**Simulated annealing in Python**

The Physics Behind the Algorithm

SA is a metaheuristic optimization technique introduced by Kirkpatrick et al. in 1983 to solve the Travelling Salesman Problem (TSP).

The SA algorithm is based on the annealing process used in metallurgy, where a metal is heated to a high temperature quickly and then gradually cooled. At high temperatures, the atoms move fast, and when the temperature is reduced, their kinetic energy decreases as well. At the end of the annealing process, the atoms fall into a more ordered state, and the material is more ductile and easier to work with.

Similarly, in SA, a search process starts with a high-energy state (an initial solution) and gradually lowers the temperature (a control parameter) until it reaches a state of minimum energy (the optimal solution).

SA has been successfully applied to a wide range of optimization problems, such as TSP, protein folding, graph partitioning, and job-shop scheduling. The main advantage of SA is its ability to escape from local minima and converge to a global minimum. SA is also relatively easy to implement and does not require a priori knowledge of the search space.

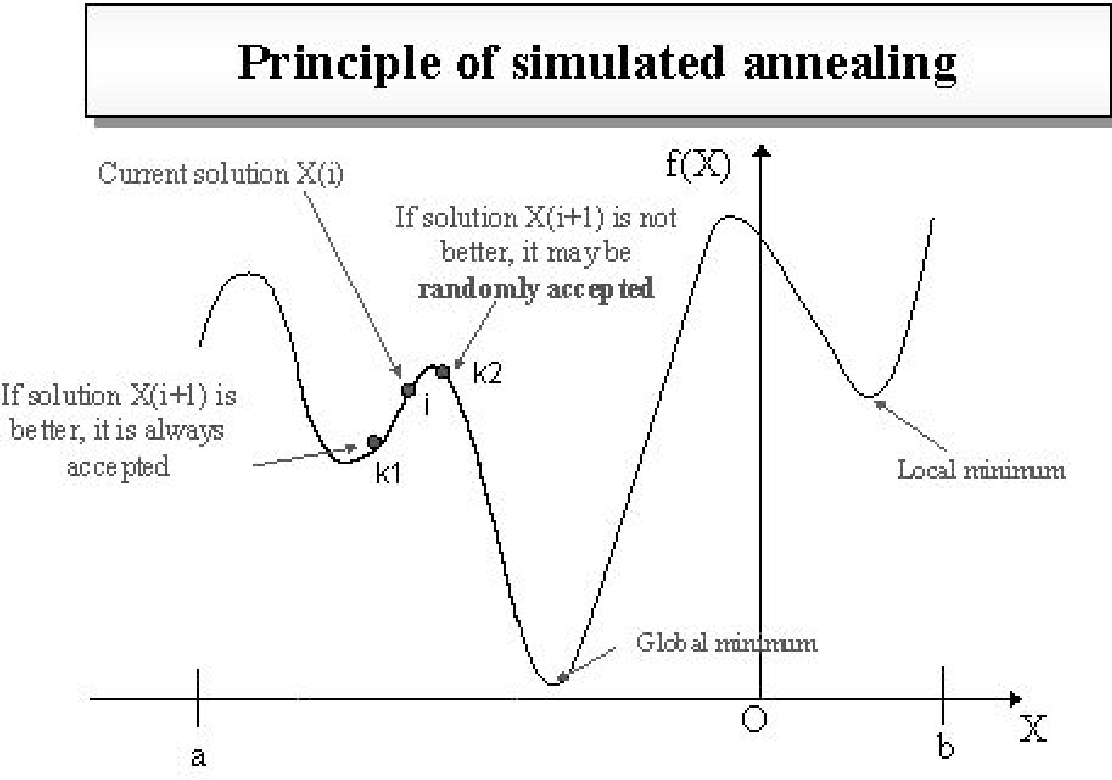
Simulated Annealing (SA) is an effective and general form of optimization.  It is useful in finding global optima in the presence of large numbers of local optima.  “Annealing” refers to an analogy with thermodynamics, specifically with the way that metals cool and anneal.  Simulated annealing uses the objective function of an optimization problem instead of the energy of a material.

Implementation of SA is surprisingly simple.  The algorithm is basically hill-climbing except instead of picking the best move, it picks a random move.  If the selected move improves the solution, then it is always accepted.  Otherwise, the algorithm makes the move anyway with some probability less than 1.  The probability decreases exponentially with the “badness” of the move, which is the amount **deltaE** by which the solution is worsened (i.e., energy is increased.)

**Prob(accepting uphill move) ~ 1 - exp(deltaE / kT))**

A parameter **T** is also used to determine this probability.  It is analogous to temperature in an annealing system.  At higher values of T, uphill moves are more likely to occur.  As **T** tends to zero, they become more and more unlikely, until the algorithm behaves more or less like hill-climbing.  In a typical SA optimization, **T** starts high and is gradually decreased according to an “annealing schedule”.  The parameter **k** is some constant that relates temperature to energy (in nature it is Boltzmann’s constant.)

Simulated annealing is typically used in discrete, but very large, configuration spaces, such as the set of possible orders of cities in the Traveling Salesman problem.

****

**Example**

from \_\_future\_\_ import print\_function, division # Python 2 compatibility if needed

import numpy as np

import numpy.random as rn

import matplotlib.pyplot as plt # to plot

import matplotlib as mpl

from scipy import optimize # to compare

import seaborn as sns

sns.set(context="talk", style="darkgrid", palette="hls", font="sans-serif", font\_scale=1.05)

FIGSIZE = (19, 8) #: Figure size, in inches!

mpl.rcParams['figure.figsize'] = FIGSIZE

def annealing(random\_start,

cost\_function,

random\_neighbour,

acceptance,

temperature,

maxsteps=1000,

debug=True):

""" Optimize the black-box function 'cost\_function' with the simulated annealing algorithm."""

state = random\_start()

cost = cost\_function(state)

states, costs = [state], [cost]

for step in range(maxsteps):

fraction = step / float(maxsteps)

T = temperature(fraction)

new\_state = random\_neighbour(state, fraction)

new\_cost = cost\_function(new\_state)

if debug: print("Step #{:>2}/{:>2} : T = {:>4.3g}, state = {:>4.3g}, cost = {:>4.3g}, new\_state = {:>4.3g}, new\_cost = {:>4.3g} ...".format(step, maxsteps, T, state, cost, new\_state, new\_cost))

if acceptance\_probability(cost, new\_cost, T) > rn.random():

state, cost = new\_state, new\_cost

states.append(state)

costs.append(cost)

# print(" ==> Accept it!")

# else:

# print(" ==> Reject it...")

return state, cost\_function(state), states, costs

interval = (-10, 10)

def f(x):

""" Function to minimize."""

return x \*\* 2

def clip(x):

""" Force x to be in the interval."""

a, b = interval

return max(min(x, b), a)

def random\_start():

""" Random point in the interval."""

a, b = interval

return a + (b - a) \* rn.random\_sample()

def cost\_function(x):

""" Cost of x = f(x)."""

return f(x)

def random\_neighbour(x, fraction=1):

"""Move a little bit x, from the left or the right."""

amplitude = (max(interval) - min(interval)) \* fraction / 10

delta = (-amplitude/2.) + amplitude \* rn.random\_sample()

return clip(x + delta)

def acceptance\_probability(cost, new\_cost, temperature):

if new\_cost < cost:

# print(" - Acceptance probabilty = 1 as new\_cost = {} < cost = {}...".format(new\_cost, cost))

return 1

else:

p = np.exp(- (new\_cost - cost) / temperature)

# print(" - Acceptance probabilty = {:.3g}...".format(p))

return p

def temperature(fraction):

""" Example of temperature dicreasing as the process goes on."""

return max(0.01, min(1, 1 - fraction))

annealing(random\_start, cost\_function, random\_neighbour, acceptance\_probability, temperature, maxsteps=30, debug=True);

state, c, states, costs = annealing(random\_start, cost\_function, random\_neighbour, acceptance\_probability, temperature, maxsteps=1000, debug=False)

state

c

def see\_annealing(states, costs):

plt.figure()

plt.suptitle("Evolution of states and costs of the simulated annealing")

plt.subplot(121)

plt.plot(states, 'r')

plt.title("States")

plt.subplot(122)

plt.plot(costs, 'b')

plt.title("Costs")

plt.show()

see\_annealing(states, costs)

def visualize\_annealing(cost\_function):

state, c, states, costs = annealing(random\_start, cost\_function, random\_neighbour, acceptance\_probability, temperature, maxsteps=1000, debug=False)

see\_annealing(states, costs)

return state, c

visualize\_annealing(lambda x: x\*\*3)

visualize\_annealing(lambda x: x\*\*2)

visualize\_annealing(np.abs)

visualize\_annealing(np.cos)

visualize\_annealing(lambda x: np.sin(x) + np.cos(x))