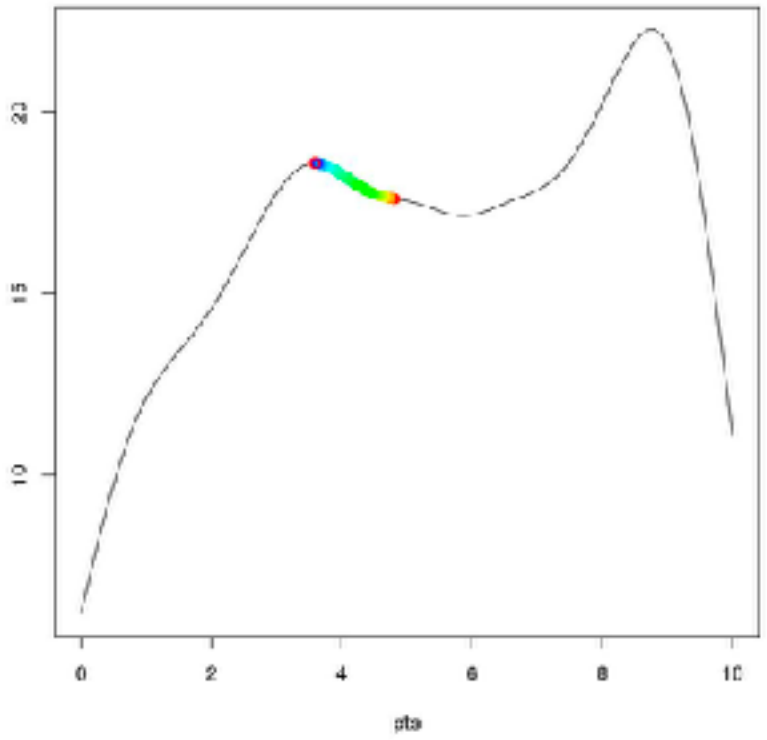
Greedy Local Search and Simulated Annealing
Kevin Peterson

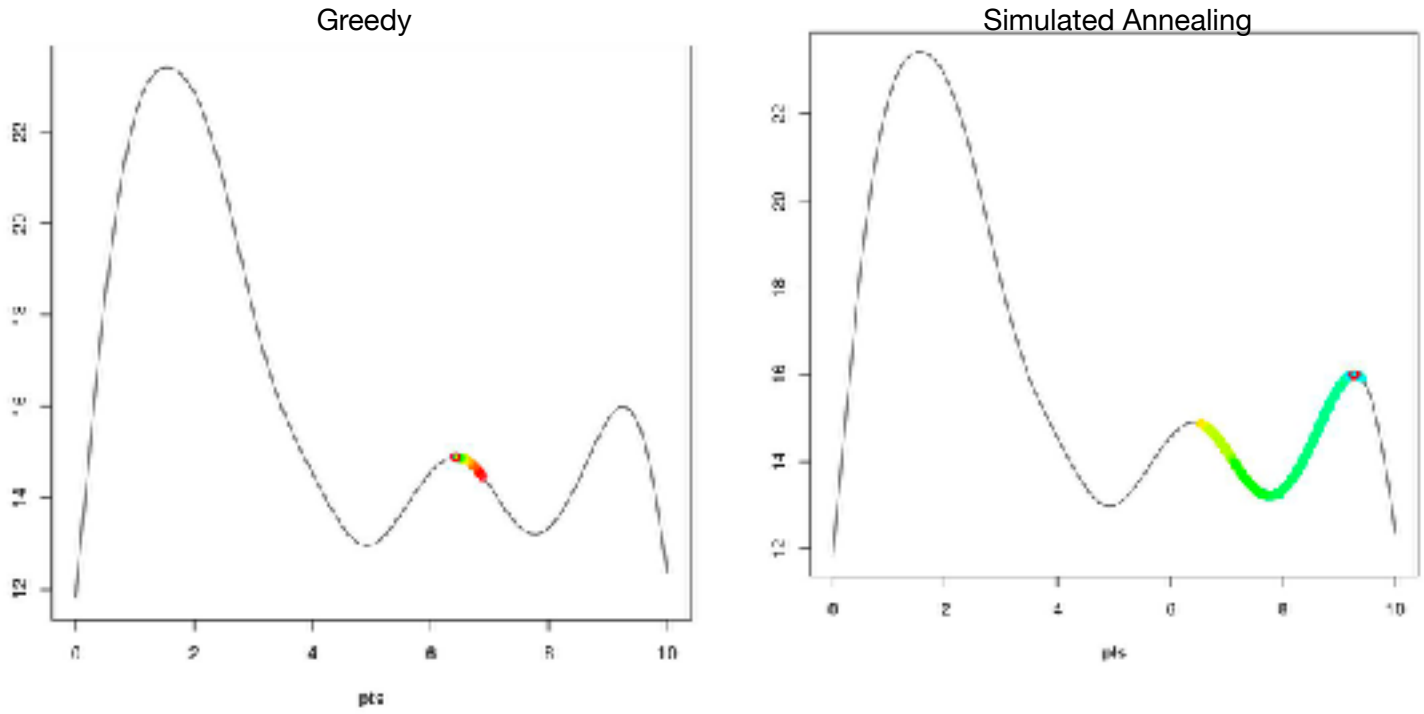
Simulated annealing and Greedy Local search are two local search algorithms that attempt to find the maximum value and location of a function. This code involved finding the maximum of a function given to to the program by the Sum of Gaussians class, which just provides a function that is the sum of a certain number of Gaussian functions.

Greedy Local Search finds the maximum of the function by moving a certain distance in the direction the function is increasing until the function stops increasing. Then the algorithm ends as it has reached the max. I implemented this by moving using a step size. The program moved by a step size of .01 * dG(x)/dx. The program will change the value of x by this step size and then check if the value of G(x) has increased. If the value of G(x) has increased more than 1*10^-8 then the program will just repeat that process again. The process repeats until the value of the function doesn't increase more than 1*10^-8. This means that the function has virtually increased none and it has reached the max. The reason we use such a small step size is so that Greedy Local Search doesn't overstep the max. If the step size is too big, the algorithm could skip the max value. Because Greedy Local Search goes to the first maximum it finds, this can cause some issues. It has a very large potential of getting stuck on a local maximum of the function, if the function has more than one maximum. The greedy algorithm has no real way of getting out of a local maximum other than starting over at a new random location. An example of a run of the greedy algorithm is shown to the right.



Unlike Greedy Local Search, Simulated Annealing allows for "bad moves" and allows the algorithm to go in a negative direction towards the beginning of the algorithm. The goal of these "bad moves" is to get the algorithm free from being stuck on a local max and find the true global max of the function. Simulated annealing works best when a lot of bad moves are made at the beginning, and as the algorithm progresses, less and less bad moves are made. This is normally implemented as "temperature." The temperature starts out high and slowly decreases. In simulated annealing, the algorithm moves the x value by a random amount from -.01 to .01. It then checks if the G(x) value of the new point is higher than the previous point. If it is higher, the

algorithm keeps that point because it is closer to the max. However, if it isn't higher, there is still a possibility of keeping that point. This is where simulated annealing differs from Greedy Local Search. In my code, if the G(Y) value is less than the G(x) value, then there is an e^((G(Y)-G(x)) /temperature) chance that we will keep the point. This equation is useful because when the temperature is high, it is near 1. However when the temperature is low, it is near 0. This means that the temperature cools off and less bad moves are made as the program goes on. Below is an example of a run of my code that shows greedy algorithm finding a local max, but Simulated annealing is able to escape and find a bigger max.



I started my temperature for Simulated Annealing at a value of 100,000 and decreased this value by .01% every iteration/move. This provided for a nice curve that decreased over time and allowed for a high temperature for the first iterations, and a temperature at or near zero towards the end iterations. The temperature was not perfect because simulated Annealing often times did not out perform greedy. However, it was able to escape from local maximums sometimes.

Below are the stats from 100 runs for each combination of Dimensions 1-5 and Number of Gaussians (5,10,50,100,500,1000). The numbers in the chart represent how many times that Greedy Local Search or Simulated Annealing found the higher value for the function. A tie of the algorithms is documented in the chart when both algorithms returned a value within 1e^-8 of each other.

Run Statistics (G = Greedy Win, S = Simulated annealing Win, T = Tie)

	D1	D2	D3	D4	D5
N5	G:75 S:9 T:16	G:67 S:29 T:4	G:46 S:32 T:22	G:46 S:32 T:22	G:13 S:16 T:71
N10	G:69 S:8 T:23	G:72 S:24 T:4	G:55 S:31 T:14	G:51 S:24 T:25	G:24 S:14 T:62
N50	G:51 S:12 T:37	G:83 S:15 T:2	G:79 S:21 T:0	G:61 S:37 T:2	G:46 S:42 T:12
N100	G:37 S:8 T:55	G:84 S:16 T:0	G:72 S:27 T:1	G:74 S:26 T:0	G:60 S:36 T:4
N500	G:35 S:5 T:60	G:61 S:13 T 26	G:75 S:25 T:0	G:64 S:36 T:0	G:74 S:26 T:0
N1000	G:13 S:10 T:77	G:53 S:18 T:29	G:73 S:26 T:1	G:62 S:38 T:0	G:64 S:36 T:0

Overall using this data, the SA algorithm was beaten by Greedy for every combination of dimensions and Gaussians. This points to the fact that the temperature cooling schedule was not as accurate as it should be if one were to ever implement Simulated Annealing. This also means that the "bad moves" made by the annealing schedule caused the SA algorithm to get stuck on an even lower local max than it started on at times. The schedule I decided on probably ended up not taking as much bad moves as it should have in the beginning. It often was not able to find the global maximum if it did not start near it. The wrong moves in the beginning of the algorithm should have spanned a much larger area. One reason this was so difficult to achieve was the small step size. The smallness of the step size made it very hard for the algorithm to move far away from its original point in the beginning of the algorithm

When comparing the overall performance of Greedy Local Search and Simulated Annealing, they both have benefits and downfalls. Greedy Local Search is much quicker, but also has much more of a potential to get stuck on a local maximum, rather than finding the global max. Simulated annealing is slower, but more likely to outperform and find a greater maximum than Greedy Local Search.