Теория оптимизации

Содержание:

Как правило, все методы оптимизации сводятся к задаче минимизации функции, то есть к нахождению глобального минимума или максимума.

Градиентный спуск – это итерационный метод. Решение задачи начинается с выбора начального приближения $\vec{x}^{[0]}$.

После вычисляется приблизительное значение $\vec{x}^{1}$ Затем $\vec{x}^{2}$ и так далее по следующему правилу:

$$\vec{x}^{[j+1]} = \vec{x}^{[j]} - \gamma^{[j]} \nabla F(\vec{x}^{[j]}), \qquad \text{где } \gamma^{[j]} \text{ — шаг градиентного спуска.}$$

а $\nabla F(\vec{x}^{[j]})$ - это градиент.

Идея: идти в направлении наискорейшего спуска, а это направление задаётся антиградиентом

$-\nabla F$

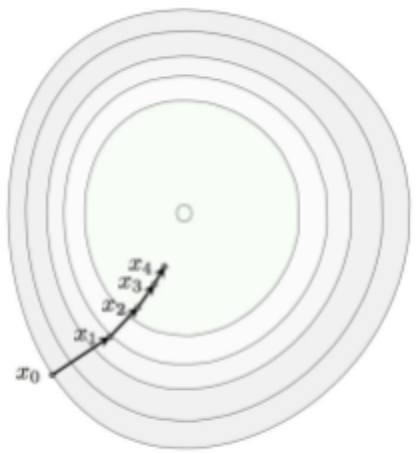
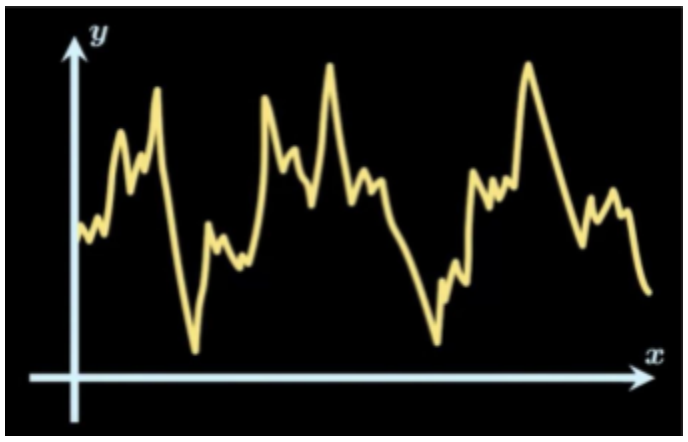


Рис. 3: Градиентный спуск

Этот метод хорош для гладких функций (производная которых – непрерывная функция). Часто же в жизни приходится минимизировать негладкие (зашумленные) функции, как например, мы видим на рисунке. Здесь много резких пиков, в которых производная не существует, то есть в этих пиках производная имеет разрыв, а значит функция f(x) не является гладкой.



И в таких случаях, даже если градиент у интересующей функции существует, оказывается, что вычислять его непрактично.

Пусть есть алгоритм с параметрами $\alpha_1, ..., \alpha_N$

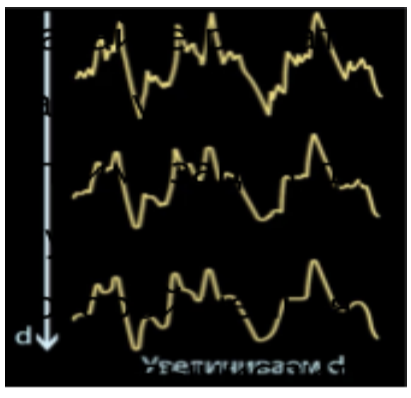
Задача: подобрать параметры так, чтобы алгоритм давал лучший результат. Можно записать в виде оптимизационной задачи

$$Q(\alpha_1, ..., \alpha_N) \to \max_{\alpha_1, ..., \alpha_N}$$

Вычисление градиента в этом случае зачастую невозможно в принципе или крайне непрактично.
Другая проблема градиентных методов — проблема локальных минимумов.

Градиентный спуск, попав на дно локального минимума, где градиент также равен нулю, там и остается. Глобальный минимум так и остается не найденным. Ведь градиентный спуск ведет только вниз. Попав в локальный минимум, спускаться уже некуда. А чтобы найти глобальный минимум, надо подняться из ямки, и уже потом снова спускаться, но уже в другую ямку.

Решить эти проблемы позволяют **методы случайного поиска**.
Общая идея этих методов заключается в намеренном введении элемента случайности.
К таким методам можно отнести **градиентный спуск без градиента.**
Будем решать задачу оптимизации для функции, у которой нельзя явно вычислить градиент.
Пример функции, у которой нельзя явно вычислить градиент



градиент.

Пусть дана некоторая такая функция $f(\vec{x})$ и требуется найти ее минимум.

Дано: $f(\vec{x}) \to \min_{\vec{x}}$

И пусть дан $d - \text{параметр метода}$, который мы далее обсудим.

Будем осуществлять градиентный спуск, который будет почти такой же, как обычный градиентный спуск, но не будем явно вычислять градиент.

Сначала выбирается $\vec{u}$ (случайный вектор $\vec{u}$ равномерно распределен по сфере)

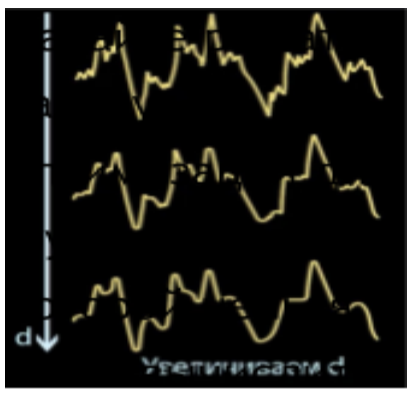
Затем вычисляем численную оценку производной по направлению $\frac{f(\vec{x}) - f(\vec{x} + d\vec{u})}{d}$.

(Здесь получаем именно оценку производной, а не саму производную, так как вычисляем отношение, а не предел отношения).

Сдвигаем точку в направлении $\vec{u}$ пропорционально величине с прошлого шага.

Еще раз следует отметить, что ни на каком шаге градиент функции не вычисляется. Вычисляется явное значение функции, а не градиент.

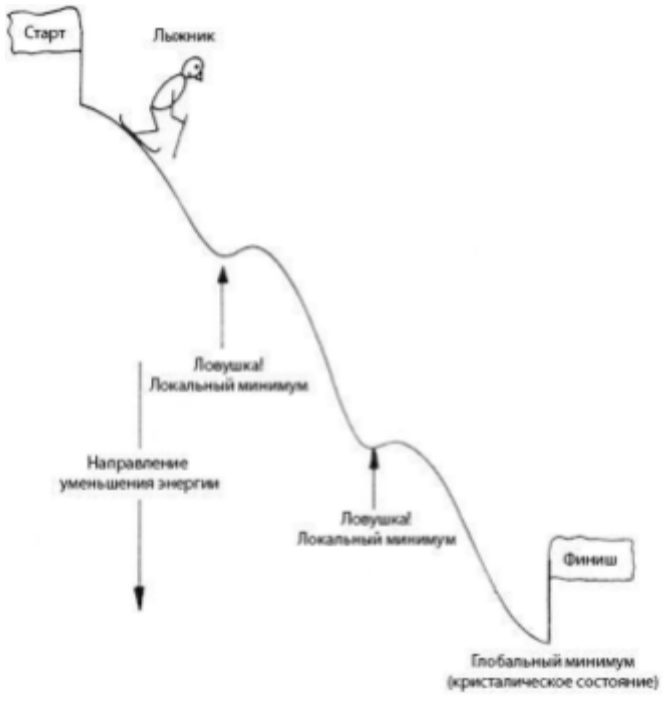
Из формулы видим, что величина смещения зависит от выражения функции в точке $\vec{x} + d\vec{u}$.
и в среднем смещение происходит смещение по антиградиенту сглаженной функции.
Число d - «параметр сглаживания» при нахождении численной оценки производной.
Рисунок. Суть метода на рисунке.



градиент.

Изначально функция имеет много острых пиков (и верхних и нижних), а при увеличении параметра d , эта функция в терминах задачи оптимизации будет выглядеть все более и более сглаженной. Тогда применение нашего метода будет эквивалентно применению градиентного спуска на сглаженной функции.

Еще раз вспомни, в чем заключается несостоятельность градиентных методов. Для этого посмотрим на лыжника. Спускаясь вниз, он может застрять в ямке локального минимума. И для того, чтобы достичь самой нижней точки, ему надо выпрыгнуть из ямки и начинать движение вниз уже из нового положения.



Для это можно применить еще один метод - **метод имитации отжига.** Алгоритм оптимизации, для которого не требуется гладкость функции.

Алгоритм основывается на имитации физического процесса, который происходит при кристаллизации вещества. Вспомним из физики, что при высокой температуре атомы вещества движутся быстрее, то есть тратят много энергии. А при остывании вещества и его кристаллизации атомы занимают такое положение в кристаллической решетке, чтобы энергия всей системы, то есть энергия взаимодействия атомов, была минимальной.

Цель отжига — привести систему, которой является образец металла, в состояние с минимальной энергией.

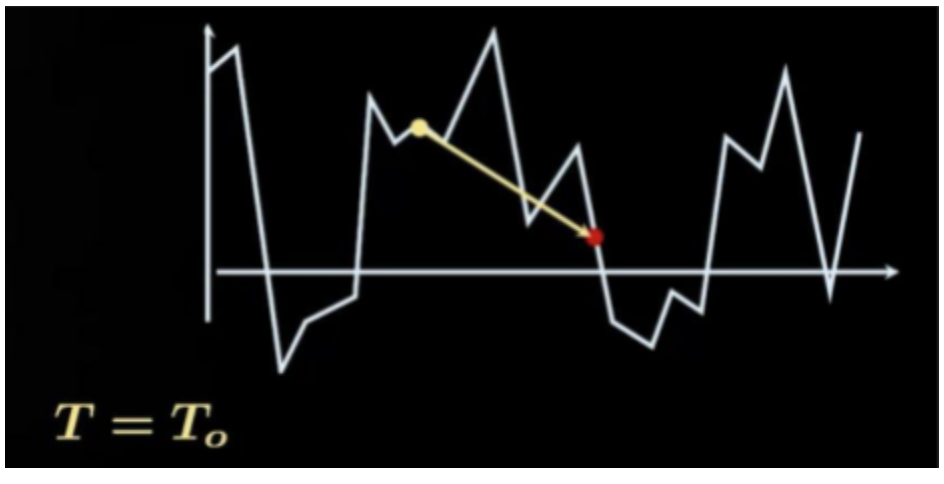
Для задач оптимизации имитация процесса может быть произведена следующим образом.

Вводится параметр T, который имеет смысл температуры, и в начальный момент ему устанавливается значение T0. Набор переменных, по которым происходит оптимизация, будет обозначаться как x.

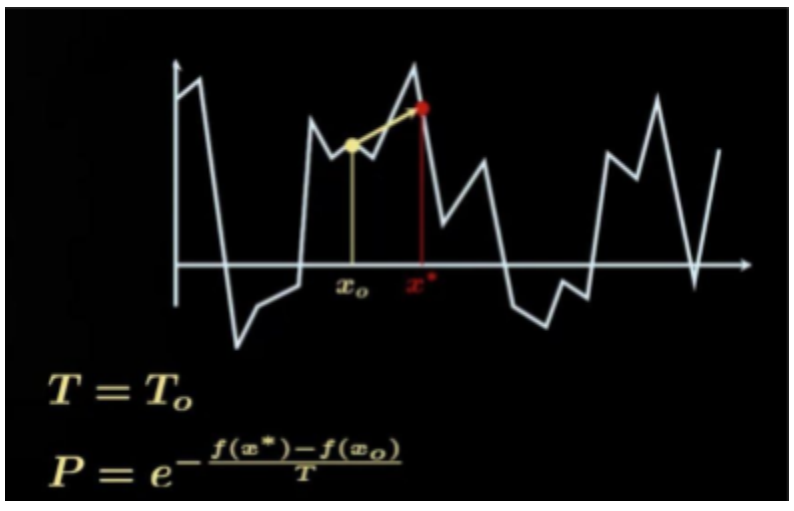
Метод имитации отжига

1. Начальное приближение - произвольная точка.
2. Выбираем x^* случайно из множества соседних состояний.
3. Если значение функции меньше, то переходим в x^*
4. Если значение функции больше x^*, то переходим в x^* с вероятностью $P = e^{-\frac{f(x^*)-f(x)}{T}}$
5. Уменьшаем значение температуры

Рассмотрим суть метода поподробнее. На рисунке начальное приближение – желтая точка, x^* - красная точка. На этом рисунке получили, что значение в красной точке меньше, то значит в эту точку и переходим.

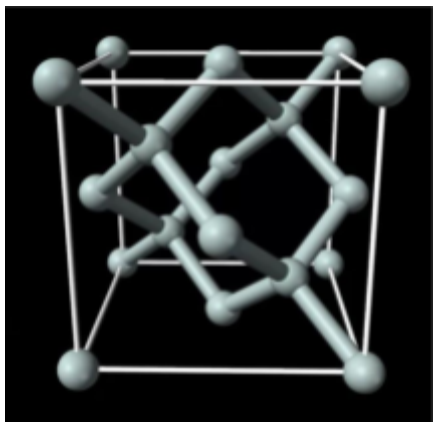


На этом рисунке значение в красной точке выше. Тогда мы поднимаемся в эту точку (выпрыгиваем из ямки) с определенной вероятностью.



Рассмотрим ближе формулу вероятности. Она зависит от разницы значений функции в точках x* и x0 и от температуры. Числитель дроби всегда положительный (f(x*)›f(x)). Значит, вероятность перехода в точку x* будет тем меньше, чем больше разница желтой и красной точки. По сути, это значит, что далеко мы прыгаем неохотно. Также заметим, что вероятность перехода тем больше, чем больше температура. По аналогии с атомами, которые при высоких температурах двигаются быстрее, а при остывании кристаллизуются, то есть занимают определенное положение.

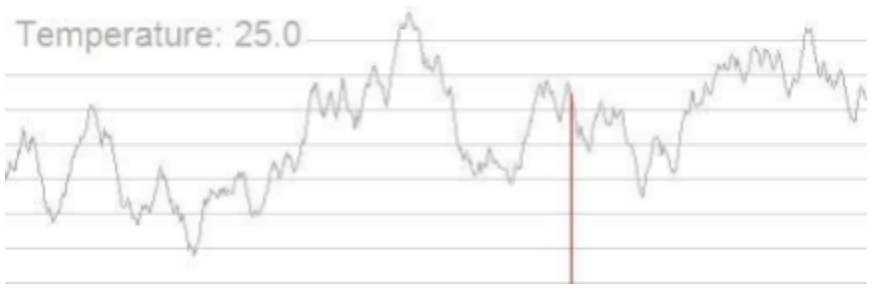
После каждого шага уменьшаем температуру. Останавливаем алгоритм тогда, когда температура достигнет определенного значения или же задаем количество шагов.



Таким образом, благодаря переходам в худшее состояние в методе имитации отжига удалось решить проблему локальных минимумов и не застревать в них. Постепенно, при уменьшении температуры, уменьшается и вероятность переходов в состояния с большим значением функции. Таким образом. в конце имитации отжига в качестве x оказывается искомый глобальный минимум.

Ссылка на википедию, где этот метод реализован для такой непростой функции на рисунке

https://en.wikipedia.org/wiki/Simulated_annealing



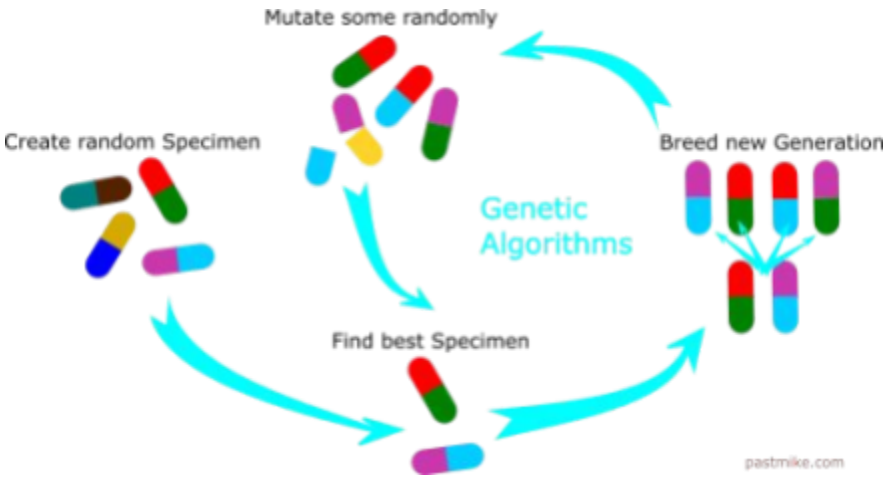
Рассмотрим еще один метод – **генетические алгоритмы.**

Генетические алгоритмы моделируют процесс естественного отбора в ходе эволюции и являются еще одним семейством методов оптимизации.

Генетические алгоритмы включают в себя стадии:

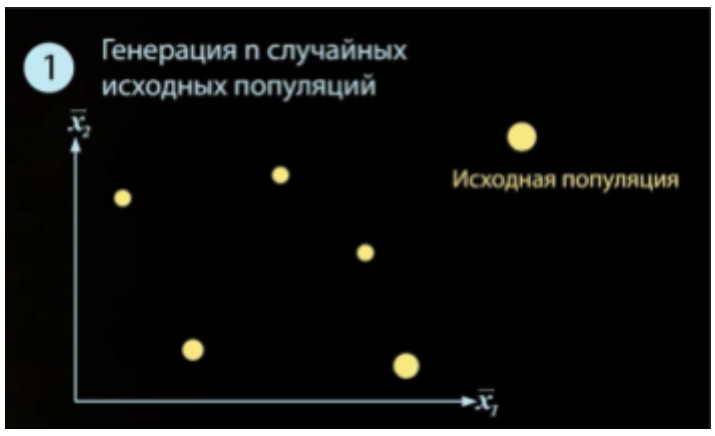
1. генерации популяции
2. мутаций
3. скрещивания
4. отбора

Порядок стадий зависит от конкретного алгоритма.



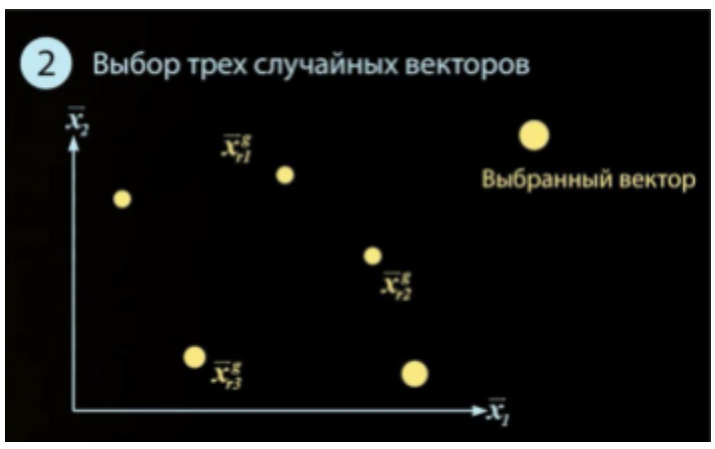
Проведем аналогии. Генерация популяции – это выбор начальной точки. Изменение точки происходит в результате мутации и скрещивания (аналогия с биологией). В результате скрещивания из двух точек получается нечто среднее от этих двух. А далее отбираем лучшие гены, то есть смотрим, какая из точек лучше приближает нас к минимуму.

Для оптимизации функции $f(\vec{x})$ вещественных переменных применяется **алгоритм дифференциальной эволюции**.
Популяция - множество векторов R^n
Размер популяции - N
Сила мутации - F [0, 2]
Вероятность мутации - P
В качестве начальной популяции выбирается набор из N случайных векторов. На каждой следующей итерации алгоритм генерирует новое поколение векторов, комбинируя векторы предыдущего поколения.



На каждой итерации для каждого вектора x_i выбираются 3 неравные ему вектора

$v_1,\ v_2,\ v_3$

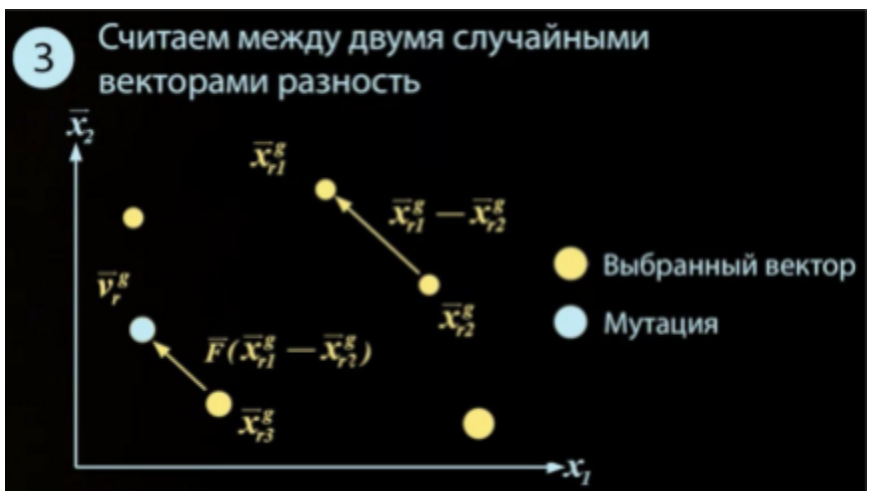


На основе этих векторов генерируется так называемый **мутантный** вектор:

$$v = v_1 + F \cdot (v_2 - v_3)$$

То есть мы определяем смещение от вектора *V3 к V2* и смещаем вектор *V1* в том же направлении. Получим какой то новый вектор v.

На **стадии скрещивания** каждая координата мутантного вектора с вероятностью p замещается соответствующей координатой вектора xi . Получившийся вектор называется пробным.

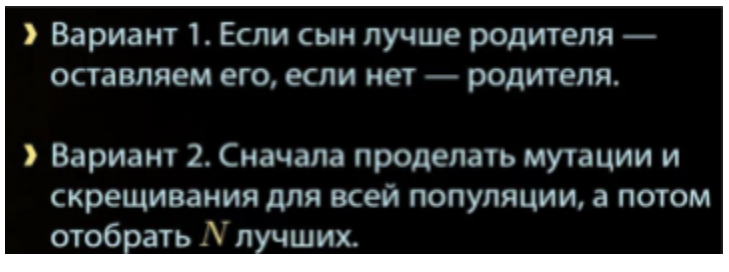


То есть, мы имели некий вектор xi с набором координат и вектор v с со своим набором координат. При скрещивании, по аналогии с биологией , мы получим новый вектор ,каждая координата которого досталась ему по наследству с какой то вероятностью либо от xi либо от v с.

И так для каждого xi мы получим пробный вектор.

Отбор можно осуществлять двумя способами.

1. Сравниваем xi и пробный вектор после каждого шага. И если пробный вектор оказывается лучше исходного xi (то есть значение исследуемой функции на пробном векторе меньше, чем на исходном), то в новом поколении он занимает его место.



2. ПослеN операций, в пространстве будет уже будет 2*N векторов. Вычислим в низ значения функции и выстроим их по возрастанию. Затем выберем N первых .

Останавливаем алгоритм тогда, когда родителей в новой популяции сильно больше, чем детей. То есть новые вектора не сильно приближают к минимуму (дети не лучше родителей), а значит родительские векторы уже в минимуме.
Рассмотренный алгоритм хорошо применим для функций, зависящих от векторов действительных чисел. Часто встречаются зависимости от бинарных векторов.

- В случае бинарных векторов можно определить **мутацию** следующим образом: с вероятностью **p** менять 0->1 и 1->0 в исходных векторах. **Скрещивание** без изменений
- Часто, чтобы увеличить эффективность алгоритма, создаются несколько независимых популяций. Для каждой такой популяции формируется свой начальный набор случайных векторов. В таком случае появляется возможность использовать генетические алгоритмы для решения задач глобальной оптимизации.
- Эффективность метода зависит от выбора операторов мутации и скрещивания для каждого конкретного типа задач.

Рассмотрим еще один метод оптимизации– это **Метод Нелдера-Мида.**
Метод Нелдера-Мида, или метод деформируемого многогранника, применяется для нахождения решения задачи оптимизации вещественных функций многих переменных. На каждой итерации для функции от **n** переменных требуется вычислить значение в **n+1** точке.
Метод прост в реализации и полезен на практике, но для него не существует теории сходимости — алгоритм может расходиться даже на гладких функциях.

Однако же именно он используется по умолчанию в функции **minimize** из **scipy.optimize**.

В чем суть метода.

1) Выбираем n+1 точку, образующие симплекс .

Симплекс - это минимальная фигура n-мерная, внутри которой какие бы две точки мы не взяли, то все точки отрезка, соединяющие эти две точки, тоже находятся внутри фигуры. На рисунке примеры для одно-, двух- и трехмерных пространств.

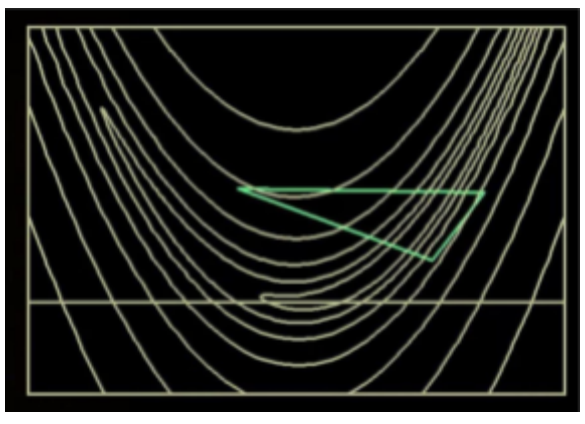
Рис. 2: Отрезок (1-симплекс)

Рис. 3: Треугольник (2-симплекс)

Рис. 4: Тетраэдр (3-симплекс)

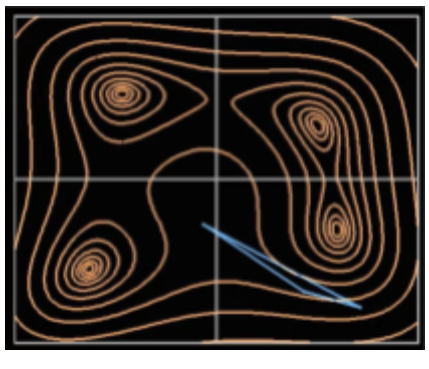
Это нужно для того, чтобы точки пространства были равноудалены друг от друга.

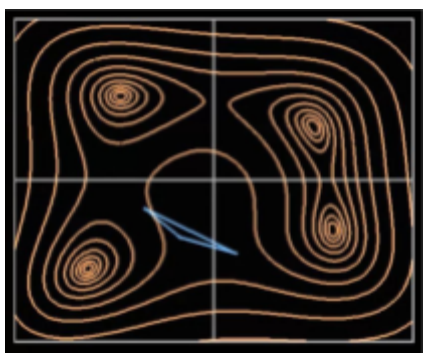
2) Сравниваем значения функции в вершинах

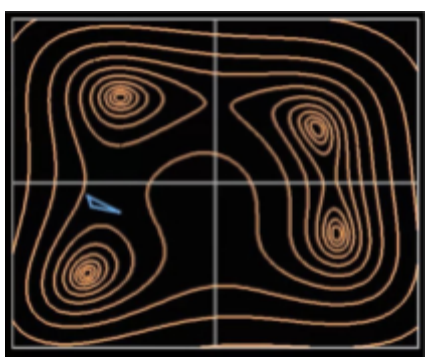


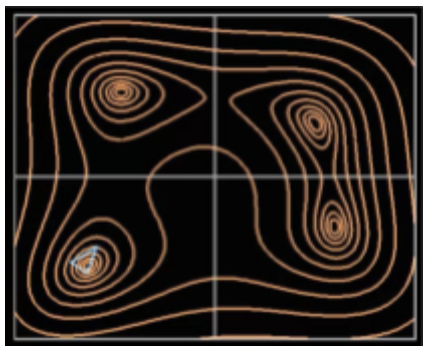
3) деформируем (отражение, сжатие, растяжение, сжатие симплекса) треугольник так, чтобы он “подползал” к минимуму функции

То есть изначально был равносторонний треугольник. А потом он смещается в сторону минимума. Как например, на рисунках









Популярность этого метода заключается в том, что точек фигуры меньше, например, количества точек в популяции. То есть метод Нелдера-Мида гораздо быстрее реализуется.

Notebook

Имитация отжига

Пусть есть функция

In [2]:
```
def f(x):
    return 5 * np.cos(14.5 * x – 0.3) + (x + 0.2) * x
```

она негладкая, поэтому применим метод имитации отжига

есть встроенная функция basinhopping .
импортируем ее из библиотеки `from scipy.optimize import basinhopping`

Задаем ей функцию, начальное приближение, метод оптимизации (писать такой как и здесь) и количество итераций.
`Basinhopping(f, x0, minimizer_kwargs=minimizer_kwargs, niter=20)`

Итог:

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

from scipy.optimize import dual_annealing
from scipy.optimize import differential_evolution
from scipy.optimize import basinhopping

%matplotlib inline
```

Имитация отжига

In [2]:
```
def f(x):
    return 5 * np.cos(14.5 * x – 0.3) + (x + 0.2) * x
```

In [3]:
```
x = np.linspace(-10, 10, 500)
fx = f(x)
```

In [4]:
```
plt.figure(figsize=(10,7))
plt.plot(x, fx)
plt.show()
```

```
In [5]:x0=[1.]
In [6]:minimizer_kwargs = {“method”: “BFGS”}
ret = basinhopping(f, x0, minimizer_kwargs=minimizer_kwargs, niter=20)
print(“global minimum: x = %.4f, f(x) = %.4f” % (ret.x, ret.fun))
```

global minimum: x = -0.1958, f(x) = -5.0008 - результат

```
In [7]:plt.figure(figsize=(10,7))
plt.plot(x, fx)
plt.plot([ret.x], [ret.fun], ‘ro’)
plt.show()
```

Эта функция basinhopping хорошо работает для функции от 2 до 10 переменных. Но плохо для функций от ,например, 100 переменных. Для второго случая есть встроенная функция dual_annealing. Но там другой алгоритм , мы его не рассматривали на лекции.

Дифференциальная эволюция

https://en.wikipedia.org/wiki/Test_functions_for_optimization

На страничке википедии (см ссылку) есть некоторые функции «трудные», на которых можно тестировать разные методы оптимизации

Зададим такую функцию

```
In [8]:def ackley(x):
    arg1 = -0.2 * np.sqrt(0.5 * (x[0] ** 2 + x[1] ** 2))
    arg2 = 0.5 * (np.cos(2. * np.pi * x[0]) + np.cos(2. * np.pi * x[1]))
    return -20. * np.exp(arg1) – np.exp(arg2) + 20. + np.e
```

задаем пределы изменения переменных

```
bounds = [(-5, 5), (-5, 5)]
```

импортируем функцию

differential_evolution(ckley, bounds, seed=42) – передаем ей функцию и границы (последнюю координату seed=42 можно не писать

и запускаем

Итог:

```
In [8]:def ackley(x):
    arg1 = -0.2 * np.sqrt(0.5 * (x[0] ** 2 + x[1] ** 2))
    arg2 = 0.5 * (np.cos(2. * np.pi * x[0]) + np.cos(2. * np.pi * x[1]))
    return -20. * np.exp(arg1) – np.exp(arg2) + 20. + np.e

bounds = [(-5, 5), (-5, 5)]
result = differential_evolution(ckley, bounds, seed=42)
result
Out[8]:     fun: 4.440892098500626e-16 – минимум
 message: ‘Optimization terminated successfully.’
    Nfev: 2973
     nit: 96
 success: True
       x: array([0., 0.])- в этой точке достигается минимум
In [9]:result.x, result.fun
Out[9]☹array([0., 0.]), 4.440892098500626e-16)
```