

# 进化算法

## 序言

进化算法 (Evolutionary Algorithm, EA) 是一类通过模拟自然界生物自然选择和自然进化的随机搜索算法。与传统搜索算法如二分法、斐波那契法、牛顿法、抛物线法等相比，进化算法有着高鲁棒性和求解高度复杂的非线性问题 (如 NP 完全问题) 的能力。

在过去的 40 年中，进化算法得到了不同的发展，现主要有三类：

- 1) 主要由美国 J. H. Holland 提出的的遗传算法 (Genetic Algorithm, GA);
- 2) 主要由德国 I. Rechenberg 提出的进化策略 (Evolution strategies, ES);
- 3) 主要由美国的 L. J. Fogel 提出的进化规划 (Evolutionary Programming, EP)。

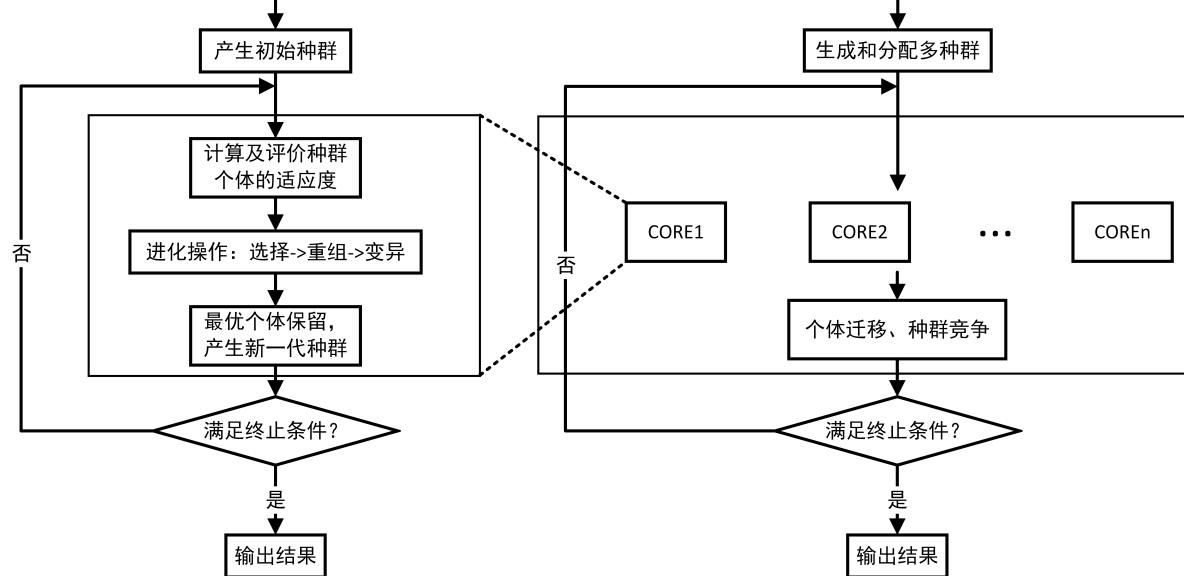
三种进化算法都是受相同的自然进化原则的启发下创立的，[Fdb94] 以及国内的诸多资料也有详细的介绍。本文档我们只介绍经典的遗传算法，不对改进的遗传算法以及用途更广的多目标优化遗传算法作详细介绍，需要进行相关研究的可以参考相关的专业性书籍。文档第一章是有关遗传算法的概述和基本框架。第二章介绍了编码。第三章是关于适应度的计算。第四章讲述了选择算法。在第五章中，介绍了不同的重组算法。第六章解释了如何变异。第七章讲解了什么是重新插入和生成后代。第八章详细讲解了与多目标优化有关的入门知识。

最后值得一提的是，虽然进化算法已经比较成熟，在近 20 年来已经得到了快速的发展，在金融、工程、信息学、数学等领域已经有广泛的应用，但是，众多新兴的进化算法 (如差分进化算法等) 以及不断改进和完善的拥有高维、多目标问题求解能力的进化算法等等，正给进化算法注入源源不断的活力。与此同时，深度神经网络的蓬勃发展让进化算法有了一个更加前沿和广阔的前景——神经进化。量子计算机的出现，也使得拥有高度并行能力的进化算法有着更大的潜能。

## 1 遗传算法概述

自然界生物在周而复始的繁衍中，基因的重组、变异等，使其不断具有新的性状，以适应复杂多变的环境，从而实现进化。遗传算法精简了这种复杂的遗传过程而抽象出一套数学模型，用较为简单的编码方式来表现复杂的现象，并通过简化的遗传过程来实现对复杂搜索空间的启发式搜索，最终能够在较大的概率下找到全局最优解，同时与生俱来地支持并行计算。

下图展示了常规遗传算法(左侧)和在并行计算下的遗传算法(右侧)的流程。



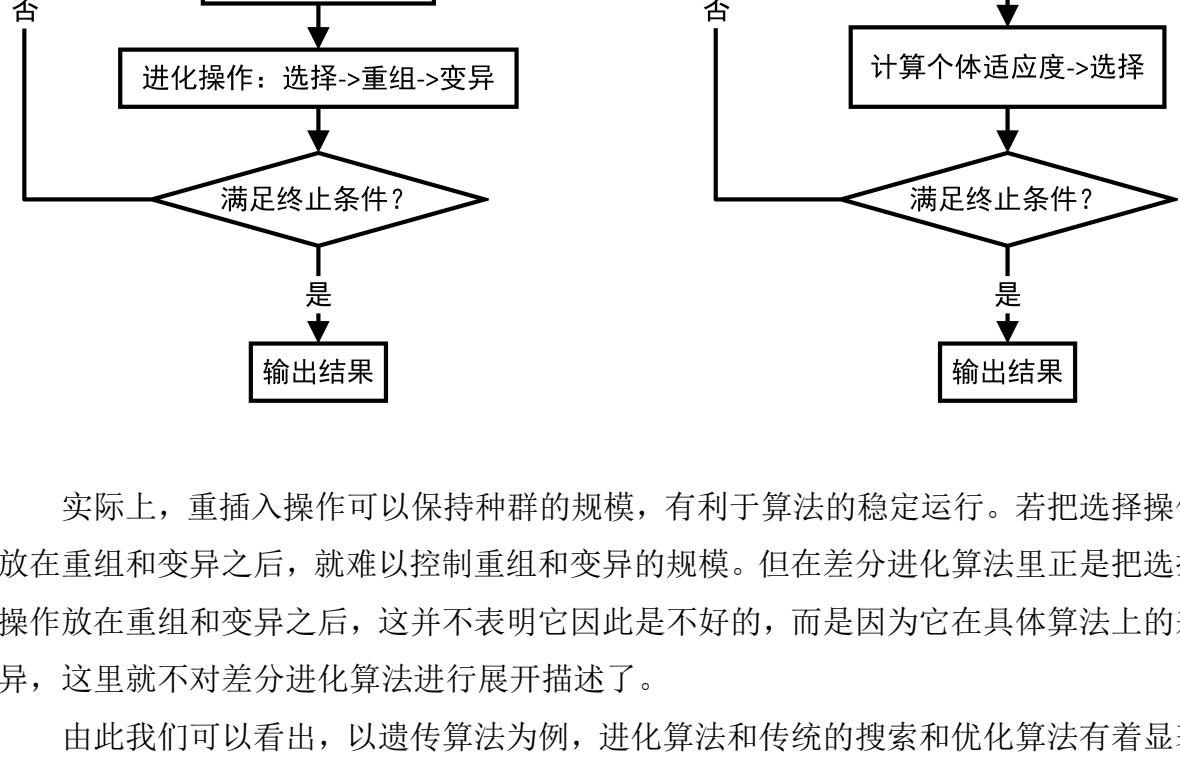
图中 $CORE_n$ 表示计算核心。不同的计算核心处理不同的单个子种群(当然也可以处理多个子种群)，种群间互相独立进行进化(区域模型)，种群间进行个体迁移和种群竞争。这只是其中一种并行计算遗传算法，另外还有全局模型和本地模型。

除了种群间可以并行计算外，中群内的若干个个体也可以利用矩阵化来实现并行计算。因此遗传算法具有很强的并行性。

值得注意的是：遗传算法中，重组和交叉并不是同一个概念，交叉是重组的一种。

对于常规遗传算法，在计算开始时，根据设计的编码规则随机初始化许多个体(形成一个或多个种群)，然后评估种群中个体的适应度，并根据适应度来选择一些个体到交配池，然后对交配池中的个体进行一定概率的重组和变异产生育种后代，最后把育种后代插入到父代种群，淘汰父代中的个体，最终得到新一代种群。

当然，你也可以调整一下上面的顺序，比如重组和变异操作后不进行重插入而直接得到新一代种群；或者是不经过选择直接对父代种群的所有个体进行重组和变异操作，然后才是进行选择操作，最后把选择操作的结果作为新一代种群。如下图所示：



实际上，重插入操作可以保持种群的规模，有利于算法的稳定运行。若把选择操作放在重组和变异之后，就难以控制重组和变异的规模。但在差分进化算法里正是把选择操作放在重组和变异之后，这并不表明它因此是不好的，而是因为它在具体算法上的差异，这里就不对差分进化算法进行展开描述了。

由此我们可以看出，以遗传算法为例，进化算法和传统的搜索和优化算法有着显著不同，最明显的差异是：

- 进化算法具有与生俱来的并行性，它可以并行地搜索一组点，而不是一个点。
- 进化算法使用的是概率转换规则，并非确定性转换规则。
- 进化算法不需要额外的信息，只有目标函数和相应的适应度影响搜索方向。
- 进化算法鲁棒性强，可以与各种算法轻松地结合在一起。
- 进化算法可以整合其他优化算法的优点，比如利用其他优化算法的优化结果来生成初始种群，这种二次搜索方式在很多场合下可以大幅度提高搜索效率。
- 进化算法可以给特定的问题提供多样化的搜索结果，让用户自己选择。比如在多目标优化的进化算法里，算法给出的是一组帕累托最优解。这些最优解可以作为多组备选方案。

选择、重组和变异是遗传算法提供的经典操作算子。很多改进的遗传算法都是围绕他们展开的。

## 2 编码

编码的设计是使用遗传算法解决实际问题中极为关键的要素。

编码就是将问题的解空间映射到编码空间(即搜索空间)上的过程。而由编码空间向问题的解空间的映射就成为解码。

编码的方式对搜索过程影响较大。按照编码结果的数据类型分类，编码可以是：一维或二维的数据列、二进制数据列或字符串等。按照映射方式来分类，编码可以是：简单映射编码、多重映射编码、排列编码、树编码等。

一般而言编码需要满足以下三个原则：

- 1) 完备性：问题的解空间中所有的点都可以映射到编码空间中的点(即染色体)。
- 2) 可靠性：编码得到的染色体必须对应问题空间中的某一个潜在解。
- 3) 非冗余性(不强制要求)：染色体和潜在解之间必须一一对应。

### 3 适应度评价

适应度是指种群个体“适应环境的能力”。适应度的计算同样是使用遗传算法中的极为关键的要素。它指定了问题解的搜索方向，并直接关系到搜索效率和最终解的质量好坏。

对于已建立的数学模型，一般可以采用目标函数值作为适应度值。为了应对最小化和最大化的两种相反的优化目标，一般遵循最小化约定，即“目标函数值越大，适应度越小”。某些版本是倒过来的，在算法内固定一种标准即可。

适应度的值必须是非负实数。某些问题中求出来的个体目标函数值有正有负，甚至是复数。因此有必要对目标函数与适应度函数之间建立合理的映射关系，保证适应度值是非负的，并且适应度增大的方向与目标函数的优化方向一致。

我们还可以对目标函数作线性变换、指数变换、幂指数变换、截断处理等得到适应度。这些变换都会对种群的多样性和算法的收敛速度带来影响。例如：

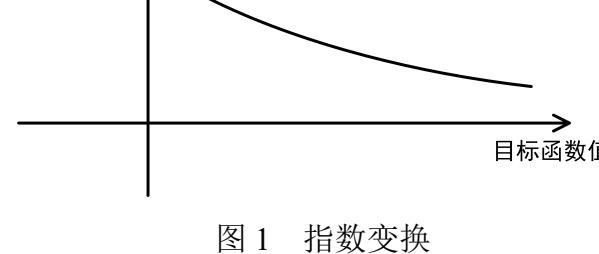


图 1 指数变换

上图进行了目标函数和适应度函数的指数变换，并且使其符合“目标函数值越大，适应度越小”的约定，同时，适应度大的个体将会“更加优秀”。指数变换可以突出优秀个体，加速搜索最优解的速度，但不利于种群多样性的保留。

在适应度计算中我们常常引入“罚函数”来解决带复杂约束的优化问题。“罚函数”通过对非可行解施加惩罚，以此来降低不符合约束条件的“非可行解”个体在下一代中的生存概率。在遗传算法中我们不完全否定非可行解。因为在搜索空间中非可行解有可能非常接近最优可行解。

下面介绍一种基于等级划分的适应度分配计算 (Rank-based fitness assignment)：

这种适应度计算方法不是直接对目标函数进行变换来计算的个体的适应度的，而是先根据目标函数值来对个体进行排序，然后根据个体在排序中的位置来确定其适应度。

这种适应度计算方法克服了基于目标函数变换来计算适应度值所带来的放缩问题(因适应度放缩不当而导致的选择压力(其概念详见下一节)过小导致的搜索收敛过慢或过分放大优势个体的适应度而导致的搜索收敛过快)。基于等级划分的适应度分配计算在整个种群中设定了统一的比例，并提供了一种控制选择压力的简单有效的办法。

基于等级划分的适应度分配计算比简单的基于比例或目标函数变换的适应度计算更加健壮，所以，它是值得选择的方法。[BH91],[Why89]

这种适应度计算中涉及目标函数值的线性与非线性排序两种方式：

1) 线性排序：

设种群规模(即种群个体数)为 $N_{ind}$ ,  $i$ 为个体在种群中的位置 $i = 1, 2, \dots, N_{ind}$ , 根据个体的目标函数值从大到小对个体进行排序，设 $SP$ 为选择压力。则个体的适应度计算如下：

$$Fitness_i = 2 - SP + 2(SP - 1) \frac{i - 1}{N_{ind} - 1}$$

线性排序中选择压力 $SP$ 的值必须在 [1.0,2.0] 之间。

[BT95] 中有这种线性排序的详细分析。其选择强度、多样性损失、选择方差(这些概念详见下一节)的计算如下：

选择强度：

$$SelInt(SP) = \frac{SP - 1}{\sqrt{\pi}}$$

多样性损失：

$$LossDiv(SP) = \frac{SP - 1}{4}$$

选择方差：

$$SelVar(SP) = 1 - \frac{(SP - 1)^2}{\pi} = 1 - SelInt(SP)^2$$

2) 非线性排序：

在 [Poh95] 中引入了一种使用非线性排序的新方法。使用非线性排序比线性排序方法允许设定更高的选择压力。

$$Fitness_i = \frac{N_{ind} \cdot X^{i-1}}{\sum_{i=1}^{N_{ind}} X^{i-1}}$$

其中 $X$ 为下面多项式的根：

$$(SP - N_{ind}) \cdot X^{N_{ind}-1} + SP \cdot X^{N_{ind}-2} + \dots + SP \cdot X + SP = 0$$

非线性排序允许选择压力 $SP$ 的值在 [1,  $N_{ind} - 2$ ] 之间。

上面的适应度计算方法都是对单目标问题而言的。然而，在许多现实问题中，为评估个体的质量好坏，必须考虑多个标准才能确定个体的优越性。这就涉及到多目标的优化问题。我们可以通过线性加权法把多个单目标的目标函数值加权得到多目标的目标函数值，进而使用上面所述的一些单目标问题的适应度计算方法。也可以采用特殊的方法，如帕累托排序法等。

## 4 选择

选择是指根据个体适应度从种群中挑选个体的过程。其对应的是生物学中的“自然选择”。下面是一些选择操作有关的术语：

**选择压力 (selective pressure):** 与所有个体的平均选择概率相比，最优个体被选中的概率。

**偏差 (bias):** 个体标准化的适应度预期期望再生概率之差的绝对值。

**个体扩展 (spread):** 在进行选择时，比较优秀个体可能会被多次选择。该参数就限定了最多可以被重复选择的个数。

**多样性损失 (loss of diversity):** 进行选择操作后，未被选择的个体数目占种群个体总数的比例。

**选择强度 (selection intensity):** 把标准高斯分布用在选择操作后的种群个体平均适应度的期望值。选择强度越大，遗传算法的最优化搜索的收敛速度就越快，但也往往意味着多样性的损失。

**选择方差 (selection variance):** 把标准高斯分布用在选择操作后的种群个体适应度的方差。

选择操作一般是基于个体的适应度来计算的(详见上一节)，下面介绍一些经典的选择算子：

1) 轮盘赌选择 (Roulette wheel selection):

轮盘赌选择是一种有回放的随机抽样选择法。种群的个体被映射到区间的连续片段，每个个体所在片段的长度与其适应度成比例。生成随机数，根据其所落在的片段选择对应的个体，并重复该过程直到获得所需数量的个体。

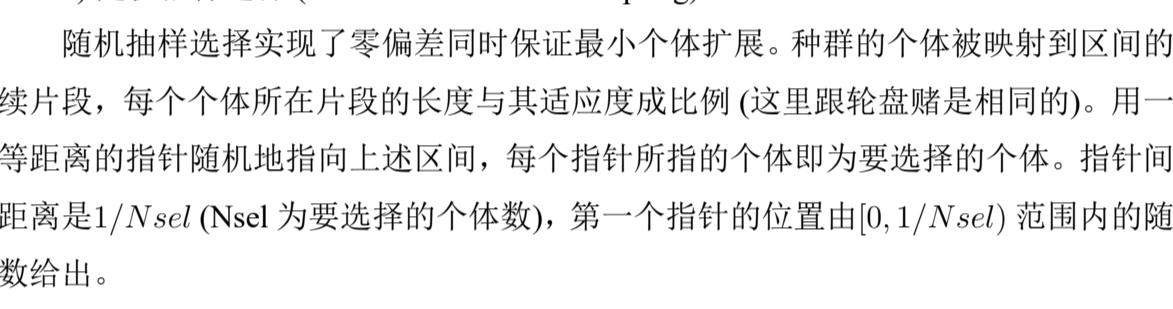


图 1 轮盘赌选择

所选择的个体为：2, 1, 5, 3, 8, 2。

轮盘赌选择算法实现了零偏差 (bias) 但不保证最小个体扩展 (spread)，因此，上面的例子中，2号个体被选择了两次(有可能所选的个体全部都是同一个)。

2) 随机抽样选择 (Stochastic universal sampling):

随机抽样选择实现了零偏差同时保证最小个体扩展。种群的个体被映射到区间的连续片段，每个个体所在片段的长度与其适应度成比例(这里跟轮盘赌是相同的)。用一组等距离的指针随机地指向上述区间，每个指针所指的个体即为要选择的个体。指针间的距离是 $1/N_{sel}$ ( $N_{sel}$ 为要选择的个体数)，第一个指针的位置由 $[0, 1/N_{sel}]$ 范围内的随机数给出。

假设要选择6个个体，那么指针间的距离则为 $1/6 \approx 0.167$ ，选择情况如下图：

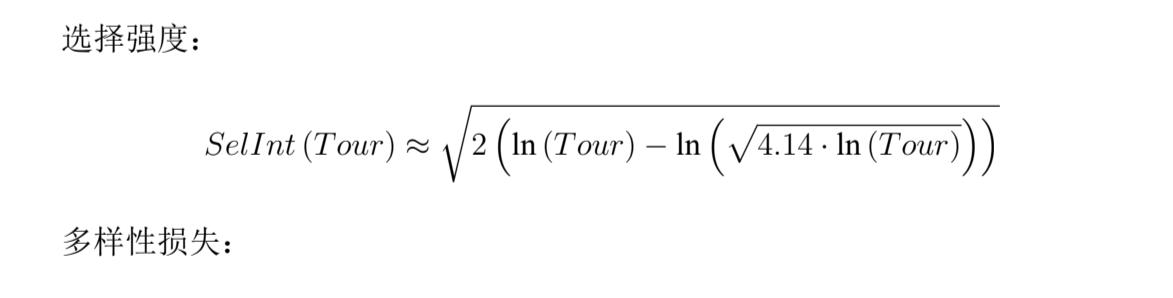


图 2 随机抽样选择

所选择的个体为：1, 2, 3, 5, 6, 9。

轮盘赌选择和随机抽样选择都属于轮盘赌模型(或称基于适应度比例的选择模型)，但后者通常能够得到更想要得到的结果，种群多样性也得以保证。

某些人认为随机抽样选择是不考虑个体的适应度、纯粹依靠随机数的结果来进行个体选择，这种想法是极其错误的。这种选择方法实际上会让遗传算法变为盲目随机搜索，而无法发挥适应度应该起到的指导搜索方向的作用。

3) 锦标赛选择 (Tournament selection):

该选择算法模仿淘汰赛制，每次从种群中挑选 $Tour$ 个个体进行比较，从中选出一个最好的个体加入被选集合。重复该操作，直到被选集合的大小达到需要选择的个体数。

$Tour$ 即为竞赛规模。在[BT95]中，可以看到锦标赛选择的相关理论分析。

选择强度：

$$SelInt(Tour) \approx \sqrt{2 \left( \ln(Tour) - \ln(\sqrt{4.14 \cdot \ln(Tour)}) \right)}$$

多样性损失：

$$LossDiv(Tour) = Tour^{-\frac{1}{Tour-1}} - Tour^{-\frac{Tour}{Tour-1}}$$

选择方差：

$$SelVar(Tour) \approx \frac{0.918}{\ln(1.186 + 1.328 \cdot Tour)}$$

4) 截断选择 (Truncation selection):

截断选择是与前面的几种选择算法比较不一样的算法。它更接近于生物学里的“人工育种”，适用于大规模种群的个体选择。

在截断选择中，根据适应度对种群个体进行分类。设定截断阈值为 $Trunc$ (表示选择作为父母的个体的比例)，低于阈值的个体不会产生后代。”选择强度”这个术语就经常用在截断选择中，“选择强度”和截断阈值之间的关系非常密切。

在[BT95]中可以看到截断选择的相关理论分析。

5) 本地选择 (Local selection):

在本地选择算法中，每个个体处在一个约束环境中(称为“本地邻域”)。个体仅与同一邻域内的个体相互作用。邻域是根据种群的结构来定义的，可以是线性的、二维的或者是三位及更加复杂的。邻域之间的距离决定了邻域的大小，而邻域的大小决定了种群个体之间遗传信息的传播速度，即决定了种群是快速繁殖还是维持高度的种群多样性。[VBS91]的模拟中得出了类似的结果，在较小的邻域中进行的本地选择比在较大的邻域中要好。

选择算法依赖于适应度的计算，不同的适应度计算方法会使得种群个体的适应度呈现不一样的分布特征，从而影响到选择操作的效果。

## 5 重组

遗传算法中的重组有时称为“交叉”，重组包含了交叉。重组算法是改进遗传算法最有效的环节，它通过结合交配群体中包含的遗传信息产生新的个体。因为遗传算法中有二进制编码、实值编码、排列编码、树编码等，因此必须也有与编码方式相适应的不同重组算法。

下面介绍几种经典的重组算法：

### 1) 重组算法的代表——离散重组算法 (Discrete recombination):

离散重组算法在个体间执行变量值的交换，在生成交配个体时，交配个体中每个变量可以等概率地挑选一个父个体对应变量作为自身的值。其几何特征表现如下，离散重组产生了父代所在的超立方体的角：

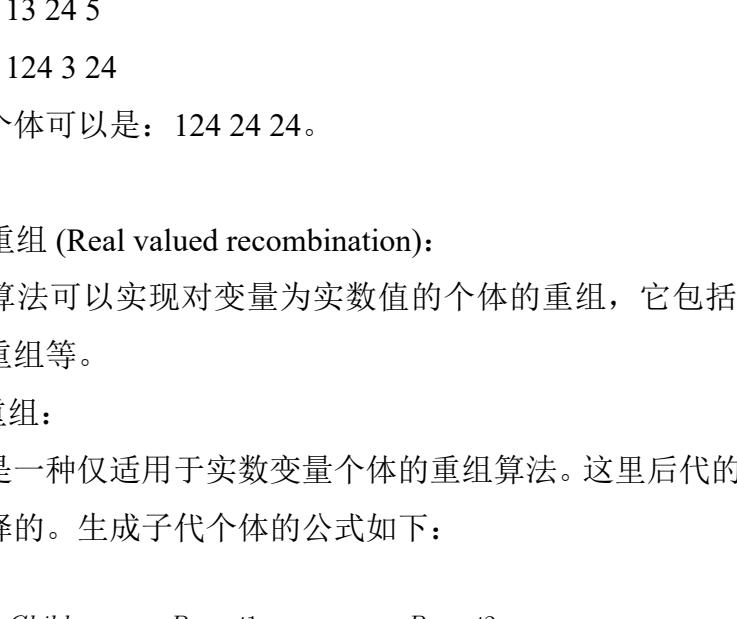


图 1 离散重组产生的可能的后代在解空间上的位置

考虑以下两个个体，每个个体有 3 个变量 (3 维)：

父个体 1: 13 24 5

父个体 2: 124 3 24

生成的子个体可以是：124 24 24。

### 2) 实数值重组 (Real valued recombination):

实值重组算法可以实现对变量为实数值的个体的重组，它包括中间重组、线性重组、扩展线性重组等。

#### 2.1) 中间重组:

中间重组是一种仅适用于实数变量个体的重组算法。这里后代的变量值是在父辈变量的区间上选择的。生成子代个体的公式如下：

$$Var_i^{Child} = Var_i^{Parent1} \cdot \alpha_i + Var_i^{Parent2} \cdot (1 - \alpha_i), i \in [1, 2, \dots, N]$$

其中， $\alpha_i$  是  $[-d, 1 + d]$  之间的随机数，它是一个随机均匀选择的比例因子。

参数  $d$  的值代表可能产生的后代的区域大小。 $d = 0$  表示后代的变量值的区域大小与父代是一样的，此时称为“(标准的) 中间重组”。但是，由于后代的大多数变量不是在可能区域的边界上生成的，因此变量所覆盖的面积有可能会越来越小。因此，仅用  $d = 0$  的标准中间重组就会发生这种变量空间收缩现象。因此，通过设置更大的  $d$  值可以防止这种现象。一般设置  $d = 0.25$ ，此时可以在统计学上保证后代的变量值的范围不会缩小。如图所示：

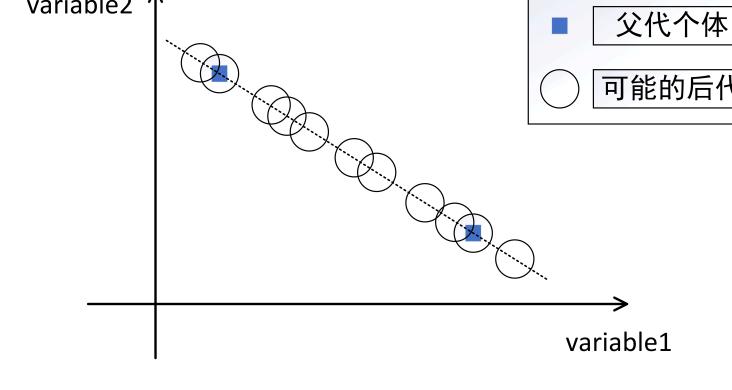


图 2 中间重组下父代与子代的变量值的区域范围比较

例如父个体为：

父个体 1: 0.4 1.2 -0.3

父个体 2: 0.2 0.7 0.6

生成的子个体可以是：0.3 0.9 0.4。

中间重组能够稍微超出父代所在的超立方体的边界，如图所示：

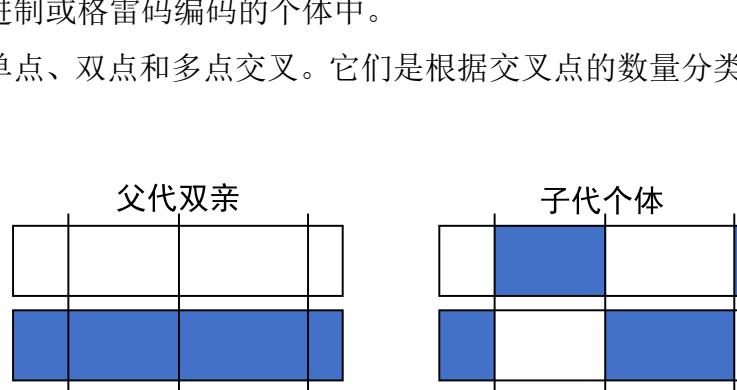


图 3 中间重组后可能的子代变量在解空间的位置

另外还有扩展线性重组，其详细介绍可以参见 [Müh94]。

### 3) 离散值重组 (Discrete valued recombination):

离散值重组的概念和方法跟实值重组类似，只不过在算法执行的过程中要限制个体基因均为整数值。

### 4) 值互换重组——交叉 (Values exchanged recombination——crossover):

这种重组方式就是两个父代个体交换染色体片段产生新个体的过程，因此也称为交叉操作。

交叉操作中，个体的染色体编码可以是实值、也可以是二进制，比如：

[0 1 0 1 1 1 0 1] 和 [1.1 2.0 3.4 2.7 1.6] 这些类型的染色体都可以进行交叉操作，一般更多地用在二进制或格雷码编码的个体中。

交叉有单点、双点和多点交叉。它们是根据交叉点的数量分类的。其中多点交叉的示意图如下：



图 5 多点交叉

此外还有均匀交叉、洗牌交叉，这里就不一一赘述了。

这里要介绍一下“减少代理”交叉：

#### 4.1) 减少代理交叉 (Crossover with reduced surrogate):

上面的交叉算法的交叉结果可能会产生和父代性状一样的个体，如果在遗传算法中想让交叉得到的子代中更多的个体拥有与父代个体不一样的性状，这时就可以用减少代理的交叉算法。

减少代理交叉算法尽可能地产生全新性状的个体，这是通过限制交叉点的位置来实现的——控制交叉点只出现在父代两个交叉个体的基因值不同的地方。

#### 4.2) 匹配交叉 (matched crossover):

对于排列编码的个体，染色体中每个变量的值都是独一无二的。这意味着不能使用上面所述的交叉算法。匹配交叉算法通过计算父代两个交叉个体中互相呈现中心对称的基因片段来设置交叉点，再进行基因片段互换。

## 6 变异

变异是指通过改变父代染色体中的一部分基因来形成新的子代染色体的过程。它能够保持种群的多样性，降低遗传算法陷入局部最优解风险。

最经典的两种突变方法是实数值突变和离散突变。

### 1) 实数值突变 (Real valued mutation):

实值突变通过给定的变异概率将父代个体的一个或多个变量的值替换成在同一范围内的随机的新值，从而生成新的个体。例如：

父代个体：[1.1 1.5 2.6 3.1 2.3]，其中第 2 个基因位发生突变，可能产生的后代为：  
[1.1 1.7 2.6 3.1 2.3]。

### 2) 离散突变 (Discrete valued mutation):

离散突变与实数值突变的概念和方法类似，只不过在算法执行的过程中要限制个体的基因均为整数值。离散突变经常应用于二进制编码的种群。

### 3) 互换突变 (Exchanged mutation):

对于排列编码的个体，染色体基因中每个变量的值都是独一无二的，这意味着进行突变操作后依然要保持染色体的这个特征。互换突变算法在染色体上随机确定两个基因片段并进行互换。这两个片段必须包含相同数量的基因。如图所示：

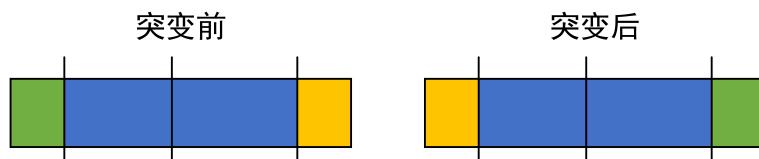


图 1 互换突变

除了这些经典突变方法外，另外在进化策略和进化规划中，还有能够调整步长的突变方法。本文档主要介绍遗传算法，因此在这里就不展开介绍了。

## 7 重插入

通过前面的选择、重组和变异后，我们得到的是育种后代，此时育种后代的个体数有可能会跟父代种群的个体数不相同。这时，为了保持种群的规模，这些育种后代可以重新插入到父代中，替换父代种群的一部分个体，或者丢弃一部分育种个体，最终形成子代种群。

重插入方案有以下两种：

### 1) 全局重插入 (Global reinsertion):

这种插入算法适用于使用除了本地选择外的其他选择算法所生成的交配池。假如在选择操作时使用了本地选择算法，那么重插入中就不能用全局重插入算法了。

- 育种个体与父代个体一样多，直接用所有的育种个体生成新一代种群 (完全重新插入)。
- 育种个体比父代个体少，把育种个体随机均匀地替换父代的个体 (均匀重新插入)。
- 育种个体比父代个体少，取代适应度较低的父代个体 (精英重新插入)。
- 育种个体比父代个体多，重新插入适应度较大的育种个体 (精英保留重新插入)

完全重新插入是最简单的重插入算法。然而最坏的情况是：父代中最优秀的个体并没有繁殖产生优秀的个体，此时在重插入时又被最差的个体替代，因此种群丢失了精英。

因此在多数情况下，推荐使用精英重新插入和精英保留重新插入的方法。

### 2) 本地重插入 (Local reinsertion):

在前面章节中介绍过本地选择算法，它在有界邻域中选择个体。那么育种后代的重插入也应该发生在相同的邻域中。这是一种基于位置信息的重插入算法，重插入时使用的邻域要跟本地选择中的一样。

以下的方法是可行的：

- 把育种个体随机均匀地替换其所在邻域内的父代个体。
- 把育种个体替换其所在邻域内的最差的父代个体。
- 把更适合插入到相邻邻域的育种个体替换相邻邻域中的最差的父代个体。
- 把更适合插入到相邻邻域的育种个体随机均匀地替换相邻邻域中的父代个体。

## 8 多目标进化优化

在许多实际问题中，我们常常要处理的数学模型不止有一个目标函数。例如在产品加工与配送系统中，通常要求加工和配送的成本尽可能低，而所花的时间尽可能少，从而使总利润最大。有些多目标优化问题中各个目标之间会有冲突，无法同时取得最优，例如工人的工资和企业的总利润。

以遗传算法为代表的许多进化算法，具有生成多个点并进行多方向搜索的特征，因此非常适合求解这种最优解的搜索空间非常复杂的多目标优化问题。

### 8.1. 多目标优化问题数学模型及最优解

多目标优化问题是在给定约束条件的前提下，求多个目标函数的最大或最小的问题。一般可表述为如下形式：

$$\begin{aligned} & \max \{z_1 = f(x), z_2 = f(x), \dots, z_r = f(x)\} \\ & s.t. \quad g_i(x) \leq 0, \quad i = 1, 2, \dots, m \end{aligned}$$

这里， $z_k = f_k(x), k = 1, 2, \dots, r$  是  $r$  个线性或非线性的目标函数。有的可能是最大化目标函数，有的可能是最小化目标函数。为了方便处理，可以把各目标函数统一换为最小化或最大化。多个目标之间可能会拥有不同的单位，也可能在优化某个目标时损害其他目标。但这并不意味着多目标优化问题可能没有最优解，事实上是可以有的，为了求出比较合理的解，这里引入多目标优化问题的合理解集——Pareto 最优解 (pareto optimal solution)。

### 8.2. Pareto 最优解

在求解单目标问题时，可以在所有候选解中选出唯一最好的解。但是在多目标优化问题里，由于各个目标之间可能存在冲突，所以最优解不一定只有一个。我们如下定义多目标的最优解：

1) 给定一个多目标优化问题  $\min f(x)$ ，设  $X^* \in \Omega$ ，如果  $\neg \exists X \in \Omega$ ，使得满足以下条件：

对于  $f(x)$  的任意子目标函数  $f_i(x)$  都有  $f_i(X) \leq f_i(X^*)$ ，同时至少存在一个子目标函数  $f_i(x)$  使得  $f_i(X) < f_i(X^*)$

那么，我们称  $X^*$  是一个强 Pareto 最优解。

另外我们还可以定义弱 Pareto 最优解：

2) 给定一个多目标优化问题  $\min f(x)$ ，设  $X^* \in \Omega$ ，如果  $\neg \exists X \in \Omega$ ，使得满足以下条件：

对于  $f(x)$  的任意子目标函数  $f_i(x)$  都有  $f_i(X) < f_i(X^*)$

那么，我们称  $X^*$  是一个弱 Pareto 最优解。

例如一个两个目标的目标空间：

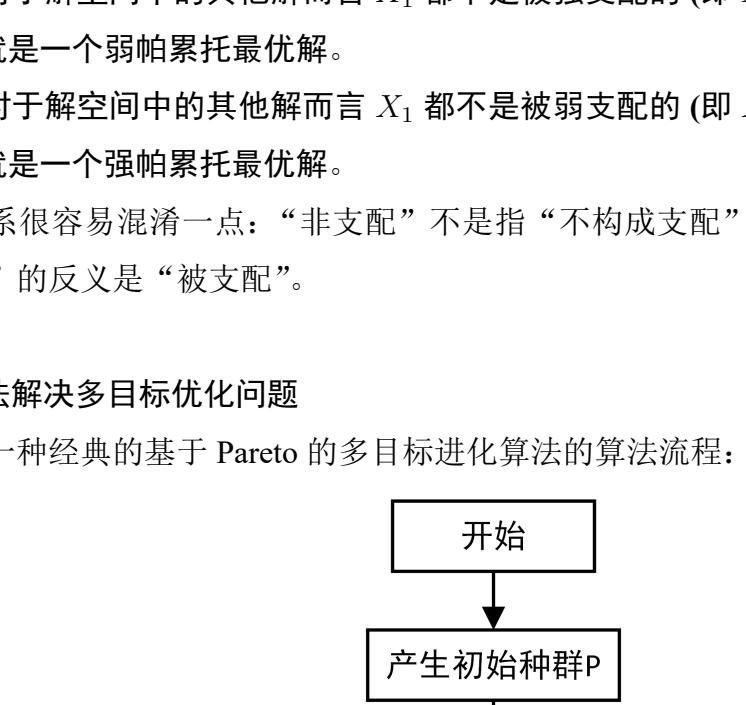


图 1 强、弱 Pareto 最优解示例

由上面的定义可知，在目标空间  $S$  中强帕累托最优解均落在粗曲线上；弱帕累托最优解则都落在细的直线上。

值得一提的是，我们常常习惯把强 Pareto 最优解简称为 Pareto 最优解。另外，Pareto 最优解又称为“非支配解”。下面我们就仔细研究解的支配关系。

### 8.3. 解的支配关系

设  $X_1, X_2$  均是解空间  $\omega$  中的解，若对于所有的目标，均有  $X_1$  比  $X_2$  好，那么就称  $X_1$  强支配  $X_2$ 。若对于所有的目标，均有  $X_1$  不差于  $X_2$ ，且至少存在一个目标使得  $X_1$  比  $X_2$  好，那么就称  $X_1$  弱支配  $X_2$ 。

一般我们把弱支配关系简称为支配关系。对于上面的  $X_1$  和  $X_2$ ，我们称  $X_1$  是“非强(弱)支配”的或简称“非支配”的， $X_2$  是“被强(弱)支配”或简称“被支配”的。“非支配”即“不被支配”。非支配是相对的， $X_1$  对于  $X_2$  是非支配的，但有可能  $X_1$  对于  $X_3$  就是被支配的了。

另外还有一种“不相关”关系。当对于某些目标有  $X_1$  比  $X_2$  好，而另外一些目标有  $X_2$  比  $X_1$  好，那么  $X_1$  和  $X_2$  就是不相关的。

若  $X_1$  对于解空间中的其他解而言  $X_1$  都不是被支配的，那么此时  $X_1$  就是一个帕累托最优解。此时我们就可以把支配关系和帕累托最优解联系起来：

1) 当  $X_1$  对于解空间中的其他解而言  $X_1$  都不是被强支配的 (即  $X_1$  是非强支配的)，那么此时  $X_1$  就是一个弱帕累托最优解。

2) 当  $X_1$  对于解空间中的其他解而言  $X_1$  都不是被弱支配的 (即  $X_1$  是非弱支配的)，那么此时  $X_1$  就是一个强帕累托最优解。

上面的关系很容易混淆一点：“非支配”不是指“不构成支配”，而是指“不被支配”。“非支配”的反义是“被支配”。

### 8.4. 用进化算法解决多目标优化问题

下面展示一种经典的基于 Pareto 的多目标进化算法的算法流程：

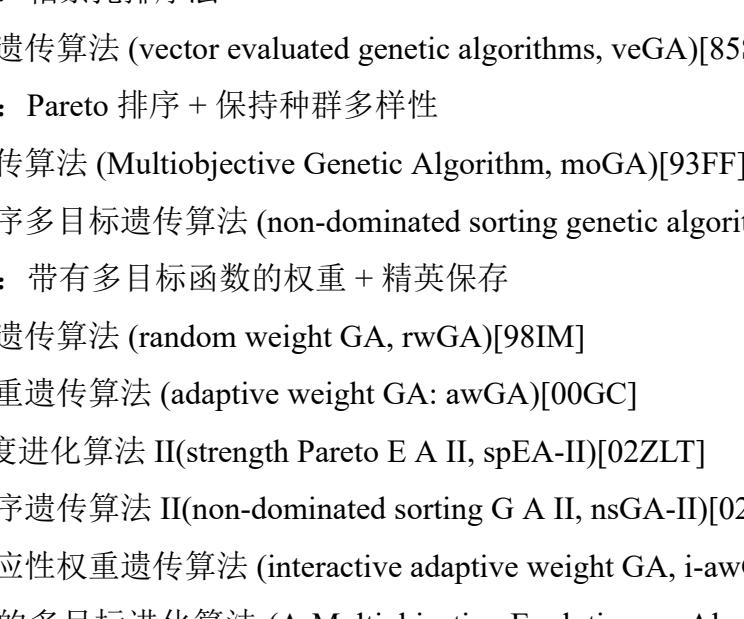


图 2 一种经典的基于 Pareto 的多目标进化算法的算法流程

进化算法如遗传算法、粒子群算法、差分进化算法等都可以很好地解决多目标优化问题。以遗传算法为例，目前具有代表性的多目标优化遗传算法按历史进程分类如下：

1) 第一代：帕累托排序法

向量评价遗传算法 (vector evaluated genetic algorithms, veGA)[85SM]

2) 第二代：Pareto 排序 + 保持种群多样性

多目标遗传算法 (Multiobjective Genetic Algorithm, moGA)[93FF]

非支配排序多目标遗传算法 (non-dominated sorting genetic algorithm, nsGA)[95Deb]

3) 第三代：带有多目标函数的权重 + 精英保存

随机权重遗传算法 (random weight GA, rwGA)[98IM]

Pareto 强度进化算法 II(strength Pareto E A II, spEA-II)[02ZLT]

非支配排序遗传算法 II(non-dominated sorting G A II, nsGA-II)[02DAM]

交互式适应性权重遗传算法 (interactive adaptive weight GA, i-awGA)[06LG]

基于分解的多目标进化算法 (A Multiobjective Evolutionary Algorithm Based on Decomposition, MOEA/D)[07ZL]

对于一个具体的多目标进化算法 (Multi-objective Evolutionary Algorithm : MOEA) 而言，如何构造非支配集，采用什么策略来调整非支配集的大小，如何保存非支配集中解的分布性，是决定其算法性能的重要内容，这些也是当前相关研究的热点。

# Geatpy 教程

## 简介

Geatpy 是一个 Python 进化算法库，由华南理工大学、华南农业大学、德州奥斯汀公立大学学生联合团队开发。它提供了许多已实现的遗传算法各项操作的函数，如初始化种群、选择、交叉、变异、重插入、多种群迁移等。你可以清晰地看到其基本结构及相关算法的实现，并利用 Geatpy 函数方便地进行自由组合，实现多种改进的遗传算法、多目标优化、并行遗传算法等，解决传统优化算法难以解决的问题。

Geatpy 提供简便易用的 Python 进化算法框架。除了简单的函数封装之外，Geatpy 提供了许多能够直接帮助解决实际问题的进化算法模板。这些都是开源的，你可以参照这些模板，加以改进或重构，以便让 Geatpy 实现的遗传算法编程与你当前正在进行的项目进行融合。

目前 Geatpy 主要支持遗传算法的编程，除了遗传算法外，Geatpy 还能进行进化策略与进化规划。这些后续会给出相关的案例。

## 特色

1. Geatpy 是一个功能强大的遗传算法库。没有过于抽象的复杂封装，入门门槛低。它提供多种格式的编码方式以及丰富的选择、交叉和变异算子。你可以在极短的时间里掌握 Geatpy 的用法，并把 Geatpy 融合到你正在进行的 Python 项目或实际问题的解决方案中。

2. Geatpy 的一大特色是提供众多自由的进化算法模板。在模板中，你可以清晰地看到算法的完整流程，更加掌握遗传算法的更多细节。通过修改进化算法模板，你可以轻松解决难以想象多的优化问题。你可以将 Geatpy 用作研究遗传和进化算法的通用测试平台，实现各种改进的遗传和进化算法。

3. 利用 Numpy+mkl 实现高度的矩阵化计算，使得 Geatpy 在算法执行效率上有着不俗的表现。

4. Geatpy 突破了传统进化算法库难以突破的大数计算问题，意味着你可以利用 Geatpy 进行超大整数的运算，从而对超大范围内的数据进行搜索和优化。

# Geatpy 总览

## 1. Geatpy 层次结构

Geatpy 是简单封装的开放式进化算法框架，可以方便、自由地与其他算法或项目相结合。其层次结构如下图：

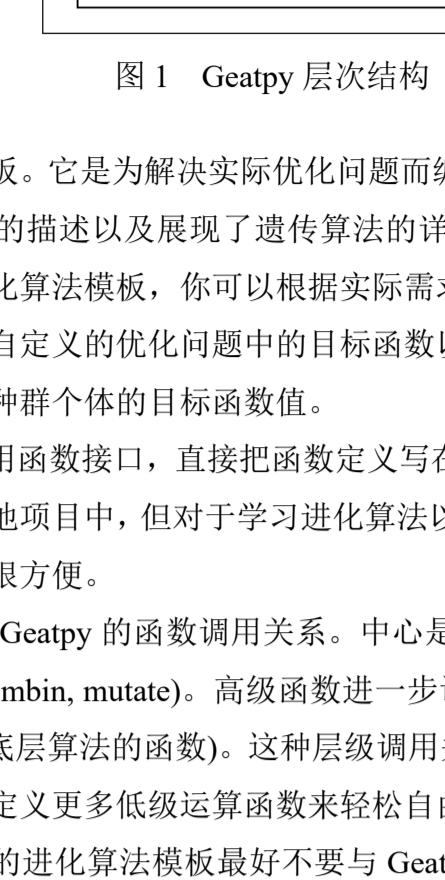


图 1 Geatpy 层次结构

中心是进化算法模板。它是为解决实际优化问题而编写的模板函数或脚本。里面清晰地定义了相关的变量的描述以及展现了遗传算法的详细流程。Geatpy 本身自带几种用于解决常见问题的进化算法模板，你可以根据实际需求修改或重构模板。

函数接口是指用户自定义的优化问题中的目标函数以及罚函数。进化算法模板可以调用该函数接口来计算种群个体的目标函数值。

此外，你也可以不用函数接口，直接把函数定义写在进化算法模板内部，虽然这样不利于扩展和接入到其他项目中，但对于学习进化算法以及做一些相关基础性实验或者解决一些简单问题也很方便。

下面第 2 点展示了 Geatpy 的函数调用关系。中心是进化算法模板，它调用高级的运算函数 (selecting, recombin, mutate)。高级函数进一步调用相关的低级运算函数 (即实现选择、交叉、变异等底层算法的函数)。这种层级调用关系使 Geatpy 的结构十分清晰，更重要的是，你可以自定义更多低级运算函数来轻松自由地扩展 Geatpy。

**特别注意：**自定义的进化算法模板最好不要与 Geatpy 自带的模板名称重复，否则会造成难以处理的冲突问题。

## 2. Geatpy 调用关系

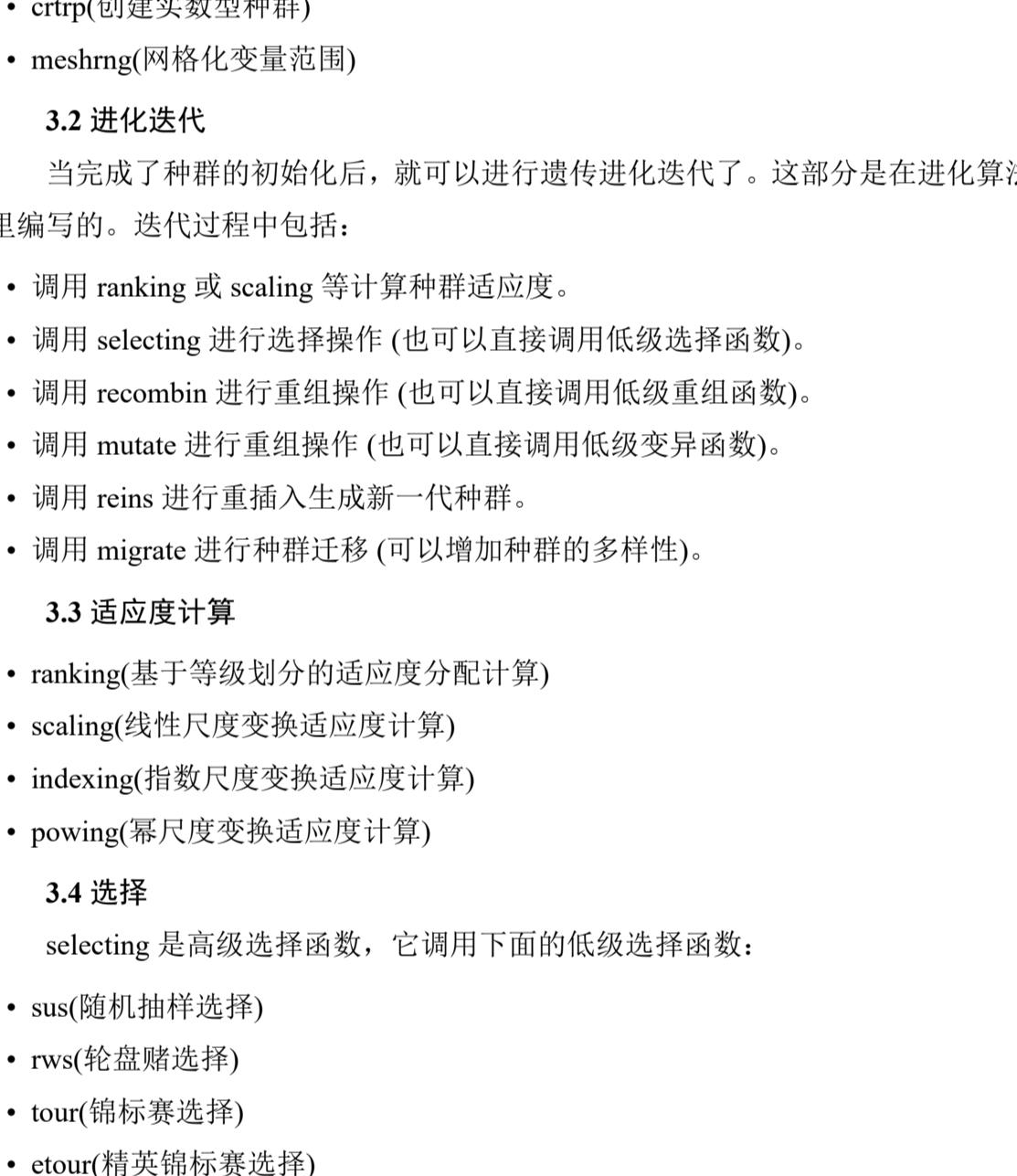


图 2 Geatpy 调用关系

图中黑色的表示 Geatpy 中内置的函数 (当然用户也可以自定义与之类似的函数); 红色的表示是用户自定义的。其中，templet\* 是进化算法模板，\* 表示名字是多样的； aimfunc\* 是优化目标函数； crt\* 用于创建种群染色体矩阵 (简称“种群矩阵”); bs2\* 用于二进制/格雷码种群的解码； ranking, powing, scaling, indexing 均是用于计算种群适应度的函数； migrate 是种群迁移函数； selecting、recombin、mutate 分别是高级的选择、重组和变异函数，对应下方的都是低级的选择、重组和变异函数； reins 是重插入操作函数； \*plot 是以 plot 名字结尾的用于可视化输出函数。\*stat 是以 stat 名字结尾的统计函数，\*NDSet 是与多目标优化帕累托最优解集的处理有关的函数。

上述结构中，aimfuc\*，templet\*，trcplot，replot 等都是可以自定义的，自定义时必须遵循相关数据结构 (详见“Geatpy 数据结构”章节)。

## 3. Geatpy 主要功能

以下是 Geatpy 的内核函数，其详细用法可以参见“Geatpy 函数”章节。

### 3.1 与初始化种群相关的函数

- crtfd(生成区域描述器)
- crtbp(创建简单离散种群、二进制编码种群)
- crtip(创建整型种群)
- crtpp(创建排列编码种群)
- crtrp(创建实型种群)
- meshrng(网格化变量范围)

### 3.2 进化迭代

当完成了种群的初始化后，就可以进行遗传进化迭代了。这部分是在进化算法模板里编写的。迭代过程中包括：

- 调用 ranking 或 scaling 等计算种群适应度。

- 调用 selecting 进行选择操作 (也可以直接调用低级选择函数)。

- 调用 recombin 进行重组操作 (也可以直接调用低级重组函数)。

- 调用 mutate 进行重组操作 (也可以直接调用低级变异函数)。

- 调用 reins 进行重插入生成新一代种群。

- 调用 migrate 进行种群迁移 (可以增加种群的多样性)。

### 3.3 适应度计算

- ranking(基于等级划分的适应度分配计算)

- scaling(线性尺度变换适应度计算)

- indexing(指数尺度变换适应度计算)

- powing(幂尺度变换适应度计算)

### 3.4 选择

selecting 是高级选择函数，它调用下面的低级选择函数：

- sus(随机抽样选择)

- rws(轮盘赌选择)

- tour(锦标赛选择)

- etour(精英锦标赛选择)

### 3.5 重组

重组包括了交叉。recombin 是高级的重组函数，它调用下面的低级重组函数：

- recdis(离散重组)

- recint(中间重组)

- reclin(线性重组)

- xovdp(两点交叉)

- xovdprs(减少代理的两点交叉)

- xovmp(多点交叉)

- xovpm(部分匹配交叉)

- xovsh(洗牌交叉)

- xovshrs(减少代理的洗牌交叉)

- xovsp(单点交叉)

- xovsprs(减少代理的单点交叉)

### 3.6 突变

mutate 是高级的突变函数，它调用下面的低级突变函数：

- mut(简单离散变异算子)

- mutbga(实数值变异算子)

- mutbin(二进制变异算子)

- mutgau(高斯突变算子)

- mutint(整数值变异算子)

- mutpp(排列编码变异算子)

### 3.7 重插入

reins 是重插入函数，它将育种个体重插入到父代种群中，生成新一代种群。

### 3.8 种群迁移

当使用多种群设计时，可用 migrate 函数实现种群中的个体迁移。

### 3.9 染色体解码

对于二进制/格雷编码的种群，我们要对其进行解码才能得到其表现型。

- bs2int(二进制/格雷码转整数)

- bs2rv(二进制/格雷码转实数)

### 3.10 可视化

- frontplot(多目标优化帕累托前沿绘图函数)

- sgaplot(单目标进化动态绘图函数)

- trcplot(单目标进化跟踪器绘图)

### 3.11 多目标相关

- ndomin(简单非支配排序)

- ndomindeb(Deb 非支配排序)

- ndominfast(快速非支配排序)

- redisNDSet(基于聚集距离的帕累托最优子集筛选)

- upNDSet(更新帕累托最优集)

### 3.12 模板相关

- sga\_real\_templet(单目标进化算法模板 (实值编码))

- sga\_code\_templet(单目标进化算法模板 (二进制/格雷编码))

- sga\_permut\_templet(单目标进化算法模板 (排列编码))

- sga\_new\_real\_templet(改进的单目标进化算法模板 (实值编码))

- sga\_new\_code\_templet(改进的单目标进化算法模板 (二进制/格雷编码))

- sga\_new\_permut\_templet(改进的单目标进化算法模板 (排列编码))

- sga\_mpc\_real\_templet(基于多种群竞争进化单目标编程模板 (实值编码))

- sga\_mps\_real\_templet(基于多种群独立进化单目标编程模板 (实值编码))

- moea\_awGA\_templet(基于适应性权重法 (awGA) 的多目标优化进化算法模板)

- moea\_rwGA\_templet(基于随机权重法 (rwGA) 的多目标优化进化算法模板)

- moea\_nsGA2\_templet(基于改进 NSGA-II 算法的多目标优化进化算法模板)

- moea\_q\_sorted\_new\_templet(改进的快速非支配排序法的多目标优化进化算法模板)

- moea\_q\_sorted\_templet(基于快速非支配排序法的多目标优化进化算法模板)

## 快速入门

### 1. 安装

你可以通过联网安装或本地安装来安装 Geatpy。在安装前，你需要安装 Numpy。

联网安装：pip install geatpy

本地安装：在 github 下载 Geatpy 安装包，在本地控制台执行以下代码安装：

```
python setup.py install
```

更新方法：

```
pip install --upgrade geatpy
```

提示：若在安装时遇到 pip 的“UnicodeDecodeError: 'utf-8'" 错误，一般是因 Windows 用户名为中文而导致。相关解决办法可以在网上搜索到。

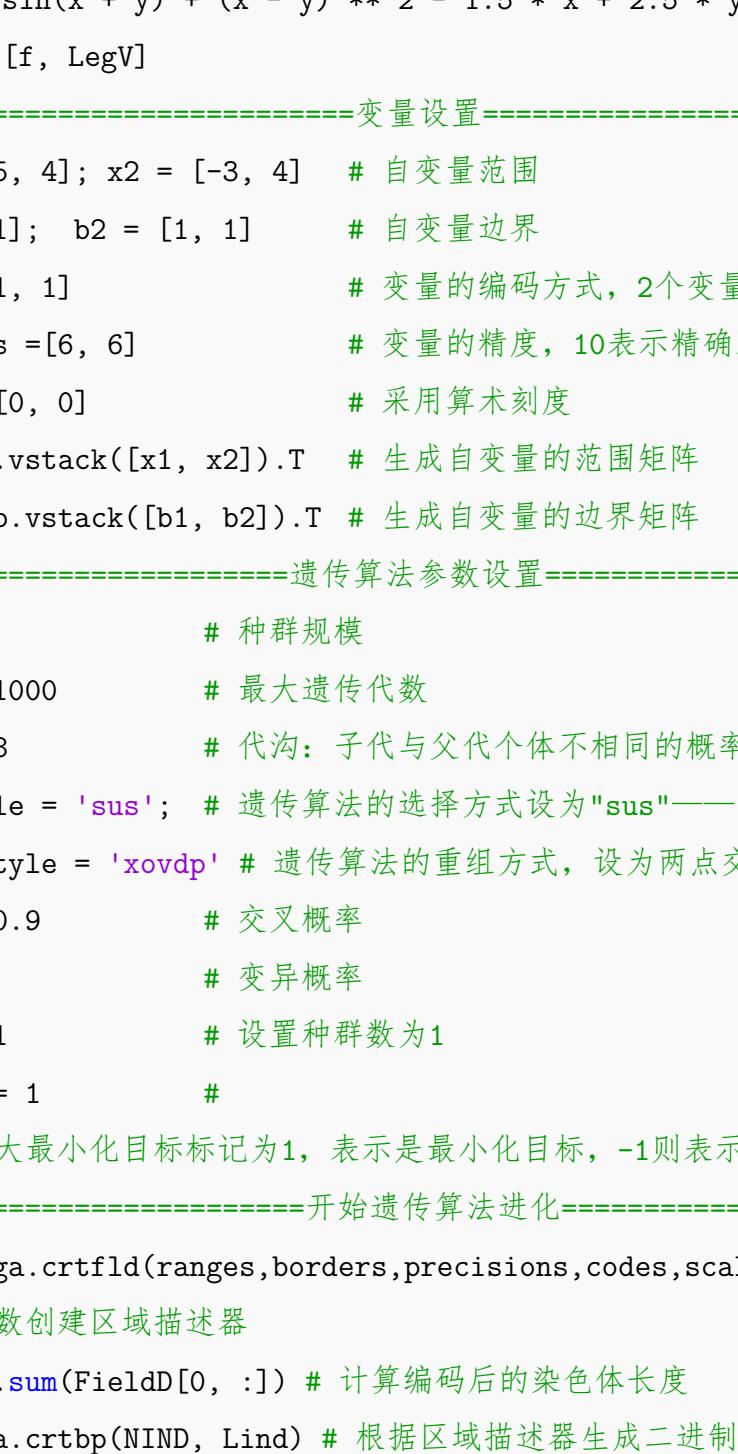
### 2. 入门示例

Geatpy 可以用最基础的脚本方式编写代码求解问题，我们将用它来快速入门，其代码风格与 Matlab 极为类似（有兴趣可以跟 Matlab 的 gatbx 和 GEATbx 工具箱用法进行比较）。

下面研究标准测试函数——MoCormick 函数的最小化搜索问题，其表达式为：

$$f(x, y) = \sin(x + y) + (x - y)^2 - 1.5x + 2.5y + 1$$

它是一个二元函数，它具有一个全局极小点： $f = (-0.54719, -1.54719) = -1.9133$ ，函数图像如下：



编写如下执行脚本：

```
# -*- coding: utf-8 -*-
"""

执行脚本quickstart_demo.py
描述：
本demo展示了如何不使用Geatpy提供的进化算法框架、
纯粹地写编程脚本来解决一个无约束的单目标优化问题。
优化目标: f = min(np.sin(x + y) + (x - y) ** 2 - 1.5 * x + 2.5 * y + 1)
注意目标函数aimfuc的输入输出参数的写法
"""

import numpy as np
import geatpy as ga # 导入geatpy库
import time

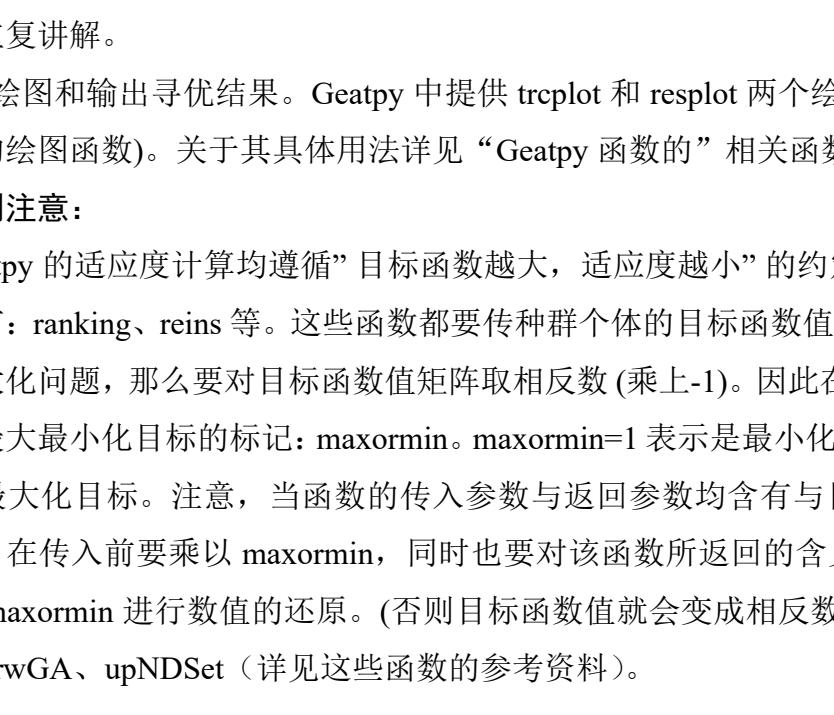
"""=====函数定义=====
# 传入种群基因表现型矩阵Phen以及种群个体的可行性列向量LegV
def aimfuc(Phen, LegV):
    x = np.array([Phen[:, 0]]).T # 取出Phen第一列得到种群所有个体的x值
    y = np.array([Phen[:, 1]]).T # 取出Phen第二列得到种群所有个体的y值
    # 计算MoCormick函数值
    f = np.sin(x + y) + (x - y) ** 2 - 1.5 * x + 2.5 * y + 1
    return [f, LegV]

"""=====变量设置=====
x1 = [-1.5, 4]; x2 = [-3, 4] # 自变量范围
b1 = [1, 1]; b2 = [1, 1] # 自变量边界
codes = [1, 1] # 变量的编码方式，2个变量均使用格雷编码
precisions = [6, 6] # 变量的精度，10表示精确到小数点后10位
scales = [0, 0] # 采用算术刻度
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 = 50 # 种群规模
MAXGEN = 1000 # 最大遗传代数
GGAP = 0.8 # 代沟：子代与父代个体不相同的概率为0.8
selectStyle = 'sus'; # 遗传算法的选择方式设为"sus"——随机抽样选择
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 = 0.1 # 变异概率
SUBPOP = 1 # 设置种群数为1
maxormin = 1 # 设置最大最小化目标标记为1，表示是最小化目标，-1则表示最大化目标

"""=====开始遗传算法进化=====
FieldD = ga.crtfld(ranges,borders,precisions,codes,scales) #
调用函数创建区域描述器
Lind = np.sum(FieldD[0, :]) # 计算编码后的染色体长度
Chrom = ga.crtbp(NIND, Lind) # 根据区域描述器生成二进制种群
Phen = ga.bs2rv(Chrom, FieldD) #对初始种群进行解码
LegV = np.ones((NIND, 1)) # 初始化种群的可行性列向量
[ObjV,LegV] = aimfuc(Phen, LegV) # 计算初始种群个体的目标函数值
# 定义进化记录器，初始值为nan
pop_trace = (np.zeros(MAXGEN, 2)) * np.nan
# 定义种群最优个体记录器，记录每一代最优个体的染色体，初始值为nan
ind_trace = (np.zeros(MAXGEN, Lind)) * np.nan
# 开始进化！
start_time = time.time() # 开始计时
for gen in range(MAXGEN):
    FitnV = ga.ranking(maxormin * ObjV, LegV) #
        根据目标函数大小分配适应度值
    SelCh=ga.selecting(selectStyle, Chrom, FitnV, GGAP, SUBPOP) # 选择
    SelCh=ga.recombin(recombinStyle, SelCh, recopt, SUBPOP) #交叉
    SelCh=ga.mutbin(SelCh, pm) # 二进制种群变异
    Phen = ga.bs2rv(SelCh, FieldD) # 对育种种群进行解码(二进制转十进制)
    LegVSel=np.ones((SelCh.shape[0], 1))#初始化育种种群的可行性列向量
    [ObjVSel,LegVSel] = aimfuc(Phen, LegVSel) # 求育种种群个体的目标函数值
    [Chrom,ObjV,LegV] =
        ga.reins(Chrom,SelCh,SUBPOP,1,1,maxormin*ObjV,maxormin*ObjVSel
        ,ObjV,ObjVSel,LegV,LegVSel) # 重插入得到新一代种群
    # 记录
    pop_trace[gen,1]=np.sum(ObjV)/ObjV.shape[0]#记录当代种群目标函数均值
    if maxormin == 1:
        best_ind = np.argmin(ObjV) # 计算当代最优个体的序号
    elif maxormin == -1:
        best_ind = np.argmax(ObjV)
    pop_trace[gen, 0]=ObjV[best_ind]#记录当代种群最优个体目标函数值
    ind_trace[gen, :] = Chrom[best_ind, :] #
        记录当代种群最优个体的变量值
# 进化完成
end_time = time.time() # 结束计时
"""=====绘图=====
ga.trcplot(pop_trace,[['最优个体目标函数值','种群的目标函数均值']],['demo_result'])

运行结果如下：
```



若要输出相关数据结果，则可以在上面脚本最下方添加以下代码：

```
"""=====输出结果=====
best_gen = np.argmax(pop_trace[:, 0]) # 计算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：')
# 最优个体记录器存储的是各代种群最优个体的染色体,
# 此处需要解码得到对应的基因表现型
variables = ga.bs2rv(ind_trace, FieldD) # 解码
for i in range(variables.shape[1]):
    print(variables[best_gen, i])
print('最优的一代是第',best_gen + 1,'代')
print('用时：', end_time - start_time, '秒')
```

对应的输出结果为：

最优的目标函数值为： -1.9132201779460867

最优的控制变量值为：

-0.5486213026787403

最优的一代是第 608 代

用时： 2.4095842838287354 秒

进化过程中各代种群最优个体的目标函数值和均值如下图：

寻优结果与理论值非常接近。

脚本详细解析：

编写 Geatpy 脚本风格与 Matlab 的 gatbx 以及 GEATbx 相近，对于新的问题，你可以复制上面的脚本，结合实际问题修改函数定义和变量设置，以及绘图和输出便可以对新问题进行求解。实际上你会发现脚本中大部分基本上可以固定不变的（前提是多次使用时采用的基本模型没变），因此在后面的章节中，我们会讲解如何用进化算法模板来代替编写脚本，这样会大大节省开发时间。

下面详细讲解编写类似脚本的流程，你需要做以下步骤：

1. 导入相关库，如 numpy、Geatpy 等。
2. 定义了优化目标函数。（在后面的章节中，我们会详细介绍如何用函数接口来取代这种把函数定义写在脚本内部的方法）。
3. 定义了优化问题的变量约束条件。其中 ranges、borders、codes、precisions、scales 是根据实际问题而编写的，但写法格式是固定的。假如优化函数中包含 3 个变量，那么我们可以这么写：

```
x1=[下界,上界]
x2=[下界,上界]
x3=[下界,上界]
b1=[0,1]#(这里0 表示变量x1 不包含下界, 1 表示包含上界)
b2=[1,0]
b2=[1,1]
ranges=np.vstack([x1,x2,x3]).T #根据x1,x2,x3 生成一个矩阵来表示所有变量的范
围
borders=np.vstack([b1,b2,b3]).T
...
```

4. 对遗传算法的运行参数进行设置，比如确定种群规模（即种群包含多少个体）、是否包含子种群（SUBPOP 设为 1 表示种群只包含 1 个种群，没有划分子种群）、最大遗传代数、代沟、重组及变异概率等。
5. 创建用于描述种群特征的“区域描述器”。我们可以发现，在 Matlab 的 gatbx 和 GEATbx 工具箱中，区域描述器是需要手动创建的，而且格式相当复杂。在 Geatpy 中，我们采用 crtfld 函数来自动化创建（当然你也可以手动创建）。crtfld 函数会根据变量的边界和精度自动地调整变量的范围，返回一个符合规范的区域描述器。
6. 根据创建的区域描述器计算染色体的长度。FieldD 区域描述器的结构参见“Geatpy 函数”章节的 crtfld 函数解析，其中讲解了区域描述器 FieldD 的第一行是代表控制各个变量的染色体基因数目。因此对它求和便可算出染色体的长度。
7. 创建进化记录器，它的写法也是固定的，2 表示该进化记录器有 2 列，每一列存放不同的数据。本例中，它的第一列是存放各代种群最优个体的目标函数值；第二列是存放各代种群的平均目标函数值。
8. 创建最优个体记录器，记录各代种群的最优个体的染色体。
9. 做完这些基本步骤后，后面都是遗传算法的基本流程了。注释中有详细解释，这里不再重复讲解。
10. 绘图和输出寻优结果。Geatpy 中提供 trcplot 和 resplot 两个绘图函数（后续会添加更多的绘图函数）。关于其具体用法详见“Geatpy 函数”的相关函数解析文档。

特别注意：

Geatpy 的适应度计算均遵循“目标函数越大，适应度越小”的约定，与之密切相关的函数有：ranking、reins 等。这些函数都要传种群个体的目标函数值，此时若研究的问题是最大化问题，那么要对目标函数值矩阵取相反数（乘上-1）。因此在 Geatpy 中常常设置一个最大最小化目标的标记：maxormin。maxormin=1 表示是最小化目标；maxormin=-1 表示是最大化目标。注意，当函数的传入参数与返回参数均含有与目标函数值相关的变量时，在传入前要乘以 maxormin，同时也要对该函数所返回的含义等价或相似的变量乘上 maxormin 进行数值的还原。（否则目标函数值就会变成相反数了）。这些函数有：awGA、rwGA、upNDSet（详见这些函数的参考资料）。

### 3. 总结

至此，你已经可以模仿着上面的脚本用遗传算法求解一些简单的优化问题了。在后面的章节我们会讲解如何使用更简单、灵活的进化算法模板和函数接口来快速编程以及融合到你正在进行的其他 Python 项目中。

## Geatpy 数据结构

Geatpy 的大部分数据都是存储在 numpy 的 array 数组里的, numpy 中另外还有 matrix 的矩阵类型, 但我们不适用它, 于是我们默认 array 就是存储“矩阵”(也可以存储一维向量, 接下来会谈到)。其中有一些细节需要特别注意: numpy 的 array 在表示行向量时会有 2 种不同的结构, 一种是 1 行 n 列的矩阵, 它是二维的; 一种是纯粹的一维行向量。因此, 在 Geatpy 教程中会严格区分这两种概念, 我们称前者为“行矩阵”, 后者为“行向量”。Geatpy 中不会使用超过二维的 array。

例如有一个行向量  $x$ , 其值为 1 2 3 4 5 6, 那么, 用 `print(x.shape)` 输出其规格, 可以得到(6,), 若  $x$  是行矩阵而不是行向量, 那么  $x$  的规格就变成是(1,6)而不再是(6,)。

在 numpy 的 array 类型中, 实际上没有“列向量”的概念。所谓“向量”是指一维的, 但用 numpy 的 array 表示列向量时, 它实际上是二维的, 只不过只有 1 列。我们不纠结于这个细节, 统一仍用“列向量”来称呼这种只有 1 列的矩阵。

在编程中, 如果对 numpy 的 array 感到疑惑, 你可以用“变量.shape”语句来输出其维度信息, 以确定其准确的维度。

### 1. 种群染色体

Geatpy 中, 种群染色体是一个二维矩阵, 简称“种群矩阵”。一般所说的“种群”是特指种群染色体矩阵。

种群矩阵一般用 Chrom 命名, 是一个 numpy 的 array 类型的, 每一行对应一条染色体, 同时也对应着一个个体。染色体的每个元素是染色体上的基因。

我们一般把种群的规模(即种群的个体数)用  $Nind$  命名; 把种群个体的染色体长度用  $Lind$  命名。

$$\text{Chrom} = \begin{pmatrix} g_{1,1} & g_{1,2} & g_{1,3} & \cdots & g_{1,Lind} \\ g_{2,1} & g_{2,2} & g_{2,3} & \cdots & g_{2,Lind} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ g_{Nind,1} & g_{Nind,2} & g_{Nind,3} & \cdots & g_{Nind,Lind} \end{pmatrix}$$

对于多种群, Geatpy 有 2 种设计, 一种是将子种群全部放在一个大的种群内, 称为“基于种群 Chrom 的子种群划分”, 此时所有子种群的染色体编码是一样的; 另一种是采用多个 Chrom 变量来表示多个子种群。此时子种群的染色体编码可以不一样。

对于基于种群染色体矩阵 Chrom 的子种群, 子种群之间必须有相同数量的个体, 并且按照下列方案进行有序排列:

$$\text{Chrom} = \begin{pmatrix} subchrom_1\_ind_1 \\ subchrom_1\_ind_2 \\ \vdots \\ subchrom_1\_ind_n \\ subchrom_2\_ind_1 \\ subchrom_2\_ind_2 \\ \vdots \\ subchrom_2\_ind_n \\ \vdots \\ subchrom_{SUBPOP}\_ind_1 \\ subchrom_{SUBPOP}\_ind_2 \\ \vdots \\ subchrom_{SUBPOP}\_ind_n \end{pmatrix}$$

比如如下种群:

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

假设它要表示 2 个子种群, 那么, 前两个个体(前两行)就是 1 号子种群, 后两个个体(后两行)就是 2 号子种群。种群个体的染色体为: 1234, 2341, 3142 和 4231。

### 2. 种群表现型

种群表现型的数据结构跟种群染色体基本一致, 也是 numpy 的 array 类型。我们一般用 Phen 来命名。它是种群矩阵 Chrom 经过解码操作后得到的基因表现型矩阵, 每一行对应一个个体, 每行中每个元素都代表着一个变量, 并用  $Nvar$  表示变量的个数。如下图:

$$\text{Phen} = \begin{pmatrix} x_{1,1} & x_{1,2} & x_{1,3} & \cdots & x_{1,Nvar} \\ x_{2,1} & x_{2,2} & x_{2,3} & \cdots & x_{2,Nvar} \\ x_{3,1} & x_{3,2} & x_{3,3} & \cdots & x_{3,Nvar} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{Nind,1} & x_{Nind,2} & x_{Nind,3} & \cdots & x_{Nind,Nvar} \end{pmatrix}$$

Phen 的值与采用的解码方式有关。Geatpy 提供二进制/格雷码编码转十进制整数或实数的解码方式。另外, 在 Geatpy 也可以使用不需要解码的“实值编码”种群, 这种种群的染色体的每个基因就对应变量的实际值, 即 Phen 等价于 Chrom。如上面的例子中, Chrom 是种群的染色体矩阵, 其第一个个体的染色体为 1234, 若该种群是实值种群, 那么这条染色体就代表了 4 个控制变量, 值分别为 1,2,3 和 4。

这里需要注意的是: 我们可以用不同的方式去解码一个种群染色体, 得到的结果往往是不同的。

对于混合编码, 可以使用多种群去表现不同的编码, 运算效率实际上会远高于把多种编码方式融合在同一个种群中。因此我们一般采用多种群去应对复杂的混合编码。

### 3. 目标函数值

Geatpy 采用 numpy 的 array 类型变量来存储种群的目标函数值。一般命名为  $ObjV$ , 每一行对应种群矩阵的每一个个体。因此它拥有与 Chrom 相同的行数。每一列代表一个目标函数值。因此对于单目标函数,  $ObjV$  会只有 1 列; 而对于多目标函数,  $ObjV$  会有多列。

例如  $ObjV$  是一个二元函数值矩阵:

$$\text{ObjV} = \begin{pmatrix} f_1(x_{1,1}, x_{1,2}, \dots, x_{1,Nvar}), f_2(x_{1,1}, x_{1,2}, \dots, x_{1,Nvar}) \\ f_1(x_{2,1}, x_{2,2}, \dots, x_{2,Nvar}), f_2(x_{2,1}, x_{2,2}, \dots, x_{2,Nvar}) \\ f_1(x_{3,1}, x_{3,2}, \dots, x_{3,Nvar}), f_2(x_{3,1}, x_{3,2}, \dots, x_{3,Nvar}) \\ \vdots \\ f_1(x_{Nind,1}, x_{Nind,2}, \dots, x_{Nind,Nvar}), f_2(x_{Nind,1}, x_{Nind,2}, \dots, x_{Nind,Nvar}) \end{pmatrix}$$

其中,  $lens$  包含染色体的每个子染色体的长度。 $\sum(lens)$  等于染色体长度。 $lb$  和  $ub$  分别代表每个变量的上界和下界。

$codes$  指明染色体串用的是标准二进制编码还是格雷编码。 $codes[i] = 0$  表示第  $i$  个变量使用的是标准二进制编码;  $codes[i] = 1$  表示使用格雷编码。

$scales$  指明每个子串用的是算术刻度还是对数刻度。 $scales[i] = 0$  为算术刻度,  $scales[i] = 1$  为对数刻度。对数刻度可以用于变量的范围较大而且不确定的情况, 对于大范围的参数边界, 对数刻度让搜索可用较少的位数, 从而减少了遗传算法的计算量。

$lbin$  和  $ubin$  指明了变量是否包含其范围的边界。0 表示不包含边界; 1 表示包含边界。

2) 对于实值编码(即前面所说的不需要解码的编码方式)的种群, 使用 2 行  $n$  列的矩阵 FieldDR 来作为区域描述器,  $n$  是染色体所表达的控制变量个数。FieldDR 的结构如下:

$$\text{FieldDR} = \begin{pmatrix} x_1 \text{ 下界} & \dots & x_n \text{ 下界} \\ x_1 \text{ 上界} & \dots & x_n \text{ 上界} \end{pmatrix}$$

也可以用 Geatpy 内置的 crtfd 函数来方便地快速生成区域描述器, 其详细用法参见“Geatpy 函数”的“crtfd 参考资料”。

### 4. 个体适应度

Geatpy 采用列向量来存储种群个体适应度。一般命名为  $FitnV$ , 它同样是 numpy 的 array 类型, 每一行对应种群矩阵的每一个个体。因此它拥有与 Chrom 相同的行数。

$$\text{FitnV} = \begin{pmatrix} fit_1 \\ fit_2 \\ fit_3 \\ \vdots \\ fit_{Nind} \end{pmatrix}$$

其中  $lens$  包含染色体的每个子染色体的长度。 $\sum(lens)$  等于染色体长度。

$lb$  和  $ub$  分别代表每个变量的上界和下界。

$codes$  指明染色体串用的是标准二进制编码还是格雷编码。 $codes[i] = 0$  表示第  $i$  个变量使用的是标准二进制编码;  $codes[i] = 1$  表示使用格雷编码。

$scales$  指明每个子串用的是算术刻度还是对数刻度。 $scales[i] = 0$  为算术刻度,  $scales[i] = 1$  为对数刻度。对数刻度可以用于变量的范围较大而且不确定的情况, 对于大范围的参数边界, 对数刻度让搜索可用较少的位数, 从而减少了遗传算法的计算量。

$lbin$  和  $ubin$  指明了变量是否包含其范围的边界。0 表示不包含边界; 1 表示包含边界。

3) 对于实值编码(即前面所说的不需要解码的编码方式)的种群, 使用 2 行  $n$  列的矩阵 FieldDR 来作为区域描述器,  $n$  是染色体所表达的控制变量个数。FieldDR 的结构如下:

$$\text{FieldDR} = \begin{pmatrix} lens \\ lb \\ ub \\ codes \\ scales \\ lbin \\ ubin \end{pmatrix}$$

其中,  $lens$  包含染色体的每个子染色体的长度。 $\sum(lens)$  等于染色体长度。

$lb$  和  $ub$  分别代表每个变量的上界和下界。

$codes$  指明染色体串用的是标准二进制编码还是格雷编码。 $codes[i] = 0$  表示第  $i$  个变量使用的是标准二进制编码;  $codes[i] = 1$  表示使用格雷编码。

$scales$  指明每个子串用的是算术刻度还是对数刻度。 $scales[i] = 0$  为算术刻度,  $scales[i] = 1$  为对数刻度。对数刻度可以用于变量的范围较大而且不确定的情况, 对于大范围的参数边界, 对数刻度让搜索可用较少的位数, 从而减少了遗传算法的计算量。

$lbin$  和  $ubin$  指明了变量是否包含其范围的边界。0 表示不包含边界; 1 表示包含边界。

4) 对于实值编码(即前面所说的不需要解码的编码方式)的种群, 使用 2 行  $n$  列的矩阵 FieldDR 来作为区域描述器,  $n$  是染色体所表达的控制变量个数。FieldDR 的结构如下:

$$\text{FieldDR} = \begin{pmatrix} x_1 \text{ 下界} & \dots & x_n \text{ 下界} \\ x_1 \text{ 上界} & \dots & x_n \text{ 上界} \end{pmatrix}$$

也可以用 Geatpy 内置的 crtfd 函数来方便地快速生成区域描述器, 其详细用法参见“Geatpy 函数”的“crtfd 参考资料”。

### 5. 个体可行性

Geatpy 采用列向量来存储种群个体可行性。一般命名为  $LegV$ , 它同样是 numpy 的 array 类型, 每一行对应种群矩阵的每一个个体。因此它拥有与 Chrom 相同的行数。

$$\text{LegV} = \begin{pmatrix} Legal_1 \\ Legal_2 \\ Legal_3 \\ \vdots \\ Legal_{Nind} \end{pmatrix}$$

其中  $lens$  包含染色体的每个子染色体的长度。 $\sum(lens)$  等于染色体长度。

$lb$  和  $ub$  分别代表每个变量的上界和下界。

$codes$  指明染色体串用的是标准二进制编码还是格雷编码。 $codes[i] = 0$  表示第  $i$  个变量使用的是标准二进制编码;  $codes[i] = 1$  表示使用格雷编码。

$scales$  指明每个子串用的是算术刻度还是对数刻度。 $scales[i] = 0$  为算术刻度,  $scales[i] = 1$  为对数刻度。对数刻度可以用于变量的范围较大而且不确定的情况, 对于大范围的参数边界, 对数刻度让搜索可用较少的位数, 从而减少了遗传算法的计算量。

$lbin$  和  $ubin$  指明了变量是否包含其范围的边界。0 表示不包含边界; 1 表示包含边界。

5) 对于实值编码(即前面所说的不需要解码的编码方式)的种群, 使用 2 行  $n$  列的矩阵 FieldDR 来作为区域描述器,  $n$  是染色体所表达的控制变量个数。FieldDR 的结构如下:

$$\text{FieldDR} = \begin{pmatrix} x_1 \text{ 下界} & \dots & x_n \text{ 下界} \\ x_1 \text{ 上界} & \dots & x_n \text{ 上界} \end{pmatrix}$$

也可以用 Geatpy 内置的 crtfd 函数来方便地快速生成区域描述器, 其详细用法参见“Geatpy 函数”的“crtfd 参考资料”。

### 6. 区域描述器

Geatpy 使用区域描述器来描述种群染色体的特征, 比如染色体中基因所表达的控制变量的范围、是否包含范围的边界、采用什么编码方式, 是否使用对数刻度等等。

1) 对于二进制/格雷编码的种群, 使用 7 行  $n$  列的矩阵 FieldD 来作为区域描述器,  $n$  是染色体所表达的控制变量个数。FieldD 的结构如下:

$$\text{FieldD} = \begin{pmatrix} lens \\ lb \\ ub \\ codes \\ scales \\ lbin \\ ubin \end{pmatrix}$$

其中,  $lens$  包含染色体的每个子染色体的长度。 $\sum(lens)$  等于染色体长度。

$lb$  和  $ub$  分别代表每个变量的上界和下界。

$codes$  指明染色体串用的是标准二进制编码还是格雷编码。 $codes[i] = 0$  表示第  $i$  个变量使用的是标准二进制编码;  $codes[i] = 1$  表示使用格雷编码。

$scales$  指明每个子串用的是算术刻度还是对数刻度。 $scales[i] = 0$  为算术刻度,  $scales[i] = 1$  为对数刻度。对数刻度可以用于变量的范围较大而且不确定的情况, 对于大范围的参数边界, 对数刻度让搜索可用较少的位数, 从而减少了遗传算法的计算量。

$lbin$  和  $ubin$  指明了变量是否包含其范围的边界。0 表示不包含边界; 1 表示包含边界。

2) 对于实值编码(即前面所说的不需要解码的编码方式)的种群, 使用 2 行  $n$  列的矩阵 FieldDR 来作为区域描述器,  $n$  是染色体所表达的控制变量个数。FieldDR 的结构如下:

$$\text{FieldDR} = \begin{pmatrix} x_1 \text{ 下界} & \dots & x_n \text{ 下界} \\ x_1 \text{ 上界} & \dots & x_n \text{ 上界} \end{pmatrix}$$

也可以用 Geatpy 内置的 crtfd 函数来方便地快速生成区域描述器, 其详细用法参见“Geatpy 函数”的“crtfd 参考资料”。

### 7. 进化追踪器

在使用 Geatpy 进行进化算法编程时, 常常建立一个进化追踪器(如 `pop_trace`)来记录种群在进化的过程中各代的最优个体, 尤其是采用无精英保留机制时, 进化追踪器帮助我们记录种群在进化的“历史长河”中产生过的最优个体。待进化完成后, 再从进化追踪器中挑选出“历史最优”的个体。这种进化记录器也是 numpy 的 array 类型, 结构如下:

$$\text{trace} = \begin{pmatrix} a_1 & b_1 & c_1 & \cdots & \omega_1 \\ a_2 & b_2 & c_2 & \cdots & \omega_2 \\ a_3 & b_3 & c_3 & \cdots & \omega_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{MAXGEN} & b_{MAXGEN} & c_{MAXGEN} & \cdots & \omega_{MAXGEN} \end{pmatrix}$$

其中  $MAXGEN$  是种群进化的代数。 $trace$  的每一列代表不同的指标, 比如第一列记录各代种群的最佳目标函数值, 第二列记录各代种群的平均目标函数值…… $trace$  的每一行对应每一代, 如第一行代表第一代, 第二行代表第二代……

### 8. 全局最优集

&lt;

## 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  $Phen$  以及种群可行性列向量  $LegV$ ，计算各个个体的目标函数值，若有约束条件，此时将对不满足约束条件的个体在  $LegV$  上对应的值设置为 0，最后返回种群的目标函数列向量  $ObjV$  以及修改后的种群可行性列向量  $LegV$ ；罚函数传入种群可行性列向量  $LegV$  和适应度值列向量  $FitnV$ ，此时对于  $LegV$  中值为 0 的个体（即非可行解个体）“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借用命名，你可以修改其命名。）

在 Geatpy 有两种方式对非可行解的个体进行惩罚：

1) 在目标函数 `aimfuc` 中，当找到非可行解个体时，当即对该个体的目标函数值进行惩罚。若为最小化目标，则惩罚时设置一个很大的目标函数值；反之设置一个很小的目标函数值。

2) 在目标函数 `aimfuc` 中，当找到非可行解个体时，并不当即对该个体的目标函数值进行惩罚，而是修改其  $LegV$  上对应位置的值为 0，同时编写罚函数 `punishing`，对  $LegV$  为 0 的个体加以惩罚——降低其适应度。事实上，在 Geatpy 内置的算法模板中，已经对  $LegV$  为 0 的个体的适应度加以一定的惩罚，因此若是使用内置模板，则不需要编写罚函数 `punishing`。此时若仍要编写罚函数 `punishing` 的话，起到的是辅助性的适应度惩罚。

这里的“罚函数 `punishing`”跟数学上的“罚函数”含义是不一样的。后者是纯粹的数学公式，而前者是直接使用 Geatpy 时自定义的一个名为“`punishing`”的根据可行性列向量  $LegV$  来对非可行解的适应度  $FitnV$  进行惩罚的一个函数。）

特别注意：如果采取上面的方法 1 对非可行解进行惩罚，在修改非可行解的目标函数值时，必须设置一个“极大或极小”的值，即当为最小化目标时，要给非可行解设置一个绝对地比所有可行解还要大的值；反之要设置一个绝对比所有可行解小的值。否则容易出现“被欺骗”的现象：即某一代的所有个体全是不可行解，而此时因为惩罚力度不足，在后续的进化中，再也没有可行解比该非可行解修改后的目标函数值要优秀（即更大或更小）。此时就会让进化算法“被欺骗”，得出一个非可行解的“最优”搜索结果。因此，在使用方法 1 的同时，可以同时对  $LegV$  加以标记为 0，两种方法结合着对非可行解进行惩罚。

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。

注意：若使用 Geatpy 的内置算法模板，目标函数和罚函数必须按照下述的输入输出格式进行定义。

```
"""目标函数aimfuc.py"""
import numpy as np
def aimfuc(Phen, LegV): # 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中取得到x1变量
    x2 = Phen[:, [1]]
    ObjV = x1*x1 + x2*x2
    exIdx = np.where(Phen == 0)[0] # 得到非可行解在种群中的位置
    # 采用方法2对非可行解进行标记，在punishing中对其进行惩罚
    LegV[exIdx] = 0 # 标记非可行解为0
    return [ObjV,LegV]
```

传入的  $Phen$  代表种群染色体的表现型，它是一个矩阵， $LegV$  代表种群个体的可行性列向量（其中元素值为 0 表示对应个体是非可行解，为 1 表示对应的是可行解）。注意返回的是  $ObjV$  和  $LegV$  两个参数，前者是一个 numpy 的 array 类型列向量，每一列对应一个个体的目标函数值。

假设传入 `aimfuc` 函数的  $Phen$  的值如下：

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

$LegV$  的值如下：

$$LegV = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

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

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

新的  $LegV$  为：

$$LegV = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

进一步地理解 numpy 的 array 类型变量的维度，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  $LegV$  标记了 0 的变量进行惩罚，使其适应度比当前种群最小适应度值还要小至少 50%：

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

下面是完整执行上述例子的代码：

```
"""test.py"""
import numpy as np
from punishing import punishing
from aimfuc import aimfuc
import geatpy as ga
# 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]])
LegV = np.ones((3, 1)) # 初始化种群可行性列向量
[ObjV, LegV] = aimfuc(Phen, LegV)
```

# 调用ranking计算适应度(因为若给ranking传入LegV参数时，ranking函数会自动对LegV标记为0的非可行解进行惩罚，这里为了展示punishing的作用，因此调用# ranking时不传入LegV)
FitnV=ga.ranking(ObjV)
FitnV=punishing(LegV,FitnV) # 得到新的适应度以及非可行解的下标
print(FitnV) # 输出结果

输出结果为：

$$FitnV = \begin{pmatrix} 2 \\ 0 \\ 0 \end{pmatrix}$$

若不调用 `punishing`，将会得到：

$$FitnV = \begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix}$$

可见种群第二个个体成功地被“惩罚”，其适应度变成了 0。

注意事项：

在采用方法 1 来惩罚非可行解时，要注意一个很容易出错的地方，请看下面的例子：

$$\min f_1(x) = -25(x_1 - 2)^2 - (x_2 - 2)^2 - (x_3 - 1)^2 - (x_4 - 5)^2 - (x_5 - 1)^2$$
$$\min f_2(x) = (x_1 - 1)^2 + (x_2 - 1)^2 + (x_3 - 1)^2 + (x_4 - 1)^2 + (x_5 - 1)^2$$

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：

$$f_i = [p_1 f_1(x), p_2 f_2(x)]$$

其中，若满足约束条件  $g_i(x)(i = 1, 2, \dots, 6)$ ，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  $p_1$ ，那么此时稍有不慎就会极有可能出现目标函数值多次被修改的情况，因为不满足各个约束条件的非可行解可能会有交集，而  $p_1 < 0$ ，所以此时意味着这些目标函数值会被反复变换正负号，这将导致罚函数反而将目标函数值变得更大的错误（“惩罚出错”）。对此，一定不能依次地对不满足各个约束条件的非可行解进行惩罚，而是要取不满足各个约束条件的非可行解的全集，对这个全集中的非可行解进行惩罚，这样能避免上述问题的“惩罚出错”的情况出现。另一种解决办法是像上文所说的，让非可行解设置一个绝对比所有可行解都要大的值。这个需要数学上证明才可去像这样设值。这样也能避免“惩罚出错”的情况出现。

因此，更建议利用惩罚方法 2 来对可行解进行惩罚。

### 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。下面我们将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  $[-5, 0] \cup (0, 5]$  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。

```
"""自定义进化算法模板mintemp1.py"""
import numpy as np
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
    LegV = np.ones((NIND, 1)) # 生成可行性列向量，元素为1表示对应个体是可行解，0表示非可行解
    [ObjV, LegV] = aimfuc(Chrom, LegV)
```

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。

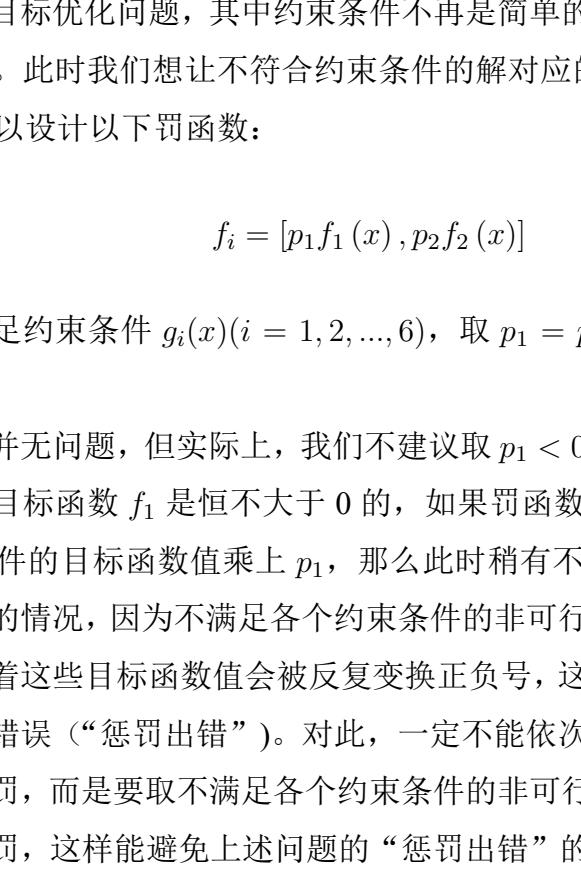
6) SUBPOP：种群中包含的子种群数量。要求 `SUBPOP` 必须能被 `NIND` 整除。

7) GGAP：进化沟谷，即子代种群与父代种群个体数不相同的概率。

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

9) recombinStyle：最小化最大化标记，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：



输出结果为：

$$FitnV = \begin{pmatrix} 2 \\ 0 \\ 0 \end{pmatrix}$$

若不调用 `punishing`，将会得到：

$$FitnV = \begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix}$$

可见种群第二个个体成功地被“惩罚”，其适应度变成了 0。

注意事项：

在采用方法 1 来惩罚非可行解时，要注意一个很容易出错的地方，请看下面的例子：

$$\min f_1(x) = -25(x_1 - 2)^2 - (x_2 - 2)^2 - (x_3 - 1)^2 - (x_4 - 5)^2 - (x_5 - 1)^2$$
$$\min f_2(x) = (x_1 - 1)^2 + (x_2 - 1)^2 + (x_3 - 1)^2 + (x_4 - 1)^2 + (x_5 - 1)^2$$

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：

$$f_i = [p_1 f_1(x), p_2 f_2(x)]$$

其中，若满足约束条件  $g_i(x)(i = 1, 2, \dots, 6)$ ，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  $p_1$ ，那么此时稍有不慎就会极有可能出现目标函数值多次被修改的情况，因为不满足各个约束条件的非可行解可能会有交集，而  $p_1 < 0$ ，所以此时意味着这些目标函数值会被反复变换正负号，这将导致罚函数反而将目标函数值变得更大的错误（“惩罚出错”）。对此，一定不能依次地对不满足各个约束条件的非可行解进行惩罚，而是要取不满足各个约束条件的非可行解的全集，对这个全集中的非可行解进行惩罚，这样能避免上述问题的“惩罚出错”的情况出现。另一种解决办法是像上文所说的，让非可行解设置一个绝对比所有可行解都要大的值。这个需要数学上证明才可去像这样设值。这样也能避免“惩罚出错”的情况出现。

因此，更建议利用惩罚方法 2 来对可行解进行惩罚。

### 3. 总结

因此，在 Geatpy 中，你可以像“快速入门”章节中展示的用纯脚本的方式来编写遗传算法程序，也可以用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 与你的其他项目相融合了。

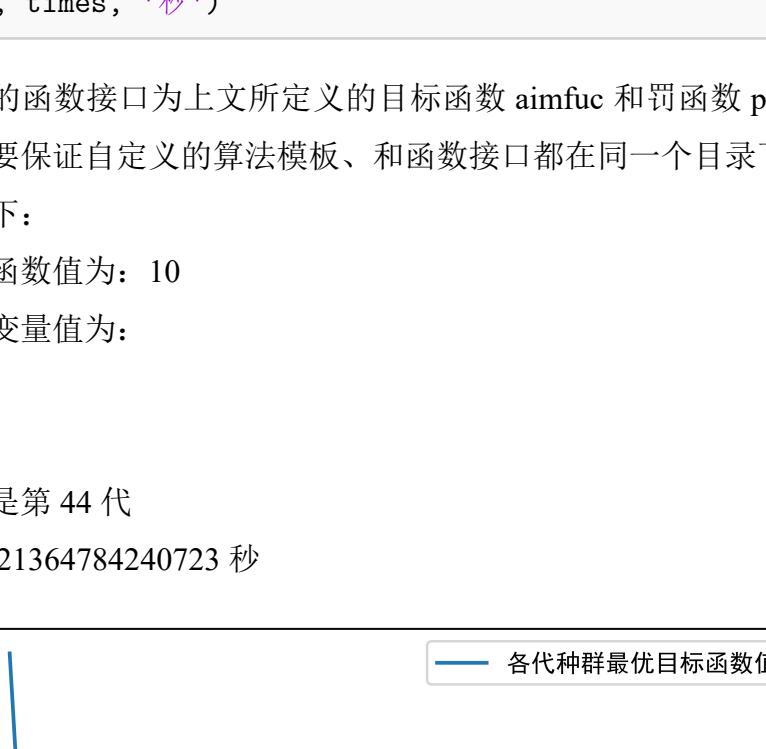
总结一下，使用进化算法模板解决遗传算法问题，你需要做 3 件事：

1) 编写程序来执行脚本 `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` 放在同一目录下（以便 `main.py` 能找到它）。

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。



## 更多进化算法模板

本章介绍 Geatpy 内置的几个进化算法模板的功能，详细的代码可以阅读相应的源码，源码中有包括程序逻辑、输入输出的解析等非常详尽的注释。

这些模板内置在 Geatpy 中，因此不需要专门找到并导入。只需统一导入 Geatpy 库（如 `import geatpy as ga`），便可以通过（`ga` 函数名）来直接调用。

你也可以把“进化算法模板”称作是“内置函数”。但封装成函数只是其存在形式，它们更多的是提供了一些利用进化算法解决问题实际问题的通用模板。你可以参照模板来实现自己的“模板”，比如一些改进型的遗传算法。

值得注意的是，为保持模板的简洁，内置模板并不进行对相关数据的合法性检查，Geatpy 核心函数会发现并捕获这些异常，并提供相关的错误信息。

Geatpy 中有以下几个内置的进化算法模板：

- `sga_real_templ`(单目标进化算法模板(实值编码))
- `sga_code_templ`(单目标进化算法模板(二进制/格雷编码))
- `sga_permut_templ`(单目标进化算法模板(排列编码))
- `sga_new_real_templ`(改进的单目标进化算法模板(实值编码))
- `sga_new_code_templ`(改进的单目标进化算法模板(二进制/格雷编码))
- `sga_new_permut_templ`(改进的单目标进化算法模板(排列编码))
- `sga_mpc_real_templ`(基于多种群竞争进化单目标编程模板(实值编码))
- `sga_mps_real_templ`(基于多种群独立进化单目标编程模板(实值编码))
- `moca_awGA_templ`(基于适应性权重法(awGA)的多目标优化进化算法模板)
- `moca_rwGA_templ`(基于随机权重法(rwGA)的多目标优化进化算法模板)
- `moca_nsga2_templ`(基于改进 NSGA-II 算法的多目标优化进化算法模板)
- `moca_q_sorted_new_templ`(改进的快速非支配排序法的多目标优化进化算法模板)
- `moca_q_sorted_templ`(基于快速非支配排序法的多目标优化进化算法模板)

下面简要介绍其中几个模板的使用。

### 1. 改进的单目标进化算法模板(二进制/格雷编码)(`sga_new_code_templ`)

功能解释：

该模板是单目标进化算法模板 (`sga_code_templ`) 的改进版，用改进的遗传算法解决控制变量是整数或实数的单目标优化问题(染色体采用二进制或格雷编码)。

该模板没有采用标准的遗传算法流程，而是先进行交叉和变异生成子代，然后父子两代合并，再从合并的种群中择优保留，从而实现了精英保留，使算法收敛速度更快。

下面以一个经典的多元多峰函数单目标优化例子来展示如何调用该进化算法模板来解决问题，为了书写方便我们直接把目标函数写在执行脚本内。

$$\begin{aligned} \max f(x_1, x_2) &= 21.5 + x_1 \sin(4\pi x_1) + x_2 \sin(4\pi x_2) \\ \text{s.t. } &\left\{ \begin{array}{l} -3.0 \leq x_1 \leq 12.1 \\ 4.1 \leq x_2 \leq 5.8 \end{array} \right. \end{aligned}$$

编写如下函数接口和执行脚本：

```
# -*- coding: utf-8 -*-
"""

aimfc.py - 目标函数demo
描述:
Geatpy的目标函数遵循本案例的定义方法, 传入种群表现型矩阵Phen, 以及可行性向量LegV
若没有约束条件, 也需要返回LegV
若要改变目标函数的输入参数、输出参数的格式, 则需要修改或自定义算法模板
"""

import numpy as np
def aimfuc(Phen, LegV):
    x1 = Phen[:, [0]]
    x2 = Phen[:, [1]]
    f = 21.5 + x1 * np.sin(4 * np.pi * x1) + x2 * np.sin(20 * np.pi * x2)
    return [f, LegV]

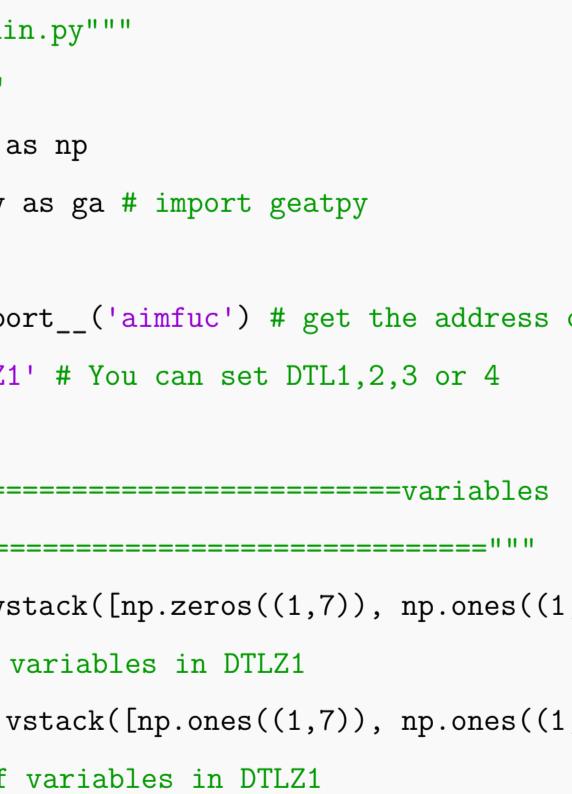
# -*- coding: utf-8 -*-
"""
执行脚本main.py
描述:
该demo是展示如何计算无约束的单目标优化问题
本案例通过调用sga_new_code_templ算法模板来解决该问题
其中目标函数写在aimfuc.py文件中
本案例调用了“sga_new_code_templ”这个算法模板, 其详细用法可利用help命令查看, 或是在github下载并查看源码
调用算法模板时可以设置drawing=2, 此时算法模板将在种群进化过程中绘制动画,
但注意执行前要在Python控制台执行命令matplotlib qt5。
"""

import numpy as np
import geatpy as ga

# 获取函数接口地址
AIM_M = __import__('aimfuc')
# 变量设置
x1 = [-3, 12.1] # 自变量1的范围
x2 = [4.1, 5.8] # 自变量2的范围
b1 = [1, 1] # 自变量1是否包含下界
b2 = [1, 1] # 自变量2是否包含上界
codes = [0, 0] # 自变量的编码方式, 0表示采用标准二进制编码
precisions = [4, 4] #
    自变量的精度(精度不宜设置太高, 否则影响搜索性能和效果)
scales = [0, 0] # 是否采用对数刻度
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
# 生成区域描述器
FieldD = ga.crtfld(ranges, borders, precisions, codes, scales)

# 调用编程模板
[pop_trace, var_trace, times] = ga.sga_new_code_templ(AIM_M,
    'aimfuc', None, None, FieldD, problem = 'R', maxormin = -1, MAXGEN = 1000, NIND = 100, SUBPOP = 1, GGAP = 0.8, selectStyle = 'sus',
    recombinStyle = 'xovdp', recopt = None, pm = None, distribute = True, drawing = 1)
```

运行结果如下：



最优的目标函数值为：38.850292

最优的控制变量值为：

11.625531103252806

5.725031281472213

有效进化代数：1000

最优的一代是第 89 代

时间已过 3.492315 秒

提示：很多相关书籍也有举过这个经典例子，并且很多都说目标函数最大值为 38.81208。实际上，在同样采用精度为 4 的二进制染色体编码下，增大进化代数，算法能搜索出最大的目标函数值为 38.8502924106，当然这也是需要耗费更多的时间。

这也提醒我们，对于一些问题，进化算法并不能搜索出真实的全局最优解，只能无限逼近（当然 38.8502924106 并不是该问题的真实最优解，真实解是一个超越数，我们只能无限逼近它）。

### 2. 单目标进化算法模板(实值编码)(`sga_real_templ`)

功能解释：

该模板用于解决控制变量是整数或实数的单目标优化问题(染色体采用实值编码)。

该模板并没有对种群使用精英保留策略，而是使用一个进化追踪器来记录各代种群的最优个体。关于实现了精英保留的同类模板，可以参考改进的单目标进化算法模板(实值编码)(`sga_new_real_templ`)。

实值编码即染色体不需要进行解码即表达了控制变量的真实值。下面同样以一个经典的整数规划问题来介绍该模板的用法：

$$\begin{aligned} \max z &= 18x_1 + 10x_2 + 12x_3 + 8x_4 \\ \text{s.t. } &\left\{ \begin{array}{l} 12x_1 + 6x_2 + 10x_3 + 4x_4 \\ x_3 + x_4 \leq 1 \\ x_3 \leq x_1 \\ x_4 \leq x_2 \\ x_j \in \{0, 1\}, (j = 1, 2, 3, 4) \end{array} \right. \end{aligned}$$

这个整数规划问题是源于一个公司选址问题。下面展示不把约束条件写在罚函数里而是和目标函数写在一起的示例：

编写目标函数：

```
"""函数接口aimfuc.py"""
import numpy as np
def aimfuc(Phen, LegV):
    x1 = Phen[:, [0]]
    x2 = Phen[:, [1]]
    x3 = Phen[:, [2]]
    x4 = Phen[:, [3]]
    f = 18 * x1 + 10 * x2 + 12 * x3 + 8 * x4
    # 约束条件
    idx1 = np.where(12 * x1 + 6 * x2 + 10 * x3 + 4 * x4 > 20)[0]
    idx2 = np.where(x3 + x4 > 1)[0]
    idx3 = np.where(x3 - x1 > 0)[0]
    idx4 = np.where(x4 - x2 > 0)[0]
    # 采用惩罚方法1
    f[idx1] = -1
    f[idx2] = -1
    f[idx3] = -1
    f[idx4] = -1
    return [f, LegV]

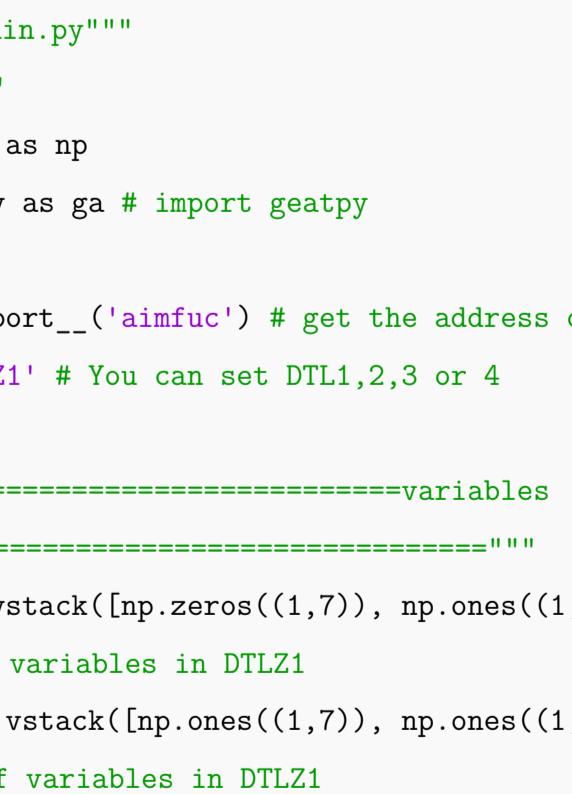
# 执行脚本main.py
描述:
该demo是展示如何计算无约束的单目标优化问题
本案例通过调用sga_new_code_templ算法模板来解决该问题
其中目标函数写在aimfuc.py文件中
本案例调用了“sga_new_code_templ”这个算法模板, 其详细用法可利用help命令查看, 或是在github下载并查看源码
调用算法模板时可以设置drawing=2, 此时算法模板将在种群进化过程中绘制动画,
但注意执行前要在Python控制台执行命令matplotlib qt5。
"""

import numpy as np
import geatpy as ga

# 获取函数接口地址
AIM_M = __import__('aimfuc')
# 变量设置
x1 = [-3, 12.1] # 自变量1的范围
x2 = [4.1, 5.8] # 自变量2的范围
b1 = [1, 1] # 自变量1是否包含下界
b2 = [1, 1] # 自变量2是否包含上界
codes = [0, 0] # 自变量的编码方式, 0表示采用标准二进制编码
precisions = [4, 4] #
    自变量的精度(精度不宜设置太高, 否则影响搜索性能和效果)
scales = [0, 0] # 是否采用对数刻度
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
# 生成区域描述器
FieldD = ga.crtfld(ranges, borders, precisions, codes, scales)

# 调用编程模板
[pop_trace, var_trace, times] = ga.sga_new_code_templ(AIM_M,
    'aimfuc', None, None, FieldD, problem = 'R', maxormin = -1, MAXGEN = 25,
    NIND = 10, SUBPOP = 1, GGAP = 0.9, selectStyle = 'sus',
    recombinStyle = 'xovdp', recopt = 0.9, pm = 0.1, distribute = True, drawing = 1)
```

运行结果如下：



最优的目标函数值为：28.000000

最优的控制变量值为：

1.0

0.0

有效进化代数：25

最优的一代是第 2 代

时间已过 0.043885 秒

提示：很多相关书籍也有举过这个经典例子，并且很多都说目标函数最大值为 38.81208。实际上，在同样采用精度为 4 的二进制染色体编码下，增大进化代数，算法能搜索出最大的目标函数值为 38.8502924106，当然这也是需要耗费更多的时间。

这也提醒我们，对于一些问题，进化算法并不能搜索出真实的全局最优解，只能无限逼近（当然 38.8502924106 并不是该问题的真实最优解，真实解是一个超越数，我们只能无限逼近它）。

### 3. 基于改进 NSGA-II 算法的多目标优化进化算法模板 (moea\_nsga2\_templ)

功能解释：

该模板用于解决控制变量是整数或实数的多目标优化问题(染色体采用实值编码)。

该模板并没有对种群使用精英保留策略，而是使用一个进化追踪器来记录各代种群的最优个体。关于实现了精英保留的同类模板，可以参考改进的单目标进化算法模板(实值编码)(`sga_new_real_templ`)。

实值编码即染色体不需要进行解码即表达了控制变量的真实值。下面同样以一个经典的整数规划问题来介绍该模板的用法：

$$\begin{aligned} \max z &= 18x_1 + 10x_2 + 12x_3 + 8x_4 \\ \text{s.t. } &\left\{ \begin{array}{l} 12x_1 + 6x_2 + 10x_3 + 4x_4 \\ x_3 + x_4 \leq 1 \\ x_3 \leq x_1 \\ x_4 \leq x_2 \\ x_j \in \{0, 1\}, (j = 1, 2, 3, 4) \end{array} \right. \end{aligned}$$

这个整数规划问题是源于一个公司选址问题。下面展示不把约束条件写在罚函数里而是和目标函数写在一起的示例：

编写目标函数：

```
"""函数接口aimfuc.py"""
import numpy as np
def aimfuc(Phen, LegV):
    x1 = Phen[:, [0]]
    x2 = Phen[:, [1]]
    x3 = Phen[:, [2]]
    x4 = Phen[:, [3]]
    f = 18 * x1 + 10 * x2 + 12 * x3 + 8 * x4
    # 约束条件
    idx1 = np.where(12 * x1 + 6 * x2 + 10 * x3 + 4 * x4 > 20)[0]
    idx2 = np.where(x3 + x4 > 1)[0]
    idx3 = np.where(x3 - x1 > 0)[0]
    idx4 = np.where(x4 - x2 > 0)[0]
    # 采用惩罚方法1
    f[idx1] = -1
    f[idx2] = -1
    f[idx3] = -1
    f[idx4] = -1
    return [f, LegV]

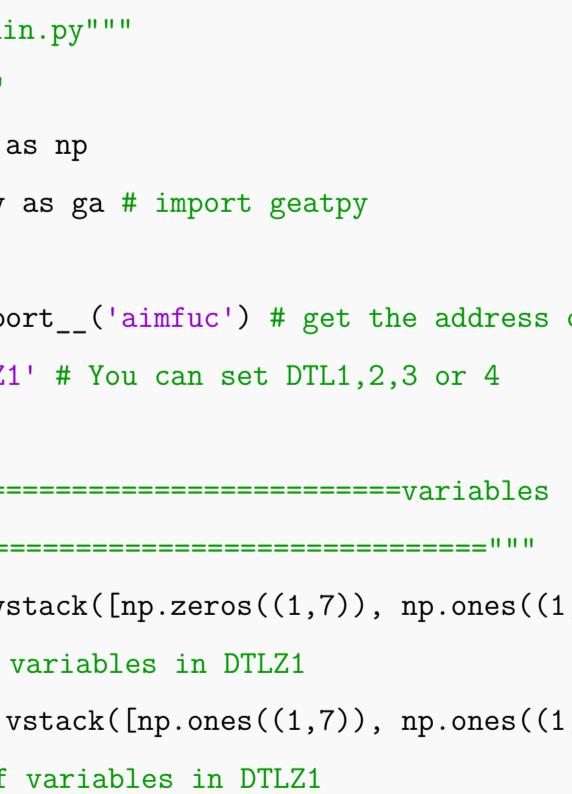
# 执行脚本main.py
描述:
该demo是展示如何计算无约束的单目标优化问题
本案例通过调用sga_new_code_templ算法模板来解决该问题
其中目标函数写在aimfuc.py文件中
本案例调用了“sga_new_code_templ”这个算法模板, 其详细用法可利用help命令查看, 或是在github下载并查看源码
调用算法模板时可以设置drawing=2, 此时算法模板将在种群进化过程中绘制动画,
但注意执行前要在Python控制台执行命令matplotlib qt5。
"""

import numpy as np
import geatpy as ga

# 获取函数接口地址
AIM_M = __import__('aimfuc')
# 变量设置
x1 = [-3, 12.1] # 自变量1的范围
x2 = [4.1, 5.8] # 自变量2的范围
b1 = [1, 1] # 自变量1是否包含下界
b2 = [1, 1] # 自变量2是否包含上界
codes = [0, 0] # 自变量的编码方式, 0表示采用标准二进制编码
precisions = [4, 4] #
    自变量的精度(精度不宜设置太高, 否则影响搜索性能和效果)
scales = [0, 0] # 是否采用对数刻度
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
# 生成区域描述器
FieldD = ga.crtfld(ranges, borders, precisions, codes, scales)

# 调用编程模板
[pop_trace, var_trace, times] = ga.sga_new_code_templ(AIM_M,
    'aimfuc', None, None, FieldD, problem = 'R', maxormin = -1, MAXGEN = 25,
    NIND = 10, SUBPOP = 1, GGAP = 0.9, selectStyle = 'sus',
    recombinStyle = 'xovdp', recopt = 0.9, pm = 0.1, distribute = True, drawing = 1)
```

运行结果如下：



最优的目标函数值为：28.000000

最优的控制变量值为：

1.0

0.0

有效进化代数：25

最优的一代是第 2 代

时间已过 0.043885 秒

### 4. 总结

本章再次回顾了如何使用进化算法模板来极大地提高编程效率。Geatpy 内置进化算法模板是对常见问题的统一建模，最后再次提醒，当需要自定义模板时，请确保定义的模板名称与 Geatpy 内置模板名称不能重复，否则会导致难以解决的冲突问题。

对于单目标和多目标优化问题、排列组合优化问题等，Geatpy 都给出了通用的进化算法模板。在后续版本中 Geatpy 会提供更多的改进的进化算法模板，比如“自适应进化算法”、“基于划分网格的进化算法”等。

```
"""函数接口aimfuc.py"""
import numpy as np
def aimfuc(Phen, LegV):
    x1 = Phen[:, [0]]
    x2 = Phen[:, [1]]
    x3 = Phen[:, [2]]
    x4 = Phen[:, [3]]
    f = 18 * x1 + 10 * x2 + 12 * x3 + 8 * x4
    # 约束条件
    idx1 = np.where(12 * x1 + 6 * x2 + 10 * x3 + 4 * x4 > 20)[0]
    idx2 = np.where(x3 + x4 > 1)[0]
    idx3 = np.where(x3 - x1 > 0)[0]
    idx4 = np.where(x4 - x2 > 0)[0]
    # 采用惩罚方法1
    f[idx1] = -1
    f[idx2] = -1
    f[idx3] = -1
    f[idx4] = -1
    return [f, LegV]

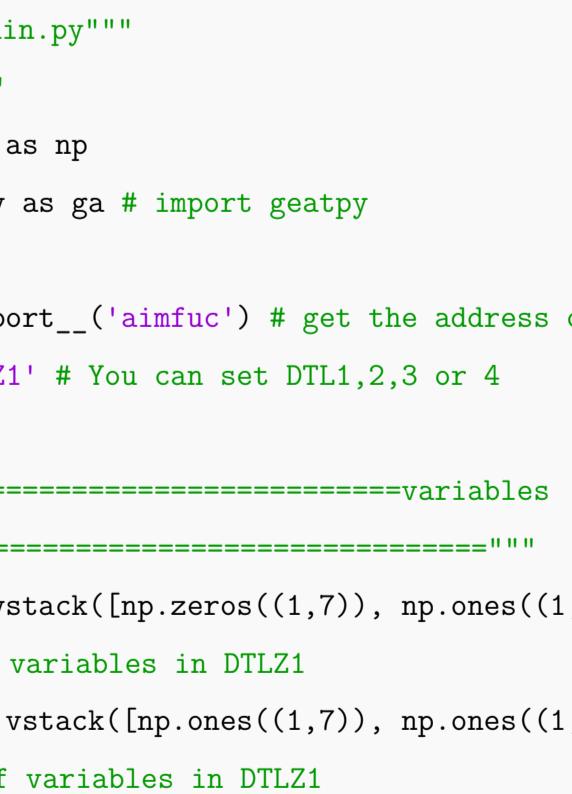
# 执行脚本main.py
描述:
该demo是展示如何计算无约束的单目标优化问题
本案例通过调用sga_new_code_templ算法模板来解决该问题
其中目标函数写在aimfuc.py文件中
本案例调用了“sga_new_code_templ”这个算法模板, 其详细用法可利用help命令查看, 或是在github下载并查看源码
调用算法模板时可以设置drawing=2, 此时算法模板将在种群进化过程中绘制动画,
但注意执行前要在Python控制台执行命令matplotlib qt5。
"""

import numpy as np
import geatpy as ga

# 获取函数接口地址
AIM_M = __import__('aimfuc')
# 变量设置
x1 = [-3, 12.1] # 自变量1的范围
x2 = [4.1, 5.8] # 自变量2的范围
b1 = [1, 1] # 自变量1是否包含下界
b2 = [1, 1] # 自变量2是否包含上界
codes = [0, 0] # 自变量的编码方式, 0表示采用标准二进制编码
precisions = [4, 4] #
    自变量的精度(精度不宜设置太高, 否则影响搜索性能和效果)
scales = [0, 0] # 是否采用对数刻度
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
# 生成区域描述器
FieldD = ga.crtfld(ranges, borders, precisions, codes, scales)

# 调用编程模板
[pop_trace, var_trace, times] = ga.sga_new_code_templ(AIM_M,
    'aimfuc', None, None, FieldD, problem = 'R', maxormin = -1, MAXGEN = 25,
    NIND = 10, SUBPOP = 1, GGAP = 0.9, selectStyle = 'sus',
    recombinStyle = 'xovdp', recopt = 0.9, pm = 0.1, distribute = True, drawing = 1)
```

运行结果如下：



## 数据可视化

本章介绍 Geatpy 的可视化功能。

当前 Geatpy 提供 3 个可视化输出的库函数: treplot, sgplot 和 frontplot。分别是单目标进化算法的可视化输出、单目标优化进化过程动画输出, 以及多目标进化算法的可视化输出。

值得注意的是, 在进行可视化输出之前, 最好在 ipython 控制台执行以下语句:

### matplotlib qt5

使绘制的图形在窗口中显示, 否则动画效果就无法显示。

#### 1. treplot

该函数根据传入的进化追踪器来实现种群进化全过程中一些参数的可视化。比如绘制各代种群最优个体的目标函数值、适应度值、种群个体的解等。其语法如下:

```
treplot(pop_trace, labels)
```

```
treplot(pop_trace, labels, titles)
```

```
treplot(pop_trace, labels, titles, save_path)
```

其中 pop\_trace 是一个进化追踪器, 每一列代表一个参数, 如第一列代表个体最优目标函数值等, 每一行对应一个“代”, 比如第一行对应的是第一代种群的最优个体等等。

因为 pop\_trace 包含许多信息, 我们可以根据这些信息绘制一张或多张图片。因此需要使用 labels 和 titles 来控制需要画多张图、每张图包含多少个变量。

因此 labels 和 titles 都是 list 类型的列表。labels 指代了图片中有哪些变量, 这些变量的图例名称是什么。例如:

1. 假设 pop\_trace 有 2 列, 含义分别是'a' 和'b', 则 labels = [['a'],['b']], 表示要画 2 张图, 每张图画 2 个变量, 图例分别是'a' 和'b'。

2. 假设 pop\_trace 有 2 列, 含义分别是'a' 和'b', 则 labels = [[['a','b']]], 表示要画 1 张图, 图中有 2 个变量, 图例分别是'a' 和'b'。

3. 假设 pop\_trace 有 3 列, 含义分别是'a', 'b' 和 'c', 则 labels = [[['a'], ['b'], 'c']], 表示要画 2 张图, 第一张图有 1 个变量, 图例是'a'; 第二张图有 2 个变量, 图例是'b' 和 'c'。

注: labels 的元素总数必须等于 pop\_trace 的列数

titles 表示各图片的标题, 元素设为空字符串则表示不显示图片标题。

如 titles = ['a'], 表示有 2 张图片, 标题分别是 a 和不设标题

注: len(titles) 必须等于 len(labels)

案例:

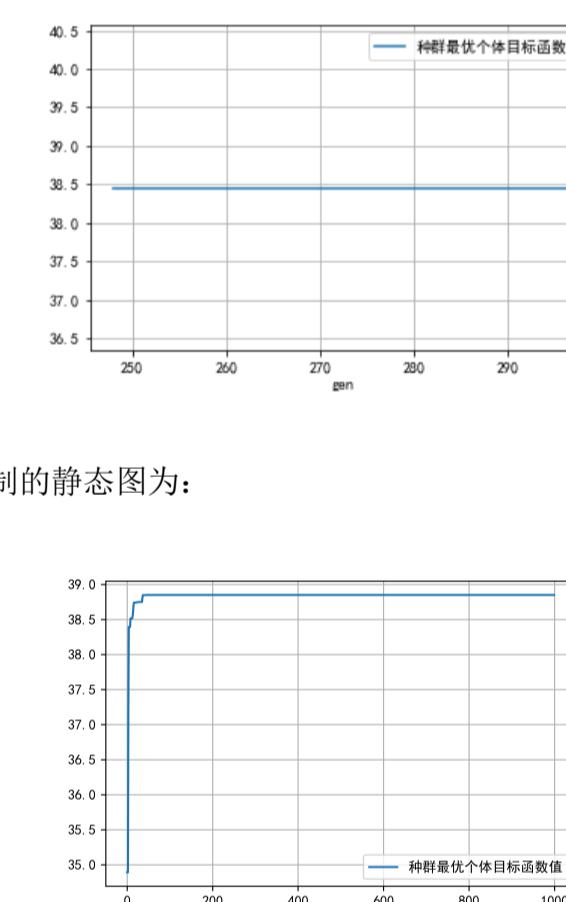
在 sga\_code\_tempalte 算法模板中, 有这样的一行代码:

```
ga.treplot(pop_trace, [['种群个体平均目标函数值',  
    '种群最优个体目标函数值']])
```

其中 pop\_trace 是一个  $n \times 2$  列的矩阵, 第一列代表种群个体平均目标函数值, 第二列代表种群最优个体目标函数值。因此传入 treplot 绘图函数的参数 labels 设为 [['种群个体平均目标函数值', '种群最优个体目标函数值']], 表示要画 1 张图, 这张图中同时绘制“种群个体平均目标函数值”以及“种群最优个体目标函数值”。

titles 参数是缺省的, 因此绘图将不显示标题。

绘图结果如下:



#### 2. sgplot

该函数根据传入的数据集和进化代数 gen 绘制动态图或进化结束后的静态图。其语法如下:

```
newAx = sgplot(ValueSet, Label, saveFlag)
```

```
newAx = sgplot(ValueSet, Label, saveFlag, ax)
```

```
newAx = sgplot(ValueSet, Label, saveFlag, ax, gen)
```

```
newAx = sgplot(ValueSet, Label, saveFlag, ax, gen, interval)
```

```
newAx = sgplot(ValueSet, Label, saveFlag, ax, gen, interval, title)
```

```
newAx = sgplot(ValueSet, Label, saveFlag, ax, gen, interval, title, save_path)
```

其中 ValueSet 是一个 numpy 的 array 类型的列向量, 一般传入该函数前是进化记录器 pop\_trace 的某一列数据。其实际含义由 Label 确定。

Label 是一个字符串, 代表数据集 ValueSet 的含义

saveFlag 是布尔类型的标记, 表示是否要保存图片。当要绘制动画时, 必须设为 False。

ax 是可选参数, 在绘制动画的时候需要传入。其代表上一帧的动画。当画第一帧时, 其值为 None。

gen 是可选参数, 表示当前进化代数, 默认为 None。当该参数没有缺省或为非 None 时, 图片将绘制动态图。

interval 是可选参数, 表示两帧动画之间的间隔时间, 默认为 0.1, 单位为‘秒’。

title 是可选参数, 表示图形的标题名称。

save\_path 是 string 类型的可选参数, 表示保存图片的路径。

newAx 代表新的图形, 是更新后的 ax。

案例:

在解决一个单目标优化问题时绘制进化过程中的各代种群最优个体的目标函数值的变化动态图。并且在进化结束后绘制一个进化全过程的各代最优个体的目标函数值静态图。

```
...  
ax = None # 存储上一帧图片  
# 开始进化  
for gen in range(MAXGEN)  
...  
# 进化操作  
...  
# 记录最优个体  
bestIdx = np.max(FitnV)  
pop_trace[gen, 1] = ObjV[bestIdx] # 记录当代目标函数的最优值  
...  
# 绘图  
ax = ga.sgplot(pop_trace[:, [1]], '种群最优个体目标函数值',  
    False, ax, gen)
```

```
# 进化结束  
ga.sgplot(pop_trace[:, [1]], '种群最优个体目标函数值',  
    True) # 绘制最终的帕累托前沿图  
# end
```

运行结果动画部分截图如下:



进化结束后绘制的静态图为:



该函数根据帕累托最优集目标函数值矩阵 NDSetObjV 绘制 2 维或 3 维的目标函数值散点图。并且可以在种群进化过程中绘制动态的帕累托最优解的 2 维或 3 维动画。

语法:

```
newAx = frontplot(NDSetObjV, saveFlag)
```

```
newAx = frontplot(NDSetObjV, saveFlag, ax)
```

```
newAx = frontplot(NDSetObjV, saveFlag, ax, gen)
```

```
newAx = frontplot(NDSetObjV, saveFlag, ax, gen, interval)
```

```
newAx = frontplot(NDSetObjV, saveFlag, ax, gen, interval, title)
```

```
newAx = frontplot(NDSetObjV, saveFlag, ax, gen, interval, title, save_path)
```

其中 NDSetObjV 为存储帕累托最优解的目标函数值矩阵, 一列对应一个目标函数值, 其列数必须为 2 或 3。

saveFlag 是布尔类型的标记, 表示是否要保存图片。当要绘制动画时, 必须设为 False。

ax 是可选参数, 在绘制动画的时候需要传入。其代表上一帧的动画。当画第一帧时, 其值为 None。

gen 是可选参数, 表示当前进化代数, 默认为 None。当该参数没有缺省或为非 None 时, 图片将绘制动态图。

interval 是可选参数, 表示两帧动画之间的间隔时间, 默认为 0.1, 单位为‘秒’。

title 是可选参数, 表示图形的标题名称。

save\_path 是 string 类型的可选参数, 表示保存图片的路径。

newAx 代表新的图形, 是更新后的 ax。

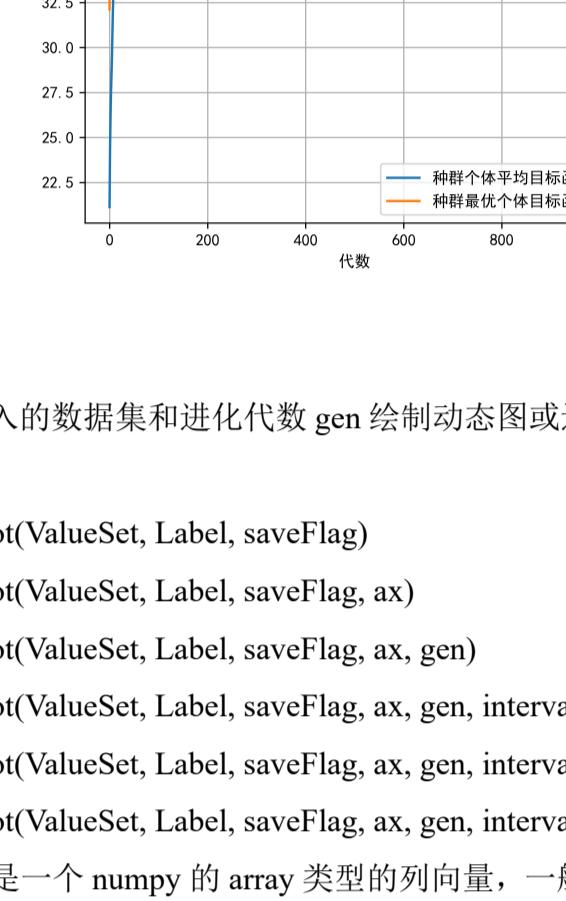
案例:

在解决 3 目标优化问题时绘制进化过程中的帕累托前沿的动画, 以观察搜索到的帕累托前沿的动态变化。并且在进化结束后绘制一个最终的帕累托前沿图。

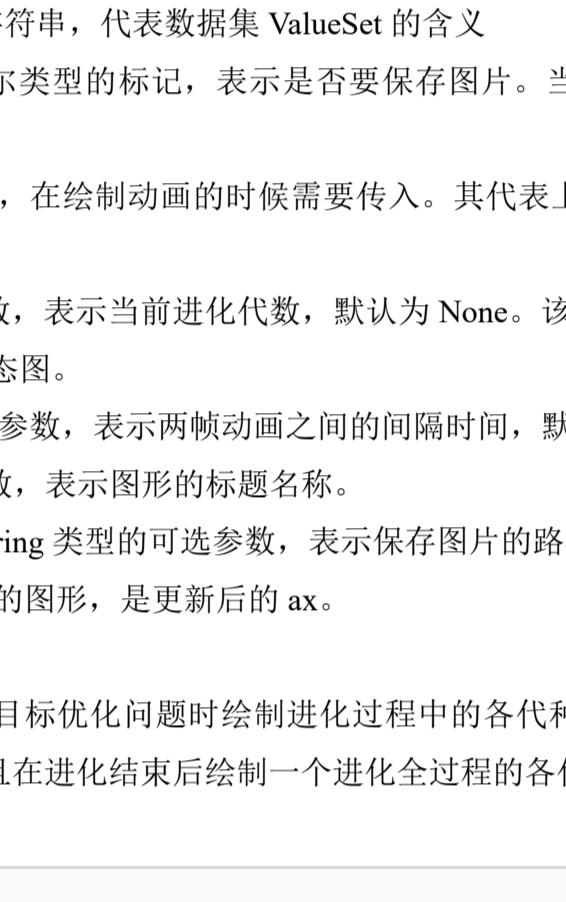
```
...  
ax = None # 存储上一帧图片  
# 开始进化  
for gen in range(MAXGEN)  
...  
# 进化操作  
...  
# 更新帕累托最优集目标函数值矩阵NDSetObjV  
...  
# 绘图  
ax = ga.frontplot(NDSetObjV, False, ax, gen + 1)
```

```
# 进化结束  
ga.frontplot(NDSetObjV, True) # 绘制最终的帕累托前沿图  
# end
```

运行结果动画部分截图如下:



最终的前沿面静态图为:



该函数根据帕累托最优集目标函数值矩阵 NDSetObjV 绘制 2 维或 3 维的目标函数值散点图。并且可以在种群进化过程中绘制动态的帕累托最优解的 2 维或 3 维动画。

语法:

```
newAx = frontplot(NDSetObjV, saveFlag)
```

```
newAx = frontplot(NDSetObjV, saveFlag, ax)
```

```
newAx = frontplot(NDSetObjV, saveFlag, ax, gen)
```

```
newAx = frontplot(NDSetObjV, saveFlag, ax, gen, interval)
```

```
newAx = frontplot(NDSetObjV, saveFlag, ax, gen, interval, title)
```

```
newAx = frontplot(NDSetObjV, saveFlag, ax, gen, interval, title, save_path)
```

其中 NDSetObjV 为存储帕累托最优解的目标函数值矩阵, 一列对应一个目标函数值, 其列数必须为 2 或 3。

saveFlag 是布尔类型的标记, 表示是否要保存图片。当要绘制动画时, 必须设为 False。

ax 是可选参数, 在绘制动画的时候需要传入。其代表上一帧的动画。当画第一帧时, 其值为 None。

gen 是可选参数, 表示当前进化代数, 默认为 None。当该参数没有缺省或为非 None 时, 图片将绘制动态图。

interval 是可选参数, 表示两帧动画之间的间隔时间, 默认为 0.1, 单位为‘秒’。

title 是可选参数, 表示图形的标题名称。

save\_path 是 string 类型的可选参数, 表示保存图片的路径。

newAx 代表新的图形, 是更新后的 ax。

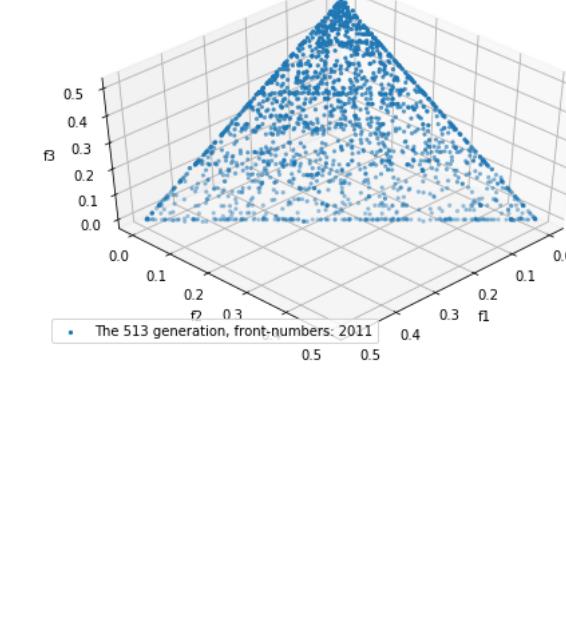
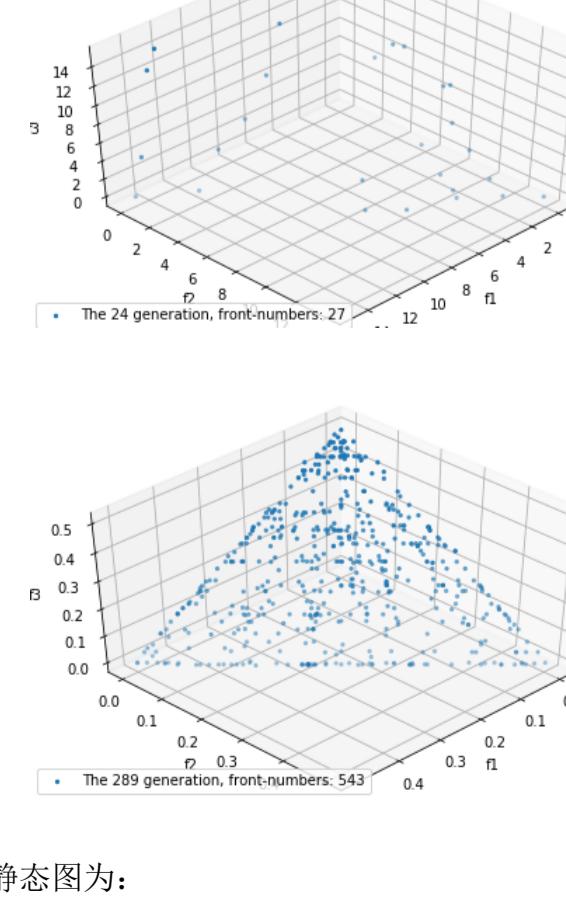
案例:

在解决 3 目标优化问题时绘制进化过程中的帕累托前沿的动画, 以观察搜索到的帕累托前沿的动态变化。并且在进化结束后绘制一个最终的帕累托前沿图。

```
...  
ax = None # 存储上一帧图片  
# 开始进化  
for gen in range(MAXGEN)  
...  
# 进化操作  
...  
# 更新帕累托最优集目标函数值矩阵NDSetObjV  
...  
# 绘图  
ax = ga.frontplot(NDSetObjV, False, ax, gen + 1)
```

```
# 进化结束  
ga.frontplot(NDSetObjV, True) # 绘制最终的帕累托前沿图  
# end
```

运行结果动画部分截图如下:



最终的前沿面静态图为:



## 注意细则

本章介绍 Geatpy 的几点注意细则，以提高编程效率。

### 1. 目标函数正确编写是关键

在使用 Geatpy 求解进化算法问题时，目标函数的正确编写是关键。目标函数可以是简单的或复杂的，只要能够把目标函数值求出来即可。这里需要注意 Geatpy 的目标函数矩阵的编写规范（详见 Geatpy 数据结构章节），目标函数矩阵一般在 Geatpy 中命名为 *ObjV* 或与之类似的名字，它是 Numpy array 类型的列向量（单目标）或矩阵（多目标）。*ObjV* 的每一列对应一个目标，每一行对应种群的一个个体。

因此在编写目标函数的时候，返回的目标函数矩阵必须是拥有与种群一样的行数。一般可以利用 Numpy 的矩阵运算来快速求出种群各个个体对应的目标函数值。当然，当目标函数比较复杂的时候，可以用循环的方法把各个个体的目标函数值求出来。

### 2. 处理约束条件的两种方法

在前面的“函数接口及进化算法模板”章节中详细介绍了 Geatpy 处理约束条件的两种方法，并介绍了 Geatpy 中用于标记个体是否满足约束条件的变量 *LegV*。这里再总结一下：

1) 在目标函数 *aimfuc* 中，当找到非可行解个体时，当即对该个体的目标函数值进行惩罚。若为最小化目标，则惩罚时设置一个很大的目标函数值；反之设置一个很小的目标函数值。

2) 在目标函数 *aimfuc* 中，当找到非可行解个体时，并不当即对该个体的目标函数值进行惩罚，而是修改其在 *LegV* 上对应位置的值为 0，同时编写罚函数 *punishing*，对 *LegV* 为 0 的个体加以惩罚——降低其适应度。事实上，在 Geatpy 内置的算法模板中，已经对 *LegV* 为 0 的个体的适应度加以一定的惩罚，因此若是使用内置模板，则不需要编写罚函数 *punishing*。此时若仍要编写罚函数 *punishing* 的话，起到的是辅助性的适应度惩罚。

(这里的“罚函数 *punishing*”跟数学上的“罚函数”含义是不一样的。后者是纯粹的数学公式，而前者是值使用 Geatpy 时自定义的一个名为‘*punishing*’的根据可行性列向量 *LegV* 来对非可行解的适应度 *FitnV* 进行惩罚的一个函数。)

**特别注意：**如果采取上面的方法 1 对非可行解进行惩罚，在修改非可行解的目标函数值时，必须设置一个“极大或极小”的值，即当为最小化目标时，要给非可行解设置一个绝对地比所有可行解还要大的值；反之要设置一个绝对比所有可行解小的值。否则会容易出现“被欺骗”的现象：即某一代的所有个体全是非可行解，而此时因为惩罚力度不足，在后续的进化中，再也没有可行解比该非可行解修改后的目标函数值要优秀(即更大或更小)。此时就会让进化算法“被欺骗”，得出一个非可行解的“最优”搜索结果。因此，在使用方法 1 的同时，可以同时对 *LegV* 加以标记为 0，两种方法结合着对非可行解进行惩罚。

**算法注意：**若采用了上述的方法 2 对非可行解个体进行了标记，则在种群进化过程中，Geatpy 的内置算法模板采用“遗忘策略”来排除这些个体。这里的“排除”并不是指不让这些非可行解个体保留到下一代，这个“排除”是对于进化记录器而言的：在进化过程中，当使用进化记录器记录各代种群的最优解时，需要排除非可行解个体对记录的影响。它是不会影响种群进化的。根据进化算法的原理非可行解个体保留到下一代的概率只会被降低，而并非将它们完全排除。（“遗忘策略”详见以’sga\_real\_templet’等进化算法模板的源代码。）

在多目标优化中，假如采取上面的惩罚方法 2，则在调用’ndomin’、’ndomindeb’、’ndominfast’ 等计算种群非支配个体及种群个体适应度时，这个记录着种群个体是否是可行解的变量 *LegV* 就起到非常重要的作用了。它会让算法排除这些个体对非支配排序的影响。

### 3. 关于 *maxormin* 的使用

Geatpy 是遵循最小化目标原则的，即认为目标函数值越小越好。因此，在需要最大化目标时，需要设置’*maxormin*’ 为 -1。在调用内核函数时，假如函数的输入或输出参数列表中包含与目标函数值有关的变量时，需要注意是否要对这些输入、输出参数乘上 *maxormin*。比如使用’*ranking*’ 函数计算种群个体的适应度时，传入的 *ObjV* 就需要乘上’*maxormin*’（详见“*ranking* 参考资料”）。当使用’*upNDSet*’ 函数来更新多目标优化的全局帕累托最优解集时，不但要对输入的’*NDSetObjV*’ 乘上’*maxormin*’，对于计算后返回的’*NDSetObjV*’，也需要乘上’*maxormin*’ 进行还原。需不需要乘’*maxormin*’，本质上取决于算法是否是根据目标函数值进行相关的计算，因此，当碰到输入输出参数列表中包含相关类型的变量时，查看以下该函数的详细文档，或使用’*help*’ 命令查看帮助文档即可。掌握规律后就可以不再依赖于文档了。

### 4. 注意区分’*percisions*’ 的含义

’*percisions*’ 是 Geatpy 的 demo 示例代码中经常出现的一个变量，它事实上是调用’*crtfld*’ 函数来创建“区域描述器”（又称“译码矩阵”）时的一个传入参数。根据“*crtfld* 参考资料”（或可用’*help*’ 命令查看相应的帮助文档），该’*percisions*’ 是有两重含义的：

1) 当使用’*crtfld*’ 创建一个表示二进制/格雷码编码的区域描述器 *FieldD* 时，’*percisions*’ 用于控制二进制/格雷码的编码精度。

**注意：**这个“精度”并不是平常所说的“精确到小数点后多少位”，而是值用二进制/格雷码编码后所能够表达小数点后多少位的控制变量。

2) 当使用’*crtfld*’ 创建一个表示实值编码的种群时，’*percisions*’ 并不代表“精度”，而是用于控制变量的边界。比如：当某个变量范围为 [-1,1] 时，假如传入’*crtfld*’ 的 *percisions* 对应的值为 2，那么，由于该变量不包含上界，因此，’*crtfld*’ 会对 [-1,1] 进行适当的调整，使得返回的区域描述器中，该变量的范围被修正为： [-1,0.99]。

### 5. 关于 Geatpy 输出动画的问题

使用 Geatpy 的内置算法模板可以绘制动画。相关的方法详见内置算法模板的源代码或“Geatpy 教程”的“数据可视化”章节。这里要注意的是，若要绘制动画，需要运行前在控制台中输入”*matplotlib qt5*”这条命令。

这里，用普通的 Python 控制台是无法执行这条命令的。因此建议使用 ipython，或者一个简单的方法是安装 Anaconda，利用里面的 Spyder 来进行 Python 代码的编写与执行。在 Spyder 的控制台中，就可以执行”*matplotlib qt5*”并顺利绘制 Geatpy 的动画了。

### 6. 关于强精英策略与增强种群个体的多样性

在 Geatpy 中，’*etour*’ 是一个增强的精英保留锦标赛选择算子。采用该算子进行个体的选择，可以保证最优个体被保留下来。另外，在进化算法模板中，可以采用“父子合并”共同选择出下一代的方法来替代传统的“重插入”生成下一代，以增强精英保留的效果。因为如果采用“重插入”的方法生成下一代，若代沟设置不合理，则会容易丢失种群中较优甚至是优解。

另外，可以采用’*indexing*’ 或’*powing*’ 算子来代替’*ranking*’ 算子进行种群适应度的计算，也可以在一定程度上增强精英策略的效果，加速算法的收敛速度。

如何增强种群个体的多样性？Geatpy 的内置进化算法模板提供增强种群多样性的机制（详见进化算法模板源码）。此时设置 *distribute* 为 True，内置模板就会调用该机制。该机制适当地修改了种群个体的适应度，在一定程度上增强了目标函数值分布稀疏的个体的适应度，从而改善了种群的分布性。这个改善效果在多目标优化中尤为明显，可增强帕累托前沿的分布性。