

## 多染色体混合编码

对于大多数复杂的实际问题，单靠一种编码是很难甚至是完全无法进行求解的。这个时候需要混合编码。Geatpy 的染色体本身有三种最基础的编码方式：'BG'(二进制/格雷编码)、'RI'(实数整数混合编码) 以及 'P'(排列编码)，这意味着一条染色体只能是这三种编码方式的其中一种。因此当需要更加复杂的编码时，需要用多条染色体来进行协同表达。

Geatpy 的四个大类中的 Population 种群类只支持单染色体，其 Chrom 属性(种群染色体矩阵)中每一行对应的是种群的一条染色体，因此只支持'BG'、'RI' 或 'P' 中的一种编码方式。

这里引入 PsyPopulation 种群类，它是继承了 Population 类的一个新的种群类，用于支持多染色体的进化优化。它用 Linds 列表代替 Population 中的 Lind 来存储各染色体的长度；用 Encodings 列表代替 Population 中的 Encoding 来存储各染色体的编码方式；用 Fields 列表代替 Population 中的 Field 来存储各染色体的译码矩阵；用 Chroms 列表代替 Population 中的 Chrom 来存储各个染色体矩阵。PsyPopulation 和 Population 的 UML 类图对比如下所示：

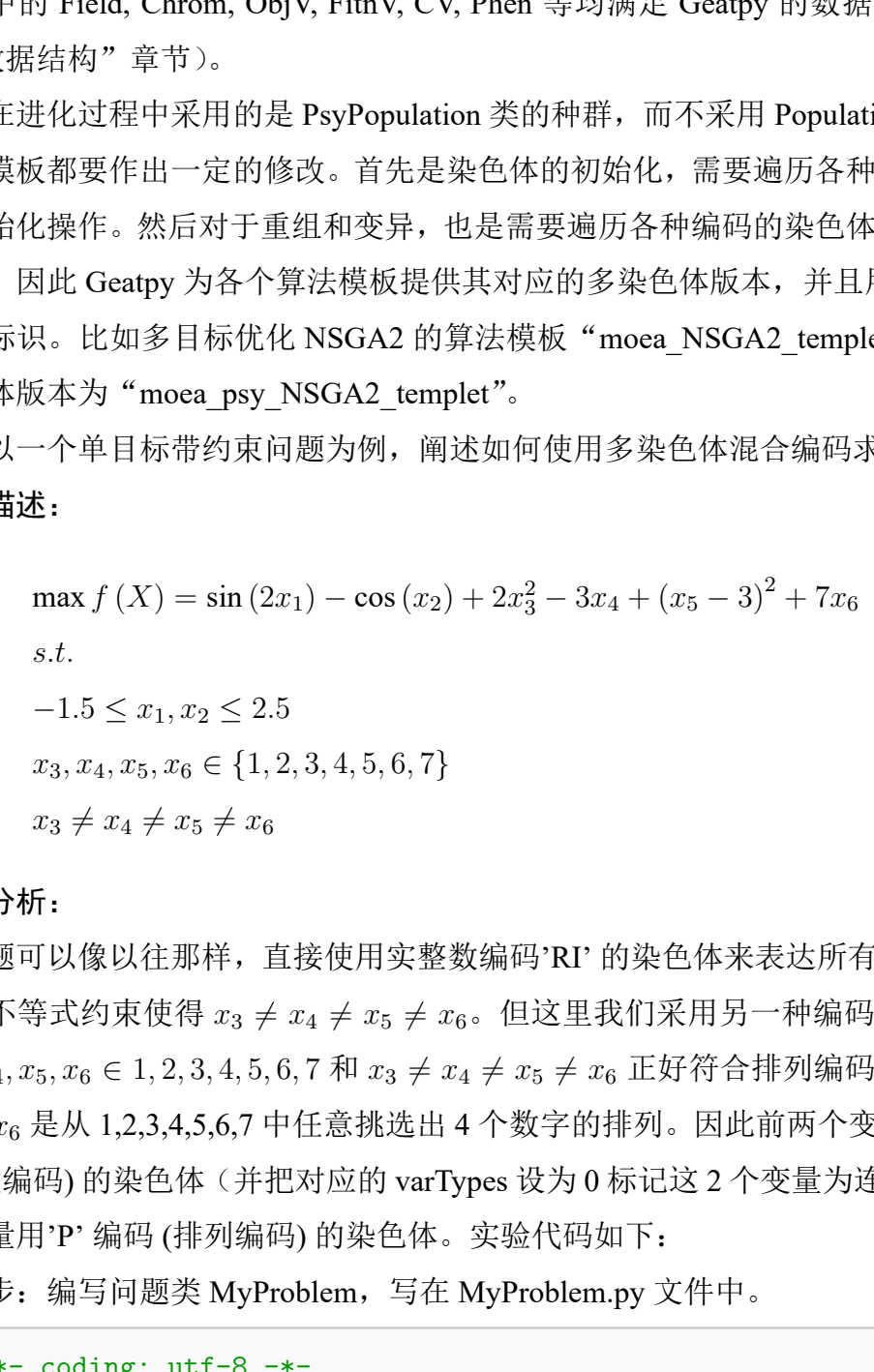


图 1 PsyPopulation 与 Population 对比图

上图中的 Field、Chrom、ObjV、FitnV、CV、Phen 等均满足 Geatpy 的数据结构（详见“Geatpy 数据结构”章节）。

如果在进化过程中采用的是 PsyPopulation 类的种群，而不采用 Population，那么原有的算法模板都要作出一定的修改。首先是染色体的初始化，需要遍历各种编码的染色体进行初始化操作。然后对于重组和变异，也是需要遍历各种编码的染色体分别进行重组和变异；因此 Geatpy 为各个算法模板提供其对应的多染色体版本，并且用“psy”字符串加以标识。比如多目标优化 NSGA2 的算法模板“moea\_NSAGA2\_templet”，其对应的多染色体版本为“moea\_psy\_NSAGA2\_templet”。

下面以一个单目标约束问题为例，阐述如何使用多染色体混合编码求解问题：

问题描述：

$$\begin{aligned} \max f(X) &= \sin(2x_1) - \cos(x_2) + 2x_3^2 - 3x_4 + (x_5 - 3)^2 + 7x_6 \\ s.t. \\ -1.5 \leq x_1, x_2 &\leq 2.5 \\ x_3, x_4, x_5, x_6 &\in \{1, 2, 3, 4, 5, 6, 7\} \\ x_3 \neq x_4 \neq x_5 \neq x_6 \end{aligned}$$

问题分析：

该问题可以像以往那样，直接使用实数编码'RI' 的染色体来表达所有的变量，然后加一个不等式约束使得  $x_3 \neq x_4 \neq x_5 \neq x_6$ ，但这里我们采用另一种编码方法：不难发现  $x_3, x_4, x_5, x_6 \in 1, 2, 3, 4, 5, 6, 7$  和  $x_3 \neq x_4 \neq x_5 \neq x_6$  正好符合排列编码的特征，即  $x_3, x_4, x_5, x_6$  是从 1,2,3,4,5,6,7 中任意挑选出 4 个数字的排列。因此前两个变量用'RI' 编码(实数编码)的染色体（并把对应的 varTypes 设为 0 标记这 2 个变量为连续型变量）；后 4 个变量用'P' 编码(排列编码)的染色体。实验代码如下：

第一步：编写问题类 MyProblem，写在 MyProblem.py 文件中。

```
# -*- coding: utf-8 -*-
"""MyProblem.py"""
import numpy as np
import geatpy as ea
class MyProblem(ea.Problem): # 继承Problem父类
    def __init__(self):
        name = 'MyProblem' # 初始化name（函数名称，可以随意设置）
        M = 1 # 初始化M（目标维数）
        # 初始化maxormins（目标最小最大化标记列表，1：最小化；-1：最大化）
        maxormins = [-1]
        Dim = 6 # 初始化Dim（决策变量维数）
        # 初始化决策变量的类型，元素为0表示变量是连续的；1为离散的
        varTypes = [0,0,1,1,1,1]
        lb = [-1.5,-1.5,1,1,1,1] # 决策变量下界
        ub = [2.5,2.5,7,7,7,7] # 决策变量上界
        lbin = [1] * Dim # 决策变量下边界
        ubin = [1] * Dim # 决策变量上边界
        # 调用父类构造方法完成实例化
        ea.Problem.__init__(self, name, M, maxormins, Dim, varTypes, lb,
                             ub, lbin, ubin)
    def aimFunc(self, pop): # 目标函数
        X = pop.Phen # 得到决策变量矩阵
        x1 = X[:, [0]]
        x2 = X[:, [1]]
        x3 = X[:, [2]]
        x4 = X[:, [3]]
        x5 = X[:, [4]]
        x6 = X[:, [5]]
        pop.ObjV = np.sin(2*x1) - np.cos(x2) + 2*x3**2 - 3*x4 + (x5-3)**2 + 7*x6 # 计算目标函数值，赋值给pop种群对象的ObjV属性
```

第二步：编写执行脚本，写在 main.py 文件中。

```
# -*- coding: utf-8 -*-
"""main.py"""
import numpy as np
import geatpy as ea # import geatpy
from MyProblem import MyProblem # 导入自定义问题接口
"""=====实例化问题对象====="""
problem = MyProblem() # 生成问题对象
"""=====种群设置====="""
NIND = 40 # 种群规模
# 创建区域描述器，这里需要创建两个，前2个变量用RI编码，其余用排列编码
Encodings = ['RI', 'P']
Field1 = ea.crtfld(Encodings[0], problem.varTypes[:2],
                  problem.ranges[:, :2], problem.borders[:, :2])
Field2 = ea.crtfld(Encodings[1], problem.varTypes[2:],
                  problem.ranges[:, 2:], problem.borders[:, 2:])
Fields = [Field1, Field2]
population = ea.PsyPopulation(Encodings, Fields, NIND) #
    实例化种群对象（此时种群还没被初始化，仅仅是完成种群对象的实例化）
"""=====算法参数设置====="""
myAlgorithm = ea.soea_psy_EGA_templet(problem, population) #
    实例化一个算法模板对象
myAlgorithm.MAXGEN = 25 # 最大进化代数
myAlgorithm.logTras = 1 #
    设置每隔多少代记录日志，若设置成0则表示不记录日志
myAlgorithm.verbose = True # 设置是否打印输出日志信息
myAlgorithm.drawing = 1 #
    设置绘图方式（0：不绘图；1：绘制结果图；2：绘制目标空间过程动画；
    3：绘制决策空间过程动画）
"""=====调用算法模板进行种群进化====="""
[BestIndi, population] = myAlgorithm.run() #
    执行算法模板，得到最优个体以及最后一代种群
BestIndi.save() # 把最优个体的信息保存到文件中
"""=====输出结果====="""
print('评价次数: %s' % myAlgorithm.evalsNum)
print('时间已过 %s 秒' % myAlgorithm.passTime)
if BestIndi.sizes != 0:
    print('最优的目标函数值为: %s' % (BestIndi.ObjV[0][0]))
    print('最优的控制变量值为: ')
    for i in range(BestIndi.Phen.shape[1]):
        print(BestIndi.Phen[0, i])
    else:
        print('没找到可行解。')

```

代码解析

Geatpy 的问题类是不用管种群用何种编码方式的，只需把问题描述清楚。上面的代码中，种群染色体采用何种编码方式是在执行脚本 main.py 中进行设置的；由于上述问题中前两个变量是实数，后四个变量是互不相等的整数，于是设置两个 Encoding，把它存储在 Encodings 列表中，因此有“Encodings = ['RI', 'P']”，由于有两种编码，因此需要创建两个译码矩阵 Field1 和 Field2，最后把它们存储到 Fields 列表中。在随后的实例化种群对象时，要注意用的种群类是 PsyPopulation 而不是 Population。因为 PsyPopulation 是 Population 衍生出来的支持多染色体混合编码的种群类。最后在实例化算法模板对象时，要注意挑选的算法模板的名称要带有“psy”字符串，表示这是一个支持多染色体混合编码的算法模板。由于本例是一个单目标优化问题，因此可以采用“soea\_SEGA\_templet”的多染色体版本：“soea\_psy\_SEGA\_templet”算法模板进行求解。

运行 main.py，得到如下结果：



种群信息导出完毕。

评价次数：976

时间已过 0.007914543151855469 秒

最优的目标函数值为：142.56831099333382

最优的控制变量值为：

1.1162071228027344

2.4643630981445312

7.0

1.0

5.0

6.0

下面详细探讨多染色体版本的算法模板跟原始算法模板有什么区别。

查看“soea\_SEGA\_templet.py”以及“soea\_psy\_SEGA\_templet.py”，代码如下：

```
# -*- coding: utf-8 -*-
"""soea_SEGA_templet.py"""
import geatpy as ea # 导入geatpy库
from sys import path as paths
from os import path
paths.append(path.split(path.split(path.realpath(__file__))[0])[0])
class soea_SEGA_templet(ea.SoeaAlgorithm):
    """
    soea_SEGA_templet : class - Strengthen Elitist GA
    templet(增强精英保留的遗传算法模板)

    算法描述：
    本模板实现的是增强精英保留的遗传算法。算法流程如下：
    1) 根据编码规则初始化N个个体的种群。
    2) 若满足停止条件则停止，否则继续执行。
    3) 对当前种群进行统计分析，比如记录其最优个体、平均适应度等等。
    4) 独立地从当前种群中选取N个母体。
    5) 独立地对这N个母体进行交叉操作。
    6) 独立地对这N个交叉后的个体进行变异。
    7) 将父代种群和交叉变异得到的种群进行合并，得到规模为2N的种群。
    8) 从合并的种群中根据选择算法选择出N个个体，得到新一代种群。
    9) 回到第2步。
    该算法宜设置较大的交叉和变异概率，
    否则生成的新一代种群中会有越来越多的重复个体。
    """
    def __init__(self, problem, population):
        ea.SoeaAlgorithm.__init__(self, problem, population) #
            先调用父类构造方法
        if population.ChromNum != 1:
            raise
                RuntimeError('传入的种群对象必须是单染色体的种群类型。')
        self.name = 'SEGA'
        self.selFunc = 'tour' # 锦标赛选择算子
        if population.Encoding == 'P':
            self.recOper = ea.Xovpmx(XOVR = 1) #
                生成部分匹配交叉算子对象
            self.mutOper = ea.Mutinv(Pm = 1) # 生成逆转变异算子对象
        else:
            self.recOper = ea.Xovdp(XOVR = 1) # 生成两点交叉算子对象
            if population.Encoding == 'BG':
                #生成二进制变异算子对象,Pm设置为None时,具体取变异算子中的默认值
                self.mutOper=ea.Mutbin(Pm=None)
            elif population.Encoding == 'RI':
                self.mutOper = ea.Mutbga(Pm = 1/self.problem.Dim,
                                         MutShrink = 0.5, Gradient = 20) # 生成breeder
                                         GA变异算子对象
            else:
                raise RuntimeError('编码方式必须为'BG'、'RI'或'P'.')
    def run(self):
        #=====初始化配置=====
        population = self.population
        NIND = population.sizes
        self.initialization() # 初始化算法模板的一些动态参数
        #=====准备进化=====
        population.initChrom(NIND) # 初始化种群染色体矩阵
        self.call_aimFunc(population) # 计算种群的目标函数值
        if prophetPop is not None:
            population = (prophetPop + population)[NIND] # 插入先知种群
        population.FitnV = ea.scaling(population.ObjV, population.CV,
                                       self.problem.maxormins) # 计算适应度
        #=====开始进化=====
        while self.terminated(population) == False:
            # 选择
            offspring = population[ea.selecting(self.selFunc,
                                                population.FitnV, NIND)]
            # 进行进化操作
            offspring.Chrom = self.recOper.do(offspring.Chrom) # 重组
            offspring.Chrom = self.mutOper.do(offspring.Encoding,
                                              offspring.Chrom, offspring.Field) # 变异
            # 求进化后个体的目标函数值
            offspring.Phen = offspring.decoding() # 染色体解码
            self.problem.aimFunc(offspring) # 计算目标函数值
            self.evalsNum += offspring.sizes # 更新评价次数
            population = population + offspring # 父子合并
            population.FitnV = ea.scaling(population.ObjV,
                                           population.CV, self.problem.maxormins) # 计算适应度
            # 得到新一代种群
            population = population[ea.selecting(self.selFunc,
                                                population.FitnV, NIND)]
            return self.finishing(population) #
                调用finishing完成后续工作并返回结果
        =====

```

```
# -*- coding: utf-8 -*-
"""soea_psy_SEGA_templet.py"""
import geatpy as ea # 导入geatpy库
from sys import path as paths
from os import path
paths.append(path.split(path.split(path.realpath(__file__))[0])[0])
class soea_psy_SEGA_templet(ea.SoeaAlgorithm):
    """
    soea_psy_SEGA_templet : class - Polysomy Strengthen Elitist GA
    templet(增强精英保留的多染色体遗传算法模板)

    模板说明：
    该模板是内置算法模板soea_SEGA_templet的多染色体版本，
    因此里面的种群对象为支持混合编码的多染色体种群类PsyPopulation类的对象。

    算法描述：
    本模板实现的是增强精英保留的遗传算法。算法流程如下：
    1) 根据编码规则初始化N个个体的种群。
    2) 若满足停止条件则停止，否则继续执行。
    3) 对当前种群进行统计分析，比如记录其最优个体、平均适应度等等。
    4) 独立地从当前种群中选取N个母体。
    5) 独立地对这N个母体进行交叉操作。
    6) 独立地对这N个交叉后的个体进行变异。
    7) 将父代种群和交叉变异得到的种群进行合并，得到规模为2N的种群。
    8) 从合并的种群中根据选择算法选择出N个个体，得到新一代种群。
    9) 回到第2步。
    该算法宜设置较大的交叉和变异概率，
    否则生成的新一代种群中会有越来越多的重复个体。
    """
    def __init__(self, problem, population):
        ea.SoeaAlgorithm.__init__(self, problem, population) #
            先调用父类构造方法
        if population.ChromNum == 1:
            raise
                RuntimeError('传入的种群对象必须是多染色体的种群类型。')
        self.name = 'psy-SEGA'
        self.selFunc = 'etour' # 锦标赛选择算子
        # 由于有多条染色体，因此需要用多个重组和变异算子
        self.recOPers = []
        self.mutOPers = []
        for i in range(population.ChromNum):
            if population.Encodings[i] == 'P':
                recOper = ea.Xovpmx(XOVR = 1) # 生成部分匹配交叉算子对象
                mutOper = ea.Mutinv(Pm = 1) # 生成逆转变异算子对象
            else:
                recOper = ea.Xovdp(XOVR = 1) # 生成两点交叉算子对象
                if population.Encodings[i] == 'BG':
                    #生成二进制变异算子对象,Pm设置为None时,具体取变异算子中的默认值
                    mutOper=ea.Mutbin(Pm=None)
                elif population.Encodings[i] == 'RI':
                    mutOper = ea.Mutbga(Pm = 1/self.problem.Dim, MutShrink
                                         = 0.5, Gradient = 20) # 生成breeder GA变异算子对象
                else:
                    raise
                        RuntimeError('编码方式必须为'BG'、'RI'或'P'.')
            self.recOPers.append(recOper)
            self.mutOPers.append(mutOper)
    def run(self):
        #=====初始化配置=====
        population = self.population
        NIND = population.sizes
        self.initialization() # 初始化算法模板的一些动态参数
        #=====准备进化=====
        population.initChrom(NIND) #
            初始化种群染色体矩阵(内含染色体解码，详见Population类的源码)
        self.call_aimFunc(population) # 计算种群的目标函数值
        if prophetPop is not None:
            population = (prophetPop + population)[NIND] # 插入先知种群
        population.FitnV = ea.scaling(population.ObjV, population.CV,
                                       self.problem.maxormins) # 计算适应度
        #=====开始进化=====
        while self.terminated(population) == False:
            # 选择
            offspring = population[ea.selecting(self.selFunc,
                                                population.FitnV, NIND)]
            # 进行进化操作，分别对各种编码的染色体进行重组和变异
            for i in range(population.ChromNum):
                offspring.Chroms[i] =
                    self.recOPers[i].do(offspring.Chroms[i]) # 重组
                offspring.Chroms[i] =
                    self.mutOPers[i].do(offspring.Encodings[i],
                                       offspring.Chroms[i], offspring.Fields[i]) # 变异
            # 求进化后个体的目标函数值
            offspring.Phen = offspring.decoding() # 染色体解码
            self.problem.aimFunc(offspring) # 计算目标函数值
            self.evalsNum += offspring.sizes # 更新评价次数
            population = population + offspring # 父子合并
            population.FitnV = ea.scaling(population.ObjV,
                                           population.CV, self.problem.maxormins) # 计算适应度
            # 得到新一代种群
            population = population[ea.selecting(self.selFunc,
                                                population.FitnV, NIND)]
            return self.finishing(population) #
                调用finishing完成后续工作并返回结果
        =====

```

代码分析：

观察上面两个算法模板的代码，可见基本流程是完全一样的，主要的两个不同之处在于多染色体版本的算法模板在构造函数上把需要用到重组和变异算子存储在一个列表中，目的是在进化过程中可以让各个染色体矩阵独立地用列表中的重组和变异算子进行重组和变异。因此，对于进化过程中的重组变异那一块的代码，多染色体版本采用一个循环来对 Chroms 列表中的每一个 Chrom(种群染色体矩阵)进行重组和变异，这是因为多染色体版本的算法模板中种群类是 PsyPopulation 而不是 Population，它的染色体矩阵是多个而不是单个（存储在 Chroms 列表中）。

以下是 Geatpy 目前内置的多染色体版本的进化算法模板，其具体代码均可在源码中查看到：

- soea\_psy\_EGA\_templet(精英保留的多染色体遗传算法模板)
- soea\_psy\_SEGA\_templet(增强精英保留的多染色体遗传算法模板)
- soea\_psy\_SGA\_templet(最简单、最经典的多染色体遗传算法模板)
- soea\_psy\_GGAP\_SGA\_templet(带代沟的多染色体简单遗传算法模板)
- soea\_psy\_studGA\_templet(多染色体种马遗传算法模板)
- soea\_psy\_steady\_GA\_templet(多染色体稳态遗传算法模板)
- moea\_psy\_awGA\_templet(基于 awGA 算法的多染色体多目标进化算法模板)
- moea\_psy\_NSAGA2\_archive\_templet(带全局存档的多染色体多目标进化 NSGA-II 算法模板)
- moea\_psy\_NSAGA2\_templet(基于 NSGA-II 算法的多染色体多目标进化算法模板)
- moea\_psy\_NSAGA3\_templet(基于 NSGA-III 算法的多染色体多目标进化算法模板)
- moea\_psy\_RVEA\_templet(基于 RVEA 算法的多染色体多目标进化算法模板)
- moea\_psy\_RVEA\_RES\_templet(基于带参考点再生策略的多染色体 RVEA 算法的多目标进化算法模板)