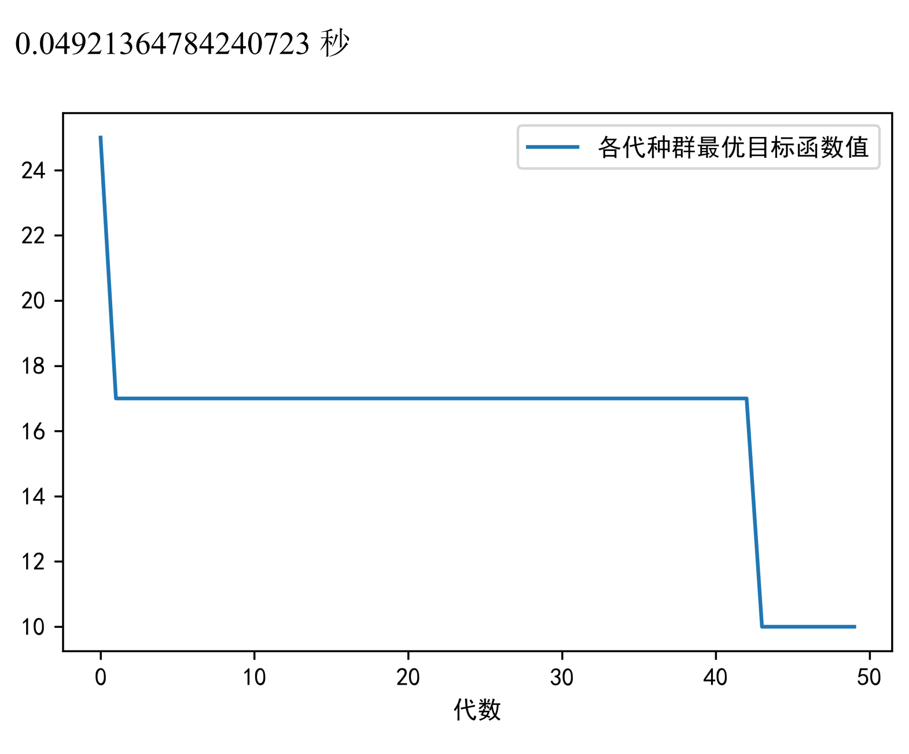
下面创建测试脚本进行问题的求解：

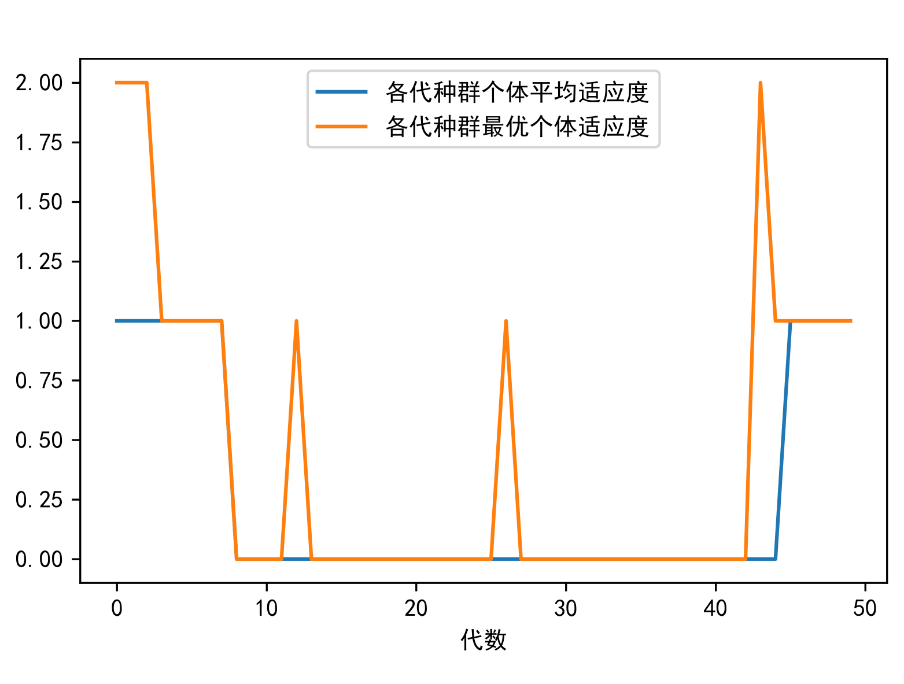
```python
"""最小化目标函数值问题求解程序执行脚本main.py"""
import numpy as np
import geatpy as gea # 导入geatpy库
from mintemp1 import mintemp1 # 导入自定义的算法模板
from aimfuc import aimfuc # 导入自定义的目标函数接口
from punishing import punishing # 导入自定义的罚函数接口

# 获取函数接口地址
AIM_M = __import__('aimfuc')
PUN_M = __import__('punishing')
"""============================变量设置============================"""
x1 = [-5, 5]; x2 = [2, 10]      # 自变量的范围
b1 = [1, 1] ; b2 = [0, 1]       # 自变量的边界
ranges=np.vstack([x1, x2]).T    # 生成自变量的范围矩阵
borders=np.vstack([b1, b2]).T   # 生成自变量的边界矩阵
"""========================遗传算法参数设置========================="""
NIND = 10;           # 种群规模
MAXGEN = 50;         # 最大遗传代数
GGAP = 0.8;          # 代沟：子代与父代的重复率为(1-GGAP)
selectStyle = 'rws';  # 遗传算法的选择方式设为"rws"——轮盘赌选择
recombinStyle = 'xovdp' # 遗传算法的重组方式，设为两点交叉
recopt = 0.9;        # 交叉概率
pm = 0.01;           # 变异概率
SUBPOP = 1           # 设置种群数为1
maxormin = 1         # 设置标记表明这是最小化目标
"""=======================调用算法模板进行种群进化==================="""
# 调用算法模板进行种群进化，得到种群进化和变量的追踪器以及运行时间
[pop_trace, var_trace, times] = mintemp1(AIM_M, 'aimfuc', PUN_M,
    'punishing', ranges, borders, MAXGEN, NIND, SUBPOP, GGAP,
    selectStyle, recombinStyle, recopt, pm, maxormin)
"""=========================绘图及输出结果========================="""
# 传入pop_trace进行绘图
gea.trcplot(pop_trace, [['各代种群最优目标函数值'],
    ['各代种群个体平均适应度值', '各代种群最优个体适应度值']],
    [['demo_result1'], ['demo_result2']])
# 输出结果
best_gen = np.argmin(pop_trace[:, 0]) # 记录最优种群是在哪一代
print('最优的目标函数值为：', np.min(pop_trace[:, 0]))
print('最优的控制变量值为：')
for i in range(var_trace.shape[1]):
    print(var_trace[best_gen, i])
print('最优的一代是第',best_gen + 1,'代')
print('用时：', times, '秒')
```

注意：本例的函数接口为上文所定义的目标函数 aimfuc 和罚函数 punishing，在运行测试脚本前，要保证自定义的算法模板、和函数接口都在同一个目录下。

运行结果如下：

```
最优的目标函数值为：10
最优的控制变量值为：
-1
3
最优的一代是第 44 代
用时：0.04921364784240723 秒
```





3. 总结

因此，在 Geatpy 中，你可以像“快速入门”章节中展示的用纯粹脚本的方式来编写遗传算法程序，也可以用进化算法模板 + 函数接口的方式。

推荐使用可自由自定义的进化算法模板来进行遗传算法编程。在融合其他项目代码时，把进化算法模板、函数接口放到项目目录下，就可以把 Geatpy 与你的其他项目相结合。

总结一下，使用进化算法模板解决遗传算法问题，你需要做 3 件事情：

1. 编写程序执行脚本 main.py
2. 编写函数接口文件，实际上是定义目标函数和罚函数 (如果有的话)，假设命名为 aimfuc.py 和 punishing.py。你也可以把函数接口直接写在程序执行脚本 main.py 中。
3. 准备好进化算法模板 (也可以用 Geatpy 自带的进化算法模板)。自定义的进化算法模板必须和函数接口文件以及程序执行脚本 main.py 放在同一个目录下 (以便 main.py 能够找到)。

在下一章中，我们将介绍几种 Geatpy 自带的几个实用进化算法模板。