**ОТЧЕТ**

**по оптимизационному алгоритму нахождения локального оптимума «Алгоритм имитации отжига»**

# Содержание

Содержание 2

Введение 2

1 Описание функций 3

2 Сценарии 7

3 Рекомендации к начальным данным 11

Список литературы 19

Листинг программы 20

# Введение

В современном мире решение сложных оптимизационных задач играет ключевую роль во многих областях, включая инженерию, науку, экономику и информационные технологии. Для эффективного решения таких задач разрабатываются и применяются разнообразные алгоритмы, основанные на различных принципах и методологиях. Однако существует класс задач, где традиционные методы оптимизации достигают своих пределов из-за высокой степени сложности и нелинейности функций.

В этом контексте, алгоритм имитации отжига (Simulated Annealing) представляет собой мощный и инновационный метод оптимизации, который находит применение в разнообразных дисциплинах. Вдохновленный физическим процессом отжига металлов, данный алгоритм эмулирует статистический механизм перемещения по пространству состояний с целью нахождения глобального оптимума функции.

# 1 Описание функций

Функция **create\_grid** – на вход принимает два списка numpy array: bounds, где в качестве строк (строка из двух значений для каждого аргумента) задаются интервалы изменения аргументов функции многих переменных, size, где в виде строки значений задаются размеры сеток для каждого аргумента функции многих переменных. Выводит функция список numpy array, в каждой строке которого находится сетка разбиения аргумента функции многих переменных (в списке, каждая строка отдельный список).

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Рисунок 1.1 – пример работы **create\_grid.**

Функция **grid\_of\_funceval** предназначена для отладки результатов функции двух переменных, с помощью ее создается сетка с посчитанными значениями функции в каждом узле сетки созданной с помощью функции create\_grid. На вход принимает список numpy array, который выводит функция create\_grid и целевую функцию, которая определяется в виде функции с помощью def. Выводит список numpy array, с посчитанными комбинациями аргументов. Функция предназначена для отладки только функции двух переменных!

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Рисунок 1.2 – пример работы функции **grid\_of\_funceval.**

Функции **min\_of\_function** и **find\_in\_grid** также являются функциями отладки, на вход обе функции принимают grid – результат работы функции grid\_of\_funceval, второй аргумент solve – результат работы функции simulated\_anelling, представляющий из себя значения локального оптимума.

Функция **start\_solve** предназначена для нахождения начального решения для функции simulated\_anelling, на вход принимает value\_grid – результат работы функции create\_grid, выводит значения аргументов начального решения, а также индексы этих значений в сетке value\_grid.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Рисунок 1.3 – пример работы функции **start\_solve.**

Функция **new\_point\_boltzman** предназначена для функции simulated\_anelling, для нахождения новых решений. На входи принимает bounds – интервалы изменения аргументов аналогично create\_grid, value\_grid – вывод функции create\_grid, temperature – текущая температура процесса, index – индекс текущего решения в сетке value\_grid. Выводит функция решения и его индекс в сетке аналогично start\_solve.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Рисунок 1.4 – пример работы функции **new\_point\_boltzman.**

Функция **find\_start\_temp** предназначена для нахождения начальной температуры, на вход функция получает bounds – интервалы изменения аргументов и func – целевая функция, выводит функция температуру.

Функция **simulated\_anelling** основная функция метода отжига на вход получает func – целевая функция, bounds – интервалы изменения аргументов, value\_grid -–сетка получаемая из функции create\_grid, size\_of\_grid – размер сетки для каждого интервала, iter\_count – количество итераций в цикле для метода отжига, start\_temp – начальная темпратура задающаяся в ручную, flag – флажок для температуру, если мы хотим чтобы оптимальная температура находилась с помощью функции find\_start\_temp ставим 1, вручную – 0, вывод solve1 – значения аргументов в точке минимума, energy1 – значения целевой функции в точке минимума, index1 – индекс в сетке create\_grid, count – общее количество вычислений (целевой функции), count1 – количество сохраненных вычислений (целевой функции).

# 2 Сценарии

Функция **try\_function\_anydemension\_butwithout\_exact\_solution** предназначена для нахождения решения целевой функции с любым количеством аргументов. На вход получает it – количество итераций в алгоритме отжига, temp – температура отжига, func – целевая функция, size\_of\_grid – размер сетки для каждого интервала, bounds – интервалы изменения аргументов. В этой функции нет сравнения с точным решением для данной целевой функции.

Изображение выглядит как текст, снимок экрана, Шрифт, дисплей

Автоматически созданное описание

Рисунок 2.1 – пример использования функции **try\_function\_anydemension\_butwithout\_exact\_solution.**

Функция **try\_function\_anydemension\_with\_exact\_solution** предназначена для нахождения решения целевой функции с любым количеством аргументов. На вход получает it – количество итераций в алгоритме отжига, temp – температура отжига, func – целевая функция, size\_of\_grid – размер сетки для каждого интервала, bounds – интервалы изменения аргументов. В этой функции производится сравнение с численным решением, которое получается из метода dual\_anelling библиотеки scipy.optimization.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Рисунок 2.2 – пример использования функции **try\_function\_anydemension\_with\_exact\_solution.**

Функция **try\_function\_2d\_demension** предназначена для нахождения решения целевой функции с двумя аргументами. На вход получает it – количество итераций в алгоритме отжига, temp – температура отжига, func – целевая функция, size\_of\_grid – размер сетки для каждого интервала, bounds – интервалы изменения аргументов. В этой функции производится сравнение с точным решением на указанной сетке, которое находится с помощью функций отладки.

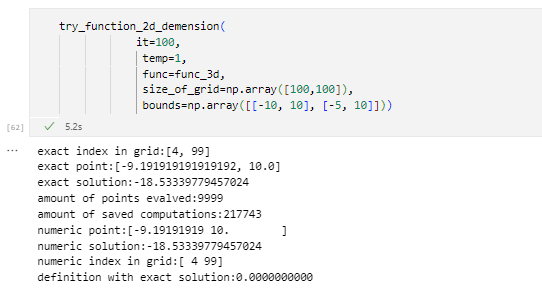


Рисунок 2.3 – пример использования функции **try\_function\_2d\_demension.**

Функция **try\_function\_2d\_demension\_withnumericmethod** предназначена для нахождения решения целевой функции с двумя аргументами. На вход получает it – количество итераций в алгоритме отжига, temp – температура отжига, func – целевая функция, size\_of\_grid – размер сетки для каждого интервала, bounds – интервалы изменения аргументов. В этой функции производится сравнение с численным решением, которое получается из метода dual\_anelling библиотеки scipy.optimization.

Изображение выглядит как текст, электроника, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 2.4 – пример использования функции **try\_function\_2d\_demension\_withnumericmethod.**

Функция **try\_function\_2d\_demension\_100tests** предназначена для нахождения решения целевой функции с двумя аргументами в 100 тестах. На вход получает it – количество итераций в алгоритме отжига, temp – температура отжига, func – целевая функция, size\_of\_grid – размер сетки для каждого интервала, bounds – интервалы изменения аргументов, flag – индикатор указывающий, какую температуру использовать 0 – с клавиатуры, 1 – автоматически найденную. В этой функции производится сравнение с точным решением на указанной сетке, которое находится с помощью функций отладки, а также выводится количество точных совпадений.

Изображение выглядит как текст, снимок экрана, Шрифт, документ

Автоматически созданное описание

Рисунок 2.5 – пример использования функции **try\_function\_2d\_demension\_100tests.**

# 3 Рекомендации к начальным данным

Две величины регулируют точность алгоритма температура и количество итераций, проведем вычисления для нахождения оптимальных значений для температуры и итераций.

Изображение выглядит как текст, снимок экрана, Шрифт, документ

Автоматически созданное описание

Рисунок 3.1 – результаты при температурах 1,10 и 100 и количестве итераций 100.

Чтобы оценить эффективность алгоритма нужно смотреть на строку definition with exact solution и amount of points evalved, первое говорит о точности решение, второе показывает сколько раз целевая функция была посчитана, за вторым значением важно следить, так как алгоритм, как минимум должен быть эффективней простого пересчета всех значений в сетке. Как мы видим при температуре 1 количество вычислений составило 9997, возникает очевидный вопрос, количество итераций равно 100, почему так выходит? Алгоритм является стохастическим и основан на принципе минимизации энергии (значений целевой функции), если энергия не уменьшается то алгоритм в цикле while стохастически пытается найти новое решение с определенной вероятностью зависящей от температуры, поэтому можно сделать вывод, что основным параметром в алгоритме является температура.

Изображение выглядит как текст, снимок экрана, Шрифт, документ

Автоматически созданное описание

Рисунок 3.2 – результаты при температуре 1 и количестве итераций 100, 1000 и 10000.

Мы видим, что в данном случае просто происходит пересчет всех значений и изменений увеличение количества итераций не привносит.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Рисунок 3.3 – результаты при температуре 5 и количестве итераций 100.

При температуре 5 результат улучшился, посмотрим с помощью функции try\_function\_2d\_demension\_100tests.

Изображение выглядит как текст, снимок экрана, документ, Шрифт

Автоматически созданное описание

Рисунок 3.4 – результаты при температуре 5 и количестве итераций 100 при 100 тестах.

47 точных совпадений из 100 тестов, но сетка в этих запусках была маленькая 100 x 100, попробуем решить задачу на сетке 1000 x 1000.

Изображение выглядит как текст, снимок экрана, дисплей, программное обеспечение

Автоматически созданное описание

Рисунок 3.5 – результаты при температуре 1 и количестве итераций 100 на сетке 1000 x 1000.

В этот раз при температуре 1, есть существенный выигрыш в вычислениях целевой функции всего 590436 вычислений при 1e6 значений целевой функции, можно сделать вывод, что при увеличении размера сетки температуру следует понижать, важно также понимать, что алгоритм стохастический и, что эти выводы сделаны для определённой функции.

Изображение выглядит как текст, снимок экрана, Шрифт, документ

Автоматически созданное описание

Рисунок 3.6 – результаты при температуре 1 и количестве итераций 100 на сетке 1000 x 1000.

Изображение выглядит как текст, снимок экрана, Шрифт, документ

Автоматически созданное описание

Рисунок 3.7 – результаты при температуре 1, 0.5, 5 и количестве итераций 100 на сетке 1000 x 1000.

Изображение выглядит как текст, Шрифт, линия, снимок экрана

Автоматически созданное описание

Рисунок 3.8 – пример задания целевой функции

# Список литературы

[1] Algorithm Organic Rankine Cycle (ORC) Power Systems Simulated Annealing, 2017

[2] Лопатин А. С. Метод отжига // СПБГУ 2005

# Листинг программы

import numpy as np

import matplotlib.pyplot as plt

from scipy.optimize import dual\_annealing

import sys

def create\_grid(bounds: np.ndarray = np.array([[1, 10], [1, 10]]),

                 size: np.ndarray = np.array([10,15])) -> np.ndarray:

    """ создание прямоугольной неравномерной сетки """

    # a = np.zeros((np.shape(bounds)[0],1),dtype=np.ndarray)

    a = np.array(bounds[:,0],dtype=np.ndarray)

    for i in range(np.shape(bounds)[0]):

        a[i] = np.linspace(bounds[i, 0], bounds[i, 1], size[i])

    return a

def grid\_of\_funceval(value\_grid: np.ndarray,

                      func) -> np.ndarray:

    """ только для пространсва D=2, для любой сетки в том числе и неравномерной"""

    idk = np.max(np.array([np.shape(value\_grid[0])[0],np.shape(value\_grid[1])[0]]))

    grid = np.full((idk,idk), np.inf)

    for i in range(np.shape(value\_grid[0])[0]):

        for k in range(np.shape(value\_grid[1])[0]):

            # grid[i,k] = func(value\_grid[np.arange(0, np.shape(value\_grid)[0]), qq])

            grid[i,k] = func(np.array([(value\_grid[0])[i],(value\_grid[1])[k]]))

    return grid

def min\_of\_function(grid: np.ndarray) -> np.ndarray:

    return [np.where(grid == np.min(grid))[0][0],\

            np.where(grid == np.min(grid))[1][0]],np.min(grid)

def find\_in\_grid(value\_grid: np.ndarray,

                  solve: np.ndarray) -> np.ndarray:

    a = np.copy(solve)

    qq = np.random.randint(0, 1, np.shape(value\_grid)[0])

    for i in range(np.shape(value\_grid)[0]):

        hqd = np.absolute(solve[i] - (value\_grid[i])[:])

        qq[i] = np.where(hqd == np.min(hqd))[0][0]

    return qq

def start\_solve(value\_grid: np.ndarray) -> np.ndarray:

    a = np.array([])

    index1 = np.array([],dtype=int)

    for i in range(np.shape(value\_grid)[0]):

        index = np.random.randint(np.shape(value\_grid[i])[0])

        a = np.append(a,(value\_grid[i])[index])

        index1 = np.append(index1,index)

    return a, index1

def new\_point\_boltzman(bounds: np.ndarray,

                        value\_grid: np.ndarray,

                        temperature: float,

                        index: np.ndarray) -> np.ndarray:

    std = np.sqrt(temperature) \* np.ones(np.shape(bounds)[0])

    xc = np.random.normal(0, 1.0, size =  np.shape(bounds)[0])

    new\_solve\_index = index + xc \* std

    new\_solve = np.copy(new\_solve\_index)

    for i in range(np.shape(bounds)[0]):

        new\_solve\_index[i] = round(index[i] + xc[i] \* std[i] \* (np.shape((value\_grid[i]))[0] - 1))

        while  new\_solve\_index[i] < 0 or  new\_solve\_index[i] > (np.shape((value\_grid[i]))[0] - 1):

            xc[i] = np.random.normal(0, 1.0)

            new\_solve\_index[i] = round(index[i] + xc[i] \* std[i] \* (np.shape((value\_grid[i]))[0] - 1))

    for i in range(np.shape(bounds)[0]):

        new\_solve[i] = (value\_grid[i])[new\_solve\_index.astype(int)[i]]

    return new\_solve, new\_solve\_index.astype(int)

def find\_start\_temp(bounds: np.ndarray,

                    func) -> np.ndarray:

    low = bounds[:,0]

    high = bounds[:,1]

    fmax = np.finfo(float).min

    fmin = np.finfo(float).max

    for \_ in range(50):

        start\_solve = np.random.uniform(size = np.shape(bounds)[0]) \* (high - low) + low

        start\_solve\_eval = func(start\_solve)

        if start\_solve\_eval > fmax:

            fmax = start\_solve\_eval

        if start\_solve\_eval < fmin:

            fmin = start\_solve\_eval

            best\_solve = start\_solve

        temp = (fmax - fmin) \* 1.5

    return temp

def simulated\_anelling(func,

                       bounds:np.ndarray,

                       value\_grid: np.ndarray,

                       size\_of\_grid: np.ndarray,

                       iter\_count:int = 100,

                       start\_temp:float = 70,

                       flag:float = 0):

    if flag == 1:

        start\_temp = find\_start\_temp(bounds,func)

        print('start temperature',start\_temp)

    grid\_of\_values = np.full(shape = size\_of\_grid, fill\_value = np.NAN)

    solve1, index1 = start\_solve(value\_grid)#global minimum

    energy1 = func(solve1)#value of global minimum

    grid\_of\_values[tuple(index1)] = energy1

    t = start\_temp

    count1 = 0

    count = 1

    solve2, energy2, index2 = solve1, energy1, index1 #current\_solve\_of\_alghorithm

    for i in range(iter\_count):

        solve3, index3 = new\_point\_boltzman(bounds, value\_grid, t,  index2)#new\_solve\_of\_alghorithm

        if np.isnan(grid\_of\_values[tuple(index3)]):

            count +=1

            energy3 = func(solve3)

            grid\_of\_values[tuple(index3)] = energy3

        else:

            count1 +=1

            energy3 = grid\_of\_values[tuple(index3)]

        diff =  energy3 - energy2

        metropolis = np.exp(-diff / t)

        if energy3 < energy1:#difference w global\_min and new\_solve\_of\_alghorithm

            solve1, energy1, index1 = solve3, energy3, index3

            solve2, energy2, index2 = solve3, energy3, index3

        else:

            while(1):

                if energy3 < energy1:

                    solve1, energy1, index1 = solve3, energy3, index3

                    solve2, energy2, index2 = solve3, energy3, index3

                    break

                if np.random.uniform(0.0, 1.0) < metropolis:

                    solve2, energy2, index2 = solve3, energy3, index3

                    # print('loool')

                    break

                solve3, index3 = new\_point\_boltzman(bounds, value\_grid, t,  index2)

                if np.isnan(grid\_of\_values[tuple(index3)]):

                    count +=1

                    energy3 = func(solve3)

                    grid\_of\_values[tuple(index3)] = energy3

                else:

                    count1 +=1

                    energy3 = grid\_of\_values[tuple(index3)]

                energy3 = func(solve3)

                diff =  energy3 - energy2

                metropolis = np.exp(-diff / t)

        t = start\_temp / np.log((i + 2) + 1)

    return [solve1, energy1, index1, count, count1]

def func2(x: np.ndarray):

    return -np.sin(0.5 \* (x[0]\*\*2) + - 0.25 \* (x[1]\*\*2) + 3) \* np.cos(2\*x[2] + 1 - np.exp(x[3]))

def func\_3d(x: np.ndarray):

    return x[1]\*np.sin(2\*np.pi\*x[0])+x[0]\*np.cos(2\*np.pi\*x[1])

def try\_function\_anydemension\_butwithout\_exact\_solution(

                it: int,

                temp: float,

                func,

                size\_of\_grid: float = np.array([1000,1000,1000,1000]),

                bounds: np.ndarray = np.array([[1, 10], [1, 10],[1, 10],[1, 10]])):

    zxc = create\_grid(bounds, size\_of\_grid)

    ok, ok1, ok2, count, count1 = simulated\_anelling(func, bounds, zxc, size\_of\_grid, it, temp)

    print(f'amount of points evalved:{count}')

    print(f'amount of saved computations:{count1}')

    print(f'numeric point:{ok}')

    print(f'numeric solution:{ok1}')

    print(f'numeric index in grid:{ok2}')

def try\_function\_anydemension\_with\_exact\_solution(

                it: int,

                temp: float,

                func,

                size\_of\_grid: float = np.array([1000,1000]),

                bounds: np.ndarray = np.array([[1, 10], [1, 10]])):

    zxc = create\_grid(bounds, size\_of\_grid)

    ret = dual\_annealing(func,bounds=bounds)

    print(f'point with other numeric method(dual\_anneling):{ret.x}')

    print(f'solution with other numeric method(dual\_anneling):{ret.fun}')

    ok, ok1, ok2, count, count1 = simulated\_anelling(func, bounds, zxc, size\_of\_grid, it, temp)

    print(f'amount of points evalved:{count}')

    print(f'amount of saved computations:{count1}')

    print(f'numeric point:{ok}')

    print(f'numeric solution:{ok1}')

    print(f'numeric index in grid:{ok2}')

    print(f'definition with exact solution:{(ok1 - ret.fun):.10f}')

def try\_function\_2d\_demension(

                it: int,

                temp: float,

                func,

                size\_of\_grid: float = np.array([1000,1000]),

                bounds: np.ndarray = np.array([[1, 10], [1, 10]])):

    zxc = create\_grid(bounds, size\_of\_grid)

    grid = grid\_of\_funceval(zxc, func)

    global point, point\_eval

    point, point\_eval = min\_of\_function(grid)

    print(f'exact index in grid:{point}')

    print(f'exact point:{[(zxc[0])[point[0]],(zxc[1])[point[1]]]}')

    print(f'exact solution:{point\_eval}')

    ok, ok1, ok2, count, count1 = simulated\_anelling(func, bounds, zxc, size\_of\_grid, it, temp)

    print(f'amount of points evalved:{count}')

    print(f'amount of saved computations:{count1}')

    print(f'numeric point:{ok}')

    print(f'numeric solution:{ok1}')

    print(f'numeric index in grid:{ok2}')

    print(f'definition with exact solution:{(ok1 - point\_eval):.10f}')

def try\_function\_2d\_demension\_withnumericmethod(

                it: int,

                temp: float,

                func,

                size\_of\_grid: float = np.array([1000,1000]),

                bounds: np.ndarray = np.array([[1, 10], [1, 10]])):

    zxc = create\_grid(bounds, size\_of\_grid)

    ret = dual\_annealing(func,bounds=bounds)

    print(f'point with other numeric method(dual\_anneling):{ret.x}')

    print(f'solution with other numeric method(dual\_anneling):{ret.fun}')

    ok, ok1, ok2, count, count1 = simulated\_anelling(func, bounds, zxc, size\_of\_grid, it, temp)

    print(f'amount of points evalved:{count}')

    print(f'amount of saved computations:{count1}')

    print(f'numeric point:{ok}')

    print(f'numeric solution:{ok1}')

    print(f'numeric index in grid:{ok2}')

    print(f'definition with exact solution:{(ok1 - ret.fun):.10f}')

def try\_function\_2d\_demension\_100tests(it: int,

                                       temp: float,

                                       func,

                                       size\_of\_grid: np.ndarray = np.array([200,100]),

                                       bounds: np.ndarray = np.array([[1, 10], [1, 10]]),

                                       flag:float = 0):

    zxc = create\_grid(bounds, size\_of\_grid)

    grid = grid\_of\_funceval(zxc, func)

    global point, point\_eval

    point, point\_eval = min\_of\_function(grid)

    print('exact index in grid', point)

    print('exact point',[(zxc[0])[point[0]],(zxc[1])[point[1]]])

    print('exact solution:', point\_eval)

    # print(np.min(zxc1))

    count = 0

    for i in range(100):

        ok,ok1,ok2,ok3,ok4 = simulated\_anelling(func, bounds, zxc, size\_of\_grid, it, temp)

        print(f"numeric index:{find\_in\_grid(zxc,ok)[0]}~{find\_in\_grid(zxc,ok)[1]} exact:{point[0]}~{point[1]}")

        if find\_in\_grid(zxc,ok)[0] - point[0] == 0 and find\_in\_grid(zxc,ok)[1] - point[1] == 0:

            count += 1

    print('count of exact solutions',count)