МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

АЭРОКОСМИЧЕСКОГО ПРИБО6РОСТРОЕНИЯ»

КАФЕДРА № 43

ОТЧЕТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Старший преподаватель |  |  |  | С.А. Рогачев |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

|  |
| --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ |
| «Оптимизация многомерных функций с помощью эволюционной стратегии» |
| по курсу: ЭВОЛЮЦИОННЫЕ МЕТОДЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНО-ИНФОРМАЦИОННЫХ СИСТЕМ |
|  |
|  |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | 4136 |  | 15.11.24 |  | А.С. Вилюмсон |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2024

**1. Задание по варианту**

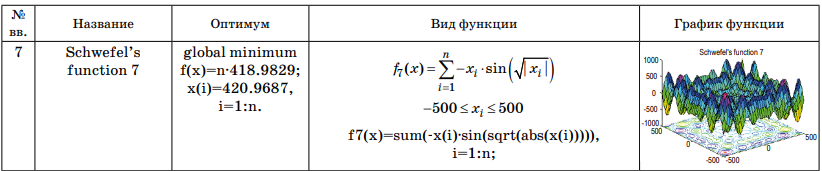
1. Создать программу, использующую ЭС для нахождения оптимума функции согласно таблице вариантов, приведенной в приложении А. Для всех Benchmark-ов оптимумом является минимум. Программу выполнить на встроенном языке пакета Matlab (или любом, доступным вам, языке программирования ).

2. Для n=2 вывести на экран график данной функции с указанием найденного экстремума, точек популяции. Для вывода графиков использовать стандартные возможности пакета Matlab. Предусмотреть возможность пошагового просмотра процесса поиска решения.

3. Исследовать зависимость времени поиска, числа поколений (генераций), точности нахождения решения от основных параметров генетического алгоритма: - число особей в популяции - вероятность мутации. Критерий остановки вычислений – повторение лучшего результата заданное количество раз или достижение популяцией определенного возраста (например, 100 эпох).

4. Повторить процесс поиска решения для n=3, сравнить результаты, скорость работы программы.

***Вариант 7***



**2. Основные сведения из теории**

Эволюционные стратегии (ЭС), также как и предыдущие парадигмы, основаны на эволюции популяции потенциальных решений, но, в отличие от них, здесь используется генетические операторы на уровне фенотипа, а не генотипа, как это делается в ГА. Разница в том, что ГА работают в пространстве генотипа – кодов решений, в то время как ЭС производят поиск в пространстве фенотипа – векторном пространстве вещественных чисел. В ЭС учитываются свойства хромосомы «в целом», в отличие от ГА, где при поиске решений исследуются отдельные гены. В природе один ген может одновременно влиять на несколько свойств организма. С другой стороны одно свойство особи может определяться несколькими генами. Естественная эволюция основана на исследовании совокупности генов, а не отдельного (изолированного) гена. В эволюционных стратегиях целью является движение особей популяции по направлению к лучшей области ландшафта фитнесс-функции.

**3. Листинг программы**

from \_\_future\_\_ import annotations  
  
import copy  
import random  
import math  
import matplotlib  
  
from lab4.main import crossover  
  
matplotlib.use('TkAgg')  
successful\_mutations = 0  
total\_mutations = 0  
class Individual:  
 def \_\_init\_\_(self, size=2, phenotype=None, std\_deviation=None):  
 if phenotype == None:  
 self.phenotype = [0] \* size  
 else:  
 self.phenotype = phenotype  
 size = len(phenotype)  
 if std\_deviation == None:  
 self.std\_deviation = [0] \* size  
 else:  
 self.std\_deviation = std\_deviation  
 size = len(phenotype)  
 self.fitness = 0  
  
 def mutation(self):  
  
 for i in range(len(self.phenotype)):  
 mutagene = random.gauss(0, self.std\_deviation[i])  
 self.phenotype[i] += mutagene  
 if self.phenotype[i] > 500:  
 self.phenotype[i] = 500  
 if self.phenotype[i] < -500:  
 self.phenotype[i] = -500  
  
  
 def crossover(self, other: Individual, strategy="1+1") -> list[Individual] | Individual | None:  
 child = Individual(size=len(other.phenotype))  
 match strategy:  
 case "1+1":  
 return None  
 case "m+1" | "m+n":  
  
 #choices = random.sample(range(0, 1), len(self.phenotype))  
 for idx in range(len(self.phenotype)):  
 parent\_idx = random.randint(0,1)  
 if parent\_idx == 0:  
 child.phenotype[idx] = self.phenotype[idx]  
 child.std\_deviation[idx] = self.std\_deviation[idx]  
 else:  
 child.phenotype[idx] = other.phenotype[idx]  
 child.std\_deviation[idx] = other.std\_deviation[idx]  
 return child  
  
  
def function(x\_list: list[float], func: str) -> float:  
 result = 0  
 for xi in x\_list:  
 result += (eval(func))  
 return result  
  
def update\_deviation(population: list[Individual], successful\_mutations, total\_mutations):  
 if successful\_mutations/total\_mutations < 0.2:  
 for i in range(len(population[0].std\_deviation)):  
 population[0].std\_deviation[i] \*= 0.82  
 elif successful\_mutations/total\_mutations > 0.2:  
 for i in range(len(population[0].std\_deviation)):  
 population[0].std\_deviation[i] \*= 1.22  
 else:  
 pass  
 return population  
def genetic\_algo\_cycle(population: list[Individual], strategy="1+1", extr="max", func="1+1",crossover\_chance=1.0, mutation\_chance=0.4):  
 pop\_size = len(population)  
 for individual in population:  
 individual.fitness = function(individual.phenotype, func)  
 match strategy:  
 case "1+1":  
 global successful\_mutations  
 global total\_mutations  
  
 new\_population = copy.deepcopy(population)  
 new\_population[0].mutation()  
  
 population[0].fitness = function(population[0].phenotype, func)  
 new\_population[0].fitness = function(new\_population[0].phenotype, func)  
 if new\_population[0].fitness < population[0].fitness:  
 population = new\_population  
 successful\_mutations += 1  
 pass  
 total\_mutations += 1  
 population = update\_deviation(population, successful\_mutations, total\_mutations)  
 case "m+1":  
 copy\_population = copy.deepcopy(population)  
 new\_population = []  
 for i in range(len(population)//2):  
 parent1 = copy\_population.pop(random.randint(0, len(copy\_population)-1))  
 parent2 = copy\_population.pop(random.randint(0, len(copy\_population)-1))  
 if random.random() < crossover\_chance:  
 child = parent1.crossover(parent2, strategy=strategy)  
 else:  
 child = parent1  
 if random.random() < mutation\_chance:  
 child.mutation()  
  
 child.fitness = function(child.phenotype, func)  
  
 new\_population.append(child)  
  
 population.extend(new\_population)  
 sorted\_population = sorted(population,key= lambda ind:ind.fitness)  
 population = copy.deepcopy(sorted\_population[:pop\_size])  
 pass  
 case "m+n":  
 copy\_population = copy.deepcopy(population)  
 child\_pop\_size = 2\*7 #n in (m+n)  
 new\_population = []  
 for i in range(len(population) // 2):  
 parent1 = copy\_population.pop(random.randint(0, len(copy\_population) - 1))  
 parent2 = copy\_population.pop(random.randint(0, len(copy\_population) - 1))  
 for \_ in range(child\_pop\_size):  
 if random.random() < crossover\_chance:  
 child = parent1.crossover(parent2, strategy=strategy)  
 else:  
 child = parent1  
 if random.random() < mutation\_chance:  
 child.mutation()  
  
  
 child.fitness = function(child.phenotype, func)  
  
 new\_population.append(child)  
  
 population.extend(new\_population)  
 sorted\_population = sorted(population, key=lambda ind: ind.fitness)  
 population = copy.deepcopy(sorted\_population[:pop\_size])  
 pass  
 case "m,n":  
 pass  
  
 return population  
  
  
def genetic\_algorithm\_init(population\_size=20, strategy="1+1", gene\_size=2, bounds=None, extr="max"):  
 if bounds is None:  
 bounds = [-500, 500]  
 if strategy == "1+1":  
 population\_size = 1  
 population = []  
 for \_ in range(population\_size):  
 phenotype = random.sample(range(bounds[0], bounds[1]), gene\_size)  
 std\_deviation = [1.0] \* gene\_size  
 individual = Individual(phenotype=phenotype, std\_deviation=std\_deviation)  
 population.append(individual)  
 return population  
  
  
import matplotlib.pyplot as plt  
from mpl\_toolkits.mplot3d import Axes3D  
import numpy as np  
  
  
def plot\_function\_and\_population(func: str, population: list[Individual], bounds: list[int]):  
 *"""  
 Построить график функции и решений из текущей популяции.  
 Позволяет интерактивное вращение графика.  
 :param func: Строка с функцией для вычисления z.  
 :param population: Список объектов Individual.  
 :param bounds: Границы графика [min, max].  
 """* # Создание сетки для функции  
 x = np.linspace(bounds[0], bounds[1], 100)  
 y = np.linspace(bounds[0], bounds[1], 100)  
 X, Y = np.meshgrid(x, y)  
 Z = np.zeros\_like(X)  
  
 # Вычисление значений функции  
 for i in range(X.shape[0]):  
 for j in range(X.shape[1]):  
 xi = X[i, j]  
 yi = Y[i, j]  
 Z[i, j] = function([xi, yi], func)  
  
 # Создание графика  
 fig = plt.figure(figsize=(10, 7))  
 ax = fig.add\_subplot(111, projection='3d')  
 ax.plot\_surface(X, Y, Z, cmap='viridis', alpha=0.7)  
  
 # Добавление точек популяции  
 for individual in population:  
 x\_val, y\_val = individual.phenotype  
 z\_val = function([x\_val, y\_val], func)  
 ax.scatter(x\_val, y\_val, z\_val, color='red', s=50)  
  
 ax.set\_title("3D Plot of Function and Population")  
 ax.set\_xlabel("X")  
 ax.set\_ylabel("Y")  
 ax.set\_zlabel("Z")  
  
 # Включение интерактивного режима  
 plt.ion() # Включить интерактивный режим  
 plt.show(block=True) # Блокирует выполнение, пока пользователь не закроет график  
  
def test\_mutation\_crossover\_accuracy(  
 func: str,  
 bounds: list[int],  
 population\_size: int,  
 gene\_size: int,  
 max\_generations: int,  
 mutation\_probs: list[float],  
 crossover\_probs: list[float],  
 num\_repeats: int = 3,  
 target\_accuracy: float = 1e-4,  
 strategy = "m+1"  
):  
 *"""  
 Тестирует достижение заданной точности для разных вероятностей мутации и кроссинговера с усреднением.  
 :param func: Строка с функцией для оптимизации.  
 :param bounds: Границы значений генов.  
 :param population\_size: Размер популяции.  
 :param gene\_size: Количество генов в индивиде.  
 :param max\_generations: Максимальное число поколений.  
 :param mutation\_probs: Список вероятностей мутации.  
 :param crossover\_probs: Список вероятностей кроссинговера.  
 :param num\_repeats: Число повторов для усреднения (по умолчанию 3).  
 :param target\_accuracy: Целевая точность (по умолчанию 1e-4).  
 """* results = []  
  
 for mutation\_prob in mutation\_probs:  
 for crossover\_prob in crossover\_probs:  
 total\_generations = 0  
 total\_best\_fitness = 0  
 success\_count = 0  
 print(f"mutation\_prob = {mutation\_prob}, crossover\_prob = {crossover\_prob}")  
 for \_ in range(num\_repeats):  
 population = genetic\_algorithm\_init(population\_size, gene\_size=gene\_size,strategy=strategy)  
 success = False  
  
 for generation in range(max\_generations):  
 population = genetic\_algo\_cycle(  
 population,  
 func=func,  
 strategy=strategy,  
 crossover\_chance = crossover\_prob,  
 mutation\_chance=mutation\_prob  
 )  
  
 # Найти лучшего индивида  
 best\_individual = population[0]  
  
  
 if best\_individual.fitness <= target\_accuracy:  
 total\_generations += generation + 1  
 total\_best\_fitness += best\_individual.fitness  
 success = True  
 success\_count += 1  
 break  
  
 if not success:  
 total\_generations += max\_generations  
 total\_best\_fitness += best\_individual.fitness  
  
 # Усреднение результатов  
 avg\_generations = total\_generations / num\_repeats  
 avg\_best\_fitness = total\_best\_fitness / num\_repeats  
 success\_rate = success\_count / num\_repeats  
  
 results.append({  
 "mutation\_prob": mutation\_prob,  
 "crossover\_prob": crossover\_prob,  
 "avg\_generations": avg\_generations,  
 "avg\_best\_fitness": avg\_best\_fitness,  
 "success\_rate": success\_rate  
 })  
  
 # Группировка результатов по вероятности мутации и кроссинговера  
 mutation\_avg = {}  
 crossover\_avg = {}  
  
 for mutation\_prob in mutation\_probs:  
 filtered = [r for r in results if r["mutation\_prob"] == mutation\_prob]  
 mutation\_avg[mutation\_prob] = {  
 "avg\_generations": sum(r["avg\_generations"] for r in filtered) / len(filtered),  
 "avg\_best\_fitness": sum(r["avg\_best\_fitness"] for r in filtered) / len(filtered),  
 "success\_rate": sum(r["success\_rate"] for r in filtered) / len(filtered)  
 }  
  
 for crossover\_prob in crossover\_probs:  
 filtered = [r for r in results if r["crossover\_prob"] == crossover\_prob]  
 crossover\_avg[crossover\_prob] = {  
 "avg\_generations": sum(r["avg\_generations"] for r in filtered) / len(filtered),  
 "avg\_best\_fitness": sum(r["avg\_best\_fitness"] for r in filtered) / len(filtered),  
 "success\_rate": sum(r["success\_rate"] for r in filtered) / len(filtered)  
 }  
  
 # Вывод результатов  
 print(f"{'Mutation':<10} {'Crossover':<10} {'Avg Gen':<10} {'Avg Fitness':<12} {'Success Rate':<12}")  
 print("-" \* 54)  
 for result in results:  
 print(f"{result['mutation\_prob']:<10.4f} {result['crossover\_prob']:<10.4f} "  
 f"{result['avg\_generations']:<10.2f} {result['avg\_best\_fitness']:<12.4f} {result['success\_rate']:<12.2%}")  
  
 print("\nAveraged Results for Mutation Probabilities:")  
 print(f"{'Mutation':<10} {'Avg Gen':<10} {'Avg Fitness':<12} {'Success Rate':<12}")  
 print("-" \* 44)  
 for mutation\_prob, data in mutation\_avg.items():  
 print(f"{mutation\_prob:<10.4f} {data['avg\_generations']:<10.2f} "  
 f"{data['avg\_best\_fitness']:<12.4f} {data['success\_rate']:<12.2%}")  
  
 print("\nAveraged Results for Crossover Probabilities:")  
 print(f"{'Crossover':<10} {'Avg Gen':<10} {'Avg Fitness':<12} {'Success Rate':<12}")  
 print("-" \* 44)  
 for crossover\_prob, data in crossover\_avg.items():  
 print(f"{crossover\_prob:<10.4f} {data['avg\_generations']:<10.2f} "  
 f"{data['avg\_best\_fitness']:<12.4f} {data['success\_rate']:<12.2%}")  
 average\_generation = sum(r["avg\_generations"] for r in results) / len(results)  
 print(f"Average Generation: {average\_generation}")  
 return results, mutation\_avg, crossover\_avg, average\_generation  
  
  
def main():  
 strategy = "m+n"  
 func = "418.9829-xi\*math.sin(math.sqrt(abs(xi)))"  
 bounds = [-500, 500]  
 print(function([420.9687, 420.9687], "418.9829-xi\*math.sin(math.sqrt(abs(xi)))"))  
 population = genetic\_algorithm\_init(50,gene\_size=2,strategy=strategy)  
 for i in range(100):  
 population = genetic\_algo\_cycle(population, func="418.9829-xi\*math.sin(math.sqrt(abs(xi)))",strategy=strategy)  
 if i%1 == 0:  
 print("GEN: ", i, " - ", population[0].phenotype, " - ", population[0].fitness)  
 plot\_function\_and\_population(func, population, bounds)  
  
 return  
def test():  
 func = "418.9829-xi\*math.sin(math.sqrt(abs(xi)))"  
 bounds = [-500, 500]  
 population\_size = 50  
 #gene\_size = 2  
 max\_generations = 500  
 mutation\_probs = [0.1 , 0.3, 0.5, 0.7, 0.9]  
 crossover\_probs = [0.2, 0.4, 0.6, 0.8, 1.0]  
 strategy = "m+n"  
 for gene\_size in range(3,6):  
 test\_mutation\_crossover\_accuracy(  
 func=func,  
 bounds=bounds,  
 population\_size=population\_size,  
 gene\_size=gene\_size,  
 max\_generations=max\_generations,  
 mutation\_probs=mutation\_probs,  
 crossover\_probs=crossover\_probs,  
 num\_repeats=1,  
 strategy = strategy  
 )  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

**4. Исследование**

Я исследовал зависимость достижения в зависимости от типа кроссинговера и вероятности кроссинговера и мутации.

В качетсве параметра остановки я взял достижение точности , не смторя на то что результат может уточняться ещё больше.

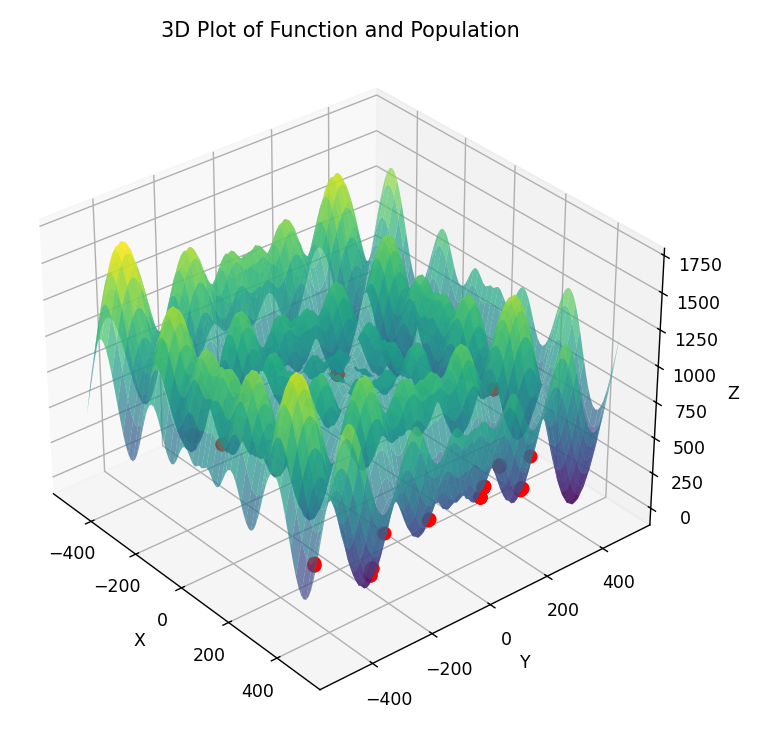
Заметно, что оптимум вероятности мутации находится около 0.5 – 0.6

Заметен оптимум в районе 0.8

Видно что в зависимости от числа измерений скорость нахождения растёт экспоненциально

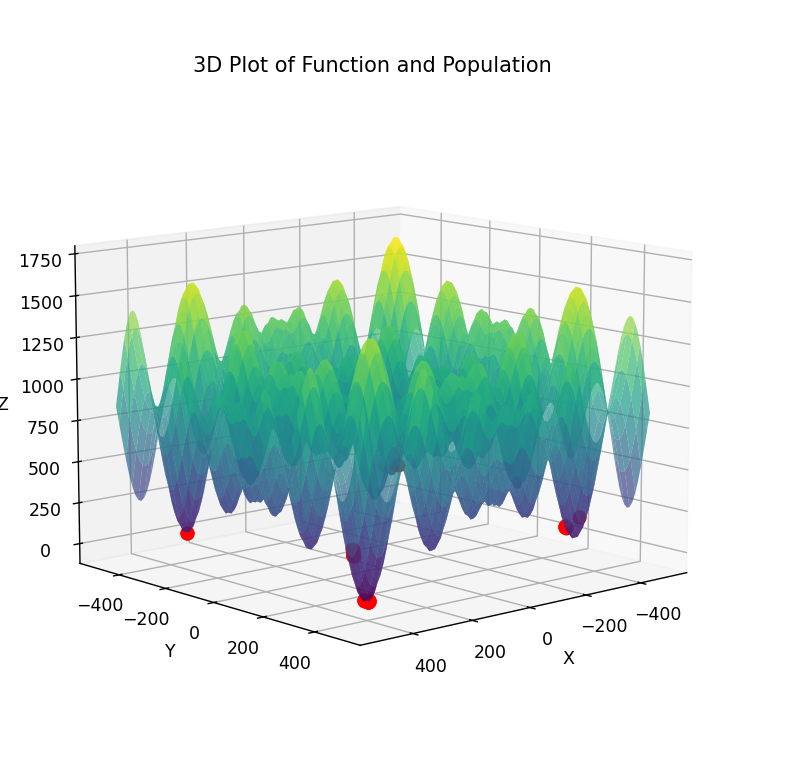
**5. Выполнение симуляции**

**Шаг 1**

****Начальная популяция

Лучшая точка - [454, -308] : 254.38355404021843

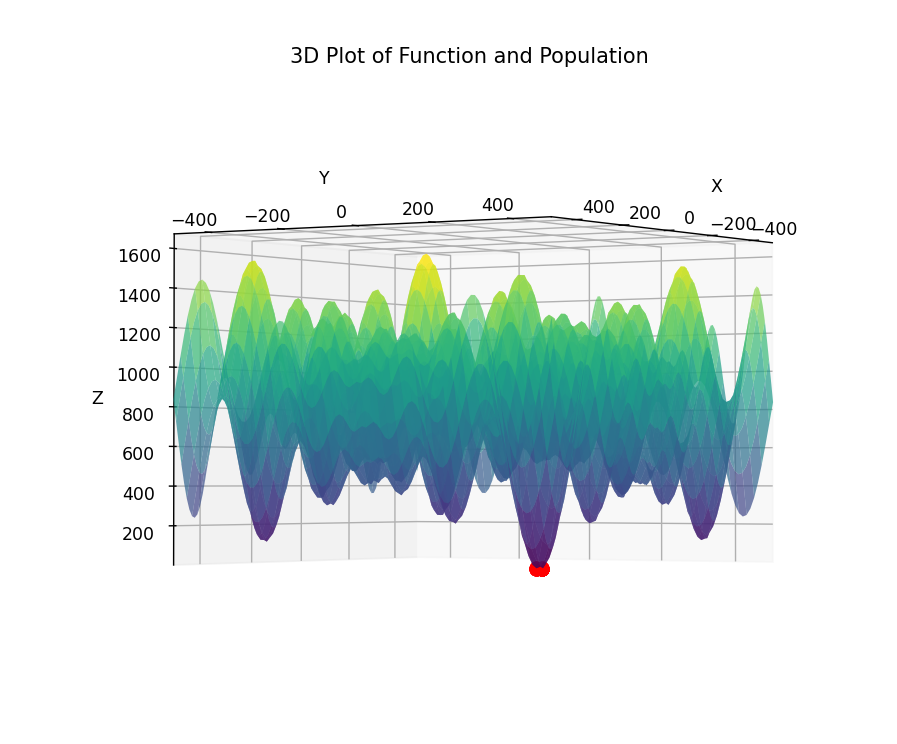
**Шаг 2**

****Популяция спустя 1шаг

Обозначился глобальный минимум, но также есть и другие

Лучшая точка - [420.54288503452403, 428.53240183064463] - 7.2420651509362415

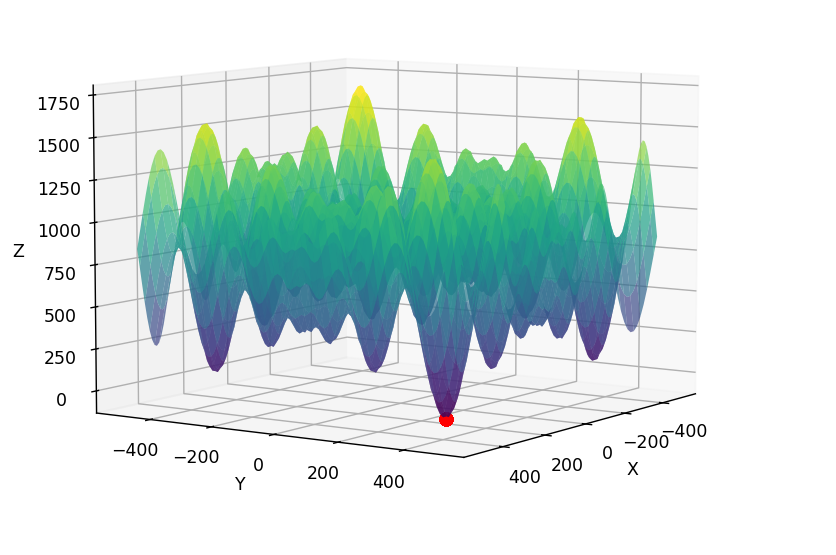
**Шаг 3**

****Популяция спустя 2 шага

Остался только глобальный минимум

Лучшая точка — [421.91183209193713, 414.8176078845198] - 4.866226808454826

**Поколение 15**

****Популяция спустя 15 шагов

Лучшая точка — [420.96415406620645, 420.9534323943223] - 5.77083098391995e-05

От реального минимума ([420.9687, 420.9687] - 2.545567497236334e-05) отличается незначительно

**6. Контрольный вопрос**

**Что такое многократная ЭС?**

Многократнся ЭС это такая ЭС, в которой размер популяции больше двух, существует операция кроссинговера, а также вероятность мутации для всех особей равна. Есть многократные стратегии с 1 потомком и с множеством потомков.