Simulated annealing from scratch

Consideramos el benchmark en optimización global de la función de Ackley

$$f(x) = -20e^{-0.2\sqrt{0.5x^2}} - e^{0.5\cos(2\pi x)} + e + 20$$

en el intervalo $[-5, 5]$.

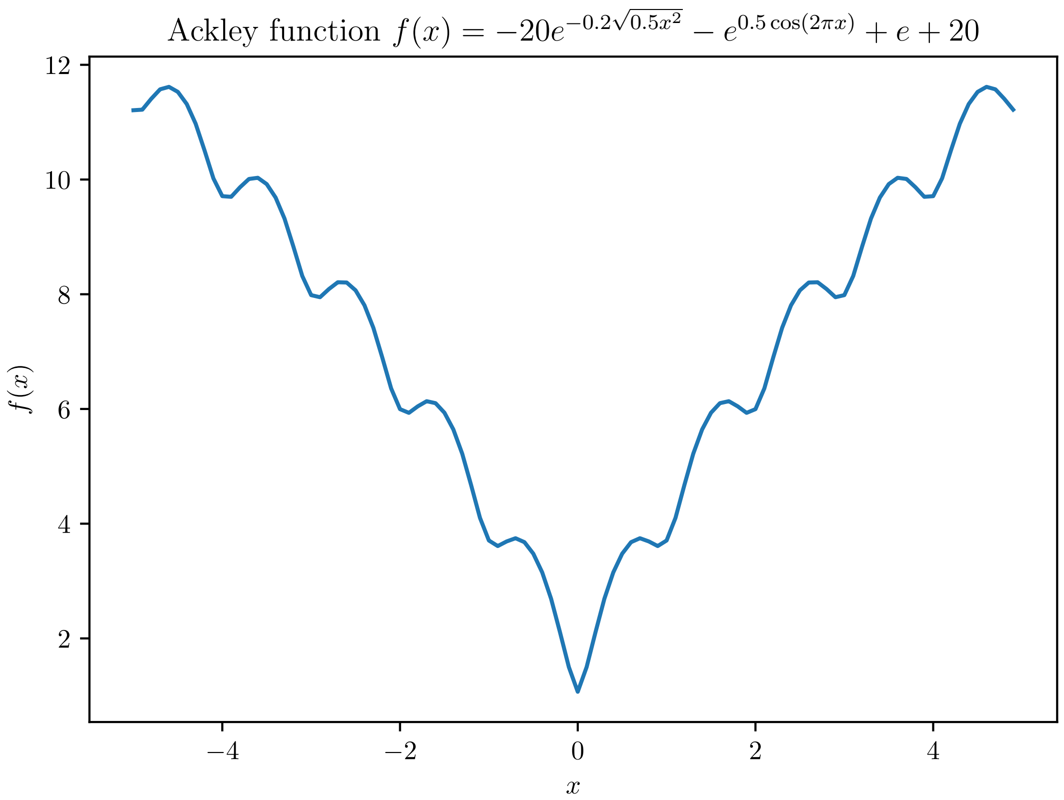
```
import numpy as np
import matplotlib.pyplot as plt

# objective function

def objective(x):
  return -20.0 * np.exp(-0.2 * np.sqrt(0.5 * (x[0] ** 2 ))) - np.exp(0.5 * (np.cos( 2 * np.pi *
   ↪  x[0]))) + np.e + 20

# define range for input
r_min, r_max = -5.0, 5.0
# sample input range uniformly at 0.1 increments
inputs = np.arange(r_min, r_max, 0.1)
# compute targets
results = [objective([x]) for x in inputs]
# parámetros para el plot
plt.rcParams.update({
  "text.usetex": True,
  "font.family": "serif",
  "text.latex.preamble": r"\usepackage{amsmath}"
})

fig, ax = plt.subplots()
ax.plot(inputs, results)
ax.set_title(r'Ackley function $f(x)=-20e^{-0.2\sqrt{0.5x^2}}-e^{0.5\cos(2\pi x)}+e+20$')
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$f(x)$')
# mostramos el gráfico
plt.show()
```




```
def simulated_annealing(objective, bounds, n_iterations, step_size, temp):
    # generate an initial point randomly using a uniform distribution
    best = bounds[:, 0] + np.random.randn(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # forzamos a que empiece lejos el óptimo global
    best = -4 * best
    # evaluate initial point
    best_eval = objective(best)
    # current working solution
    curr, curr_eval = best, best_eval
    scores = list()
    # run the algorithm
    for i in range(n_iterations):
        # take a step randomly following a normal distribution
        candidate = curr + np.random.randn(len(bounds)) * step_size
        # evaluate candidate point
        candidate_eval = objective(candidate)
        # check for new best solution
        if candidate_eval < best_eval:
            # store new best point
            best, best_eval = candidate, candidate_eval
            scores.append(best_eval)
            # report progress
            print('>%d f(%s) = %.5f' % (i, best, best_eval))
        # difference between candidate and current point evaluation
        diff = candidate_eval - curr_eval
```

```
        # calculate temperature for current epoch
        t = temp / float(i + 1)
        # calculate metropolis acceptance criterion
        metropolis = np.exp(-diff / t)
        # check if we should keep the new point (menor coste o al elegir un número aleatorio)
        if diff < 0 or np.random.rand() < metropolis:
            # store the new current point
            curr, curr_eval = candidate, candidate_eval
    return [best, best_eval, scores]
```

Ejecutamos varias veces el simulated annealing algo porque la aleatoriedad juega un papel decisivo

```
# seed the pseudorandom number generator for reproducibility
# np.random.seed(1)
# define range for input
bounds = np.asarray([[-5.0, 5.0]])
# define the total iterations
n_iterations = 1000
# define the maximum step size
step_size = 0.1
# initial temperature
temp = 10
# perform the simulated annealing search
best, score, scores = simulated_annealing(objective, bounds, n_iterations, step_size, temp)
print('Done!')
print('f(%s) = %f' % (best, score))

fig, ax = plt.subplots()
ax.plot(scores, '.-')
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$f(x)$')

plt.show()
```

```
## >0 f([-28.62425942]) = 21.66813
## >1 f([-28.64673732]) = 21.63103
## >4 f([-28.65818797]) = 21.60952
## >11 f([-28.33819032]) = 21.58608
## >12 f([-28.27023107]) = 21.41266
## >24 f([-28.25933566]) = 21.37956
## >25 f([-28.23300251]) = 21.29456
## >26 f([-28.18285851]) = 21.11948
## >27 f([-28.14933127]) = 21.00099
## >29 f([-27.8641465]) = 20.94047
## >30 f([-28.02008691]) = 20.69585
## >31 f([-28.01753405]) = 20.69416
```

```
## >35 f([-27.99802184]) = 20.68818
## >178 f([-27.99838728]) = 20.68818
## Done!
## f([-27.99838728]) = 20.688177
```