目录

[一、 粒子群算法求解函数优化问题 3](#_Toc102333153)

[二、 粒子个数、迭代次数的探究 4](#_Toc102333154)

[使用对数坐标系进行可视化 4](#_Toc102333155)

[对照试验 5](#_Toc102333156)

[ 算法实现 5](#_Toc102333157)

[ 实验设计 5](#_Toc102333158)

[ 实验分析 6](#_Toc102333159)

[三、 惯性因子W的探究 10](#_Toc102333160)

[实验设置 11](#_Toc102333161)

[需要衡量的参数 11](#_Toc102333162)

[ 算法找到的最优值 11](#_Toc102333163)

[ 跳出局部最优的能力（冲出平台期概率） 11](#_Toc102333164)

[ 快速性 13](#_Toc102333165)

[ 稳定性 14](#_Toc102333166)

[ 结论与算法改进 15](#_Toc102333167)

[四、 学习率C1,C2的探究 15](#_Toc102333168)

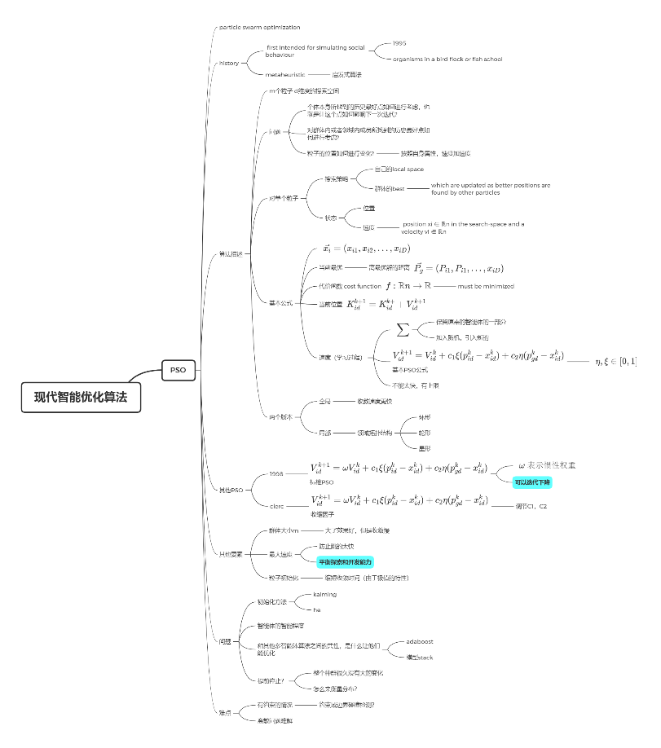
[五、 大作业收获 16](#_Toc102333169)

程序的在线运行地址，为了完整保留实验过程，使用jupyter notebook来完成。地址如下：

<https://colab.research.google.com/drive/12uEouDs0WgpGLXXJuKCdpZBuNEWTzwnU?usp=sharing>

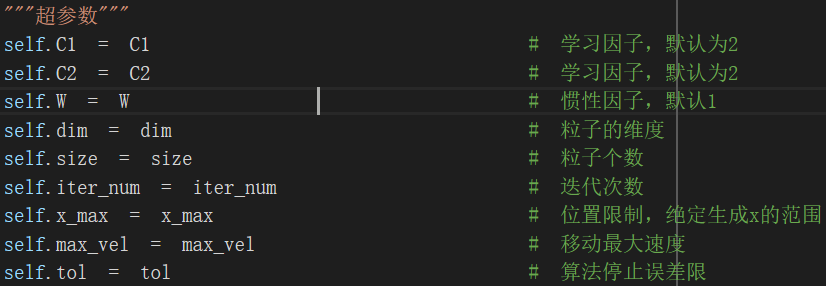
# 粒子群算法求解函数优化问题

粒子群算法开始用于解决连续函数的最小值问题，而本次作业中使用函数是离散概率情况下的多变数扩展，只需要用概率生成给定数量（30）维的标准分布变量，带入方程离散化累加即可。

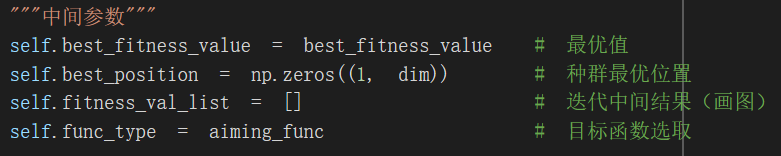


课程笔记：PSO算法的思维导图（大图在文件夹子下）

根据PSO课程上记录的笔记，可以得到算法的流程图和重要参数。



算法超参数



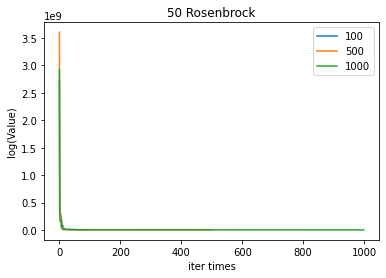
算法运行中间参数

# 粒子个数、迭代次数的探究

要求：所有测试函数维数n=30。观察PSO算法在粒子个数分别设置为50、 100和300，迭代次数分别设置为100、 500和1000情况下的寻找最优解的过程。

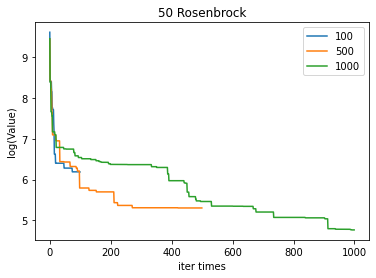
## 使用对数坐标系进行可视化

当使用30维的多变数扩展测试函数时，算法搜索范围可以达到（1e0,1e9），跨度很大。如下图所示，可以观察到三种颜色表示的三种参数下的三条曲线紧贴在一起，无法直观观察的算法的搜索过程。



使用正常坐标系的三种下降曲线

因此，考虑到模电和自控原理中bode图的对数坐标系，我将数据进行了对数化，使用纵轴为对数坐标系的方法，使得不同参数情况下算法的可视化可以直观体现出来。具体实现时，可以采用matplotlib的对数系，也可以采用numpy将数据整体转换。



使用对数坐标系的三种下降曲线

观察到三条曲线成功分离开了，便于算法性能的对比研究。

但是也要注意，对尺度进行缩放也导致差距较为接近的两条线水平方向重合，其斜率也被去了对数，变化趋势不敏感。因此对于算法执行末期，数量级差距不大情况下，还应该回到标准坐标系下探究。

## 对照试验

### 算法实现

同时，吸取前两次实验中算法实现使用顺序结构的弊端。设置过多全局变量定义的超参数导致进行重复实验时出现变量泄露等问题，这些问题在python中（尤其是jupyter notebook中）debug很困难。此次算法实现用类来封装，超参数使用类内变量，使得创建和销毁都更安全，做多组对照实验时出现问题较少。

### 实验设计

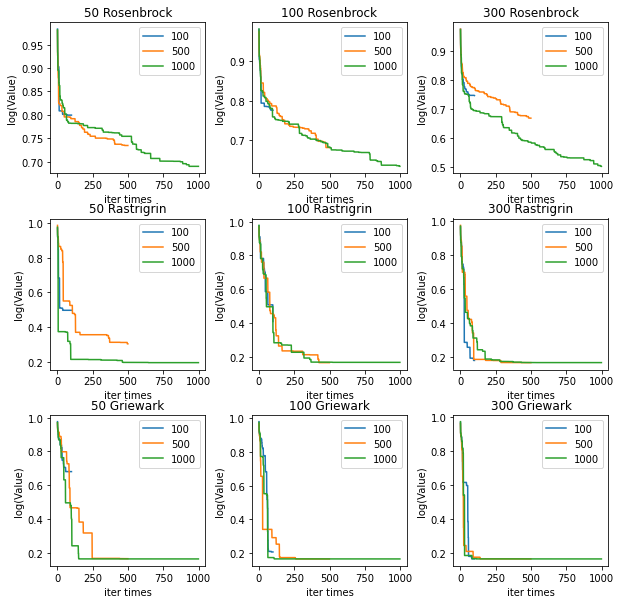
考虑到实验中涉及到三组变量：目标函数、迭代次数、种群量。为了直观展现这些变量之间的关系，我使用了matplotlib的子图绘制和隐函数的迭代器，可以方便的进行重复对照实验。

同时，我也使用了变量控制，每个子图中的三条下降曲线都只有一个超参数变化。比如：当使用50种群搜索rosenbrock函数时，迭代次数分别设置为100、500、1000进行试验，其图像放置于大图的（1，1）位置。

图例：

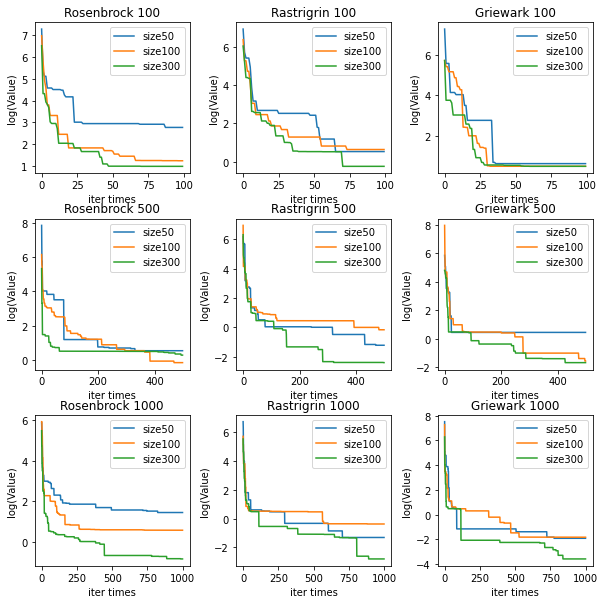
表头为

### 实验分析



对迭代次数的探究

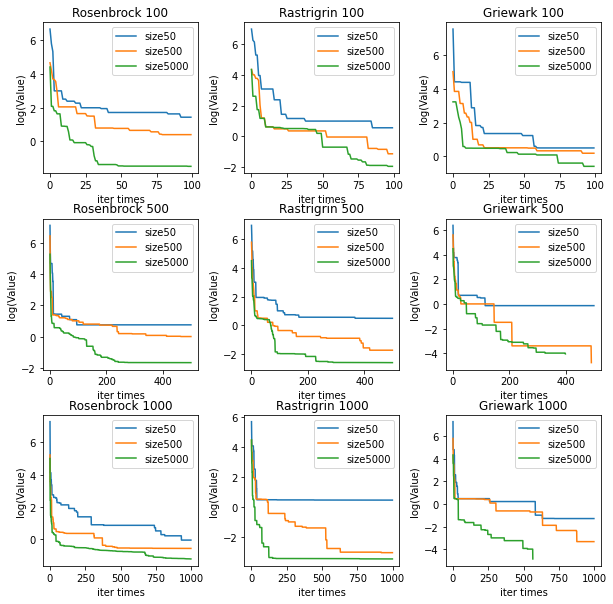
可以看到不论对于何种函数或采用何种规模种群，其中总存在非常大的“平台期”，也存在有平台期对应的拐点。比如观察到对于Rastrigrin和Griewark函数，在250iter后搜索的边际效益逐渐降低，种群已经逐步落入局部最优的陷阱，很难再有进步。继续增加iter，算法几乎没有进步。而对于Rosenbrock函数，由于其搜索难度较大，因此更多的迭代次数能显著增加结果的最优性。根据100次的重复实验，也可以判断对Rosenbrock函数，其搜索结果与迭代次数具有较强线性相关性。



对种群大小的探究

通过衰减速率来看（需要注意的是，图示衰减速率取了对数，但不影响大小关系），大的种群能有更持久的搜索能力，图（2，1）、图（2，3）可以明显看出较小的种群容易陷入某个局部最优值，而种群数量大就有更多机会持续找到更优的局部最优值。既对于大多数情况，大种群的收敛位置要由于小种群。这样的想法也在更大规模、更细密的采样上得到了实验。这里推荐使用google的colab pro平台，其提供的服务器使用多个Intel(R) Xeon(R) CPU @ 2.20GHz，运行速度要比本地快很多，支持了本次大规模测试。

但是同时，我通过做额外的对照试验发现，过大的种群（5000）反而会导致收敛非常慢，计算时间也呈指数倍增长。而且根据问题不同，搜索算法也不是一定会明显更好。种群规模与算法效率、算法性能之间存在trade-off，需要在具体实现测试样例中分析合适的参数取值。比如在griewark测试样例中，算法性能就没有明显提升。

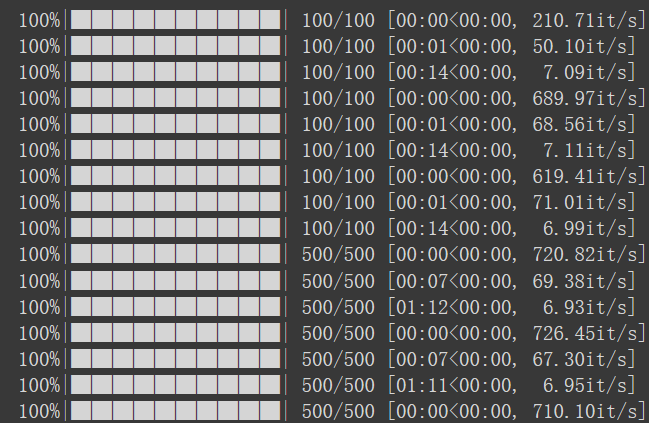


种群量额外的对照实验（种群量为50、500、5000）

这里运算计算时间利用python的修饰器完成，既将计算函数传入@开头的另一个修饰器函数中。下面是修饰器的具体实现：

1. *# 定义一个修饰器函数用来统计函数的运行时间*
2. import time
3. def timmer(func):    *#传入的参数是一个函数*
4. def deco(\*args, \*\*kwargs): *#本应传入运行函数的各种参数*
5. if cntTime:
6. print('\n函数：{\_funcname\_}开始运行：'.format(\_funcname\_=func.\_\_name\_\_))
7. start\_time = time.time()*#调用代运行的函数，并将各种原本的参数传入*
8. res = func(\*args, \*\*kwargs)
9. end\_time = time.time()
10. print('函数:{\_funcname\_}运行了 {\_time\_}秒'
11. .format(\_funcname\_=func.\_\_name\_\_, \_time\_=(end\_time - start\_time)))
12. return res*#返回值为函数*
13. return deco

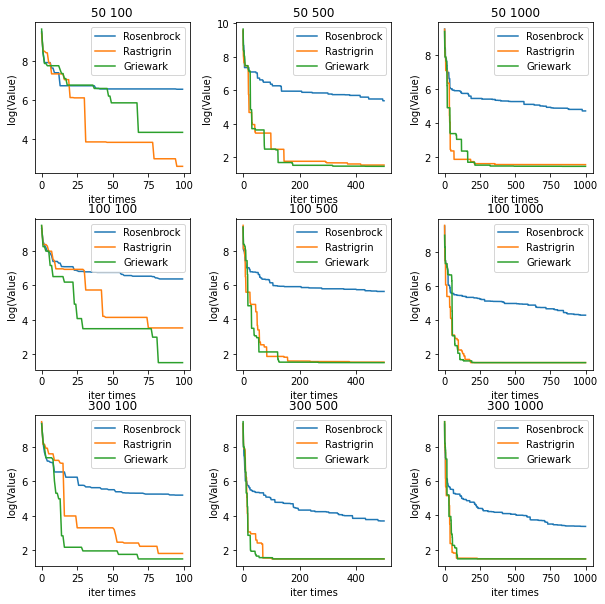
也可以使用tqdm的迭代器来完成，还可以用python自带的分析测试工具完成。观察到计算时间随着种群数量指数级上升，图中也展示了多次运行求平均（减少随机性干扰）的思想。需要注意的是，python中可以使用魔术指令%timeit（或者cProfile分析）来自动重复运行单元格，获取平均算法耗时，但是多次运行后，部分变量会被系统优化进入L2chahe缓存、或者被其他解释器自带优化方法优化，其统计的平均耗时将会小于真实情况。因此手写重复运算可能是更合适的方法，这样内循环的变量会被python自动回收，不会被机器优化。



不同种群量计算耗时分析（单位：每秒多少次迭代）

图中绿线总是表示最大的种群（300），可以观察到其下降速度是最快的。通过搜索速度来看，相同时间（iter数量）内，大种群收敛要明显快于小种群。其找到最优值的能力明显好于小种群。

在9张图中，都可以明显发现算法的搜索能力在不同测试函数上差距非常明显。



对三种目标函数的探究

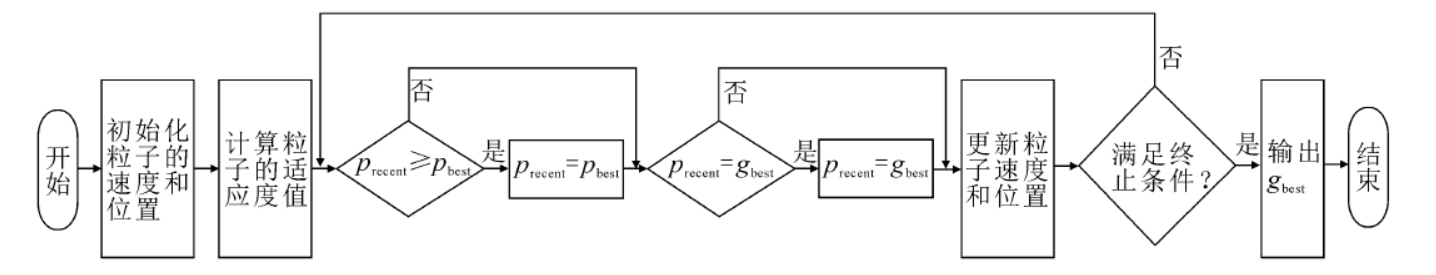
可以观察到，不论在何种情况，Rosenbrock在多变数下的扩展（蓝色）都是最难搜索的。这一结论也与常识符合，作为最优化中常用到的香蕉函数（Rosenbrock别称），其抛物线形的山谷搜索困难。将Rosenbrock函数延伸到随机函数时，因为其随机的特性，任何以梯度下降法为基础的最佳化演算法均无法用来求得此随机函数的最小值。因此其难以搜索是较为平凡的。

Rastrigrin（橙色）和Griewark（绿色）搜索难度接近，但在运行多次后，Griewark的末端下降幅度要在转换到一般坐标系下时（以上图片都是对数坐标系）较明显好于Rastrigrin，说明其搜索难度更低。

此外也可以观察到各函数的下降特性，每个函数都有其下降拐点，达到某一范围后就很难再有进步。这里推测可能是极值在更为细小的区间上，而这里的搜索区间无法缩小，只能在局部震荡。这里可以尝试使用梯度下降方法改进（参考我的实验一），随时间降低动量参数，给于更强的局部搜索能力。

# 惯性因子W的探究

在SHI的论文中首次提出在粒子群算法中引入惯性因子，以平衡算法的全局搜索能力和局部搜索能力。这一改进算法被称为标准PSO算法，在查阅提升算法搜索能力的Trick时，我关注到绝大多数算法都是在此方法基础上改进的。在大多数算法中，其整体结构符合下图的流程，惯性因子主要影响粒子更新的环节。



粒子群算法标准流程

在对照实验中，我将通过分析此参数不同的取值策略对常用测试函数的优化结果的影响，来探究动量（W）参数对算法性能的影响。

使用（5，5）的矩阵中，探索动量W参数对于算法搜索性能的差别。

## 实验设置

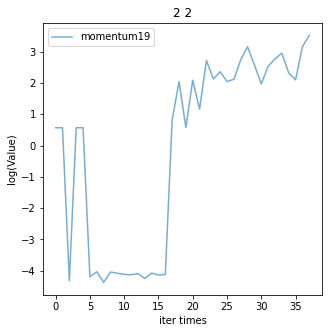
需要注意的是，根据以上实验分析，“香蕉函数”的搜索难度最大。因此这里使用了“香蕉函数”来比对结果。而额外的实验表明，使用不同函数对W函数有着不同的敏感度。因此根据控制变量原则，后续实验全部使用“香蕉函数（Rosenbrock）”。

同时也结合以前实验，将搜索iter数定较为合适的500，C1、C2参数定为2。这主要是考虑到迭代次数过多带来的计算时间过长，以及大多数情况下在500iters后算法很难具有局部跳出的能力。

## 评价指标

### 算法找到的最优值大小

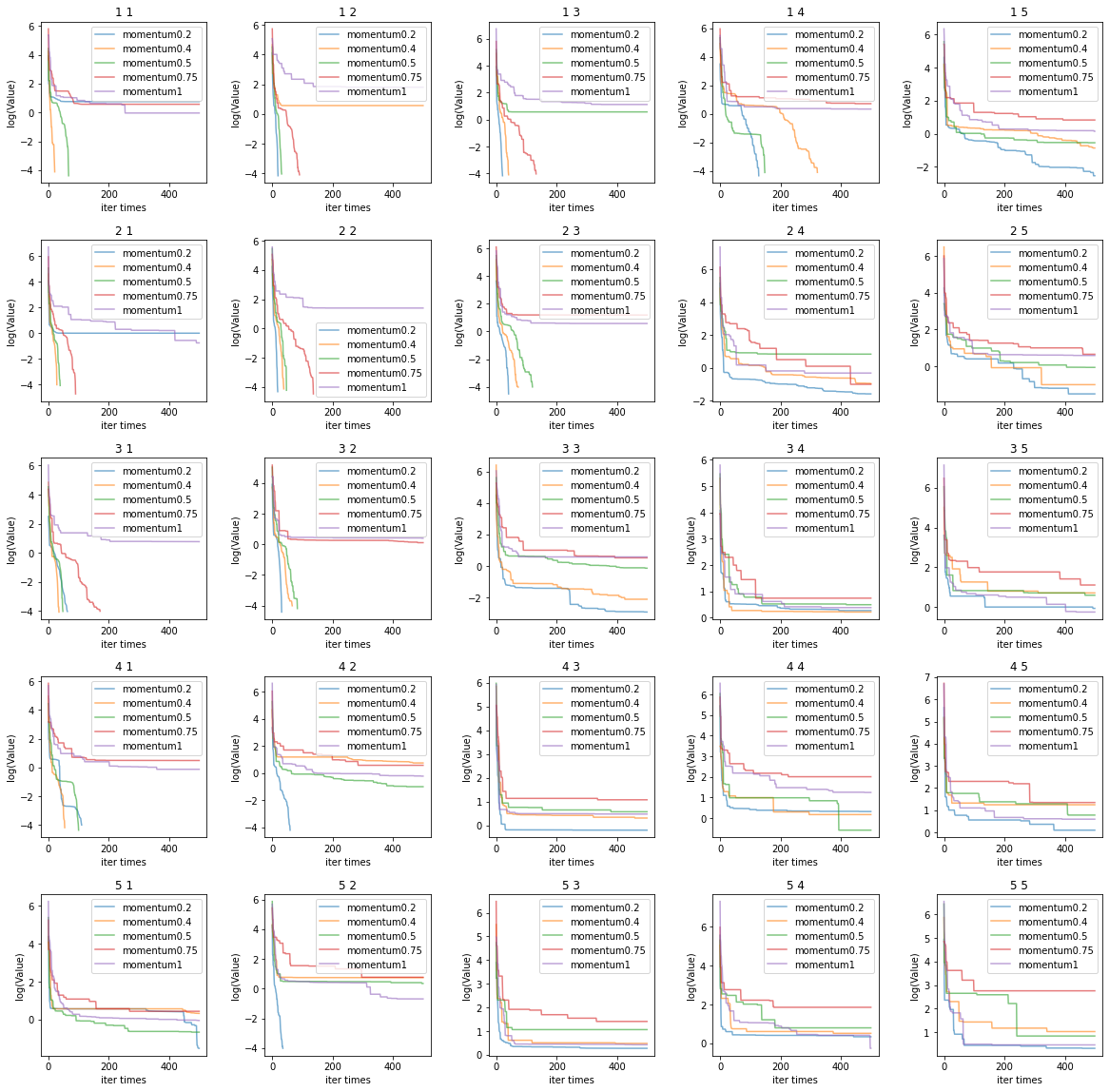
观察到在W大于1后，搜索的下降能力有限，无法在规定的500iter内找到最优解。同时也注意到最优情况出现在编号5-15的动量参数下，对应的动量范围为（0.25-0.65）。W大于1且继续增长后，算法结果越来越差。此图中出现的跳变现象是由于不同区间下，W的设置的粒度不一样



最终搜索到的解（W从0.05增加到20）

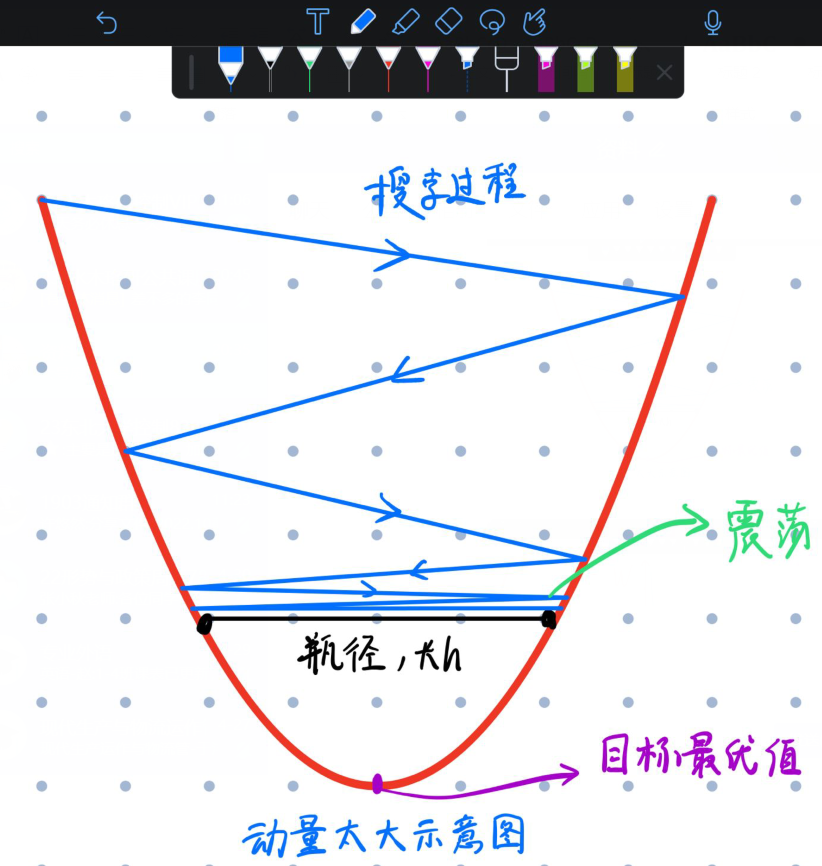
### 跳出局部最优的能力（冲出平台期概率）

从下图中可以看到小的W更容易很快得到最优值。并且其到达“平台期”，既搜索结果开始在最优位置的瓶颈处震荡的概率也小的多。下面的实验中，不同小图分别使用了不同组合的C1、C2参数（共有5\*5组），每个子图内相同的初始解生成方案，仅动量参数不同。



不同动量（W在0.2到1之间）下算法的搜索过程

可以看到，紫色和红色对应的大动量搜索算法平台期较长，说明收敛到最优解较为困难。而小动量反倒能取得较好的搜索效果，甚至有一定概率快速得到较好解，触发提前停止。究其原因，就是动量过大后算法的最优值只存在小范围的“缝隙中”，而过大动量会使得种群速度太大，始终无法进入瓶颈区域。进而导致算法无法进一步优化。因此结果只能在平台上长时间保持，在最优解区域来回震荡。下面以一元函数举例，瓶颈表示由于动量过大导致种群无法进入的阈值，相信在坑洼更复杂的测试样例函数中这一现象会更加明显。这里也可以加入N元高斯分布函数来测试我的想法，但是由于时间不足，将作为日后工作探究。



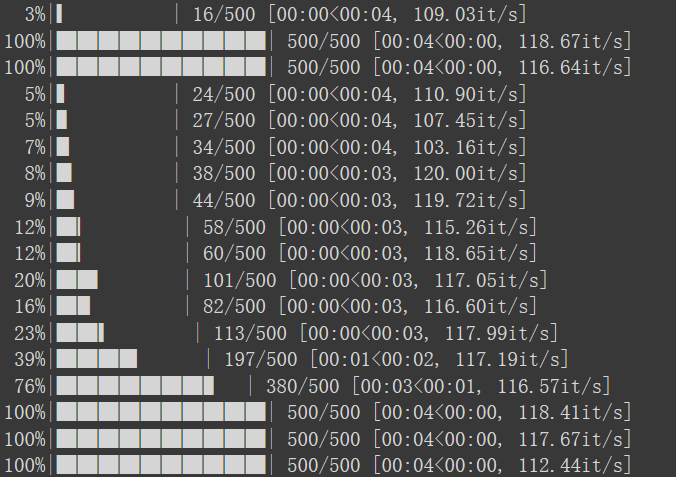
动量过大的搜索震荡示意图

这一发现说明算法不宜采用过大的W，否则会使得种群无法更细粒度的搜索可能存在最优解的区域。造成搜索过程中长时间的“平台期”，而且这一平台期与算法初始解无关，即使是很好的初始解（比如小动量算法很快收敛），大动量算法也难以跳出平台期的，体现了大动量的稳定性和小动量的初值依赖性。

### 快速性

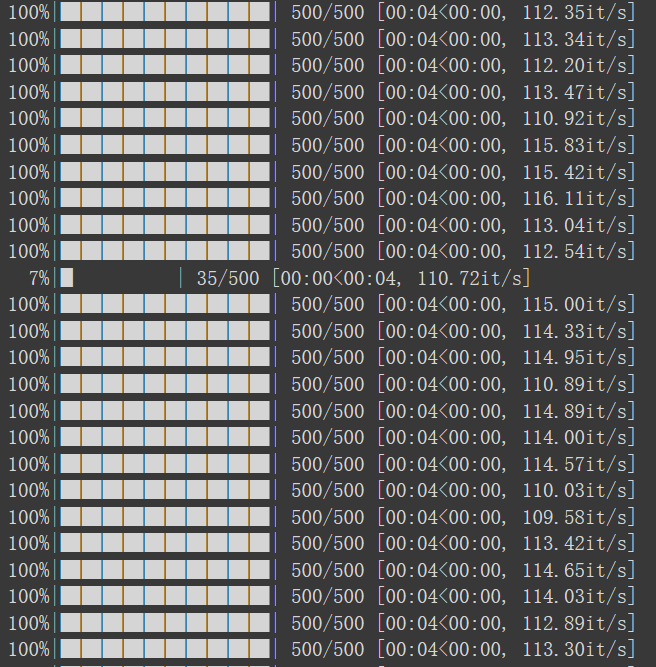
快速性：快速达到最优解的能力。达到提前停止阈值时所需迭代次数。提前停止概率来表示快速搜索能力。为了便于在同一坐标轴观察，实验分为，0＜W＜1和W＞1两部分。

如下图所示，展示了算法实验过程中的提前停止的情况。同时我也观察到算法存在少数异常点（如第二、三行），这可能源于计算log步骤中的溢出，或者是算法不慎收敛到无法跳出的局部最优解区域。



提前停止位置（W从0.05增加到1）

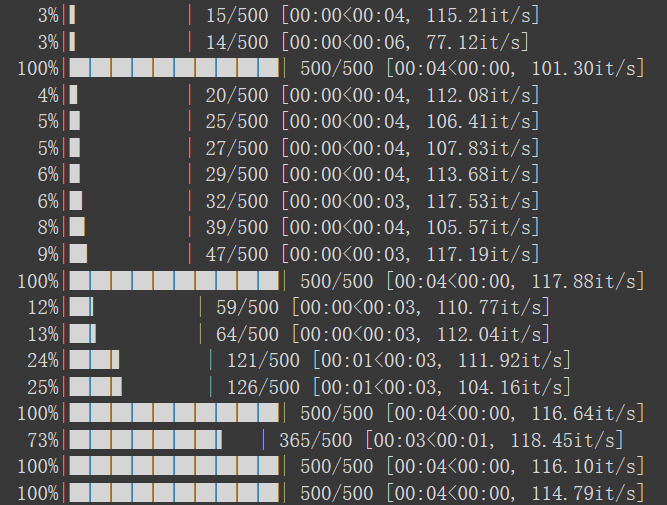
W过大后提前停止出现概率大大降低， 说明收敛到最优解较为困难。究其原因，就是动量过大后算法的最优值只存在小范围的“缝隙中”，而过大动量会使得种群始终无法进入。进而导致算法无法进一步优化。



W过大后提前停止几乎不会出现

### 稳定性

稳定性：算法多次运行，都能在近似的迭代效果下收敛到类似的结果，而较少依赖于初始解的生成情况。这主要表现在离群点出现的概率上。比如，过小的动量使得算法经常落入难以跳出的局部最优区域。主要表现是，使用同一W参数，部分情况下算法直到运行到迭代次数停止准则上限，也无法进入到可接受解的范围（既没有提前停止）。也因此可以使用提前停止的概率来表示算法稳定性。



异常点出现频率（仅截取部分）

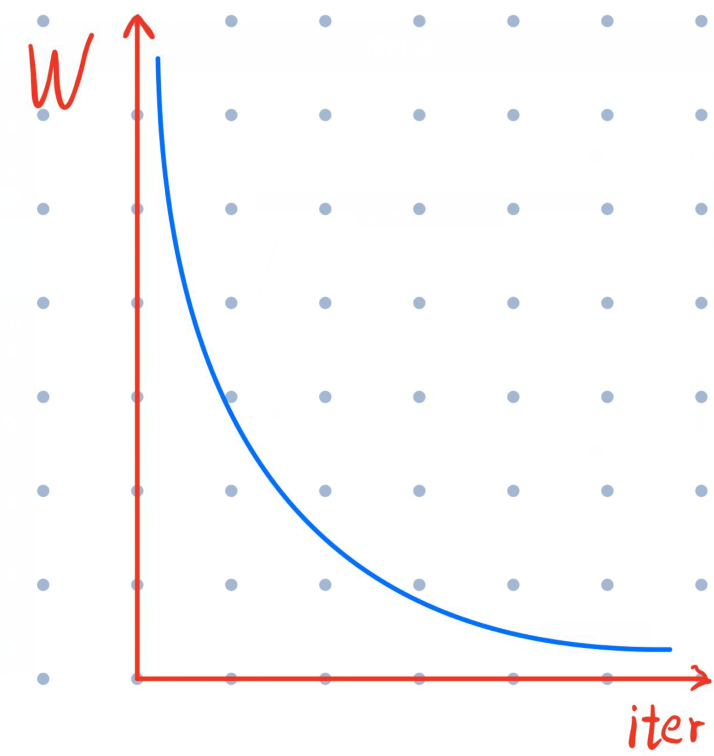
观察到，在过小的W上更容易观察到异常点的产生。说明在过小的W参数下，算法更依赖初始解，如果初始解生成的位置为局部最优山谷，算法将没有能力获得足够多的速度来跳出去。

### 结论与算法改进

综上所述，W参数过大时，粒子群难以及时减速，可能在瓶颈处开始震荡。而W参数过小时，粒子群容易无法获得足够的速度，从小的局部最优值中跳出，从而使得算法稳定性下降，过于依赖初始解的生成。

因此，根据实验效果和类比机器学习中的学习率衰减，我找到了算法的改进点：W参数的分阶段衰减策略。既在算法所处的不同迭代环节使用不同的W策略。开始时使用较大动量（W>1），利用其全局搜索能力来保证算法能跑到总体更好的局部区域。然后衰减动量，使其在后续迭代过程中能“渗入”此局部解域中的细小最优解域中。进而能能快、更稳、更不依赖手工调参W地得到更好的结果。

这里我选择更容易实现的对数动量衰减策略，还可以参考深度学习常用的余弦退火，相信效果更好。



对数动量衰减的W（t）方案

并且实验表明，其综合了算法搜索初期大W的快速移动能力，和算法搜索末期能快速迭代进入局部最优而不在瓶颈处震荡的优势。充分发挥了算法的性能，也避免了切换不同评价函数时需要手工实验调参的问题，使得算法的泛化性能得到提升。

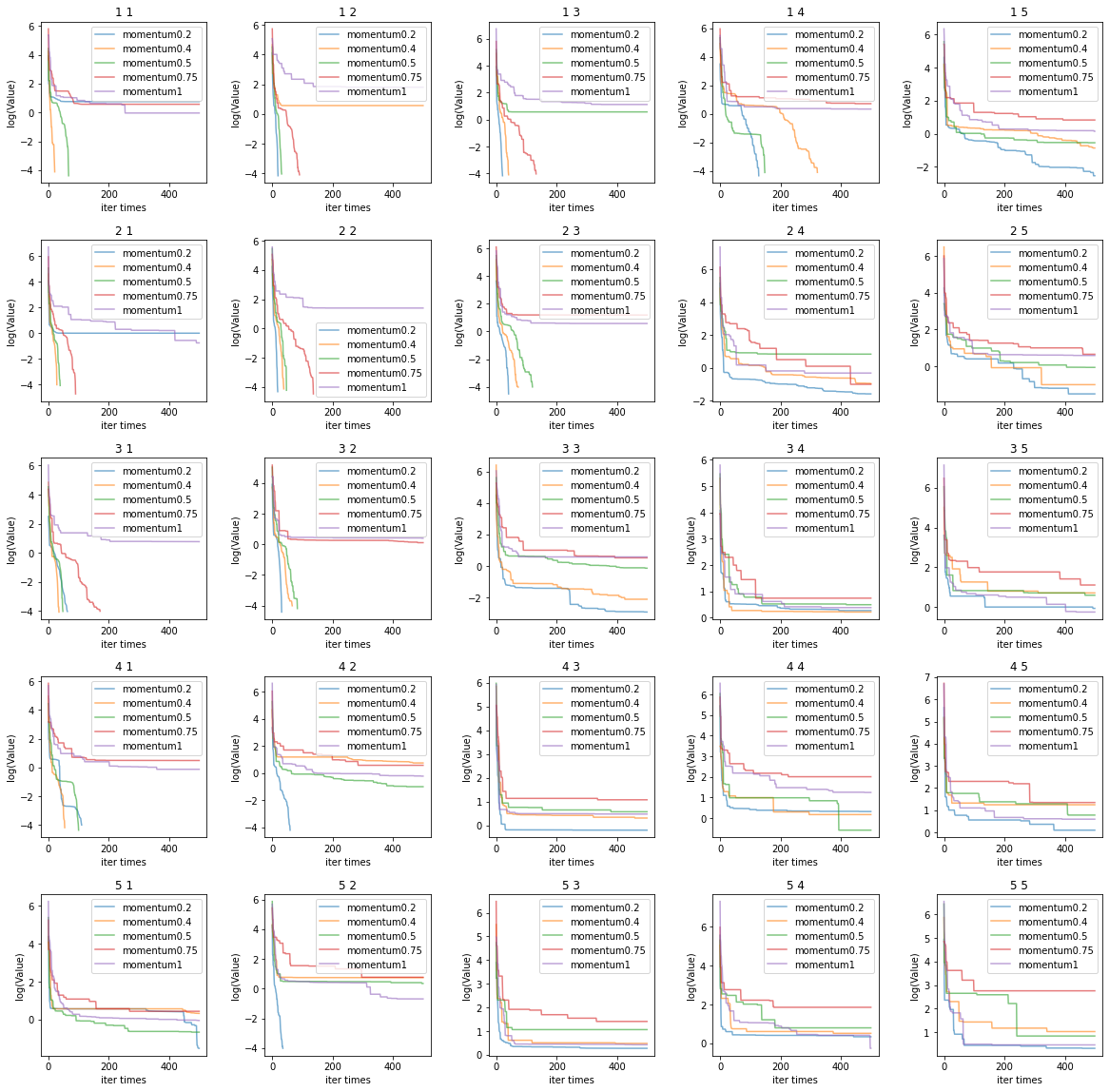
# 学习率C1,C2的探究

由于前一部分实验浪费了过多时间，此部分实验将较为简易。

首先给出结论：

C1表示种群中单独一个个体的学习能力。C1过大时，算法中粒子对自己过度自信（参考下图纵向），也因此更容易出现破禁止。

C2表示种群向群体中当前最好粒子靠近的能力



实验过程（小图上方索引，第一个是C1的值，第二个是C2的值）

# 大作业收获

算法实现尽量使用类封装，保持超参数生存周期和安全性。否则后期大规模实验会出奇怪的bug。

多变量进行对比实验时，要严格进行控制变量。

涉及到随机数的算法都要多次运行，否则会出现小概率事件导致的谬误。尤其涉及到初值依赖的算法，还需要固定随机数种子。

算法复现过程中，先实现最基础的功能，再根据算法效果逐步提升性能。

实验过程中参考了一些开源代码写法，较以前提升较为明显。

实验过程、实验设置的对照、消融等方法、实验的改进、分析思路均来自深度学习，我发现算法之间存在诸多共性。比如多参数、多效果时选取参数的trade-off；学习率类比C1、C2、W的情况等等。