Национальный исследовательский университет ИТМО
Факультет информационных технологий и программирования
Прикладная математика и информатика

Методы оптимизации

Отчет по лабораторной работе №3

Работу
выполняли:
Кольченко Антон M32371
Гайнанов Ильдар M32371
Муфтиев Руслан M32331

Санкт-Петербург
2023

Введение

Постановка задачи:

1. Реализовать методы Gauss-Newton и Powell Dog Leg для решения нелинейной регрессии. Сравнить эффективность с методами, реализованными в предыдущих работах.

2. Реализовать метод BFGS и исследовать его сходимость при минимизации различных функций. Сравнить с другими реализованными методами.

3. Реализовать и исследовать метод L-BFGS.

Глава 1

Теоретическая часть

1.1 Методы регрессии

Вход: множество точек $\{(x_i, y_i) \mid x_i, y_i \in \mathbb{R}\}$
Выход: зависимость y от x, выраженная как $\sum_{i=0}^{p} c_i x^i + \epsilon$.
Далее: $w := \{c_1, c_2, ...c_p\}$
$r(w) := y - f(x, w)$, где f - функция, для которой мы восстанавливаем коэффициенты.

Метод Гаусса-Ньютона

Принцип работы:

Метод Гаусса-Ньютона находит решение задачи поиска наименьших квадратов, используя якобиану J функции ошибки. Он основан на методе Ньютона, который использует гессиан (матрицу вторых производных), который слишком долго высчитывается для того, чтобы быть применимым в реальной жизни. Вместо этого гессиан апроксимируется сочетанием якобианов (матриц первой производной).

Алгоритм:

1. $J := J_r(w)$.
2. $\delta = -(J^T J)^{-1} J^T r(w)$.
3. $w = w + \delta$.
4. Повторять шаги, пока $|\Delta MSE| > \varepsilon$.

Метод Powell's Dog Leg

Принцип работы:

Метод Dog Leg объединяет градиентный спуск и метод Гаусса-Ньютона для решения задачи наименьших квадратов. Более того, в алгоритме используется доверительная область TR с радиусом R, которой будут ограничены размеры шага алгоритма.

$TR_w := \{w' : dist(w, w') < R\}$
Алгоритм:

1. $J := J_r(w)$
2. $\delta_{Gauss-Newton} = -(J^T J)^{-1} J^T r(w)$.
3. $\delta_{GradientDescent} = -J^T r(w)$.
4. $t = -\dfrac{\delta^T_{GradientDescent} \cdot J^T \cdot r(w)}{||J \cdot \delta_{GradientDescent}||^2}$.
5. $\delta = \begin{cases} \delta_{Gauss-Newton} & \delta_{Gauss-Newton} \in TR_w, \\ R \cdot \dfrac{\delta_{GradientDescent}}{||\delta_{GradientDescent}||} & \delta_{Gauss-Newton} \notin TR_w,\ \delta_{GradientDescent} \notin TR_w, \\ t\delta_{GradientDescent} + R(\delta_{Gauss-Newton} - t \cdot \delta_{GradientDescent}) & \text{иначе.} \end{cases}$
6. Повторять шаги, пока $|\Delta MSE| > \varepsilon$.

1.2 BFGS, L-BFGS

Вход: функция $f : \mathbb{R}^n \to \mathbb{R}$, стартовая точка $x = (x_1,x_2,...,x_n)$, точность ε.
Выход: найденная точка локального минимума

BFGS

Принцип работы:

В BFGS и производных мы пользуемся не только градиентом (т.е. вектором первых производных), но и гессианом (матрицей вторых производных). Однако гессиан вычисляется слишком долго, поэтому будем его апроксимировать (а точнее не его, а обратную к нему матрицу).

C - апроксимация H^{-1}. $C^{[0]}$ можно задать как честный H^{-1} или как I. $A \times B$ - здесь внешнее произведение A и B. Алгоритм:

1. $p^{[k+1]} = -C^{[k]} \cdot \nabla f(x^{[k]})$.
2. $\delta^{[k+1]} = \alpha p^{[k+1]}$, где α находится по признаку Вольфе.
3. $x^{[k+1]} = x^{[k]} + \delta$
4. $y^{[k+1]} = \nabla f(x^{[k+1]}) - \nabla f(x^{[k]})$
5. $\rho^{[k+1]} = \dfrac{1}{(y^{[k+1]})^T \cdot \delta^{[k+1]}}$

6. $C^{[k+1]} = (I - \rho \cdot \delta^{[k+1]} \times (y^{[k+1]})^T) C^{[k]} (I - \rho \cdot y^{[k+1]} \times (\delta^{[k+1]})^T) + \rho \cdot \delta^{[k+1]} \times (\delta^{[k+1]})^T$
7. Повторять шаги, пока $||\nabla f(x^{[k]})|| > \varepsilon$.

L-BFGS

Вход: дополнительно m - количество предыдущих сохраненных итераций алгоритма.
Принцип работы:

У BFGS есть существенный недостаток - расход памяти, т.к. гессиан занимает $O(n^2)$ памяти. L-BFGS частично жертвует точностью, чтобы занимать $O(mn)$ памяти ($m << n$).

C - апроксимация H^{-1}. $C^{[0]}$ можно задать как честный H^{-1} или как I. $A \times B$ - здесь внешнее произведение A и B. Алгоритм:

1. $q = \nabla f(x^{[k]})$.
2. Для каждого $i = k-1, k-2, ..., k-m$:
   $\alpha^{[i]} = \rho^{[i]} \cdot (s^{[i]})^T \times q$
   $q = q - \alpha^{[i]} \cdot y^{[i]}$.
3. $\gamma = \dfrac{(s^{[k]})^T \cdot y^{[k]}}{(y^{[k]}) \cdot y^{[k]}}$.
4. $C = \gamma I$.
5. $z = Cq$.
6. Для каждого $i = k-m, k-m+1, ..., k-1$:
   $\beta^{[i]} = \rho^{[i]} \cdot (y^{[i]})^T \times z$
   $z = z + s^{[i]} \cdot (\alpha^{[i]} - \beta^{[i]})$.
7. $p^{[k+1]} = -z$
8. $\delta^{[k+1]} = \alpha p^{[k+1]}$, где α находится по признаку Вольфе.
9. $x^{[k+1]} = x^{[k]} + \delta$
10. $y^{[k+1]} = \nabla f(x^{[k+1]}) - \nabla f(x^{[k]})$
11. $\rho^{[k+1]} = \dfrac{1}{(y^{[k+1]})^T \cdot \delta^{[k+1]}}$
12. Повторять шаги, пока $||\nabla f(x^{[k]})|| > \varepsilon$.

Глава 2

Практическая часть

2.1 Методы решения задачи нелинейной регрессии

Реализация метода Гаусса-Ньютона

```
1   def gauss_newton(p, points, num_iters=100):
2       w = np.zeros(p+1)
3       A = np.array([[x ** i for i in range(p + 1)] for x in points[:, 0]])
4       x = points[:, 0]
5       y = points[:, 1]
6       r = lambda w: y - np.array(sum(w[i] * x ** i for i in range(p + 1)))
7       err = lambda w: sum(map(lambda ri: ri ** 2, r(w)))
8       for cnt in range(num_iters):
9           J = jacobian(r, w)
10
11          delta = -(np.linalg.inv(J.T @ J) @ J.T @ r(w))
12
13          prev_err = err(w)
14          w += delta
15          # print(sum(w[i] * x ** i for i in range(p + 1)))
16
17          if abs(err(w) - prev_err) < 1e-5:
18              break
19
20      return w, cnt
21
```

Листинг 2.1: Метод Гаусса-Ньютона

Реализация метода Dog Leg

```
def dog_leg(p, points, trust_region=1, num_iters=100):
  def step(J, r: np.ndarray):
          gn_delta = -np.linalg.inv(J.T @ J) @ J.T @ r # gauss-newton
          if np.linalg.norm(gn_delta) <= trust_region:
              return gn_delta

          st_delta = -J.T @ r # gradient descent
          if np.linalg.norm(st_delta) > trust_region:
              return st_delta / np.linalg.norm(st_delta) * trust_region

          t = (np.linalg.norm(st_delta) / np.linalg.norm(J @ st_delta)) **
 2
          return t * st_delta + trust_region * (gn_delta - t * st_delta)

      w = np.zeros(p+1)
      A = np.array([[x ** i for i in range(p + 1)] for x in points[:, 0]])
      x = points[:, 0]
      y = points[:, 1]
      r = lambda w: y - np.array(sum(w[i] * x ** i for i in range(p + 1)))
      err = lambda w: sum(map(lambda ri: ri ** 2, r(w)))
      for cnt in range(num_iters):
          J = jacobian(r, w)

          delta = step(J, r(w))

          prev_err = err(w)
          w += delta

          if abs(err(w) - prev_err) < 1e-5:
              break


      return w, cnt

```

Листинг 2.2: Метод Powell's Dog Leg

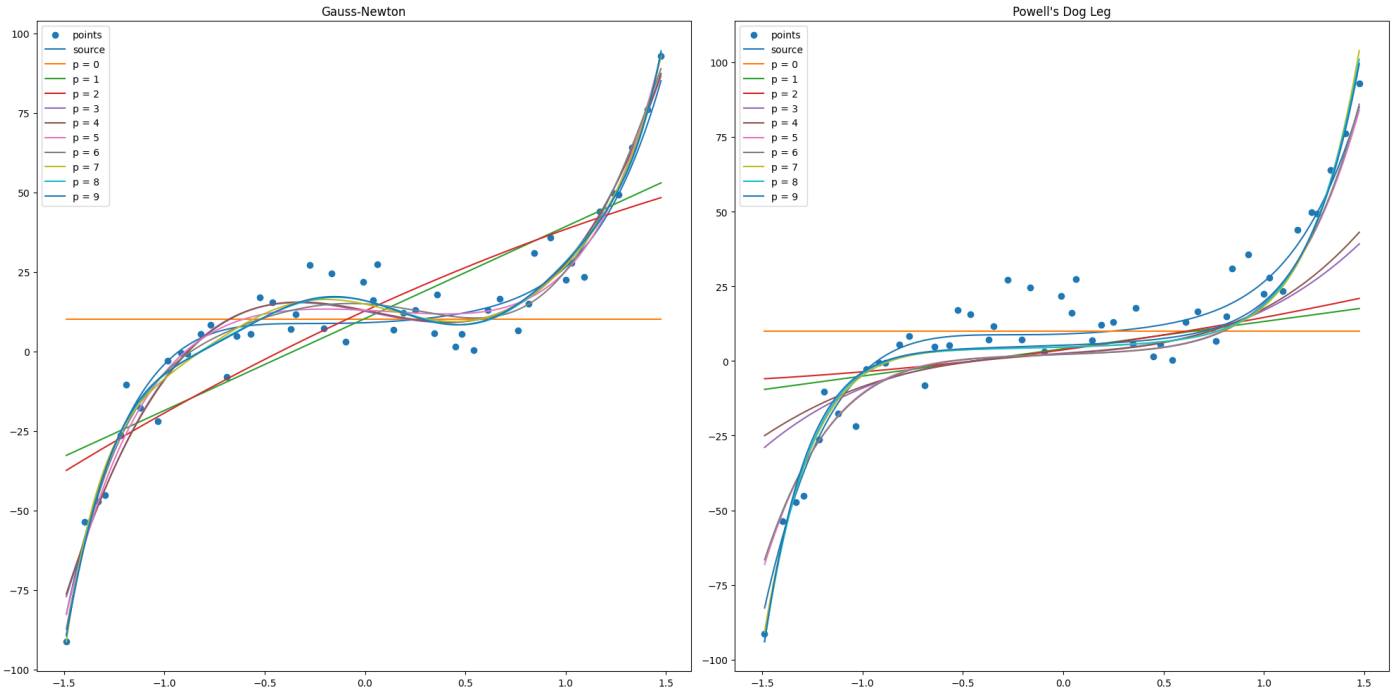

Рис. 2.1: График сходимости методов на полиномиальной функции

Сравнение

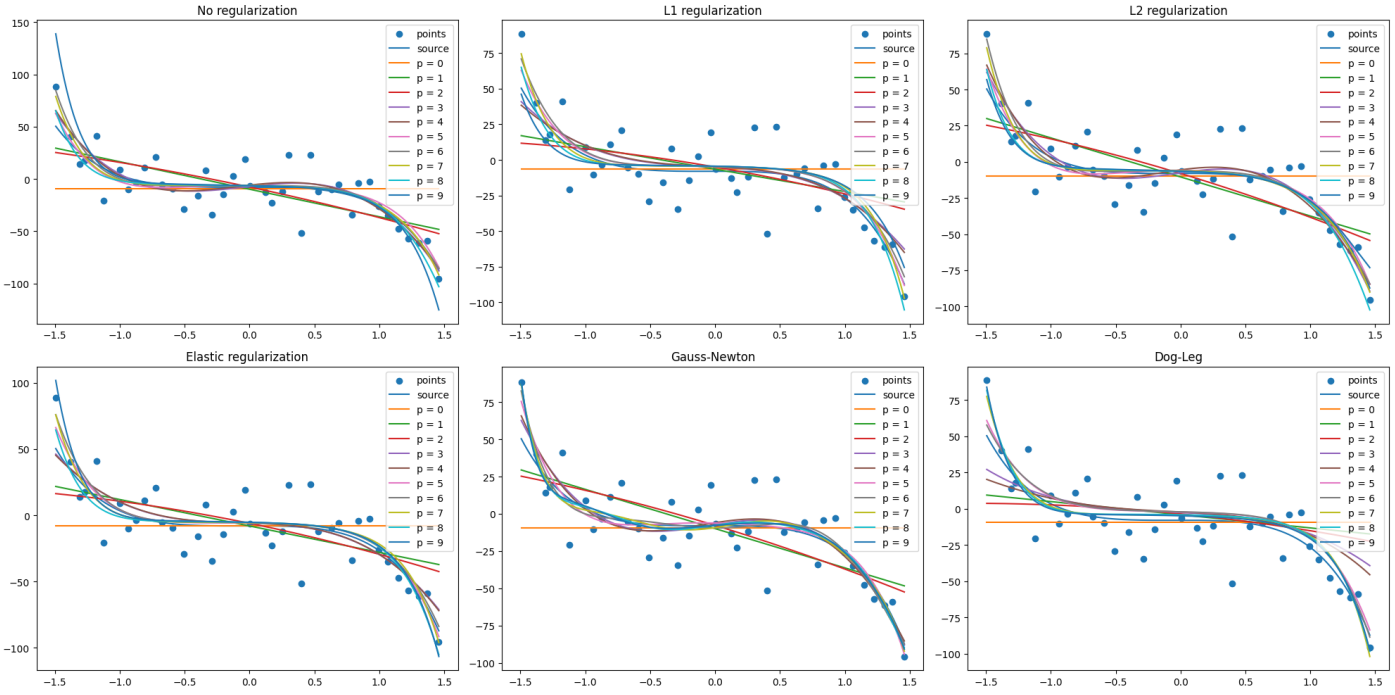

Рис. 2.2: График сходимости методов на полиномиальной функции

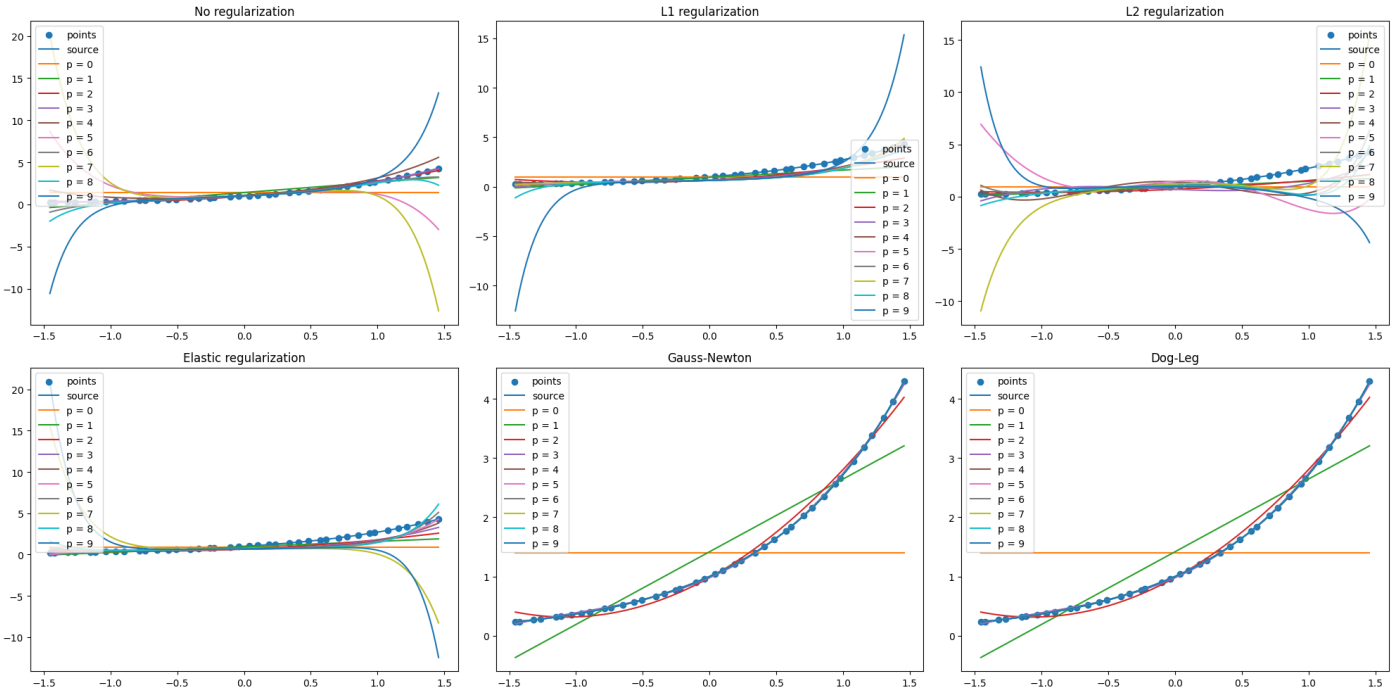

Рис. 2.3: График сходимости методов на экспоненциальной функции

Выводы

- Метод Гаусса-Ньютона сходится быстрее всех остальных, даже чем Dog Leg. Полиномиальные регрессии он нередко находит за 1 итерацию.
- Тем не менее, метод Гаусса-Ньютона имеет склонность к переобучению, в отличие от Dog Leg.
- Оба метода умеют решать задачу поиска регрессии для функций, которые даже не являются полиномиальными.

2.2 BFGS, L-BFGS

Реализация

```
def fast_bfgs_gd(f, x, lim=500):
    n = len(x)
    points = []
    g = None
    C = np.linalg.inv(hessian(f, x))
    points.append(x)
    while True:
        if g is None:
            g = grad(f, x)

        if np.linalg.norm(g) < eps:
            break

        p = -C @ g

        alpha = find_wolfe(f, x, p)
        delta = p * alpha
        x = x + delta
        points.append(x)

        if (len(points) > lim):
            break

        newg = grad(f, x)
        y = newg - g
        g = newg

        I = np.eye(n)
        rho = 1 / (y.T @ delta)
        C = (I - rho * np.outer(delta, y.T)) @ C @ (I - rho * np.outer(y,
 delta.T)) + \
            rho * np.outer(delta, delta.T)

    return np.array(points)
```

Листинг 2.3: Реализация BFGS

```
def l_bfgs_gd(f, x, m=8, lim=500):
    n = len(x)
    points = []
    rho_q = deque(maxlen=m)
    s_q = deque(maxlen=m)
    y_q = deque(maxlen=m)
    g = None
    points.append(x)
    while True:
        if g is None:
            g = grad(f, x)

```

```
13          if np.linalg.norm(g) < eps:
14              break
15
16          alpha_q = []
17
18          q = g
19          for s, rho, y in zip(reversed(s_q), reversed(rho_q), reversed(y_q))
    :
20              alpha = rho * np.outer(s.T, q)
21              alpha_q.append(alpha)
22              q = q - alpha @ y
23
24          try:
25              gamma = (s_q[-1].T @ y_q[-1]) / (y_q[-1].T @ y_q[-1])
26              H = gamma * np.eye(n)
27          except IndexError:
28              H = np.linalg.inv(hessian(f, x))
29
30          z = H @ q
31
32          for s, rho, y, alpha in zip(s_q, rho_q, y_q, reversed(alpha_q)):
33              beta = rho * np.outer(y.T, z)
34              z = z + s @ (alpha - beta)
35
36          p = -z
37          alpha = find_wolfe(f, x, p)
38          delta = p * alpha
39          s_q.append(delta)
40          x = x + delta
41          points.append(x)
42
43          if (len(points) > lim):
44              break
45
46          newg = grad(f, x)
47          y = newg - g
48          y_q.append(y)
49          g = newg
50
51          rho = 1 / (y.T @ delta)
52          rho_q.append(rho)
53
54      return np.array(points)
55
```

Листинг 2.4: Реализация L-BFGS

Рассмотрим вычислительные затраты методов ($dim = 100$):

	time, s	iters	mem usage	∇	f
SGD	14.5	46	238951	n	$2n$
Momentum	48.4	155	376662	n	$2n$
Nesterov	10.0	32	245543	n	$2n$
AdaGrad	57.5	184	409672	n	$2n$
RMSProp	18.3	59	256806	n	$2n$
Adam	39.4	125	356683	n	$2n$
BFGS	58.6	28	15426729	n	$2n$
L-BFGS	60.7	29	15432309	n	$2n$

Таблица 2.1: Таблица расхода ресурсов на использование методов

Примечание: в нашей реализации подсчет градиента также вызывает функцию. Такие вызовы функции из таблицы были исключены. n - количество итераций.

Сравнение

Выводы

Как в 2 лабораторной работе, если судить только по количеству итераций, то BFGS и L-BFGS обеспечивают более быструю сходимость, при этом не требуя подбора размера шага и подобных параметров. Более того, использование гессиана позволяет этим методам сходиться на функциях, которые считаются нетривиальными для других алгоритмов, например, функции Розенброка. Однако:

- Даже с апроксимацией гессиана методы *-BFGS используют максимальное среди всех методов количество ресурсов. Рекомендется использовать эти методы тогда, когда доподлинно известно, что функция будет сложной.

- L-BFGS при слишком малых размерностях не только не имеет выигрыша в памяти, но и проигрывает во времени исполнения и точности. В моем примере BFGS и L-BFGS используют примерно одинаковое количество памяти из-за особенностей сборщика мусора в языке Python, который очищал используемую память не сразу.

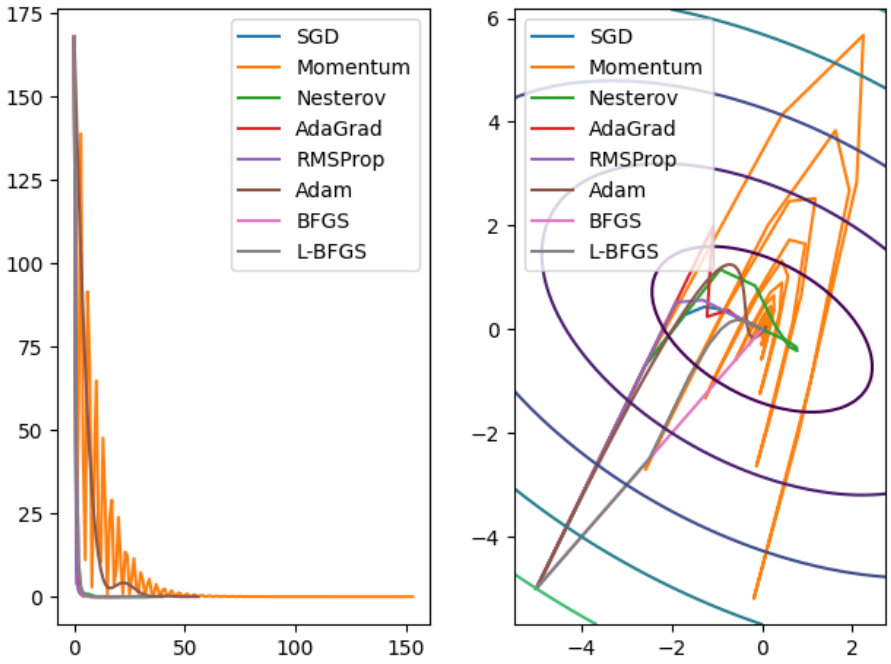

Рис. 2.4: Демонстрация графиков сходимости различных модификаций стохастического градиентного спуска

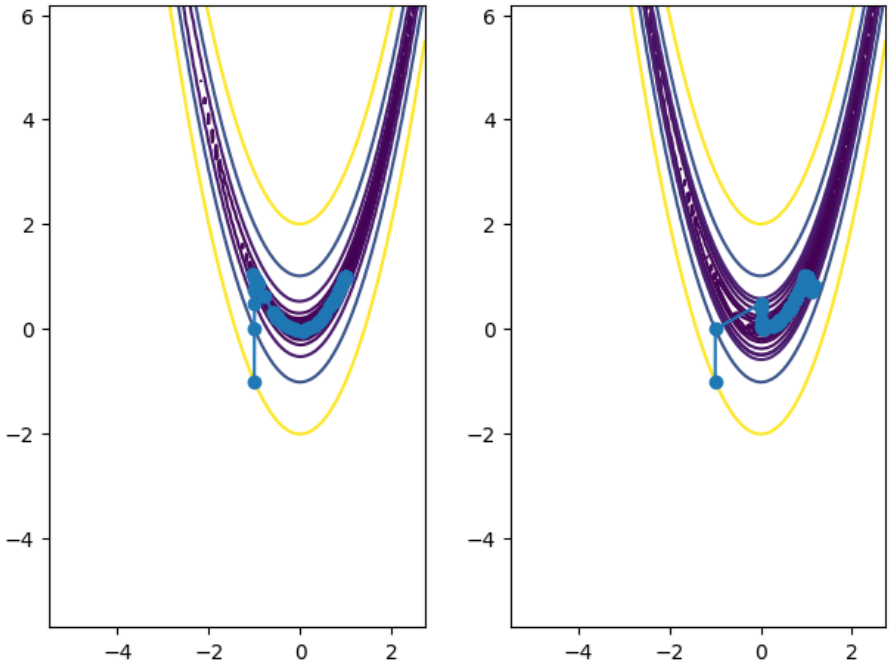

Рис. 2.5: Демонстрация графиков сходимости BFGS и L-BFGS на функции Розенброка