遺伝的アルゴリズムと粒子群最適化の実験レポート

氏名：文翊涵（ぶん　よく　かん）　学習番号：201894008

1　遺伝的アルゴリズム

1.1　ソースコード

1.1.1　Utils Toolkit

「Utils.py」

```
import numpy as np


POP_SIZE = 100  #群衆の数
N_GENERATIONS = 100  #Iteration の数
X_MAX = 15 #解答空間の最大値
NUM_GENERATIONS = 100  #迭代次数
BEST_INDIVIDAUL = []
#最適の個体の記述
INTENSIFICATION = 5 # 最適解の倍数を強化
def Fun(x1, x2, x3):#関数
    return 2*x1**2-3*x2**2-4*x1+5*x2+x3
def clamp(n, minn, maxn):# 独立変数の値の範囲を限定すること
    n = np.maximum(n, minn)
    n = np.minimum(n, maxn)
    return n
```

1.1.2　適応度の最大化

「GA_MAX.py」

```
import matplotlib.pyplot as plt
import numpy as np
import utils


DNA_LENGTH = 4  #遺伝子の長さ
POP_NUMBERT = 100  #群の数
SINGEL_POINT_CROSSOVER_RATE = 0.8 #交叉確率
MUTATION_RATE = 0.1 #変異確率
NUM_GENERATIONS = 100  #迭代次数
BEST_INDIVIDAUL = []
```

```
#最適の個体の記述
INTENSIFICATION = 5 # 最適解の倍数を強化
class Genetic_ALG(object):#遺伝的アルゴリズムのクラス化
    def __init__(self, POP_NUMBERT=POP_NUMBERT, DAN_SIZE=DNA_LENGTH,
SINGEL_POINT_CROSSOVER_RATE=SINGEL_POINT_CROSSOVER_RATE,MUTATION_RATE=MUTATION_RATE):
        self.dim = 3  # 搜索空间的维度
        self.DNA_LENGTH=DAN_SIZE #遺伝子の長さ
        self.POP_NUMBERT=POP_NUMBERT #群の数
        self.SINGEL_POINT_CROSSOVER_RATE=SINGEL_POINT_CROSSOVER_RATE#交叉確率
        self.MUTATION_RATE=MUTATION_RATE#変異確率
    def Process(self,POPL):#アルゴリズムの繰り返し
        fitness = self.get_fitness(POPL)
        for _ in range(NUM_GENERATIONS):
            POPL = np.array(self.SINGLE_POINT_CROSSOVER_MUTATION(POPL))
            fitness = self.get_fitness(POPL)
            POPL = self.select(POPL, fitness) #新たな群の選択
        print(max(BEST_INDIVIDAUL))
    def DNA_2_TO_10(self,POPL):#DNA を 2 進法から 10 進法へ変換
        x1_pop = POPL[:,0:4]
        x1 = x1_pop.dot(2**np.arange(self.DNA_LENGTH))
        #X1 を示す
        x2_pop = POPL[:,4:8]
        x2 = x2_pop.dot(2**np.arange(self.DNA_LENGTH))
        #X2 を示す
        x3_pop = POPL[:,8:12]
        x3 = x3_pop.dot(2**np.arange(self.DNA_LENGTH))
        #X3 を示す
        return x1,x2,x3
    def SINGLE_POINT_CROSSOVER_MUTATION(self,POPL):
        Updated_Individual = []
        for P1 in POPL:
            #群のトラバーサル
            child = P1      #子供はまずＰ1 の DNA を獲得し
            if np.random.rand() < self.SINGEL_POINT_CROSSOVER_RATE:         #ある確率で交叉の発生
                P2 = POPL[np.random.randint(POP_NUMBERT)]   #群衆中にＰ2 の選択
                cross_points = np.random.randint(low=0, high=DNA_LENGTH*3)  #交叉の位置の選択
                child[cross_points:] = P2[cross_points:]        #交叉

            if np.random.rand() < MUTATION_RATE:                #ある確率で変異の発生
                mutate_point = np.random.randint(0, DNA_LENGTH) #
                child[mutate_point] = child[mutate_point]^1     #変異点の 2 進数値の変更
            Updated_Individual.append(child)
        return Updated_Individual
    def get_fitness(self,POPL):   #適合性の算出
```

```
        x1,x2,x3 = self.DNA_2_TO_10(POPL)
        fitness = self.Fun(x1, x2, x3)
        BEST_INDIVIDAUL.append(fitness.max()) #最適解の記録
        fitness[np.argmax(fitness)] *= INTENSIFICATION #最適解の倍数を強化
        return (fitness - np.min(fitness)) + 1e-3 #下限値の設定
    def select(self, POPL, fitness):
        idx = np.random.choice(np.arange(POP_NUMBERT), size=POP_NUMBERT,
replace=True,p=(fitness)/(fitness.sum())) #ルーレットで DNA の選択
        return POPL[idx]

    def Fun(self, x1, x2, x3):#適応度の関数
        return 2*x1**2-3*x2**2-4*x1+5*x2+x3
if __name__ == "__main__":
    POPL = np.random.randint(2, size=(POP_NUMBERT, DNA_LENGTH*3))
    AG=Genetic_ALG()
    a=AG.Process(POPL)
    #绘制最优解
    plt.plot(BEST_INDIVIDAUL)
    plt.show()
```

1.1.3　適応度の最小化

「GA_MIN.py」

```
import matplotlib.pyplot as plt
import numpy as np
import utils

DNA_LENGTH = 4  #遺伝子の長さ
POP_NUMBERT = 100  #群の数
SINGEL_POINT_CROSSOVER_RATE = 0.8 #交叉確率
MUTATION_RATE = 0.1 #変異確率
NUM_GENERATIONS = 100  #迭代次数
BEST_INDIVIDAUL = []
#最適の個体の記述
INTENSIFICATION = 5 # 最適解の倍数を強化
class Genetic_ALG(object):#遺伝的アルゴリズムのクラス化
    def __init__(self, POP_NUMBERT=POP_NUMBERT, DAN_SIZE=DNA_LENGTH,
SINGEL_POINT_CROSSOVER_RATE=SINGEL_POINT_CROSSOVER_RATE,MUTATION_RATE=MUTATION_RATE):
        self.dim = 3  # 搜索空间的维度
        self.DNA_LENGTH=DAN_SIZE #遺伝子の長さ
        self.POP_NUMBERT=POP_NUMBERT #群の数
        self.SINGEL_POINT_CROSSOVER_RATE=SINGEL_POINT_CROSSOVER_RATE#交叉確率
        self.MUTATION_RATE=MUTATION_RATE#変異確率
```

```
    def Process(self,POPL):#アルゴリズムの繰り返し
        fitness = self.get_fitness(POPL)
        for _ in range(NUM_GENERATIONS):
            Ppop = np.array(self.SINGLE_POINT_CROSSOVER_MUTATION(POPL,
SINGEL_POINT_CROSSOVER_RATE))
            fitness = self.get_fitness(POPL)
            POPL = self.select(POPL, fitness) # 新世代
        print("best answer is:",min(BEST_INDIVIDAUL))
    def DNA_2_TO_10(self,POPL):#DNA を 2 進法から 10 進法へ変換
        x1_pop = POPL[:,::3] # 3n 列は x1 を示す
        x2_pop = POPL[:,1::3] # 3n+1 列は x2 を示す
        x3_pop = POPL[:,2::3] # 3n+2 列は x3 を示す
        x1 = x1_pop.dot(2**np.arange(DNA_LENGTH-1,-1,-1))
        x2 = x2_pop.dot(2**np.arange(DNA_LENGTH-1,-1,-1))
        x3 = x3_pop.dot(2**np.arange(DNA_LENGTH-1,-1,-1))
        return x1,x2,x3

    def SINGLE_POINT_CROSSOVER_MUTATION(self,POPL,SINGEL_POINT_CROSSOVER_RATE):
        new_pop = []
        FATHER = POPL[::2]
        MOTHER = POPL[1::2]
        for father, mother in zip(FATHER, MOTHER):
            child1 = father #子供 1 は父親の遺伝子をすべてコピーする
            child2 = mother #子供 2 は母親の遺伝子をすべてコピーする
            if np.random.rand() < SINGEL_POINT_CROSSOVER_RATE: #CROSSOVER_RATE の確率による交叉
                cross_points = np.random.randint(low=0, high=DNA_LENGTH*3)
        #1 点交叉 ランダムに選んだ
                child1[cross_points:] = mother[cross_points:] #交叉点の前後で遺伝子を入れ替える
                child2[:cross_points] = father[:cross_points] #交叉点の前後で遺伝子を入れ替える
            self.mutation(child1) #各子孫は一定の確率で突然変異を起こす
            self.mutation(child2)

            new_pop.append(child1)
            new_pop.append(child2)
        return new_pop
    def mutation(self,child): # 突然変異
        if np.random.rand() < MUTATION_RATE: #MUTATION_RATE の確率による変異
            mutate_point = np.random.randint(0, DNA_LENGTH) #変異させたい遺伝子の位置を表すランダム
な実数を生成する
            child[mutate_point] = child[mutate_point]^1 #ビット反転
    def get_fitness(self,POPL):   #適合性の算出
        x1,x2,x3 = self.DNA_2_TO_10(POPL)
        fitness = self.Fun(x1, x2, x3)*(-1) #適合性の算出
        BEST_INDIVIDAUL.append(fitness.max()*(-1)) #最適解の記録
```

```
        fitness[np.argmax(fitness)] *= INTENSIFICATION #最优解的强化倍数
        return (fitness - np.min(fitness)) + 1e-3 #下限値の設定
    def select(self, POPL, fitness):
        idx = np.random.choice(np.arange(POPL_NUMBERT), size=POPL_NUMBERT,
replace=True,p=(fitness)/(fitness.sum()))
# このステップでは、新世代の個体をフィルタリングします。序列番号のみをフィルタリングし、確率によって、以
前と同じサイズの個体の選択を繰り返すことができます。
        return POPL[idx]
# 上記は序数なので、今度はそれを実際の遺伝子表に取り込み、データを返します。データの一行は、x1,x2,x3 の
集合を表しているので、個人に直接対応しています。

    def Fun(self, x1, x2, x3):#関数
        return 2*x1**2-3*x2**2-4*x1+5*x2+x3

if __name__ == "__main__":
    POPL = np.random.randint(2, size=(POPL_NUMBERT, DNA_LENGTH*3))
    AG=Genetic_ALG()
    a=AG.Process(POPL)
    #绘制最优解
    plt.plot(BEST_INDIVIDAUL)
    plt.show()
```

1.2　GA アルゴリズムの説明

遺伝的アルゴリズムとは、遺伝学と自然淘汰の原理に基づいた探索型の最適化手法で、最適解やそれに近い解を見つけ出すためによく使われます。

1.2.1 GA の仕組み

図１に示すように、いくつかの個体が集まって集団を構成しているとします。これを世代ｔの集団とします。個体にはそれぞれ適合度が決まっています。これらの個体は生殖活動（recombination、reproduction）を行い、次の世代ｔ＋１の子孫を作り出します。生殖の際には、適合度の高いものほど子孫をより多くつくるように、適合度の低いものほど死にやすいようにします（選択もしくは淘汰）。その結果、世代ｔ＋１の各個体の適合度は、世代ｔのそれよりも良いことが期待され、集団全体を見たときの適合度が上がっているはずです。同様に、世代ｔ＋１、ｔ＋２、…と繰り返していくと世代が上がるにつれ、次第に集団全体が良くなっていく、というのがＧＡの基本的な考え方となります。

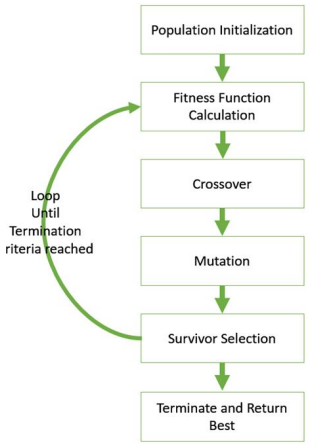

図 1

1.2.2 GA における選択

1.2.2.1 ルーレット選択方式

合度に比例した割合で個体を選択する方式である。これを実現する一番単純な方法は、適合度に比例した面積を有するルーレットをつくり、それを回して当たった場所の個体を選択するという方法である。番目の個体の確率は：

$$p_i = w(\frac{f_i}{\sum_{k=1}^{N} f_k}) \qquad (1)$$

```
def select(self, pop, fitness):
    idx = np.random.choice(np.arange(POP_SIZE), size=POP_SIZE, replace=True,p=(fitness)/(fitness.sum()))
    return pop[idx]
```

1.2.3 GA における ONE POINT CROSSOVER

図 2 に示すように、DNA を同じ位置で切断し、前後の 2 本の糸を交差させて結合させ、新しい 2 本の染色体を形成した。 遺伝子組換え、ハイブリッド化ともいう。

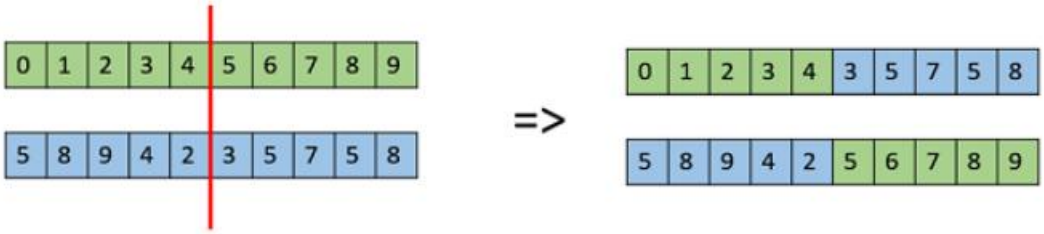

図 2

1.2.4 GA における突然変異

図 3 に示すように、このタスクでは、突然変異の確率を設定し、突然変異が発生した場合には、ランダムに選択された遺伝子セグメントに反転操作を施して子孫を得る。

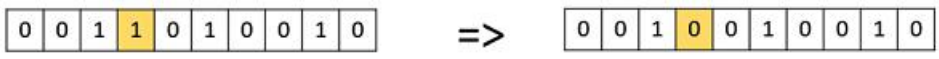

図３

```python
def SINGLE_POINT_CROSSOVER_MUTATION(self,POPL):
    Updated_Individual = []
    for P1 in POPL:
        #群のトラバーサル
        child = P1
        if np.random.rand() < self.SINGEL_POINT_CROSSOVER_RATE:
            P2 = POPL[np.random.randint(POP_NUMBERT)]
            cross_points = np.random.randint(low=0, high=DNA_LENGTH*3)
            child[cross_points:] = P2[cross_points:]
        if np.random.rand() < MUTATION_RATE:
            mutate_point = np.random.randint(0, DNA_LENGTH)
            child[mutate_point] = child[mutate_point]^1
        Updated_Individual.append(child)
    return Updated_Individual
```

1.3 GA の実験

1.3.1 パラメータと環境の設定

以下のパラメータに基づき、VSCODE と Python3.8 において、六回の実験をする。

```python
DNA_LENGTH = 4  #遺伝子の長さ
POP_NUMBERT = 100  #群の数
SINGEL_POINT_CROSSOVER_RATE = 0.8 #交叉確率
MUTATION_RATE = 0.1 #変異確率
NUM_GENERATIONS = 100  #迭代次数
BEST_INDIVIDAUL = []
#最適の個体の記述
INTENSIFICATION = 1# 最適解の倍数を強化
```

1.3.2 結果（最大化の場合）

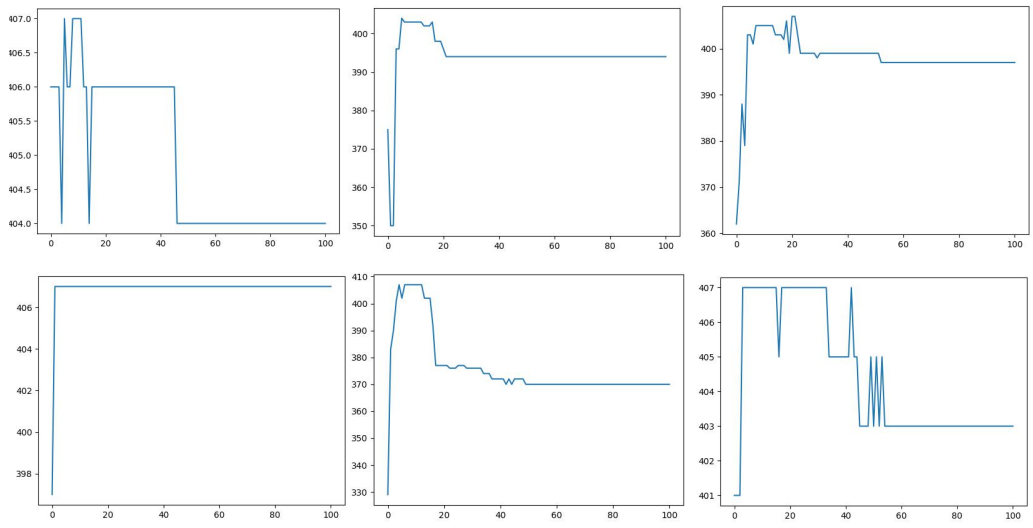

図４　最適解の倍数が 1 の GA の最大化実験

```
(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\Genetic_Algorithm.py
407

(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\Genetic_Algorithm.py
406

(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\Genetic_Algorithm.py
406

(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\Genetic_Algorithm.py
407

(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\Genetic_Algorithm.py
407

(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\Genetic_Algorithm.py
404
```

1.3.3 結果（最小化の場合）

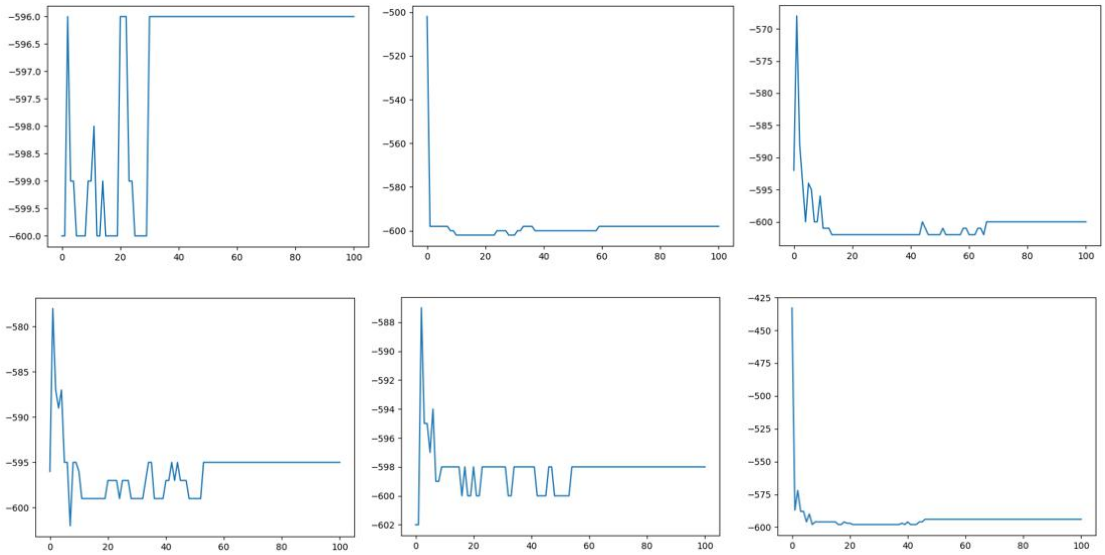

図 5　最適解の倍数が 1 の GA の最小化実験

```
(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\testmin.py
best answer is: -602

(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\testmin.py
best answer is: -594

(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\testmin.py
best answer is: -518

(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\testmin.py
best answer is: -602

(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\testmin.py
best answer is: -600

(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\testmin.py
best answer is: -602
```

1.3.4 分析

最大の 407、最小の-602 を保存するのは難しく、1.4 節で改善し、最適化する。

1.4　アルゴリズム性能の最適化

最適解の倍数を５になり、最適解が保持される確率を高めるのを実現できる。

```python
DNA_LENGTH = 4  #遺伝子の長さ
POP_NUMBERT = 100  #群の数
SINGEL_POINT_CROSSOVER_RATE = 0.8 #交叉確率
MUTATION_RATE = 0.1 #変異確率
NUM_GENERATIONS = 100  #迭代次数
BEST_INDIVIDAUL = []
#最適の個体の記述
INTENSIFICATION = 5 # 最適解の倍数を強化
```

```python
def get_fitness(self,POPL):   #適合性の算出
    x1,x2,x3 = self.DNA_2_TO_10(POPL)
    fitness = self.Fun(x1, x2, x3)*(-1) #適合性の算出
    BEST_INDIVIDAUL.append(fitness.max()*(-1)) #最適解の記録
    fitness[np.argmax(fitness)] *= INTENSIFICATION #最优解的强化倍数
    return (fitness - np.min(fitness)) + 1e-3 #下限値の設定
```

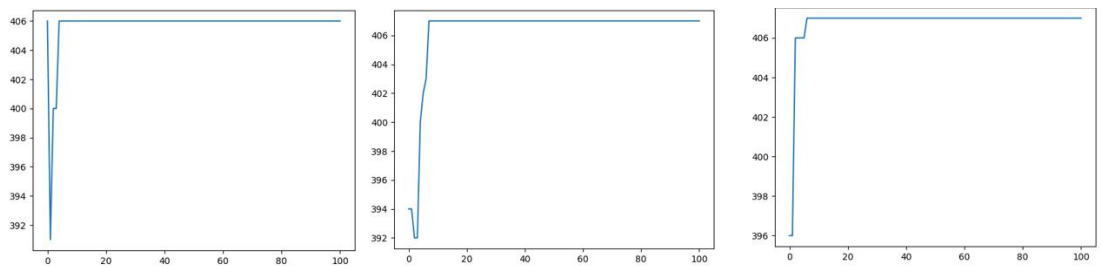

図 6　最適化した後の最大化

```
(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\test2.py
407

(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\test2.py
407

(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\test2.py
407

(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\test2.py
407

(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\test2.py
407

(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\test2.py
407
```

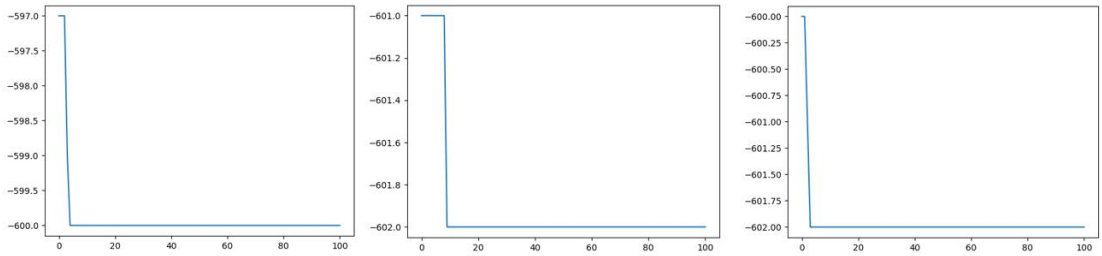

図 7　最適化した後の最小化

```
(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\testorigh.py
best answer is: -602

(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\testorigh.py
best answer is: -602

(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\testorigh.py
best answer is: -602

(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\testorigh.py
best answer is: -600

(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\testorigh.py
best answer is: -600

(my_py_torch) D:\BRDF\cui>python C:\Users\wenyihan\Desktop\code\testorigh.py
best answer is: -602
```

2　粒子群最適化の実験

2.1　ソースコード

2.1.1　Utils　Toolkit

「Utils.py」

```python
import numpy as np
POP_SIZE = 100  #每次迭代的种群总个数
NUM_GENE = 100  #迭代的总次数
X_MAX = 15 #解答空間の最大値
SPEED_INITAIL_Adjust = 10
SPEED_MIDDLE_Adjust = 5
SPEED_FINALLY_Adjust = 1
#最適の個体の記述
INTENSIFICATION = 5 # 最適解の倍数を強化
C1 = 20 #各探索個体の最良位置の学習係数
C2 = 20 #個体群全体での最良値の学習係数
def Fun(x1, x2, x3):#関数
    return 2*x1**2-3*x2**2-4*x1+5*x2+x3
def clamp(n, minn, maxn):# 独立変数の値の範囲を限定すること
    n = np.maximum(n, minn)
    n = np.minimum(n, maxn)
    return n
def adjust_update_rate(v, time): # 進化中に学習係数を調整、模倣焼きなまし法とも言える
    if(time < NUM_GENE *0.3): # 初期段階
        learning_rate_max = [[SPEED_INITAIL_Adjust]*3]*POP_SIZE
        learning_rate_min = [[SPEED_INITAIL_Adjust*(-1)]*3]*POP_SIZE
        v = np.minimum(v, learning_rate_max)
        v = np.maximum(v, learning_rate_min)
    elif(time < NUM_GENE *0.7): # 中期段階
        learning_rate_max = [[SPEED_MIDDLE_Adjust]*3]*POP_SIZE
        learning_rate_min = [[SPEED_MIDDLE_Adjust*(-1)]*3]*POP_SIZE
        v = np.minimum(v, learning_rate_max)
        v = np.maximum(v, learning_rate_min)
    else: # 後期段階
        learning_rate_max = [[SPEED_FINALLY_Adjust]*3]*POP_SIZE
        learning_rate_min = [[SPEED_FINALLY_Adjust*(-1)]*3]*POP_SIZE
        v = np.minimum(v, learning_rate_max)
        v = np.maximum(v, learning_rate_min)
    return v
def adjust_learning_rate(C1, C2, time): # 進化中に学習係数を調整
```

```
    if(time < NUM_GENE *0.3): # 初期段階
        C1 = 10
        C2 = 10
    elif(time < NUM_GENE *0.7): # 中期段階
        C1 = 2
        C2 = 2
    else: # 後期段階
        C1 = 0.5
        C2 = 0.5
    return C1,C2
```

2.1.2　適応度の最大化

「POS_MAX.py」

```
import numpy as np
import matplotlib.pyplot as plt
import utils

POP_NUM = 100 #群の個体数
NUM_GEN = 100 #世代数
C1 = 20 #各探索個体の最良位置の学習係数
C2 = 20 #個体群全体での最良値の学習係数
W = 0.6 #慣性定数
X_MIN = 0 #解答空間の最小値
X_MAX = 15 #解答空間の最大値
BEST_ANSWERS = [] #最適な結果を記録

class PSO(object): # 群
    def __init__(self, POP_SIZE=POP_NUM, max_steps=NUM_GEN):
        self.POP_SIZE = POP_SIZE # 群の個体数
        self.DIM = 3 # x1,x2,x3
        self.max_steps = max_steps # 世代数
        self.x_bound = [X_MIN, X_MAX] # 解答空間の空间的範囲

        self.x = np.random.randint(self.x_bound[0],self.x_bound[1]+1, size=(POP_NUM, self.DIM))

        self.v = np.random.rand(self.POP_SIZE, self.DIM) # 粒子群の速度の初期化
        fitness = self.get_fitness(self.x)
        self.p = self.x # 各探索個体の最良値
        self.pg = self.x[np.argmax(fitness)] # 個体群全体での最良位置
        self.individual_best_fitness = fitness # 各探索個体の最优適応度
        self.global_best_fitness = np.max(fitness) # 個体群全体での最良値
```

```
        BEST_ANSWERS.append(self.global_best_fitness) # 最適な結果を記録
    def get_fitness(self, x): # 適応度の算出
        x1 = x[:,0] # 1 列は x1
        x2 = x[:,1] # 2 列は x2
        x3 = x[:,2] # 3 列は x3
        pred = utils.Fun(x1, x2, x3)
        return pred
    def evolve(self):
        for _ in range(self.max_steps):
            r1 = np.random.rand(self.POP_SIZE, self.DIM)
            r2 = np.random.rand(self.POP_SIZE, self.DIM)
            # 更新速度と重み付け
            self.v = W*self.v+C1*r1*(self.p-self.x)+C2*r2*(self.pg-self.x) # 個体の速度を計算する
            self.v = utils.adjust_update_rate(self.v, _) # 段階的速度制限
            minn = [[self.x_bound[0]]*self.DIM]*self.POP_SIZE
            maxn = [[self.x_bound[1]]*self.DIM]*self.POP_SIZE
            if _ == 0:
                fitness = self.get_fitness(self.x)
            best_fitness_place_value = self.x[np.argmax(fitness)] # 最良位置の値
            best_fitness_place = np.argmax(fitness) # 最良位置のナンバリング
            self.x = utils.clamp(self.v + self.x, minn, maxn) # 粒子の位置を更新する

            self.x[best_fitness_place] = best_fitness_place_value # 保留最优解
            fitness = self.get_fitness(self.x)# 個体の更新
            update_id = np.greater(self.individual_best_fitness, fitness)
            self.p[update_id] = self.x[update_id]
            self.individual_best_fitness[update_id] = fitness[update_id] # 個体群全体での最良値の
更新
            if np.max(fitness) > self.global_best_fitness:
                self.pg = self.x[np.argmax(fitness)]
                self.global_best_fitness = np.max(fitness)
            BEST_ANSWERS.append(np.max(fitness))
pso = PSO()
pso.evolve()
print("MAX is:",max(BEST_ANSWERS))
plt.plot(BEST_ANSWERS)
plt.show()
```

2.1.2　適応度の最小化

「POS_MIN.py」

```
import numpy as np
```

```python
import matplotlib.pyplot as plt
import utils

POP_NUM = 100 #群の個体数
NUM_GEN = 100 #世代数
C1 = 20 #各探索個体の最良位置の学習係数
C2 = 20 #個体群全体での最良値の学習係数
W = 0.6 #慣性定数
X_MIN = 0 #解答空間の最小値
X_MAX = 15 #解答空間の最大値
BEST_ANSWERS = [] #最適な結果を記録

class PSO(object): # 群
    def __init__(self, POP_SIZE=POP_NUM, max_steps=NUM_GEN,C1=C1,C2=C2):
        self.POP_SIZE = POP_SIZE # 群の個体数
        self.DIM = 3 # x1,x2,x3
        self.max_steps = max_steps # 世代数
        self.x_bound = [X_MIN, X_MAX] # 解答空間の空间的範囲
        self.C1=C1
        self.C2=C2
        self.x = np.random.randint(self.x_bound[0],self.x_bound[1]+1, size=(POP_NUM, self.DIM))

        self.v = np.random.rand(self.POP_SIZE, self.DIM) # 粒子群の速度の初期化
        fitness = self.get_fitness(self.x)
        self.p = self.x # 各探索個体の最良値
        self.pg = self.x[np.argmax(fitness)] # 個体群全体での最良位置
        self.individual_best_fitness = fitness # 各探索個体の最优適応度
        self.global_best_fitness = np.max(fitness) # 個体群全体での最良値
        BEST_ANSWERS.append(self.global_best_fitness) # 最適な結果を記録
    def get_fitness(self, x): # 適応度の算出
        x1 = x[:,0] # 1 列は x1
        x2 = x[:,1] # 2 列は x2
        x3 = x[:,2] # 3 列は x3
        pred = utils.Fun(x1, x2, x3)
        return pred
    def evolve(self):
        for _ in range(self.max_steps):
            r1 = np.random.rand(self.POP_SIZE, self.DIM)
            r2 = np.random.rand(self.POP_SIZE, self.DIM)
            # 更新速度と重み付け
            CA,CB=utils.adjust_learning_rate(self.C1,self.C2,_)
            self.v = W*self.v+CA*r1*(self.p-self.x)+CB*r2*(self.pg-self.x) # 個体の速度を計算する
            self.v = utils.adjust_update_rate(self.v, _) # 段階的速度制限
```

```
            minn = [[self.x_bound[0]]*self.DIM]*self.POP_SIZE
            maxn = [[self.x_bound[1]]*self.DIM]*self.POP_SIZE
            if _ == 0:
                fitness = self.get_fitness(self.x)
            best_fitness_place_value = self.x[np.argmax(fitness)] # 最良位置の値
            best_fitness_place = np.argmax(fitness) # 最良位置のナンバリング
            self.x = utils.clamp(self.v + self.x, minn, maxn) # 粒子の位置を更新する

            self.x[best_fitness_place] = best_fitness_place_value # 保留最优解
            fitness = self.get_fitness(self.x)# 個体の更新
            update_id = np.greater(self.individual_best_fitness, fitness)
            self.p[update_id] = self.x[update_id]
            self.individual_best_fitness[update_id] = fitness[update_id] # 個体群全体での最良値の
更新
            if np.max(fitness) > self.global_best_fitness:
                self.pg = self.x[np.argmin(fitness)]
                self.global_best_fitness = np.min(fitness)
            BEST_ANSWERS.append(np.min(fitness))
pso = PSO()
pso.evolve()
print("MIN is:",min(BEST_ANSWERS))
plt.plot(BEST_ANSWERS)
plt.show()
plt.show()
```

2.2　POS アルゴリズムの説明

PSO において群れを構成する各個体は，現在の“位置（状態量）”とそのときの“速度”の情報を持っている。この位置と速度の情報から次世代の各個体の位置を更新する。目的関数 f（x）は変数である状態量 x によって一意に決定されるものとする。各個体はそれぞれの個体の最良位置 pbest （personal best）と群れ全体の最良位置 gbest （global best）に近づく方向に(2)式で速度を修正しながら現在位置（探索点）を(３)式により更新する。PSO における探索点の修正方法を図 8 に示す。

$$v = wv + c_1 r_1 (G_{best} - x) + c_2 r_2 (P_{best} - x) \quad (2)$$

$$x = x + v \quad (3)$$

```
self.v = W*self.v+C1*r1*(self.p-self.x)+C2*r2*(self.pg-self.x) # 個体の速度を計算する
self.v = utils.adjust_learning_rate(self.v, _) # 段階的速度制限
```

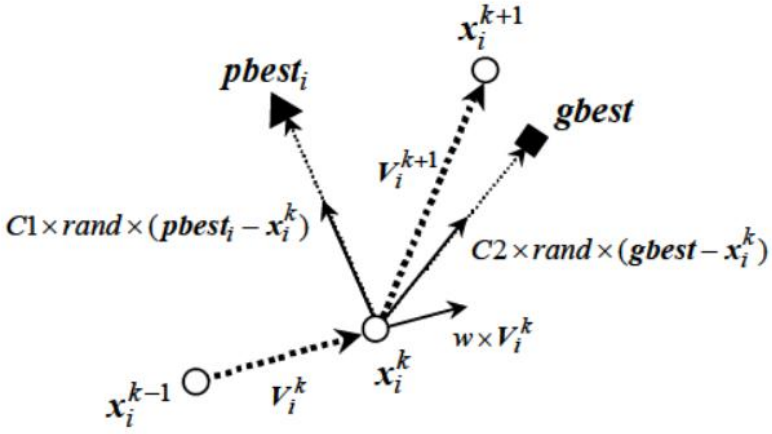

図 8　探索点の修正

2.3　GA の実験

2.3.1 パラメータと環境の設定

以下のパラメータに基づき、VSCODE と Python3.8 において、六回の実験をする。

```
POP_NUM = 100 #群の個体数
NUM_GEN = 100 #世代数
C1 = 20 #各探索個体の最良位置の学習係数
C2 = 20 #個体群全体での最良値の学習係数
W = 0.6 #慣性定数
X_MIN = 0 #解答空間の最小値
X_MAX = 15 #解答空間の最大値
BEST_ANSWERS = [] #最適な結果を記録
```

2.3.2 結果

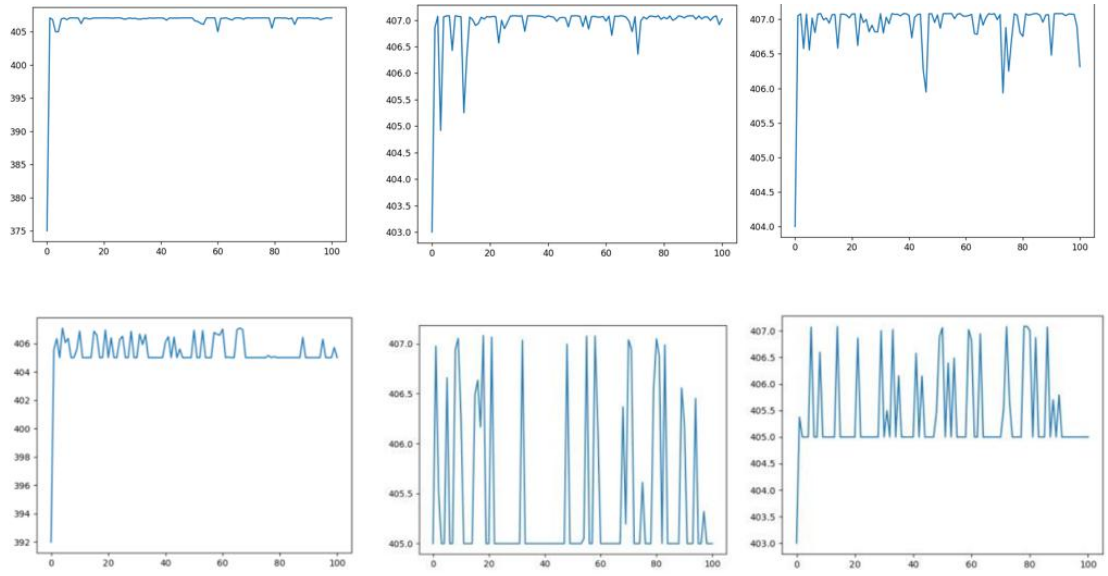

図 9　POS の最大化実験

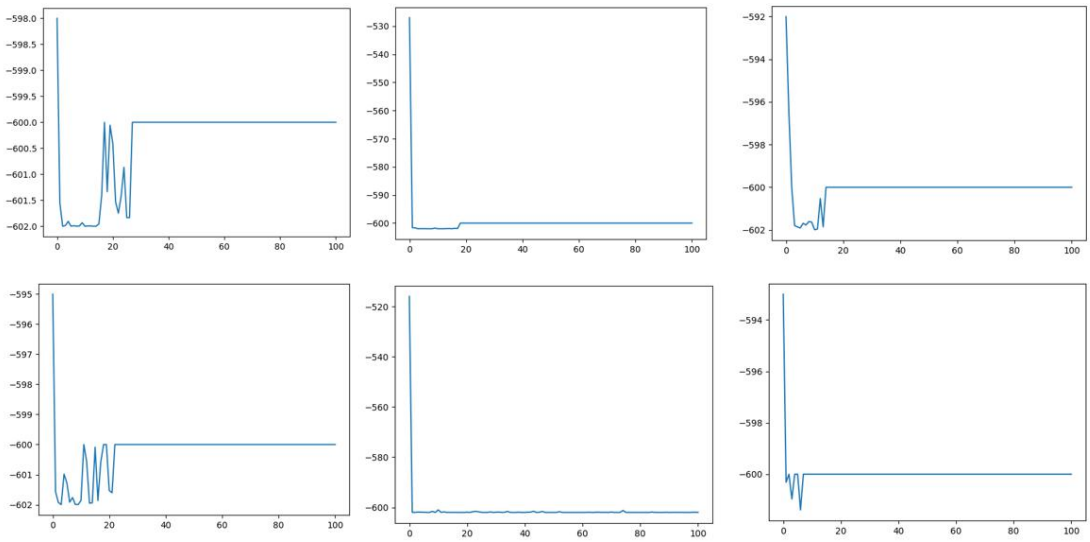

図 10　POS の最小化実験

2.4 分析

最大の 407、最小の-602 を保存できるが、安定性に欠けている。2.5 節で、更新係数の調整を通し、問題を改善する。

2.5　進化中に更新係数の調整

今回の調査では、各探索個体の最良位置個体群と全体での最良位置は、更新係数と学習係数に大きく影響されている。

```
def adjust_update_rate(v, time): # 進化中に更新係数を調整する
    if(time < NUM_GENE *0.3):
        learning_rate_max = [[SPEED_INITAIL_Adjust]*3]*POP_SIZE
        learning_rate_min = [[SPEED_INITAIL_Adjust*(-1)]*3]*POP_SIZE
        v = np.minimum(v, learning_rate_max)
        v = np.maximum(v, learning_rate_min)
    elif(time < NUM_GENE *0.7):
        learning_rate_max = [[SPEED_MIDDLE_Adjust]*3]*POP_SIZE
        learning_rate_min = [[SPEED_MIDDLE_Adjust*(-1)]*3]*POP_SIZE
        v = np.minimum(v, learning_rate_max)
        v = np.maximum(v, learning_rate_min)
    else:
        learning_rate_max = [[SPEED_FINALLY_Adjust]*3]*POP_SIZE
        learning_rate_min = [[SPEED_FINALLY_Adjust*(-1)]*3]*POP_SIZE
        v = np.minimum(v, learning_rate_max)
        v = np.maximum(v, learning_rate_min)
    return v
```

```python
def adjust_learning_rate(C1, C2, time): # 進化中に学習係数を調整

    if(time < NUM_GENE *0.3): # 初期段階
        C1 = 10
        C2 = 10
    elif(time < NUM_GENE *0.7): # 中期段階
        C1 = 2
        C2 = 2
    else: # 後期段階
        C1 = 0.5
        C2 = 0.5
    return C1,C2
```

```python
# 更新速度と重み付け
CA,CB=utils.adjust_learning_rate(self.C1,self.C2,_)
self.v = W*self.v+CA*r1*(self.p-self.x)+CB*r2*(self.pg-self.x) # 個体の速度を計算する
self.v = utils.adjust_update_rate(self.v, _) # 段階的速度制限
```

それらの調整を通し、図 10 に示すように、素晴らしい効果が実現できる。

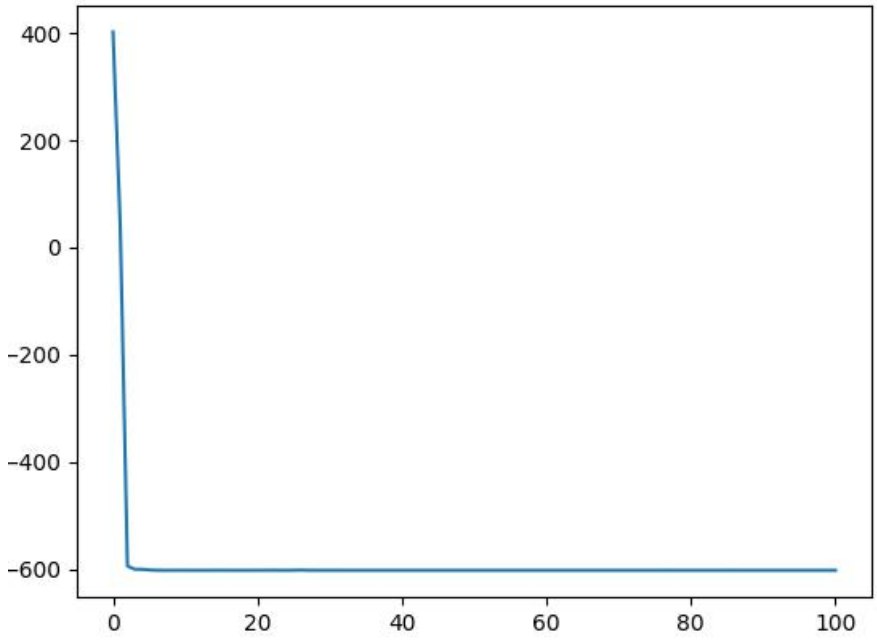

図 10　POS を最適化するたとの最小化実験

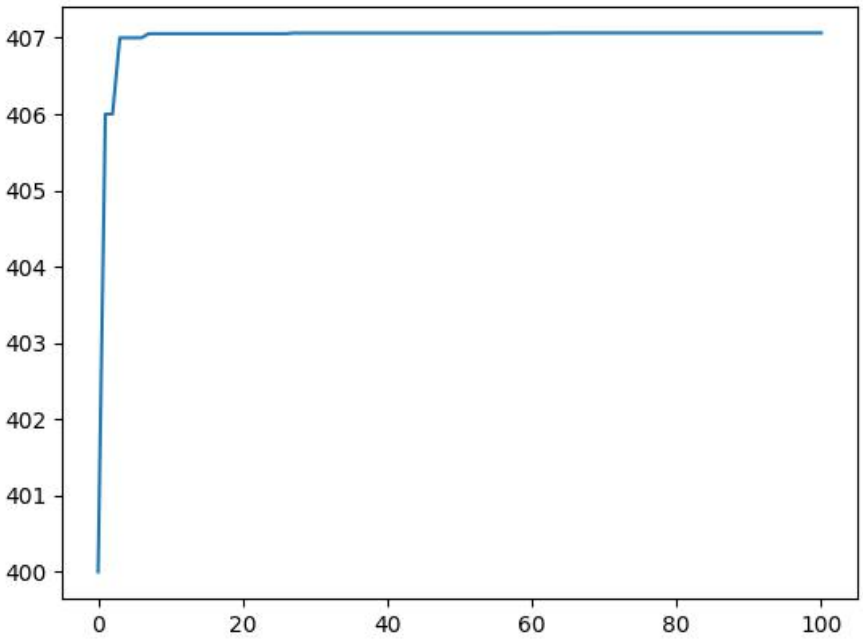

図 11　POS を最適化するたとの最大化実験