

## 化繁为简

本文档将会把整个 Geatpy 工具箱的使用方法再次捋一下，把关键点、核心点梳理以便，以达到“理解本质、化繁为简”的目的。

### 一、Geatpy 工具箱的组成及基本用法

Geatpy 工具箱由：[内核函数](#) 以及[进化算法框架](#)（简称 EA 框架）两部分组成。

内核函数是封装起来的高性能计算单元，内部有数万行代码，它与 EA 框架是高度脱耦合的，可以在 python 上直接调用这些内核函数。内核函数一览可见网站：[https://github.com/geatpy-dev/geatpy/tree/master/\\_core/Windows/lib64/v3.5](https://github.com/geatpy-dev/geatpy/tree/master/_core/Windows/lib64/v3.5)

以变异函数“mutpolyn”为例，其具体用法可通过 import geatpy as ea; help(ea.mutpolyn) 查看。

EA 框架由：种群类、问题类、算法模板类、进化算子类四个大类组成。这里如果对类的概念不太熟悉，可先通过互联网学习一下 Python 的面向对象编程的相关知识。以 soca\_demo1 为例，在使用 Geatpy 解决一个具体的优化问题时，首先要做的是编写一个自定义问题类，然后编写一个执行脚本（在案例中通常被命名为 main.py）来启动 Geatpy。其中最核心的是编写自定义问题类，需要根据具体的优化模型，确定好决策变量、待优化的目标函数以及约束条件来编写具体的代码。例如：

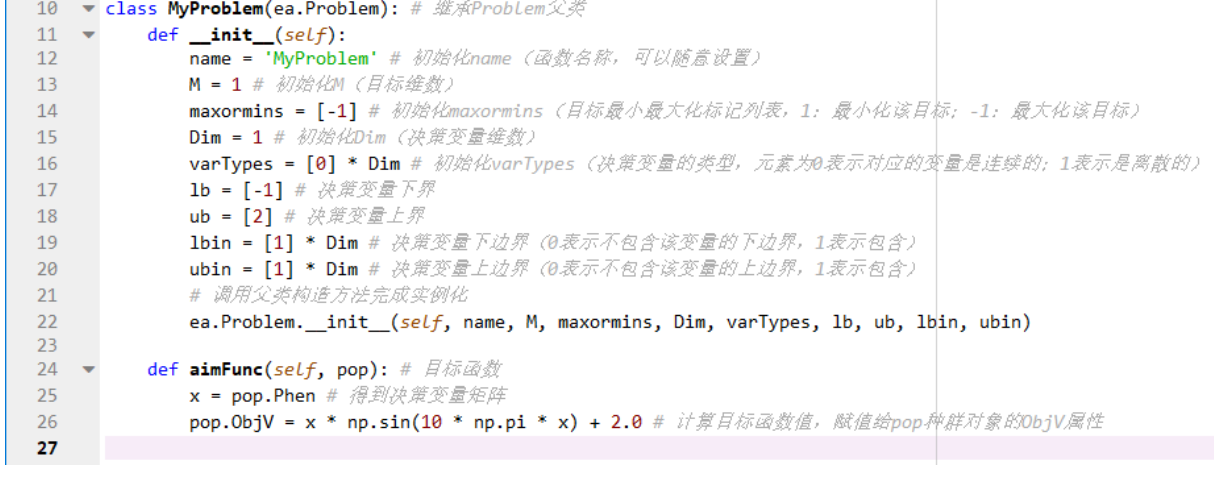


图 1

如果 Python 基础比较薄弱，可能会对上述代码中的 varTypes = [0] \* Dim 表示困惑，这里实际上是创建了一个长度为 Dim 的、元素全为 0 的 Python list 类型列表。假如 Dim 为 3，则 [0] \* Dim 相当于 [0, 0, 0]。

在编写 aimFunc() 函数的过程中最容易出错的是算得的目标函数值矩阵 ObjV 和违反约束程度矩阵 CV 不符合 Geatpy 的数据结构。在 Geatpy 中，约定 ObjV 和 CV 都是 Numpy ndarray 类型的 2D 数组，每一行对应种群的一个个体。假如是单目标优化，那么得到的 ObjV 就应该是一个列向量；假如只有一个约束条件，那么得到的 CV 就应该是一个列向量。

由于 Geatpy 的数据结构规定种群的表现型矩阵 Phen（一般等价于代表模型中的决策变量）是 Numpy ndarray 类型的矩阵，这意味着 aimFunc() 中假如无法通过矩阵运算的方法“同时”计算种群所有个体的目标函数值，那么需要用循环或者并行并发、分布式等方法分别计算种群每个个体的目标函数值，最后拼成一个种群染色体矩阵 ObjV。例如：

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

那么利用 Phen[0, :] 便可以得到种群的第一个个体的“决策变量”。以此类推。

注意：Geatpy 中“决策变量”的含义仅仅指的是种群染色体矩阵 Chrom 根据其编码设定（“RI”，“P”或“BG”）进行一次解码后得到的内容，比如假设编码为“BG”的染色体：1 1 1 0 1 1 解码后为：7 3，则这个“7 3”便被称作决策变量。假如用户需要更进一步解码，比如将 7 3 通过映射得到 6.1 4.2，那么这个 6.1 4.2 在用户所求解的模型中是决策变量，但在 Geatpy 中，只把染色体 1 1 1 0 1 1 根据区域描述器解码得到的 7 3 称作是“决策变量”。

编写完自定义问题类后，就到编写执行脚本（main.py）这一步了。执行脚本要做的工作是：

- 实例化问题类的对象；
- 实例化一个不带染色体信息的种群对象；
- 实例化算法模板对象；
- 调用算法模板对象的 run() 函数进行进化优化；
- 返回结果。

例如：



图 2

这里需要强调的是种群染色体的编码方式。Geatpy 内置 3 种基本的种群染色体编码：实整数编码“RI”、排列编码“P”以及二进制/格雷码编码“BG”。前两个编码的种群的染色体矩阵 Chrom 和种群表现型矩阵 Phen 的内容是一模一样的（详见《Geatpy 数据结构》章节）；而“BG”编码的种群染色体矩阵 Chrom 是一个元素全为 0、1 的矩阵，它要通过解码才能得到种群表现型矩阵 Phen。

### 二、Geatpy 数据流动情况简述

本节将描述 Geatpy 中的数据是怎么流动的，您可以通过单步调试的方法对照着本文进行学习，以图 2 为例，完整代码在 soca\_demo1 中。

1、从执行脚本 main.py 开始看，实例化问题类对象的过程就不用多说了，首先发生重要的数据流动的是创建区域描述器 Field。它是调用 ea.crtfld() 这个内核函数完成的，具体 API 可通过 import geatpy as ea; help(ea.crtfld) 查看。区域描述器的概念可见《Geatpy 数据结构》章节，它并非很特殊的数据结构，而仅仅是一个 Numpy ndarray 类型的矩阵，用于描述与种群染色体有关的“范围信息”。比如实整数编码的染色体对应的区域描述器是一个 3 行 N 列的矩阵，第一行代表染色体每一位数字的下界，第二行代表染色体每一位数字的上界，第三行代表染色体每一位数字的“类型”（0 表示是实数；1 表示是整数）。

2、然后到实例化种群对象，这里将会跳转到 Population.py 文件中，执行种群类的构造函数完成种群对象的实例化。通过查看 Population.py 可以发现构造方法里面并没有初始化种群对象的染色体属性，这也就对应执行脚本中的注释“此时种群还没被初始化，仅仅是完成种群对象的实例化”。一般来说种群对象的染色体矩阵 Chrom 被赋值之后，这个种群才算是被“初始化”。

3、接着是实例化算法模板对象。这里由于要实例化的类为“soea\_SEGA\_templet”，因此将会跳转到 soca\_SEGA\_templet.py 文件的构造函数中。如下图所示：

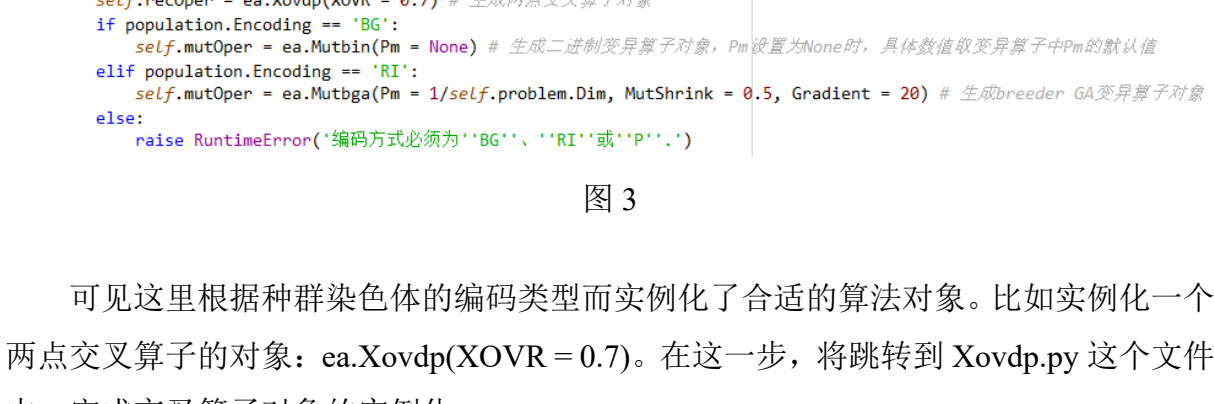


图 3

可见这里根据种群染色体的编码类型而实例化了合适的算法对象。比如实例化一个两点交叉算子的对象：ea.Xovdp(XOVR = 0.7)。在这一步，将跳转到 Xovdp.py 这个文件中，完成交叉算子对象的实例化。

这里需要提及 Xovdp 这个类和 xovdp() 这个内核函数的区别：前者是在 EA 框架中设置的一个类，目的是为了更方便地调用内核函数 xovdp()。而真正进行两点交叉的是在内核函数 xovdp() 中。有了这个 Xovdp 类的对象，就可以实现“先实例化、设置好相关的参数，然后再调用 xovdp()”这样的功能。

4、回到 main.py，紧接着便是调用算法模板对象的 run() 函数，这里就开始了进化优化了。具体代码将跳转到 soca\_SEGA\_templet.py 文件的 run() 函数中。可以发现里面的一些代码在 soca\_SEGA\_templet.py 文件中并没有定义，例如判断进化是否应该停止的 self.terminated() 函数。如果了解 Python 面向对象编程，便显然可知这个 self.terminated() 实际上是 soca\_SEGA\_templet 算法模板类的父类：SoeaAlgorithm 所实现的，详见 Algorithm.py 文件。

5、在 soca\_SEGA\_templet.py 中可以看到这个进化算法的完整流程。这里有些用户可能会对一些特殊的写法有些疑惑，比如：“population = population + offspring”。这个实际上是 Population 种群类的一个运算符重载，详见 Population.py，它把加号“+”重载为让两个种群合并为一个大的种群，中间过程是两个种群类的各个属性（如染色体矩阵 Chrom、目标函数值矩阵 ObjV 等等）的合并。

在进化优化中，一定要注意“种群的适应度”和“种群的目标函数值”是不一样的，前者是一个列向量，后者是一个矩阵。如果进化算法的流程里需要依靠“适应度”来选择个体，那么需要给种群对象的 FitnV 属性赋值。一般来说适应度可以根据种群目标函数值矩阵 ObjV 来计算得出。

算法模板 soca\_SEGA\_templet 的内容如下图：



图 4

6、执行完算法模板对象的 run() 函数后，返回最优个体以及最后一代种群。

以上便是使用 Geatpy 进行进化优化的数据流动的情况简述。可以看到 Geatpy 的 EA 框架的基于 Geatpy 的工具箱内核函数来实现运作的，可以明显感受到模块之间的耦合程度比较小，可以很清晰地看到进化算法的整个实现过程。整个进化算法的核心在与算法模板类，如果需要实现新的算法，只需根据上文描述的数据流动情况来实现一个自定义算法模板类，即可在 Geatpy 中运行自己设计的进化算法。