CS 767 Machine Learning
Project #2
Using Simulated Annealing to Minimize a Function

Spring 2017
3/22/17
Chris Henson

Introduction:

This project is an implementation of simulated annealing to minimize a given function. It has been designed and implemented in accordance with the stipulations and guidelines of BU CS 767, Machine Learning Project #2, Spring 2017. It is written in the R language, and the seed for the random number generator has been set within the code so that the results are reproducible. It is designed using the standard simulated annealing algorithm. If a better solution is generated with the neighborhood search, then it is taken. Otherwise, the option is still taken with a random distribution function involving the temperature setting. The higher the temperature, the higher the likelihood of taking one of these random options. This prevents only settling at local optima and increasing the chance of finding another, better solution.

Project Notes:

1. The initial temperature and the threshold to end the loop were chosen initially at random, and then by experimenting with multiple runs to determine better outputs.

2. The stopping condition was also arrived at experimentally, with many iterations being conducted to determine the right match that produces the more optimal output.

3. As with the other parameters, this was initially a random number, and was then adjusted to help with optimizing the function's output.

4. As with the other parameters above, the number of iterations was an experimentation with multiple iterations to determine optimality.

5. The initial starting positions were determined randomly, using the seeded random number generator included in the code.

The output to the function given: $\min\{f(x_1, x_2, x_2) = x_1^2 + x_2^2 + x_3^2\}$ has optimality at 0 for all dimensions. While 0 was never reached in any iteration (this implementation would not be able to reach that unless the initial position happened to start at 0 because of how it segments the neighborhood search) the algorithm was tuned to the point where the resultant is 7.3e-07, so it is close to optimal. One thing to note is that because of the combination of large double loops, this program does take up to a couple of minutes to fully execute, depending on the machine on which it is executed. One interesting thing to note was that I also built this program in Python, but couldn't seem to get quite as high output accuracy. I believe this is due to the random seeding differences between R and Python.

The original code is below. This implementation can easily be modified to accommodate more variables by simply adding them at each point where the other variables are created/modified (although this will increase computation time).

Final output:
x_1: **0.0004523863**
x_2: **0.0001159100**
x_3: **0.0007156569**
Function evaluation value: **7.302532e-07**

R-Code:

```
t <- 0
#random number generator seed
set.seed(123)

#produces randomly generated starting points
x <- runif(3, -10, 10)

#initial temperature setting
T0 <- 10000

eval <- function(x, y, z) {
        return (x^2 + y^2 + z^2)
}

x1 <- x[1]
x2 <- x[2]
x3 <- x[3]

curr <- eval(x1, x2, x3)

#temperature iterations
while (t < 2500) {
        count <- 0
   #runs a set number of iterations for each temp setting
        while (count < 2500) {
          xx <- runif(3, -.3, .3)
          x11 <- xx[1] + x1
          x22 <- xx[2] + x2
          x33 <- xx[3] + x3
          test <- eval(x11, x22, x33)
          if (test < curr) {
                    curr <- test
                    x1 <- x11
                    x2 <- x22
                    x3 <- x33
                    }
          else {
                    if ((runif(1, 0, 1)) < exp((curr - test)/T0)) {
                              curr <- test
                              x1 <- x11
                              x2 <- x22
                              x3 <- x33
                              }
                    }
          count <- count + 1
          }
   #reduces the temperature setting
        T0 <- T0/2
        t <- t + 1
        }
#final result
c(x1, x2, x3 )
eval(x1, x2, x3)
```