

Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования Московский
государственный технический университет имени Н.Э. Баумана

Лабораторная работа №7. Вариант 6.
«Методы решения задач многокритериальной и
многоэкстремальной оптимизации»
по курсу
«Методы оптимизации»

Студент группы ИУ9-82

Иванов Г.М

Преподаватель

Каганов Ю.Т.

Москва, 2019

Содержание

1	Цель работы	3
2	Постановка задачи	4
2.1	Задача 7.1	4
2.2	Задача 7.2	4
3	Исследование	6
3.1	Задача 7.1	6
3.1.1	Алгоритм метода ближайшего соседа	6
3.1.2	Алгоритм метода конкурирующих точек	7
3.1.3	Генетический алгоритм	7
3.2	Задача 7.2	9
3.2.1	Методы свертки критериев (метод «идеальной точки»)	9
4	Практическая реализация	10
5	Результаты.	18

1 Цель работы

1. Изучение методов решения задач многоэкстремальной и многокритериальной оптимизации.
2. Разработка программ реализации алгоритмов мультистарта (метода ближайшего соседа, метода конкурирующих точек), генетического алгоритма, алгоритмом многокритериальной оптимизации.
3. Решение задачи:
 - (а) многоэкстремальной оптимизации для функции Шекеля;
 - (б) многоэкстремальной оптимизации для заданных многоэкстремальных функций;
 - (с) многокритериальной оптимизации.

2 Постановка задачи

2.1 Задача 7.1

Дано: 1 Вариант. Функция Шекеля:

$$f(x) = - \sum_{i=1}^3 \frac{a}{f_i^0 + b \sum_{j=1}^3 (x_j - x_{ij}^0)^2},$$
$$\sum_{i=1}^3 \frac{1}{f_i^0} = f^0 = -c, x_{ij}^0 \in [\alpha, \beta]^3, \quad (1)$$

где $a = 2$, $b = 3$, $c = 1$, $\alpha = 0$, $\beta = 5$.

1. Реализовать алгоритмы многоэкстремальной оптимизации методом мултистарта на одном из языков высокого уровня.
2. Решить задачу многоэкстремальной оптимизации с помощью метода мултистарта.
1. Реализовать генетический алгоритм многоэкстремальной оптимизации на одном из языков высокого уровня.
2. Решить задачу многоэкстремальной оптимизации с помощью генетического алгоритма.

2.2 Задача 7.2

Дано: 1 Вариант.

$$\begin{cases} f_1(x) = x_1^2 + x_2^2 \rightarrow \min, \\ f_2(x) = 4(x_1 - 5)^2 + 2(x_2 - 6)^4 \rightarrow \min, \\ g_1(x) = x_2 - x_1 + 1 \leq 0, \\ g_2(x) = x_1 + x_2 - 2 \leq 0. \end{cases} \quad (2)$$

1. Реализовать метод свертки критериев (метод «идеальной точки») многокритериальной оптимизации на одном из языков высокого уровня.
2. Требуется найти множества сильно эффективных решений (оптимальных по Парето) и слабо эффективных решений (оптимальных по Слейтеру) для (2).
3. Поиск четырех эффективных точек определяется путем задания бинарных отношений между критериями и получением вектора весовых коэффициентов методом Т. Саати для (2).
4. Исследовать полученное представление решений в пространстве критериев с учетом заданных ограничений для (2).
5. Реализовать алгоритмы программированием на C++ и Python

3 Исследование

Найдем экстремум для функции Шекеля:

$$f(x) = - \sum_{i=1}^3 \frac{a}{f_i^0 + b \sum_{j=1}^3 (x_j - x_{ij}^0)^2},$$
$$\sum_{i=1}^3 \frac{1}{f_i^0} = f^0 = -c, x_{ij}^0 \in [\alpha, \beta]^3, \quad (3)$$

где $a = 2, b = 3, c = 1, \alpha = 0, \beta = 5$, с помощью сервиса WolframAlpha.com:

$$\min(f(x)) = 18, \quad (x_1, x_2, x_3) = (4, 4, 4) \quad (4)$$

3.1 Задача 7.1

3.1.1 Алгоритм метода ближайшего соседа

1. Генерируются K точек (z_1, z_2, \dots, z_k) случайным образом с помощью, например, алгоритма Монте-Карло. Считается, что все они принадлежат различным кластерам (кластер ассоциируется с окрестностью локального минимума).
2. Находится ближайшая пара точек (z_i, z_j) , т.е. такая пара, для которой $\rho_{ij} = \operatorname{argmin}_{k,l=1,\dots,K} \rho_{kl}$.
3. Если расстояние $\rho_{ij} = \rho(z_i, z_j)$ между ближайшими соседями не превосходит некоторое достаточно малое число $\sigma > 0$, то точки z_i и z_j , а также соответствующие им кластеры объединяются и число кластеров K уменьшается на единицу. После этого происходит переход на Ш.2.
4. Остановка алгоритма происходит, либо если расстояние между ближайшими соседями превосходит σ , либо если остается лишь один кластер.

В качестве метрики $\rho(z_i, z_j)$ обычно выбирается евклидова метрика. Качество алгоритма как процедуры кластеризации существенно зависит от того, насколько удачно выбрано число σ . Это число должно быть достаточно малым, во всяком случае меньше расстояний между соседними точками локальных минимумов.

3.1.2 Алгоритм метода конкурирующих точек

1. Моделируется равномерное на X распределение. В результате получается N точек x_1, x_2, \dots, x_N .
2. Используя точки x_1, x_2, \dots, x_N в качестве начальных, проводится одна или несколько итераций какого либо алгоритма локальной оптимизации. Получаем точки (z_1, z_2, \dots, z_N) .
3. Применяется какой либо метод кластеризации к точкам (z_1, z_2, \dots, z_N) . Пусть m - число получившихся кластеров. Если $m = 1$, то переход на Ш.5. Иначе переход на Ш.4.
4. Выбираются представители x_1, x_2, \dots, x_m от всех кластеров (естественно выбирать в кластерах с наименьшим значением целевой функции). Положим $N = m$ и переход на Ш.2.
5. Считаем, что находимся в окрестности точки глобального минимума. Выбрать представителя от единственного кластера. Используя его в качестве начальной точки для алгоритма локальной оптимизации, обладающего высокой скоростью сходимости в окрестности точки экстремума.

3.1.3 Генетический алгоритм

Генетический алгоритм - это эвристический алгоритм поиска, используемый для решения задач оптимизации и моделирования путём

случайного подбора, комбинирования и вариации искоемых параметров с использованием механизмов, аналогичных естественному отбору в природе.

В нем используется основных 5 принципов:

1. **Формирование исходной популяции** - в зависимости от поставленной задачи генерируются N особей $x^k = (x_1^k, \dots, x_n^k)^T$, для которой вычисляется функция фитнеса;
2. **Отбор (селекция)** - операция, которая осуществляет отбор особей (хромосом) x^k в соответствии со значениями функции фитнеса $\mu(x^k)$ для последующего их скрещивания.
3. **Кроссинговер (скрещивание)** – это операция, при которой из нескольких, обычно двух хромосом (особей) (x_i, x_j) , $i \neq j$, называемых родителями, порождается одна или несколько новых, называемых потомками.
4. **Мутация** - это преобразование хромосомы, случайно изменяющее один или несколько из её генов. Оператор мутации предназначен для того, чтобы поддерживать разнообразие особей в популяции. Необходимо определить параметр $Pm \in (0, 1]$ - вероятность мутации.
5. **Формирование новой популяции**
 - (a) С равной вероятностью из потомков мутантов предыдущего шага выбирается один $x^m = (x_1, x_2, \dots, x_p^M, \dots, x_n)$.
 - (b) Выбранный потомок добавляется в популяцию вместо хромосомы, которой соответствует наименьшее значение функции фитнеса (наихудшее из допустимых значений).
 - (c) Вычисляется значение функции фитнеса для мутантного потомка $\mu_M = \mu(x^M)$.

6. Проверка условия останова генетического алгоритма - условием окончания работы генетического алгоритма является формирование заданного количества популяций $t = Nr$.

3.2 Задача 7.2

3.2.1 Методы свертки критериев (метод «идеальной точки»)

1. Решение оптимизационных задач для каждого из критериев отдельно и получение «идеальной точки» с учетом ограничений.
2. Выбор весовых коэффициентов. Величины весовых коэффициентов - подбираются исходя из предварительно обусловленных соображений, которые могут быть выражены в виде дополнительных неформализуемых критериев. Чаще всего весовые коэффициенты выбираются путём бинарных сравнений и построения обратно симметричной матрицы. Далее вычисляются собственные значения и, для максимального собственного значения, выбирается соответствующий собственный вектор, элементы которого используются как весовые коэффициенты (метод Т. Саати). В данном случае весовые коэффициенты могут варьироваться различным образом для исследования пространства эффективных точек.
3. Получение оптимальных по Парето и Слейтеру решений. Для решения задачи можно использовать методы штрафных функций или метод модифицированных функций Лагранжа.

4 Практическая реализация

Все методы были реализованы на языке программирования **Python**. Для алгоритмов с кластеризацией использовались следующие классы:

Листинг 1. Вспомогательные классы для кластеризации.

```
1 class FunValue:
2     def __init__(self, x_values, fun_value):
3         self.x_values = x_values
4         self.fun_value = fun_value
5
6     def __str__(self):
7         return "{} , {}".format(self.x_values, self.fun_value)
8
9
10 class Cluster:
11     def __init__(self, fun_values=None):
12         if fun_values is None:
13             fun_values = []
14         self.funValues = [fun_values]
15
16     def euclidean_distance(self, another_cluster):
17         min_distance = float("INF")
18         for i in self.funValues:
19             distance = 0
20             for j in another_cluster.funValues:
21                 distance += pow(i.fun_value - j.fun_value, 2)
22
23             if distance < min_distance:
24                 min_distance = distance
25
26         return math.sqrt(min_distance)
27
28     def merge(self, another_cluster):
29         self.funValues += another_cluster.funValues
30
31     def min_by_value(self):
32         min_fun_value = None
33
34         for item in self.funValues:
35             if min_fun_value is None:
36                 min_fun_value = item
37             elif item.fun_value < min_fun_value.fun_value:
38                 min_fun_value = item
```

```

39         assert min_fun_value is not None
40         return min_fun_value
41
42     def __str__(self):
43         return "{}".format(self.funValues)
44 
```

Листинг 2. Метод ближайшего соседа.

```

1  def nearest_neighbor(clusters, epsilon):
2      while True:
3          min_distance = float("INF")
4          min_i = -1
5          min_j = -1
6
7          for i in range(len(clusters)):
8              for j in range(len(clusters)):
9                  if i != j and
10                     ↪ clusters[i].euclidean_distance(clusters[j])
11                     ↪ < min_distance:
12                         min_distance =
13                             ↪ clusters[i].euclidean_distance(clusters[j])
14                         min_i = i
15                         min_j = j
16
17             assert min_i > -1
18             assert min_j > -1
19
20             if min_distance < epsilon:
21                 clusters[min_i].merge(clusters[min_j])
22                 clusters.remove(clusters[min_j])
23             else:
24                 break
25
26             if len(clusters) <= 1:
27                 break
28
29             # for cluster in clusters:
30             #     print()
31             #     for funValue in cluster.funValues:
32             #         print(funValue)
33
34         return clusters

```

Листинг 3. Метод конкурирующих точек

```

1  def competing_points(cluster_count, epsilon):
2      not_changed_in_inter = False
3      step = (beta - alpha) / (cluster_count - 1)
4      clusters = [generate_cluster([alpha + i * step for _ in range(3)])
5                  ↪ for i in range(cluster_count)]
6
7      while True:
8          for cluster in clusters:
9              for item in cluster.funValues:
10                 nelder_mead = nelderMead(item.x_values)
11
12                 item.fun_value = nelder_mead.fun
13                 item.x_values = nelder_mead.x
14
15             clusters = nearest_neighbor(clusters, epsilon)
16
17             print(len(clusters))
18             if len(clusters) != 1:
19                 if not_changed_in_inter:
20                     not_changed_in_inter = False
21                     epsilon *= 10
22                 else:
23                     not_changed_in_inter = True
24
25                 min_from_clusters = []
26                 for cluster in clusters:
27                     min_from_clusters.append(Cluster(cluster.min_by_value()))
28
29                 clusters = min_from_clusters
30             else:
31                 min_fun_value = clusters[0].min_by_value()
32                 nelder_mead = nelderMead(min_fun_value.x_values)
33                 print("f({}) = {}".format(nelder_mead.x,
34                 ↪ nelder_mead.fun))
35                 break

```

Листинг 4. Генетический алгоритм

```

1  class GeneticAlgorithm():
2      def __init__(self):
3          self.alpha = 0
4          self.beta = 5
5          self.population_size = 60
6          self.number_of_genes = 20

```

```

7         self.crossover_probability = 0.5
8         self.mutation_probability = 1/self.number_of_genes
9         self.tournament_selection_parameter = 0.75
10        self.tournament_size = 3
11        self.number_of_variables = 3
12        self.variable_range = 5
13        self.number_of_generations = 150
14        self.number_of_best_individual_copies = 1
15        self.fitness = [0 for x in range(self.population_size)]
16
17        self.population =
18            ↪ self.initialize_population(self.population_size,
19            ↪ self.number_of_genes)
20
21    def run_ga(self):
22        k = 0
23        for iGenerations in range(self.number_of_generations):
24            print(k)
25            k += 1
26
27            maximum_fitness = 0.0
28            x_best = [0 for _ in range(self.number_of_variables)]
29            best_individual = None
30
31            # Decode chromosome and evaluate individual
32            for i in range(self.population_size):
33                chromosome = self.population[i]
34                x = self.decode_chromosome(chromosome,
35                ↪ self.number_of_variables,
36                ↪ self.variable_range)
37                self.fitness[i] = self.evaluate_individual(x)
38                if self.fitness[i] > maximum_fitness:
39                    maximum_fitness = self.fitness[i]
40                    x_best = x
41                    best_individual = chromosome
42            temp_population = self.population
43
44            # Tournament selection
45            for i in range(round(self.population_size/2)):
46                j = i*2
47                i1 = self.tournament_select(self.fitness,
48                ↪ self.tournament_selection_parameter,
49                ↪ self.tournament_size)
50                i2 = self.tournament_select(self.fitness,
51                ↪ self.tournament_selection_parameter,
52                ↪ self.tournament_size)
53                chromosome_1 = self.population[i1]
54                chromosome_2 = self.population[i2]

```

```

48         # Crossover
49         r = random.random()
50         if r < self.crossover_probability:
51             new_chromosome_pair =
52                 ↪ self.cross(chromosome_1,
53                 ↪ chromosome_2)
54             temp_population[j] =
55                 ↪ new_chromosome_pair[0]
56             temp_population[j+1] =
57                 ↪ new_chromosome_pair[1]
58         else:
59             temp_population[j] = chromosome_1
60             temp_population[j+1] = chromosome_2
61
62         # Mutate
63         for i in range(self.population_size):
64             original_chromosome = temp_population[i]
65             mutated_chromosome =
66                 ↪ self.mutate(original_chromosome,
67                 ↪ self.mutation_probability)
68             temp_population[i] = mutated_chromosome
69
70         # Insert best individual
71         if best_individual is not None:
72             temp_population =
73                 ↪ self.insert_best_individual(temp_population,
74                 ↪ best_individual,
75                 ↪ self.number_of_best_individual_copies)
76             population = temp_population
77
78         print("f({}) = {}".format(x_best, 1/maximum_fitness))
79
80     def initialize_population(self, population_size, number_of_genes):
81         population = [[0 for y in range(number_of_genes)] for x in
82             ↪ range(population_size)]
83         step = (self.beta - self.alpha) / self.number_of_genes
84         for i in range(population_size):
85             for j in range(number_of_genes):
86                 while True:
87                     population[i][j] = step *
88                         ↪ random.random() + step
89
90                     if 0 <= population[i][j] <= 1:
91                         break
92
93         return population
94
95     def decode_chromosome(self, chromosome, number_of_variables,
96         ↪ variable_range):
97         n_genes = len(chromosome)

```

```

85         n_split = round(n_genes/self.number_of_variables)
86         x = [0.0 for x in range(self.number_of_variables)]
87
88         for i in range(self.number_of_variables):
89             for j in range(n_split):
90                 x[i] = x[i] +
91                     ↪ chromosome[n_split*(i-1)+j]*2**(-j)
92                 x[i] = -self.variable_range +
93                     ↪ 2*self.variable_range*x[i]/(1-2**(-n_split))
94         return x
95
96     def evaluate_individual(self, x):
97         # The fitness function
98         x_values = x
99         if len(x_values) == 3:
100             a = 2
101             b = 3
102             c = 1
103
104             g = - 3 * a / (
105                 ↪ -c / 3.0 + b * (pow(x_values[0] -
106                     ↪ 4.0, 2) + pow(x_values[1] - 4.0,
107                     ↪ 2) + pow(x_values[2] - 4.0, 2)))
108
109             fitness_value = 1/g
110         else:
111             print("ERROR: Wrong size individual")
112             sys.exit()
113         return fitness_value
114
115     def tournament_select(self, fitness, tournament_selection_parameter,
116         ↪ tournament_size):
117         i_tmp_vector = [0 for x in range(tournament_size)]
118         fitness_vector = [0 for x in range(tournament_size)]
119         i_selected = None
120         for i in range(tournament_size):
121             i_tmp_vector[i] =
122                 ↪ int(random.random()*self.population_size)
123             fitness_vector[i] = fitness[i_tmp_vector[i]]
124
125         no_chosen_index = True
126         while no_chosen_index:
127             idx_maximum =
128                 ↪ fitness_vector.index(max(fitness_vector))
129             if len(fitness_vector) > 1:
130                 if random.random() <
131                     ↪ tournament_selection_parameter:
132                     i_selected =
133                         ↪ i_tmp_vector[idx_maximum]

```

```

125         no_chosen_index = False
126     else:
127         fitness_vector.pop(idx_maximum)
128         i_tmp_vector.pop(idx_maximum)
129     else:
130         i_selected = i_tmp_vector[0]
131         no_chosen_index = False
132
133     return i_selected
134
135 def cross(self, chromosome_1, chromosome_2):
136     n_genes = len(chromosome_1)
137     crossover_point = round(random.random() * n_genes)
138
139     new_chromosome_pair = [[0 for y in range(n_genes)] for x in
140     ↪ range(2)]
141     for j in range(n_genes):
142         if j < crossover_point:
143             new_chromosome_pair[0][j] = chromosome_1[j]
144             new_chromosome_pair[1][j] = chromosome_2[j]
145         else:
146             new_chromosome_pair[0][j] = chromosome_2[j]
147             new_chromosome_pair[1][j] = chromosome_1[j]
148
149     return new_chromosome_pair
150
151 def mutate(self, chromosome, mutation_probability):
152     mutated_chromosome = chromosome
153     for j in range(self.number_of_genes):
154         if random.random() < mutation_probability:
155             mutated_chromosome[j] = 1-chromosome[j]
156
157     return mutated_chromosome
158
159 def insert_best_individual(self, population, best_individual,
160 ↪ number_of_best_individual_copies):
161     for i in range(number_of_best_individual_copies):
162         population[i] = best_individual
163
164     return population

```

Листинг 5. Метод "идеальной точки".

```

1  r0 = 1.0
2  accelerator = 2.0
3  epsilon = 10 ** -3

```



```

4   k = 0
5
6   def function_cut(a):
7       if a > 0:
8           return a
9       return 0
10
11  def function_1(x):
12      return x[0] ** 2 + x[1] ** 2
13
14  def function_2(x):
15      return 4 * (x[0] - 5) ** 2 + 2 * (x[1] - 6) ** 4
16
17  def cond_1(x):
18      return x[1] - x[0] + 1
19
20  def cond_2(x):
21      return x[0] + x[1] - 2
22
23
24  def effective_points(w1, w2, x1, x2):
25      def F(x, r):
26          return (w1 * (function_1(x) - function_1(x1)) + w2 *
27                  ↪ (function_2(x) - function_2(x2)) + (r / 2.) * (
28                      (function_cut(cond_1(x)) ** 2) +
29                      ↪ (function_cut(cond_2(x)) ** 2)))
30
31      def P(x, r):
32          return (r / 2.) * ((function_cut(cond_1(x)) ** 2) +
33                              ↪ (function_cut(cond_2(x)) ** 2))
34
35      def barrier_method(r, k, x0):
36          res = (minimize(lambda x: F(x, r), x0, method='CG'))
37          newx = res.x
38
39          if np.fabs(P(newx, r)) <= epsilon:
40              return newx
41          else:
42              return barrier_method(accelerator * r, k + 1, newx)
43
44  result = barrier_method(r0, k, [0.0, 0.0])
45  print('w1 = {}, w2 = {}, \n x = {}, f_1(x) = {:.2f}, f_2(x) =
46  ↪ {:.2f}\n'.format(w1, w2, result, function_1(result),
47  ↪ function_2(result)) )

```

5 Результаты.

Были получены следующие результаты:

Листинг 5. Результаты выполнения программ.

```
1 Start Competitng points method method:
2   f([4.0605263157894731, 4.0605263157894731, 4.0605263157894731]) =
   ↪ 19.975867872760627
3
4 Start Genetic Algorithm:
5   f([4.038834661591123, 4.030962344985451, 3.9695648282003884]) =
   ↪ 18.56699665784427
6
7 Start Effective points method:
8   w1 = 1, w2 = 1.0,
9   x = [ 1.500000002  0.50000124], f_1(x) = 2.50, f_2(x) = 1879.12
10
11   w1 = 2, w2 = 0.5,
12   x = [ 1.500000002  0.50000247], f_1(x) = 2.50, f_2(x) = 1879.12
13
14   w1 = 3, w2 = 0.3333333333333333,
15   x = [ 1.5          0.50000329], f_1(x) = 2.50, f_2(x) = 1879.12
16
17   w1 = 4, w2 = 0.25,
18   x = [ 1.499999992  0.5000049 ], f_1(x) = 2.50, f_2(x) = 1879.12
```