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

Лабораторная работа №4. Вариант 6. «Методы многомерного поиска. Методы 0-го, 1-го и 2-го порядка» по курсу «Методы оптимизации»

Студент группы ИУ9-82 Иванов Г.М

Преподаватель Каганов Ю.Т.

Москва, 2019

Contents

1 Цель работы

1. Изучение алгоритмов многомерного поиска.
2. Разработка программ реализации алгоритмов многомерного поиска 0-го, 1-го и 2-го порядка.
3. Вычисление экстремумов функции.

2 Постановка задачи

Дано: 1 Вариант. Функция Розенброка на множестве R^2:

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

где

$$a = 30, \quad b = 2, \quad f_0 = 80, \quad n = 2, \tag{2}$$

тогда функция $f(x)$ будет выглядеть следующим образом:

$$f(x) = 30 * (x_0^2 - x_1)^2 + 2 * (x_0 - 1)^2 + 80 \tag{3}$$

2.1 Задача 4.1

1. Найти экстремум методами:
   (a) Конфигураций (метод Хука-Дживса).
   (b) Нелдера-Мида.
2. Найти все стационарные точни и значения функций, соотвествующие этим точкам.
3. Оценить скорость сходимости указанных алгоритмов.
4. Реализовать алгоритмы с помощью языка программирования высокого уровня.

2.2 Задача 4.2

1. Найти экстремум методами:
   (a) Наискорейшего градиентного спуска.

(b) Флетчера-Ривза.

(c) Девидона-Флетчера-Пауэлла.

(d) Левенберга-Марквардта

2. Найти все стационарные точни и значения функций, соотвествующие этим точкам.

3. Оценить скорость сходимости указанных алгоритмов.

4. Реализовать алгоритмы с помощью языка программирования высокого уровня.

3 Исследование

Найдем глобальные экстремумы функции

$$f(x) = 30 * (x_0^2 - x_1)^2 + 2 * (x_0 - 1)^2 + 80 \tag{4}$$

с помощью сервиса WolframAlpha.com:

$$min(f(x)) = 80, \quad (x_0, x_1) = (1, 1) \tag{5}$$

3.1 Задача 4.1

3.1.1 Метод конфигураций (метод Хука-Дживса).

Метод конфигураций предназначен для решения задач оптимизации целевой функции многих переменных, при этом целевая функция не обязательно гладкая. Этот метод представляет собой комбинацию исследующего поиска с циклическим изменением переменных и ускоряющего поиска по образцу. Исследующий поиск ориентирован на выявление локального поведения целевой функции и определения направления её убывания вдоль «оврагов». Полученная информация используется при поиске по образцу при движении вдоль «оврагов».

3.1.2 Метод Нелдера-Мида.

Метод Нелдера-Мида, также известный как метод деформируемого многогранника и симплекс-метод, - метод безусловной оптимизации функции от нескольких переменных, не использующий производной (точнее — градиентов) функции, а поэтому легко применим к негладким и/или зашумлённым функциям.

Метод деформируемых симплексов (не следует путать с симплекс-методом линейного программирования) также как и метод Хука-Дживса относится к классу методов 0-го порядка.

Суть метода заключается в последовательном перемещении и деформировании симплекса вокруг точки экстремума. Метод находит локальный экстремум и может «застрять» в одном из них. Если всё же требуется найти глобальный экстремум, можно пробовать выбирать другой начальный симплекс. Более развитый подход к исключению локальных экстремумов предлагается в алгоритмах, основанных на методе Монте-Карло, а также в эволюционных алгоритмах.

3.2 Задача 4.2

3.2.1 Метод наискорейшего градиентного спуска.

Стратегия решения задачи состоит в построении последовательности точек $x^k, k = 0, 1$ таких, что $f(x^{k+1}) < f(x^k)$. Точки последовательности x^k вычисляются по правилу $x^{k+1} = x^k - \alpha^k \bigtriangledown f(x^k)$, где точка x^0 задается пользователем; величина шага α^k определяется для каждого значения из условия: $\phi(\alpha^k) = f(x^k - \alpha^k \bigtriangledown f(x^k)) \to \min_{\alpha^k}$.

3.2.2 Метод Флетчера-Ривза.

Стратегия метода Флетчера-Ривза состоит в построении последовательности точек x^k, таких, что $f(x^{k+1}0 < f(x^k)$. Точки последовательности x^k вычисляются по правилу:

$$x^{k+1} = x^k + \alpha^k d^l \tag{6}$$

$$d^k = - \bigtriangledown f(x^k) + w^{k-1} d^{k-1} \tag{7}$$

$$d^0 = - \bigtriangledown f(x^0) \tag{8}$$

$$w^{k-1} = \frac{|| \bigtriangledown f(x^k)||^2}{|| \bigtriangledown f(x^{k-1})||^2} \tag{9}$$

Точка задается пользователем, величина шага α^k определяется для каждого значения k из условия $\alpha^k = Arg \min_{\alpha \in R} f(x^k + \alpha^k d^k)$.

Решение задачи одномерной минимизации может осуществляться либо из условия $\frac{d\phi(\alpha^k)}{d\alpha^k} = 0$, $\frac{d^2\phi(\alpha^k)}{d\alpha^{k^2}}$, либо численно, с использованием методов многомерной минимизации, когда решается задача: $\phi(\alpha^k) \to \min_{\alpha^k \in [a,b]}$.

3.2.3 Метод Девидона-Флетчера-Пауэлла.

Стратегия метода Девидона-Флетчера-Пауэлла состоит в построении последовательности точек x^k, таких, что $f(x^{k+1}) < f(x^k)$. Точки последовательности x^k вычисляются по правилу $x^{k+1} = x^k - \alpha^k G^{k+1} \bigtriangledown f(x^k)$, где G^{k+1} - матрица размера nxn, являющаяся аппроксимацией обратной матрицы Гессе.

3.2.4 Метод Левенберга-Марквардта.

Стратегия метода Левенберга-Марквардта состоит в построении последовательности точек x^k, таких, что $f(x^{k+1}) < f(x^k)$. Точки последовательности x^k вычисляются по правилу

$$x^{k+1} = x^k - [H(x) + \mu^k E]^{-1} \bigtriangledown f(x^k), \tag{10}$$

где точка x^0 задается пользователем, E - единичная матрица, μ^k - последовательность положительных чисел, таких, что матрица $[H(x) + \mu^k E]^{-1}$ положительно определена.

4 Практическая реализация

Все методы были реализованы на языке программирования **Python**.

4.1 Задача 4.1

Листинг 1. Метод конфигураций.

```
def hooke_jeeves(dim, start_point, rho, eps, f):
    print_function.print_start("Hooke Jeeves")

    new_x = start_point.copy()
    x_before = start_point.copy()

    delta = np.zeros(dim)

    for i in range(0, dim):
        if start_point[i] == 0.0:
            delta[i] = rho
        else:
            delta[i] = rho * abs(start_point[i])

    step_length = rho
    k = 0
    f_before = f(new_x)

    while k < maxIterations and eps < step_length:

        k = k + 1

        for i in range(0, dim):
            new_x[i] = x_before[i]

        new_x, new_f = best_near_by(new_x, f, 0)

        keep = True

        while new_f < f_before and keep:

            for i in range(0, dim):
                if new_x[i] <= x_before[i]:
                    delta[i] = - abs(delta[i])
                else:
```

```
                    delta[i] = abs(delta[i])
                tmp = x_before[i]
                x_before[i] = new_x[i]
                new_x[i] = new_x[i] + new_x[i] - tmp

            f_before = new_f
            new_x, new_f = best_near_by(new_x, f, 0)

            if f_before <= new_f:
                break
            keep = False

            for i in range(0, dim):
                if 0.5 * abs(delta[i]) < abs(new_x[i] - x_before[i]):
                    keep = True
                    break

        if eps <= step_length and f_before <= new_f:
            step_length = step_length * rho
            for i in range(0, dim):
                delta[i] = delta[i] * rho

    end_point = x_before.copy()

    print_function.print_end_function(k, end_point, f)
    return end_point
```

Листинг 2. Метод Нелдера-Мида.

```
def nelder_mead(f, x_start, step=0.1, no_improve_thr=10e-6,
    no_improve_break=10, max_iter=0, alpha=1., gamma=2.,
                rho=-0.5, sigma=0.5):
    print_function.print_start("Nelder Mead")
    # init
    dim = len(x_start)
    prev_best = f(x_start)
    no_improve = 0
    res = [[x_start, prev_best]]

    for i in range(dim):
        x = copy.copy(x_start)
        x[i] = x[i] + step
        score = f(x)
        res.append([x, score])

    # simplex iter
```

```
    k = 0
    while 1:
        # order
        res.sort(key=lambda elem: elem[1])
        best = res[0][1]

        # break after max_iter
        if max_iter and k >= max_iter:
            print_function.print_end_function(k, res[0][0], f)
            return res[0][0]
        k += 1

        if best < prev_best - no_improve_thr:
            no_improve = 0
            prev_best = best
        else:
            no_improve += 1

        if no_improve >= no_improve_break:
            print_function.print_end_function(k, res[0][0], f)
            return res[0][0]

        # centroid
        x0 = [0.] * dim
        for tup in res[:-1]:
            for i, c in enumerate(tup[0]):
                x0[i] += c / (len(res) - 1)

        # reflection
        xr = x0 + alpha * (np.array(x0) - res[-1][0])
        r_score = f(xr)
        if res[0][1] <= r_score < res[-2][1]:
            del res[-1]
            res.append([xr, r_score])
            continue

        # expansion
        if r_score < res[0][1]:
            xe = x0 + gamma * (np.array(x0) - res[-1][0])
            e_score = f(xe)
            if e_score < r_score:
                del res[-1]
                res.append([xe, e_score])
                continue
            else:
                del res[-1]
                res.append([xr, r_score])
                continue

```

```
66          # contraction
67          xc = x0 + rho * (np.array(x0) - res[-1][0])
68          c_score = f(xc)
69          if c_score < res[-1][1]:
70              del res[-1]
71              res.append([xc, c_score])
72              continue
73
74          # reduction
75          x1 = res[0][0]
76          n_res = []
77          for tup in res:
78              red_x = x1 + sigma * (tup[0] - x1)
79              score = f(red_x)
80              n_res.append([red_x, score])
81          res = n_res
```

4.2 Задача 4.2

Листинг 3. Метод наискорейшего градиентного спуска.

```
1   def gradient_descend(x0, eps1, eps2, f, gradient):
2       print_function.print_start("Gradient Descend")
3
4       xk = x0[:]
5       k = 0
6       while True:
7           gradient_value = gradient(xk)
8
9           if np.linalg.norm(gradient_value) < eps1:
10              print_function.print_end_function(k, xk, f)
11              return xk
12          if k >= maxIterations:
13              print_function.print_end_function(k, xk, f)
14              return xk
15
16          t = 0.0
17          min_value_func = f(xk - t * gradient_value)
18          for i in np.arange(0.0, 2.0, 0.001):
19              if i == 0.0:
20                  continue
21              func_value = f(xk - i * gradient_value)
22              if func_value < min_value_func:
```

```
                min_value_func = func_value
                t = i

        xk_new = xk - t * gradient_value

        if np.linalg.norm(xk_new - xk) < eps2 and np.linalg.norm(f(xk_new) -
        ↪  f(xk)):
            print_function.print_end_function(k - 1, xk_new, f)
            return
        else:
            k += 1
            xk = xk_new
```

Листинг 4. Метод Флетчера-Ривза.

```
def flatcher_rivz(x0, eps1, eps2, f, gradient):
    print_function.print_start("Flatcher-Rivz")

    xk = x0[:]
    xk_new = x0[:]
    xk_old = x0[:]

    k = 0
    d = []

    while True:
        gradient_value = gradient(xk)

        if np.linalg.norm(gradient_value) < eps1:
            print_function.print_end_function(k, xk, f)
            return

        if k >= maxIterations:
            print_function.print_end_function(k, xk, f)
            return
        if k == 0:
            d = -gradient_value
        beta = np.linalg.norm(gradient(xk_new)) /
        ↪  np.linalg.norm(gradient(xk_old))
        d_new = np.add(-gradient(xk_new), np.multiply(beta, d))
        t = 0.1
        min_value_func = f(xk + t * d_new)
        for i in np.arange(0.0, 1.0, 0.001):
            if i == 0.0:
                continue
            func_value = f(xk + i * d_new)
```

```
            if func_value < min_value_func:
                min_value_func = func_value
                t = i
        xk_new = xk + t * d_new
        if np.linalg.norm(xk_new - xk) < eps2 and np.linalg.norm(f(xk_new) -
        ↪  f(xk)):
            print_function.print_end_function(k - 1, xk_new, f)
            return
        else:
            k += 1

            xk_old = xk
            xk = xk_new
            d = d_new
```

Листинг 5. Метод Девидона-Флетчера-Пауэлла.

```
def davidon_flatcher_powell(x0, eps1, eps2, f, gradient):
    print_function.print_start("Davidon-Flatcher-Powell")
    eps1 /= 100
    eps2 /= 100
    k = 0
    xk_new = copy.deepcopy(x0[:])
    xk_old = copy.deepcopy(x0[:])

    a_new = np.eye(2, 2)
    a_old = np.eye(2, 2)

    while True:
        gradient_value = gradient(xk_old)

        if np.linalg.norm(gradient_value) < eps1:
            print_function.print_end_function(k, xk_old, f)
            return xk_old

        if k >= maxIterations:
            print_function.print_end_function(k, xk_old, f)
            return xk_old
        if k != 0:
            delta_g = gradient(xk_new) - gradient_value
            delta_x = xk_new - xk_old

            num_1 = delta_x @ delta_x.T
            den_1 = delta_x.T @ delta_g

```

```
            num_2 = a_old @ delta_g @ delta_g.T * a_old
            den_2 = delta_g.T @ a_old @ delta_g
            a_c = num_1 / den_1 - num_2 / den_2
            a_old = a_new
            a_new = a_old + a_c

        minimizing_function = minimizing(xk_new, a_new @ gradient_value.T, f)

        alpha = find_min(0.0, minimizing_function)

        xk_old = xk_new
        xk_new = xk_old - alpha * a_new @ gradient_value
        if np.linalg.norm(xk_new - xk_old) < eps2 and
        ↪  np.linalg.norm(f(xk_new) - f(xk_old)) < eps2:
            print_function.print_end_function(k - 1, xk_new, f)
            return xk_new
        else:
            k += 1
```

Листинг 6. Метод Левенберга-Марквардта.

```
def levenberg_markvardt(x0, eps1, f, gradient, hessian):
    print_function.print_start("Levenberg-Markvardt")
    k = 0
    xk = x0[:]
    nu_k = 10 ** 4
    while True:
        gradient_value = gradient(xk)
        if np.linalg.norm(gradient_value) < eps1:
            print_function.print_end_function(k, xk, f)
            return xk
        if k >= maxIterations:
            print_function.print_end_function(k, xk, f)
            return xk
        while True:
            hess_matrix = hessian(xk)
            temp = np.add(hess_matrix, nu_k * np.eye(2))
            temp_inv = np.linalg.inv(temp)
            d_k = - np.matmul(temp_inv, gradient_value)
            xk_new = xk + d_k
            if f(xk_new) < f(xk):
                k += 1
                nu_k = nu_k / 2
                xk = xk_new
                break
```

```
            else:
                nu_k = 2 * nu_k
        continue
```

5 Результаты.

При последовательном запуске всех алгоритмов со следующими параметрами:

$$\epsilon = 10^{-4} \tag{11}$$

были получены следующие результаты:

Листинг 7. Результаты выполнения программ.

```
Start Hooke Jeeves
         Iteration(s): 13
         f([0.9995, 0.9995]) = {80.00000799250188}

Start Gradient Descend
         Iteration(s): 544
         f([0.99379591 0.98751589]) = {80.00007737427414}

Start Flatcher-Rivz
         Iteration(s): 18
         f([1.00006056 1.00012813]) = {80.00000000881083}

Start Davidon-Flatcher-Powell
         Iteration(s): 350
         f([1.00039361 1.0007921 ]) = {80.00000031052592}

Start Levenberg-Markvardt
         Iteration(s): 19
         f([0.99998791 0.99997543]) = {80.00000000029688}
```

Все результаты с небольшой погрешностью совпадают с результатами полученными с помощью сервиса WolframAlpha.com в пункте 3.