

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, 但我们不使用它, 于是本文档默认 `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“`print(变量.shape)`”语句来输出其维度信息, 以确定其准确的维度。

一. 基本数据结构

1.1 种群染色体

`Geatpy` 中, 种群染色体是一个 `numpy` 的 `array` 类型的二维矩阵, 一般用 `Chrom` 命名, 每一行对应一个个体的一条染色体。若要采用多染色体, 则可以创建多个相关联的 `Chrom` 即可。默认一个 `Chrom` 的一行对应的只有一条染色体。

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, 则 `Chrom` 的结构如下所示:

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

`Geatpy2` 中不再对一个种群染色体矩阵划分若干份形成多个子种群染色体矩阵。因此若需要使用多种群, 可以使用多个 `Chrom` 来存储各个种群的染色体。

1.2 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。它是种群染色体矩阵 `Chrom` 经过解码后得到的基因表现型矩阵, 每一行对应一个个体, 每一列对应一个决策变量。若用 `Nvar` 表示变量的个数, 则种群表现型矩阵 `Phen` 的结构如下图:

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

`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一位就对应着决策变量的实际值, 即这种编码方式下 `Phen` 等价 `Chrom`。

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。

1.3 目标函数值

`Geatpy` 采用 `numpy` 的 `array` 类型矩阵来存储种群的目标函数值。一般命名为 `ObjV`, 每一行对应每一个个体, 因此它拥有与 `Chrom` 相同的行数; 每一列对应一个目标函数, 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:

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

1.4 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。

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

`Geatpy` 中的适应度遵循“最小适应度为 0”的约定。

1.5 违反约束程度

`Geatpy` 采用 `numpy array` 类型的矩阵 `CV`(Constraint Violation Value) 来存储种群个体违反各个约束条件的程度。一般命名为 `CV`, 它的每一行对应种群的每一个个体, 因此它拥有与 `Chrom` 相同的行数; 每一列对应一个约束条件, 因此若有一个约束条件, 那么 `CV` 矩阵就会只有一列, 如有多个约束条件, `CV` 矩阵就会有多列。如果设有 `num` 个约束, 则 `CV` 矩阵的结构如下图所示:

$$\text{CV} = \begin{pmatrix} c_{1,1} & c_{1,2} & c_{1,3} & \cdots & c_{1,num} \\ c_{2,1} & c_{2,2} & c_{2,3} & \cdots & c_{2,num} \\ c_{3,1} & c_{3,2} & c_{3,3} & \cdots & c_{3,num} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ c_{Nind,1} & c_{Nind,2} & c_{Nind,3} & \cdots & c_{Nind,num} \end{pmatrix}$$

`CV` 矩阵的某个元素若小于或等于 0, 则表示该元素对应的个体满足对应的约束条件。若大于 0, 则表示违反约束条件, 在大于 0 的条件下值越大, 该个体违反该约束的程度就越高。`Geatpy` 提供两种处理约束条件的方法, 一种是罚函数法, 另一种是可行性法则。在使用可行性法则处理约束条件时, 需要用到 `CV` 矩阵。具体用法可详见后面两章中关于使用可行性法则来处理约束条件的相关说明。

1.6 译码矩阵

`Geatpy` 使用译码矩阵(俗称区域描述器)来描述种群染色体的特征, 如染色体中的每一位元素所表达的决策变量的范围、是否包含范围的边界、采用二进制还是格雷码、是否使用对数刻度、染色体解码后所代表的决策变量的是连续型变量还是离散型变量等等。

在只使用工具箱的库函数而不使用 `Geatpy` 提供的面向对象的进化算法框架时, 译码矩阵可以单独使用。若采用 `Geatpy` 提供的面向对象的进化算法框架时, 译码矩阵可以与一个存储着种群染色体编码方式的字符串 `Encoding` 来配合使用。目前 `Geatpy` 中有三种 `Encoding`, 分别为:

- ‘BG’ (二进制/格雷码)
- ‘RI’ (实整数编码, 即实数和整数的混合编码)
- ‘P’ (排列编码, 即染色体每一位的元素都是互异的)

这里有个小小的归类值得注意: ‘RI’ 和 ‘P’ 编码的染色体都不需要解码, 染色体上的每一位本身就代表着决策变量的真实值, 因此“实整数编码”和“排列编码”可统称为“实值编码”。

1) 对于 `Encoding = ‘BG’` 的种群, 使用 8 行 n 列的矩阵 `FieldD` 来作为译码矩阵, n 是染色体所表达的决策变量个数。`FieldD` 的结构如下:

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

`lens`, `lb`, `ub`, `codes`, `scales`, `lbin`, `ubin`, `varTypes` 均为长度等于决策变量个数的行向量。

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` 分别代表每个决策变量的上界和下界。

`codes` 指明染色体子串用的是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。(注意: 当某个变量是对数刻度时, 其取值范围中不能有 0, 即要么上下界都大于 0, 要么上下界都小于 0。)

从 **2.5.0 版本开始**, 取消了对对数刻度的支持, 该参数暂时保留, 但不再起作用。

`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。

`varTypes` 指明了决策变量的类型, 元素为 0 表示对应位置的决策变量是连续型变量; 1 表示对应的是离散型变量。

例如:

$$\text{FieldD} = \begin{pmatrix} 4 & 4 & 5 \\ 1 & 2 & 3 \\ 10 & 9 & 15 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

它表示待解码的种群染色体矩阵 `Chrom` 解码后可以表示成 3 个决策变量, 每个决策变量的取值范围分别是 `[1,10]`, `[2,9]`, `[3,15]`。其中第一第二个变量采用的是二进制编码, 第三个变量采用的是格雷编码, 且第一、第三个决策变量为连续型变量; 第二个为离散型遍历。若 `Chrom` 为:

$$\text{Chrom} = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

则可以执行以下语句进行解码:

```
import geatpy as ea

Phen = ea.bs2ri(Chrom, FieldD)
```

解码后得到的种群表现型矩阵为:

$$\text{Phen} = \begin{pmatrix} 7.6 & 5 & 11.51612903 \\ 3.4 & 8 & 6.87096774 \\ 1.6 & 7 & 5.32258065 \end{pmatrix}$$

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 3 行 n 列的矩阵 `FieldDR` 来作为译码矩阵, n 是染色体所表达的控制变量个数。`FieldDR` 的结构如下:

$$\text{FieldDR} = \begin{pmatrix} x_1\text{下界} & \cdots & x_n\text{下界} \\ x_1\text{上界} & \cdots & x_n\text{上界} \\ varTypes_1 & \cdots & varTypes_n \end{pmatrix}$$

这种结构的译码矩阵适用于 `Encoding = ‘RI’`(实整数编码)和‘P’(排列编码)的种群染色体的解码。其中‘P’(排列编码)的译码矩阵会稍微有一点特殊之处: 它要求 `FieldDR` 的第一行所有元素都相等, 第二行所有元素也都相等, 且第三行元素均为 1(这是因为排列编码本身变量是离散的)。此时若记 `FieldDR` 有 `Lind` 列(即染色体长度为 `Lind`), 则要求上界 - 下界 + 1 \geq `Lind`。例如:

$$\text{FieldDR} = \begin{pmatrix} 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

它若是作为排列编码种群的译码矩阵, 则表示种群染色体是由集合 `2, 3, 4, 5, 6, 7, 8, 9, 10` 中任意抽取 7 个数出来的全排列, 比如 `Chrom` 为:

$$\text{Chrom} = \begin{pmatrix} 2 & 3 & 4 & 7 & 6 & 8 & 10 \\ 8 & 7 & 6 & 4 & 9 & 5 & 3 \end{pmatrix}$$

上面的 `FieldD` 和 `FieldDR` 都是 `numpy` 的 `array` 类型, 可统称为“`Field`”。可以直接用代码创建, 例如:

```
import numpy as np

FieldDR=np.array([-3, -4, 0, 2],

[ 2, 3, 2, 2],

[ 0, 0, 0, 0]])
```

也可以用 `Geatpy` 内置的 `crtfld` 函数来方便地快速生成区域描述器, 其详细用法可执行 `help(crtfld)` 或查看 API 文档。

1.7 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。这种进化记录器有多种, 其中一种是 `numpy` 的 `array` 类型的, 结构如下:

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

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……

另外一种进化记录器是一个列表, 列表中的每一个元素都是一个拥有相同数据类型的数据。比如在 `Geatpy` 的面向对象进化算法框架中的 `pop_trace`, 它是一个列表, 列表中的每一个元素都是历代的种群对象。

二. 种群结构

2.1 Population 类

在 `Geatpy` 提供的面向对象进化算法框架中, 种群类(`Population`) 是一个存储着与种群个体相关信息的类。它有以下基本属性:

`sizes`: `int` - 种群规模, 即种群的个体数目。

`ChromNum`: `int` - 染色体的数目, 即每个个体有多少条染色体。

`Encoding`: `str` - 染色体编码方式。

`Field`: `array` - 译码矩阵, 可以是 `FieldD` 或 `FieldDR`。

`Chrom`: `array` - 种群染色体矩阵, 每一行对应一个个体的一条染色体。

`Lind`: `int` - 种群染色体长度。

`ObjV`: `array` - 种群目标函数值矩阵。

`FitnV`: `array` - 种群个体适应度列向量。

`CV`: `array` - 种群个体违反约束条件程度的矩阵。

`Phen`: `array` - 种群表现型矩阵。

可以直接对种群对象进行提取个体、个体合并等操作, 比如 `pop1` 和 `pop2` 是两个种群对象, 则通过语句“`pop3 = pop1 + pop2`”, 即可把两个种群的个体合并, 得到一个新的种群。在合并的过程中, 实际上是把种群的各个属性进行合并, 然后用合并的数据来生成一个新的种群(详见 `Population.py`)。又比如执行语句“`pop3 = pop1[[0]]`”, 可以把种群的第 0 号个体抽取出来, 得到一个新的只有一个个体的种群对象 `pop3`。值得注意的是, 种群的这种个体抽取操作要求下标必须为列表或是 `Numpy array` 类型的行向量, 不能是标量(详见 `Population.py`)。

易错注意: 在 `Geatpy` 中, 必要地对种群对象的这些成员属性进行合法性检查是必要的, 但过多的检查会在一定程度上降低框架的性能。其中最易使得种群对象成员属性出现异常的地方在于目标函数值矩阵 `ObjV` 以及 `CV` 矩阵的生成。在 `Geatpy` 中, `ObjV` 和 `CV` 是在 `Problem` 问题类的目标函数接口 `aimFunc()` 中计算生成的, 无论中间过程它们具体是如何计算的, 计算得到的结果必须满足: `ObjV` 和 `CV` 都是 `Numpy array` 类型矩阵, 且行数等于种群的个体数目。`ObjV` 的每一行对应一个个体, 每一列对应一个优化目标。`CV` 矩阵的每一行也是对应一个个体, 而每一列对应一个约束条件。根据 `Geatpy` 数据结构可知, 种群对象中的 `Chrom`、`ObjV`、`FitnV`、`CV` 和 `Phen` 都是 `Numpy array` 类型的行数等于种群规模 `sizes` 的矩阵, 即这些成员属性的每一行都跟种群的每一个个体是一一对应的。`Geatpy` 框架在运行过程中所抛出大多数异常都是由于这些变量不合法所致。此时可以使用“`shape`”来输出变量的维度信息, 比如:

```
print(pop.sizes)

print(pop.Chrom.shape)

print(pop.ObjV.shape)

print(pop.CV.shape)
```

其中 `pop` 为一个种群对象。

特殊用法: 在多种群进化优化中, 往往需要将所有种群的所有个体进行统一的适应度评价, 此时由于各个种群的编码方式 `Encoding`、译码矩阵 `Field` 以及染色体的长度 `Lind` 可能会不尽相同, 不能直接把所有种群对象合并成一个种群对象。此时, 可以构造一个 `Encoding` 为 `None` 的种群, 这种类型的种群不携带有效的 `Field`、`Chrom` 的信息(这些值都被自动设置为了 `None`)。这种特殊用法允许直接将所有种群合并成一个“联合种群”, 从而方便进行个体适应度评价、进化记录、多目标优化非支配解的记录等操作。

2.2 PsyPopulation 类

`PsyPopulation` 类是 `Population` 的子类, 它提供 `Population` 类所不支持的多染色体混合编码。它有以下基本属性:

`sizes`: `int` - 种群规模, 即种群的个体数目。

`ChromNum`: `int` - 染色体的数目, 即每个个体有多少条染色体。

`Encodings`: `list<string>` - 存储各染色体编码方式的列表。

`Fields`: `list<array>` - 存储各染色体对应的译码矩阵的列表。

`Chroms`: `list<array>` - 存储种群各染色体矩阵的列表。

`Linds`: `list<int>` - 存储种群各染色体长度的列表。

`ObjV`: `array` - 种群目标函数值矩阵。

`FitnV`: `array` - 种群个体适应度列向量。

`CV`: `array` - 种群个体违反约束条件程度的矩阵。

`Phen`: `array` - 种群表现型矩阵。

可见 `PsyPopulation` 类基本与 `Population` 类一样, 不同之处是采用 `Linds`、`Encodings`、`Fields` 和 `Chroms` 分别存储多个 `Lind`、`Encoding`、`Field` 和 `Chrom`。

`PsyPopulation` 类的对象往往与带“`psy`”字样的进化算法模板配合使用, 以实现多染色体混合编码的进化优化。

2.3 多种群

多种群的数据结构是一个 `list` 列表, 即用一个列表来存储所有种群(每个种群都是 `Population` 类的对象), 采用多种群协同进化的例子详见“多种群进化优化”章节。