Randomized Optimization: Simulated Annealing Algorithm

Neba Nfonsang
University of Denver

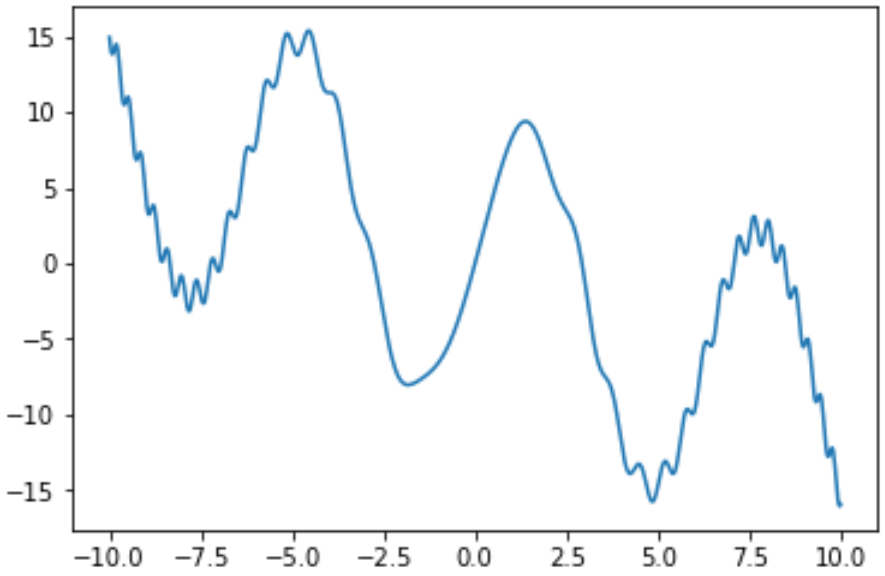
In [1]:
```
# import packages needed
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Overview

- The simulated annealing algorithm can be used to optimize a cost function given a search space. A cost function is a function that maps a search space to real values. This algorithm is useful for minimization or maximization problems. A use case for the algorithm includes finding the optimal solution to the travelling salesman problem, that is, finding the shortest distribution route.
- This stochastic algorithm start by randomly selecting a state in the search space, then takes a random walk to find the optimal value of the cost function. The algorithm is able to skip over local minima or maxima, which makes it more advantageous compared to other optimization algorithms that get stuck in local optima.
- Simulated annealing finds the optimal solution by starting off at a high ‘temperature’ and slowly cooling down until a solution is found, close to the crystal structure (optimal value of the cost function).
- This algorithm is inspired by the annealing technique used in metallurgy, in cooling hot metals until a stable crystal structure is obtained.

A Function with Local and Global Optima

In [2]:
```
# create data with local and global optima
x_input = np.linspace(-10, 10, 1000)
def func(x):
    return np.sin((x)**2) + 10*np.sin(x) - x

y_output = func(x_input)
plt.plot(x_input, y_output)
plt.show()
```



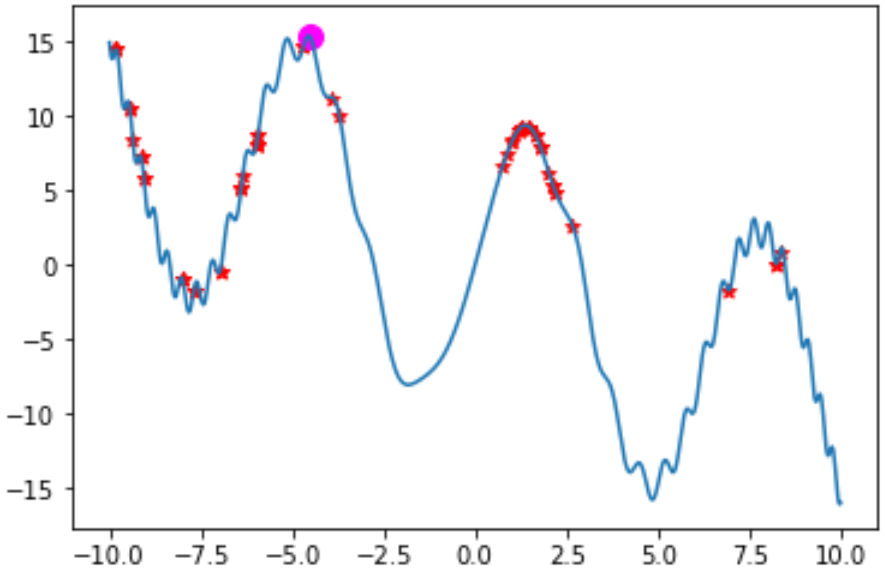
The Simulated Annealing Algorithm

In [3]:
```python
# define the simulated annealing function

def simulated_annealing(search_space, func, T=10, alpha=0.1):
    """
    Returns the maximum value of the search_space
    """
    global search_history
    global x_best
    search_history = []

    np.random.seed(0)
    x_current = np.random.choice(search_space)
    x_best = x_current

    while T > 0:
        x_next = np.random.choice(search_space) # random walk
        f_current = func(x_current)
        f_next = func(x_next)
        if f_next > f_current:
            x_current = x_next
            if func(x_next) > func(x_best):
                x_best = x_next
        elif np.exp((f_next - f_current)/T) > np.random.random():
            x_current = x_next
        T = T-alpha # reduce T
        search_history.append(x_current)

    return x_best, func(x_best)

simulated_annealing(x_input, func)
```

Out[3]:
```
(-4.534534534534535, 15.366774224096893)
```

The Search History and the Maximum Point

In [4]:
```
# plot the search history and optimal point
history_output = func(np.array(search_history))
plt.scatter(search_history, history_output, marker="*", c="r")
plt.scatter(x_best, func(x_best), s=100, marker="o", c="magenta")
plt.plot(x_input, y_output)
plt.show()
```



The algorithm was able to find the maximum point (the magenta dot on the graph)

Resource

see pseudo code in https://www.lancaster.ac.uk/~varty/RTOne.pdf (https://www.lancaster.ac.uk/~varty/RTOne.pdf)