

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ  
Национальный исследовательский университет  
«Высшая школа Экономики»

ФАКУЛЬТЕТ ЭКОНОМИЧЕСКИХ НАУК

ОБРАЗОВАТЕЛЬНАЯ ПРОГРАММА «ЭКОНОМИКА»

---

## КУРСОВАЯ РАБОТА

Стохастические методы оптимизации

---

Выполнил:  
студент группы БЭК1812  
Хайкин ГЛЕБ АЛЕКСЕЕВИЧ

Научный руководитель:  
старший преподаватель  
Борзых Дмитрий Александрович



МОСКВА — 2020

# Оглавление

<b>1. Введение . . . . .</b>	<b>3</b>
<b>2. Методы и их применение . . . . .</b>	<b>4</b>
2.1. Имитация отжига . . . . .	4
2.1.1. Алгоритм . . . . .	4
2.1.2. N ферзей . . . . .	5
2.1.3. Минимизация негладкой функции . . . . .	10
2.1.4. Задача коммивояжера . . . . .	12
2.1.5. Вывод . . . . .	15
2.2. Метод роения частиц . . . . .	16
2.2.1. Алгоритм . . . . .	16
2.2.2. Функция Розенброка . . . . .	18
<b>Заключение . . . . .</b>	<b>24</b>
<b>Список литературы . . . . .</b>	<b>25</b>

# 1. Введение

jjj.

## 2. Методы и их применение

### 2.1. Имитация отжига

*Имитация отжига* (*Simulated Annealing*, SA) представляет собой алгоритм решения задачи поиску глобального оптимума некоторой функции  $F: \mathbb{X} \rightarrow \mathbb{R}$  через упорядоченный стохастический поиск, базирующийся на моделировании физического процесса кристаллизации вещества из жидкого состояния в твердое.

ДОБАВИТЬ ВВЕДНИЕ

#### 2.1.1. Алгоритм

Для описания метода рассмотрим задачу нахождения глобального минимума:

$$F(x) \rightarrow \min_{x \in \mathbb{X}}$$

где  $x = (x_1, \dots, x_m)$  — вектор всех состояний,  $\mathbb{X}$  — множество всех состояний.

Положим, что  $k = 0$  и изначально температура зафиксирована на определенном уровне  $T(k) = \text{const.}$

1. Из множества всех состояний выберем случайный элемент  $\hat{x}(k) \equiv x_i$ ,

$i \in (1, \dots, m)$ .

2. Понизим температуру одним из следующих способов:

а) Больцмановский отжиг

$$T(k) = \frac{T(0)}{\ln(1 + k)}, \quad k > 0 \tag{2.1}$$

б) Отжиг Коши

$$T(k) = \frac{T(0)}{k} \tag{2.2}$$

в) Метод тушения

$$T(k + 1) = \alpha T(k), \quad \alpha \in (0, 1) \tag{2.3}$$

3. Пусть следующий элемент зависит от функции из семейства симметричных вероятностных распределений  $G: \mathbb{X} \rightarrow \mathbb{X}$ , порождающей новое состояние:

$$\tilde{x}(k) \sim G(\hat{x}(k), T(k)).$$

a) Часто G выбирается из семейства нормальных распределений:

$$G(\tilde{x}; \hat{x}, T) = \frac{1}{\sqrt{(2\pi)^D T}} \exp \left\{ \frac{-|\tilde{x} - \hat{x}|^2}{2T} \right\}, \quad (2.4)$$

где  $\hat{x}$  — математическое ожидание,  $T$  — дисперсия,  $D$  — размерность пространства всех состояний.

b) Также для  $D = 1$  используется распределение Коши с плотностью:

$$G(\tilde{x}; \hat{x}, T) = \frac{1}{\pi} \frac{T}{|\tilde{x} - \hat{x}|^2 + T^2}, \quad (2.5)$$

где  $\hat{x}$  — параметр сдвига,  $T$  — параметр масштаба.

4. Рассчитываем разницу двух функций:

$$\Delta F = F(\tilde{x}(k)) - F(\hat{x}(k)).$$

5. Принимаем  $\tilde{x}(k)$  за новый элемент, то есть  $\hat{x}(k+1) \equiv \tilde{x}(k)$ , с вероятностью

$$\mathbb{P}(\{\hat{x}(k+1) = \tilde{x}(k)\}) = \begin{cases} 1, & \Delta F < 0 \\ \exp \left\{ -\frac{\Delta F}{T(k)} \right\}, & \Delta F \geq 0 \end{cases} \quad (2.6)$$

и отвергаем его, то есть  $\hat{x}(k+1) \equiv \hat{x}(k)$ , с вероятностью

$$q = 1 - \mathbb{P}(\{\hat{x}(k+1) = \tilde{x}(k)\}).$$

Заметим, чем выше температура, тем больше вероятность принять состояние хуже текущего ( $\Delta F \geq 0$ ).

6. Возвращаемся к пункту 2, пока не достигнем глобального минимума.

## 2.1.2. N ферзей

Рассмотрим задачу, в которой необходимо расставить  $N$  ферзей на шахматной доске размера  $N \times N$  так, чтобы ни один из них не «бил» другого.

В таком случае, множество всех состояний  $\mathbb{X}$  будет содержать всевозможные расстановки ферзей на шахматной доске. Общее число возможных расположений  $n$  ферзей на  $N \times N$ -клеточной доске равно:

$$\binom{N \times N}{n} = \frac{N \times N!}{n!(N \times N - n)!}$$

Тогда функция  $F : \mathbb{X} \rightarrow \mathbb{R}$  будет выдавать количество атак ферзей, и решением данной задачи будет нахождение такого расположения  $x^*$ , что  $F(x^*) \equiv 0$ .

Зафиксируем изначальное расположение ферзей на шахматной доске. Очевидно, что несколько ферзей не могут находиться на одной вертикали или горизонтали, ибо тогда они будут находиться под ударом друг-друга. Следовательно, наша задача сужается к поиску расположения:

$$x^* = (q_1, \dots, q_n) = \{(1, h_1), \dots, (n, h_n)\}, h_1 \neq \dots \neq h_n, \quad (2.7)$$

где  $(i, h_i)$  — расположение ферзя  $q_i$  на  $i$ -ой вертикали по горизонтали  $h_i$ .

Отметим, что такая задача имеет  $N!$  решений.

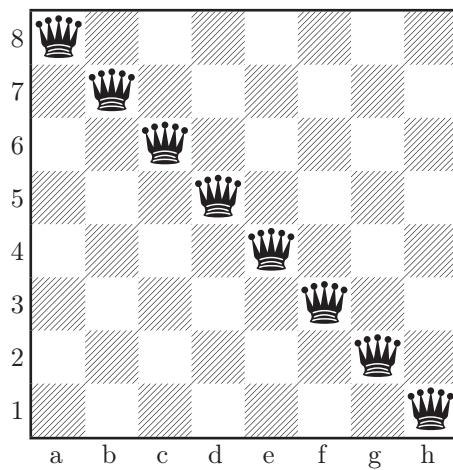


Рисунок 2.1 — Изначальное расположение.

Определим функцию, которая будет создавать изначальное неоптимальное расположение, в общем виде. Учтем, что несколько ферзей не могут находиться на одной вертикали или горизонтали.

```
In:  def queens(N):
      ver = np.arange(1, N + 1)
      hor = np.arange(1, N + 1)
      np.random.shuffle(hor)
      return np.column_stack((ver, hor)) # получаем массив
# размерности (N, 2), отображающий расположение ферзей
```

Выведем первоначальное расположение ферзей для стандартной доски  $8 \times 8$ , где первый столбец массива — расположение по вертикали, второй столбец массива — расположение по горизонтали. Для наглядности — презентации оптимизационного процесса — выстроим изначальную расстановку на главной диагонали (рис. 2.1).

```
In: matrix = queens(8)
matrix
```

```
Out: array([[1, 1],
           [2, 2],
           [3, 3],
           [4, 4],
           [5, 5],
           [6, 6],
           [7, 7],
           [8, 8]])
```

Функция F, которая выявляет количество атак ферзей, выглядит следующим образом:

```
In: def F(Q, N):
    cnt = 0
    for i in range(N):
        for j in range(i + 1, N):
            if abs(Q[i, 0] - Q[j, 0]) == abs(Q[i, 1] - Q[j, 1]):
                cnt += 1
    return cnt * 2 # учтемыаем взаимные атаки
```

Посмотрим, сколько у атак у исходной расстановки.

```
In: F(matrix, 8)
```

```
Out: 56
```

В нашей задачи функция G будет случайной незначительной перестановкой номеров горизонтали в исходном наборе.

```
In: def G(Q, N):
    pos = Q.copy()
    while True:
        i = np.random.randint(0, N - 1)
        j = np.random.randint(0, N - 1)
        if i != j:
            break
    pos[i, 1], pos[j, 1] = pos[j, 1], pos[i, 1]
    return pos # получаем новое расположение
```

Теперь выведем и сам метод имитации отжига.

```
In: def SA(Q, T, schedule):
    N = np.shape(Q)[0]
    x_hat = Q.copy()
```

```

while F(x_hat, N) != 0:
    x_tilda = G(x_hat, N)
    delta = F(x_tilda, N) - F(x_hat, N)
    prob = np.exp(- delta / T)
    if (delta < 0) or (prob >= np.random.random()):
        x_hat = x_tilda
    # используем метод тушения для понижения температуры
    T *= schedule
return x_hat

```

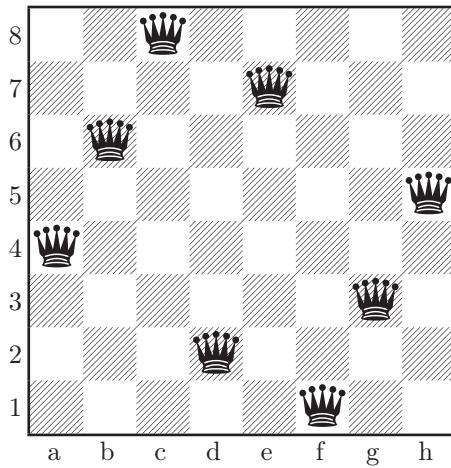


Рисунок 2.2 — Оптимальное расположение

Так для нашего примера с гиперпараметрами  $T(0) = 100, \alpha = 0.9$  мы получаем следующее оптимальное решение (рис. 2.2):

In: SA(matrix, 100, 0.9)

Out: array([[1, 4],  
 [2, 6],  
 [3, 8],  
 [4, 2],  
 [5, 7],  
 [6, 1],  
 [7, 3],  
 [8, 5]])

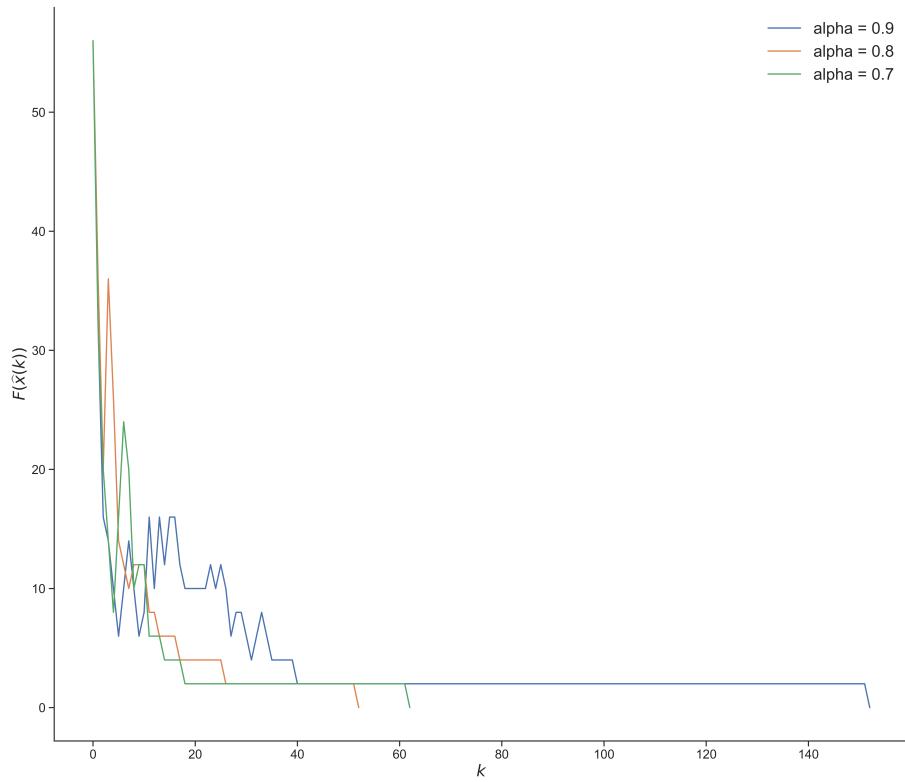


Рисунок 2.3 — Оптимизация расстановки 8 ферзей в зависимости от гиперпараметра  $\alpha$ .

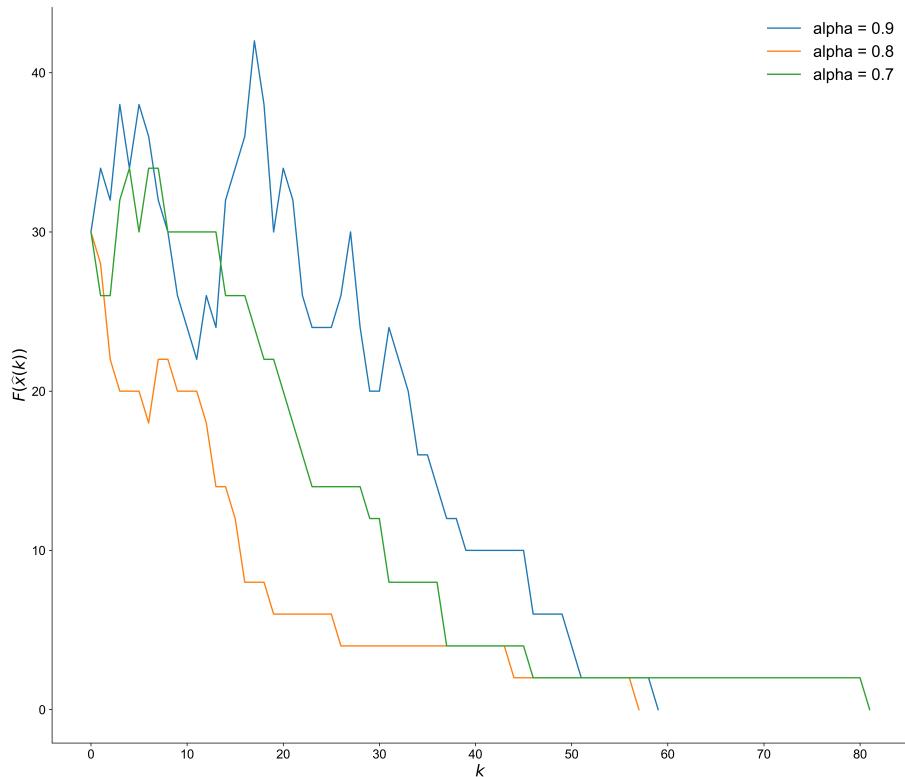


Рисунок 2.4 — Оптимизация расстановки 25 ферзей в зависимости от гиперпараметра  $\alpha$ .

### 2.1.3. Минимизация негладкой функции

Воспользуемся алгоритмом имитации отжига для нахождения глобального минимума следующей функции:

$$f(x) = x^2(1 + |\sin 80x|).$$

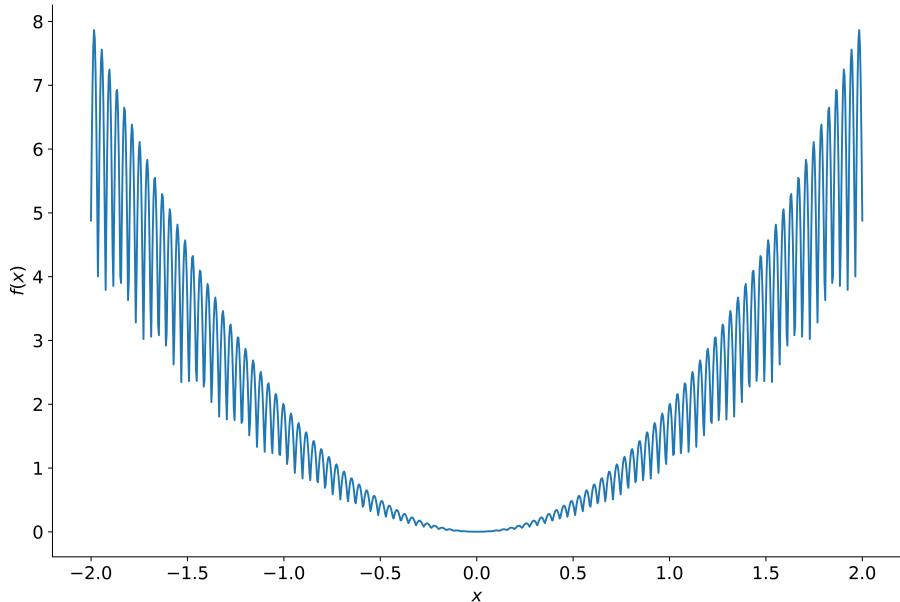


Рисунок 2.5

Стандартные методы оптимизации — к примеру, метод градиентного спуска — в данном случае не применимы. Вследствие наличия модуля эта функция не дифференцируема. Также она имеет очень большое количество локальных минимумов, что затрудняет, к примеру, мультистарт — запуск градиентного спуска из разных начальных направлений.

Применим наш алгоритм к данной задаче. Пусть  $T(0) = 0.6$ . Для понижения температуры будем использовать Больцмановский отжиг (2.1), а в качестве функции вероятностных распределений  $G$  — семейство нормальных распределений (2.4).

```
In: def SA(space, T, epsilon): # за space берется np.linspace(-2, 2, 1000)
    x_hat = np.random.choice(space)
    T_0 = T
    k = 1
    while True:
        x_tilda = np.random.normal(x_hat, T)
        delta = F(x_tilda) - F(x_hat)
        prob = np.exp(-delta / T)
```

```

if (delta < 0) or (prob >= np.random.random()):
    x_hat = x_tilda
if (x_hat < epsilon) and (x_hat > 0):
    return x_hat
T = T_0 / np.log(1 + k)
k += 1

```

Остановка итерационного процесса и скорость метода зависят от того, насколько близко мы хотим приблизиться к глобальному минимуму. Так, при точности  $10^{-1}$ , что довольно много, для 1000 повторений алгоритма метод отжига находит глобальный минимум в среднем за 1.97e-3 секунды со стандартным отклонением в 3.47e-5 секунды. Однако, увеличив точность до  $10^{-6}$ , среднюю скорость занимает уже 1.27 секунды со стандартным отклонением в 2.1e-5 секунды. Это наглядно представлено на рисунке 2.6.

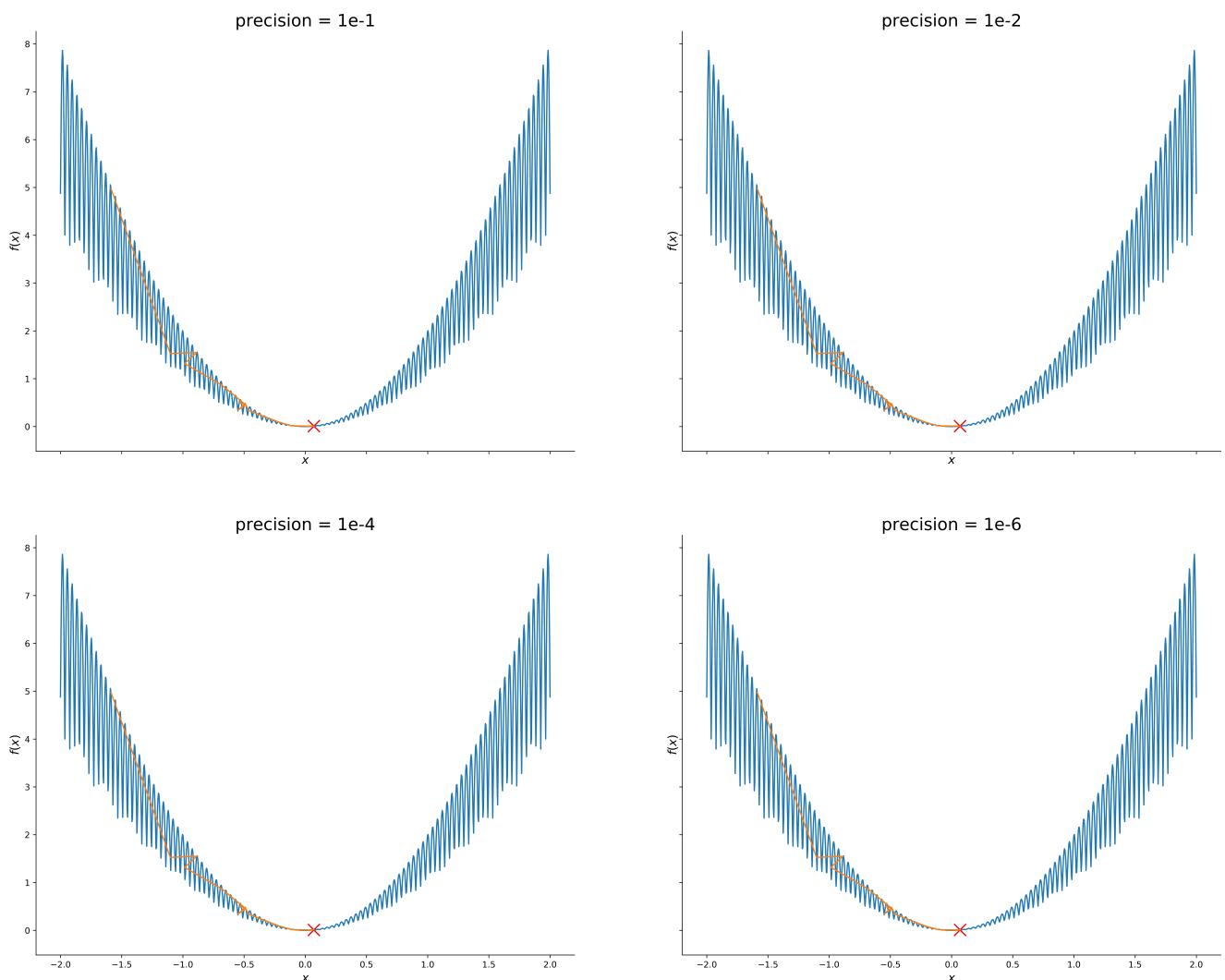


Рисунок 2.6 — Оптимизационный процесс в зависимости от точности.

## 2.1.4. Задача коммивояжера

*Задача коммивояжера* (*Traveling Salesman Problem*, TSP) является образцовым методом проверки многих оптимизационных алгоритмов и заключается в поиске кратчайшего маршрута между городами. Путь должен быть проложен так, чтобы маршрут единственно проходил через все города и его конечная точка совпадала с изначальной.

TSP имеет множество приложений в планировании и логистике, а также выступает в качестве подзадачи во многих других областях. В таком случае города могут представлять, к примеру, клиентов, а расстояние между городами — время или стоимость путешествия.

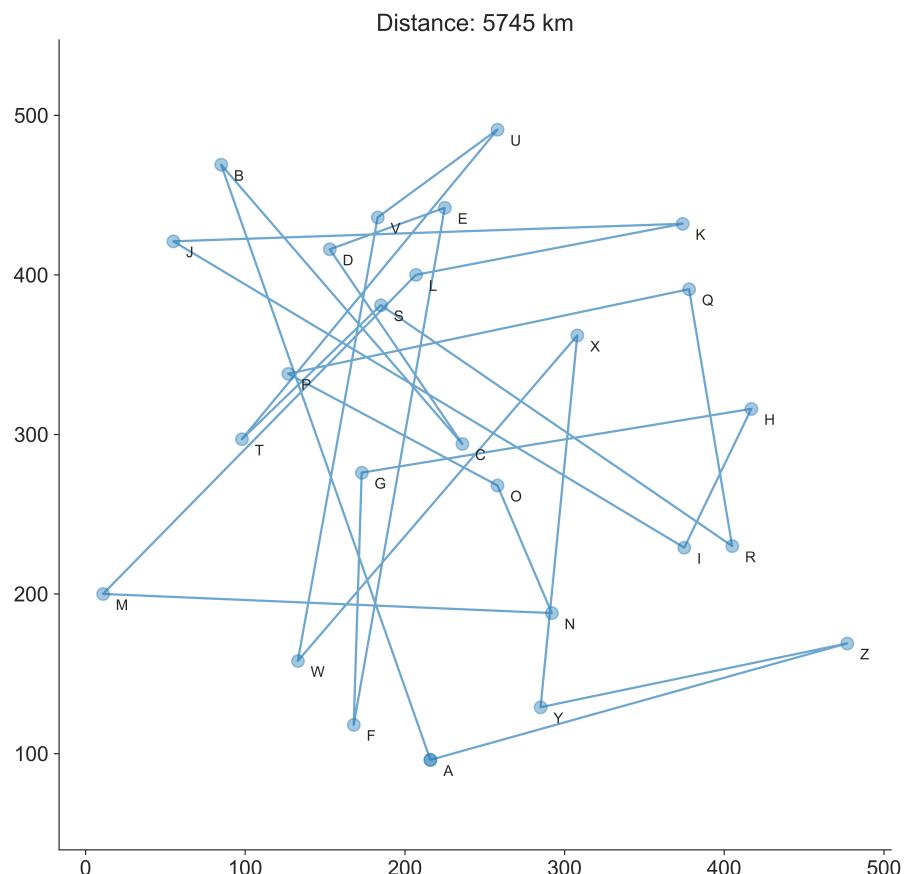


Рисунок 2.7 — Изначальный маршрут для 26-ти городов.

Для нашего примера создадим карту. Функция `map_city` будет принимать желаемое количество городов в качестве входных данных и выдавать два списка, из которых далее создается общий список кортежей. Первым элементом кортежа является наименование города, а вторым — его расположение в декартовой системе координат.

```
In: def map_city(cities_num):
    letters = [chr(i) for i in range(65, 65 + cities_num)]
    coord = np.random.randint(1, 500, size=(cities_num, 2))
    return letters, coord
```

Наш маршрут будет состоять из 26-ти городов.

```
In: names, cities = map_city(26)
store_val = list(zip(names, cities))
```

Определим расстояние от города  $i$  до всех остальных посредством функции `distance_dict`. Для измерения расстояния между городами будем использовать евклидову метрику. Напомним, что евклидово расстояние между точками  $x = (x_1, \dots, x_d)$  и  $u = (u_1, \dots, u_d)$  задается как:

$$d(x, u) = \sqrt{\sum_{i=1}^d (x_i - u_i)^2}$$

```
In: def distance_dict(cities, n):
    d = dict()
    for i in range(n):
        city = dict()
        for j in range(n):
            if i == j:
                continue
            c_a = cities[i][1]
            c_b = cities[j][1]
            dist = np.sqrt((c_a[0] - c_b[0])**2 + (c_a[1] - c_b[1])**2)
            city[cities[j][0]] = dist
        d[cities[i][0]] = city
    return d
```

```
In: cities_d = distance_dict(store_val, len(store_val))
```

Функция `F` для подсчета общего расстояния путешествия:

```
In: def F(path, cities):
    dist = 0
    for i in range(len(path) - 1):
        dist += cities[path[i]][path[i + 1]]
    dist += cities[path[-1]][path[0]]
    return dist
```

За функцию `G` будет выступать простая перестановка как и в задаче о  $N$  ферзях.

```
In: def G(path, n):
    pos = path.copy()
    while True:
        i = np.random.randint(0, n - 1)
        j = np.random.randint(0, n - 1)
        if i != j:
            break
        pos[i], pos[j] = pos[j], pos[i]
    return pos
```

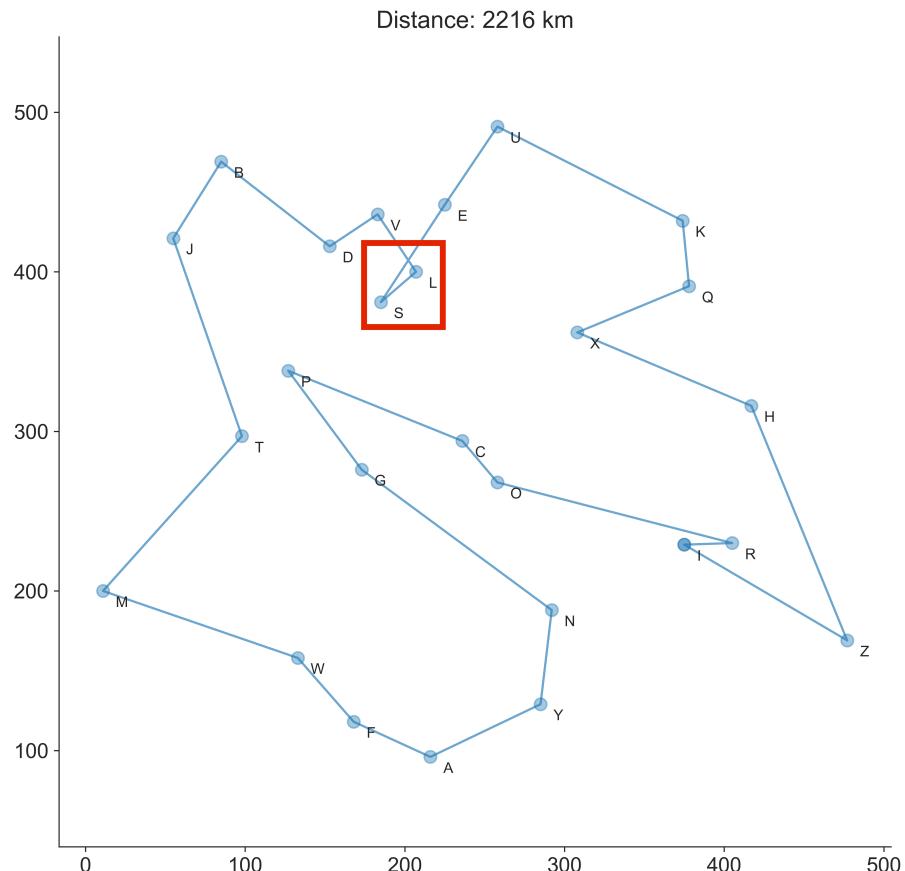


Рисунок 2.8 — Применение метода отжига для построения оптимального маршрута для 26-ти городов.

Для понижения температуры используем Больцмановский отжиг (2.1).

```
In: def SA(path, T):
    path_hat = path
    n = len(path_hat)
    np.random.shuffle(path_hat)
    T_0 = T
    k = 1
    for i in range(100000):
        path_tilda = G(path_hat, n)
        delta = F(path_tilda, cities_d) - F(path_hat, cities_d)
```

```
prob = np.exp(- delta / T)
```

Теперь построим оптимальный маршрут (рис. 2.8).

```
In: path_opt = SA(names, 100)
```

Несмотря на то, что метод отжига сократил преодолеваемую дистанцию, TSP была решена неидеально. К примеру, можно выделить соединение вершин E-S-L-V: очевидно, оно не оптимально, поскольку путь E-L-S-V имеет меньшее расстояние. Тем не менее, само расположение городов весьма удовлетворительно.

## 2.1.5. Вывод

Рассмотрев метод отжига, выявим его плюсы и минусы.

### Преимущества

1. Оптимизационный процесс поставленной задачи не застревает в локальных минимумах.
2. Имитация отжига имеет достаточно простую реализацию.
3. Даже для сложных задач наподобие TSP алгоритм, как правило, дает вполне приемлемое решение.

### Недостатки

1. Алгоритм достаточно трудоемок.
2. Не подходит для задач с небольшим количеством локальных минимумов. В таком случае градиентный спуск с мультистартом будет работать намного лучше.
3. Метод отжига не всегда сходится к решению.

## 2.2. Метод роения частиц

*Метод роения частиц* (*Particle Swarm Optimization*, PSO) является одним из алгоритмов коллективной оптимизации и основывается на имитации социального поведения в колонии живых организмов — к примеру, стаи птиц или колонии муравьев, — выполняющих коллективный поиск места с наилучшими условиями для существования. При поиске пищи каждая особь колонии передвигается по окружающей среде независимо от остальных организмов с некой долей случайности в своих движениях. Рано или поздно одна из особей находит пропитание и, будучи социальным организмом, сообщает об этом остальным, что стягивает ее соседей к данной пище.

### 2.2.1. Алгоритм

Найдем глобальный экстремум функции  $F: \mathbb{R}^n \rightarrow \mathbb{R}$ . Для определенности будем искать глобальный минимум:

$$F(x_i) \rightarrow \min_{x_i \in \mathbb{R}^n}$$

Пусть в нашем рое существует  $\ell$  частиц, тогда рой имеет вид  $x = \{x_i\}_{i=1}^\ell$ ,  $x_i \in U \subseteq \mathbb{R}^n$ . Пусть также определен вектор скорости  $v = \{v_i\}_{i=1}^\ell$ ,  $i$ -я компонента которого является скоростью  $i$ -й частицы,  $v_i \in U \subseteq \mathbb{R}^n$ .

1. Изначально случайным образом выбираем расположение роя  $x(0)$  и скорость движения каждой частицы  $v(0)$ .
2. Определяем новое расположение роя:

$$x(t+1) \equiv x(t) + v(t)$$

3. Выбираем наилучшую точку для  $i$ -й частицы:

$$p_i(t) = \begin{cases} x_i(t), & F(x_i(t+1)) \geq F(x_i(t)), \\ x_i(t+1), & F(x_i(t+1)) < F(x_i(t)) \end{cases}, \quad 1, \dots, \ell$$

Тогда вектор наилучших позиций для каждой частицы имеет вид:

$$p(t) = \{p_i(t)\}_{i=1}^\ell$$

4. Выбираем наилучшую точку для всего сообщества:

$$g(t) \equiv \operatorname{argmin}_{i \in (1, \dots, \ell)} F(p_i(t))$$

5. Обновляем скорость для  $i$ -й частицы посредством следующей формулы:

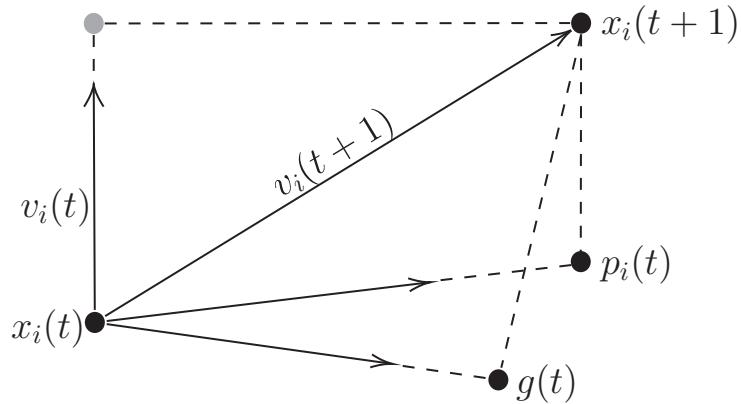
$$v_i(t+1) \equiv w v_i(t) + c_1 \xi_1(t)(p_i(t) - x_i(t)) + c_2 \xi_2(t)(g(t) - x_i(t)), \quad i = 1, \dots, \ell$$

где  $w \in \mathbb{R}$  — инерционный вес,  $c_1, c_2 \in \mathbb{R}$  — коэффициенты ускорения,  $\xi_1, \xi_2 \sim U(0, 1)$ .

Тогда вектор скорости имеет вид:

$$v(t+1) = \{v_i(t+1)\}_{i=1}^{\ell}$$

Три вектора влияют на частицу в положении  $x_i(t)$ . С одной стороны, когнитивное воздействие побуждает частицу двигаться к ее лучшей позиции  $p_i(t)$ , с другой стороны — социальное воздействие побуждает частицу продвигаться в сторону лучшей позиции роя  $g(t)$ . Кроме того, собственная скорость  $v_i(t)$  обеспечивает движение по инерции, что позволяет частице преодолевать локальные минимумы и исследовать неизвестные области заданного пространства. Таким образом, происходит переход от точки  $x_i(t)$  в точку  $x_i(t+1)$ , что представлено на следующем графике:



Алгоритм использует две последовательности равномерно распределенных случайных величин  $\xi_1(0), \dots, \xi_1(t)$  и  $\xi_2(0), \dots, \xi_2(t)$ , которые масштабируются по константам  $c_1, c_2$ . Данные константы влияют на максимальный размер шага, который частица может сделать за одну итерацию. При  $c_1 = 0$  метод роения частиц будет опираться только на наилучшую позицию сообщества — в таком случае алгоритм будет быстро сходиться, однако маловероятен факт нахождения глобального оптимума. При  $c_1 > 0$  метод использует связь всего сообщества — скорость конвергенции падает, но глобальный оптимум оказывается более вероятным.

6. Возвращаемся к 2 шагу, пока не достигнем оптимума.

## 2.2.2. Функция Розенброка

Функция Розенброка — это невыпуклая функция вида

$$f(x, y) = (a - x)^2 + b(y - x^2)^2,$$

использующаяся в качестве оценки производительности оптимизационных алгоритмов. Она имеет глобальный минимум в точке  $(a, a^2)$ , где  $f(a, a^2) = 0$ .

Обычно  $a = 1$ ,  $b = 100$ . Тогда функция Розенброка примет следующий вид (рис. 2.9 и рис. 2.10):

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

Глобальный минимум данной функции находится внутри «долины»: в нашем случае — в точке  $(1, 1)$ . Найти долину достаточно легко, однако приблизиться к глобальному минимуму, считается, довольно сложно.

Только при  $a = 0$ , функция является симметричной, а ее минимум находится в начале координат.

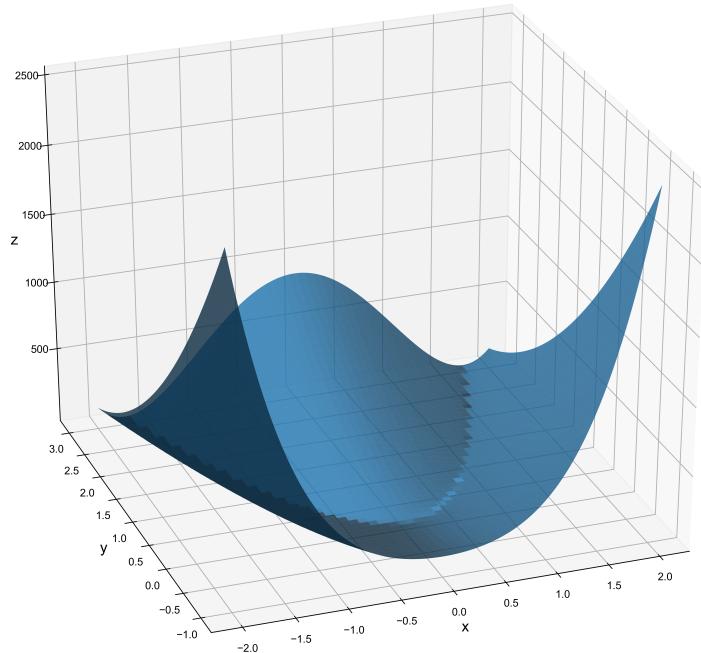


Рисунок 2.9 — Функция Розенброка с параметрами  $a = 1$ ,  $b = 100$ .

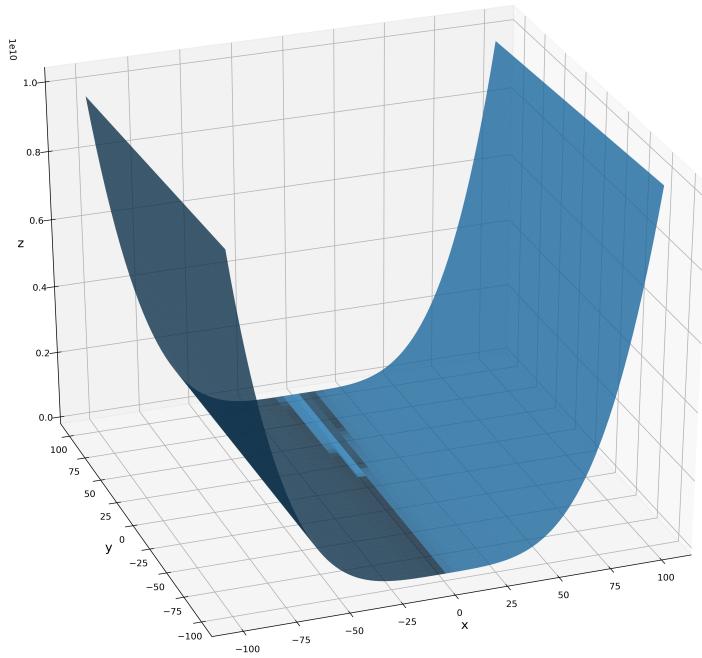


Рисунок 2.10 — Функция Розенброка с параметрами  $a = 1$ ,  $b = 100$ .

Определим функцию Розенброка:

```
In: def func(x):
    return (1 - x[0]) ** 2 + 100 * (x[1] - x[0] ** 2) ** 2
```

Инициализируем метод роения частиц, используя объектно-ориентированное программирование. В первую очередь создадим класс частиц.

```
In: class Particle:
    def __init__(self, arg, space):
        self.pos = np.asarray([])          # расположение частицы
        self.velocity = np.asarray([])      # вектор скорости частицы
        self.pos_best = None               # лучшее расположение
        for i in range(arg):
            pos_i = np.random.uniform(space[i][0], space[i][1])
            self.pos = np.append(self.pos, pos_i)
            vel_i = np.random.uniform(0.2*space[i][0], 0.2*space[i][1])
            self.velocity = np.append(self.velocity, vel_i)
        # pos_best --- это список, состоящий из лучшего расположения
        # частицы и значения функции в данной точке
        self.pos_best = [self.pos.copy(), func(self.pos)]

    def update_position(self):
        self.pos += self.velocity

    def update_velocity(self, w, c1, c2, swarm_best):
```

```

inertion = w * self.velocity
xi_1 = np.random.uniform()
xi_2 = np.random.uniform()
cognitive_acceler = c1 * xi_1 * (self.pos_best[0] - self.pos)
social_acceler = c2 * xi_2 * (swarm_best - self.pos)
self.velocity = inertion + cognitive_acceler + social_acceler

def choose_personal_best(self):
    if func(self.pos) < func(self.pos_best[0]):
        self.pos_best[0] = self.pos.copy()
        self.pos_best[1] = func(self.pos)

```

Теперь создадим класс ParticleSwarmOptimisation. Поскольку инерционный вес  $w$  должен быть близким к 1, а коэффициенты ускорения  $c_1, c_2$  должны быть достаточно малыми, при инициализации класса установим гиперпараметры по умолчанию равными следующим величинам:  $w = 1.0$ ,  $c1 = 0.2$ ,  $c2 = 0.2$ . Посредством метода search\_global будем искать глобальный минимум функции Розенброка.

```

In: class ParticleSwarmOptimisation:
    def __init__(self, ell=40, w=1.0, c1=0.2, c2=0.2, max_iter=1000,
                 tol=1e-6):
        """
        PARAMETERS:
        ell --- количество частиц в рое.
        w --- инерционный вес.
        c1 --- коэффициент ускорения когнитивного воздействия на частицу.
        c2 --- коэффициент ускорения социального воздействия на частицу.
        max_iter --- максимальное количество итераций.
        tol --- точность.
        """

        self.ell = ell
        self.w = w
        self.c1 = c1
        self.c2 = c2
        self.max_iter = max_iter
        self.tol = tol
        self.swarm_best = None # лучшее расположение для всего роя
        self.swarm = None       # расположение всех частиц (рой)

```

```

def search_global(self, arg, space):
    """
    PARAMETERS:
    arg --- количество аргументов функции.
    space --- область поиска оптимума. Задается как список из
    кортежей, где кортеж --- это область значений
    одного аргумента функции.
    """

    self.arg = arg
    self.space = np.array(space)
    self.swarm = np.asarray([])

    # генерируем расположение роя
    for _ in range(self.ell):
        self.swarm = np.append(self.swarm,
                              Particle(self.arg, self.space))

    for k in range(self.max_iter):
        for i in range(self.ell):
            # обновляем расположение частицы
            self.swarm[i].update_position()
            # сравниваем с лучшей точкой частицы
            self.swarm[i].choose_personal_best()

        # выбираем лучшую точку для роя
        if k != 0:
            dist_0 = self.dist(self.swarm_best[0])
            self.choose_social_best()
            dist_1 = self.dist(self.swarm_best[0])

            # останавливаем поиск в условиях заданной точности
            if (dist_0 != dist_1) and (abs(dist_0-dist_1) <= self.tol):
                break
        else:
            self.choose_social_best()

        # обновляем вектор скорости
        for i in range(self.ell):
            self.swarm[i].update_velocity(self.w, self.c1,
                                         self.c2, self.swarm_best[0])

    print(f"Глобальный оптимум: {self.swarm_best[0]}")
    print(f"Значение функции в данной точке: {self.swarm_best[1]}")

```

```

def choose_social_best(self):
    best = min([[self.swarm[i].pos_best[0],
                 self.swarm[i].pos_best[1]] for i in range(self.ell)],
               key=lambda x: x[1])
    self.swarm_best = best

def dist(self, x):
    return np.sqrt(np.sum(x ** 2))

```

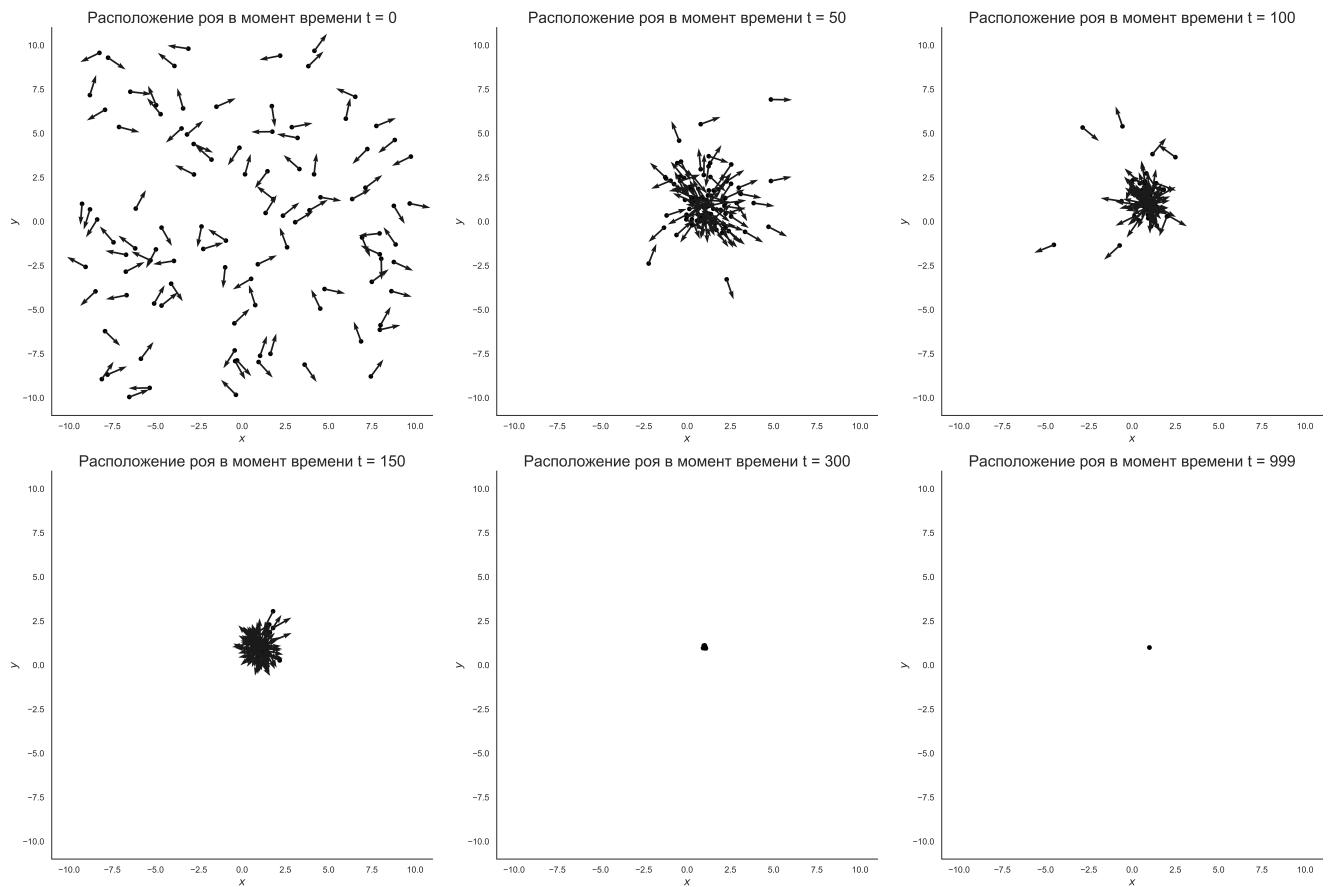


Рисунок 2.11

Инициализируем рой с 100 частицами.

In: `sw = ParticleSwarmOptimisation(w=0.95, ell=100, tol=1e-20)`

Отыщем глобальный минимум. Перемещение роя в поисках глобального оптимума представлено на рисунке 2.11.

In: `np.random.seed(53)`  
`sw.search_global(2, [(-10, 10), (-10, 10)])`

Out: Глобальный оптимум: [1. 1.] .

Значение функции в данной точке: 6.366439023928984e-22.

Из графиков на рисунке 2.12 видно, что за 1000 итераций метод роения частиц сходится к глобальному минимуму функции Розенборока с экспоненциальной скоростью с точностью 6.37e-22. Средняя скорость реализации алгоритма составляет 1.79 секунды со стандартным отклонением в 0.111 секунд.

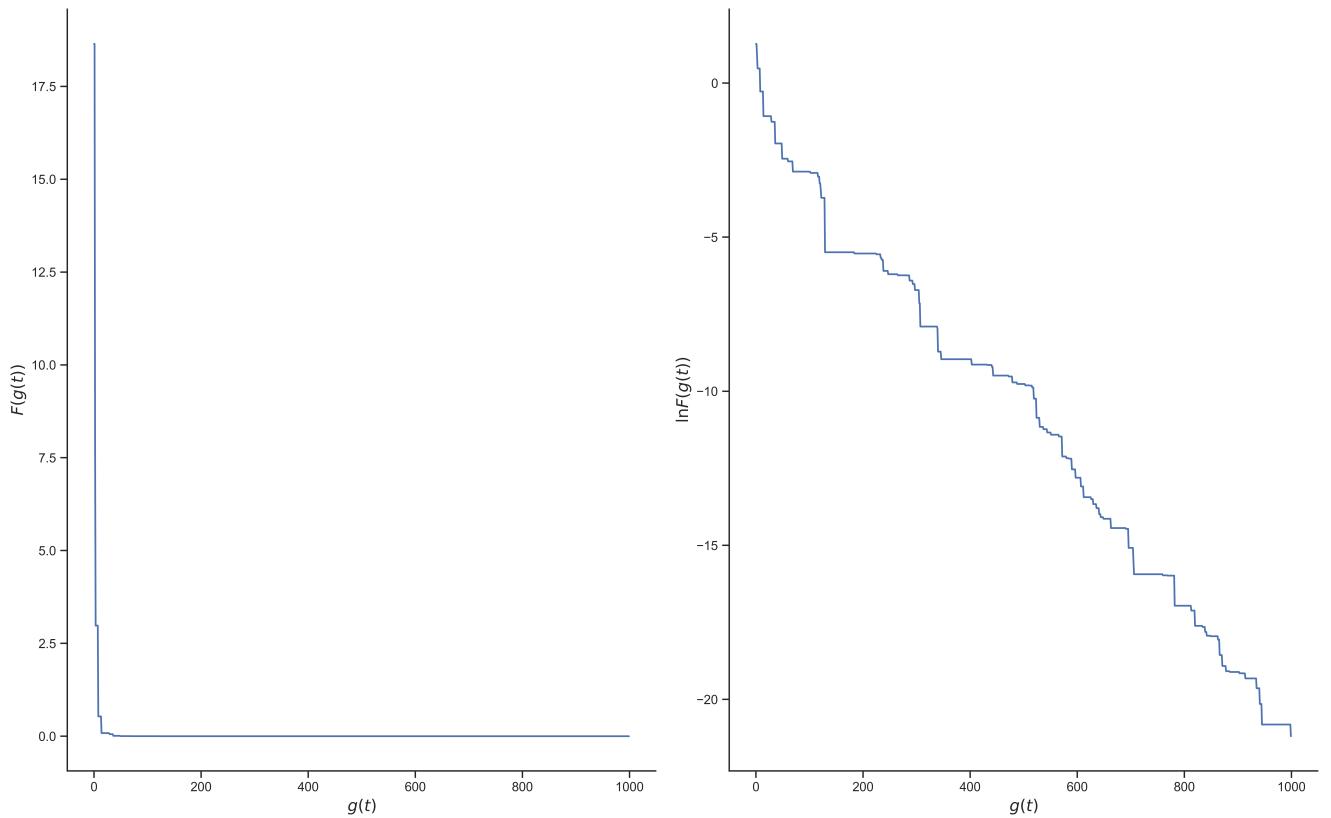


Рисунок 2.12 — Оптимизационный процесс.

# **Заключение**

Основные результаты работы заключаются в следующем.

## Список литературы

- [1] *Лопатин А. С.* (2005) Стохастическая оптимизация в информатике. Метод отжига. // Сайт Math.spbu.ru. URL: <https://www.math.spbu.ru/user/gran/optstoch.htm> (дата обращения: 20.01.2020)
- [2] *Шамин Р. В.* (2019) Практическое руководство по машинному обучению. // Москва: Научный канал.
- [3] *Weise T.* (2009) Global Optimization Algorithms — Theory and Application. // Self-Published.
- [4] *Pedersen M. E. H.* (2010) Tuning and Simplifying Heuristical Optimization. Thesis for the degree of Doctor of Philosophy. // University of Southampton.
- [5] *Nedjah N. and Mourelle L.* (2006) Swarm Intelligent Systems. // Berlin, Heidelberg: Springer Berlin Heidelberg.