遗传算法(求函数极值)

(一) 算法步骤

1. 初始化种群(编码)

- 这里采用了**二进制编码**，根据给定的染色体长度 `chrom_length` 给出随机的 0-1 串，直到达到种群总数 `population_size` 为止，并**没有考虑正负**
- 种群中每个个体**对应着函数定义域**中的一个值

```
def chroms_encoding(population_size, chrom_length):
    pop_chroms = []

    for i in range(population_size):
        chrom = []
        for j in range(chrom_length):
            chrom.append(random.randint(0, 1))
        pop_chroms.append(chrom)

    return pop_chroms
```

2. 评估种群适应度

- 首先将基因的编码做归一化处理：

  不考虑正负的 n 位二进制编码的表示空间为 $[0, 2^n - 1]$，而函数的定义域为 $[a, b]$，需要建立表示空间区间到函数定义域之间的映射(映射部分其实放在编码和解码部分更好)

  把染色体解码的值 $\hat{x}$ 后输入映射函数 $x = a + \frac{b-a}{2^n-1}\hat{x}$，才得到基因实际对应的定义域上的值 x

- 接着将每个个体的染色体对应的值输入适应度函数，便可得到个体对应的适应度

  这里的适应度函数即为**要求极值的函数**

```
def evaluate(pop_chroms, func, chrom_length, gene_min=0, gene_max=10):
    population_values = []
    pop_chroms_ture = []

    scale = gene_max - gene_min

    for chrom in pop_chroms:
        # 十进制数
        chrom_unnor = chrom_decoding(chrom)
        # 对x做区间映射变化 规范定义域
        chrom_true = scale / (math.pow(2, chrom_length) - 1) * (chrom_unnor - 0)
+ gene_min

        pop_chroms_ture.append(chrom_true)
        population_values.append(func(chrom_true))

    return pop_chroms_ture, population_values
```

3. 自然选择(轮盘赌)

- 首先根据上面得到的适应度计算个体被选择的概率

  适应度可能出现负值，这里模仿 `softmax` 加入了 `exp` 函数将所有适应度映射成大小顺序不变的正数，再将每个个体对应的正数除以全体正数的和，即得到对应个体被选择的概率

  计算公式为$prob_i = \frac{e^{v_i}}{\sum_{k=1}^{n} e^{v_k}}$

- 接着按照个体被选择的概率**由小到大**给种群的染色体数组和概率数组排序
- 再根据种群的概率数组计算累加概率数组
- 最后根据累加概率数组，随机生成概率后使用**二分查找**找出被选择的个体，加入新种群中

  值得注意的是，**新种群的大小没有变化。**交配之后种群中会加入子代种群，但自然选择会在这个同时包含父代和子代的种群中选出新一代的种群，且大小与原种群一致

```
def select(pop_chroms, population_values, population_size):
    # 根据个体适应度计算个体繁衍概率
    population_values_ = np.exp(population_values)
    value_sum = sum(population_values_)
    population_probs = []
    for value in population_values_:
        population_probs.append(value / value_sum)

    # 排序
    order = np.argsort(population_probs)
    population_probs = np.array(population_probs)[order]
    pop_chroms = np.array(pop_chroms)[order]

    # 计算累加概率 方便计算轮盘赌
    population_sum_probs = []
    probs_sum = 0
    for i in range(len(population_probs)):
        probs_sum += population_probs[i]
        population_sum_probs.append(probs_sum)

    # 轮盘赌选择 成为新种群
    new_pop_chroms = []
    for i in range(population_size):
        rand_prob = random.random()
        # prob_k < prob < prob_{k+1}
        # Notice: 直接使用二分查找的返回值 有可能溢出列表
        chrom_selected = bisect(population_sum_probs, rand_prob)
        new_pop_chroms.append(pop_chroms[chrom_selected])

    return new_pop_chroms
```

4. 交叉繁衍

- 根据输入的交配概率 `mating_rate` 依次判断个体是否交配
- 若第 i 个个体交配，则将其作为父本，随机在种群挑选一个作为个体作为母本。随机给出染色体长度限制内的一段区间 [`gene_min`, `gene_max`]，并将父本与母本对应区间的基因进行交换，得到两个新的子代 `child1` 、 `child2` 并加入种群。

```
# 交配
def crossover_mating(pop_chroms, chrom_length, mating_rate):
    population_len = len(pop_chroms)
```

```
    for i in range(population_len):
        # 第i个体交配
        if random.random() > mating_rate:
            child1 = pop_chroms[i]  # female
            child2 = pop_chroms[random.randint(0, population_len - 1)]  # male

            gene_min = random.randint(0, chrom_length - 1)
            gene_max = random.randint(0, chrom_length - 1)
            if gene_max < gene_min:
                gene_min, gene_max = gene_max, gene_min

            # 两个子代的基因交换
            for j in range(gene_min, gene_max + 1):
                child1[j], child2[j] = child2[j], child1[j]

            pop_chroms.append(child1)
            pop_chroms.append(child2)

        else:
            continue

    return pop_chroms
```

5. 变异

- 根据输入的交配概率 `mutation_rate` 依次判断个体是否变异
- 若个体变异，则随机改变其染色体上某个位点基因的值，即**单点变异**

```
def mutation(pop_chroms, chrom_length, mutation_rate):
    for i in range(len(pop_chroms)):
        # 单个基因变异
        if random.random() < mutation_rate:
            gene_mut = random.randint(0, chrom_length - 1)
            if pop_chroms[i][gene_mut] == 1:
                pop_chroms[i][gene_mut] = 0
            else:
                pop_chroms[i][gene_mut] = 1

    return pop_chroms
```

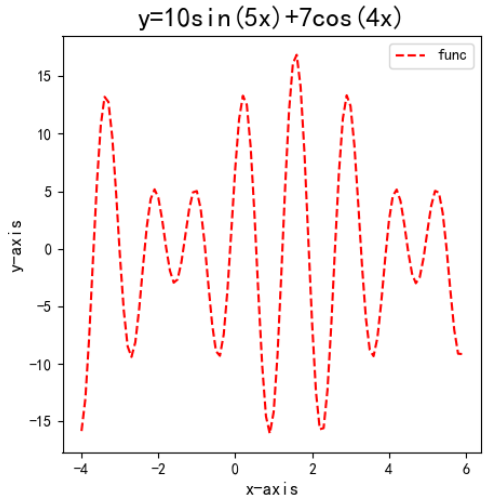
(二) 参数影响分析

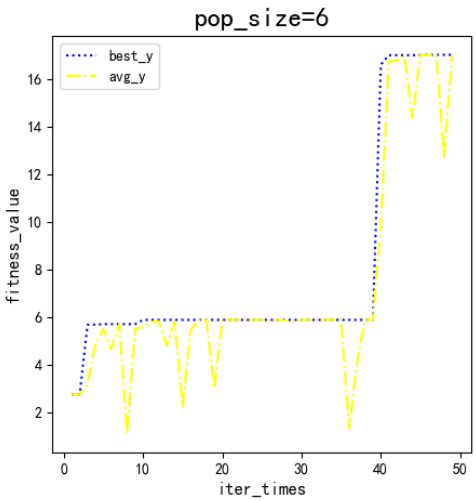
种群大小

实验设置

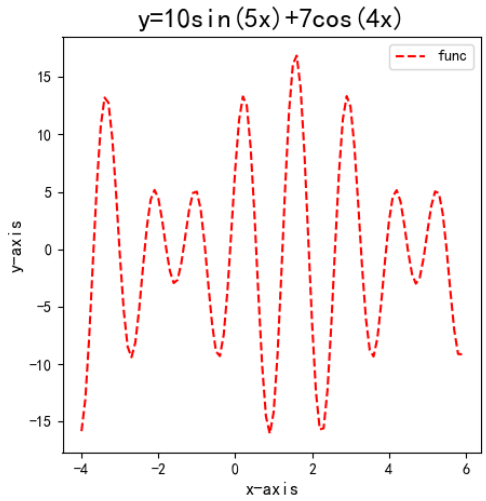
> 染色体长度 chrom_length = 50
> 交叉概率　pc = 0.4
> 变异概率　pm = 0.1

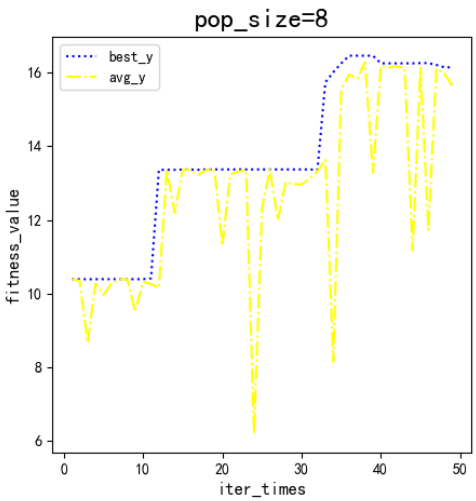
1. population_size = 6



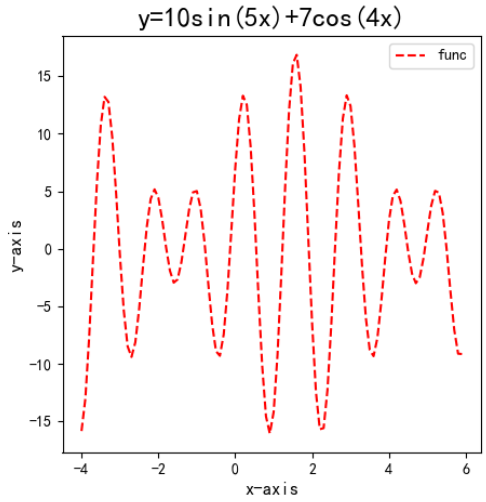


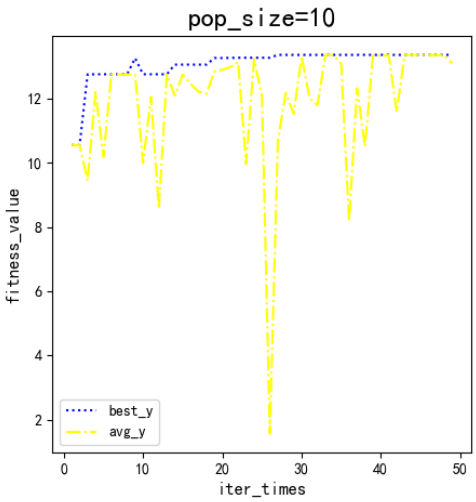
2. population_size = 8





3. population_size = 10





分析

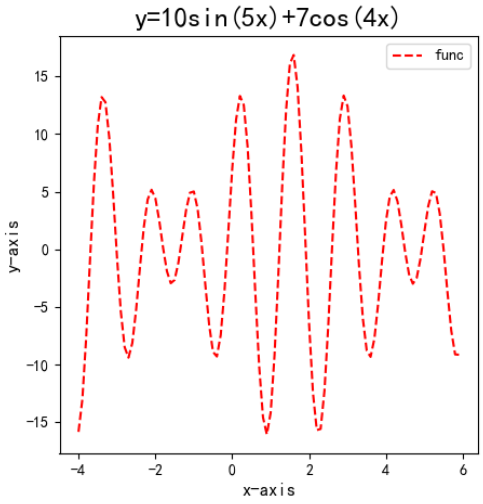
种群数量越大，其中最优个体就越可能更优秀

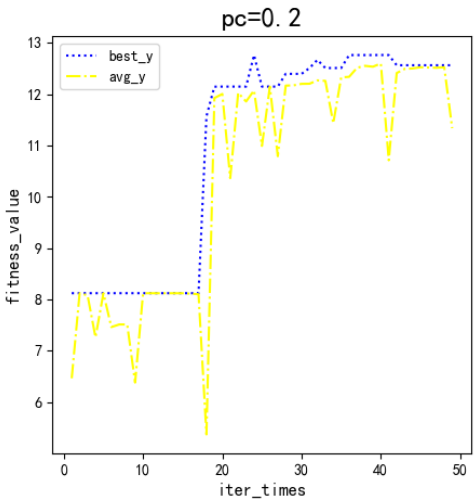
交叉概率

实验设置

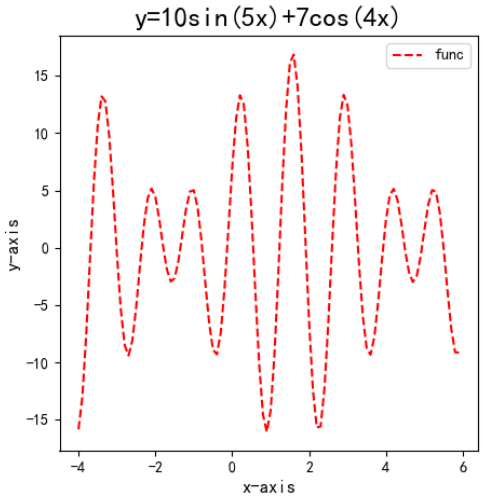
> 种群大小 population_size = 12
> 染色体长度 chrom_length = 50
> 变异概率　pm = 0.1

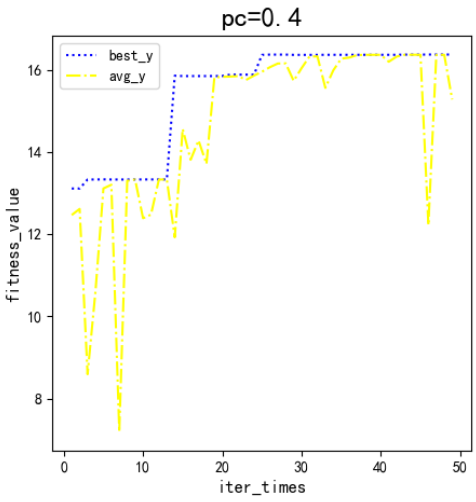
1. pc = 0.2



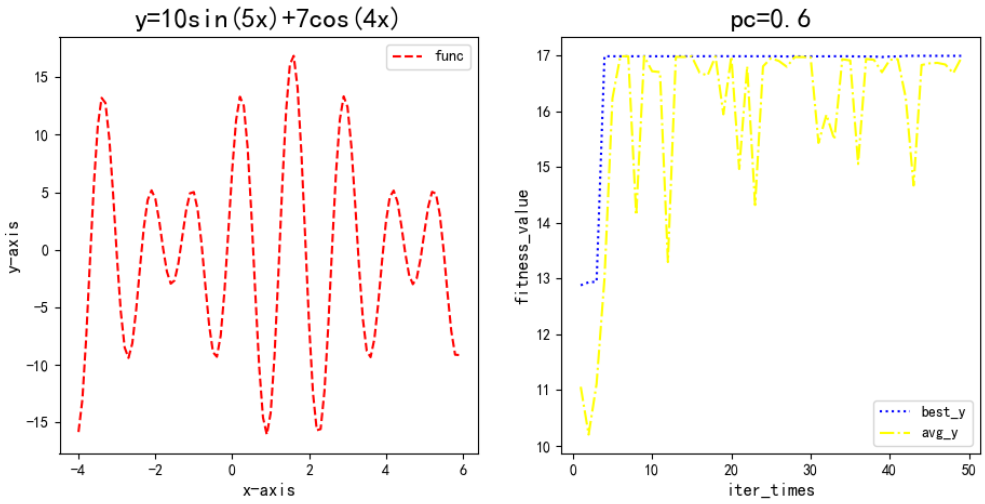


2. pc = 0.4





3. pc = 0.6



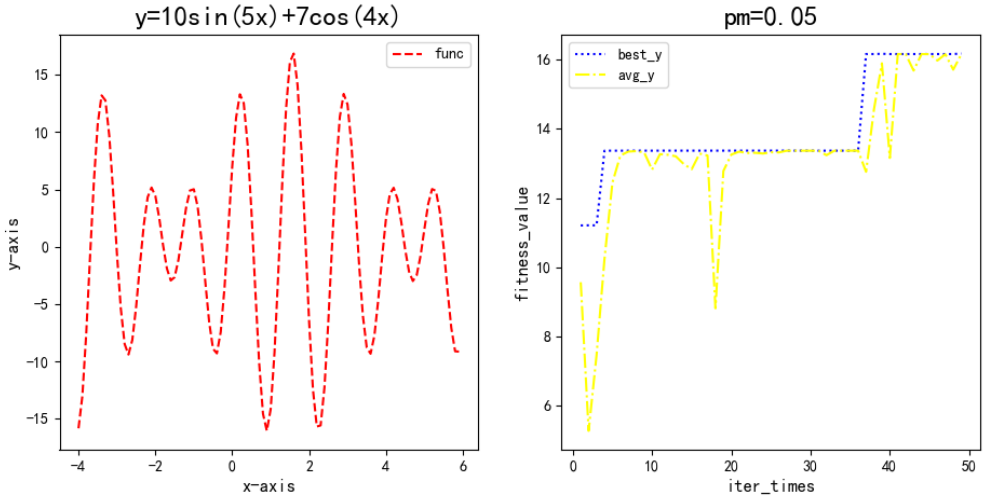
分析

受到初始化的影响，到达最优值的速度不同，但可以看出来交叉概率越大进化是越快的

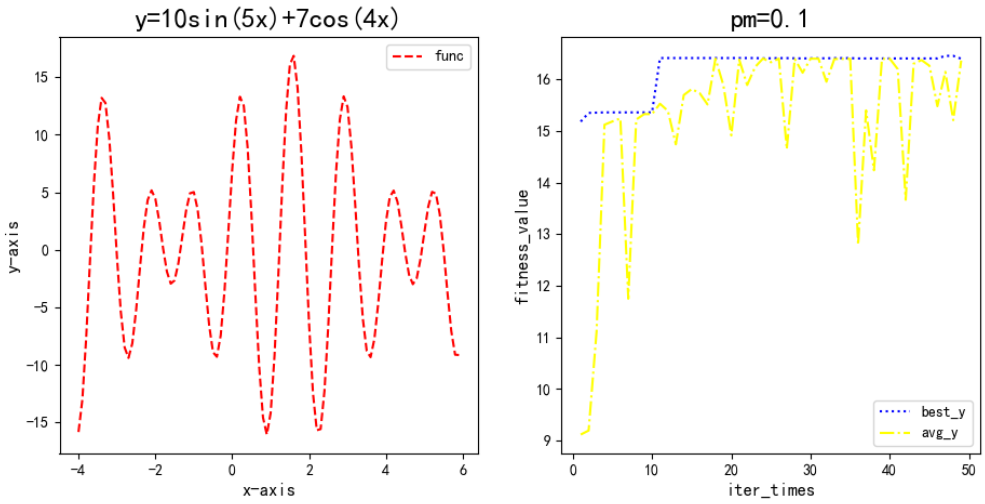
变异概率

> 种群大小 population_size = 20
> 染色体长度 chrom_length = 50
> 交叉概率　pc = 0.4

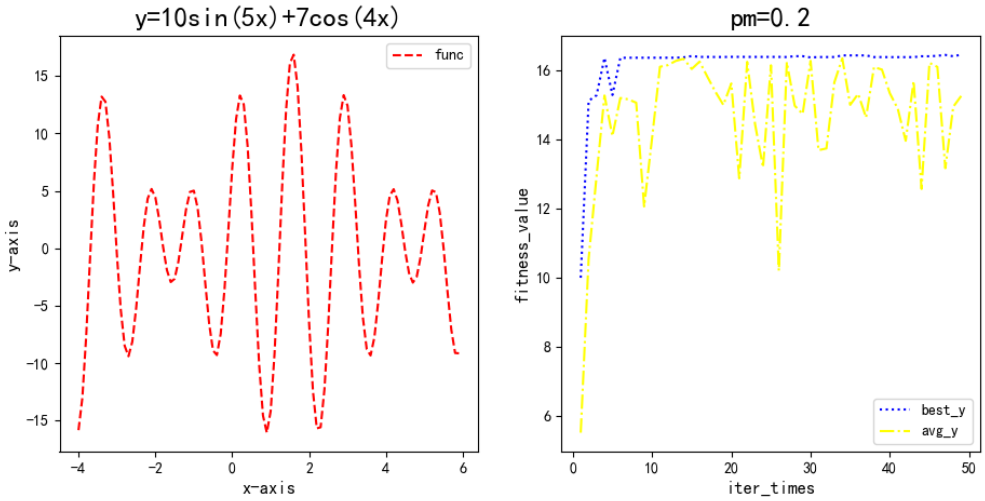
1. pm=0.05



2. pm = 0.1



3. pm=0.2



分析

总体影响不大，感觉遗传算法进化的核心更倾向于交叉而非变异

(三) 与差分进化算法的比较

- 遗传算法的步骤为：编码初始化、适应度计算、选择、交叉、变异
- 差分进化算法的步骤为：编码初始化、适应度计算、变异、交叉、适应度计算、选择

从步骤顺序上已经可以看出两个算法一定的不同。除此之外，对比遗传算法，差分进化算法还有鲁棒性更强、收敛更快、全局搜索能力更好等优点。以下具体探讨具体步骤的差异

编码

遗传算法并没有规定具体使用的编码方式，不过主要采用**二进制编码**，而差分进化算法采用**实数编码**

选择

遗传算法主要采用轮盘赌，劣者概率淘汰；差分采用贪婪绝对淘汰制，劣者绝对淘汰

交叉

遗传算法将种群中任意两个个体的染色体**若干点位的基因互换**来得到新的个体。而差分进化算法的交叉步骤在变异步骤之后，它将变异个体作为中间个体，将任意个体与对应变异个体对应基因互换(保证至少有一点变异基因)，来得到新的个体并将其加入种群，而变异个体抛弃

也即遗传算法中通过父代产生子代来进化，差分进化算法中通过父代自身变异来进化

变异

遗传算法中的变异是通过**随机替换**个体中某个或某段基因来得到新的个体。这样的变异方向随机，变异得到的新的基因仍可能与种群中其他个体的对应基因**重合**，并没有产生“差异”。这样的变异对种群的进化是没有意义的，容易使得种群陷入“**局部最优**”

差分进化算法中的变异是在现有两个个体的差异上乘以缩放因子之后加到第三个个体上来得到新的个体。虽然新的个体由原来的三个个体计算得出，但离三个个体都有一定距离。在**优化后期**种群**内部距离缩小**的情况下，几个个体便可一定程度反应总体的水平，因此离三个个体有一定距离便是离种群有一定距离，这样变异得到的新个体更具有探索意义，相对种群更有"差异"