**Оптимизация негладких функций**

Не все функции одинаково хороши.  
  
Представим, что мы обучаем какой-нибудь алгоритм машинного обучения, благо мы их расскажем в оставшейся части специализации целую уйму, и у него есть какие-то параметры α1, α2, ..., αN. Эти параметры могут быть самыми разными: действительные числа, целые числа, просто нолики и единички — ну в общем, какие-то числа. И хотим мы их подобрать таким образом, чтобы качество работы нашего алгоритма было максимальным. Казалось бы, причем здесь математика? Но оказывается, значений параметров может быть очень много, комбинаций очень много, и поэтому просто в лоб перебрать достаточно затруднительно. И это повод для оптимизационного процесса.

Давайте просто рассмотрим качество работы алгоритма как функцию от его параметров и будем подбирать параметры так, чтобы максимизировать качество. Ну или если мы измеряем ошибку алгоритма, то тогда подбираем параметры так,

чтобы минимизировать ошибку. Неважно, в любом случае мы решаем задачу поиска экстремума. Мы с вами уже знаем один из методов численной оптимизации — это градиентный спуск.

Поможет ли нам градиентный спуск в произвольном случае?

Существует несколько подводных камней:

* наша функция от параметров запросто может не иметь градиента,
* может быть негладкой
* может быть достаточно затруднительно этот градиент вычислять, даже если он есть.

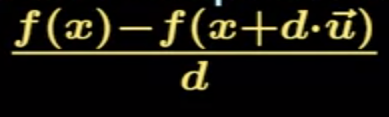
Помимо этого, существует также проблема локальных минимумов. Локальные минимумы — конечно, минимумы, но попасть-то хочется в глобальный, потому что мы хотим получить наименьшее значение функции, и, конечно же,

градиентный спуск ничего особо не может сделать с этими локальными минимумами.

Возникает идея придумать какие-нибудь новые методы,   
например методы *случайного поиска*.

Общая идея этих методов заключается в том, что на каждом следующем шаге мы переходим в некоторую *случайную точку*, но с такой *вероятностью*, чтобы в среднем мы двигались туда, куда нужно, то есть к *минимуму.*

На шаге 0 мы зафиксируем какой-то числовой параметр d и выберем вектор u, который будет задавать некоторое направление случайным образом из равномерного распределения на единичной сфере. На следующем шаге мы просто сместимся в направлении вектора u на величину

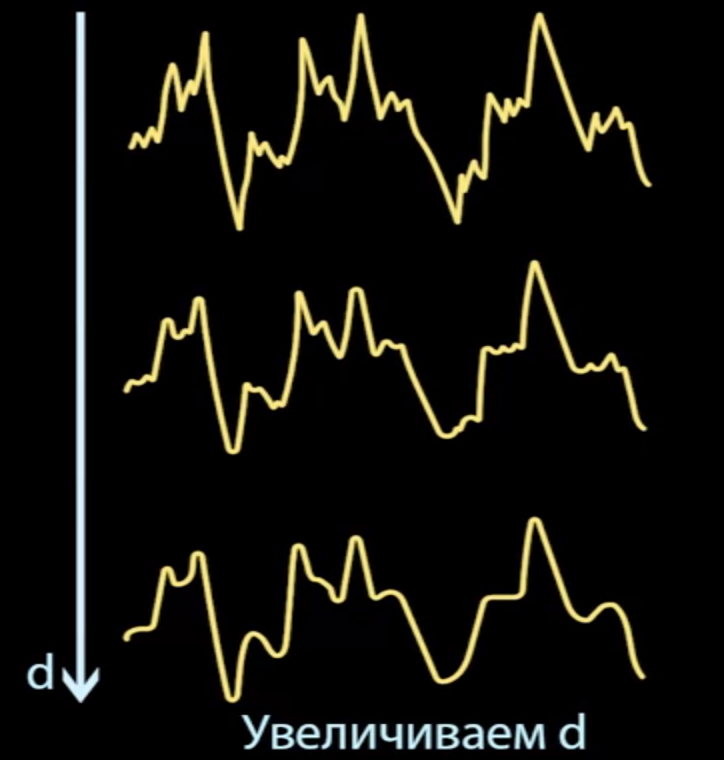


Обратим внимание: мы нигде не вычисляем градиент, и смещаемся мы в каких-то случайных направлениях,

но за счет того, что величина этого смещения зависит от значения функции в точке (x + d \* u), мы добиваемся того,

что в среднем мы двигаемся по антиградиенту, ну если он есть. Если его нет, то в среднем мы двигаемся по антиградиенту сглаженной функции.

Параметр d влияет на то, как сильно мы сглаживаем функцию.



Таким образом, выбирая большие d, мы отступаем больше и функция получается более сглаженной.

**Метод имитации отжига**

Это один из методов глобальной оптимизации, которому совершенно не важно, обладает ли функция свойством гладкости, это один из вариантов случайного поиска, и также вы можете встретить его под названием «алгоритм Метрополиса» по имени автора.



Алгоритм был придуман достаточно просто.

Зачем нам самим выдумывать какой-то сложный алгоритм?

Мы можем подсмотреть какую-то хорошую идею у природы.

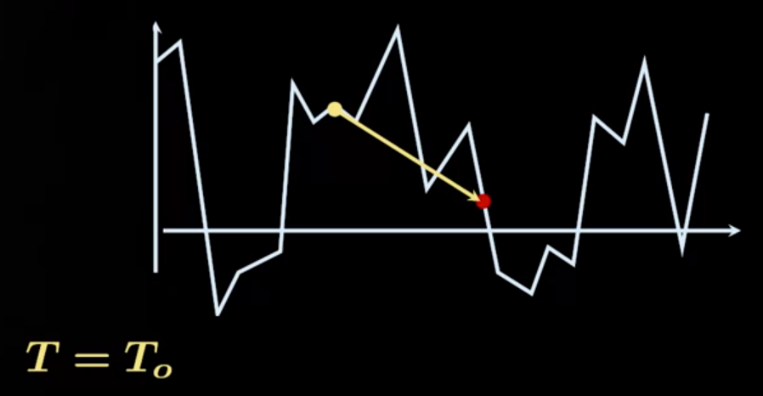
И вот действительно, есть такой хороший физический процесс, который наблюдается при отжиге металлов, в котором атомы оказываются в ситуации, когда они уже заняли какие-то свои положения в кристаллической решётке, но в

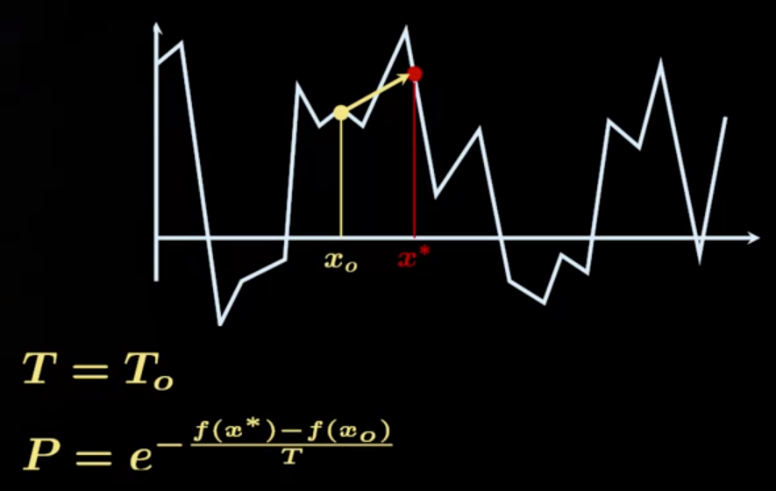
то же время всё ещё возможны небольшие переходы из одной ячейки в другую. При этом температура падает,

с падением температуры вероятность переходов тоже падает, ну и, кроме того, решётка стремится принять состояние, в котором энергия атомов будет минимальной. Ну то есть это некоторая естественная оптимизация, которую мы можем наблюдать в природе. Поэтому переходы атомов происходят тогда, когда энергия атомов от этого перехода уменьшится.



Зададим начальное значение температуры и дальше попробуем как-то случайно выбрать следующую точку недалеко от предыдущей. Ну чтобы мы не выбивались слишком сильно из уже хороших найденных точек. Если эта точка подходит нам больше, то есть значение функции в ней меньше, если мы решаем задачу минимизации, то тогда сразу переходим в неё.

Если эта точка подходит нам больше, то есть значение  
функции в ней меньше, если мы решаем задачу минимизации, то тогда сразу переходим в неё.



Если же значение функции в этой точке больше,

то мы всё равно можем перейти в неё, это обеспечит нам возможность выбираться из каких-то локальных минимумов,

но давайте это делать с некоторой вероятностью, указанной на картинке (*P*). Обратите внимание: с ростом температуры эта вероятность становится всё *больше*.

*T = T1*  
(*Tk  должны быть положительными и убывать с ростом* ***k***)

После завершения шага давайте обновим значение температуры, уменьшив его, и будем продолжать шаги до тех пор, пока температура продолжает падать или пока мы не нашли хорошее решение, из которого никуда не уходим.

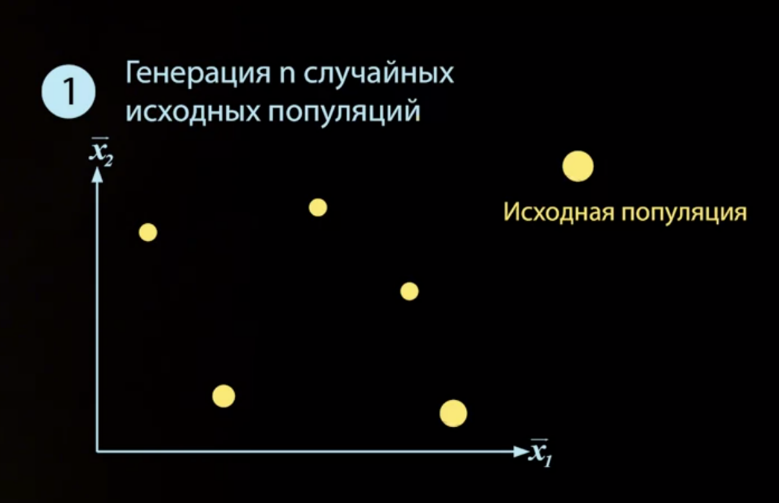
*На самом деле, когда мы говорим об алгоритмах оптимизации,*

*не всегда подразумевается, что мы хотим найти именно минимум. Бывают ситуации, когда мы просто хотим найти значение функции поменьше, чем то, которое мы уже знаем.   
нахождение минимума и выбор следующей точки — достаточно тесно связанные между собой вещи. При некоторых способах выбора следующей точки можно гарантировать, что алгоритм уж по крайней мере будет находить какую-нибудь точку получше начального приближения (разумеется, если она есть).*

**Генетические алгоритмы**

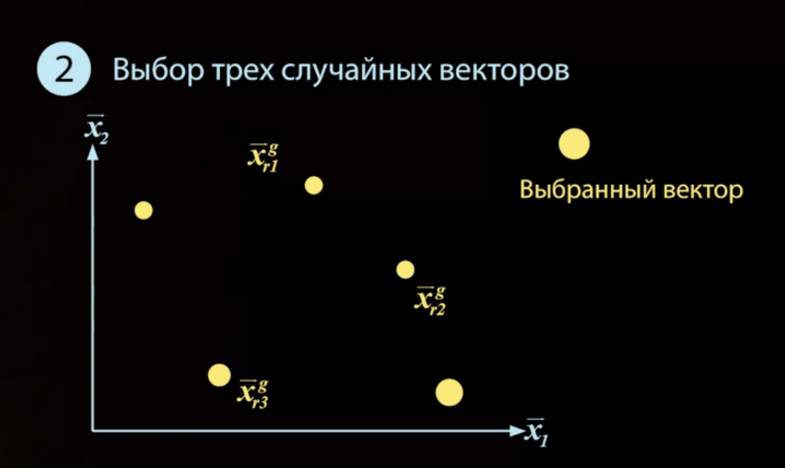
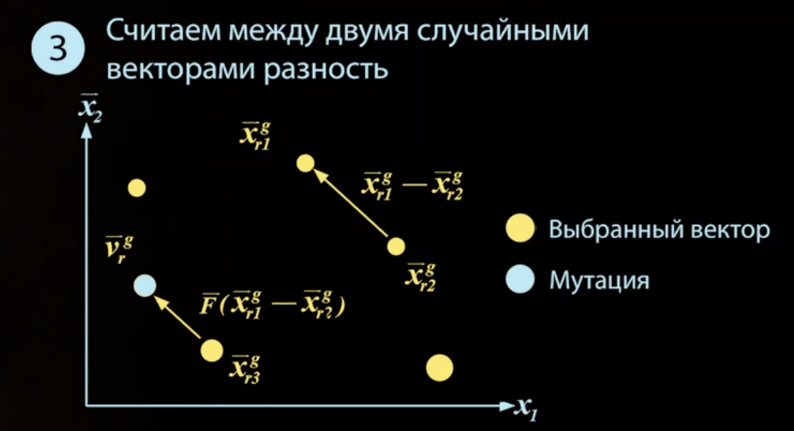
Генетические алгоритмы моделируют процесс эволюции, поэтому содержат в себе стадии генерации популяции и повторяемые стадии мутации, скрещивания и отбора.

Порядок стадий может варьироваться.

Дифференциальная эволюция — это алгоритм, который используется для оптимизации функции от вещественных переменных.  
  


Для начала сгенерируем n случайных векторов и будем называть их в дальнейшем

«популяцией».

  
  
Теперь выберем какой-то вектор из популяции и выберем еще  
три случайных вектора из популяции.  
  
  
  
Два из этих случайных используем, чтобы посчитать между ними разность и в направлении этой разности сдвинуться от третьего, получив таким образом некоторый итоговый вектор, называемый «мутантным вектором».

Дальше будем выполнять операцию скрещивания мутантного вектора и вектора, который мы выбрали изначально.

Потомок должен обладать свойствами обоих родителей,

поэтому операция скрещивания может выполняться, например, следующим образом: давайте пройдемся по всем координатам и с некоторой фиксированной вероятностью p будем брать первого родителя и брать его координату,

а с вероятностью 1 − p будем брать координату от второго родителя.

После этого наступает стадия отбора. Здесь у нас есть много вариантов, как ее провести:

1. Первый вариант — мы можем смотреть, стал ли сын лучше родителя. То есть, стало ли значение функции меньше оттого,

что мы подставляем в нее вместо родителя — сына.

Если сын действительно лучше, то удаляем родителя из популяции и используем сына.

2. Второй вариант — это пройти по всем векторам из популяции,

применить скрещивание и мутации, сгенерировать еще

n объектов для популяции и потом уже из 2n объектов выбрать N лучших.

Походя по всей популяции, выполняя стадии мутации, скрещивания и отбора, и повторяя это до сходимости мы получаем метод оптимизации, который находит какие-то достаточно хорошие значения функции, ну то есть те значения, в которых она будет меньше, чем в начальных приближениях.

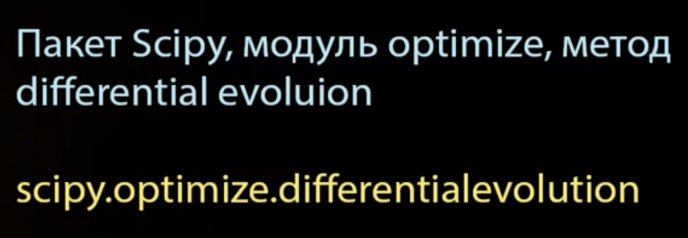
*Другой пример:* если мы оптимизируем функцию от векторов из *ноликов и единичек*. В этом случае операция мутации из дифференциальной эволюции может быть не очень хороша. Ну хотя бы потому, что у нас может получиться не вектор из ноликов и единичек. Поэтому можно ее изменить: можно просто пройтись по всем координатам и для каждой координаты с *некоторой вероятностью* делать изменения. То есть вместо нолика сделать единичку, вместо единички сделать нолик. Операция скрещивания остается без изменений.

Иногда применяются некоторые трюки для того,

чтобы методы основанные на эволюционных идеях, работали лучше. Например, можно создавать несколько *«островков»* из разных популяций и применять алгоритм для каждой популяции. С ростом числа *координат* метод начинает работать резко медленней. Это тоже нужно учитывать и не пытаться его применять в очень многомерных

пространствах. Кроме того, стоит помнить, что операторы мутации и скрещивания бывают разными,

Реализация на Python:



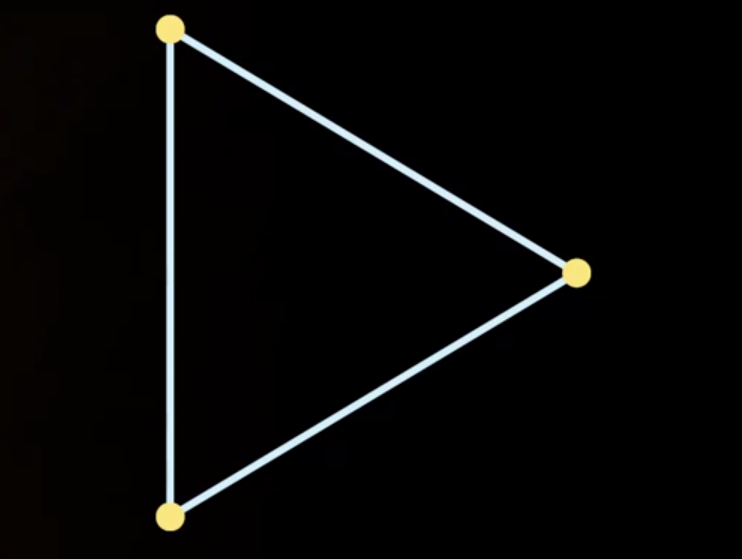
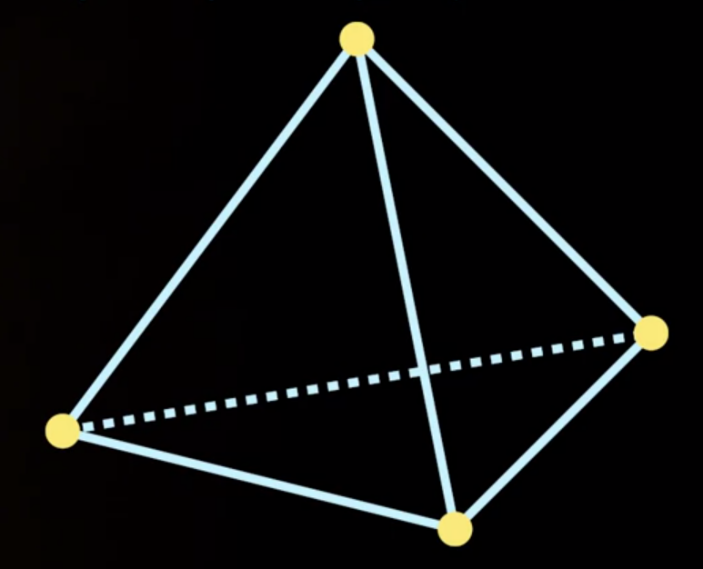
**Метод Нелдер-Мида**

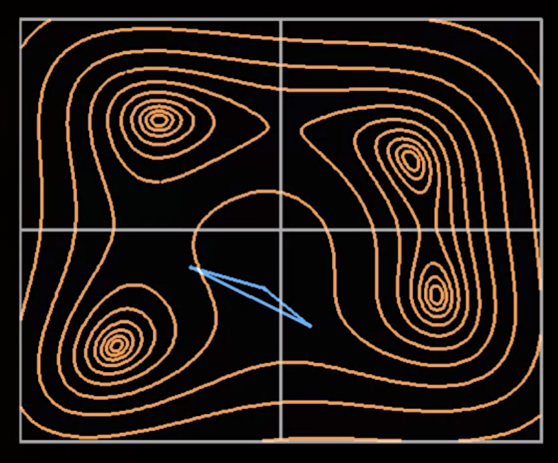
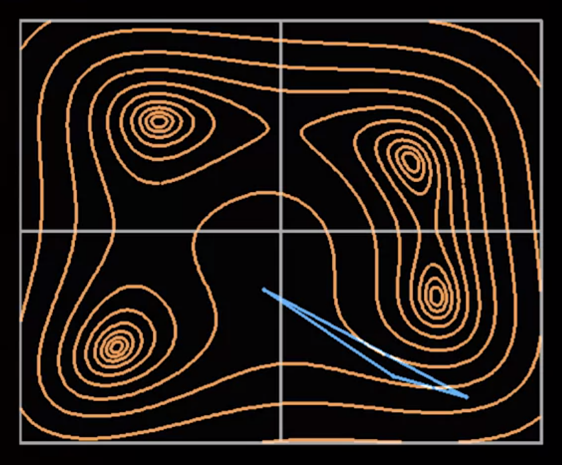
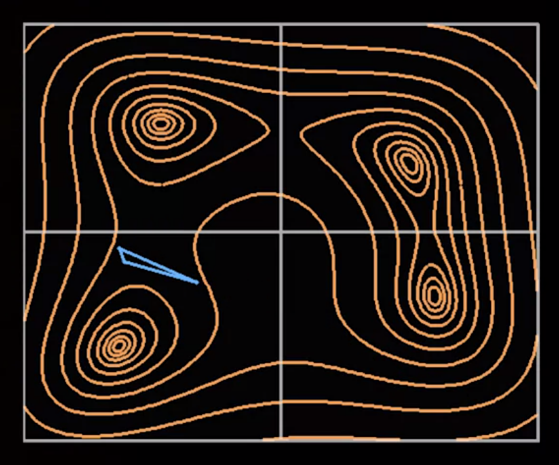
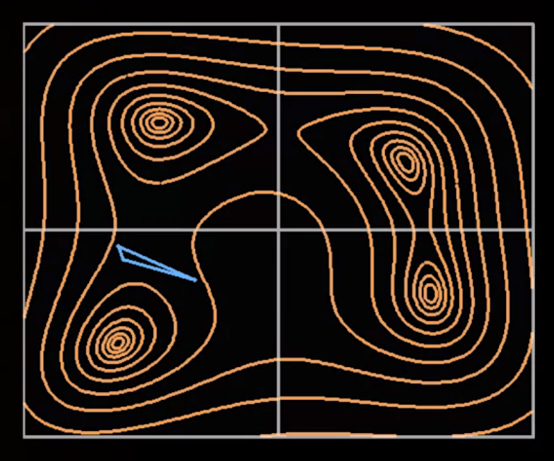
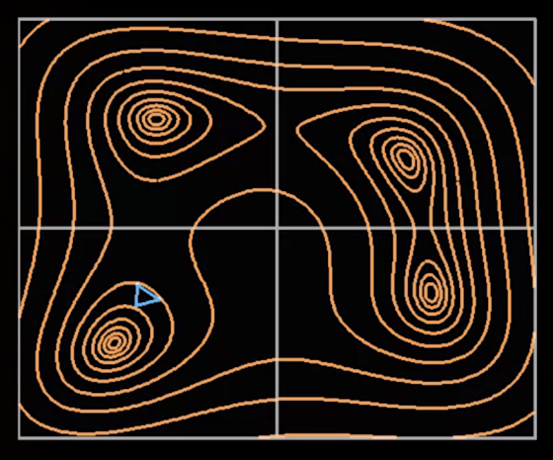
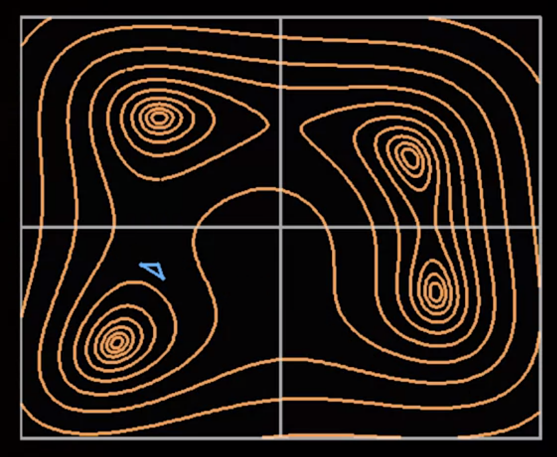
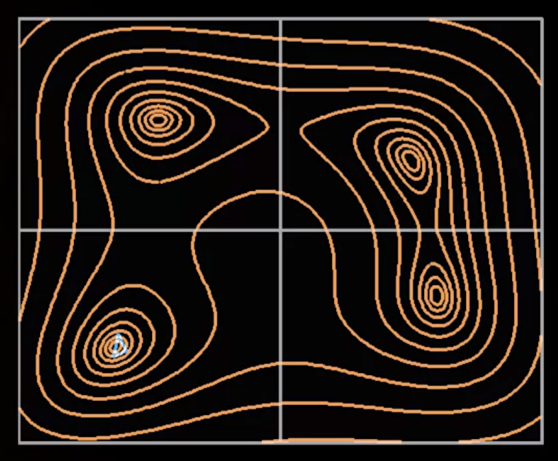
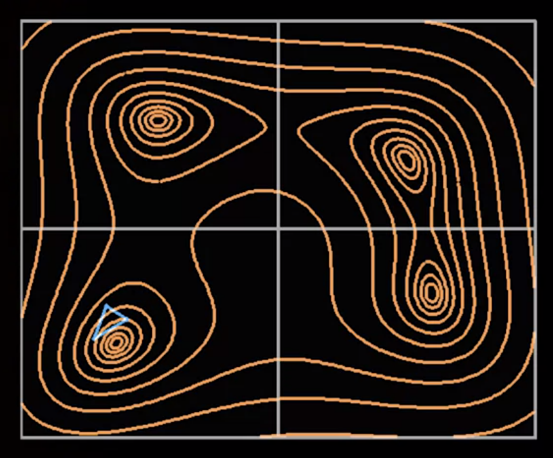
Этот метод оптимизации используется очень часто, работает довольно неплохо, поэтому знать его будет полезно.

Применяется он для оптимизации функций, в том числе негладких и зашумленных.

Устроен очень просто, и его легко реализовать.

При этом в подробном изложении этого метода есть большое количество нетривиальных эвристик, которые, однако, работают. Этот метод используется по умолчанию в функции *minimize* из модуля *scipy.optimize*, и еще он может иногда не сходиться или сходиться не к минимуму, но обычно очень хорош на практике.

Начало работы алгоритма выглядит следующим образом: мы выбираем n + 1 начальную точку для случая оптимизации функции от n переменных и пытаемся их выбрать так, чтобы они были не слишком далеко от предполагаемого экстремума, то есть не надо специально забираться подальше, и эта n + 1 точка образует симплекс.   
  
  
  
Для прямой (то есть в одномерном пространстве) — это будет просто отрезок,  
  
  
  
для плоскости — это будет треугольник,   
  
  
  
для пространства трехмерного — это будет тетраэдр. Ну а в n-мерном случае — это просто некоторое n-мерное обобщение тетраэдра.

**Симплекс** или n-мерный **тетраэдр** — геометрическая фигура, являющаяся n-мерным обобщением треугольника.  
  
  
  
Если мы посмотрим в двумерном случае на линии уровня функции, то получится, что мы выбрали какие-то три точки, смотрим на треугольник, построенный на этих трех точках, и дальше просто пытаемся деформировать этот треугольник таким образом, чтобы он постепенно подползал к минимуму и стягивался вокруг него.  
  
  
  
  


Именно из-за этого поведения этот метод иногда называют «Амебой».