



Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ _____ «Информатика и системы управления»

КАФЕДРА _____ «Теоретическая информатика и компьютерные технологии»

Лабораторная работа № 3
по курсу «Теория искусственных нейронных сетей»
«Методы многомерного поиска. Генетический алгоритм»

Студентка группы ИУ9-72Б Самохвалова П. С.

Преподаватель Каганов Ю. Т.

Москва 2023

1 Цель работы

1. Изучение алгоритмов многомерного поиска 1-го и 2-го порядка.
2. Разработка программ реализации алгоритмов многомерного поиска 1-го и 2-го порядка.
3. Вычисление экстремумов функции.
4. Изучение методов решения задач многоэкстремальной оптимизации на основе генетического алгоритма.
5. Разработка программы реализации генетического алгоритма.
6. Решение задачи многоэкстремальной оптимизации для заданных многоэкстремальных функций.

2 Задание

Требуется найти минимум тестовой функции Розенброка:

$$f(x) = \sum_{i=1}^{n-1} [a(x_i^2 - x_{i+1})^2 + b(x_i - 1)^2] + f_0$$

1. Методами сопряженных градиентов (методом Флетчера-Ривза и методом Полака-Рибьера).
2. Квазиньютоновским методом (Девидона-Флетчера-Пауэлла).
3. Методом Левенберга-Марквардта.
4. При помощи генетического алгоритма.

Вариант 19

$$a = 220, b = 3, f_0 = 12, n = 2$$

3 Практическая реализация

Исходный код программы представлен в листинге 1.

Листинг 1: Методы многомерного поиска. Генетический алгоритм

```
1 import numpy as np
2 import copy
3 import math
4 import random
5
6
7 def method_svann(x_start, step_size, function):
8     k = 0
9     x_values = [x_start]
10
11     fun_result_without_step_size = function(x_start - step_size)
12     fun_result_on_start = function(x_start)
13     fun_result_with_step_size = function(x_start + step_size)
14
15     start = x_start - step_size
16     end = x_start
17
18     if fun_result_without_step_size >= fun_result_on_start and
19     fun_result_on_start <= fun_result_with_step_size:
20         return start, end
21     else:
22         delta = 0.0
23         k += 1
24         if fun_result_without_step_size >= fun_result_on_start >=
25         fun_result_with_step_size:
26             delta = step_size
27             start = x_values[0]
28             x_values.insert(k, x_start + step_size)
29             elif fun_result_without_step_size <= fun_result_on_start <=
30             fun_result_with_step_size:
31                 delta = -step_size
32                 end = x_values[0]
33                 x_values.insert(k, x_start - step_size)
34             while True:
35                 x_values.insert(k + 1, (x_values[k] + (2 ** k) * delta))
36                 if function(x_values[k + 1]) >= function(x_values[k]):
37                     if delta > 0:
38                         end = x_values[k + 1]
39                     elif delta < 0:
40                         start = x_values[k + 1]
41                 else:
42                     if delta > 0:
43                         start = x_values[k]
44                     elif delta < 0:
45                         end = x_values[k]
```

```

43         if function(x_values[k + 1]) >= function(x_values[k]):
44             break
45         k += 1
46     return start, end
47
48
49 def golden_section_method(eps, start, end, function):
50     k = 0
51     phi = (1 + math.sqrt(5.0)) / 2
52
53     while abs(end - start) > eps:
54         z = (end - (end - start) / phi)
55         y = (start + (end - start) / phi)
56         if function(y) <= function(z):
57             start = z
58         else:
59             end = y
60         k += 1
61
62     return (start + end) / 2
63
64
65 def find_min(start, function):
66     step = 0.01
67     x_start, x_end = method_svann(start, step, function)
68     return golden_section_method(0.001, x_start, x_end, function)
69
70
71 def minimizing(point, grad_value, function):
72     def func(gamma):
73         return function(point - grad_value.T * gamma)
74     return func
75
76
77 def gradient_descend_method(x0, eps1, eps2, f, gradient):
78
79     print("Gradient Descend method")
80
81     xk = x0[:]
82     k = 0
83     while True:
84         gradient_value = gradient(xk)
85
86         if np.linalg.norm(gradient_value) < eps1:
87             print("Number of iterations: %d" % k)
88             print(xk)

```

```

89         print(f(xk))
90         print()
91         return xk
92
93     if k >= max_iter:
94         print("Number of iterations: %d" % k)
95         print(xk)
96         print(f(xk))
97         print()
98         return xk
99
100     a = 0.0
101     min_value_func = f(xk - a * gradient_value)
102     for i in np.arange(0.0, 2.0, 0.001):
103         if i == 0.0:
104             continue
105         func_value = f(xk - i * gradient_value)
106         if func_value < min_value_func:
107             min_value_func = func_value
108             a = i
109
110     xk_new = xk - a * gradient_value
111
112     if np.linalg.norm(xk_new - xk) < eps2 and np.linalg.norm(f(
xk_new) - f(xk)):
113         print("Number of iterations: %d" % (k - 1))
114         print(xk_new)
115         print(f(xk_new))
116         print()
117         return
118     else:
119         k += 1
120         xk = xk_new
121
122
123 def flatcher_rivz_method(x0, eps1, eps2, f, gradient):
124     print("Flatcher - Rivz method")
125
126     xk = x0[:]
127     xk_new = x0[:]
128     xk_old = x0[:]
129
130     k = 0
131     d = []
132
133     while True:

```

```

134     gradient_value = gradient(xk)
135
136     if np.linalg.norm(gradient_value) < eps1:
137         print("Number of iterations: %d" % k)
138         print(xk)
139         print(f(xk))
140         print()
141         return
142
143     if k >= max_iter:
144         print("Number of iterations: %d" % k)
145         print(xk)
146         print(f(xk))
147         print()
148         return
149     if k == 0:
150         d = -gradient_value
151         beta = np.linalg.norm(gradient(xk_new)) / np.linalg.norm(
152             gradient(xk_old))
153         d_new = np.add(-gradient(xk_new), np.multiply(beta, d))
154         t = 0.1
155         min_value_func = f(xk + t * d_new)
156         for i in np.arange(0.0, 1.0, 0.001):
157             if i == 0.0:
158                 continue
159             func_value = f(xk + i * d_new)
160             if func_value < min_value_func:
161                 min_value_func = func_value
162                 t = i
163         xk_new = xk + t * d_new
164         if np.linalg.norm(xk_new - xk) < eps2 and np.linalg.norm(f(
165             xk_new) - f(xk)):
166             print("Number of iterations:", k - 1)
167             print(xk_new)
168             print(f(xk_new))
169             print()
170             return
171         else:
172             k += 1
173
174             xk_old = xk
175             xk = xk_new
176             d = d_new
177
178 def davidon_fletcher_powell_method(x0, eps1, eps2, f, gradient):

```

```

178
179 print ("Davidon - Fletcher - Powell method")
180
181 eps1 /= 100
182 eps2 /= 100
183 k = 0
184 xk_new = copy.deepcopy(x0[:])
185 xk_old = copy.deepcopy(x0[:])
186
187 a_new = np.eye(2, 2)
188 a_old = np.eye(2, 2)
189
190 while True:
191     gradient_value = gradient(xk_old)
192
193     if np.linalg.norm(gradient_value) < eps1:
194         print ("Number of iterations: %d" % k)
195         print (xk_old)
196         print (f(xk_old))
197         print ()
198         return xk_old
199
200     if k >= max_iter:
201         print ("Number of iterations: %d" % k)
202         print (xk_old)
203         print (f(xk_old))
204         print ()
205         return xk_old
206
207     if k != 0:
208         delta_g = gradient(xk_new) - gradient_value
209         delta_x = xk_new - xk_old
210
211         num_1 = delta_x @ delta_x.T
212         den_1 = delta_x.T @ delta_g
213
214         num_2 = a_old @ delta_g @ delta_g.T * a_old
215         den_2 = delta_g.T @ a_old @ delta_g
216         a_c = num_1 / den_1 - num_2 / den_2
217         a_old = a_new
218         a_new = a_old + a_c
219
220     minimizing_function = minimizing(xk_new, a_new @ gradient_value.
221 T, f)
222
223     alpha = find_min(0.0, minimizing_function)

```

```

223
224     xk_old = xk_new
225     xk_new = xk_old - alpha * a_new @ gradient_value
226
227     if np.linalg.norm(xk_new - xk_old) < eps2 and np.linalg.norm(f(
xk_new) - f(xk_old)) < eps2:
228         print("Number of iterations: %d" % (k - 1))
229         print(xk_new)
230         print(f(xk_new))
231         print()
232         return xk_new
233     else:
234         k += 1
235
236
237 def levenberg_markvardt_method(x0, eps1, f, gradient, hessian):
238
239     print("Levenberg-Markvardt method")
240
241     k = 0
242     xk = x0[:]
243     nu_k = 10 ** 4
244     while True:
245         gradient_value = gradient(xk)
246
247         if np.linalg.norm(gradient_value) < eps1:
248             print("Number of iterations: %d" % k)
249             print(xk)
250             print(f(xk))
251             print()
252             return xk
253
254         if k >= max_iter:
255             print("Number of iterations:", k)
256             print(xk)
257             print(f(xk))
258             print()
259             return xk
260
261         while True:
262             hess_matrix = hessian(xk)
263             temp = np.add(hess_matrix, nu_k * np.eye(2))
264             temp_inv = np.linalg.inv(temp)
265             d_k = - np.matmul(temp_inv, gradient_value)
266             xk_new = xk + d_k
267             if f(xk_new) < f(xk):

```



```

268         k += 1
269         nu_k = nu_k / 2
270         xk = xk_new
271         break
272     else:
273         nu_k = 2 * nu_k
274     continue
275
276
277 def function(x):
278     return 220 * (x[0] ** 2 - x[1]) ** 2 + 3 * (x[0] - 1) ** 2 + 12
279
280
281 def gradient(x):
282     return np.array([880 * (x[0] ** 2 - x[1]) * x[0] + 6 * (x[0] - 1),
283                     -440 * (x[0] ** 2 - x[1])])
284
285
286 def gessian(x):
287     return np.array([[880 * (3 * x[0] ** 2 - x[1]) + 6, -880 * x[0]],
288                     [-880 * x[0], 440]])
289
290
291 max_iter = 10000
292
293 x_start = [0, 0]
294 eps = 0.0001
295
296 gradient_descend_method(x_start, eps, eps, function, gradient)
297 flatcher_rivz_method(x_start, eps, eps, function, gradient)
298 davidon_flatcher_powell_method(x_start, eps, eps, function, gradient)
299 levenberg_markvardt_method(x_start, eps, function, gradient, gessian)
300
301 def genetic_algorithm(Mp, Np, f):
302     print("Genetic algorithm")
303
304     population = [[random.uniform(-10, 10) for _ in range(2)] for _ in
305                   range(Mp)]
306     for k in range(Np):
307         fitness = [1 / f(population[i]) for i in range(Mp)]
308         fit = sum(fitness)
309         p = [0] * Mp
310         for i in range(Mp):
311             for j in range(i + 1):
312                 p[i] += fitness[j] / fit

```

```

311     p = [0] + p
312     cross = []
313     for i in range(Mp):
314         r = random.uniform(1e-7, 1.0)
315         for j in range(1, Mp + 1):
316             if p[j - 1] < r <= p[j]:
317                 cross.append(population[j - 1])
318     population_n = []
319     for i in range(Mp):
320         r = random.uniform(1e-7, 1 - 1e-7)
321         population_n.append([r * cross[i][0] + (1 - r) * cross[i][1]
for _ in range(2)])
322     pm = random.uniform(0.05, 0.2)
323     mutations = []
324     for i in range(Mp):
325         r = random.uniform(0, 1)
326         if r < pm:
327             mutations.append(population_n[i])
328     for i in range(len(mutations)):
329         mutations[i][random.randint(0, 1)] = random.uniform(-10, 10)
330     if len(mutations) > 0:
331         fitness = [1 / f(population_n[i]) for i in range(Mp)]
332         population_n[np.argmin(fitness)] = mutations[random.randint
(0, len(mutations) - 1)]
333     for i in range(Mp):
334         population[i] = population_n[i]
335     fitness = [1 / f(population[i]) for i in range(Mp)]
336     ind = np.argmax(fitness)
337     print(population[ind])
338     print(f(population[ind]))
339
340
341 Mp = 50
342 Np = 1000
343
344 genetic_algorithm(Mp, Np, function)

```

4 Результаты

Результаты работы программы представлены на рисунках 1- 2.

```
Gradient Descend method
Number of iterations: 10000
[1.09732064 1.20520757]
12.028677698158202

Flatcher-Rivz method
Number of iterations: 310
[0.99966582 0.99934829]
12.000000395263656

Davidon-Flatcher-Powell method
Number of iterations: 2790
[0.99910296 0.99819568]
12.000002440885549

Levenberg-Markvardt method
Number of iterations: 21
[0.99999978 0.99999957]
12.000000000000014
```

Рис. 1 — Результаты работы методов многомерного поиска

```
Genetic algorithm
[0.9999928027097036, 0.9999928027097036]
12.000000011551457
```

Рис. 2 — Результаты работы генетического алгоритма

5 Выводы

В результате выполнения лабораторной работы были изучены алгоритмы многомерного поиска, генетический алгоритм, разработаны программы реализации алгоритмов многомерного поиска и генетического алгоритма.